The Foundation Framework

Framework: NextLibrary/Frameworks/Foundation.framework

Header File Directories: NextLibrary/Frameworks/Foundation.framework/Headers

Introduction

The Foundation Framework defines a base layer of Objective-C classes for OpenStep. In addition to providing a set of useful primitive object classes, it introduces several paradigms that define functionality not covered by the Objective-C language. The Foundation Framework is designed with these goals in mind:

- Provide a small set of basic utility classes
- Make software development easier by introducing consistent conventions for things such as deallocation
- Support Unicode strings, object persistence, and object distribution
- Provide a level of OS independence, to enhance portability

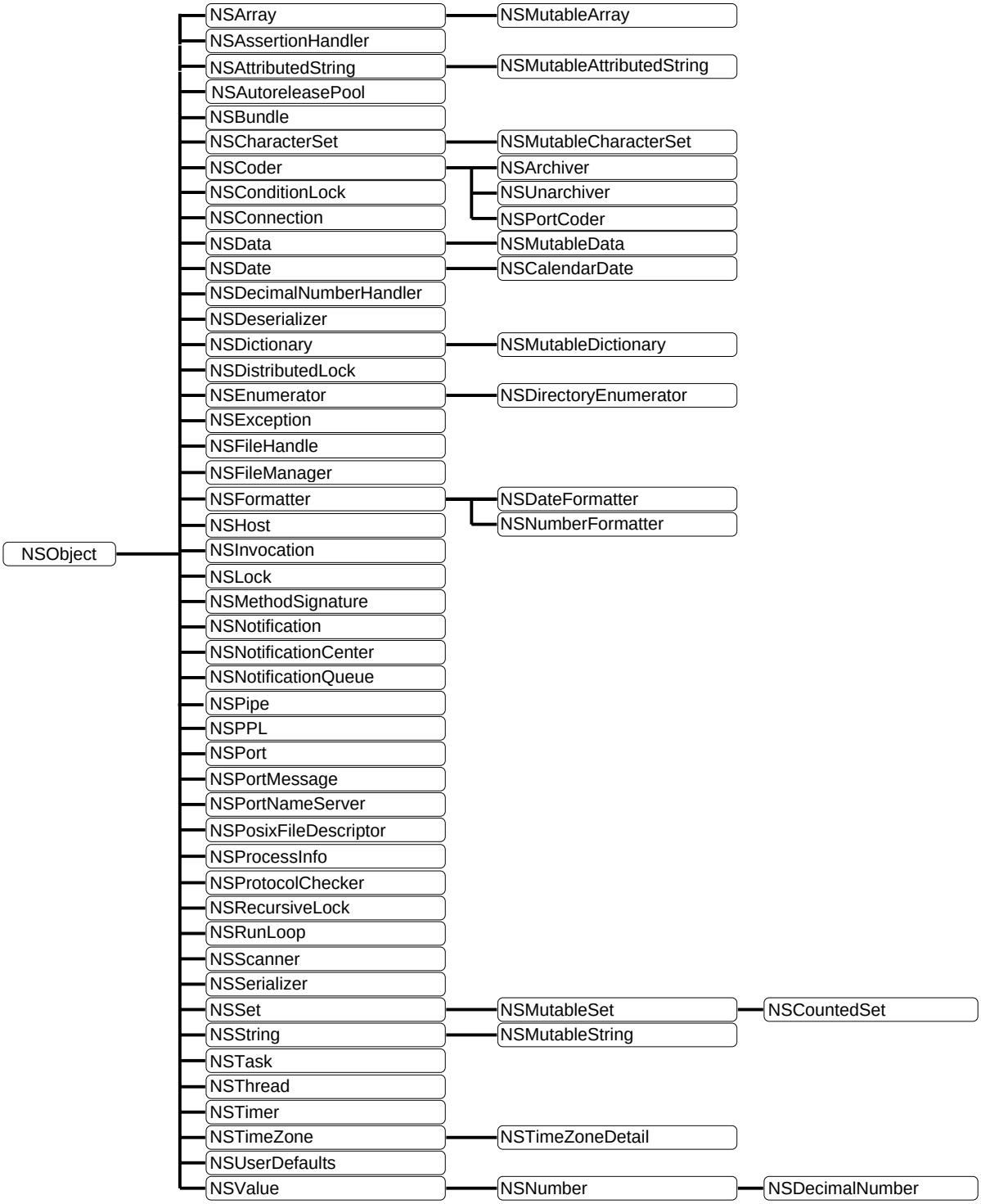
The Foundation Framework includes the root object class, classes representing basic data types such as strings and byte arrays, collection classes for storing other objects, classes representing system information such as dates, and classes representing communication ports. See XREF for a list of those classes that make up the Foundation Framework.

The Foundation Framework introduces several paradigms to avoid confusion in common situations, and to introduce a level of consistency across class hierarchies. This is done with some standard policies, such as that for object ownership (that is, who is responsible for disposing of objects), and with abstract classes like NSEnumerator. These new paradigms reduce the number of special and exceptional cases in API and allow you to code more efficiently by reusing the same mechanisms with various kinds of objects.

Foundation Framework Classes

The OpenStep class hierarchy is rooted in the Foundation Framework's NSObject class (see Figure 1). The remainder of the Foundation Framework consists of several related groups of classes as well as a few individual classes. Many of the groups form what are called class clusters–abstract classes that work as umbrella interfaces to a versatile set of private subclasses. NSString and NSMutableString, for example, act as brokers for instances of various private subclasses optimized for different kinds of storage needs. Depending on the method you use to create a string, an instance of the appropriate optimized class will be returned to you.

Figure 1 The Foundation Framework class hierarchy



Many of these classes have closely related functionality:

- **Data storage**. NSData and NSString provide object-oriented storage for arrays of bytes. NSValue and NSNumber provide object-oriented storage for arrays of simple C data values. NSArray, NSDictionary, NSPPL, and NSSet provide storage for Objective-C objects of any class.
- **Text and strings**. NSCharacterSet represents various groupings of characters which are used by the NSString and NSScanner classes. The NSString classes represent text strings and provide methods for searching, combining, and comparing strings. An NSScanner object is used to scan numbers and words from an NSString object.
- **Dates and times**. The NSDate and NSTimeZone classes store times and dates. They offer methods for calculating date and time differences, for displaying dates and times in many formats, and for adjusting times and dates based on location in the world.
- **Application coordination and timing**. NSNotification, NSNotificationCenter, and NSNotificationQueue provide systems that an object can use to notify all interested observers of changes that occur. You can use a NSTimer object to send a message to another object at specific intervals.
- **Object creation and disposal**. NSAutoreleasePools are used to implement the delayed-release feature of the Foundation Framework.
- **Object distribution and persistence**. The data that an object contains can be represented in an architecture-independent way using NSSerializer. The NSCoder and its subclasses take this process a step further by allowing class information to be stored along with the data. The resulting representations are used for archiving and for object distribution.
- **Operating system services**. Several classes are designed to insulate you from the idiosynccracies of various operating systems. NSFileManager provides a consistent interface for file operations (creating, renaming, deleting, and so on). NSThread and NSProcessInfo let you create multi-threaded applications and query the environment in which an application runs.

NSArchiver

Inherits From: NSCoder : NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSArchiver.h

Class at a Glance

Purpose

An NSArchiver encodes objects into a format that can be written to a file. The archiving process traverses a set of interconnected objects, making sure to encode each one only once.

Principal Attributes

- An NSMutableData object containing the encoded data

Creation

– initForWritingWithMutableData:

Commonly Used Methods

+ archiveRootObject:ToFile:	Archives a graph of objects to a file.
+ archivedDataWithRootObject:	Archives a graph of objects into an NSMutableData object.

Class Description

NSArchiver, a concrete subclass of NSCoder, provides a way to encode Objective-C objects into an architecture-independent format that can be stored in a file. When you archive a set of objects, the class information and instance variables for each object are written to the archive. NSArchiver's companion class, NSUnarchiver, decodes the data in an archive and creates a set of objects equivalent to the original set.

NSArchiver stores the archive data in a mutable-data object (NSMutableData). After encoding the objects, you can have the NSArchiver object write this mutable-data object immediately to a file, or you can retrieve the mutable-data object for some other use.

Archiving a Graph of Objects

The easiest way to archive an object is to invoke a single class method—either **archiveRootObject:toFile:** or **archivedDataWithRootObject:**, depending on whether you want the encoded data to be stored in a file immediately. These convenience methods create a temporary NSArchiver and send it an **encodeRootObject:** message—you need do no more. However, if you want to customize the archiving process (for example, by substituting certain classes for others), you must instead create an instance of NSArchiver yourself, configure it as desired, and send it an **encodeRootObject:** message explicitly.

The "root object" that you specify as the argument to any of these three methods indicates the starting point for archiving. The NSArchiver commences archiving by invoking the root object's **encodeWithCoder:** method. That method typically encodes the root object's instance variables, which isn't necessarily a straightforward process—the instance variables can themselves be other objects that respond to **encodeWithCoder:**, and so on, yielding a possibly complex graph of objects that need to be archived.

The fact that many objects contain references to other objects poses two problems for archiving. The first is redundancy. An object graph isn't necessarily a simple tree structure. Two objects can contain references to each other, for example, creating a cycle. To address this problem, NSArchiver overrides NSCoder's **encodeRootObject:** method to keep track of all the objects encountered while traversing the graph. The first time an object is encountered, the object is encoded normally. On subsequent occurances of the same object, a reference to the original object is encoded instead of the object itself.

The second problem is that it's not always appropriate to archive the entire graph. To use an example from the Application Kit, when you archive an NSView as the root object, its subviews should be archived, but not its superview. In this case, the superview is considered an extraneous part of the graph. To solve this dilemma, NSArchiver implements conditional archiving, overriding the minimal **encodeConditionalObject:** method that's inherited from NSCoder. A class's **encodeWithCoder:** method can invoke **encodeConditionalObject:** to archive inessential object instance variables. The NSArchiver doesn't actually archive a conditionally encoded object unless some other object in the graph encodes it unconditionally (using one of the other **encode...Object:** methods declared by NSCoder). When everything is unarchived, all original references to the conditionally encoded object are properly restored as references to the single unarchived object. For example, an NSView encodes its superview with **encodeConditionalObject:**, because it doesn't own the superview but does need to preserve its connection to it if some other object archives the superview.

In contrast, **encodeObject:** unconditionally instructs an object to encode itself. This method is most often used in a class's **encodeWithCoder:** method for instance variables that are intrinsic to the receiver and essential for proper functioning.

All the objects to be placed in a single archive must be interconnected members of a single graph. In other words, there can only be one root object per archive. The only recommended way to archive objects is to

send an NSArchiver a single **encodeRootObject:** message, whether directly, or indirectly by invoking **archiveRootObject:toFile:** or **archivedDataWithRootObject:**. Don't try to add data to the archive by invoking any of NSCoder's other **encode...** methods, except from within the **encodeWithCoder:** method of each object that's part of the graph. (These **encodeWithCoder:** methods are invoked automatically when you encode the root object.)

To extract an object graph from an archive, use the NSUnarchiver class method **unarchiveObjectWithFile:** or **unarchiveObjectWithData:**, assigning the return value to the desired root object.

Archiving other Data Types

It's possible to create an archive that doesn't contain any objects. To archive other data types, invoke **encodeValueOfObjCType:** directly for each data item to be archived, instead of using **encodeRootObject:** . When you create an archive in this way, the corresponding unarchiving code must follow exactly the same sequence of data types.

This approach shouldn't be used to archive objects. Use **encodeRootObject:** instead, to avoid the problems mentioned in the previous section and to simplify unarchiving.

An NSSerializer provides another means to store data in an architecture-independent format. See the NSSerializer class specification for more information.

Superclass Methods to Avoid

NSArchiver's superclass, NSCoder, supplies methods for both encoding and decoding. However, only the encoding methods are applicable to NSArchiver—don't send an NSArchiver any **decode...** messages. (Similarly, don't send **encode...** messages to an NSUnarchiver.)

Method Types

Initializing an NSArchiver

– initForWritingWithMutableData:

Archiving data

+ archivedDataWithRootObject:
+ archiveRootObject:toFile:
– encodeRootObject:
– encodeConditionalObject:

Getting the archived data

– archiverData

Substituting classes or objects

– classNameEncodedForTrueClassName:
– encodeClassName:intoClassName:
– replaceObject:withObject:

Class Methods

archiveRootObject:toFile:

+ (BOOL)**archiveRootObject:**(id)*rootObject* **toFile:**(NSString *)*path*

Creates a temporary instance of NSArchiver and archives *rootObject* by encoding it into a data object. Once the object has been encoded, this method writes the resulting data object to the file *path*. This convenience method invokes **archivedDataWithRootObject:** to get the encoded data, and then sends that data object the message **writeToFile:atomically:**, using *path* for the first argument and YES for the second. Returns YES upon success.

See also: **+ archivedDataWithRootObject:**, **– writeToFile:atomically:** (NSData)

archivedDataWithRootObject:

+ (NSData *)**archivedDataWithRootObject:**(id)*rootObject*

Returns a data object containing the encoded form of the object graph whose root object is *rootObject*. This method invokes **initForWritingWithMutableData:** and **encodeRootObject:** to create a temporary archiver that encodes the object graph.

See also: **– initForWritingWithMutableData:**, **– encodeRootObject**:

Instance Methods

archiverData

– (NSMutableData *)**archiverData**

Returns the archived data. The returned data object is the same one that was specified as the argument to **initForWritingWithMutableData:**. It contains whatever data has been encoded thus far by invocations of the various encoding methods. It's safest not to invoke this method until after **encodeRootObject:** has returned. In other words, although it's possible for a class to invoke this method from within its **encodeWithCoder:** method, that method must not alter the data.

classNameEncodedForTrueClassName:

– (NSString *)**classNameEncodedForTrueClassName:**(NSString *)*trueName*

Returns the class name used to archive instances of the class *trueName*.

See also: – **encodeClassName:intoClassName:**

encodeClassName:intoClassName:

– (void)**encodeClassName:**(NSString *)*trueName* **intoClassName:**(NSString *)*inArchiveName*

Encodes a substitute name for the class named *trueName*. Any subsequently encountered objects of class *trueName* will be archived as instances of class *inArchiveName*. It's safest not to invoke this method during the archiving process (that is, within an **encodeWithCoder:** method). Instead, invoke it before **encodeRootObject:**.

See also: – **classNameEncodedForTrueClassName:**

encodeConditionalObject:

– (void)**encodeConditionalObject:**(id)*object*

Archives *object* conditionally. This method overrides the superclass implementation to allow *object* to be encoded only if it's also encoded unconditionally by another object in the object graph. Conditional encoding lets you encode one part of a graph detached from the rest. (See the class description for more information.)

This method should be invoked only from within an **encodeWithCoder:** method. If *object* is **nil**, the NSArchiver encodes it unconditionally as **nil**. This method raises an NSInvalidArgumentException if no root object has been encoded.

encodeRootObject:

– (void)**encodeRootObject:**(id)*rootObject*

Archives *rootObject* along with all the objects it's connected to. If any object is encountered more than once while traversing the graph, it's encoded only once, but the multiple references to it are stored. (See the discussion of object graphs in the class description.)

This message must not be sent more than once to a given NSArchiver; an NSInvalidArgumentException is raised if a root object has already been encoded. Therefore, don't attempt to reuse an NSArchiver; instead, create a new one. To encode multiple object graphs, use distinct NSArchivers.

initForWritingWithMutableData:

– (id)**initForWritingWithMutableData:**(NSMutableData *)*data*

Initializes an archiver, encoding stream and version information into *data*. Raises an NSInvalidArgumentException if *data* is **nil**.

See also: – **archiverData**

replaceObject:withObject:

– (void)**replaceObject:**(id)*object* **withObject:**(id)*newObject*

Causes the NSArchiver to treat subsequent requests to encode *object* as though they were requests to encode *newObject*. Both *object* and *newObject* must be valid objects.

NSArray Class Cluster

Class Cluster Description

The NSArray class clusters manage arrays of objects. The cluster's two public classes, NSArray and NSMutableArray, declare the programmatic interface for static and dynamic arrays, respectively.

The objects you create using these classes are referred to as *arrays*. Because of the nature of class clusters, arrays are not actual instances of the NSArray or NSMutableArray classes but of one of their private subclasses. Although an array's class is private, its interface is public, as declared by these abstract superclasses, NSArray and NSMutableArray.

Generally, you instantiate an array by sending one of the **array...** messages to either the NSArray or NSMutableArray class object. These methods return an array containing the elements you pass in as arguments. (Note that arrays can't contain **nil**.) In general, objects that you add to an array aren't copied; rather, each object receives a **retain** message before its **id** is added to the array. When an object is removed from an array, it's sent a **release** message.

The NSArray class adopts the NSCopying and NSMutableCopying protocols, making it convenient to convert an array of one type to the other.

NSArray

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSMutableCopying NSObject (NSObject)
Declared In:	Foundation/NSArray.h

Class at a Glance

Purpose

An NSArray stores an immutable array of objects.

Principal Attributes

- A count of the number of objects in the array.
- The set of objects contained in the array.

Creation

+ array	Returns an empty array.
+ arrayWithArray:	Returns an array containing the elements from another array.
+ arrayWithContentsOfFile:	Returns an array initialized from the contents of a file.
+ arrayWithObject:	Returns an array containing a single object.
+ arrayWithObjects:	Returns an array containing multiple objects.
+ arrayWithObjects:count:	Returns an array containing a specified number of objects.

Commonly Used Methods

– count:	Returns the number of objects currently in the array.
– objectAtIndex:	Returns the object located at the specified index.

Primitive Methods

– count

– objectAtIndex:

Class Description

NSArray declares the programmatic interface to an object that manages an unchanging array of objects. NSArray's two primitive methods—**count** and **objectAtIndex:**—provide the basis for all other methods in

its interface. The **count** method returns the number of elements in the array. **objectAtIndex:** gives you access to the array elements by index, with index values starting at 0.

The methods **objectEnumerator** and **reverseObjectEnumerator** also grant sequential access to the elements of the array, differing only in the direction of travel through the elements. These methods are provided so that arrays can be traversed in a manner similar to that used for objects of other collection classes such as NSDictionary. See the **objectEnumerator** method description for a code excerpt that shows how to use these methods to access the elements of an array.

NSArray provides methods for querying the elements of the array. **indexOfObject:** searches the array for the object that matches its argument. To determine whether the search is successful, each element of the array is sent an **isEqual:** message, as declared in the NSObject protocol. Another method, **indexOfObjectIdenticalTo:**, is provided for the less common case of determining whether a specific object is present in the array. **indexOfObjectIdenticalTo:** tests each element in the array to see whether its **id** matches that of the argument.

NSArray's **makeObjectsPerformSelector:** and **makeObjectsPerformSelector:withObject:** methods let you send messages to all objects in the array. To act on the array as a whole, a variety of other methods are defined. You can create a sorted version of the array (**sortedArrayUsingSelector:** and **sortedArrayUsingFunction:context:**), extract a subset of the array (**subarrayWithRange:**), or concatenate the elements of an array of NSStrings into a single string (**componentsJoinedByString:**). In addition, you can compare two arrays using the **isEqualToArray:** and **firstObjectCommonWithArray:** methods. Finally, you can create new arrays that contain the objects in an existing array and one or more additional objects with **arrayByAddingObject:** and **arrayByAddingObjectsFromArray:**.

Adopted Protocols

NSCoding	
	– encodeWithCoder:
	– initWithCoder:
NSCopying	
	– copyWithZone:
NSMutableCopying	
	– mutableCopyWithZone:

Method Types

Creating an array

+ allocWithZone:
+ array
+ arrayWithArray:
+ arrayWithContentsOfFile:
+ arrayWithObject:
+ arrayWithObjects:
+ arrayWithObjects:count:
– initWithArray:
– initWithContentsOfFile:
– initWithObjects:
– initWithObjects:count:

Querying the array

– containsObject:
– count
– getObjects:
– getObjects:range:
– indexOfObject:
– indexOfObject:inRange:
– indexOfObjectIdenticalTo:
– indexOfObjectIdenticalTo:inRange:
– lastObject
– objectAtIndex:
– objectEnumerator
– reverseObjectEnumerator

Sending messages to elements

– makeObjectsPerformSelector:
– makeObjectsPerformSelector:withObject:

Comparing arrays

– firstObjectCommonWithArray:
– isEqualToArray:

Deriving new arrays

– arrayByAddingObject:
– arrayByAddingObjectsFromArray:
– sortedArrayHint
– sortedArrayUsingFunction:context:
– sortedArrayUsingFunction:context:hint:
– sortedArrayUsingSelector:
– subarrayWithRange:

Working with string elements

– componentsJoinedByString:
– pathsMatchingExtensions:

Creating a description of the array

– description
– descriptionWithLocale:
– descriptionWithLocale:indent:
– writeToFile:atomically:

Class Methods

allocWithZone:

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

Creates and returns an uninitialized array in the specified zone. If the receiver is the NSArray class object, an instance of an immutable private subclass is returned; otherwise, an object of the receiver's class is returned.

Typically, you create temporary arrays using the **array...** class methods, not the **allocWithZone:** and **init...** methods. Note that it's your responsibility to free objects created with the **allocWithZone:** method.

array

+ (id)**array**

Creates and returns an empty array. This method is used by mutable subclasses of NSArray.

See also: + **arrayWithObject:**, + **arrayWithObjects:**

arrayWithArray:

+ (id)**arrayWithArray:**(NSArray *)*anArray*

Creates and returns an array containing the objects in *anArray.*

See also: + **arrayWithObjects:**, **– initWithObjects:**

arrayWithContentsOfFile:

+ (id)**arrayWithContentsOfFile:**(NSString *)*aPath*

Creates and returns an array containing the contents of the file specified by *aPath*. The file identified by *aPath* must contain a string representation produced by the **writeToFile:atomically:** method. In addition, the array representation must contain only property list objects (NSString, NSData, NSArray, or NSDictionary objects).

Returns **nil** if the file can't be opened or if the contents of the file can't be parsed into an array.

See also: **– writeToFile:atomically:**

arrayWithObject:

+ (id)**arrayWithObject:**(id)*anObject*

Creates and returns an array containing the single element *anObject*.

See also: + **array**, + **arrayWithObjects:**

arrayWithObjects:

+ (id)**arrayWithObjects:**(id)*firstObj*, ...

Creates and returns an array containing the objects in the argument list. The argument list is a comma-separated list of objects ending with **nil**.

This code example creates an array containing three different types of elements (assuming *aPath* exists):

```
NSArray *myArray;
NSData *someData = [NSData dataWithContentsOfFile:aPath];
NSValue *aValue = [NSNumber numberWithInt:5];
NSString *aString = @"a string";

myArray = [NSArray arrayWithObjects:someData, aValue, aString, nil];
```

See also: + **array**, + **arrayWithObject:**

arrayWithObjects:count:

+ (id)**arrayWithObjects:**(id *)*objects* **count:**(unsigned)*count*

Creates and returns an array containing *count* objects from *objects*.

See also: – **getObjects:**, – **getObjects:range:**

Instance Methods

arrayByAddingObject:

– (NSArray *)**arrayByAddingObject:**(id)*anObject*

Returns a new array that is a copy of the receiver with *anObject* added to the end. Since *anObject* is added to the array, it receives a **retain** message. If *anObject* is **nil**, an NSInvalidArgumentException is raised.

See also: – **addObject:** (NSMutableArray)

arrayByAddingObjectsFromArray:

– (NSArray *)**arrayByAddingObjectsFromArray:**(NSArray *)*otherArray*

Returns a new array that is a copy of the receiver with the objects contained in *otherArray* added to the end.

See also: – **addObjectsFromArray:** (NSMutableArray)

componentsJoinedByString:

– (NSString *)**componentsJoinedByString:**(NSString *)*separator*

Constructs and returns an NSString that is the result of interposing *separator* between the elements of the receiver's array. For example, this code excerpt writes the path **/System/Developer** to the console:

```
NSArray *pathArray = [NSArray arrayWithObjects:@"System", @"Developer", nil];
NSLog("The path is /%@.\n", [pathArray componentsJoinedByString:@"/"]);
```

Each element of the receiver's array must be an NSString or an error occurs. If the receiver has no elements, an NSString representing an empty string is returned.

See also: – **componentsSeparatedByString:** (NSString)

containsObject:

– (BOOL)**containsObject:**(id)*anObject*

Returns YES if *anObject* is present in the array. This method determines whether an object is present in the array by sending an **isEqual:** message to each of the array's objects (and passing *anObject* as the parameter to each **isEqual:** message).

See also: – **indexOfObject:**, – **indexOfObjectIdenticalTo:**, – **isEqual:** (NSObject)

count

– (unsigned int)**count**

Returns the number of objects currently in the array.

See also: – **objectAtIndex:**

description

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

Returns a string that represents the contents of the receiver, formatted as a property list.

See also: **– descriptionWithLocale:, – descriptionWithLocale:indent:**

descriptionWithLocale:

– (NSString *)**descriptionWithLocale:**(NSDictionary *)*locale*

Returns a string that represents the contents of the receiver, formatted as a property list. *locale* specifies options used for formatting each of the receiver's elements (where recognized); specify **nil** if you don't want the elements formatted.

For a description of how *locale* is applied to each element in the receiving array, see **descriptionWithLocale:indent:**.

See also: **– description, – descriptionWithLocale:indent:**

descriptionWithLocale:indent:

– (NSString *)**descriptionWithLocale:**(NSDictionary *)*locale* **indent:**(unsigned int)*level*

Returns a string that represents the contents of the receiver, formatted as a property list. *locale* specifies options used for formatting each of the receiver's elements; specify **nil** if you don't want the elements formatted. *level* allows you to specify a level of indent, to make the output more readable: set *level* to 0 to use four spaces to indent, or 1 to indent the output with a tab character.

The returned NSString contains the string representations of each of the receiver's elements, in order, from first to last. To obtain the string representation of a given element, **descriptionWithLocale:indent:** proceeds as follows:

- If the element is an NSString, it is used as is.
- If the element responds to **descriptionWithLocale:indent:**, that method is invoked to obtain the element's string representation.

- If the element responds to **descriptionWithLocale:**, that method is invoked to obtain the element's string representation.
- If none of the above conditions are met, the element's string representation is obtained by invoking its **description** method.

See also: **– description, – descriptionWithLocale:**

firstObjectCommonWithArray:

– (id)**firstObjectCommonWithArray:**(NSArray *)*otherArray*

Returns the first object contained in the receiver that's equal to an object in *otherArray*. If no such object is found, this method returns **nil**. This method uses **isEqual:** to check for object equality.

See also: – **containsObject:**, – **isEqual:** (NSObject)

getObjects:

– (void)**getObjects:**(id *)*aBuffer*

Copies the objects contained in the receiver to *aBuffer*.

See also: + **arrayWithObjects:count:**

getObjects:range:

– (void)**getObjects:**(id *)*aBuffer* **range:**(NSRange)*aRange*

Copies the objects contained in the receiver that fall within the specified range to *aBuffer*.

See also: + **arrayWithObjects:count:**

hash

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

Returns an unsigned integer that can be used as a table address in a hash table structure. For an array, **hash** returns the number of elements in the array. If two arrays are equal (as determined by the **isEqual:** method), they will have the same hash value.

See also: – **isEqual:** (NSObject)

indexOfObject:

– (unsigned int)**indexOfObject:**(id)*anObject*

Searches the receiver for *anObject* and returns the lowest index whose corresponding array value is equal to *anObject*. Objects are considered equal if they have the same **id** or if **isEqual:** returns YES. If none of the objects in the receiver are equal to *anObject*, **indexOfObject:** returns NSNotFound.

See also: – **containsObject:**, – **indexOfObjectIdenticalTo:**, – **isEqual:** (NSObject)

indexOfObject:inRange:

– (unsigned)**indexOfObject:**(id)*anObject* **inRange:**(NSRange)*aRange*

Searches the specified range within the receiver for *anObject* and returns the lowest index whose corresponding array value is equal to *anObject*. Objects are considered equal if they have the same **id** or if **isEqual:** returns YES. If none of the objects in the specified range are equal to *anObject*, **indexOfObject:** returns NSNotFound.

See also: – **containsObject:**, – **indexOfObjectIdenticalTo:**, – **isEqual:** (NSObject)

indexOfObjectIdenticalTo:

– (unsigned int)**indexOfObjectIdenticalTo:**(id)*anObject*

Searches the receiver for *anObject* (testing for equality by comparing object **id**s) and returns the lowest index whose corresponding array value is equal to *anObject*. If none of the objects in the receiver are equal to *anObject*, **indexOfObject:** returns NSNotFound.

See also: – **containsObject:**, – **indexOfObject:**, – **isEqual:** (NSObject)

indexOfObjectIdenticalTo:inRange:

– (unsigned)**indexOfObjectIdenticalTo:**(id)*anObject* **inRange:**(NSRange)*aRange*

Searches the specified range within the receiver for *anObject* (testing for equality by comparing object **id**s) and returns the lowest index whose corresponding array value is equal to *anObject*. If none of the objects in the specified range are equal to *anObject*, **indexOfObject:** returns NSNotFound.

See also: – **containsObject:**, – **indexOfObject:**, – **isEqual:** (NSObject)

initWithArray:

– (id)**initWithArray:**(NSArray *)*anArray*

Initializes a newly allocated array by placing in it the objects contained in *array*. Each object in *array* receives a **retain** message as it's added to the array. After an immutable array has been initialized in this way, it can't be modified. Returns **self**.

See also: **+ arrayWithObject:**, – **initWithObjects:**

initWithContentsOfFile:

– (id)**initWithContentsOfFile:**(NSString *)*aPath*

Initializes a newly allocated array with the contents of the file specified by *aPath*. The file identified by *aPath* must contain a string representation produced by the **writeToFile:atomically:** method. In addition, the array representation must contain only property list objects (NSString, NSData, NSArray, or NSDictionary objects).

Returns **self** if the receiver is successfully initialized, or **nil** if the file can't be opened or if the contents of the file can't be parsed into an array.

See also: – **writeToFile:atomically:**

initWithObjects:

– (id)**initWithObjects:**(id)*firstObj*, ...

Initializes a newly allocated array by placing in it the objects in the argument list. This list is a comma-separated list of objects ending with **nil**. Objects are retained as they're added to the array. After an immutable array has been initialized in this way, it can't be modified. Returns **self**.

See also: – **initWithObjects:count:**, **+ arrayWithObjects:**, – **initWithArray:**

initWithObjects:count:

– (id)**initWithObjects:**(id *)*objects* **count:**(unsigned int)*count*

Initializes a newly allocated array by placing in it *count* objects from the *objects* array. Each object in the *objects* array receives a **retain** message as it's added to the array. After an immutable array has been initialized in this way, it can't be modified. Returns **self**.

See also: – **initWithObjects:**, **+ arrayWithObjects:**, – **initWithArray:**

isEqual:

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

Returns YES if the receiver and *anObject* are equal; otherwise returns NO. A YES return value indicates that the receiver and *anObject* are both instances of classes that inherit from NSArray and that they both contain the same objects (as determined by the **isEqualToArray:** method).

See also: – **isEqualToArray:**

isEqualToArray:

– (BOOL)**isEqualToArray:**(NSArray *)*otherArray*

Compares the receiving array to *otherArray.* If the contents of *otherArray* are equal to the contents of the receiver, this method returns YES. If not, it returns NO.

Two arrays have equal contents if they each hold the same number of objects and objects at a given index in each array satisfy the **isEqual:** test.

See also: – **isEqual:** (NSObject)

lastObject

– (id)**lastObject**

Returns the object in the array with the highest index value. If the array is empty, **lastObject** returns **nil**.

See also: – **removeLastObject** (NSMutableArray)

makeObjectsPerformSelector:

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

Sends the *aSelector* message to each object in the array, starting with the first object and continuing through the array to the last object. The *aSelector* method must be one that takes no arguments. It shouldn't have the side effect of modifying the receiving array. The messages are sent using the **makeObjectsPerformSelector:withObject:** method.

See also: **– makeObjectsPerformSelector:withObject:**

makeObjectsPerformSelector:withObject:

– (void)**makeObjectsPerformSelector:**(SEL)*aSelector* **withObject:**(id)*anObject*

Sends the *aSelector* message to each object in the array, starting with the first object and continuing through the array to the last object. The message is sent each time with *anObject* as an argument, so the *aSelector* method must be one that takes a single argument of type **id**. The *aSelector* method shouldn't, as a side effect, modify the receiving array. The messages are sent using the **performSelector:withObject:** method declared in the NSObject protocol.

See also: **– makeObjectsPerformSelector:**

objectAtIndex:

– (id)**objectAtIndex:**(unsigned int)*index*

Returns the object located at *index*. If *index* is beyond the end of the array (that is, if *index* is greater than or equal to the value returned by **count**), an NSRangeException is raised.

See also: – **count**

objectEnumerator

– (NSEnumerator *)**objectEnumerator**

Returns an enumerator object that lets you access each object in the array, in order, starting with the element at index 0, as in:

```
NSEnumerator *enumerator = [myArray objectEnumerator];
id anObject;

while ((anObject = [enumerator nextObject])) {
    /*  code to act on each element as it is returned */
}
```

When this method is used with mutable subclasses of NSArray, your code shouldn't modify the array during enumeration.

See also: – **reverseObjectEnumerator,** – **nextObject** (NSEnumerator)

pathsMatchingExtensions:

– (NSArray *)**pathsMatchingExtensions:**(NSArray *)*filterTypes*

Returns a new array that contains those string objects in the receiver that have a filename extension (as determined by NSString's **pathExtension** method) that matches one of the extensions in *filterTypes*. *filterTypes* should be an array of NSStrings, each of which identifies a filename extension to be matched

(such as “tiff” or “eps”). Filenames that don’t have an extension aren’t included in the result. This method can be used to identify those files with a particular extension (or set of extensions) within a directory.

reverseObjectEnumerator

– (NSEnumerator *)**reverseObjectEnumerator**

Returns an enumerator object that lets you access each object in the array, in order, from the element at the highest index down to the element at index 0. Your code shouldn’t modify the array during enumeration.

See also: – **objectEnumerator,** – **nextObject** (NSEnumerator)

sortedArrayHint

– (NSData *)**sortedArrayHint**

Analyzes the receiver and returns a “hint” that speeds the sorting of the array when the hint is supplied to **sortedArrayUsingFunction:context:hint:**.

sortedArrayUsingFunction:context:

– (NSArray *)**sortedArrayUsingFunction:**(int(*)(id, id, void *))*comparator* **context:**(void *)*context*

Returns a new array that lists the receiver’s elements in ascending order as defined by the comparison function *comparator*. The new array contains references to the receiver’s elements, not copies of them. The retain count is incremented for each element in the receiving array.

The comparison function is used to compare two elements at a time and should return NSOrderedAscending if the first element is smaller than the second, NSOrderedDescending if the first element is larger than the second, and NSOrderedSame if the elements are equal. Each time the comparison function is called, it’s passed *context* as its third argument. This allows the comparison to be based on some outside parameter, such as whether character sorting is case-sensitive or case-insensitive.

Given *anArray* (an array of NSNumber objects) and a comparison function of this type:

```
int intSort(id num1, id num2, void *context)
{
    int v1 = [num1 intValue];
    int v2 = [num2 intValue];
    if (v1 < v2)
        return NSOrderedAscending;
    else if (v1 > v2)
        return NSOrderedDescending;
    else
        return NSOrderedSame;
}
```

A sorted version of *anArray* is created in this way:

```
NSArray *sortedArray;
sortedArray = [anArray sortedArrayUsingFunction:intSort context:NULL];
```

See also: – **sortedArrayUsingSelector:**

sortedArrayUsingFunction:context:hint:

– (NSArray *)**sortedArrayUsingFunction:**(int (*)(id, id, void *))*compare* **context:**(void *)*context* **hint:**(NSData *)*hint*

Similar to **sortedArrayUsingFunction:context:**, except that it uses the supplied *hint* to speed the sorting process. To obtain an appropriate *hint*, use **sortedArrayHint**. When you know that the array is nearly sorted, this method is faster than **sortedArrayUsingFunction:context:**.

sortedArrayUsingSelector:

– (NSArray *)**sortedArrayUsingSelector:**(SEL)*comparator*

Returns an array that lists the receiver's elements in ascending order, as determined by the comparison method specified by the selector *comparator*. The new array contains references to the receiver's elements, not copies of them. The retain count is incremented for each element in the receiving array.

The *comparator* message is sent to each object in the array, and has as its single argument another object in the array. The comparator method is used to compare two elements at a time and should return NSOrderedAscending if the receiver is smaller than the argument, NSOrderedDescending if the receiver is larger than the argument, and NSOrderedSame if they are equal.

For example, an array of NSStrings can be sorted by using the **compare:** method declared in the NSString class. Assuming *anArray* exists, a sorted version of the array can be created in this way:

```
NSArray *sortedArray = [anArray sortedArrayUsingSelector:@selector(compare:)];
```

See also: – **sortedArrayUsingFunction:context:**

subarrayWithRange:

– (NSArray *)**subarrayWithRange:**(NSRange)*range*

Returns a new array containing the receiver's elements that fall within the limits specified by *range*. If *range* isn't within the receiver's range of elements, an NSRangeException is raised. Each object receives a **retain** message as it's added to the array.

For example, the following code example creates an array containing the elements found in the first half of *wholeArray* (assuming that *wholeArray* exists).

```
NSArray *halfArray;
NSRange theRange;

theRange.location = 0;
theRange.length = [wholeArray count] / 2;

halfArray = [wholeArray subarrayWithRange:theRange];
```

writeToFile:atomically:

– (BOOL)**writeToFile:**(NSString *)*path* **atomically:**(BOOL)*flag*

Writes the contents of the receiver to the file specified by *path*. If the receiver's contents are all property list objects (NSString, NSData, NSArray, or NSDictionary objects), the file written by this method can be used to initialize a new array with the class method **arrayWithContentsOfFile:** or the instance method **initWithContentsOfFile:**.

If *flag* is YES, the array is written to an auxiliary file, and then the auxiliary file is renamed to *path*. If *flag* is NO, the array is written directly to *path*. The YES option guarantees that *path*, if it exists at all, won't be corrupted even if the system should crash during writing.

If *path* contains a tilde (~) character, you must expand it with **stringByExpandingTildeInPath:** before invoking this method.

This method returns YES if the file is written successfully, and NO otherwise.

See also: – **initWithContentsOfFile:**

NSMutableArray

Inherits From:	NSArray : NSObject
Conforms To:	NSCoding NSCopying NSMutableCopying (NSArray) NSObject (NSObject)
Declared In:	Foundation/NSArray.h

Class at a Glance

Purpose

An NSMutableArray stores a modifiable array of objects.

Principal Attributes

- A count of the number of objects in the array.
- The set of objects contained in the array.

Creation

+ arrayWithCapacity:	An empty array with enough allocated memory to hold a specified number of objects

Commonly Used Methods

– insertObject:atIndex:	Inserts an object at a specified index.
– removeObject:	Removes all occurrences of an object.
– removeObjectAtIndex:	Removes the object at a given index.
– replaceObjectAtIndex:withObject:	Replaces the object at a given index.

Primitive Methods

– addObject:
– replaceObjectAtIndex:withObject:
– removeLastObject

Class Description

The NSMutableArray class declares the programmatic interface to objects that manage a modifiable array of objects. This class adds insertion and deletion operations to the basic array-handling behavior inherited from NSArray.

NSMutableArray methods are conceptually based on these three primitive methods:

- addObject:

- replaceObjectAtIndex:withObject:
- removeLastObject

The other methods in its interface provide convenient ways of inserting an object into a specific slot in the array and removing an object based on its identity or position in the array.

When an object is removed from a mutable array, it receives a **release** message. If there are no further references to the object, the object is deallocated. Note that if your program keeps a reference to such an object, the reference will become invalid unless you remember to send the object a **retain** message before it's removed from the array. For example, if anObject isn't retained before removing it from the array, the third statement below could result in a run-time error:

```
id anObject = [[anArray objectAtIndex:0] retain];
[anArray removeObjectAtIndex:0];
[anObject someMessage];
```

A Note for Those Creating Subclasses of NSMutableArray

Although conceptually the NSMutableArray class has three primitive methods, two others also access the array's data directly. These methods are:

- **insertObject:atIndex:**
- **removeObjectAtIndex:**

These methods could be implemented using the primitives listed above but doing so would incur unnecessary overhead. For instance, objects would receive **retain** and **release** messages as they were shifted to accommodate the insertion or deletion of an element.

Method Types

Creating an NSMutableArray

+ arrayWithCapacity:
– initWithCapacity:

Adding and replacing objects

– addObject:
– addObjectsFromArray:
– insertObject:atIndex:
– replaceObjectAtIndex:withObject:
– replaceObjectsInRange:withObjectsFromArray:
– replaceObjectsInRange:withObjectsFromArray:range:
– setArray:

Removing objects

– removeAllObjects
– removeLastObject
– removeObject:
– removeObject:inRange:
– removeObjectAtIndex:
– removeObjectIdenticalTo:
– removeObjectIdenticalTo:inRange:
– removeObjectsFromIndices:numIndices:
– removeObjectsInArray:
– removeObjectsInRange:

Rearranging objects

– sortUsingFunction:context:
– sortUsingSelector:

Class Methods

arrayWithCapacity:

+ (id)**arrayWithCapacity:**(unsigned int)*numItems*

Creates and returns an NSMutableArray, giving it enough allocated memory to hold *numItems* objects. NSMutableArrays expand as needed, so *numItems* simply establishes the object's initial capacity.

See also: – **initWithCapacity:**

Instance Methods

addObject:

– (void)**addObject:**(id)*anObject*

Inserts *anObject* at the end of the receiver. The object receives a **retain** message as it's added to the array. If *anObject* is **nil**, an NSInvalidArgumentException is raised.

See also: – **addObjectsFromArray:**, – **removeObject:**, – **setArray:**

addObjectsFromArray:

– (void)**addObjectsFromArray:**(NSArray *)*otherArray*

Adds the objects contained in *otherArray* to the end of the receiver's array of objects.

See also: **– setArray:**, – **removeObject:**

initWithCapacity:

– (id)**initWithCapacity:**(unsigned int)*numItems*

Initializes a newly allocated array, giving it enough memory to hold *numItems* objects. Mutable arrays expand as needed, so *numItems* simply establishes the object's initial capacity. Returns **self**.

See also: + **arrayWithCapacity:**

insertObject:atIndex:

– (void)**insertObject:**(id)*anObject* **atIndex:**(unsigned int)*index*

Inserts *anObject* into the receiver at *index*. If *index* is already occupied, the objects at *index* and beyond are shifted down one slot to make room. *index* cannot be greater than the number of elements in the array. *anObject* receives a **retain** message as it's added to the array. This method raises an NSInvalidArgumentException if *anObject* is **nil** and raises an NSRangeException if *index* is greater than the number of elements in the array.

Note that NSArrays are not like C arrays. That is, even though you do an "initWithCapacity:," the specified capacity is regarded as a "hint"; the actual size of the array is still 0. Because of this, you can only insert new objects in ascending order—with no gaps. Once you add two objects, the array's size is 2, so you can add objects at indexes 0, 1, or 2. Index 3 is illegal and out of bounds; if you try to add an object at index 3 (when the size of the array is 2), NSMutableArray raises an exception.

See also: – **removeObjectAtIndex:**

removeAllObjects

– (void)**removeAllObjects**

Empties the receiver of all its elements. Each removed object is sent a **release** message.

See also: – **removeObject:**, – **removeLastObject**, – **removeObjectAtIndex:**, – **removeObjectIdenticalTo:**

removeLastObject

– (void)**removeLastObject**

Removes the object with the highest-valued index in the array and sends it a **release** message. **removeLastObject** raises an NSRangeException if there are no objects in the array.

See also: – **removeAllObjects**, – **removeObject:**, – **removeObjectAtIndex:**, – **removeObjectIdenticalTo:**

removeObject:

– (void)**removeObject:**(id)*anObject*

Removes all occurrences of *anObject* in the array. This method uses **indexOfObject:** to locate matches and then removes them by using **removeObjectAtIndex:**. Thus, matches are determined on the basis of an object's response to the **isEqual:** message.

See also: – **removeAllObjects**, – **removeLastObject**, – **removeObjectAtIndex:**, – **removeObjectIdenticalTo:**, – **removeObjectsInArray:**

removeObject:inRange:

– (void)**removeObject:**(id)*anObject* **inRange:(NSRange)***aRange*

Removes all occurrences of *anObject* within the specified range in the array. Matches are determined on the basis of an object's response to the **isEqual:** message and by comparing **id**s.

See also: – **removeAllObjects**, – **removeLastObject**, – **removeObjectAtIndex:**, – **removeObjectIdenticalTo:**, – **removeObjectsInArray:**

removeObjectAtIndex:

– (void)**removeObjectAtIndex:**(unsigned int)*index*

Removes the object at *index* and moves all elements beyond *index* up one slot to fill the gap. The removed object receives a **release** message. This method raises an NSRangeException if *index* is beyond the end of the array.

See also: **– insertObject:atIndex:**, – **removeAllObjects**, – **removeLastObject**, – **removeObject:**, – **removeObjectIdenticalTo:, – removeObjectsFromIndices:numIndices:**

removeObjectIdenticalTo:

– (void)**removeObjectIdenticalTo:**(id)*anObject*

Removes all occurrences of *anObject* in the array. This method uses the **indexOfObjectIdenticalTo:** method to locate matches and then removes them by using **removeObjectAtIndex:**. Thus, matches are determined on the basis of an object's **id**.

See also: – **removeAllObjects**, – **removeLastObject**, – **removeObject:**, – **removeObjectAtIndex:**

removeObjectIdenticalTo:inRange:

– (void)**removeObjectIdenticalTo:(id)***anObject* **inRange:**(NSRange)*aRange*

Removes all occurrences of *anObject* within the specified range in the array. Matches are determined by comparing object **id**s.

See also: – **removeAllObjects**, – **removeLastObject**, – **removeObject:**, – **removeObjectAtIndex:**

removeObjectsFromIndices:numIndices:

– (void)**removeObjectsFromIndices:**(unsigned int *)*indices* **numIndices:**(unsigned int)*count*

This method is similar to **removeObjectAtIndex:**, but allows you to efficiently remove multiple objects with a single operation. *count* indicates the number of objects to be removed, while *indices* points to the first in a list of indexes. Note that if you sort the list of indexes in ascending order, you will improve the speed of this operation.

This method does not distribute and therefore should be used sparingly.

See also: – **insertObject:atIndex:**, – **removeObjectAtIndex:**, **– removeObjectsInRange:**

removeObjectsInArray:

– (void)**removeObjectsInArray:**(NSArray *)*otherArray*

This method is similar to **removeObject:**, but allows you to efficiently remove large sets of objects with a single operation. It assumes that all elements in *otherArray*—which are the objects to be removed—respond to **hash** and **isEqual:**.

This method does not distribute and therefore should be used sparingly.

See also: – **removeAllObjects**, – **removeObjectIdenticalTo:**

removeObjectsInRange:

– (void)**removeObjectsInRange:**(NSRange)*aRange*

Removes each of the objects within the specified range in the receiver using **removeObjectAtIndex:**.

replaceObjectAtIndex:withObject:

– (void)**replaceObjectAtIndex:**(unsigned int)*index* **withObject:**(id)*anObject*

Replaces the object at *index* with *anObject*. *anObject* receives a **retain** message as it's added to the array, and the previous object at *index* receives a **release** message. This method raises an

NSInvalidArgumentException if *anObject* is **nil** and raises an NSRangeException if *index* is beyond the end of the array.

See also: – **insertObject:atIndex:**, – **removeObjectAtIndex:**

replaceObjectsInRange:withObjectsFromArray:

– (void)**replaceObjectsInRange:**(NSRange)*aRange* **withObjectsFromArray:**
(NSArray *)*otherArray*

Replaces the objects in the receiver specified by *aRange* with all of the objects from *otherArray*. If *otherArray* has fewer objects than are specified by *aRange*, the extra objects in the receiver are removed. If *otherArray* has more objects than are specified by *aRange*, the extra objects from *otherArray* are inserted into the receiver.

See also: – **insertObject:atIndex:**, – **removeObjectAtIndex:**, – **replaceObjectAtIndex:withObject:**

replaceObjectsInRange:withObjectsFromArray:range:

– (void)**replaceObjectsInRange:**(NSRange)*aRange* **withObjectsFromArray:**
(NSArray *)*otherArray* **range:**(NSRange)*otherRange*

Replaces the objects in the receiver specified by *aRange* with the objects in *otherArray* specified by *otherRange*. *aRange* and *otherRange* don't have to be equal; if *aRange* is greater than *otherRange*, the extra objects in the receiver are removed. If *otherRange* is greater than *aRange*, the extra objects from *otherArray* are inserted into the receiver.

See also: – **insertObject:atIndex:**, – **removeObjectAtIndex:**, – **replaceObjectAtIndex:withObject:**

setArray:

– (void)**setArray:**(NSArray *)*otherArray*

Sets the receiver's elements to those in *otherArray*. Shortens the receiver, if necessary, so that it contains no more than the number of elements in *otherArray*. Replaces existing elements in the receiver with the elements in *otherArray*, releasing those objects that are being replaced and retaining those objects that are replacing them. Finally, if there are more elements in *otherArray* than there are in the receiver, the additional items are then added (and **retain** is sent to each object as it is added to the receiver).

See also: – **addObjectsFromArray:**, – **replaceObjectAtIndex:withObject:**

sortUsingFunction:context:

– (void)**sortUsingFunction:**(int (*)(id, id, void *))*compare* **context:**(void *)*context*

Sorts the receiver's elements in ascending order as defined by the comparison function *compare*. The comparison function is used to compare two elements at a time and should return NSOrderedAscending if the first element is smaller than the second, NSOrderedDescending if the first element is larger than the second, and NSOrderedSame if the elements are equal. Each time the comparison function is called, it's passed *context* as its third argument. This allows the comparison to be based on some outside parameter, such as whether character sorting is case-sensitive or case-insensitive.

See also: – **sortUsingSelector:**, **– sortedArrayUsingFunction:context:** (NSArray)

sortUsingSelector:

– (void)**sortUsingSelector:**(SEL)*comparator*

Sorts the receiver's elements in ascending order, as determined by the comparison method specified by the selector *comparator*. The *comparator* message is sent to each object in the array, and has as its single argument another object in the array. The comparator method is used to compare two elements at a time and should return NSOrderedAscending if the receiver is smaller than the argument, NSOrderedDescending if the receiver is larger than the argument, and NSOrderedSame if they are equal.

See also: – **sortUsingFunction:context:**, **– sortedArrayUsingSelector:** (NSArray)

NSAssertionHandler

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	Foundation/NSException.h

Class Description

NSAssertionHandler objects are automatically created to handle false assertions. Assertion macros are used to evaluate a condition and, if the condition evaluates to false, the macros pass a string to an NSAssertionHandler object describing the failure. Each thread has its own NSAssertionHandler object. When invoked, an NSAssertionHandler prints an error message that includes the method and class (or function) containing the assertion, and raises an NSInternalInconsistencyException.

You use an assortment of macros to evaluate a condition—these macros serve as a front end to NSAssertionHandler. These macros fall into two categories: those you use in Objective-C methods, and those you use in C functions. For example, **NSAssert()** is for use within methods and **NSCAssert()** is for use within functions. Each macro has two arguments: the condition, an expression that evaluates to true or false, and the NSString describing the failure. Other macros are available if more arguments are needed for a **printf()**-style description. For example, **NSAssert1()** is used within methods if an additional argument is needed as in:

```
NSAssert1((0 <= component) && (component <= 255),
    @"Value %i out of range!", component);
```

For more details on these macros see **NSAssert()**.

You create assertions only using the above macros—you rarely need to invoke NSAssertionHandler methods directly. The macros for use inside methods and functions will send **handleFailureInMethod:object:file:lineNumber:description:...** and **handleFailureInFunction:file:lineNumber:description:...** messages respectively to the current assertion handler. The assertion handler for the current thread is obtained using the **currentHandler** class method.

Assertions are *not* compiled into code if the preprocessor macro NS_BLOCK_ASSERTIONS is defined.

Using a Custom Assertion Handler

In some cases, you might want to define your own assertion handler to print error messages to a different error console or to raise custom exceptions, instead of the generic NSInternalInconsistencyException. To implement these features, you must define a subclass of NSAssertionHandler and override its

handleFailureInMethod:object:file:lineNumber:description:... and **handleFailureInFunction:file:lineNumber:description:...** methods to handle assertions in methods and functions, respectively.

To add your assertion handler to a thread, you must add the assertion handler to the thread's attributes dictionary. Use the thread's **threadDictionary** method to retrieve the dictionary. Add your assertion-handler object to the dictionary using the key "NSAssertionHandler".

Typically, you should add your assertion handler to the thread dictionary immediately after creating the thread. However, a default assertion handler is not created until an assertion macro is encountered and you can always replace the existing assertion handler in the thread dictionary. If your assertion handler already exists in the thread dictionary, it is used in place of the default assertion handler.

Method Types

Getting the thread's handler

+ currentHandler

Handling assertion failures

– handleFailureInFunction:file:lineNumber:description:
– handleFailureInMethod:object:file:lineNumber:description:

Class Methods

currentHandler

+ (NSAssertionHandler *)**currentHandler**

Returns the NSAssertionHandler object associated with the current thread. If no assertion handler is associated with the current thread, this method creates one and assigns it to the thread.

Instance Methods

handleFailureInFunction:file:lineNumber:description:

– (void)**handleFailureInFunction:**(NSString *)*functionName*
file:(NSString *)*fileName*
lineNumber:(int)*line*
description:(NSString *)*format,...*

Logs an error message (using **NSLog()**) that includes the name of the function *functionName*, the name of the file *fileName*, and the line number *line*. Raises NSInternalInconsistencyException.

handleFailureInMethod:object:file:lineNumber:description:

– (void)**handleFailureInMethod:**(SEL)*selector*
object:(id)*object*
file:(NSString *)*fileName*
lineNumber:(int)*line*
description:(NSString *)*format,...*

Logs an error message (using **NSLog()**) that includes the *selector*, *object*'s class name, the name of the file *fileName*, and the line number *line*. Raises NSInternalInconsistencyException.

NSAttributedString Class Cluster

Class Cluster Description

NSAttributedString objects manage character strings and associated sets of attributes (for example, font and kerning) that apply to individual characters or ranges of characters in the string. An association of characters and their attributes is called an *attributed string*. The cluster's two public classes, NSAttributedString and NSMutableAttributedString, declare the programmatic interface for read-only attributed strings and modifiable attributed strings, respectively. The Foundation Kit defines only the basic functionality for attributed strings; the remainder of the classes' interfaces is actually defined by the Application Kit. The Application Kit also uses a subclass of NSMutableAttributedString, called NSTextStorage, to provide the storage for NeXT's extended text-handling system.

Note: NSAttributedString is *not* a subclass of NSString. It contains a string object to which it applies attributes. This protects users of attributed strings from ambiguities caused by the semantic differences between simple and attributed string. For example, equality can't be simply defined between an NSString and an attributed string.

Because of the nature of class clusters, attributed string objects are not actual instances of the NSAttributedString or NSMutableAttributedString classes, but are instances of one of their private concrete subclasses. Although an attributed string object's class is private, its interface is public, as declared by these abstract superclasses, NSAttributedString and NSMutableAttributedString. The attributed string classes adopt the NSCopying and NSMutableCopying protocols, making it convenient to convert an attributed string from one type to the other.

You create an NSAttributedString object by using one of **initWithString:**, **initWithString:attributes:**, or **initWithAttributedString:**. These methods initialize an attributed string with data you provide. The Application Kit also provides methods for creating attributed strings from RTF data.

An attributed string provides basic access to its contents with the **string** and **attributesAtIndex: effectiveRange:** methods, which yield characters and attributes, respectively. These two primitive methods are used by the other access methods to retrieve attributes individually by name, by functional group (font or ruler attributes, for example), and so on.

The attribute values assigned to an attributed string become the property of that string, and shouldn't be modified in any way by other objects. Doing so can render inconsistent the attributed string's internal state. Always use NSMutableAttributedString's **setAttributes:range:** and related methods to change attribute values.

For more information on using attributed strings, see the Application Kit's specification for this class cluster.

NSAttributedString

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSMutableCopying NSObject (NSObject)
Declared In:	Foundation/NSAttributedString.h

Class Description

The NSAttributedString class declares the programmatic interface to an object that manages an immutable attributed string. NSAttributedString's two primitive methods provide the basis for all the other methods in the class. The primitive **string** method returns the NSString object to which attributes are applied. The primitive **attributesAtIndex:effectiveRange:** method returns an NSDictionary containing the attribute names and values for characters around a specified index.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

NSCopying

– copyWithZone:

NSMutableCopying

– mutableCopyWithZone:

Method Types

Creating an NSAttributedString

– initWithString:
– initWithAttributedString:
– initWithString:attributes:

Retrieving character information
– string
– length

Retrieving attribute information
– attributesAtIndex:effectiveRange:
– attributesAtIndex:longestEffectiveRange:inRange:
– attribute:atIndex:effectiveRange:
– attribute:atIndex:longestEffectiveRange:inRange:

Comparing attributed strings
– isEqualToAttributedString:

Extracting a substring
– attributedSubstringFromRange:

Instance Methods

attribute:atIndex:effectiveRange:

– (id)**attribute:**(NSString *)*attributeName*
atIndex:(unsigned int)*index*
effectiveRange:(NSRange *)*aRange*

Returns the value for the attribute named *attributeName* of the character at *index*, or **nil** if there is no such attribute. If the named attribute exists at *index* and *aRange* is non-NULL, it's filled with a range over which the named attribute's value applies. If the named attribute doesn't exist at *index* and *aRange* is non-NULL, *aRange* is filled instead with the range over which the attribute doesn't exist. This range isn't necessarily the maximum range covered by *attributeName*, and its extent is implementation-dependent. If you need the maximum range, use **attribute:atIndex:longestEffectiveRange:inRange:**.

Raises an NSRangeException if *index* lies beyond the end of the receiver's characters.

See also: – **attributesAtIndex:effectiveRange:**

attribute:atIndex:longestEffectiveRange:inRange:

– (id)**attribute:**(NSString *)*attributeName*
atIndex:(unsigned int)*index*
longestEffectiveRange:(NSRange *)*aRange*
inRange:(NSRange)*rangeLimit*

Returns the value for the attribute named *attributeName* of the character at *index*, or **nil** if there is no such attribute. If the named attribute exists at *index* and *aRange* is non-NULL, it's filled with the full range over which the value of the named attribute is the same as that at *index*, clipped to *rangeLimit*. If the named

attribute doesn't exist at *index* and *aRange* is non-NULL, *aRange* is filled instead with the full range over which the attribute doesn't exist, clipped to *rangeLimit*.

Raises an NSRangeException if *index* or any part of *rangeLimit* lies beyond the end of the receiver's characters.

If you don't need the longest effective range, it's far more efficient to use the **attribute:atIndex: effectiveRange:** method to retrieve the attribute value.

See also: – **attributesAtIndex:longestEffectiveRange:inRange:**

attributedSubstringFromRange:

– (NSAttributedString *)**attributedSubstringFromRange:**(NSRange)*aRange*

Returns an NSAttributedString object consisting of the characters and attributes within *aRange* in the receiver. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

attributesAtIndex:effectiveRange:

– (NSDictionary *)**attributesAtIndex:**(unsigned int)*index* **effectiveRange:**(NSRange *)*aRange*

Returns the attributes for the character at *index*. If *aRange* is non-NULL it's filled with the range over which the attributes and values apply. This range isn't necessarily the maximum range covered, and its extent is implementation-dependent. If you need the maximum range, use **attributesAtIndex: longestEffectiveRange:inRange:**.

Raises an NSRangeException if *index* lies beyond the end of the receiver's characters.

See also: – **attribute:atIndex:effectiveRange:**

attributesAtIndex:longestEffectiveRange:inRange:

– (NSDictionary *)**attributesAtIndex:**(unsigned int)*index*
longestEffectiveRange:(NSRange *)*aRange*
inRange:(NSRange)*rangeLimit*

Returns the attributes for the character at *index*. If *aRange* is non-NULL, it's filled with the maximum range over which the attributes and values are the same as those at *index*, clipped to *rangeLimit*.

Raises an NSRangeException if *index* or any part of *rangeLimit* lies beyond the end of the receiver's characters.

If you don't need the range information, it's far more efficient to use the **attributesAtIndex: effectiveRange:** method to retrieve the attribute value.

See also: – **attribute:atIndex:longestEffectiveRange:inRange:**

initWithAttributedString:

– (id)**initWithAttributedString:**(NSAttributedString *)*attributedString*

Initializes a newly allocated NSAttributedString object with the characters and attributes of *attributedString*. Returns **self**.

See also: – **initWithRTF:documentAttributes:** (NSAttributedString Additions in the Application Kit)

initWithString:

– (id)**initWithString:**(NSString *)*aString*

Initializes a newly allocated NSAttributedString object with the characters of *aString* and with no attribute information. Returns **self**.

See also: – **initWithRTF:documentAttributes:** (NSAttributedString Additions in the Application Kit)

initWithString:attributes:

– (id)**initWithString:**(NSString *)*aString* **attributes:**(NSDictionary *)*attributes*

Initializes a newly allocated NSAttributedString object with the characters of *aString* and the attributes of *attributes*. Returns **self**.

See also: – **initWithRTF:documentAttributes:** (NSAttributedString Additions in the Application Kit)

isEqualToAttributedString:

– (BOOL)**isEqualToAttributedString:**(NSAttributedString *)*otherString*

Returns YES if the receiver is equal to *otherString*. Attributed strings must match in both characters and attributes to be equal.

length

– (unsigned int)**length**

Returns the length of the receiver's string object.

See also: – **length** (NSString), – **size** (NSAttributedString Additions in the Application Kit)

string

– (NSString *)**string**

Returns the character contents of the receiver as an NSString object. This method doesn't strip out attachment characters; use NSText's **string** method to extract just the linguistically significant characters.

This primitive method must guarantee efficient access to an attributed string's characters; subclasses should implement it to execute in O(1) time.

NSMutableAttributedString

Inherits From: NSAttributedString : NSObject

Conforms To: NSCoding (NSAttributedString)
NSCopying (NSAttributedString)
NSMutableCopying (NSAttributedString)
NSObject (NSObject)

Declared In: Foundation/NSAttributedString.h

Class Description

NSMutableAttributedString declares the programmatic interface to objects that manage mutable attributed strings. You can add and remove characters (raw strings) and attributes separately, or together as attributed strings. When working with the Application Kit, you must also clean up changed attributes using the various **fix...** methods. See the Application Kit's specification for this class cluster for more information on fixing attributes.

NSMutableAttributedString adds two primitive methods to those of NSAttributedString. These primitive methods provide the basis for all the other methods in its class. The primitive **replaceCharactersInRange: withString:** method replaces a range of characters with those from a string, leaving all attribute information outside that range intact. The primitive **setAttributes:range:** method sets attributes and values for a given range of characters, replacing any previous attributes and values for that range.

Method Types

Retrieving character information
– mutableString

Changing characters
– replaceCharactersInRange:withString:
– deleteCharactersInRange:

Changing attributes
– setAttributes:range:
– addAttribute:value:range:
– addAttributes:range:
– removeAttribute:range:

Changing characters and attributes

– appendAttributedString:
– insertAttributedString:atIndex:
– replaceCharactersInRange:withAttributedString:
– setAttributedString:

Grouping changes

– beginEditing
– endEditing

Instance Methods

addAttribute:value:range:

– (void)**addAttribute:**(NSString *)*name*
value:(id)*value*
range:(NSRange)*aRange*

Adds an attribute with the given *name* and *value* to the characters in *aRange*. Raises an NSInvalidArgumentException if *name* or *value* is **nil**, and an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **addAttributes:range:**, – **removeAttribute:range:**

addAttributes:range:

– (void)**addAttributes:**(NSDictionary *)*attributes* **range:**(NSRange)*aRange*

Adds the collection of attributes in *attributes* to the characters in *aRange*. Raises an NSInvalidArgumentException if *attributes* is **nil** and an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **addAttribute:value:range:**, – **removeAttribute:range:**

appendAttributedString:

– (void)**appendAttributedString:**(NSAttributedString *)*attributedString*

Adds the characters and attributes of *attributedString* to the end of the receiver.

See also: – **insertAttributedString:atIndex:**, + **attributedStringWithAttachment:** (NSAttributedString Additions in the Application Kit)

beginEditing

– (void)**beginEditing**

Overridden by subclasses to buffer or optimize a series of changes to the receiver's characters or attributes, until it receives a matching **endEditing** message, upon which it can consolidate changes and notify any observers that it has changed. You can nest pairs of **beginEditing** and **endEditing** messages.

deleteCharactersInRange:

– (void)**deleteCharactersInRange:**(NSRange)*aRange*

Deletes the characters in *aRange* along with their associated attributes. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **replaceCharactersInRange:withAttributedString:**, – **replaceCharactersInRange: withString:**

endEditing

– (void)**endEditing**

Overridden by subclasses to consolidate changes made since a previous **beginEditing** message and to notify any observers of the changes. NSMutableAttributedString's implementation does nothing. NSTextStorage, for example, overrides this method to invoke **fixAttributesInRange:** and to inform its NSLayoutManagers that they need to re-lay the text.

See also: – **processEditing** (NSTextStorage class in the Application Kit)

insertAttributedString:atIndex:

– (void)**insertAttributedString:**(NSAttributedString *)*attributedString* **atIndex:**(unsigned int)*index*

Inserts the characters and attributes of *attributedString* into the receiver, so that the new characters and attributes begin at *index* and the existing characters and attributes from *index* to the end are shifted by the length of *attributedString*. Raises an NSRangeException if *index* lies beyond the end of the receiver's characters.

See also: – **appendAttributedString:**, + **attributedStringWithAttachment:** (NSAttributedString Additions in the Application Kit)

mutableString

– (NSMutableString *)**mutableString**

Returns the character contents of the receiver as an NSMutableString object. The receiver tracks changes to this string and keeps its attribute mappings up to date.

See also: – **replaceCharactersInRange:withString:**

removeAttribute:range:

– (void)**removeAttribute:**(NSString *)*name* **range:**(NSRange)*aRange*

Removes the attribute named *name* from the characters in *aRange*. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **addAttribute:value:range:**, – **addAttributes:range:**

replaceCharactersInRange:withAttributedString:

– (void)**replaceCharactersInRange:**(NSRange)*aRange* **withAttributedString:**
(NSAttributedString *)*attributedString*

Replaces the characters and attributes in *aRange* with the characters and attributes of *attributedString*. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **insertAttributedString:atIndex:**

replaceCharactersInRange:withString:

– (void)**replaceCharactersInRange:**(NSRange)*aRange* **withString:**(NSString *)*aString*

Replaces the characters in *aRange* with the characters of *aString*. The new characters inherit the attributes of the first replaced character from *aRange*. Where the length of *aRange* is zero, the new characters inherit the attributes of the character preceding *aRange* if it has any, otherwise of the character following *aRange*.

Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **deleteCharactersInRange:**

setAttributedString:

– (void)**setAttributedString:**(NSAttributedString *)*attributedString*

Replaces the receiver's entire contents with the characters and attributes of *attributedString*.

See also: – **appendAttributedString:**

setAttributes:range:

– (void)**setAttributes:**(NSDictionary *)*attributes* **range:**(NSRange)*aRange*

Sets the attributes for the characters in *aRange* to *attributes*. These new attributes replace any attributes previously associated with the characters in *aRange*. Raises an NSRangeException if any part of *aRange* lies beyond the end of the receiver's characters.

See also: – **addAttributes:range:**, – **removeAttribute:range:**

NSAutoreleasePool

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSAutoreleasePool.h

Class Description

The NSAutoreleasePool class is used to implement the Foundation Kit's autorelease mechanism. An NSAutoreleasePool object simply contains objects that have received an **autorelease** message, and when deallocated sends a **release** message to each of those objects. An object can be put into the same pool several times, and receives a **release** message for each time it was put into the pool. Thus, sending **autorelease** instead of **release** to an object extends the lifetime of that object until the pool itself is released or longer if the object is retained. This class specification contains information on fine-tuning your application's handling of autorelease pools; see "Object Ownership and Automatic Disposal" in the Programming Topics documentation for general information on using the autorelease mechanism.

You create an NSAutoreleasePool with the usual **alloc** and **init** messages, and dispose of it with **release** (an exception will be raised if you send **autorelease** or **retain** to an autorelease pool). An autorelease pool should always be released in the same context (invocation of a method or function, or body of a loop) that it was created. When a thread terminates, it automatically releases all of the autorelease pools associated with itself.

Autorelease pools can be nested, so you can include them in any function or method. For example, a **main()** function may create an autorelease pool and call another function that creates another autorelease pool. Or a single method might have an autorelease pool for an outer loop, and another autorelease pool for an inner loop. Each thread in a program maintains autorelease pools on a stack; the most recently created (and unreleased) autorelease pool is the top pool on the stack. The ability to nest autorelease pools is a definite advantage, but there are side effects when exceptions occur (see "Implications of Nested Autorelease Pools").

NSAutoreleasePools are automatically created and destroyed in applications based on the Application Kit, so your code normally doesn't have to worry about them. (The Application Kit creates a pool at the beginning of the event loop and releases it at the end). There are two cases, though, where you might wish to create and destroy your own autorelease pools. If you're writing a program that's not based on the Application Kit, such as a command-line tool, there's no built-in support for autorelease pools; you must create and destroy them yourself. Also, if you write a loop that creates many temporary objects, you might wish to create an NSAutoreleasePool inside the loop to dispose of those objects before the next iteration.

Enabling the autorelease mechanism in a program that's not based on the Application Kit is easy. Many programs have a top-level loop where they do most of their work. To enable the autorelease mechanism

you create an NSAutoreleasePool at the beginning of this loop and release it at the end. An **autorelease** message sent in the body of the loop automatically puts its receiver into this pool.

Your **main()** function might look like this:

```
void main()
{
    NSArray *args = [[NSProcessInfo processInfo] arguments];
    unsigned count, limit = [args count];

    for (count = 1; count < limit; count++){
        NSAutoreleasePool *pool =[[NSAutoreleasePool alloc] init];
        NSString *fileContents;
        NSString *fileName;

        fileName = [args objectAtIndex:count];
        fileContents = [[NSString alloc] initWithContentsOfFile:fileName];
        [fileContents autorelease];

        /* Process the file, creating and autoreleasing more objects. */

        [pool release];
    }

    /* Do whatever cleanup is needed. */

    exit (EXIT_SUCCESS);
}
```

This program processes files passed in on the command line. The **for** loop processes one file at a time. An NSAutoreleasePool is created at the beginning of this loop and released at the end. Therefore, any object sent an **autorelease** message inside the **for** loop, such as **fileContents**, is added to **pool**, and when **pool** is released at the end of the loop those objects are also released.

Similarly, NSAutoreleasePools can be created inside any loop, even in a program based on the Application Kit, that creates and releases many objects during each iteration.

Implications of Nested Autorelease Pools

It's common to speak of autorelease pools as being nested because of the enclosure evident in code; for instance, you can have an autorelease pool in an outer loop that contains an inner loop with its own autorelease pool. But you can also think of nested autorelease pools as being on a stack, with the "inmost" autorelease pool being on top of the stack. As noted earlier, this is actually how nested autorelease pools are implemented: Each thread (NSThread) in a program maintains a stack of autorelease pools. When you create an autorelease pool, it is pushed onto the top of the current thread's stack. Autoreleased objects—that is, objects which have received an **autorelease** message or which are added through the **addObject:** class method—are always put in the autorelease pool at the top of the stack.

Released autorelease pools, if not on the top of the stack, will cause all (unreleased) autorelease pools above them on the stack to be released, along with all their objects. If you neglect to send **release** to an autorelease pool when you're finished with it (something *not* recommended), it will be released when one of the autorelease pools in which it nests is released.

This behavior has implications for exceptional conditions. If an exception occurs, and the thread suddenly transfers out of the current context, the pool associated with that context is released. However, if that pool is not the top pool on the thread's stack, all the pools above the released pool are also released (releasing all their objects in the process). The top autorelease pool on the thread's stack then becomes the pool previously underneath the released pool associated with the exceptional condition. Because of this behavior, exception handlers do not need to release objects that were sent **autorelease**. Neither is it necessary or even desireable for an exception handler to send **release** to its autorelease pool, unless the handler is re-raising the exception.

Guaranteeing the Foundation Ownership Policy

By creating an NSAutoreleasePool instead of simply releasing objects, you extend the lifetime of temporary objects to the lifetime of that pool. After an NSAutoreleasePool is deallocated, you should regard any object that was autoreleased while that pool was active as "disposed of", and not send a message to that object or return it to the invoker of your method.

If you must use a temporary object beyond an autorelease context, you can do so by sending a **retain** message to the object within the context and then send it **autorelease** after the pool has been released as in:

```
- findMatchingObject:anObject
{
    id match = nil;

    while (match == nil) {
        NSAutoreleasePool *subpool = [[NSAutoreleasePool alloc] init];

        /* Do a search that creates a lot of temporary objects. */
        match = [self expensiveSearchForObject:anObject];

        if (match != nil)
            [match retain]; /* Keep match around. */

        [subpool release];
    }

    return [match autorelease];   /* Let match go and return it. */
}
```

By sending **retain** to **match** while **subpool** is in effect and sending **autorelease** to it after **subpool** has been released, **match** is effectively moved from **subpool** to the pool that was previously active. This extends the lifetime of **match** and allows it to receive messages outside the loop and be returned to the invoker of **findMatchingObject:**.

Class Methods

addObject:

+ (void)**addObject:**(id)*anObject*

Adds *anObject* to the active autorelease pool in the current thread, so that it will be sent a **release** message when the pool itself is deallocated. The same object may be added several times to the active pool and will receive a **release** message for each time it was added. Normally you don't invoke this method directly—send **autorelease** to *anObject* instead.

See also: **– addObject:**

Instance Methods

addObject:

– (void)**addObject:**(id)*anObject*

Adds *anObject* to the receiver, so that it will be sent a **release** message when the receiver is deallocated. The same object may be added several times to the same pool and will receive a **release** message for each time it was added. Normally you don't invoke this method directly—send **autorelease** to *anObject* instead.

See also: + **addObject:**

NSBundle

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSBundle.h

Class at a Glance

Purpose

An NSBundle represents a location in the file system that groups code and resources that can be used in a program. NSBundles locate program resources, dynamically load executable code, and assist in localization. You

build a bundle in Project Builder using one of these project types: Application, Framework, Loadable Bundle, Palette.

Principal Attribute

- Directory path

Creation

– initWithPath:	(designated initializer)
+ mainBundle	Returns the NSBundle for the application wrapper.
+ bundleForClass:	Returns the NSBundle in which the class is implemented.
+ bundleWithPath:	Returns the NSBundle at a location in the file system.

Commonly Used Methods

– localizedStringForKey:value:table:	Returns a localized version of a string.
– pathForResource:ofType:	Returns the path for the specified resource.
– principalClass	Returns the principal class, dynamically loading code if needed.

Class Description

An NSBundle is an object that corresponds to a directory where related resources—including executable code—are stored. The directory, in essence, "bundles" a set of resources used by an application into convenient chunks, and the NSBundle object makes those resources available to the application. NSBundle can find requested resources in the directory and can dynamically load executable code. The term *bundle* refers both to the object and to the directory it represents.

Bundles are useful in a variety of contexts. Since bundles combine executable code with the resources used by that code, they facilitate installation and localization. NSBundles are also used to locate specific resources, to obtain localized strings, to load code dynamically, and to determine which classes are loaded.

Each resource in a bundle usually resides in its own file. Bundled resources include such things as:

- Executable code
- Images—TIFF or EPS images used by an application's user interface

- Sounds
- Localized character strings
- Nib files—Interface Builder files describing user-interface objects and their relationships

The Project Builder application defines four types of projects that build bundles as *file packages*. A file package is a directory that the Workspace Manager presents to users as if it were a simple file; the contents of the directory are hidden. The four types of Project Builder bundles are:

- **Application**. The application wrapper is a bundle that contains the resources needed to launch the application, including the application executable. This bundle is also known as the main bundle. Its extension is ".app".
- **Framework**. A framework is a directory containing a dynamic shared library and all the resources that go with that library, such as header files, images, and documentation. Its extension is ".framework".
- **Loadable Bundle**. Like an application, a loadable bundle usually contains executable code and associated resources. Loadable bundles differ from applications and frameworks because they must be explicitly loaded into a running application. (See "Loadable Bundles," below for more information.) The extension of a loadable bundle is conventionally ".bundle" but can be something else (for example, ".preference").
- **Palette**. A palette is a type of loadable bundle specialized for Interface Builder. It contains custom user-interface objects and compiled code that are loaded into an Interface Builder palette.

For all types of bundles, the executable-code file of a bundle (of which there can be only one) is in the immediate bundle directory and takes the same name as the bundle, minus the extension. Bundles also encode (as a property list) the important attributes of the bundle, such as the main nib file name, executable name, document extensions, and so forth. You can access these attributes with NSBundle's **infoDictionary** method, which returns the file's contents as an NSDictionary.

You shouldn't attempt to create an NSBundle subclass since the designated initializer, **initWithPath:**, might substitute another NSBundle for **self**.

The Main Bundle

Every application has at least one bundle—its *main bundle*—which is the ".app" directory where its executable file is located. This file is loaded into memory when the application is launched. It includes at least the **main()** function and other code necessary to start up the application. You obtain an NSBundle object corresponding to the main bundle with the class method **mainBundle**.

Framework Bundles

Frameworks are bundles that package dynamic shared libraries along with the nib files, images, and other resources that support the executable code and with the header files and documentation that describe the associated APIs. As long as your applications are dynamically linked with frameworks, you should have little need to do anything explicitly with those frameworks thereafter; in a running application, the

framework code is automatically loaded, as needed. You can however, get an NSBundle object associated with a framework by invoking the class method **bundleForClass:** specifying, as the argument, a class that's defined in the framework.

Loadable Bundles and Dynamic Loading

An application can be organized into any number of other bundles in addition to the main bundle and the bundles of linked-in frameworks. Although these loadable bundles usually reside inside the application file package, they can be located anywhere in the file system. Each loadable-bundle directory—by convention, with a ".bundle" extension—is represented in the application by a separate NSBundle object. Through this object the application can dynamically load the code and resources in the bundle when it needs them. For example, an application for managing PostScript printers may have a bundle full of PostScript code to be downloaded to printers.

The executable code files in loadable bundles hold class (and category) definitions that the NSBundle object can dynamically load while the application runs. When asked for a certain class (through the invocation of **classNamed:** or **principalClass**), the NSBundle loads the object file that contains the class definition (if it's not already loaded) and returns the class object; it also loads other classes and categories that are stored in the file.

The major advantage of bundles is application extensibility. A set of bundled classes often supports a small collection of objects that can be integrated into the larger object network already in place. (NEXTSTEP Preferences is one example of this.) The linkage is established through an instance of the *principal class*. This object might have methods to return other objects that the application can talk to, but typically all messages from the application to the subnetwork are funneled through the one instance.

Since each bundle can have only one executable file, that file should be kept free of localizable content. Anything that needs to be localized should be segregated into separate resource files and stored in localized-resource subdirectories.

Note: To create a loadable bundle—a bundle with dynamically loadable code—without using Project Builder, use the **ld**(1) **-bundle** flag on the **cc** command line.

Localized Resources

If an application is to be used in more than one part of the world, its resources may need to be customized, or "localized," for language, country, or cultural region. An application may need, for example, to have separate Japanese, English, French, Hindu, and Swedish versions of the character strings that label menu commands.

Resource files specific to a particular language are grouped together in a subdirectory of the bundle directory. The subdirectory has the name of the language (in English) followed by a ".lproj" extension (for "language project"). The application mentioned above, for example, would have **Japanese.lproj**, **English.lproj**, **French.lproj**, **Hindi.lproj**, and **Swedish.lproj** subdirectories. Each ".lproj" subdirectory in a bundle has the same set of files; all versions of a resource file must have the same name. Thus, **Hello.snd**

in **French.lproj** should be the French counterpart to the Swedish **Hello.snd** in **Swedish.lproj**, and so on. If a resource doesn't need to be localized at all, it's stored in the bundle directory itself, not in the ".lproj" subdirectories.

The user determines which set of localized resources will actually be used by the application. NSBundle objects rely on the language preferences set by the user in the Preferences application. Preferences lets users order a list of available languages so that the most preferred language is first, the second most preferred language is second, and so on.

When an NSBundle is asked for a resource file, it provides the path to the resource that best matches the user's language preferences. For details, see the descriptions of **pathForResource:ofType:inDirectory** and **pathForResource:ofType:.**

How NSBundles Locate Resources

A bundle's resources are typically stored in a directory named **Resources** within the bundle directory. Within the **Resources** directory are non-localized resources and localized resource directories (**English.lproj**, **Swedish.lproj**, and so on). Generally, when an NSBundle looks for resources, it starts at the top, in a non-localized location, and searches "downward" toward the localized directories.

For example, suppose you want to find a resource with name "Main" and type "nib". NSBundle searches the **Resource** directory at the "top" non-localized level for the file "Main.nib". If it doesn't find the file there, NSBundle then searches each of the language subdirectories in the user's preferred order of languages. By this scheme, the localized version of a resource that also exists at the top level will not be found, but the non-localized one will be. For this reason, you should place all non-localized resources in the top level of Resources, and you shouldn't put any localized resources there.

When it finds a resource, an NSBundle checks if a resource of the name "Main-*$(PLATFORM_OS)*" of type "nib" exists. *$(PLATFORM_OS)* represents the make variable of the same name, and takes on the same values that the make variable takes on at compile time. For example, on Windows, during a project build, "$(PLATFORM_OS)" in the **Makefile.postamble** will yield the value "winnt". When "Main.nib" is found in a directory, the NSBundle also sees if "Main-winnt.nib" exists in that same directory. If it does, the NSBundle returns the path to the platform-specific resource; if it does not, it returns the path to "Main.nib". Note that for "Main-winnt.nib" to be found, a file named "Main.nib" must exist in the same directory (including language-specific resource directories). You typically give one platform-general version of a resource the name without this *$(PLATFORM_OS)* suffix, and give the platform-specific versions of the resource extended names. The values that *$(PLATFORM_OS)* can take are currently "winnt" (on OPENSTEP for Windows), "nextstep" (on OPENSTEP for MACH), "hpux"(on PDO for HP-UX), and "solaris" (on PDO for Solaris).

Another way to accommodate platform-specific resources is to use the **inDirectory:** parameter of NSBundle's resource-searching methods. The **inDirectory:** parameter is intended for locating resources of a common type or purpose that are put in a single directory; for example, all images could be put into a directory called "Images" within the **Resource** directory. (However, note that the Application Kit's **imageNamed:** class method in NSImage does not support searching arbitrary directories for images.) You

can thus use specialized resource subdirectories and the **inDirectory:** parameter to manage platform-specific resources. This is not "automatic" functionality, however, since it requires the developer to specify the name of the platform-specific directory as well as replicate the required resources and language projects within that directory. The *$(PLATFORM_OS)* mechanism is simpler to use, particularly if the number of platform-specific resources is small.

Application Kit Additions to NSBundle

The Application Kit defines three categories of NSBundle: one for locating image resources, one for loading nib files, and one for accessing context help. The methods in these categories become part of the NSBundle class only for those applications that use the Application Kit. For details, see the NSBundleAdditions specification in the Application Kit reference documentation.

Method Types

Initializing an NSBundle

– initWithPath:

Getting an NSBundle

+ bundleForClass:
+ bundleWithPath:
+ mainBundle
+ allBundles
+ allFrameworks

Getting a bundled class

– classNamed:
– principalClass

Finding a resource

+ pathForResource:ofType:inDirectory:
– pathForResource:ofType:
– pathForResource:ofType:inDirectory:
– pathsForResourcesOfType:inDirectory:
– resourcePath

Getting the bundle directory

– bundlePath

Getting bundle information

– infoDictionary

Managing localized resources

– localizedStringForKey:value:table:

Loading a bundle's code

– load

Class Methods

allBundles

+ (NSArray *)**allBundles**

Returns an array of all the application's non-framework bundles. This includes the main bundle and all bundles that have been dynamically created but doesn't contain any bundles that represent frameworks.

allFrameworks

+ (NSArray *)**allFrameworks**

Returns an array of all of the application's bundles that represent frameworks. This includes frameworks which are linked into an application when the application is built and bundles for frameworks which have been dynamically created.

bundleForClass:

+ (NSBundle *)**bundleForClass:**(Class)*aClass*

Returns the NSBundle that dynamically loaded *aClass* (a loadable bundle), the NSBundle for the framework in which *aClass* is defined, or the main bundle object if *aClass* was not dynamically loaded or is not defined in a framework.

See also: + **mainBundle**, + **bundleWithPath:**

bundleWithPath:

+ (NSBundle *)**bundleWithPath:**(NSString *)*path*

Returns an NSBundle that corresponds to the specified directory *path* or **nil** if *path* does not identify an accessible bundle directory. This method allocates and initializes the returned object if it doesn't already exist.

See also: + **mainBundle**, + **bundleForClass:**

mainBundle

+ (NSBundle *)**mainBundle**

Returns an NSBundle that corresponds to the directory where the application executable is located or **nil** if this executable is not located in a accessible bundle directory. This method allocates and initializes the returned NSBundle if it doesn't already exist.

In general, the main bundle corresponds to an application file package or application wrapper: a directory that bears the name of the application and is marked by a ".app" extension.

See also: + **bundleForClass:**, + **bundleWithPath:**

pathForResource:ofType:inDirectory:

+ (NSString *)**pathForResource:**(NSString *)*name*
ofType:(NSString *)*extension*
inDirectory:(NSString *)*bundlePath*

Returns the full pathname for the resource identified by *name*, having the specified file name *extension*, and residing in the directory *bundlePath*; returns **nil** if no matching resource file exists in the bundle. The argument *bundlePath* must be a valid bundle directory. The argument *extension* can be an empty string or **nil**; in either case the pathname returned is the first one encountered with *name*, regardless of the extension. If *bundlePath* is specified, the method searches in this order:

bundlePath/**Resources**/*name.extension*
bundlePath/**Resources**/<language.lproj>/*name.extension*
bundlePath/*name.extension*
bundlePath/<language.lproj>/*name.extension*

The order of language directories searched corresponds to the user's preferences.

See also: – **localizedStringForKey:value:table:**, – **pathForResource:ofType:**, – **pathForResource: ofType:inDirectory:**, – **pathsForResourcesOfType:inDirectory:**

Instance Methods

bundlePath

– (NSString *)**bundlePath**

Returns the full pathname of the receiver's bundle directory.

classNamed:

– (Class)**classNamed:**(NSString *)*className*

Returns the class named *className*. If the bundle's executable code is not yet loaded, this method dynamically loads it into memory. The method returns **nil** if *className* isn't one of the classes associated with the receiver or if there is an error in loading the executable code containing the class implementation. Classes (and categories) are loaded from just one file within the bundle directory; this code file has the same name as the directory, but without the extension (".bundle," ".app," ".framework"). As a side-effect of code loading, the receiver posts NSBundleNotification for each class and category loaded; see "Notifications," below for details.

The following example loads a bundle's executable code containing the class "FaxWatcher."

```
- (void)loadBundle:(id)sender
{
    Class exampleClass;
    id newInstance;
    NSString *str = @"/me/Projects/BundleExample/BundleExample.bundle";
    NSBundle *bundleToLoad = [NSBundle bundleWithPath:str];
    if (exampleClass = [bundleToLoad classNamed:@"FaxWatcher"]) {
        newInstance = [[exampleClass alloc] init];
    // [newInstance doSomething];
    }
}
```

See also: – **principalClass, – load**

infoDictionary

– (NSDictionary *)**infoDictionary**

Returns a dictionary that contains information about the receiver. This information is extracted from the property list (**Info.plist**) associated with the bundle. The returned dictionary is empty if no **Info.plist** can be found. Common keys for accessing the values of the dictionary are NSExecutable, NSExtensions, NSIcon, NSMainNibFile, and NSPrincipalClass.

See also: – **principalClass**

initWithPath:

– (id)**initWithPath:**(NSString *)*fullPath*

Returns an NSBundle corresponding to the directory *fullPath*. This method initializes and returns a new instance only if there is no existing NSBundle associated with *fullPath*, in which case it deallocates **self** and returns the existing object. *fullPath* must be a full pathname for a directory; if it contains any symbolic links,

they must be resolvable. If the directory doesn't exist or the user doesn't have access to it, this method returns **nil**.

It's not necessary to allocate and initialize an instance for the main bundle; use the **mainBundle** class method to get this instance. You can also use the **bundleWithPath:** class method to obtain a bundle identified by its directory path.

See also: + **bundleForClass:**

load

– (BOOL)**load**

Dynamically loads the bundle's executable code into a running program, if the code has not already been loaded. A bundle attempts to load its code—if it has any—only once. Returns YES if the method successfully loaded the bundle's code or if the code had already been loaded. Returns NO if the method failed to load the code.

Note: You don't need to load a bundle's executable code to search the bundle's resources.

See also: – **classNamed:**, – **principalClass**

localizedStringForKey:value:table:

– (NSString *)**localizedStringForKey:**(NSString *)*key*
value:(NSString *)*value*
table:(NSString *)*tableName*

Returns a localized version of the string designated by *key* in table *tableName*. The argument *tableName* specifies the receiver's string table to search. If *tableName* is **nil** or is an empty string, the method attempts to use the table in **Localizable.strings**. The *value* argument specifies the value to return if *key* is **nil** or if a localized string for *key* can't be found in the table. If *value* is **nil** or an empty string, and a localized string is not found in the table, the method returns *key*. If *key* and *value* are both **nil**, the method returns the empty string.

Note: Using the user default NSShowNonLocalizedStrings, you can alter the behavior of **localizedStringForKey:value:table:** to log a message when the method can't find a localized string. If you set this default to YES (in the global domain or in the application's domain), then when the method can't find a localized string in the table, it logs a message to the console and capitalizes *key* before returning it.

The following example cycles through a static array of keys when a button is clicked, gets the value for each key from a strings table named **Buttons.strings**, and sets the button title with the returned value.

```
- (void)changeTitle:(id)sender
{
    static int keyIndex = 0;
    NSBundle *thisBundle = [NSBundle bundleForClass:[self class]];

    NSString *locString = [thisBundle
        localizedStringForKey:assortedKeys[keyIndex++]
        value:@"" table:@"Buttons"];
    [sender setTitle:locString];
    if (keyIndex == MAXSTRINGS) keyIndex=0;
}
```

See also: – **pathForResource:ofType:**, – **pathForResource:ofType:inDirectory:**, – **pathsForResourcesOfType:inDirectory:**, + **pathForResource:ofType:inDirectory:**

pathForResource:ofType:

– (NSString *)**pathForResource:**(NSString *)*name* **ofType:**(NSString *)*extension*

Returns the full pathname for the resource identified by *name* and having the specified file *extension*. If the *extension* argument is **nil** or an empty string (@“”), the resource sought is identified by *name*, with no extension. The method first looks for a non-localized resource in the immediate bundle directory; if the resource is not there, it looks for the resource in the language-specific “.lproj” directory (the local language is determined by user defaults).

The following code fragment gets the path to a localized sound, creates an Sound instance from it, and plays the sound.

```
NSString *soundPath;
Sound *thisSound;
NSBundle *thisBundle = [NSBundle bundleForClass:[self class]];
if (soundPath = [thisBundle pathForResource:@"Hello" ofType:@"snd"]) {
    thisSound = [[[Sound alloc] initFromSoundfile:soundPath] autorelease];
    [thisSound play];
}
```

See also: – **localizedStringForKey:value:table:**, – **pathForResource:ofType:inDirectory:**, – **pathsForResourcesOfType:inDirectory:**, + **pathForResource:ofType:inDirectory:**

pathForResource:ofType:inDirectory:

– (NSString *)**pathForResource:**(NSString *)*name*
ofType:(NSString *)*extension*
inDirectory:(NSString *)*bundlePath*

Returns the full pathname for the resource identified by *name*, having the specified file name *extension*, and residing in the directory *bundlePath*; returns **nil** if no matching resource file exists in the bundle. The argument *bundlePath* must be a valid bundle directory or **nil**. The argument *extension* can be an empty string or **nil**; in either case the pathname returned is the first one encountered with *name*, regardless of the extension. If *bundlePath* is specified, the method searches in this order:

<main bundle path>/**Resources**/*bundlePath*/*name*.*extension*
<main bundle path>/**Resources**/*bundlePath*/<language.lproj>/*name*.*extension*
<main bundle path>/*bundlePath*/*name*.*extension*
<main bundle path>/*bundlePath*/<language.lproj>/*name*.*extension*

The order of language directories searched corresponds to the user's preferences. If *bundlePath* is **nil**, the same search order as described above is followed, minus *bundlePath*.

See also: – **localizedStringForKey:value:table:**, – **pathForResource:ofType:**, – **pathsForResourcesOfType:inDirectory:**, **+ pathForResource:ofType:inDirectory:**

pathsForResourcesOfType:inDirectory:

– (NSArray *)**pathsForResourcesOfType:**(NSString *)*extension*
inDirectory:(NSString *)*bundlePath*

Returns an array containing pathnames for all bundle resources having the specified file name *extension* and residing in the directory *bundlePath*; returns an empty array if no matching resource files are found. This method provides a means for dynamically discovering bundle resources. The argument *bundlePath* must be a valid bundle directory or **nil**. The *extension* argument can be an empty string or **nil**; if you specify either of these for *extension*, however, all bundle resources are returned. Although there is no guaranteed search order, all of the following directories will be searched:

<main bundle path>/**Resources**/*bundlePath*/*name*.*extension*
<main bundle path>/**Resources**/*bundlePath*/<language.lproj>/*name*.*extension*
<main bundle path>/*bundlePath*/*name*.*extension*
<main bundle path>/*bundlePath*/<language.lproj>/*name*.*extension*

The language directories searched corresponds to the user's preferences. If *bundlePath* is **nil**, the same search order as described above is followed, minus *bundlePath*.

See also: – **localizedStringForKey:value:table:**, – **pathForResource:ofType:**, – **pathForResource:ofType:inDirectory:**, **+ pathForResource:ofType:inDirectory:**

principalClass

– (Class)**principalClass**

Returns the NSBundle's principal class after ensuring that the code containing the definition of that class is dynamically loaded. If the NSBundle encounters errors in loading or if it can't find the executable code file in the bundle directory, it returns **nil**. The principal class typically controls all the other classes in the bundle; it should mediate between those classes and classes external to the bundle. Classes (and categories) are loaded from just one file within the bundle directory. NSBundle obtains the name of the code file to load from the dictionary returned from **infoDictionary**, using "NSExecutable" as the key. The NSBundle determines its principal class in one of two ways:

- It first looks in its own information dictionary, which extracts the information encoded in the bundle's property list (**Info.plist**). NSBundle obtains the principal class from the dictionary using the key NSPrincipalClass. For non-loadable bundles (applications and frameworks), if the principal class is not specified in the property list, the method returns **nil**.
- If the principal class is not specified in the information dictionary, NSBundle identifies the first class loaded as the principal class. When several classes are linked into a dynamically loadable file, the default principal class is the first one listed on the **ld** command line. In the following example, Reporter would be the principal class:

```
ld -o myBundle -r Reporter.o NotePad.o QueryList.o
```

Note: The order of classes in Project Builder's project browser is the order in which they will be linked. To designate the principal class, Control-drag the file containing its implementation to the top of the list.

As a side-effect of code loading, the receiver posts NSBundleDidLoadNotification after each class and category is loaded; see "Notifications," below for details.

The following method obtains a bundle by specifying its path (**bundleWithPath:**), then loads the bundle with **principalClass** and uses the returned class object to allocate and initialize an instance of that class.

```
- (void)loadBundle:(id)sender
{
    Class exampleClass;
    id newInstance;
    NSString *path = @"/tmp/Projects/BundleExample/BundleExample.bundle";
    NSBundle *bundleToLoad = [NSBundle bundleWithPath:path];
    if (exampleClass = [bundleToLoad principalClass]) {
        newInstance = [[exampleClass alloc] init];
        [newInstance doSomething];
    }
}
```

See also: – **classNamed:**, – **infoDictionary**, **– load**

resourcePath

– (NSString *)**resourcePath**

Returns the full pathname of the receiving bundle's subdirectory containing resources.

See also: – **bundlePath**

Notifications

The following notification is declared and posted by NSBundle.

NSBundleDidLoadNotification

This notification contains a notification object and a userInfo dictionary. The notification object is the NSBundle that dynamically loads classes. The userInfo dictionary contains these keys and values:

Key	Value
NSLoadedClasses	An NSArray containing the names (as NSStrings) of each class that was loaded

NSBundle posts NSBundleDidLoadNotification to notify observers which classes have been dynamically loaded. When a request is made to an NSBundle for a class (**classNamed:** or **principalClass**), the bundle dynamically loads the executable code file that contain the class implementation and all other class definitions contained in the file. After the module is loaded, the NSBundle posts a notification with a **userInfo** dictionary containing all classes that were loaded.

In a typical use of this notification, an object might want to enumerate the **userInfo** NSArray to check if each loaded class conformed to a certain protocol (say, a protocol for a plug-and-play tool set); if a class does conform, the object would create an instance of that class and add the instance to another NSArray.

NSCalendarDate

Inherits From:	NSDate : NSObject
Conforms To:	NSCoding, NSCopying (NSDate) NSObject (NSObject)
Declared In:	Foundation/NSDate.h

Class Description

NSCalendarDate is a public subclass of NSDate that creates concrete date objects for western calendars. These objects have time zones and calendar formats as attributes and are especially suited for representing and manipulating dates according to western calendrical systems.

An NSCalendarDate object stores a date as the number of seconds relative to the absolute reference date (the first instance of 1 January 2001, GMT). It uses an NSTimeZone object to adjust the visible representation of that date to reflect its associated time zone. The time zone object only affects the way the date is displayed; it does not affect the value stored. Because the value is stored independent of the time zone, you can accurately compare NSCalendarDates with any other NSDate objects or use them to create other NSDate objects. It also means that you can track an NSCalendarDate object across different time zones; that is, as you can change the NSTimeZone object to see how the particular date is represented in that time zone.

NSCalendarDate provides both class and instance methods for creating objects. Some of these methods allow you to initialize NSCalendarDate objects from strings while others create objects from sets of integers corresponding the standard time values (months, hours, seconds, and so on).

To retrieve conventional elements of an NSCalendarDate object, use the **…Of…** methods. For example, **dayOfWeek** returns a number that indicates the day of the week (0 is Sunday). The **monthOfYear** method returns a number between 1 and 12 that indicates the month.

NSCalendarDate performs date computations based on western calendrical systems, primarily the Gregorian. (The algorithms are derived from public domain software described in “Calendrical Calculations,” a two-part series by Nachum Dershowitz and Edward M Reingold in *Software — Practice and Experience*).

The Calendar Format

Each NSCalendarDate object has a calendar format associated with it. This format is a string that contains date-conversion specifiers that are very similar to those used in the standard C library function **strftime()**. NSCalendarDate interprets dates that are represented as strings conforming to this format. You can set the

default format for an NSCalendarDate object at initialization time or using the **setCalendarFormat:** method. Several methods allow you to specify formats other than the one bound to the object.

The date conversion specifiers cover a range of date conventions:

Conversion Specifier	Description
%%	a '%' character
%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%c	shorthand for %X %x, the locale format for date and time
%d	day of the month as a decimal number (01-31)
%e	same as %d but does not print the leading 0 for days 1 through 9
%F	milliseconds as a decimal number (000-999)
%H	hour based on a 24-hour clock as a decimal number (00-23)
%I	hour based on a 12-hour clock as a decimal number (01-12)
%j	day of the year as a decimal number (001-366)
%m	month as a decimal number (01-12)
%M	minute as a decimal number (00-59)
%p	AM/PM designation for the locale
%S	second as a decimal number (00-59)
%w	weekday as a decimal number (0-6), where Sunday is 0
%x	date using the date representation for the locale
%X	time using the time representation for the locale
%y	year without century (00-99)
%Y	year with century (such as 1990)

Conversion Specifier	Description
%Z	time zone abbreviation (such as PDT)
%z	time zone offset in hours and minutes from GMT (HHMM)

NSString Representations for NSCalendarDates

NSCalendarDate provides several **description...** methods for representing dates as strings. These methods—**description**, **descriptionWithLocale:**, **descriptionWithCalendarFormat:**, and **descriptionWithCalendarFormat:locale:**—take an implicit or explicit calendar format. The user's locale information affects the returned string. NSCalendarDate accesses the locale information as an NSDictionary. If you use **descriptionWithLocale:** or **descriptionWithCalendarFormat:locale:**, you can specify a different locale dictionary. The following keys in the locale dictionary affect NSCalendarDates:

Locale Key	Description
NSTimeDateFormatString	Specifies how dates with times are printed, affecting strings that use the format specifiers %c, %X, or %x. The default is to use abbreviated months and days with a 24 hour clock, as in "Sun Jan 01 23:00:00 +6 2001".
NSAMPMDesignation	Specifies how the morning and afternoon designations are printed, affecting strings that use the %p format specifier. The default is AM and PM.
NSMonthNameArray	Specifies the names for the months, affecting strings that use the %B format specifier.
NSShortMonthNameArray	Specifies the abbreviations for the months, affecting strings that use the %b format specifier.
NSWeekDayNameArray	Specifies the names for the days of the week, affecting strings that use the %A format specifier.
NSShortWeekDayNameArray	Specifies the abbreviations for the days of the week, affecting strings that use the %a format specifier.

If you subclass NSCalendarDate and override **description**, you should also override **descriptionWithLocale:**. NSString's **stringWithFormat:** method uses **descriptionWithLocale:** instead of **description** when you use the %@ conversion specifier to print an NSCalendarDate. That is, this message:

```
[NSString stringWithFormat:@"The current date and time are %@",
       [MyNSCalendarDateSubclass date]]
```

will invoke **descriptionWithLocale:**.

Method Types

Creating an NSCalendarDate instance

+ calendarDate
+ dateWithString:calendarFormat:
+ dateWithString:calendarFormat:locale:
+ dateWithYear:month:day:hour:minute:second:timeZone:
– initWithString:
– initWithString:calendarFormat:
– initWithString:calendarFormat:locale:
– initWithYear:month:day:hour:minute:second:timeZone:

Retrieving date elements

– dayOfMonth
– dayOfWeek
– dayOfYear
– hourOfDay
– minuteOfHour
– monthOfYear
– secondOfMinute
– yearOfCommonEra
– dayOfCommonEra

Adjusting a date

– dateByAddingYears:months:days:hours:minutes:seconds:

Computing date intervals

– years:months:days:hours:minutes:seconds:sinceDate:

Representing dates as NSStrings

– description
– descriptionWithCalendarFormat:
– descriptionWithCalendarFormat:locale:
– descriptionWithLocale:

Getting and setting calendar formats

– setCalendarFormat:
– calendarFormat

Setting time zones

– setTimeZone:

Class Methods

calendarDate

+ (id)**calendarDate**

Creates and returns an NSCalendarDate initialized to the current date and time.

See also: + **date** (NSDate)

dateWithString:calendarFormat:

+ (id)**dateWithString:**(NSString *)*description* **calendarFormat:**(NSString *)*format*

Creates and returns an NSCalendarDate initialized with the date specified in the string *description*. NSCalendarDate uses *format* both to interpret the *description* string and as the default calendar format for this new object. *format* consists of conversion specifiers similar to those used in **strftime()**. See the class description for a discussion of date conversion specifiers. If *description* does not match *format* exactly, this method returns **nil**.

For example, let's say your company's convention for dates on correspondence takes the form "Friday, 1 July 1994, 11:45 AM." To get an NSCalendarDate object with a temporal value corresponding to this string, you would use this statement:

```
NSCalendarDate *today = [NSCalendarDate
    dateWithString:@"Friday, 1 July 1994, 11:45 AM"
    calendarFormat:@"%A, %d %B %Y, %I:%M %p"];
```

If you include a time zone in the *description* argument, this method verifies it and can substitute an alternative time zone. If the method does supply a new time zone, it applies the difference in offsets-from-GMT values between the substituted and the original time zone to the NSCalendarDate being created.

See also: + **dateWithString:calendarFormat:locale:**, – **calendarFormat**, – **initWithString: calendarFormat:**

dateWithString:calendarFormat:locale:

+ (id)**dateWithString:**(NSString *)*description*
calendarFormat:(NSString *)*format*
locale:(NSDictionary *)*locale*

Creates and returns an NSCalendarDate initialized with the date specified in the string *description*. NSCalendarDate uses *format* both to interpret the *description* string and as the default calendar format for this new object. *format* consists of conversion specifiers similar to those used in **strftime()**. The keys and values that represent the locale data in *locale* are used when parsing the string. See the class description for

a list of the date conversion specifiers and appropriate locale dictionary keys. If *description* does not match *format* exactly, this method returns **nil**.

If you include a time zone in the *description* argument, this method verifies it and can substitute an alternative time zone. If the method does supply a new time zone, it applies the difference in offsets-from-GMT values between the substituted and the original time zone to the NSCalendarDate being created.

See also: + **dateWithString:calendarFormat:**, – **calendarFormat**, – **initWithString:calendarFormat:locale:**

dateWithYear:month:day:hour:minute:second:timeZone:

+ (id)**dateWithYear:**(int)*year*
month:(unsigned int)*month*
day:(unsigned int)*day*
hour:(unsigned int)*hour*
minute:(unsigned int)*minute*
second:(unsigned int)*second*
timeZone:(NSTimeZone *)*aTimeZone*

Creates and returns an NSCalendarDate initialized with the specified values for year, month, day, hour, minute, and second and the NSTimeZone object. The *year* value must include the century (for example, 1995 instead of 95). The other values are the standard ones: 1 through 12 for months, 1 through 31 for days, 0 through 23 for hours and 0 through 59 for both minutes and seconds.

The method verifies the time zone *aTimeZone* and can substitute an alternative time zone. If the method does supply a new time zone, it applies the difference in offsets-from-GMT values between the substituted and the original time zone to the NSCalendarDate being created.

The following code fragment shows an NSCalendarDate created with a date on the fourth of July, 9 PM, Eastern Standard Time (**timeZoneWithName:** returns the NSTimeZone object that represents the time zone with the specified name).

```
NSCalendarDate *fireworks = [NSCalendarDate dateWithYear:1994 month:7
    day:4 hour:21 minute:0 second:0
    timeZone:[NSTimeZone timeZoneWithName:@"EST"]];
```

See also: – **initWithYear:month:day:hour:minute:second:timeZone:**

Instance Methods

calendarFormat

– (NSString *)**calendarFormat**

Returns the receiver's default calendar format (used when the format is unspecified). You can set this format when you create the NSCalendarDate using one of the class methods **dateWithString:calendarFormat:** or **dateWithString:calendarFormat:locale:**, or you can change the format using the instance method **setCalendarFormat:**. If you do not specify a default calendar format, NSCalendarDate substitutes its own default: an international format of "%Y-%m-%d %H:%M:%S %z" (for example, 1994-01-14 16:45:12 +0900). See the class description for a discussion of date conversion specifiers.

See also: – **description**, – **descriptionWithLocale:**

dateByAddingYears:months:days:hours:minutes:seconds:

– (NSCalendarDate *)**dateByAddingYears:**(int)*year*
months:(int)*month*
days:(int)*day*
hours:(int)*hour*
minutes:(int)*minute*
seconds:(int)*second*

Returns an NSCalendarDate that is updated with the year, month, day, hour, minute, and second offsets specified as arguments. The offsets can be positive (future) or negative (past). This method preserves "clock time" across changes in Daylight Savings Time zones and leap years. For example, adding one month to an NSCalendarDate with a time of 12 noon correctly maintains time at 12 noon.

The following code fragment shows an NSCalendarDate created with a date a week later than an existing NSCalendarDate.

```
NSCalendarDate *now = [NSCalendarDate calendarDate];
NSCalendarDate *nextWeek = [now dateByAddingYears:0 months:0 days:7 hours:0
    minutes:0 seconds:0];
```

See also: – **years:months:days:hours:minutes:seconds:sinceDate:**

dayOfCommonEra

– (int)**dayOfCommonEra**

Returns the number of days since the beginning of the Common Era. The base year of the Common Era is 1 A.C.E. (which is the same as 1 A.D.).

See also: – **dayOfMonth**, – **dayOfWeek**, – **dayOfYear**, – **hourOfDay**, – **minuteOfHour**, – **monthOfYear**, – **secondOfMinute**, – **yearOfCommonEra**

dayOfMonth

– (int)**dayOfMonth**

Returns a number that indicates the day of the month (1 through 31) of the receiver.

See also: – **dayOfCommonEra**, – **dayOfWeek**, – **dayOfYear**, – **hourOfDay**, – **minuteOfHour**, – **monthOfYear**, – **secondOfMinute**, – **yearOfCommonEra**

dayOfWeek

– (int)**dayOfWeek**

Returns a number that indicates the day of the week (0 through 6) of the receiver; 0 indicates Sunday.

See also: – **dayOfCommonEra**, – **dayOfMonth**, – **dayOfYear**, – **hourOfDay**, – **minuteOfHour**, – **monthOfYear**, – **secondOfMinute**, – **yearOfCommonEra**

dayOfYear

– (int)**dayOfYear**

Returns a number that indicates the day of the year (1 through 366) of the receiver.

See also: – **dayOfCommonEra**, – **dayOfMonth**, – **dayOfWeek**, – **hourOfDay**, – **minuteOfHour**, – **monthOfYear**, – **secondOfMinute**, – **yearOfCommonEra**

description

– (NSString *)**description**

Returns a string representation of the receiver. The string is formatted as specified by the receiver's default calendar format. You can find out what the default calendar format is using the method **calendarFormat**.

Because NSCalendarDate implements **descriptionWithLocale:**, **descriptionWithLocale:** is used to print the date when you use the %@ conversion specifier. That is, the following statement invokes **descriptionWithLocale:**, not **description**.

```
NSLog(@"The current date and time is %@", [NSCalendarDate date]);
```

See also: – **descriptionWithCalendarFormat:**, – **descriptionWithCalendarFormat:locale:**, – **descriptionWithLocale:**, – **setCalendarFormat:**

descriptionWithCalendarFormat:

– (NSString *)**descriptionWithCalendarFormat:**(NSString *)*format*

Returns a string representation of the receiver. The string is formatted as specified by the conversion specifiers in the calendar format string *format*. The conversion specifiers cover a range of date conventions. See the class description for a listing of these specifiers.

This example displays the current date formatted as “Tues 3/1/94 3:30 PM” in a text field:

```
NSCalendarDate *now = [NSCalendarDate calendarDate];
NSString *datestr =
    [now descriptionWithCalendarFormat:@"%a %m/%d/%y %I:%M %p"];
[dateField setStringValue:datestr];
```

See also: – **description**, – **descriptionWithCalendarFormat:locale:**, – **descriptionWithLocale:**

descriptionWithCalendarFormat:locale:

– (NSString *)**descriptionWithCalendarFormat:**(NSString *)*format* **locale:**(NSDictionary *)*locale*

Returns a string representation of the receiver. The string is formatted according to the conversion specifiers in *format* and represented according to the locale information in *locale*. See the class description for a list of the date conversion specifiers and appropriate locale dictionary keys.

See also: – **description**, – **descriptionWithCalendarFormat:**, – **descriptionWithLocale:**

descriptionWithLocale:

– (NSString *)**descriptionWithLocale:**(NSDictionary *)*locale*

Returns a string representation of the receiver. The string is formatted as specified by the receiver’s default calendar format and represented according to the locale information in *locale*. You can find out what the default calendar format is using the method **calendarFormat**. See the class description for a list of the locale dictionary keys that affect calendar formats.

This method is used to print the NSCalendarDate when the %@ conversion specifier is used. That is, this statement invokes **descriptionWithLocale:**.

```
NSLog(@"The current date and time is %@", [NSCalendarDate date]);
```

See also: – **description**, – **descriptionWithCalendarFormat:**, – **descriptionWithCalendarFormat: locale:**, – **setCalendarFormat:**

hourOfDay

– (int)**hourOfDay**

Returns the hour value (0 through 23) of the receiver. On Daylight Savings “fall back” days, a value of 1 is returned for two consecutive hours, but with a different time zone (the first in daylight savings time and the second in standard time).

See also: – **dayOfCommonEra**, – **dayOfMonth**, – **dayOfWeek**, – **dayOfYear**, – **minuteOfHour**, – **monthOfYear**, – **secondOfMinute**, – **yearOfCommonEra**

initWithString:

– (id)**initWithString:**(NSString *)*description*

Returns an NSCalendarDate initialized with the date specified as a string in *description*. The *description* string must conform to the default calendar format “%Y-%m-%d %H:%M:%S %z” (for example, 1994-01-14 16:45:12 +0900). If *description* does not match this format exactly, this method returns **nil**. See the class description for a discussion of date conversion specifiers.

See also: + **dateWithString:** (NSDate)

initWithString:calendarFormat:

– (id)**initWithString:**(NSString *)*description* **calendarFormat:**(NSString *)*format*

Returns an NSCalendarDate initialized with the date specified as a string in *description*. This method uses *format* both to interpret the *description* string and as the default calendar format for this object. *format* consists of conversion specifiers similar to those used in **strftime()**. See the class description for a listing of these specifiers. If *description* does not match *format* exactly, this method returns **nil**.

If you include a time zone in the *description* argument, this method verifies it and can substitute an alternative time zone. If the method does supply a new time zone, it applies the difference in offsets-from-GMT values between the substituted and the original time zone to the NSCalendarDate being created.

For example, let’s assume you want to initialize an NSCalendarDate object with a string obtained from a text field. This date string takes the form “03.21.94 22:00 PST”:

```
NSCalendarDate *newDate = [[[NSCalendarDate alloc]
    initWithString:[dateField stringValue]
    calendarFormat:@"%m.%d.%y %H:%M %Z"] autorelease];
```

See also: + **dateWithString:calendarFormat:**, – **calendarFormat**

initWithString:calendarFormat:locale:

– (id)**initWithString:**(NSString *)*description*
calendarFormat:(NSString *)*format*
locale:(NSDictionary *)*locale*

Returns an NSCalendarDate initialized with the date specified in the string *description*. NSCalendarDate uses *format* both to interpret the *description* string and as the default calendar format for this object. *format* consists of conversion specifiers similar to those used in **strftime()**. The keys and values that represent the locale data from *locale* are used when parsing the string. See the class description for a list of the date conversion specifiers and appropriate locale dictionary keys.

If you include a time zone in the *description* argument, this method verifies it and can substitute an alternative time zone. If the method does supply a new time zone, it applies the difference in offsets-from-GMT values between the substituted and the original time zone to the NSCalendarDate being created.

If you specify a locale dictionary that has a month name array with more than 12 elements or a day name array with more than 7 arguments, **initWithString:calendarFormat:locale:** returns **nil**.

See also: **+ dateWithString:calendarFormat:locale:, – calendarFormat**

initWithYear:month:day:hour:minute:second:timeZone:

– (id)**initWithYear:**(int)*year*
month:(unsigned int)*month*
day:(unsigned int)*day*
hour:(unsigned int)*hour*
minute:(unsigned int)*minute*
second:(unsigned int)*second*
timeZone:(NSTimeZone *)*aTimeZone*

Returns an NSCalendarDate initialized with the specified values for year, month, day, hour, minute, and second and the NSTimeZone object. The *year* value must include the century (for example, 1995 instead of 95). The other values are the standard ones: 1 through 12 for months, 1 through 31 for days, 0 through 23 for hours and 0 through 59 for both minutes and seconds.

The method verifies the time zone supplied as an argument and can substitute an alternative time zone. If the method does supply a new time zone, it applies the difference in offsets-from-GMT values between the substituted and the original time zone to the NSCalendarDate being created.

The following code fragment shows an NSCalendarDate created with a date on the fourth of July, 9 PM, Eastern Standard Time (**timeZoneWithName:** returns the NSTimeZone object that represents the time zone with the specified name).

```
NSCalendarDate *fireworks = [[[NSCalendarDate alloc] initWithYear:1994
    month:7 day:4 hour:21 minute:0 second:0
    timeZone:[NSTimeZone timeZoneWithName:@"EST"]] autorelease];
```

See also: + **dateWithYear:month:day:hour:minute:second:timeZone:**

minuteOfHour

– (int)**minuteOfHour**

Returns the minutes value (0 through 59) of the receiver.

See also: – **dayOfCommonEra**, – **dayOfMonth**, – **dayOfWeek**, – **dayOfYear**, – **hourOfDay**, – **monthOfYear**, – **secondOfMinute**, – **yearOfCommonEra**

monthOfYear

– (int)**monthOfYear**

Returns a number that indicates the month of the year (1 through 12) of the receiver.

See also: – **dayOfCommonEra**, – **dayOfMonth**, – **dayOfWeek**, – **dayOfYear**, – **hourOfDay**, – **minuteOfHour**, – **secondOfMinute**, – **yearOfCommonEra**

secondOfMinute

– (int)**secondOfMinute**

Returns the seconds value (0 through 59) of the receiver.

See also: – **dayOfCommonEra**, – **dayOfMonth**, – **dayOfWeek**, – **dayOfYear**, – **hourOfDay**, – **minuteOfHour**, – **monthOfYear**, – **yearOfCommonEra**

setCalendarFormat:

– (void)**setCalendarFormat:**(NSString *)*format*

Sets the default calendar format for the receiver. A calendar format is a string formatted with date-conversion specifiers listed in the class description. If you do not specify a calendar format for an object, NSCalendarDate substitutes its own default. This is the international format of “%Y-%m-%d %H:%M:%S %z” (for example, 1994-01-14 16:45:12 +0900).

See also: – **calendarFormat**, – **description**, – **descriptionWithLocale:**

setTimeZone:

– (void)**setTimeZone:**(NSTimeZone *)*aTimeZone*

Sets the time zone for the receiver. If you do not specify a time zone for an object at initialization time, NSCalendarDate uses the default time zone for the locale. Use this method to set it to another time zone.

See also: – **timeZone**

timeZone

– (NSTimeZone *)**timeZone**

Returns the time zone object associated with the receiver. You can set the time zone when you create the NSCalendarDate using the class methods **dateWithString:calendarFormat:** or **dateWithString: calendarFormat:locale:** by including the time zone in the description and format arguments. Or you can explicitly set the time zone to an NSTimeZone object using **dateWithYear:month:day:hour:minute: second:timeZone:**. If you do not specify a time zone for an object at initialization time, NSCalendarDate uses the default time zone for the locale.

See also: – **setTimeZone:**

yearOfCommonEra

– (int)**yearOfCommonEra**

Returns a number that indicates the year, including the century, of the receiver (for example, 1995). The base year of the Common Era is 1 A.C.E. (which is the same as 1 A.D).

See also: – **dayOfCommonEra**, – **dayOfMonth**, – **dayOfWeek**, – **dayOfYear**, – **hourOfDay**, – **minuteOfHour**, – **monthOfYear**, – **secondOfMinute**

years:months:days:hours:minutes:seconds:sinceDate:

– (void)**years:**(int *)*yearsPointer*
months:(int *)*monthsPointer*
days:(int *)*daysPointer*
hours:(int *)*hoursPointer*
minutes:(int *)*minutesPointer*
seconds:(int *)*secondsPointer*
sinceDate:(NSCalendarDate *)*date*

Computes the calendrical time difference between the receiver and *date* and returns it in *yearsPointer*, *monthsPointer*, *daysPointer*, *hoursPointer*, *minutesPointer*, and *secondsPointer*. You can choose any representation you wish for the time difference by passing NULL for the arguments you want to ignore. For

example, the following code fragment computes the difference in months, days, and years between two dates:

```
NSCalendarDate *momsBDay = [NSCalendarDate dateWithYear:1936
    month:1 day:8 hour:7 minute:30 second:0
    timeZone:[NSTimeZone timeZoneWithName:@"EST"]];
NSCalendarDate *dateOfBirth = [NSCalendarDate dateWithYear:1965
    month:12 day:7 hour:17 minute:25 second:0
    timeZone:[NSTimeZone timeZoneWithName:@"EST"]];
int years, months, days;

[dateOfBirth years:&years months:&months days:&days hours:NULL
    minutes:NULL seconds:NULL sinceDate:momsBDay];
```

This message returns 29 years, 10 months, and 29 days. If you want to express the years in terms of months, you pass NULL for the years argument:

```
[dateOfBirth years:NULL months:&months days:&days hours:NULL
    minutes:NULL seconds:NULL sinceDate:momsBDay];
```

This message returns 358 months and 29 days.

See also: – **dateByAddingYears:months:days:hours:minutes:seconds:**

NSCharacterSet Class Cluster

Class Cluster Description

An NSCharacterSet object represents a set of Unicode characters. NSString and NSScanner objects use NSCharacterSets to group characters together for searching operations, so that they can find any of a particular set of characters during a search. The cluster's two public classes, NSCharacterSet and NSMutableCharacterSet, declare the programmatic interface for static and dynamic character sets, respectively.

The objects you create using these classes are referred to as *character set objects* (and when no confusion will result, merely as *character sets*). Because of the nature of class clusters, character set objects aren't actual instances of the NSCharacterSet or NSMutableCharacterSet classes but of one of their private subclasses. Although a character set object's class is private, its interface is public, as declared by these abstract superclasses, NSCharacterSet and NSMutableCharacterSet. The character set classes adopt the NSCopying and NSMutableCopying protocols, making it convenient to convert a character set of one type to the other.

Building a Character Set

NSCharacterSet defines class methods that return commonly used character sets, such as letters (uppercase or lowercase), decimal digits, whitespace, and so on. These "standard" character sets are always immutable, even if created by sending a message to NSMutableCharacterSet. See ""Standard Character Sets and Unicode Definitions"" below for more information on standard character sets.

You can use a standard character set as a starting point for building a custom set by making a mutable copy of it and changing that. (You can also start from scratch by creating a mutable character set with **alloc** and **init** and adding characters to it.) For example, this fragment creates a character set containing letters, digits, and basic punctuation:

```
NSMutableCharacterSet *workingSet;
NSCharacterSet *finalCharSet;

workingSet = [[NSCharacterSet alphanumericCharacterSet] mutableCopy];
[workingSet addCharactersInString:@";:,."];
finalCharSet = [workingSet copy];
[workingSet release];
```

For performance reasons (explained in ""Using a Character Set""), always finish by converting the working mutable character set into an immutable set. If you need to keep changing the character set after you've created it, of course, you should just use the mutable set.

If your application frequently uses a custom character set, you'll want to save its definition in a resource file and load that instead of explicitly adding individual characters each time you need to create the set. You

can save a character set by getting its bitmap representation (an NSData object) and saving that object to a file:

```
NSString *filename;    /* Assume this exists. */
NSString *absolutePath;
NSData *charSetRep;
BOOL result;

absolutePath = [filename stringByStandardizingPath];
charSetRep = [finalCharSet bitmapRepresentation];
result = [charSetRep writeToFile:absolutePath atomically:YES];
```

Character set filenames by convention use the extension **.bitmap**. If you intend for others to use your character set files, you should follow this convention. To read a character set file with a **.bitmap** extension, simply use the **characterSetWithContentsOfFile:** method.

Using a Character Set

A character set object doesn't perform any tasks; it simply holds a set of character values to limit operations on strings. The NSString and NSScanner classes define methods that take NSCharacterSets as arguments to find any of several characters. For example, this code excerpt finds the range of the first uppercase letter in **myString**:

```
NSString *myString = @"some text in an NSString...";
NSRange letterRange;

letterRange = [myString rangeOfCharacterFromSet:[NSCharacterSet
    uppercaseLetterCharacterSet]];
```

After this fragment executes, **letterRange.location** is equal to the index of the first "N" in "NSString" after **rangeOfCharacterFromSet:** is invoked. If the first letter of the string were "S" then **letterRange.location** would be 0.

Because character sets often participate in performance-critical code, you should be aware of the aspects of their use that can affect the performance of your application. Mutable character sets are generally much more expensive than immutable character sets. They consume more memory and are costly to invert (an operation often performed in scanning a string). Because of this, you should follow these guidelines:

- Create as few mutable character sets as possible.
- Cache character sets (in a global dictionary, perhaps) instead of continually recreating them.
- When creating a custom set that doesn't need to change after creation, make an immutable copy of the final character set for actual use, and dispose of the working mutable character set. Alternatively, create a character set file as described in ""Building a Character Set"" and store it in your application's main bundle.

- Similarly, avoid archiving character set objects; store them in character set files instead. Archiving can result in a character set being duplicated in different archive files, resulting in wasted disk space and duplicates in memory for each separate archive read.

Standard Character Sets and Unicode Definitions

The standard character sets, such as that returned by **letterCharacterSet**, are formally defined in terms of the normative and informative categories established by the Unicode standard, such as Uppercase Letter, Combining Mark, and so on. The formal definition of a standard character set is in most cases given as one or more of the categories defined in the standard. For example, the set returned by **lowercaseLetterCharacterSet** include all characters in normative category Lowercase Letters, while the set returned by **letterCharacterSet** includes the characters in all of the Letter categories.

Note that the definitions of the categories themselves may change with new versions of the Unicode standard. You can download the files that define category membership from **http://www.unicode.org/**.

NSCharacterSet

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSMutableCopying NSObject (NSObject)
Declared In:	Foundation/NSCharacterSet.h

Class Description

The NSCharacterSet class declares the programmatic interface for an object that manages a set of Unicode characters (see the NSString class cluster specification for information on Unicode). NSCharacterSet's principal primitive method, **characterIsMember:**, provides the basis for all other instance methods in its interface. A subclass of NSCharacterSet needs only to implement this method, plus **mutableCopyWithZone:**, for proper behavior. For optimal performance, a subclass should also override **bitmapRepresentation** which otherwise works by invoking **characterIsMember:** for every possible Unicode value.

Adopted Protocols

NSCoding	
	– encodeWithCoder:
	– initWithCoder:
NSCopying	
	– copyWithZone:
NSMutableCopying	
	– mutableCopyWithZone:

Method Types

Creating a standard character set

+ alphanumericCharacterSet
+ controlCharacterSet
+ decimalDigitCharacterSet
+ decomposableCharacterSet
+ illegalCharacterSet
+ letterCharacterSet
+ lowercaseLetterCharacterSet
+ nonBaseCharacterSet
+ punctuationCharacterSet
+ uppercaseLetterCharacterSet
+ whitespaceAndNewlineCharacterSet
+ whitespaceCharacterSet

Creating a custom character set

+ characterSetWithRange:
+ characterSetWithCharactersInString:
+ characterSetWithBitmapRepresentation:

Opening a character set file

+ characterSetWithContentsOfFile:

Testing set membership

– characterIsMember:

Inverting a character set

– invertedSet

Getting a binary representation

– bitmapRepresentation

Class Methods

alphanumericCharacterSet

+ (NSCharacterSet *)**alphanumericCharacterSet**

Returns a character set containing the characters in the categories Letters, Marks, and Numbers. Informally, this is the set of all characters used as basic units of alphabets, syllabaries, ideographs, and digits.

See also: + **letterCharacterSet**, + **decimalDigitCharacterSet**

characterSetWithBitmapRepresentation:

+ (NSCharacterSet *)**characterSetWithBitmapRepresentation:**(NSData *)*data*

Returns a character set containing characters determined by the bitmap representation *data*. This method is useful for creating a character set object with data from a file or other external data source.

A raw bitmap representation of a character set is a byte array of 2**16** bits (that is, 8192 bytes). The value of the bit at position *n* represents the presence in the character set of the character with decimal Unicode value *n*. To add a character with decimal Unicode value *n* to a raw bitmap representation, use a statement such as:

```
unsigned char bitmapRep[8192];

bitmapRep[n >> 3] |= (((unsigned int)1) << (n & 7));
```

To remove that character:

```
bitmapRep[n >> 3] &= ~(((unsigned int)1) << (n & 7));
```

See also: – **bitmapRepresentation**, + **characterSetWithContentsOfFile:**

characterSetWithCharactersInString:

+ (NSCharacterSet *)**characterSetWithCharactersInString:**(NSString *)*aString*

Returns a character set containing the characters in *aString*. Returns an empty character set if *aString* is empty.

characterSetWithContentsOfFile:

+ (NSCharacterSet *)**characterSetWithContentsOfFile:**(NSString *)*path*

Returns a character set read from the bitmap representation stored in the file at *path*, which must end with the extension **.bitmap**. To read a bitmap representation from any file, use NSData's **dataWithContentsOfFile:** method and pass the result to **characterSetWithBitmapRepresentation:**.

This method doesn't perform filename-based uniquing of the character sets it creates. To prevent duplication of character sets in memory, cache them and make them available through an API that checks whether the requested set has already been loaded.

characterSetWithRange:

+ (NSCharacterSet *)**characterSetWithRange:**(NSRange)*aRange*

Returns a character set containing characters whose Unicode values are given by *aRange*. *aRange.location* is the value of the first character, and *aRange.location* + *aRange.length* – 1 is the value of the last. Returns an empty character set if *aRange.length* is 0.

This code excerpt creates a character set object containing the lowercase English alphabetic characters:

```
NSRange lcEnglishRange;
NSCharacterSet *lcEnglishLetters;

lcEnglishRange.location = (unsigned int)'a';
lcEnglishRange.length = 26;
lcEnglishLetters = [NSCharacterSet characterSetWithRange:lcEnglishRange];
```

controlCharacterSet

+ (NSCharacterSet *)**controlCharacterSet**

Returns a character set containing the characters in the categories of Control or Format Characters. These are specifically the Unicode values U+0000 to U+001F and U+007F to U+009F.

See also: + **illegalCharacterSet**

decimalDigitCharacterSet

+ (NSCharacterSet *)**decimalDigitCharacterSet**

Returns a character set containing the characters in the category of Decimal Numbers. Informally, this is the set of all characters used to represent the decimal values 0 through 9. These include, for example, the decimal digits of the Indic scripts and Arabic.

See also: + **alphanumericCharacterSet**

decomposableCharacterSet

+ (NSCharacterSet *)**decomposableCharacterSet**

Returns a character set containing all individual Unicode characters that can also be represented as composed character sequences (such as for letters with accents), by the definition of "standard decomposition" in version 1.1 of the Unicode character encoding standard. These include compatibility characters as well as precomposed characters.

Note: This character set doesn't currently include the Hangul characters defined in version 2.0 of the Unicode standard.

See also: + **nonBaseCharacterSet**

illegalCharacterSet

+ (NSCharacterSet *)**illegalCharacterSet**

Returns a character set containing values in the category of Non-Characters, or that have not yet been defined in version 2.0 of the Unicode standard.

See also: + **controlCharacterSet**

letterCharacterSet

+ (NSCharacterSet *)**letterCharacterSet**

Returns a character set containing the characters in the categories Letters and Marks. Informally, this is the set of all characters used as letters of alphabets and ideographs.

See also: + **alphanumericCharacterSet**, + **lowercaseLetterCharacterSet**, + **uppercaseLetterCharacterSet**

lowercaseLetterCharacterSet

+ (NSCharacterSet *)**lowercaseLetterCharacterSet**

Returns a character set containing the characters in the category of Lowercase Letters. Informally, this is the set of all characters used as lowercase letters in alphabets which make case distinctions.

See also: + **uppercaseLetterCharacterSet**, + **letterCharacterSet**

nonBaseCharacterSet

+ (NSCharacterSet *)**nonBaseCharacterSet**

Returns a character set containing the characters in the category of Marks. This set is also defined as all legal Unicode characters with a non-spacing priority greater than zero. Informally, this is the set of all characters used as modifiers of base characters.

See also: + **decomposableCharacterSet**

punctuationCharacterSet

+ (NSCharacterSet *)**punctuationCharacterSet**

Returns a character set containing the characters in the category of Punctuation. Informally, this is the set of all non-whitespace characters used to separate linguistic units in scripts, such as periods, dashes, parentheses, and so on.

uppercaseLetterCharacterSet

+ (NSCharacterSet *)**uppercaseLetterCharacterSet**

Returns a character set containing the characters in the category of Uppercase Letters. Informally, this is the set of all characters used as uppercase letters in alphabets which make case distinctions.

See also: + **lowercaseLetterCharacterSet**, + **letterCharacterSet**

whitespaceAndNewlineCharacterSet

+ (NSCharacterSet *)**whitespaceAndNewlineCharacterSet**

Returns a character set containing only the whitespace characters space (U+0020) and tab (U+0009) and the newline character (U+000A).

See also: + **whitespaceCharacterSet**

whitespaceCharacterSet

+ (NSCharacterSet *)**whitespaceCharacterSet**

Returns a character set containing only the in-line whitespace characters space (U+0020) and tab (U+0009). This set doesn't contain the newline or carriage return characters.

See also: + **whitespaceAndNewlineCharacterSet**

Instance Methods

bitmapRepresentation

– (NSData *)**bitmapRepresentation**

Returns an NSData object encoding the receiving character set in binary format. This format is suitable for saving to a file or otherwise transmitting or archiving.

A raw bitmap representation of a character set is a byte array of 2**16** bits (that is, 8192 bytes). The value of the bit at position *n* represents the presence in the character set of the character with decimal Unicode value *n*. To test for the presence of a character with decimal Unicode value *n* in a raw bitmap representation, use an expression such as:

```
unsigned char bitmapRep[8192];

if (bitmapRep[n >> 3] & (((unsigned int)1) << (n & 7))) {
    /* Character is present. */
}
```

See also: + **characterSetWithBitmapRepresentation:**

characterIsMember:

– (BOOL)**characterIsMember:**(unichar)*aCharacter*

Returns YES if *aCharacter* is in the receiving character set, NO if it isn't.

invertedSet

– (NSCharacterSet *)**invertedSet**

Returns a character set containing only characters that *don't* exist in the receiver. Inverting an immutable character set is much more efficient that inverting a mutable character set.

See also: – **invert** (NSMutableCharacterSet)

NSMutableCharacterSet

Inherits From: NSCharacterSet : NSObject

Conforms To: NSCoding (NSCharacterSet)
NSCopying (NSCharacterSet)
NSMutableCopying (NSCharacterSet)
NSObject (NSObject)

Declared In: Foundation/NSCharacterSet.h

Class Description

The NSMutableCharacterSet class declares the programmatic interface to objects that manage a modifiable set of Unicode characters. You can add or remove characters from a mutable character set as numeric values in NSRanges or as character values in strings; combine character sets by union or intersection; and invert a character set.

Mutable character sets are less efficient to use than immutable character sets. If you don't need to change a character set after creating it, create an immutable copy with **copy** and use that.

NSMutableCharacterSet defines no primitive methods. Subclasses must implement all methods declared by this class in addition to the primitives of NSCharacterSet. They must also implement **mutableCopyWithZone:**.

Method Types

Adding and removing characters
- – addCharactersInRange:
- – removeCharactersInRange:
- – addCharactersInString:
- – removeCharactersInString:

Combining character sets
- – formIntersectionWithCharacterSet:
- – formUnionWithCharacterSet:

Inverting a character set
- – invert

Instance Methods

addCharactersInRange:

– (void)**addCharactersInRange:**(NSRange)*aRange*

Adds the characters whose integer values are given by *aRange* to the receiver. *aRange.location* is the value of the first character to add, *aRange range.location* + *aRange.length* – 1 is the value of the last. If *aRange.length* is 0 this method has no effect.

See also: – **removeCharactersInRange:**, – **addCharactersInString:**

addCharactersInString:

– (void)**addCharactersInString:**(NSString *)*aString*

Adds the characters in *aString* to those in the receiver. This method has no effect if *aString* is empty.

See also: – **removeCharactersInString:**, – **addCharactersInRange:**

formIntersectionWithCharacterSet:

– (void)**formIntersectionWithCharacterSet:**(NSCharacterSet *)*otherSet*

Modifies the receiver so that it contains only characters that exist in both the receiver and in *otherSet*.

See also: – **formUnionWithCharacterSet:**

formUnionWithCharacterSet:

– (void)**formUnionWithCharacterSet:**(NSCharacterSet *)*otherSet*

Modifies the receiver so that it contains all characters that exist in either the receiver or *otherSet*.

See also: – **formIntersectionWithCharacterSet:**

invert

– (void)**invert**

Replaces all of the characters in the receiver with all the characters it didn’t previously contain. Inverting a mutable character set, whether by **invert** or by **invertedSet**, is much less efficient that inverting an immutable character set with **invertedSet**.

See also: – **invertedSet** (NSCharacterSet)

removeCharactersInRange:

– (void)**removeCharactersInRange:**(NSRange)*aRange*

Removes from the receiver the characters whose integer values are given by *aRange*. *aRange.location* is the value of the first character to remove, and *aRange.location* + *aRange.length* – 1 is the value of the last. If *aRange.length* is 0 this method has no effect.

See also: – **addCharactersInRange:**, – **removeCharactersInString:**

removeCharactersInString:

– (void)**removeCharactersInString:**(NSString *)*aString*

Removes the characters in *aString* from those in the receiver. This method has no effect if *aString* is empty.

See also: – **addCharactersInString:**, – **removeCharactersInRange:**

NSCoder

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	Foundation/NSCoder.h Foundation/NSGeometry.h Foundation/NSCompatibility.h

Class Description

The NSCoder abstract class declares the interface used by concrete subclasses to transfer objects and other Objective-C data items between memory and some other format. This capability provides the basis for archiving (where objects and data items are stored on disk) and distribution (where objects and data items are copied between different processes or threads). The concrete subclasses provided by Foundation for these purposes are NSArchiver, NSUnarchiver, and NSPortCoder. Concrete subclasses of NSCoder are referred to in general as *coder classes*, and instances of these classes as *coder objects* (or simply *coders*). A coder object that can only encode values is referred to as an *encoder object*, and one that can only decode values as a *decoder object*.

NSCoder operates on **id**'s, scalars, C arrays, structures, and strings, and on pointers to these types. It does not handle types whose implementation varies across platforms, such as **union**, **void** *, function pointers, and long chains of pointers. It can also operate on user-defined structures as well as pointers to any of these data types. A coder object stores object type information along with the data, so an object decoded from a stream of bytes is normally of the same class as the object that was originally encoded into the stream. An object can change its class when encoded, however; this is described in the NSCoding protocol specification under ""Making Substitutions During Coding"."

Encoding and Decoding Objects and Data Items

To encode or decode an object or data item, you must first create a coder object, then send it a message defined by NSCoder or by a concrete subclass to actually encode or decode the item. NSCoder itself defines no particular method for creating a coder; this typically varies with the subclass. NSArchiver and NSUnarchiver, for example, use **initForWritingWithMutableData:** and **initForReadingWithData:**. NSPortCoders are created and used by NSConnection objects; you never create one of these yourself.

To encode an object or data item, use any of the **encode...** methods, such as **encodeRootObject:**, **encodeValueOfObjCType:at:**, and so on. This sample code fragment uses the NSArchiver concrete subclass of NSCoder to archive a custom object called **myMapView**:

```
MapView *myMapView;   /* Assume this exists. */
NSMutableData *data
NSArchiver *archiver;
BOOL result;

data = [NSMutableData data];
archiver = [[NSArchiver alloc] initForWritingWithMutableData:data];
[archiver encodeRootObject:myMapView];
result = [data writeToFile:@"/tmp/MapArchive" atomically:YES];
```

NSArchiver also provides a convenience method for archiving directly to a file, rendering the example above as:

```
result = [NSArchiver archiveRootObject:myMapView toFile:@"/tmp/MapArchive"];
```

To decode an object or data item, simply use the **decode...** method corresponding to the original **encode...** method (as given in the individual method descriptions). Matching these is important, as the method originally used determines the format of the encoded data. See the NSCoding protocol specification for an example.

NSCoder's interface is quite general. Concrete subclasses aren't required to properly implement all of NSCoder's methods, and may explicitly restrict themselves to certain types of operations. For example, NSArchiver doesn't implement the **decode...** methods, and NSUnarchiver doesn't implement the **encode...** methods.

When to Retain a Decoded Object

You can decode an object value in two ways. The first is explicitly, using the **decodeObject** method (or any **decode...Object** method). When decoding an object explicitly you must follow the object ownership convention, and retain the object returned if you intend to keep it. Otherwise the object is owned by the coder and will be released when the coder is released.

The second means of decoding an object is implicitly, using the **decodeValueOfObjCType:at:** method or one of its variants, **decodeArrayOfObjCType:count:at:** and **decodeValuesOfObjCTypes:**. These methods fill a value already claimed by the invoker, so you are responsible for releasing decoded object values. This behavior can prove useful for optimizing large decoding operations, as it obviates the need for sending a **retain** message to each decoded object.

Managing Object Graphs

Objects frequently contain pointers to other objects, which may in turn contain pointers to other objects. When analyzed, a group of objects may contain circular references or one object may be referred to by several other objects. In these cases, the objects form an object graph and require special encoding methods to preserve the graph structure. NSCoder declares the following methods to manage object graphs: **encodeRootObject:**, **encodeObject:**, and **encodeConditionalObject:**.

As implemented by a subclass, **encodeRootObject:** should encode the object and any objects to which it refers. It is the responsibility of the overriding **encodeRootObject:** method to keep track of multiple references to objects, thus preserving the structure of any object graphs.

The **encodeConditionalObject:** method allows an object to be excluded from the encoding process. Objects are encoded unconditionally by default, that is, the contents of the object are always encoded. Subclasses can implement **encodeConditionalObject:** to encode an object only if it was encoded previously by a call to a method other than **encodeConditionalObject:**.

NSCoder's implementations of **encodeRootObject:** and **encodeConditionalObject:** simply encode the object unconditionally, whether or not it's already been encoded. A concrete subclass that supports object graphs must override these two methods. See the NSArchiver class specification for more information on managing object graphs.

Creating a Subclass of NSCoder

If you define a subclass of NSCoder, at a minimum your subclass must override the following methods:

encodeValueOfObjCType:at:
decodeValueOfObjCType:at:
encodeDataObject:
decodeDataObject:
versionForClassName:

In addition, your subclass may override other methods to provide specialized handling for certain situations. In particular, you can implement any of the following methods:

(an initialization method)
encodeRootObject:
encodeConditionalObject:
encodeBycopyObject:
encodeByrefObject:
decodeObject
setObjectZone:
objectZone

See the individual method descriptions for more information. See also the NSArchiver class specification for an example of a concrete subclass.

Note that **encodeObject:** and **decodeObject** are not among the basic methods. They're defined abstractly to invoke **encodeValueOfObjCType:at:** or **decodeValueOfObjCType:at:** with an Objective-C type code of "@". Your implementations of the latter two methods must handle this case, invoking the object's **encodeWithCoder:** or **initWithCoder:** method and sending the proper substitution messages (as described in the NSCoding protocol specification) to the object before encoding it and after decoding it.

With objects, the object being coded is fully responsible for coding itself. However,a few classes hand this responsibility back to the coder object, either for performance reasons or because proper support depends

on more information than the object itself has. The two notable classes in Foundation that do this are NSData and NSPort. NSData's low-level nature makes optimization important. For this reason, an NSData object always asks its coder to handle its contents directly using the **encodeDataObject:** and **decodeDataObject:** methods described in this class specification. Similarly, an NSPort object asks its coder to handle it using the **encodePortObject:** and **decodePortObject:** methods (which only NSPortCoder implements). This is because an NSPort represents information kept in the operating system itself, which requires special handling for transmission to another process.

These special cases don't affect users of coder objects, since the redirection is handled by the classes themselves in their NSCoding protocol methods. An implementor of a concrete coder subclass, however, must encode NSData and NSPort objects itself, and take care not to send an **encodeWithCoder:** or **initWithCoder:** message to the NSData or NSPort object. Failure to do so can result in an infinite loop.

Method Types

Encoding data

– encodeArrayOfObjCType:count:at:
– encodeBycopyObject:
– encodeByrefObject:
– encodeBytes:length:
– encodeConditionalObject:
– encodeDataObject:
– encodeObject:
– encodePropertyList:
– encodePoint:
– encodeRect:
– encodeRootObject:
– encodeSize:
– encodeValueOfObjCType:at:
– encodeValuesOfObjCTypes:

Decoding data

– decodeArrayOfObjCType:count:at:
– decodeBytesWithReturnedLength:
– decodeDataObject
– decodeObject
– decodePropertyList
– decodePoint
– decodeRect
– decodeSize
– decodeValueOfObjCType:at:
– decodeValuesOfObjCTypes:

Managing zones

– objectZone
– setObjectZone:

Getting version information

– systemVersion
– versionForClassName:

Instance Methods

decodeArrayOfObjCType:count:at:

– (void)**decodeArrayOfObjCType:**(const char *)*itemType*
count:(unsigned int)*count*
at:(void *)*address*

Decodes an array of *count* items, whose Objective-C type is given by *itemType*. The items are decoded into the buffer beginning at *address*, which must be large enough to contain them all. *itemType* must contain exactly one type code. NSCoder's implementation invokes **decodeValueOfObjCType:at:** to decode the entire array of items. If you use this method to decode an array of Objective-C objects, you are responsible for releasing each object.

This method matches an **encodeArrayOfObjCType:count:at:** message used during encoding.

For information on creating an Objective-C type code suitable for *itemType*, see the Type Encoding section in Chapter 3: Objective-C Extensions of *Object-Oriented Programming and the Objective-C Language*.

See also: – **decodeValuesOfObjCTypes:**

decodeBytesWithReturnedLength:

– (void *)**decodeBytesWithReturnedLength:**(unsigned int *)*numBytes*

Decodes a buffer of data whose types are unspecified. NSCoder's implementation invokes **decodeValueOfObjCType:at:** to decode the data as a series of bytes, which this method then places into a buffer and returns. The buffer's length is returned by reference in *numBytes*. If you need the bytes beyond the scope of the current autorelease pool, you must copy them.

This method matches an **encodeBytes:length:** message used during encoding.

See also: – **encodeArrayOfObjCType:count:at:**

decodeDataObject

– (NSData *)**decodeDataObject**

Must be overridden by subclasses to decode and return an NSData object.

The implementation of your overriding method must match the implementation of your **encodeDataObject:** method. For example, a typical **encodeDataObject:** method encodes the number of bytes of data followed by the bytes themselves. Your override of this method must read the number of bytes, create an NSData object of the appropriate size, and decode the bytes into the new NSData object. Your overriding method should send an **autorelease** message to the new NSData object before returning it.

decodeObject

– (id)**decodeObject**

Decodes an Objective-C object that was previously encoded with any of the **encode...Object:** methods. NSCoder's implementation invokes **decodeValueOfObjCType:** to decode the object data.

Subclasses may need to override this method if they override any of the corresponding **encode...Object:** methods. For example, if an object was encoded conditionally using the **encodeConditionalObject:** method, this method needs to check whether the object had actually been encoded.

See also: – **encodeBycopyObject:**, – **encodeByrefObject:**, – **encodeObject:**

decodePoint

– (NSPoint)**decodePoint**

Decodes and returns an **NSPoint** structure that was previously encoded with **encodePoint:**. Subclasses should not override this method.

decodePropertyList

– (id)**decodePropertyList**

Decodes a property list that was previously encoded with **encodePropertyList:**. Subclasses should not override this method. See the NSPPL class specification for information on property lists.

decodeRect

– (NSRect)**decodeRect**

Decodes and returns an **NSRect** structure that was previously encoded with **encodeRect:**. Subclasses should not override this method.

decodeSize

– (NSSize)**decodeSize**

Decodes and returns an **NSSize** structure that was previously encoded with **encodeSize:**. Subclasses should not override this method.

decodeValueOfObjCType:at:

– (void)**decodeValueOfObjCType:**(const char *)*valueType* **at:**(void *)*data*

Decodes a single value, whose Objective-C type is given by *valueType*. *valueType* must contain exactly one type code and the buffer specified by *data* must be large enough to hold the value corresponding to that type code. For information on creating an Objective-C type code suitable for *valueType*, see the Type Encoding section in Chapter 3: Objective-C Extensions of *Object-Oriented Programming and the Objective-C Language*.

Subclasses must override this method and provide an implementation to decode the value. In your overriding implementation, decode the value into the buffer beginning at *data*. If your overriding method is capable of decoding an Objective-C object, your method must also retain that object. Clients of this method are then responsible for releasing the object.

This method matches an **encodeValueOfObjCType:at:** message used during encoding.

See also: – **decodeArrayOfObjCType:count:at:**, – **decodeValuesOfObjCTypes:**, – **decodeObject**

decodeValuesOfObjCTypes:

– (void)**decodeValuesOfObjCTypes:**(const char *)*valueTypes*, ...

Decodes a series of potentially different Objective-C types. *valueTypes* is a single string containing any number of type codes. The variable arguments to this method consist of one or more pointer arguments, each of which specifies the buffer in which to place a single decoded value. For each type code in *valueTypes*, you must specify a corresponding pointer argument whose buffer is large enough to hold the decoded value. If you use this method to decode Objective-C objects, you are responsible for releasing them.

This method matches an **encodeValuesOfObjCTypes:** message used during encoding.

NSCoder's implementation invokes **decodeValueOfObjCType:at:** to decode individual types. Subclasses that implement the **decodeValueOfObjCType:at:** method do not need to override this method.

For information on creating Objective-C type codes suitable for *valueTypes*, see the description of the **@encode()** compiler directive in *Object-Oriented Programming and the Objective-C Language*.

See also: – **decodeArrayOfObjCType:count:at:**

encodeArrayOfObjCType:count:at:

– (void)**encodeArrayOfObjCType:**(const char *)*itemType*
count:(unsigned int)*count*
at:(const void *)*address*

Encodes an array of *count* items, whose Objective-C type is given by *itemType*. The values are encoded from the buffer beginning at *address*. *itemType* must contain exactly one type code. NSCoder's implementation invokes **encodeValueOfObjCType:at:** to encode the entire array of items. Subclasses that implement the **encodeValueOfObjCType:at:** method do not need to override this method.

This method must be matched by a subsequent **decodeArrayOfObjCType:count:at:** message.

For information on creating an Objective-C type code suitable for *itemType*, see the description of the **@encode()** compiler directive in *Object-Oriented Programming and the Objective-C Language*.

See also: – **encodeValueOfObjCType:at:**, – **encodeValuesOfObjCTypes:**, **– encodeBytes:length:**

encodeBycopyObject:

– (void)**encodeBycopyObject:**(id)*object*

Can be overridden by subclasses to encode *object* so that a copy rather than a proxy is created upon decoding. NSCoder's implementation simply invokes **encodeObject:**.

This method must be matched by a subsequent **decodeObject** message.

See also: – **encodeRootObject:**, – **encodeConditionalObject:**, – **encodeByrefObject:**

encodeByrefObject:

– (void)**encodeByrefObject:**(id)*object*

Can be overridden by subclasses to encode *object* so that a proxy, rather than a copy, is created upon decoding. NSCoder's implementation simply invokes **encodeObject:**.

This method must be matched by a subsequent **decodeObject** message.

See also: – **encodeBycopyObject:**, – **replacementObjectForPortCoder:** (NSObject)

encodeBytes:length:

– (void)**encodeBytes:**(void *)*address* **length:**(unsigned int)*numBytes*

Encodes a buffer of data whose types are unspecified. The buffer to be encoded begins at *address*, and its length in bytes is given by *numBytes*.

This method must be matched by a subsequent **decodeBytesWithReturnedLength:numBytes:** message.

See also: – **encodeArrayOfObjCType:count:at:**

encodeConditionalObject:

– (void)**encodeConditionalObject:**(id)*object*

Can be overridden by subclasses to conditionally encode *object*, preserving common references to that object. In the overriding method, *object* should be encoded only if it's unconditionally encoded elsewhere (with any other **encode...Object:** method).

This method must be matched by a subsequent **decodeObject** message. Upon decoding, if *object* was never encoded unconditionally, **decodeObject** returns **nil** in place of *object*. However, if *object* was encoded unconditionally, all references to *object* must be resolved.

NSCoder's implementation simply invokes **encodeObject:**.

See also: – **encodeRootObject:**, – **encodeObject:**, – **encodeBycopyObject:**, – **encodeConditionalObject:** (NSArchiver)

encodeDataObject:

– (void)**encodeDataObject:**(NSData *)*data*

Must be overridden by subclasses to encode the NSData object *data*. This method must be matched by a subsequent **decodeDataObject** message.

See also: – **encodeObject:**

encodeObject:

– (void)**encodeObject:**(id)*object*

Encodes *object*. NSCoder's implementation simply invokes **encodeValueOfObjCType:at:** to encode the object. Subclasses can override this method to encode a reference to *object* instead of *object* itself. For example, NSArchiver detects duplicate objects and encodes a reference to the original object rather than encode the same object twice.

This method must be matched by a subsequent **decodeObject** message.

See also: – **encodeRootObject:**, – **encodeConditionalObject:**, – **encodeBycopyObject:**

encodePoint:

– (void)**encodePoint:**(NSPoint)*point*

Encodes *point*. NSCoder's implementation invokes **encodeValueOfObjCType:at:** to encode *point*. Subclasses should not need to override this method.

This method must be matched by a subsequent **decodePoint** message.

encodePropertyList:

– (void)**encodePropertyList:**(id)*aPropertyList*

Encodes a property list. (See the NSPPL class specification for information on property lists.) NSCoder's implementation invokes **encodeValueOfObjCType:at:** to encode *aPropertyList*. Subclasses should not need to override this method.

This method must be matched by a subsequent **decodePropertyList** message.

encodeRect:

– (void)**encodeRect:**(NSRect)*rect*

Encodes *rect*. NSCoder's implementation invokes **encodeValueOfObjCType:at:** to encode *rect*. Subclasses should not need to override this method.

This method must be matched by a subsequent **decodeRect** message.

encodeRootObject:

– (void)**encodeRootObject:**(id)*rootObject*

Can be overridden by subclasses to encode an interconnected group of Objective-C objects, starting with *rootObject*. NSCoder's implementation simply invokes **encodeObject:**.

This method must be matched by a subsequent **decodeObject** message.

See also: – **encodeObject:**, – **encodeConditionalObject:**, – **encodeBycopyObject:**, – **encodeRootObject:** (NSArchiver)

encodeSize:

– (void)**encodeSize:**(NSSize)*size*

Encodes *size*. NSCoder's implementation invokes **encodeValueOfObjCType:at:** to encode *size*. Subclasses should not need to override this method.

This method must be matched by a subsequent **decodeSize** message.

encodeValueOfObjCType:at:

– (void)**encodeValueOfObjCType:**(const char *)*valueType* **at:**(const void *)*address*

Must be overridden by subclasses to encode a single value residing at *address*, whose Objective-C type is given by *valueType*. *valueType* must contain exactly one type code.

This method must be matched by a subsequent **decodeValueOfObjCType:at:** message.

For information on creating an Objective-C type code suitable for *valueType*, see the description of the **@encode()** compiler directive in *Object-Oriented Programming and the Objective-C Language*.

See also: – **encodeArrayOfObjCType:count:at:**, – **encodeValuesOfObjCTypes:**

encodeValuesOfObjCTypes:

– (void)**encodeValuesOfObjCTypes:**(const char *)*valueTypes*, ...

Encodes a series of values of potentially differing Objective-C types. *valueTypes* is a single string containing any number of type codes. The variable arguments to this method consist of one or more pointer arguments, each of which specifies a buffer containing the value to be encoded. For each type code in *valueTypes*, you must specify a corresponding pointer argument.

This method must be matched by a subsequent **decodeValuesOfObjCTypes:** message.

NSCoder's implementation invokes **encodeValueOfObjCType:at:** to encode individual types. Subclasses that implement the **encodeValueOfObjCType:at:** method do not need to override this method. However, subclasses that provide a more efficient approach for encoding a series of values may override this method to implement that approach.

For information on creating Objective-C type codes suitable for *valueTypes*, see the description of the **@encode()** compiler directive in *Object-Oriented Programming and the Objective-C Language*.

See also: – **encodeArrayOfObjCType:count:at:**, – **encodeValueOfObjCType:at:**

objectZone

– (NSZone *)**objectZone**

Returns the memory zone used to allocate decoded objects. NSCoder's implementation simply returns the default memory zone, as given by **NSDefaultMallocZone()**.

Subclasses must override this method and the **setObjectZone:** method to allow objects to be decoded into a zone other than the default zone. In its overriding implementation of this method, your subclass should return the current memory zone (if one has been set) or the default zone (if no other zone has been set).

See also: – **setObjectZone:**

setObjectZone:

– (void)**setObjectZone:**(NSZone *)*zone*

Can be overridden by subclasses to set the memory zone used to allocate decoded objects. NSCoder's implementation of this method does nothing.

Subclasses must override this method and the **objectZone** method to allow objects to be decoded into a zone other than the default zone. In its overriding implementation of this method, your subclass should store a reference to the current memory zone.

See also: – **objectZone**

systemVersion

– (unsigned int)**systemVersion**

During encoding, this method should return the system version currently in effect. During decoding, this method should return the version that was in effect when the data was encoded.

By default, this method returns the current system version, which is appropriate for encoding but not for decoding. Subclasses that implement decoding must override this method to return the system version of the data being decoded.

versionForClassName:

– (unsigned int)**versionForClassName:**(NSString *)*className*

Must be overridden by subclasses to return the version in effect for the class named *className* when it was encoded. Returns NSNotFound if no class named

NSConditionLock

Inherits From:	NSObject
Conforms To:	NSLocking NSObject (NSObject)
Declared In:	Foundation/NSLock.h

Class Description

The NSConditionLock class defines objects whose locks can be associated with specific, user-defined conditions. Using an NSConditionLock object, you can ensure that a thread can acquire a lock only if a certain condition is met. Once it has acquired the lock and executed the critical section of code, the thread can relinquish the lock and set the associated condition to something new. The conditions themselves are arbitrary: You define them as needed for your application.

Typically, you use an NSConditionLock object when threads in your application need to execute in a particular order, such as when one thread produces data that another consumes. While the producer is executing, the consumer sleeps waiting to acquire a lock that's conditional upon the producer's completion of its operation. An application can have multiple NSConditionLock objects, each protecting different sections of code. It's safest to create all of the locks before the application becomes multi-threaded, to avoid race conditions. If you want to create additional locks after the application becomes multi-threaded, you should create the new lock inside a criticial code section that is itself protected by an existing lock.

The locking and unlocking methods that NSConditionLock objects respond to can be used in any combination. For example, you can pair a **lock** message with **unlockWithCondition:**, or a **lockWhenCondition:** message with **unlock**.

The following example shows how the producer-consumer problem might be handled using condition locks. Imagine that an application contains a queue of data. A producer thread adds data to the queue, and consumer threads extract data from the queue.

The producer need not wait for a condition, but must wait for the lock to be made available so it can safely add data to the queue. For example, a producer could use a lock this way:

```
id condLock = [[NSConditionLock alloc] initWithCondition:NO_DATA];

[condLock lock];
/* Add data to the queue. */
[condLock unlockWithCondition:HAS_DATA];
```

Note that in acquiring the lock, the producer sets its condition to the user-defined value NO_DATA. After adding data to the queue, the producer relinquishes the lock, setting its condition to HAS_DATA.

A consumer thread waits until there's data available and all other threads are out of locked critical sections. In the following code, the consumer sleeps until there is data in the queue and a lock can be acquired:

```
[condLock lockWhenCondition:HAS_DATA];
/* Remove data from the queue. */
[condLock unlockWithCondition:(isEmpty ? NO_DATA : HAS_DATA)];
```

The consumer removes some data from the queue and then relinquishes the lock, setting its value to NO_DATA or HAS_DATA, depending on whether the queue is now empty.

The NSConditionLock, NSLock, and NSRecursiveLock classes all adopt the NSLocking protocol with various features and performance characteristics; see the other class descriptions for more information.

Adopted Protocols

NSLocking

– lock
– unlock

Method Types

Initializing an NSConditionLock

– initWithCondition:

Returning the condition

– condition

Acquiring and releasing a lock

– lockBeforeDate:
– lockWhenCondition:
– lockWhenCondition:beforeDate:
– tryLock
– tryLockWhenCondition:
– unlockWithCondition:

Instance Methods

condition

– (int)**condition**

Returns the condition that's associated with the receiver. If no condition has been set, returns 0.

initWithCondition:

– (id)**initWithCondition:**(int)*condition*

Initializes a newly allocated NSConditionLock and sets its condition to *condition*. The value of the *condition* argument is user-defined; see the class description for more information. Returns **self**.

lockBeforeDate:

– (BOOL)**lockBeforeDate:**(NSDate *)*limit*

Attempts to acquire a lock before the date represented by *limit*. The thread is blocked until the receiver acquires the lock or *limit* is reached. Returns YES if the lock is acquired within the time limit. Returns NO if the time limit expires before a lock can be acquired.

The condition associated with the receiver isn't taken into account in this operation.

See also: – **lockWhenCondition:beforeDate:**

lockWhenCondition:

– (void)**lockWhenCondition:**(int)*condition*

Attempts to acquire a lock. The receiver's condition must be equal to *condition* before the locking operation will succeed. This method blocks the thread's execution until the lock can be acquired.

See also: – **lockWhenCondition:beforeDate:**, – **unlockWithCondition:**

lockWhenCondition:beforeDate:

– (BOOL)**lockWhenCondition:**(int)*condition* **beforeDate:**(NSDate *)*limit*

Attempts to acquire a lock before the date represented by *limit*. The receiver's condition must be equal to *condition* before the locking operation will succeed. Returns YES if the lock is acquired within this time limit. Returns NO if the time limit expires before a lock can be acquired. This method blocks the thread's execution until the lock can be acquired or *limit* is reached.

See also: – **lockBeforeDate:**, – **lockWhenCondition:**

tryLock

– (BOOL)**tryLock**

Attempts to acquire a lock without regard to the receiver's condition. Returns immediately with a value of YES if successful and NO otherwise.

See also: – **tryLockWhenCondition:**

tryLockWhenCondition:

– (BOOL)**tryLockWhenCondition:**(int)*condition*

Attempts to acquire a lock if *condition* is true. As part of its implementation, this method invokes *lockWhenCondition:beforeDate:*. Returns immediately, with a value of YES if successful and NO otherwise.

See also: – **tryLock**

unlockWithCondition:

– (void)**unlockWithCondition:**(int)*condition*

Relinquishes the lock and sets the receiver's condition to *condition*.

See also: – **lockWhenCondition:**

NSConnection

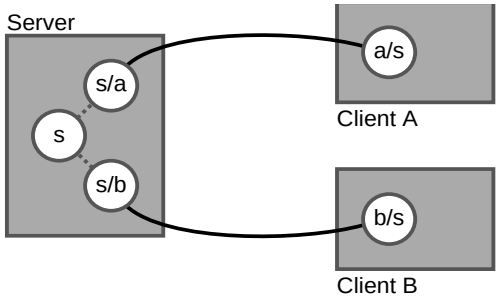
Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSConnection.h

Class Description

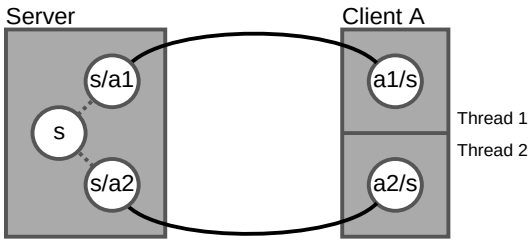
NSConnection objects manage communication between objects in different threads or tasks, on a single host or over the network. They form the backbone of the distributed objects mechanism, and normally operate in the background. You use NSConnection API explicitly when making an object available to other applications, when accessing such a vended object, and when altering default communication parameters; the rest of the time you simply interact with the distributed objects themselves.



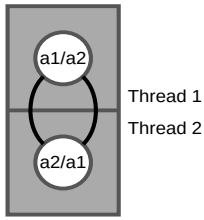
NSConnection objects work in pairs, one in each communicating application or thread. A server application has an NSConnection for every client application connected to it, as shown above (the NSConnection labeled **s** is used to form new connections, as described under “Vending an Object” and “Getting a Vended Object”). The circles represent NSConnection objects, and the labels indicate the application itself and the application it’s connected to. For example, in **s/a** the **s** stands for the server and the **a** stands for client A. If a link is formed between clients A and B in this example, two new NSConnection objects get created: **a/b** and **b/a**.

Note: For Apple’s implementation the small letters represent NSPort objects. The letter on the left indicates the *receive port*, which messages arrive on; the letter on the right indicates the *send port*, which outgoing messages leave through. Similar mechanisms exist for other OpenStep-compliant systems, but any API exporting them isn’t part of the OpenStep specification. Subclasses of NSConnection that use them (which they typically must do) are therefore not portable among OpenStep implementations.

Under normal circumstances, all distributed objects passed between applications are tied through one pair of NSConnection objects. NSConnection objects can't be shared by separate threads, though, so for multithreaded applications a separate NSConnection must be created for each thread. This is shown in here:



Finally, an application can use distributed objects between its own threads to make sending messages thread-safe (see the following figure). This is useful for coordinating work with the Application Kit, for example.



Vending an Object

To make an object available to other applications, set it up as the root object of an NSConnection and register the NSConnection by name on the network. This code fragment vends **serverObject**:

```
id serverObject;    /* Assume this exists. */
NSConnection *theConnection;

theConnection = [NSConnection defaultConnection];
[theConnection setRootObject:serverObject];
if ([theConnection registerName:@"server"] == NO) {
    /* Handle error. */
}
```

This fragment takes advantage of the fact that every thread has a default NSConnection object, which can be set up as a server. An NSConnection can vend only one object, so the default NSConnection might not be available. In this case, you can create additional NSConnections to vend objects with the usual **alloc** and **init** methods.

An NSConnection set up this way is called a *named NSConnection*. A named NSConnection rarely has a channel to any other NSConnection (in the illustrations above the named NSConnection is the circle labeled

s). When a client contacts the server, a new pair of NSConnection objects is created specifically to handle communication between the two. The following sections describe this in more detail.

An NSConnection adds itself to the current NSRunLoop when it's initialized. In an application based on the Application Kit, the NSRunLoop is already running, so there's nothing more to do to vend an object. In an application that doesn't use the NSApplication object, you have to start the NSRunLoop explicitly to capture incoming connection requests and messages. This is usually as simple as getting the current thread's NSRunLoop and sending it a **run** message:

```
[[NSRunLoop currentRunLoop] run];
```

See "Configuring an NSConnection" and the NSRunLoop class description for more information on setting NSConnections up to handle requests.

```
NSConnection *theConnection;
id theProxy;

theConnection = [NSConnection connectionWithRegisteredName:@"server"
        host:@"*"];
theProxy = [[theConnection rootProxy] retain];
[theProxy setProtocolForProxy:@protocol(ServerProtocol)];
```

Getting a Vended Object

An application gets a vended object by creating a *proxy*, or a stand-in, for that object in its own address space. The proxy forwards messages sent to it through its NSConnection back to the vended object. An application can get a proxy for a vended object in two ways. First, the **rootProxyForConnectionWithRegisteredName:host:** class method returns the proxy directly:

```
id theProxy;

theProxy = [[NSConnection
        rootProxyForConnectionWithRegisteredName:@"server" host:@"*"]
        retain];
[theProxy setProtocolForProxy:@protocol(ServerProtocol)];
```

This message returns a proxy to the root object of the NSConnection named "server". The host name of "*" indicates that any host on the local subnet with such an NSConnection can be used; you can specify a specific host name to restrict the server to an identified host. If the host name is **nil** or is empty then only the local host is searched for a registered NSConnection.

The invocation of **setProtocolForProxy:** informs the distributed objects system of the set of messages that **theProxy** responds to. Normally, the first time a particular selector is forwarded by a proxy the NSConnection object must confirm the argument and return types with the real object. This can add significant overhead to distributed messages. Setting a protocol records this information so that no confirmation is needed for the messages in the protocol, and only the message forwarding costs are incurred.

Another way to get a proxy is to get an NSConnection to the server and then ask for the proxy of its root object:

```
NSConnection *theConnection;
id theProxy;

theConnection = [NSConnection connectionWithRegisteredName:@"server"
        host:@"*"];
theProxy = [[theConnection rootProxy] retain];
[theProxy setProtocolForProxy:@protocol(ServerProtocol)];
```

This is useful if you need to interact with the NSConnection as well as the proxy (note, though, that **connection** isn't retained in this example).

A named NSConnection spawns a child NSConnection to handle communication between two applications (**s** spawning **s/b** and **s/a** in the first figure). Though the child NSConnection doesn't have a name, it shares the root object and other configuration attributes of its parent (but not the delegate). You shouldn't register a child NSConnection with a name or change its root object, but you can change its other attributes, as described in the next section.

Forming Connections Between Threads

You can use distributed objects between threads in a single application for thread-safe use of shared objects, such as those in the Application Kit. The Application Kit objects run in the main thread, while special calculations or other computations run in other threads. You can set up distributed objects between threads by registering them under names, but doing so exports the object over the entire network, making them available to other applications. This may not be desirable.

Instead of registering your distributed objects, you can manually set up NSConnections that are linked to each other with NSPorts using the **connectionWithReceivePort:sendPort:** message. In the main (Application Kit) thread, you create an NSConnection with two NSPorts, then switch them to create another NSConnection in a separate thread. Suppose you have an application that uses a Calculator object to perform intensive calculations. It might make sense to have this object run in a separate thread and communicate through an NSConnection. In the following example, the initial work of setting up a Calculator object is done in the NSApplication delegate's **applicationDidFinishLaunching:** method:

```
- (void)applicationDidFinishLaunching:(NSNotification *)note
{
    NSPort *port1;
    NSPort *port2;
    NSArray *portArray;

    port1 = [NSPort port];
    port2 = [NSPort port];

    kitConnection = [[NSConnection alloc] initWithReceivePort:port1 sendPort:port2];
    [kitConnection setRootObject:self];

    /* Ports switched here. */
    portArray = [NSArray arrayWithObjects:port2, port1, nil];

    [NSThread detachNewThreadSelector:@selector(connectWithPorts:)
           toTarget:[Calculator class] withObject:portArray];

    return;
}
```

The delegate creates **kitConnection**, which is an instance variable, and sets itself up as the root object so that the Calculator object can find it. To set up the server thread, the delegate packages the NSPorts in an NSArray (note that they're in reverse order) and creates a Calculator object in a separate thread by invoking **detachNewThreadSelector:toTarget:withObject:**. This message starts the new thread by sending **connectWithPorts:** to the Calculator class, which creates the Calculator object that the application uses as an internal server and connects that object to the application delegate:

```
+ (void)connectWithPorts:(NSArray *)portArray
{
    NSAutoreleasePool *pool;
    NSConnection *serverConnection;
    Calculator *serverObject;

    pool = [[NSAutoreleasePool alloc] init];

    serverConnection = [NSConnection
           connectionWithReceivePort:[portArray objectAtIndex:0]
           sendPort:[portArray objectAtIndex:1]];

    serverObject = [[self alloc] init];
    [(id)[serverConnection rootProxy] setServer:serverObject];
    [serverObject release];

    [[NSRunLoop currentRunLoop] run];
    [pool release];
    [NSThread exit];

    return;
}
```

connectWithPorts: is responsible for creating the server object and connecting it to the client thread. This method first creates an NSAutoreleasePool to prevent objects from being leaked if the NSThread ever exits. It then creates **serverConnection** using the two NSPorts from **kitConnection**, and proceeds to set up the server object. Once this is done, it passes the server object back to the client with a **setServer:** message. Finally, it starts the NSRunLoop for its thread. If the NSRunLoop ever stops, the NSAutoreleasePool is cleaned up and the NSThread is made to exit.

The application delegate's **setServer:** message simply sets the protocol and stores the server object:

```
- (void)setServer:(id)anObject
{
    [anObject setProtocolForProxy:@protocol(CalculatorMethods)];
    calculator = (id <CalculatorMethods>)[anObject retain];
    return;
}
```

Because *anObject* arrived through the NSConnections set up between threads, it's really a proxy to the **serverObject** created in **connectWithPorts:** above.

Configuring an NSConnection

You can control some factors of distributed objects communication by configuring NSConnection objects. You can set timeouts to limit the amount of time an NSConnection will wait on a remote message, set the mode it awaits requests and responses on, and control how an NSConnection manages multiple remote

messages. In addition to these parameter settings, you can change an NSConnection's registered name or root object for dynamic alteration of your distributed application.

An NSConnection uses two kinds of timeouts, one for outgoing messages and one for replies. An outgoing network message may take some time to send. Once it goes out, there's usually a delay before any return value arrives. If either of these operations exceeds its timeout, the NSConnection raises an NSPortTimeoutException. You can set the values for these timeouts with the **setRequestTimeout:** and **setReplyTimeout:** messages, respectively. By default these timeouts are set to the maximum possible value.

NSConnections that vend objects await new connection requests in NSDefaultRunLoopMode (as defined by the NSRunLoop class). When an NSConnection sends a remote message out, it awaits the return value in NSConnectionReplyMode. You can't change this mode, but you can use it to set up NSTimers or other input mechanisms that need to be processed while awaiting replies to remote messages. Use **addRequestMode:** to add input mechanisms for this mode.

Normally an NSConnection forwards remote messages to their intended recipients as it receives them. If your application returns to the run loop or uses distributed objects either directly or indirectly, it can receive a remote message while it's already busy processing another. Suppose a server is processing a remote message and sends a message to another application through distributed objects. If another application sends a message to the server, its NSConnection immediately forwards it to the intended recipient, even though the server is also awaiting a reply on the outgoing message. This behavior can cause problems if a remote message causes a lengthy change in the server application's state that renders it inconsistent for a time: Other remote messages may interfere with this state, either getting incorrect results or corrupting the state of the server application. You can turn this behavior off with the **setIndependentConversationQueueing:** method, so that only one remote message is allowed to be in effect at any time within the NSConnection's thread. When independent conversation queueing is turned on, the NSConnection forwards incoming remote messages only when no other remote messages are being handled in its thread. This only affects messages between objects, not requests for new connections; new connections can be formed at any time.

Warning: Because independent conversation queueing causes remote messages to block where they normally don't, it can cause deadlock to occur between applications. Use this method only when you know the nature of the interaction between two applications. Specifically, note that multiple callbacks between the client and server aren't possible with independent conversation queueing.

One other way to configure a named NSConnection is to change its name or root object. This effectively changes the object that applications get using the techniques described in "Getting a Vended Object", but doesn't change the proxies that other applications have already received. You might use this technique to field-upgrade a distributed application with an improved server object class. For example, to install a new server process have the old one change its name, perhaps from "Analysis Server" to "Old Analysis Server". This hides it from clients attempting to establish new connections, but allows its root object to serve existing connections (when those connections close, the old server process exits). In the meantime, launch the new server which claims the name "Analysis Server" so that new requests for analyses contact the updated object.

The Delegate

An NSConnection can be assigned a delegate, which has two possible responsibilities: approving the formation of new connections, and authenticating messages that pass between NSConnections.

When a named NSConnection is contacted by a client and forms a child NSConnection to communicate with that client, it sends **connection:shouldMakeNewConnection:** to its delegate first to approve the new connection. If the delegate returns NO the connection is refused. This method is useful for limiting the load on a server. It's also useful for setting the delegate of a child NSConnection (since delegates aren't shared automatically between parent and child).

Portable Distributed Objects adds message authentication to NSConnection's OpenStep API. Delegates in different applications can cooperate to validate the messages passing between them by implementing **authenticationDataForComponents:** and **authenticateComponents:withData:**. The first method requests an authentication stamp for an outgoing message, which is used by the second method to check the validity of the message when it's received.

authenticationDataForComponents: provides the packaged components for an outgoing network message in the form of NSData and NSPort objects. The delegate should use only the NSData objects to create the authentication stamp, by hashing the data, calculating a checksum, or some other method. The stamp should be small enough not to adversely affect network performance. The delegate in the receiving application receives an **authenticateComponents:withData:** message to confirm the message, and should recalculate the stamp for the components and compare it with the stamp provided. If it returns YES the message is forwarded; if it returns NO, an NSFailedAuthenticationException is raised and a message is logged to the console.

Handling NSConnection Errors

NSConnections make use of network resources that can become unavailable at any time. When a server machine loses power, for example, the objects on that machine that have been vended to other applications simply cease to exist. In such a case, the NSConnections handling those objects invalidate themselves and post an NSConnectionDidDieNotification to any observers. This notification allows objects to clean up their state as much as possible in the face of an error.

To register for the notification, add an observer to the default NSNotificationCenter:

```
[[NSNotificationCenter defaultCenter] addObserver:proxyUser
        selector:@selector(connectionDidDie:)
        name:NSConnectionDidDieNotification
        object:serverConnection];
```

The fragment above registers the **proxyUser** object to receive a **connectionDidDie:** message when the **serverConnection** object in the application posts an NSConnectionDidDieNotification. This allows it to release any proxies it holds and to handle the error as gracefully as possible. See the NSNotification and NSNotificationCenter class specifications for more information on notifications.

A less serious error is a timeout on a remote message. This can happen for an outgoing message, meaning the message was never sent to its recipient, or for a reply to a message successfully sent, meaning either that the message failed to reach its recipient or that the reply couldn't be delivered back to the original sender. An application can put an exception handler in place for critical messages, and if a timeout exception is raised it can send the message again, check that the server is still running or take whatever other action it needs to recover.

Method Types

Getting the default instance

+ defaultConnection

Creating instances

+ connectionWithReceivePort:sendPort:
– initWithReceivePort:sendPort:

Running the connection in a new thread

– runInNewThread

Vending an object

– registerName:
– setRootObject:
– rootObject

Getting a remote object

+ connectionWithRegisteredName:host:
– rootProxy
+ rootProxyForConnectionWithRegisteredName:host:

Getting all NSConnections

+ allConnections

Configuring instances

– setRequestTimeout:
– requestTimeout
– setReplyTimeout:
– replyTimeout
– setIndependentConversationQueueing:
– independentConversationQueueing
– addRequestMode:
– removeRequestMode:
– requestModes
– invalidate

Getting ports

– receivePort
– sendPort

Getting statistics

– statistics

Setting the delegate

– setDelegate:
– delegate

Class Methods

allConnections

+ (NSArray *)**allConnections**

Returns all valid NSConnections in the process.

See also: – **isValid**

connectionWithReceivePort:sendPort:

+ (NSConnection *)**connectionWithReceivePort:**(NSPort *)*receivePort* **sendPort:**(NSPort *)*sendPort*

Returns an NSConnection that communicates using *receivePort* and *sendPort*. See **initWithReceivePort: sendPort:** for more information.

See also: + **defaultConnection**

connectionWithRegisteredName:host:

+ (NSConnection *)**connectionWithRegisteredName:**(NSString *)*name* **host:**(NSString *)*hostName*

Returns the NSConnection whose send port links it to the NSConnection registered under *name* on the host named *hostName*. Returns **nil** if no NSConnection can be found for *name* and *hostName*. The returned NSConnection is a child of the default NSConnection for the current thread (that is, it shares the default NSConnection's receive port).

If *hostName* is **nil** or empty then only the local host is searched for the named NSConnection. If *hostName* is "*" then all hosts on the local subnet are queried for an NSConnection registered under *name;* where there are duplicates the connection is made with an arbitrary host, which is then used for every subsequent request from the local host.

To get the object vended by the NSConnection, use the **rootProxy** instance method. The **rootProxyForConnectionWithRegisteredName:host:** class method immediately returns this object.

See also: + **defaultConnection**

currentConversation

+ (id)**currentConversation**

<< Description forthcoming. >>

defaultConnection

+ (NSConnection *)**defaultConnection**

Returns the default NSConnection for the current thread, creating it if necessary. The default NSConnection uses a single NSPort for both receiving and sending, and is useful only for vending an object; use the **setRootObject:** and **registerName:** methods to do this.

rootProxyForConnectionWithRegisteredName:host:

+ (NSDistantObject *)**rootProxyForConnectionWithRegisteredName:**(NSString *)*name* **host:** (NSString *)*hostName*

Returns a proxy for the root object of the NSConnection registered under *name* on the host named *hostName*, or **nil** if that NSConnection has no root object set. Also returns **nil** if no NSConnection can be found for *name* and *hostName*. The NSConnection of the returned proxy is a child of the default NSConnection for the current thread (that is, it shares the default NSConnection's receive port).

If *hostName* is **nil** or empty then only the local host is searched for the named NSConnection. If *hostName* is "*" then all hosts on the local subnet are queried for an NSConnection registered under *name;* where there are duplicates the connection is made with an arbitrary host, which is then used for every subsequent request from the local host.

This method invokes **connectionWithRegisteredName:host:** and sends the resulting NSConnection object a **rootProxy** message.

See also: – **setRootObject:**

Instance Methods

addRequestMode:

– (void)**addRequestMode:**(NSString *)*mode*

Adds *mode* to the set of run loop input modes that the NSConnection uses for connection requests. The default input mode is NSDefaultRunLoopMode. See the NSRunLoop class specification for more information on input modes.

See also: – **addPort:forMode:** (NSRunLoop)

addRunLoop:

– (void)**addRunLoop:**(NSRunLoop *)*runloop*

Adds runloop to the list of run loops the NSConnection monitors and will respond to requests from. This is invoked automatically when a request comes in from a new run loop if **enableMultipleThreads** has been set.

See also: – **enableMultipleThreads**

delegate

– (id)**delegate**

Returns the NSConnection's delegate.

See also: – **setDelegate:**

enableMultipleThreads

– (void)**enableMultipleThreads**

Configures the NSConnection to automatically allow requests from multiple threads to the remote object, without requiring the threads to each maintain their own connection.

independentConversationQueueing

– (BOOL)**independentConversationQueueing**

Returns YES if the NSConnection handles remote messages atomically, NO otherwise. See "Configuring an NSConnection" in the class description for more information on independent conversation queueing.

See also: – **setIndependentConversationQueueing:**

initWithReceivePort:sendPort:

– (id)**initWithReceivePort:**(NSPort *)*receivePort* **sendPort:**(NSPort *)*sendPort*

Initializes a newly created NSConnection with *receivePort* and *sendPort*. The new NSConnection adds *receivePort* to the current NSRunLoop with NSDefaultRunLoopMode as the mode. If the application doesn't use an NSApplication object to handle events, it needs to run the NSRunLoop with one of its various **run...** messages. Returns **self**.

This method posts an NSConnectionDidInitializeNotification once the connection is initialized.

receivePort and *sendPort* affect initialization as follows:

- If an NSConnection with the same ports already exists, releases the receiver, retains the existing NSConnection, and returns it.
- If an NSConnection exists that uses the same ports, but switched in role, then the new NSConnection communicates with it. Messages sent to a proxy held by either NSConnection are forwarded through the other NSConnection. This rule applies both within and across address spaces.

  This behavior is useful for setting up distributed objects connections between threads within an application. See "Forming Connections Between Threads" in the class description for more information.
- If *receivePort* is **nil**, deallocates the receiver and returns **nil**.
- If *sendPort* is **nil** or if both ports are the same, the NSConnection uses *receivePort* for both sending and receiving, and is useful only for vending an object. Use the **registerName:** and **setRootObject:** instance methods to vend an object.
- If an NSConnection exists that uses *receivePort* as both of its ports, it's treated as the parent of the new NSConnection, and its root object and all of its configuration settings are applied to the new NSConnection. You should neither register a name for nor set the root object of the new NSConnection. See "Configuring an NSConnection" in the class description for more information.
- If *receivePort* and *sendPort* are different and neither is shared with another NSConnection, the receiver can be used to vend an object as well as to communicate with other NSConnections. However, it has no other NSConnection to communicate with until one is set up.
- *receivePort* can't be shared by NSConnections in different threads.

This method is the designated initializer for the NSConnection class. Because it isn't part of the OpenStep specification, subclasses of NSConnection aren't portable among different OpenStep implementations.

See also: + **defaultConnection**

invalidate

– (void)**invalidate**

Invalidates (but doesn't release) the receiver. After withdrawing the ports that it has registered with the current run loop, **invalidate** posts an NSConnectionDidDieNotification and then invalidates all remote objects and exported local proxies.

See also: – **isValid**, – **removePort:forMode:** (NSRunLoop), – **requestModes**

isValid

– (BOOL)**isValid**

Returns NO if the NSConnection is known to be invalid, YES otherwise. An NSConnection becomes invalid when either of its ports becomes invalid, but only notes that it has become invalid when it tries to send or receive a message. When this happens it posts an NSConnectionDidDieNotification to the default notification center.

See also: – **invalidate**, – **isValid** (NSPort)

localObjects

– (NSArray *)**localObjects**

<< Description forthcoming. >>

multipleThreadsEnabled

– (BOOL)**multipleThreadsEnabled**

Returns YES if **enableMultipleThreads** has ever been invoked.

receivePort

– (NSPort *)**receivePort**

Returns the NSPort that the NSConnection receives incoming network messages on. You can inspect this object for debugging purposes or use it to create another NSConnection, but shouldn't use it to send or receive messages explicitly. Don't set the delegate of the receive port; it already has a delegate established by the NSConnection.

See also: – **sendPort**, – **initWithReceivePort:sendPort:**

registerName:

– (BOOL)**registerName:**(NSString *)*name*

Registers the NSConnection under *name* on the local host, returning YES if successful, NO if not (for example, if another NSConnection on the same host is already registered under *name*). Other NSConnections can then contact it using the **connectionWithRegisteredName:host:** and **rootProxyForConnectionWithRegisteredName:host:** class methods.

If the NSConnection was already registered under a name and this method returns NO, the old name remains in effect. If this method is successful, it also unregisters the old name.

To unregister an NSConnection, simply invoke **registerName:** and supply **nil** as the connection name.

See also: – **setRootObject:**

remoteObjects

– (NSArray *)**remoteObjects**

<< Description forthcoming. >>

removeRequestMode:

– (void)**removeRequestMode:**(NSString *)*mode*

Removes *mode* from the set of run loop input modes that the NSConnection uses for connection requests.

See also: – **removePort:forMode:** (NSRunLoop)

removeRunLoop:

– (void)**removeRunLoop:**(NSRunLoop *)*runloop*

<< Description forthcoming. >>

replyTimeout

– (NSTimeInterval)**replyTimeout**

Returns the timeout interval for replies to outgoing remote messages. If a non-**oneway** remote message is sent and no reply is received by the timeout, an NSPortTimeoutException is raised.

See also: – **requestTimeout**, – **setReplyTimeout:**

requestModes

– (NSArray *)**requestModes**

Returns the set of request modes (as an array of NSStrings) that the NSConnection's receive port is registered for with its NSRunLoop.

See also: – **addRequestMode:**, – **addPort:forMode:** (NSRunLoop), – **removeRequestMode:**

requestTimeout

– (NSTimeInterval)**requestTimeout**

Returns the timeout interval for outgoing remote messages. If a remote message can't be sent before the timeout, an NSPortTimeoutException is raised.

See also: – **replyTimeout**, – **setRequestTimeout:**

rootObject

– (id)**rootObject**

Returns the object that the NSConnection (or its parent) makes available to other applications or threads, or **nil** if there is no root object. To get a proxy to this object in another application or thread, invoke the **rootProxyForConnectionWithRegisteredName:host:** class method with the appropriate arguments.

See also: – **rootProxy**, – **setRootObject:**

rootProxy

– (NSDistantObject *)**rootProxy**

Returns the proxy for the root object of NSConnection's peer in another application or thread. The proxy returned can change between invocations if the peer NSConnection's root object is changed.

Note: If the NSConnection uses separate send and receive ports and has no peer, when you invoke rootProxy it will block for the duration of the reply timeout interval, waiting for a reply.

See also: – **rootObject**

runInNewThread

– (void)**runInNewThread**

Creates and starts a new NSThread and then runs the receiving connection in the new thread. If the newly-created thread is the first to be detached from the current thread, this method posts the notification NSBecomingMultiThreaded with nil to the default notification center.

sendPort

– (NSPort *)**sendPort**

Returns the NSPort that the NSConnection sends outgoing network messages through. You can inspect this object for debugging purposes or use it to create another NSConnection, but shouldn't use it to send or receive messages explicitly. Don't set the delegate of the send port; it already has a delegate established by the NSConnection.

See also: – **receivePort**, – **initWithReceivePort:sendPort:**

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Sets the NSConnection's delegate to *anObject*. Doesn't retain *anObject*.

setIndependentConversationQueueing:

– (void)**setIndependentConversationQueueing:**(BOOL)*flag*

Sets whether the NSConnection handles remote messages atomically. The default is NO: An NSConnection normally forwards remote message to the intended recipients as they come in. See "Configuring an NSConnection" in the class description for more information.

See also: – **independentConversationQueueing**

setReplyTimeout:

– (void)**setReplyTimeout:**(NSTimeInterval)*seconds*

Sets the timeout interval for replies to outgoing remote messages to *seconds*. If a non-**oneway** remote message is sent and no reply is forthcoming by the timeout, an NSPortTimeoutException is raised. The default timeout is the maximum possible value.

See also: – **setRequestTimeout:**, – **replyTimeout**

setRequestTimeout:

– (void)**setRequestTimeout:**(NSTimeInterval)*seconds*

Sets the timeout interval for outgoing remote messages to *seconds*. If a remote message can't be sent before the timeout, an NSPortTimeoutException is raised. The default timeout is the maximum possible value.

See also: – **setReplyTimeout:**, – **requestTimeout**

setRootObject:

– (void)**setRootObject:**(id)*anObject*

Sets the object that the NSConnection makes available to other applications or threads to *anObject*. This only affects new connection requests and **rootProxy** messages to established NSConnections; application that have proxies to the old root object can still send messages through it.

See also: – **rootObject**

statistics

– (NSDictionary *)**statistics**

Returns an NSDictionary containing various statistics for the NSConnection, such as the number of vended objects, the number of requests and replies, and so on. The statistics dictionary should be used only for debugging purposes; see the release notes for more information on its contents.

Methods Implemented By the Delegate

authenticateComponents:withData:

– (BOOL)**authenticateComponents:**(NSArray *)*components* **withData:**
(NSData *)*authenticationData*

Returns YES if the *authenticationData* provided is valid for *components*, NO otherwise. *components* contains NSData and NSPort objects belonging to an NSPortMessage object. See the NSPortMessage class specification for more information. *authenticationData* should have been created by the delegate of the peer NSConnection with **authenticationDataForComponents:**.

Use this message for validation of incoming messages. An NSConnection raises an NSFailedAuthenticationException on receipt of a remote message that the delegate doesn't authenticate.

authenticationDataForComponents:

– (NSData *)**authenticationDataForComponents:**(NSArray *)*components*

Returns an NSData object to be used as a authentication stamp for an outgoing message. *components* contains the elements of a network message, in the form of NSPort and NSData objects. The delegate should use only the NSData elements to create the authentication stamp. See the NSPortMessage class specification for more information on the components.

If **authenticationDataForComponents:** returns **nil**, an NSGenericException will be raised. If the delegate determines that the message shouldn't be authenticated, it should return an empty NSData object (you can generate an empty NSData object with **[NSData data]**). The delegate on the other side of the connection must then be prepared to accept an empty NSData as the second parameter to **authenticateComponents: withData:** and to handle the situation appropriately.

components will be validated on receipt by the delegate of the peer NSConnection with **authenticateComponents:withData:**.

connection:shouldMakeNewConnection:

– (BOOL)**connection:**(NSConnection *)*parentConnection* **shouldMakeNewConnection:** (NSConnection *)*newConnnection*

Returns YES if *parentConnection* should allow *newConnnection* to be created and set up, NO if *parentConnection* should refuse and immediately release *newConnection*. Use this method to limit the amount of NSConnections created in your application or to change the parameters of child NSConnections.

makeNewConnection:sender:

– (BOOL)**makeNewConnection:**(NSConnection *)*newConnection* **sender:**(NSConnection *)*parentConnection*

Returns YES if *parentConnection* should allow *newConnnection* to be created and set up, NO if *parentConnection* should refuse and immediately release *newConnection*. Use this method to limit the amount of NSConnections created in your application or to change the parameters of child NSConnections.

This delegate method is obsolete, and shouldn't be used. Use **connection:shouldMakeNewConnection:** instead.

Notifications

NSConnectionDidDieNotification

Posted when the NSConnection is deallocated or when it's notified that its NSPort has become invalid. This notification contains a notification object but no userInfo dictionary. The notification object is the NSConnection object.

The NSConnection object posting this notification is no longer useful, so all receivers should unregister themselves for any notifications involving the NSConnection.

See also: **NSPortDidBecomeInvalidNotification** (NSPort notification)

NSConnectionDidInitializeNotification

Posted when the NSConnection is initialized using **initWithReceivePort:sendPort:** (the designated initializer for NSConnection). This notification contains a notification object but no userInfo dictionary. The notification object is the NSConnection object.

See also: **– initWithReceivePort:sendPort:** (NSConnection)

NSCountedSet

Inherits From: NSMutableSet : NSSet : NSObject

Conforms To: NSCoding
NSCopying
NSMutableCopying (NSSet)
NSObject (NSObject)

Declared In: Foundation/NSSet.h

Class at a Glance

Purpose

An NSCountedSet object stores a modifiable set of objects, where a given object can be included in the set multiple times.

Principal Attributes

- The objects that make up the set.
- For each object in the set, a count of the number of times the object is included in the set.

Commonly Used Methods

– addObject:	Adds an object to the set.
– removeObject:	Removes an object from the set.

Class Description

The NSCountedSet class declares the programmatic interface to an object that manages a mutable set of objects. NSCountedSet provides support for the mathematical concept of a *counted set*. A counted set, both in its mathematical sense and in the implementation of NSCountedSet, is an unordered collection of elements, just as in a regular set, but the elements of the set aren't necessarily distinct. A counted set is also known as a *bag*.

Each distinct object inserted into an NSCountedSet object has a counter associated with it. NSCountedSet keeps track of the number of times objects are inserted and requires that objects be removed the same number of times. Thus, there is only one instance of an object in an NSSet even if the object has been added to the set multiple times. The NSSet and NSMutableSet classes are provided for static and dynamic sets (respectively) whose elements *are* distinct.

You add or remove objects from a counted set using the **addObject:** and **removeObject:** methods. An NSCountedSet may be queried using the **objectEnumerator** method, which provides for traversing elements of the set one by one. The **countForObject:** method returns the number of times the specified object has been added to this set**.**

Method Types

Initializing an NSCountedSet

– initWithArray:
– initWithSet:
– initWithCapacity:

Adding and removing entries

– addObject:
– removeObject:

Accessing the members

– allObjects
– count
– countForObject:
– objectEnumerator

Instance Methods

addObject:

– (void)**addObject:**(id)*anObject*

Adds *anObject* to the receiver if it isn't already a member. If *anObject* is already a member, **addObject:** increments the count associated with the object. In either case, *anObject* is then sent a *retain* message.

allObjects

– (NSArray *)**allObjects**

Returns an array containing the set's members, or an empty array if the set has no members. Each object is only represented in the array once—that is, if you add an object to the set more than once, it will appear

only once in the array that is returned by **allObjects**. The order of the objects in the array isn't defined. This method invokes **objectEnumerator** as part of its implementation.

count

– (unsigned int)**count**

Returns the number of unique members in the set. Objects that are added to the set multiple times are only reflected in the count once.

countForObject:

– (unsigned int)**countForObject:**(id)*anObject*

Returns the count associated with *anObject* in the receiver, which can be thought of as the number of occurrences of *anObject* that are present in the set.

initWithArray:

– (id)**initWithArray:**(NSArray *)*anArray*

Initializes a newly allocated counted set object with the contents of *anArray*. Returns **self**.

See also: – **initWithArray:** (NSSet), + **setWithArray:**(NSSet)

initWithCapacity:

– (id)**initWithCapacity:**(unsigned)*numItems*

The designated initializer, this method initializes a newly allocated set object, giving it enough memory to hold *numItems* objects.

See also: – **initWithCapacity:** (NSMutableSet), + **setWithCapacity:** (NSMutableSet)

initWithSet:

– (id)**initWithSet:**(NSSet *)*aSet*

Initializes a newly allocated counted set object with the contents of *aSet*. Returns **self**.

See also: – **initWithSet:**(NSSet), +**setWithSet:**(NSSet)

objectEnumerator

– (NSEnumerator *)**objectEnumerator**

Returns an enumerator object that lets you access each object in the set, independent of its count. This means that if you add a given object to the counted set multiple times, an enumeration of the set will produce that object only once.

When this method is used with a counted set, your code shouldn't modify the set during enumeration. If you intend to modify the set, use the **allObjects** method to create a "snapshot," then enumerate the snapshot and modify the original set.

See also: – **nextObject** (NSEnumerator)

removeObject:

– (void)**removeObject:**(id)*anObject*

If *anObject* is present in the set, decrements the count associated with it. If the count is decremented to zero, *anObject* is removed from the set and is sent a **release** message. **removeObject:** does nothing if *anObject* is not present in the receiver.

See also: – **countForObject:**

NSData Class Cluster

Class Cluster Description

NSData objects provide an object-oriented wrapper for byte buffers. This enables simple allocated buffers (that is, data with no embedded pointers) to take on the behavior of Foundation Kit objects. NSData is typically used for data storage. It is also useful in Distributed Objects applications, where data contained in NSData objects can be copied or moved between applications.

NSData objects can be used to wrap data of any size. When the data size is more than a few memory pages, NSData uses virtual memory management. NSData can also be used to wrap pre-existing data, regardless of how the data was allocated. NSData contains no information about the data itself (such as its type); the responsibility for deciding how to use the data lies with the client. In particular, it will not handle byte-order swapping when distributed between big-endian and little-endian machines. For typed data, use NSValue.

NSData provides an operating system-independent way to benefit from copy-on-write memory. The copy-on-write technique means that when data is copied through a virtual memory copy, an actual copy of the data is not made until there is an attempt to modify it.

The cluster's two public classes, NSData and NSMutableData, declare the programmatic interface for static and dynamic NSData objects, respectively.

The objects you create using these classes are referred to as *data objects*. Because of the nature of class clusters, data objects are not actual instances of the NSData or NSMutableData classes but instead are instances of one of their private subclasses. Although a data object's class is private, its interface is public, as declared by these abstract superclasses, NSData and NSMutableData.

Generally, you instantiate a data object by sending one of the **data...** messages to either the NSData or NSMutableData class object. These methods return a data object containing the bytes you pass in as arguments. If you use one of the **data...** methods whose name *does not* include "NoCopy" (such as **dataWithBytes:length:**), the bytes to be contained by the data object are copied as part of the instantiation process, and the data object then contains the copied bytes. When you subsequently release a data object that has been instantiated in this manner, the bytes contained by the data object—those that were copied during instantiation—are automatically freed. If you instantiate a data object with one of the methods whose name includes "NoCopy," however, (such as **dataWithBytesNoCopy:length:**) the bytes are not copied and are freed when the data object is released.

The NSData classes adopt the NSCopying and NSMutableCopying protocols, making it convenient to convert between efficient, read-only data objects and mutable data objects.

NSData

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSMutableCopying NSObject (NSObject)
Declared In:	Foundation/NSData.h Foundation/NSSerialization.h

Class at a Glance

Purpose

An NSData object stores immutable data in the form of bytes.

Principal Attributes

- A count of the number of bytes in the data object.
- The sequence of bytes contained in the data object.

Creation

+ data	Returns an empty data object.
+ dataWithBytes:length:	Returns a data object that contains a copy of the specified bytes.
+ dataWithBytesNoCopy:length:	Returns a data object that contains the specified bytes (without copying them).
+ dataWithContentsOfFile:	Returns a data object initialized with the contents of a file.
+ dataWithContentsOfMappedFile:	Returns a data object initialized with the contents of a mapped file.
+ dataWithData:	Returns a data object initialized with the contents of another data object.

Commonly Used Methods

– bytes	Returns a pointer to the data object's contents.
– getBytes:	Copies the data object's contents into a buffer.
– length	Returns the number of bytes contained by the data object.

Primitive Methods

– bytes
– length

Class Description

The NSData class declares the programmatic interface to an object that contains immutable data in the form of bytes. NSData's two primitive methods—**bytes** and **length**—provide the basis for all of the other methods in the interface. The **bytes** method returns a pointer to the bytes contained in the data object. **length** returns the number of bytes contained in the data object.

NSData provides access methods for copying bytes from a data object into a specified buffer. **getBytes** copies all of the bytes into a buffer, whereas **getBytes:length:** copies bytes into a buffer of a given length. **getBytes:range:** copies a range of bytes from a starting point within the bytes themselves. You can also obtain a data object that contains a subset of the bytes in another data object by using the **subdataWithRange:** method. Or, you can use the **description** method to return an NSString representation of the bytes in a data object.

For determining if two data objects are equal, NSData provides the **isEqualToData:** method, which does a byte-for-byte comparison.

The **writeToFile:atomically:** method enables you to write the contents of a data object to a file.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

NSCopying

– copyWithZone:

NSMutableCopying

– mutableCopyWithZone:

Method Types

Creating data objects

+ allocWithZone:
+ data
+ dataWithBytes:length:
+ dataWithBytesNoCopy:length:
+ dataWithContentsOfFile:
+ dataWithContentsOfMappedFile:
+ dataWithData:
+ dataWithStream:
– initWithBytes:length:
– initWithBytesNoCopy:length:
– initWithContentsOfFile:
– initWithContentsOfMappedFile:
– initWithData:
– initWithStream:

Accessing data

– bytes
– description
– getBytes:
– getBytes:length:
– getBytes:range:
– subdataWithRange:

Deserializing data

– deserializeAlignedBytesLengthAtCursor:
– deserializeBytes:length:atCursor:
– deserializeDataAt:ofObjCType:atCursor:context:
– deserializeIntAtCursor:
– deserializeIntAtIndex:
– deserializeInts:count:atCursor:
– deserializeInts:count:atIndex:

Testing data

– isEqualToData:
– length

Storing data

– writeToFile:atomically:

Class Methods

allocWithZone:

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

Creates and returns an uninitialized data object in the specified zone. If the receiver is the NSData class object, an instance of an appropriate immutable subclass is returned; otherwise, an object of the receiver's class is returned.

Typically, you create temporary data objects using the **data...** class methods, not the **alloc...** and **init...** methods.

data

+ (id)data

Creates and returns an empty data object. This method is declared primarily for the use of mutable subclasses of NSData.

dataWithBytes:length:

+ (id)**dataWithBytes:**(const void *)*bytes* **length:**(unsigned int)*length*

Creates and returns a data object containing *length* bytes copied from the buffer *bytes*.

See also: **+ dataWithBytesNoCopy:length:**

dataWithBytesNoCopy:length:

+ (id)**dataWithBytesNoCopy:**(void *)*bytes* **length:**(unsigned int)*length*

Creates and returns a data object that holds *length* bytes from the buffer *bytes*.

See also: + **dataWithBytes:length:**

dataWithContentsOfFile:

+ (id)**dataWithContentsOfFile:**(NSString *)*path*

Creates and returns a data object by reading every byte from the file specified by *path*.

The following code example creates a data object *myData* initialized with the contents of **myFile.txt**. The path must be absolute.

```
NSString *thePath = @"/u/smith/myFile.txt";
NSData *myData = [NSData dataWithContentsOfFile:thePath];
```

See also: + **dataWithContentsOfMappedFile:**

dataWithContentsOfMappedFile:

+ (id)**dataWithContentsOfMappedFile:**(NSString *)*path*

Creates and returns a data object from the mapped file specified by *path*. Because of file mapping restrictions, this method should only be used if the file is guaranteed to exist for the duration of the data object's existence. It is generally safer to use the **dataWithContentsOfFile:** method.

This methods assumes that mapped files are available from the underlying operating system. A mapped file uses virtual memory techniques to avoid copying pages of the file into memory until they are actually needed.

See also: + **dataWithContentsOfFile:**

dataWithData:

+ (id)**dataWithData:**(NSData *)*aData*

Creates and returns a data object containing the contents of another data object, *aData*.

dataWithStream:

+ (id)**dataWithStream:**(NXStream *)*stream*

Creates and returns a data object containing the contents of *stream*.

See also: **– initWithStream:**

Instance Methods

bytes

– (const void *)**bytes**

Returns a pointer to the data object's contents. This method returns read-only access to the data.

See also: **– description, – getBytes:, – getBytes:length:, – getBytes:range:**

description

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

Returns an NSString object that contains a hexadecimal representation of the receiver's contents in the property list format for NSData objects.

See also: – **bytes, – getBytes:, – getBytes:length:, – getBytes:range:**

deserializeAlignedBytesLengthAtCursor:

– (unsigned int)**deserializeAlignedBytesLengthAtCursor:**(unsigned *)*cursor*

Reads a sequence of bytes from the receiver beginning at location *cursor* and returns them formatted as an unsigned integer. On return, *cursor* is set to the location just past the bytes that were read.

Use this method to read an integer that was serialized using the **serializeAlignedBytesLength:** method of NSMutableData. This method ignores any filler bytes that were serialized by the **serializeAlignedBytesLength:** method.

See also: **– deserializeIntAtCursor:, – deserializeIntAtIndex:**

deserializeBytes:length:atCursor:

– (void)**deserializeBytes:**(void *)*buffer*
length:(unsigned int)*bytes*
atCursor:(unsigned int*)*cursor*

Reads a sequence of bytes from the receiver beginning at location *cursor* and places them in *buffer*. The *bytes* parameter specifies the number of bytes to be read. On return, *cursor* is set to the location just beyond the bytes that were read.

See also: **– getBytes:range:**

deserializeDataAt:ofObjCType:atCursor:context:

– (void)**deserializeDataAt:**(void *)*data*
ofObjCType:(const char *)*type*
atCursor:(unsigned int*)*cursor*
context:(id <NSObjCTypeSerializationCallBack>)*callback*

Reads a sequence of bytes from the receiver beginning at location *cursor* and places them in *data*. The bytes are formatted according to the Objective-C type code given in *type*. If *type* specifies an object, *callback* is used to deserialize the object; in such a case, *callback* must itself be an object that conforms to the NSObjCTypeSerializationCallBack protocol. If *type* does not specify an object, *callback* can be **nil**.

For information on on creating an Objective-C type code suitable for *type*, see the description of the **@encode()** compiler directive in *Object-Oriented Programming and the Objective-C Language*.

deserializeIntAtCursor:

– (int)**deserializeIntAtCursor:**(unsigned int*)*cursor*

Reads a sequence of bytes from the receiver beginning at location *cursor* and returns them as an integer value. On return, *cursor* is set to the location just past the bytes that were read.

See also: – deserializeIntAtIndex:, – serializeInt: (NSMutableData)

deserializeIntAtIndex:

– (int)**deserializeIntAtIndex:**(unsigned int)*index*

Reads a sequence of bytes from the receiver starting at *index* and returns them as an integer value.

See also: – getBytes:range:, – serializeInt: (NSMutableData)

deserializeInts:count:atCursor:

– (void)**deserializeInts:**(int *)*intBuffer*
count:(unsigned int)*numInts*
atCursor:(unsigned int*)*cursor*

Reads *numInts* integers as a sequence of bytes from the receiver and copies them into *intBuffer*. The bytes are read from the receiver beginning at location *cursor*. On return, *cursor* is set to the location just past the integers that were read.

See also: – getBytes:range:, – deserializeIntAtCursor:, – serializeInts:count: (NSMutableData)

deserializeInts:count:atIndex:

– (void)**deserializeInts:**(int *)*intBuffer*
count:(unsigned int)*numInts*
atIndex:(unsigned int)*index*

Reads *numInts* integers as a sequence of bytes from the receiver and copies them into *intBuffer*. The bytes are read from the receiver starting at *index*.

See also: – getBytes:range:, – deserializeIntAtIndex:, – serializeInts:count: (NSMutableData)

getBytes:

– (void)**getBytes:**(void *)*buffer*

Copies a data object's contents into *buffer.*

For example, the following code excerpt initializes a data object *myData* with the NSString *myString*. It then uses **getBytes:** to copy the contents of *myData* into *aBuffer*.

```
unsigned char aBuffer[20];
NSString *myString = @"Test string.";
NSData *myData = [NSData dataWithBytes:[myString cString]
        length:[myString cStringLength]];

[myData getBytes:aBuffer];
```

See also: – **bytes**, – **description**, – **getBytes:length:**, – **getBytes:range:**

getBytes:length:

– (void)**getBytes:**(void *)*buffer* **length:**(unsigned int)*length*

Copies up to *length* bytes from the start of the receiver into *buffer.*

See also: – **bytes**, – **description**, – **getBytes:,** – **getBytes:range:**

getBytes:range:

– (void)**getBytes:**(void *)*buffer* **range:**(NSRange)*range*

Copies the receiver's contents into *buffer*, from *range* that is within the bytes in the object. If *range* isn't within the receiver's range of bytes, an NSRangeException is raised.

See also: – **bytes**, – **description**, – **getBytes:,** – **getBytes:length:**

hash

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

Returns an unsigned integer that can be used as a table address in a hash table structure. For a data object, **hash** returns the length of the data object. If two data objects are equal (as determined by the **isEqual:** method), they have the same hash value.

See also: – **isEqual:**

initWithBytes:length:

– (id)**initWithBytes:**(const void *)*bytes* **length:**(unsigned int)*length*

Initializes a newly allocated data object by adding to it length bytes of data copied from the buffer bytes. Returns self.

See also: + **dataWithBytes:length:**, – **initWithBytesNoCopy:length:**

initWithBytesNoCopy:length:

– (id)**initWithBytesNoCopy:**(void *)*bytes* **length:**(unsigned int)*length*

Initializes a newly allocated data object by adding to it length bytes of data from the buffer bytes. Returns self.

See also: + **dataWithBytes:length:**, – **initWithBytes:length:**

initWithContentsOfFile:

– (id)**initWithContentsOfFile:**(NSString *)*path*

Initializes a newly allocated data object by reading into it the data from the file specified by *path*. **Returns self.**

See also: + **dataWithContentsOfFile:**, – **initWithContentsOfMappedFile:**

initWithContentsOfMappedFile:

– (id)**initWithContentsOfMappedFile:**(NSString *)*path*

Initializes a newly allocated data object by reading into it the mapped file specified by *path*. **Returns self.**

See also: + **dataWithContentsOfMappedFile:**, – **initWithContentsOfFile:**

initWithData:

– (id)**initWithData:**(NSData *)*data*

Initializes a newly allocated data object by placing in it the contents of another data object, *data*. **Returns self.**

initWithStream:

– (id)**initWithStream:**(NXStream *)*stream*

Initializes a newly allocated data object by placing in it the contents of *stream*. **Returns self.**

isEqual:

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

Returns YES if the receiver and *anObject* are equal; otherwise returns NO. A YES return value indicates that the receiver and *anObject* are both instances of classes that inherit from NSData and that both contain the same data (as determined by the **isEqualToData:** method).

See also: – **isEqualToData:**

isEqualToData:

– (BOOL)**isEqualToData:**(NSData *)*otherData*

Compares the receiving data object to *otherData*. If the contents of *otherData* are equal to the contents of the receiver, this method returns YES. If not, it returns NO. Two data objects are equal if they hold the same number of bytes, and if the bytes at the same position in the objects are the same.

See also: – **isEqual:**

length

– (unsigned int)**length**

Returns the number of bytes contained in the receiver.

subdataWithRange:

– (NSData *)**subdataWithRange:**(NSRange)*range*

Returns a data object containing a copy of the receiver's bytes that fall within the limits specified by *range*. If *range* isn't within the receiver's range of bytes, an NSRangeException is raised.

For example, the following code excerpt initializes a data object, *data2*, to contain a sub-range of *data1*:

```
NSString *myString = @"ABCDEFG";
NSRange range = {2, 4};
NSData *data1, *data2;

data1 = [NSData dataWithBytes:[myString cString]
        length:[myString cStringLength]];

data2 = [data1 subdataWithRange:range];
```

The result of this excerpt is that *data2* contains “CDEF**”.**

writeToFile:atomically:

– (BOOL)**writeToFile:**(NSString *)*path* **atomically:**(BOOL)*flag*

Writes the bytes in the receiver to the file specified by *path*. If *flag* is YES, the data is written to a backup file and then, assuming no errors occur, the backup file is renamed to the specified file name. Otherwise, the data is written directly to the specified file.

If *path* contains a tilde (~) character, you must expand it with **stringByExpandingTildeInPath:** before invoking this method.

YES is returned if the operation succeeded, otherwise NO is returned.

NSMutableData

Inherits From:	NSData : NSObject
Conforms To:	NSCoding NSCopying NSMutableCopying (NSData) NSObject (NSObject)
Declared In:	Foundation/NSData.h Foundation/NSSerialization.h

Class at a Glance

Purpose

An NSMutableData object stores mutable data in the form of bytes.

Principal Attributes

- A count of the number of bytes in the mutable data object.
- The sequence of bytes contained in the mutable data object.

Creation

+ dataWithCapacity:	Returns an NSMutableData with enough allocated memory to hold a specified number of bytes.
+ dataWithLength:	Returns an NSMutableData that contains a specified number of zero-filled bytes.

Commonly Used Methods

– mutableBytes	A pointer to the bytes in the NSMutableData object.
– replaceBytesInRange:withBytes:	Replaces a range of bytes in the NSMutableData object.

Primitive Methods

– mutableBytes
– setLength:

Class Description

The NSMutableData class declares the programmatic interface to an object that contains modifiable data in the form of bytes. NSMutableData's two primitive methods—**mutableBytes** and **setLength:**—provide the basis for all of the other methods in its interface. The **mutableBytes** method returns a pointer for writing into the bytes contained in the mutable data object. **setLength:** allows you to truncate or extend the length of a mutable data object.

increaseLengthBy: also allows you to change the length of a mutable data object.

The **appendBytes:length:** and **appendData:** methods let you append bytes or the contents of another data object to a mutable data object. You can replace a range of bytes in a mutable data object with zeros (using the **resetBytesInRange:** method), or with different bytes (using the **replaceBytesInRange:withBytes:** method).

Method Types

Creating an NSMutableData

+ allocWithZone:
+ dataWithCapacity:
+ dataWithLength:
– initWithCapacity:
– initWithLength:

Adjusting capacity

– increaseLengthBy:
– setLength:

Accessing data

– mutableBytes

Adding data

– appendBytes:length:
– appendData:

Serializing data

– serializeAlignedBytesLength:
– serializeDataAt:ofObjCType:context:
– serializeInt:
– serializeInt:atIndex:
– serializeInts:count:
– serializeInts:count:atIndex:

Modifying data

– replaceBytesInRange:withBytes:
– resetBytesInRange:
– setData:

Class Methods

allocWithZone:

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

Creates and returns an uninitialized data object in the specified zone. If the receiver is the NSMutableData class object, an instance of an appropriate subclass is returned; otherwise, an object of the receiver's class is returned.

Typically, you create objects using the **data...** class methods, not the **alloc...** and **init...** methods. Note that it's your responsibility to release objects created with the **alloc...** methods.

dataWithCapacity:

+ (id)**dataWithCapacity:**(unsigned int)*aNumItems*

Creates and returns an NSMutableData object, initially allocating enough memory to hold *aNumItems* objects. Mutable data objects allocate additional memory as needed, so *aNumItems* simply establishes the object's initial capacity.

Note: **dataWithCapacity:** doesn't necessarily allocate its memory at the time of method invocation. When it does allocate its memory, though, it initially allocates the specified amount.

See also: **+ dataWithLength:**, – **initWithCapacity:**, – **initWithLength:**

dataWithLength:

+ (id)**dataWithLength:**(unsigned int)*length*

Creates an autoreleased, mutable data object of *length* bytes, filled with zeros.

See also: **+ dataWithCapacity:**, – **initWithCapacity:**, – **initWithLength:**

Instance Methods

appendBytes:length:

– (void)**appendBytes:**(const void *)*bytes* **length:**(unsigned int)*length*

Appends *length* bytes to the receiver from the buffer *bytes*.

This excerpt copies the bytes in *data2* into *aBuffer*, and then appends *aBuffer* to *data1*.

```
NSMutableData *data1, *data2;
NSString *firstString =  @"ABCD";
NSString *secondString = @"EFGH";
unsigned char *aBuffer;
unsigned len;

data1 = [NSMutableData dataWithBytes:[firstString cString]
        length:[firstString cStringLength]];
data2 = [NSMutableData dataWithBytes:[secondString cString]
        length:[secondString cStringLength]];

len = [data2 length];
aBuffer = malloc(len);

[data2 getBytes:aBuffer];
[data1 appendBytes:aBuffer length:len];
```

The final value of *data1* is the series of ASCII characters “ABCDEFGH”.

See also: – **appendData:**

appendData:

– (void)**appendData:**(NSData *)*otherData*

Appends the contents of a data object *otherData* to the receiver.

See also: – **appendBytes:length:**

increaseLengthBy:

– (void)**increaseLengthBy:**(unsigned int)*extraLength*

Increases the length of the receiver by *extraLength*. The additional bytes are all set to zero.

See also: – **setLength:**

initWithCapacity:

– (id)**initWithCapacity:**(unsigned int)*capacity*

Initializes a newly allocated mutable data object, giving it enough memory to hold *capacity* bytes. Sets the length of the data object to 0. Returns **self**.

See also: + **dataWithCapacity:**, – **initWithLength:**

initWithLength:

– (id)**initWithLength:**(unsigned int)*length*

Initializes a newly allocated mutable data object, giving it enough memory to hold *length* bytes. Fills the object with zeros up to *length*. Returns **self**.

See also: **+ dataWithCapacity:**, **+ dataWithLength:**, **– initWithCapacity:**

mutableBytes

– (void *)**mutableBytes**

Returns a pointer to the receiver's data.

In the following code example, **mutableBytes** is used to return a pointer to the bytes in *data2*. The bytes in *data2* are then overwritten with the contents of *data1*.

```
NSMutableData *data1, *data2;
NSString *myString = @"string for data1";
NSString *yourString = @"string for data2";
unsigned char *firstBuffer, secondBuffer[20];

/* initialize data1, data2, and secondBuffer... */
data1 = [NSMutableData dataWithBytes:[myString cString]
        length:[myString length]];
data2 = [NSMutableData dataWithBytes:[yourString cString]
        length:[yourString length]];

[data2 getBytes:secondBuffer];
NSLog(@"data2 before: \"%s\"\n", (char *)secondBuffer);

firstBuffer = [data2 mutableBytes];
[data1 getBytes:firstBuffer];
NSLog(@"data1: \"%s\"\n", (char *)firstBuffer);

[data2 getBytes:secondBuffer];
NSLog(@"data2 after: \"%s\"\n", (char *)secondBuffer);
```

This is the output from the above code example:

```
Oct  3 15:59:51 [1113] data2 before: "string for data2"
Oct  3 15:59:51 [1113] data1: "string for data1"
Oct  3 15:59:51 [1113] data2 after: "string for data1"
```

replaceBytesInRange:withBytes:

– (void)**replaceBytesInRange:**(NSRange)*range* **withBytes:**(const void *)*bytes*

Specifies a range within the contents of a mutable data object to be replaced by *bytes*. If the location of *range* isn't within the receiver's range of bytes, an NSRangeException is raised. The receiver is resized to accomodate the new bytes, if necessary.

In the following code excerpt, a range of bytes in *data1* is replaced by the bytes in *data2*.

```
NSMutableData *data1, *data2;
NSString *myString = @"Liz and John";
NSString *yourString = @"Larry";
unsigned len;
unsigned char *aBuffer;
NSRange range = {8, [yourString cStringLength]};

data1 = [NSMutableData dataWithBytes:[myString cString]
        length:[myString cStringLength]];

data2 = [NSMutableData dataWithBytes:[yourString cString]
        length:[yourString cStringLength]];

len = [data2 length];
aBuffer = malloc(len);
[data2 getBytes:aBuffer];
[data1 replaceBytesInRange:range withBytes:aBuffer];
```

The contents of *data1* changes from "Liz and John" to "Liz and Larry."

See also: – **resetBytesInRange:**

resetBytesInRange:

– (void)**resetBytesInRange:**(NSRange)*range*

Specifies a range within the contents of a mutable data object to be replaced by zeros. If the location of *range* isn't within the receiver's range of bytes, an NSRangeException is raised. The receiver is resized to accomodate the new bytes, if necessary.

See also: – **replaceBytesInRange:withBytes:**

serializeAlignedBytesLength:

– (void)**serializeAlignedBytesLength:**(unsigned int)*length*

Appends the bytes of *length* to the end of the receiver. This method may add extra filler bytes to increase the efficiency of deserializing subsequent data. Use of this method is optional; you can invoke **serializeInt:**

instead. However, if you use this method, you should match its use by invoking **deserializeAlignedBytesLengthAtCursor:** to read the bytes of *length* later.

serializeDataAt:ofObjCType:context:

– (void)**serializeDataAt:**(const void *)*data*
ofObjCType:(const char *)*type*
context:(id <NSObjCTypeSerializationCallBack>)*callback*

Appends a sequence of bytes, specified by *data*, to the receiver. The bytes are formatted according to the Objective-C type code given in *type*. If *type* specifies an object, *callback* is used to serialize the object pointed to by *data*; in such a case, *callback* must itself be an object that conforms to the NSObjCTypeSerializationCallBack protocol. If *type* does not specify an object, *callback* can be **nil**.

For informationon on creating an Objective-C type code suitable for *type*, see the description of the **@encode()** compiler directive in *Object-Oriented Programming and the Objective-C Language*.

See also: **– deserializeDataAt:ofObjCType:atCursor:context:** (NSData)

serializeInt:

– (void)**serializeInt:**(int)*value*

Appends the bytes of *value* to the end of the receiver.

See also: – **serializeAlignedBytesLength:**

serializeInt:atIndex:

– (void)**serializeInt:**(int)*value* **atIndex:**(unsigned int)*index*

Replaces the bytes of an integer at location *index* in the receiver with the bytes of *value*.

See also: **– replaceBytesInRange:withBytes:**

serializeInts:count:

– (void)**serializeInts:**(int *)*intBuffer* **count:**(unsigned int)*numInts*

Appends the bytes of *numInts* integers in *intBuffer* to the receiver.

See also: **– serializeInt:**

serializeInts:count:atIndex:

– (void)**serializeInts:**(int *)*intBuffer*
count:(unsigned int)*numInts*
atIndex:(unsigned int)*index*

Replaces the bytes of *numInts* integers currently in the receiver with *numInts* integers in *intBuffer*.

See also: **– replaceBytesInRange:withBytes:**

setData:

– (void)**setData:**(NSData *)*aData*

Uses **replaceBytesInRange:withBytes:** to replace the entire contents of the receiver with the contents of *aData*.

setLength:

– (void)**setLength:**(unsigned int)*length*

Extends or truncates a mutable data object to *length*. If the mutable data object is extended, the additional bytes are filled with zero.

See also: – **increaseLengthBy:**

NSDate Class Cluster

Class Cluster Description

NSDate objects represent a single point in time. The NSDate cluster's single public superclass, NSDate, declares the programmatic interface for specific and relative time values.

The objects you create using NSDate are referred to as *date objects*. They are immutable objects. Because of the nature of class clusters, objects returned by the NSDate class are not instances of that abstract class but of one of its private subclasses. Although a date object's class is private, its interface is public, as declared by the abstract superclass, NSDate. (See the "Class Clusters" programming topic for more information on class clusters and creating subclasses within a cluster.)

Generally, you instantiate a suitable date object by invoking one of the **date...** class methods.

The date classes adopt the NSCopying and NSCoding protocols.

NSDate

Inherits From:	NSObject
Conforms To:	NSCoding, NSCopying NSObject (NSObject)
Declared In:	Foundation/NSDate.h

Class at a Glance

Purpose

An NSDate object stores a date and time that can be compared to other dates and times.

Principal Attributes

- Seconds since absolute reference date (1 January, 2001, GMT)

Creation

+ date	Represents the current date and time.
+ dateWithTimeIntervalSinceNow:	Represents a date relative to the current date and time.
+ dateWithTimeIntervalSinceReferenceDate:	Represents a date relative to the absolute reference date.

Commonly Used Methods

– earlierDate:	Compares the receiver to the argument and returns the earlier of the two.
– isEqualToDate:	Returns YES if the receiver and the argument are equal.
– laterDate:	Compares the receiver to the argument and returns the later of the two.
– timeIntervalSinceNow	Returns the number of seconds difference between the receiver and the current date and time.

Primitive Method

– timeIntervalSinceReferenceDate

Class Description

NSDate is an abstract class that provides behavior for creating dates, comparing dates, representing dates, computing intervals, and similar functionality. It presents a programmatic interface through which suitable date objects are requested and returned. Date objects returned from NSDate are lightweight and immutable since they represent a invariant point in time. This class is designed to provide the foundation for arbitrary calendrical representations. Its subclass NSCalendarDate offers date objects that are suitable for representing dates according to western calendrical systems.

"Date" as used above implies clock time as well. The standard unit of time for date objects is a value typed as NSTimeInterval and expressed as seconds. The NSTimeInterval type makes possible a wide and fine-grained range of date and time values, giving accuracy within milliseconds for dates 10,000 years apart.

NSDate and its subclasses compute time as seconds *relative* to an absolute reference date. This reference date is the first instant of 1 January, 2001, Greenwich Mean Time (GMT). NSDate converts all date and time representations to and from NSTimeInterval values that are relative to this absolute reference date. A positive interval relative to a date represents a point in the future, a negative interval represents a time in the past.

Note: Rhapsody and supported UNIX systems implement time according to the Network Time Protocol (NTP) standard, which is based on Coordinated Universal Time. The current private implementations of NSDate follow the NTP standard. However, they do not account for leap seconds and therefore are not synchronized with International Atomic Time (the most accurate).

Like various other Foundation classes, NSDate enables you to obtain operating-system functionality (dates and times) without depending on operating-system internals. It also provides a basis for the NSRunLoop and NSTimer classes, which use concrete date objects to implement local event loops and timers.

NSDate's sole primitive method, **timeIntervalSinceReferenceDate**, provides the basis for all the other methods in the NSDate interface. It returns a time value relative to an absolute reference date.

Creating NSDate Objects

Use a date object to store a point in time. If you want to store the current time, use the **date** class method to create the date object. If you want to store some time other than the current time, use one of the **dateWithTimeInterval...** methods.

The **dateWithTimeInterval...** methods create date objects relative to a particular time, which the method name describes. You specify (in seconds) how much more recent or how much more in the past you want your date object to be. To specify a date that occurs earlier than the method's reference date, use a negative number of seconds. The code fragment below defines two date objects. **tomorrow** is exactly 24 hours from the current date and time, and **yesterday** is exactly 24 hours earlier than the current date and time.

```
NSTimeInterval secondsPerDay = 24 * 60 * 60;
NSDate *tomorrow = [NSDate
          dateWithTimeIntervalSinceNow:secondsPerDay];
NSDate *yesterday = [NSDate
          dateWithTimeInstervalSinceNow:-secondsPerDay];
```

To get new date objects with date-and-time values adjusted from existing date objects, use **addTimeInterval:**.

```
NSTimeInterval secondsPerDay = 24 * 60 * 60;
NSDate *today = [NSDate date];
NSDate *tomorrow, yesterday;

tomorrow = [today addTimeInterval:secondsPerDay];
yesterday = [today addTimeInterval:-secondsPerDay];
```

NSCalendarDate

The NSDate class cluster provides, for your convenience, a public concrete subclass of NSDate that satisfies many requirements for dates and times. This subclass, NSCalendarDate, enables you to represent dates as arbitrary strings, to create new date objects from string representations, to extract date and time elements from date objects, and to do other calendar-related functions. You can create an NSCalendarDate out of your NSDate using the **dateWithCalendarFormat:timeZone:** method.

Comparing NSDate Objects

To obtain the difference between a date object and another point in time, send a **timeInterval...** message to the date object. For instance, **timeIntervalSinceNow** gives you the time, in seconds, between the current time and the receiving date object.

To compare dates, use the **isEqualToDate:**, **compare:**, **laterDate:**, and **earlierDate:** methods. These methods perform exact comparisons, which means they will detect subsecond differences between dates. You might want to compare dates with a less fine granularity. For example, you might want to consider two dates equal if they are within a minute of each other. If this is the case, use **timeIntervalSinceDate:** to compare the two dates or use NSCalendarDate objects instead. The following code shows how to use **timeIntervalSinceDate:** to see if two dates are within one minute (60 seconds) of each other.

```
if (fabs([date2 timeIntervalSinceDate:date1]) < 60) ...
```

NSString Representations for NSDates

To represent your date object as an NSString, use the **description...** methods. The simplest method, **description**, prints out the date in the format YYYY-MM-DD HH:MM:SS ±HHMM, where ±HHMM represents the time zone offset in hours and minutes from GMT. (Adding the offset to the specific time yields the equivalent GMT.) To have a specific locale dictionary affect the representation of your NSDate, use **descriptionWithLocale:** instead of **description**. The following keys in the locale dictionary affect NSDates:

Key	Description
NSTimeDateFormatString	Specifies how dates with times are printed. The default is to use abbreviated months and days with a 24 hour clock, as in “Sun Jan 01 23:00:00 +6 2001.”

Key	Description
NSAMPMDesignation	Specifies how the morning and afternoon designations are printed. The default is AM and PM.
NSMonthNameArray	Specifies the names for the months.
NSShortMonthNameArray	Specifies the abbreviations for the months.
NSWeekDayNameArray	Specifies the names for the days of the week.
NSShortWeekDayNameArray	Specifies the abbreviations for the days of the week.

Subclassing NSDate

If you want to subclass NSDate to obtain behavior different than that provided by the private subclasses, you must do these things:

- Declare a suitable instance variable to hold the date and time value (relative to an absolute reference date).
- Override the **timeIntervalSinceReferenceDate** instance method to provide the correct date and time value based on your instance variable.
- Override **initWithTimeIntervalSinceReferenceDate:**, the designated initializer method.
- Override **descriptionWithLocale:** if you want to affect the way your NSDate subclass is converted to a string. When you use the %@ conversion specifier inside of an NSString to convert an NSDate object, NSDate's **descriptionWithLocale:** method decides how to represent the date.

Your subclass may use a different reference date than the absolute reference date used by NSDate (the first instance of 1 January 2001 GMT). If it does, it must still use the absolute reference date in its implementations of the methods **timeIntervalSinceReferenceDate** and **initWithTimeIntervalSinceReferenceDate:**. That is, the reference date referred to in the titles of these methods is the absolute reference date. If you do not use the absolute reference date in these methods, comparisons between NSDate objects of your subclass and NSDate objects of a private subclass will not work.

Adopted Protocols

NSCopying

– copyWithZone:

NSCoding

– encodeWithCoder:
– initWithCoder:

Method Types

Creating an NSDate instance

+ date
+ dateWithTimeIntervalSinceNow:
+ dateWithString:
+ dateWithNaturalLanguageString:
+ dateWithNaturalLanguageString:locale:
+ dateWithTimeIntervalSinceReferenceDate:
+ dateWithTimeIntervalSince1970:
+ distantFuture
+ distantPast
– initWithTimeIntervalSinceNow:
– initWithString:
– initWithTimeInterval:sinceDate:
– init
– initWithTimeIntervalSinceReferenceDate:
– addTimeInterval:

Comparing dates

– isEqualToDate:
– earlierDate:
– laterDate:
– compare:

Getting time intervals

– timeIntervalSinceDate:
– timeIntervalSinceNow
+ timeIntervalSinceReferenceDate
– timeIntervalSinceReferenceDate
– timeIntervalSince1970

Representing dates as NSStrings

– description
– descriptionWithLocale:
– descriptionWithCalendarFormat:timeZone:locale:

Converting to an NSCalendarDate object

– dateWithCalendarFormat:timeZone:

Class Methods

date

+ (id)**date**

Creates and returns an NSDate set to the current date and time. This method uses the default initializer method for the class, **init**.

A typical example of using **date** to get the current date is:

```
NSDate *today = [NSDate date];
```

See also: **– init**

dateWithNaturalLanguageString:

+ **dateWithNaturalLanguageString:**(NSString *)*string*

Creates and returns an NSDate set to the date and time specified by *string*. The argument *string* can be a colloquial specification of a date, such as “last Tuesday at dinner,” “3pm December 31, 1995,” “12/31/95,” or “31/12/95.” In parsing *string*, this method uses the date and time preferences stored in the user’s defaults database. (See **dateWithNaturalLanguageString:locale:** for a list of the specific items used.)

See also: + **dateWithNaturalLanguageString:locale:**

dateWithNaturalLanguageString:locale:

+ **dateWithNaturalLanguageString:**(NSString *)*string* **locale:**(NSDictionary *)*localeDictionary*

Creates and returns an NSDate set to the date and time specified by *string*. The argument *string* can be a colloquial specification of a date, such as “last Tuesday at dinner,” “3pm December 31, 1995,” “12/31/95,” or “31/12/95.” The keys and values that represent the locale data from *localeDictionary* are used when

parsing the string. In addition to the locale keys listed in the class description, these keys are used when parsing natural language strings:

Key	Description
NSDateTimeOrdering	Determines how to use ambiguous numbers. Specify this value as a permutation of the letters M (month), D (day), Y (year), and H (hour). For example, MDYH treats “2/3/95 10” as the 3rd day of February 1995 at 10:00am, whereas DMYH treats the same value as the 2nd day of March 1995 at 10:00am. If fewer numbers are specified than are needed, the numbers are prioritized to satisfy day first, then the month, and then the year. For example, if you supply only the value 12, it means the 12th day of this month in this year because the day must be specified. If you supply “2 12” it means either February 12 or December 2, depending on if the ordering is “MDYH” or “DMYH.”
NSEarlierTimeDesignations	An array of strings that denote a time in the past. These are adjectives that modify values from NSYearMonthWeekDesignations. The defaults are “prior,” “last,” “past,” and “ago.”
NSHourNameDesignations	Strings that identify the time of day. These strings should be bound to an hour. The default is this array of arrays: (0, midnight), (12, noon, lunch), (10, morning), (14, afternoon), (19, dinner).
NSLaterTimeDesignations	An array of strings that denote a time in the future. This is an adjective that modifies a value from NSYearMonthWeekDesignations. The default is “next.”
NSNextDayDesignations	A string that identifies the day after today. The default is “tomorrow.”
NSNextNextDayDesignations	A string that identifies the day after tomorrow. The default is “nextday.”
NSPriorDayDesignations	A string that identifies the day before today. The default is “yesterday.”
NSThisDayDesignations	A string that identifies what this day is called. The default is “today.”
NSYearMonthWeekDesignations	An array of strings that specify the word for year, month, and week in the current locale. The defaults are “year,” “month,” and “week.”

See also: + **dateWithNaturalLanguageString:**

dateWithString:

+ (id)**dateWithString:**(NSString *)*aString*

Creates and returns an NSDate with a date and time value specified by the international string-representation format: YYYY-MM-DD HH:MM:SS ±HHMM, where ±HHMM is a time zone offset in hours and minutes from GMT (for example, “1994-05-23 10:45:32 +0600”). You must specify all fields of the format, including the time zone offset, which must have a plus- or minus-sign prefix.

See also: – **initWithString:**

dateWithTimeIntervalSinceNow:

+ (id)**dateWithTimeIntervalSinceNow**:(NSTimeInterval)*seconds*

Creates and returns an NSDate set to the specified number of seconds from the current date and time. Use a negative argument value to specify a date before the current date.

See also: – **initWithTimeIntervalSinceNow:**

dateWithTimeIntervalSince1970:

+ (id)**dateWithTimeIntervalSince1970:**(NSTimeInterval)*seconds*

Creates an returns an NSDate set to the specified number of seconds from the reference date 1 January 1970. Use a negative argument to specify a date before this date.

This method is useful for creating NSDate objects from **time_t** values returned by BSD system functions.

See also: – **timeIntervalSince1970**

dateWithTimeIntervalSinceReferenceDate:

+ (id)**dateWithTimeIntervalSinceReferenceDate**:(NSTimeInterval)*seconds*

Creates and returns an NSDate set to a specified number of seconds from the absolute reference date (the first instant of 1 January 2001, GMT). Use a negative argument to specify a date and time before the reference date.

See also: – **initWithTimeIntervalSinceReferenceDate:**

distantFuture

+ (id)**distantFuture**

Creates and returns an NSDate that represents a date in the distant future (in terms of centuries).

You can pass this value where an NSDate is required to have the date argument essentially ignored. For example, the NSWindow method **nextEventMatchingMask:untilDate:inMode:dequeue:** returns **nil** if an event specified in the event mask does not happen before the specified date. You can use the object returned by **distantFuture** as the date argument to wait indefinitely for the event to occur.

```
myEvent = [myWindow nextEventMatchingMask:myEventMask
    untilDate:[NSDate distantFuture]
    inMode:NSDefaultRunLoopMode
    dequeue:YES];
```

See also: + **distantPast**

distantPast

+ (id)**distantPast**

Creates and returns an NSDate that represents a date in the distant past (in terms of centuries). You can use this object in your code as a control date, a guaranteed temporal boundary.

See also: + **distantFuture**

timeIntervalSinceReferenceDate

+ (NSTimeInterval)**timeIntervalSinceReferenceDate**

Returns the interval between the system's absolute reference date (the first instance of 1 January 2001, GMT) and the current date and time. Currently, this value is a negative number.

See also: – **timeIntervalSinceReferenceDate**, – **timeIntervalSinceDate:**, – **timeIntervalSince1970**, – **timeIntervalSinceNow**

Instance Methods

addTimeInterval:

– (id)**addTimeInterval:**(NSTimeInterval)*seconds*

Returns an NSDate object that is set to a specified number of seconds relative to the receiver. Use a negative value for *seconds* to have the returned object specify a date before the receiving object. The date returned might have a representation different from the receiver's.

See also: – **initWithTimeInterval:sinceDate:**, – **timeIntervalSinceDate:**

compare:

– (NSComparisonResult)**compare:**(NSDate *)*anotherDate*

Compares the receiving date to *anotherDate*, using **timeIntervalSinceDate:**, and returns a value of type NSComparisonResult. If the two dates are exactly equal to each other, this method returns NSOrderedSame. If the receiving object in the comparison is more recent than *anotherDate*, the method returns NSOrderedDescending. If it is older, it returns NSOrderedAscending.

This method detects subsecond differences between dates. If you want to compare dates with a less fine granularity, use **timeIntervalSinceDate:** to compare the two dates or use NSCalendarDate objects instead.

See also: – **earlierDate:**, – **isEqual**: (NSObject protocol), – **laterDate:**

dateWithCalendarFormat:timeZone:

– (NSCalendarDate *)**dateWithCalendarFormat**:(NSString *)*formatString* **timeZone**:
(NSTimeZone *)*timeZone*

Converts the NSDate to an NSCalendarDate object bound to *formatString* and the time zone *timeZone*. If you specify **nil** for either or both of these arguments, the default format string and time zone are assumed. (The default time zone is the one specific to the current locale; the default format string, which is "%Y-%m-%d %H:%M:%S %z", conforms to the international format YYYY-MM-DD HH:MM:SS ±HHMM.)

The conversion specifiers for *formatString* cover a range of date conventions:

Specifier	Description
%%	a '%' character
%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%c	shorthand for %X %x, the locale format for date and time
%d	day of the month as a decimal number (01-31)
%e	same as %d but does not print the leading 0 for days 1 through 9
%F	milliseconds as a decimal number (000 - 999)
%H	hour based on a 24-hour clock as a decimal number (00-23)
%I	hour based on a 12-hour clock as a decimal number (01-12)
%j	day of the year as a decimal number (001-366)
%m	month as a decimal number (01-12)
%M	minute as a decimal number (00-59)
%p	AM/PM designation for the locale
%S	second as a decimal number (00-61)
%w	weekday as a decimal number (0-6), where Sunday is 0
%x	date using the date representation for the locale
%X	time using the time representation for the locale
%y	year without century (00-99)
%Y	year with century (such as 1990)
%Z	time zone abbreviation (such as PDT)
%z	time zone offset in hours and minutes from GMT (HHMM)

See also: – **description**, – **descriptionWithCalendarFormat:timeZone:locale:**, – **descriptionWithLocale:**, + **dateWithString:calendarFormat:** (NSCalendarDate)

description

– (NSString *)**description**

Returns an NSString representing the NSDate object and conforming to the international format YYYY-MM-DD HH:MM:SS ±HHMM, where ±HHMM represents the time zone offset in hours and minutes from GMT (for example, “1994-05-23 10:45:32 +0600”).

See also: – **dateWithCalendarFormat:timeZone:**, – **descriptionWithCalendarFormat:timeZone:locale:**, – **descriptionWithLocale:**, – **description** (NSCalendarDate)

descriptionWithCalendarFormat:timeZone:locale:

– (NSString *)**descriptionWithCalendarFormat:**(NSString *)*formatString* **timeZone:**(NSTimeZone *)*aTimeZone*
locale:(NSDictionary *)*localeDictionary*

Returns an NSString representing the NSDate, formatted as specified by the conversion specifiers in *formatString* (see the method description for **dateWithCalendarFormat:timeZone:** for a list of these). Specify the time zone for the date in *aTimeZone* and specify keys and values representing the locale in *localeDictionary*. (See the class description for a list of the appropriate locale dictionary keys.) If you specify **nil** for one of these arguments, its default value is assumed. (The default time zone is the one specific to the current locale; the default format string, which is “%Y-%m-%d %H:%M:%S %z”, conforms to the international format YYYY-MM-DD HH:MM:SS ±HHMM.)

You could use this method to print the current time as follows:

```
sprintf(aString, "The current time is %s\n", [[[NSDate date]
    descriptionWithCalendarFormat:@"%H:%M:%S %Z" timeZone:nil
    locale:nil] cString]);
```

See also: – **description**, – **descriptionWithCalendarFormat:locale:** (NSCalendarDate), – **descriptionWithLocale:**

descriptionWithLocale:

– (NSString *)**descriptionWithLocale:**(NSDictionary *)*localeDictionary*

Returns an NSString representing the NSDate that conforms to the international format YYYY-MM-DD HH:MM:SS ±HHMM, where ±HHMM represents the time zone offset in hours and minutes from GMT

(for example, “1994-05-23 10:45:32 +0600”). Included are the keys and values that represent the locale data from *localeDictionary* (see the class description).

See also: – **description**, – **descriptionWithCalendarFormat:timeZone:locale:**, – **descriptionWithLocale:** (NSCalendarDate)

earlierDate:

– (NSDate *)**earlierDate:**(NSDate *)*anotherDate*

Compares the receiver date to *anotherDate*, using **timeIntervalSinceDate:**, and returns the earlier of the two.

See also: – **compare:**, – **isEqual**: (NSObject protocol), – **laterDate:**

init

– (id)**init**

When sent to the object returned by **alloc** or **allocWithZone:**, this method returns an NSDate initialized to the current date and time. This method uses the designated initializer, **initWithTimeIntervalSinceReferenceDate:**.

See also: + **date**, – **initWithTimeIntervalSinceReferenceDate:**

initWithString:

– (id)**initWithString:**(NSString *)*description*

Returns an NSDate with a date and time value specified by the international string-representation format: YYYY-MM-DD HH:MM:SS ±HHMM, where ±HHMM is a time zone offset in hours and minutes from GMT (for example, “1994-05-23 10:45:32 +0600”). You must specify all fields of the format, including the time zone offset, which must have a plus- or minus-sign prefix. This method uses the designated initializer, **initWithTimeIntervalSinceReferenceDate:**.

See also: + **dateWithString:**, – **description**, – **initWithTimeIntervalSinceReferenceDate:**

initWithTimeInterval:sinceDate:

– (id)**initWithTimeInterval:**(NSTimeInterval)*seconds* **sinceDate:**(NSDate *)*anotherDate*

Returns an NSDate initialized relative to another NSDate by a specified number of *seconds* (plus or minus). This method uses the designated initializer, **initWithTimeIntervalSinceReferenceDate:**.

See also: – **initWithTimeIntervalSinceReferenceDate:**

initWithTimeIntervalSinceNow:

– (id)**initWithTimeIntervalSinceNow:**(NSTimeInterval)*seconds*

Returns an NSDate initialized relative to the current date and time by the specified number of *seconds* (plus or minus). This method uses the designated initializer, **initWithTimeIntervalSinceReferenceDate:**.

See also: + **dateWithTimeIntervalSinceNow:**, – **initWithTimeIntervalSinceReferenceDate:**

initWithTimeIntervalSinceReferenceDate:

– (id)**initWithTimeIntervalSinceReferenceDate:**(NSTimeInterval)*seconds*

When sent to the object returned by **alloc** or **allocWithZone:**, this method returns an NSDate initialized relative to the absolute reference date (the first instant of 1 January 2001, GMT) by the specified number of seconds (plus or minus).

This method is the designated initializer for the NSDate class and is declared primarily for the use of subclasses of NSDate. When you subclass NSDate to create a concrete date class, you must override this method.

See also: + **dateWithTimeIntervalSinceReferenceDate:**

isEqualToDate:

– (BOOL)**isEqualToDate:**(NSDate *)*anotherDate*

Returns YES if the two objects compared are NSDate objects and are exactly equal to each other, NO if one of the objects is not of the NSDate class or if their date and time values differ. This method detects subsecond differences between dates. If you want to compare dates with a less fine granularity, either use **timeIntervalSinceDate:** to compare the two dates or use NSCalendarDate objects instead.

See also: **– compare:**, – **earlierDate**:, – **isEqual:** (NSObject protocol), – **laterDate:**

laterDate:

– (NSDate *)**laterDate:**(NSDate *)*anotherDate*

Compares the receiver to *anotherDate*, using **timeIntervalSinceDate:**, and returns the later of the two.

See also: – **compare:**, – **earlierDate:**, – **isEqual**: (NSObject protocol)

timeIntervalSinceDate:

– (NSTimeInterval)**timeIntervalSinceDate:**(NSDate *)*anotherDate*

Returns the interval between the receiver and *anotherDate*. If the receiver is earlier than *anotherDate*, the return value is negative.

See also: – **timeIntervalSince1970**, – **timeIntervalSinceNow**, – **timeIntervalSinceReferenceDate**

timeIntervalSince1970

– (NSTimeInterval)**timeIntervalSince1970**

Returns the interval between the receiver and the reference date 1 January 1970. If the receiver is earlier than 1 January 1970, the return value is negative.

See also: – **timeIntervalSinceDate:**, – **timeIntervalSinceNow**, – **timeIntervalSinceReferenceDate**

timeIntervalSinceNow

– (NSTimeInterval)**timeIntervalSinceNow**

Returns the interval between the receiver and the current date and time. If the receiver is earlier than the current date and time, the return value is negative.

See also: – **timeIntervalSinceDate:**, – **timeIntervalSince1970**, – **timeIntervalSinceReferenceDate**

timeIntervalSinceReferenceDate

– (NSTimeInterval)**timeIntervalSinceReferenceDate**

Returns the interval between the receiver and the system's absolute reference date, 1 January 2001, GMT. If the receiver is earlier than the absolute reference date, the return value is negative.

This is the primitive method for NSDate. If you subclass NSDate, you must override this method with your own implementation for it.

See also: – **timeIntervalSinceDate:**, – **timeIntervalSince1970**, – **timeIntervalSinceNow**, + **timeIntervalSinceReferenceDate**

NSDateFormatter

Inherits From: NSFormatter : NSObject

Conforms To: NSObject (NSObject)
NSCoding
NSCopying

Declared In: Foundation/NSDateFormatter.h

Class Description

Instances of NSDateFormatter format the textual representation of cells that contain NSDates (including NSCalendarDates), and convert textual representations of dates and times into NSDates. You can express the representation of dates and times very flexibly: “Thu 22 Dec 1994” is just as acceptable as “12/22/94”. With natural-language processing for dates enabled, users can also express dates colloquially, such as “today,” “day after tomorrow,” and “a month from today.”

To use an NSDateFormatter, allocate an instance of it and initialize it with **initWithDateFormat: allowNaturalLanguage:** In the first argument use **strftime**-style conversion specifiers to compose the format string for textual representation. (For more information on these specifiers, see the see the description of NSCalendarDate’s **dateWithCalendarFormat:timeZone:** method.) Then use NSCell's **setFormatter:** method to associate the NSDateFormatter object with a cell. The value of a cell (NSCell) is represented by an object, typically an NSDate object in this case. When this value needs to be displayed or edited, the cell passes its object to the NSDateFormatter instance, which returns the formatted string. When the user enters a string, or when one is programmatically written in a cell (using **setStringValue:**), the cell obtains the equivalent NSDate object from the NSDateFormatter.

NSControl provides delegation methods that permit you to validate cell contents and to handle errors in formatting. See the specification of the NSFormatter class for details.

When a cell with a NSDateFormatter is copied, the new cell retains the NSDateFormatter object instead of copying it. You remove an NSDateFormatter from a cell by specifying **nil** as the argument of **setFormatter:** .

Instances of NSDateFormatter are immutable.

Method Types

Initializing an NSDateFormatter

– initWithDateFormat:allowNaturalLanguage:

Determining attributes

– allowsNaturalLanguage
– dateFormat

Instance Methods

allowsNaturalLanguage

– (BOOL)**allowsNaturalLanguage**

Returns YES if the NSDateFormatter attempts to process dates entered as a vernacular string (“today,” “day before yesterday,” and so on). Returns NO if the NSDateFormatter does not do any natural-language processing of these date expressions.

dateFormat

– (NSString *)**dateFormat**

Returns the date format string used by an NSDateFormatter object. See the description of NSCalendarDate’s **dateWithCalendarFormat:timeZone:** for a list of the conversion specifiers permitted in date format strings.

initWithDateFormat:allowNaturalLanguage:

– (id)**initWithDateFormat:**(NSString *)*format*
allowNaturalLanguage:(BOOL)*flag*

Initializes and returns an NSDateFormatter instance that uses the date *format* in its conversions. See the description of NSCalendarDate’s **dateWithCalendarFormat:timeZone:** for a list of conversion specifiers permitted in date format strings. Set *flag* to YES if you want the NSDateFormatter to process dates entered as expressions in the vernacular (for example, “tomorrow”); NSDateFormatter attempts natural-language processing only after it fails to interpret an entered string according to *format*. The following example creates a date formatter with the format string (as example) “Mar 15 1994” and then associates the formatter with the cells of a form (**contactsForm**).

```
NSDateFormatter *dateFormat = [[NSDateFormatter alloc]
    initWithDateFormat:@"%b %d %Y” allowNaturalLanguage:NO];
[[contactsForm cells] makeObjectsPerform:@selector(setFormatter:)
     withObject:dateFormat]
```

NSDecimalNumber

Inherits From:	NSNumber : NSValue : NSObject
Conforms To:	NSDecimalNumberBehavior NSObject (NSObject)
Declared In:	Foundation/NSDecimalNumber.h

Class Description

NSDecimalNumber, an immutable subclass of NSNumber, provides an object-oriented wrapper for doing base-10 arithmetic. An instance can represent any number that can be expressed as *mantissa* **x** 10**exponent** where *mantissa* is a decimal integer up to 38 digits long, and *exponent* is an integer between -128 and 127.

In the course of doing arithmetic, a method may produce calculation errors, such as division by zero. It may also meet circumstances where it has a choice of ways to round a number off. The way the method acts on such occasions is called its "behavior."

Behavior is set by methods in the NSDecimalNumberBehaviors protocol. Every NSDecimalNumber argument called *behavior* requires an object that conforms to this protocol. For more on behaviors, see the specifications for the NSDecimalNumberBehaviors protocol and the NSDecimalNumberHandler class. Also see the **defaultBehavior** method description, below.

C Interface to Decimal Numbers

The arithmetic and rounding methods of NSDecimalNumber are also accessible through group of ordinary C functions, defined in NSDecimal.h. You might consider the C interface if you don't need to treat NSDecimalNumbers as objects—that is, if you don't need to store them in an object-oriented collection like an NSArray or NSDictionary.

You might also consider the C interface if you need maximum efficiency. The C interface is faster and uses less memory than the NSDecimalNumber class.

If you need mutability, you can combine the two interfaces. Use functions from the C interface and convert their results to NSDecimalNumbers.

The C functions—**NSDecimalCompact()**, **NSDecimalCompare()**, **NSDecimalRound()**, **NSDecimalNormalize()**, **NSDecimalAdd()**, **NSDecimalSubtract()**, **NSDecimalMultiply()**, **NSDecimalDivide()**, **NSDecimalPower()**, **NSDecimalMultiplyByPowerOf10()**, **NSDecimalString()**—are all documented in the "Functions" chapter of the *Foundation Framework Reference*.

Method Types

Creating an NSDecimalNumber

+ decimalNumberWithDecimal:
+ decimalNumberWithMantissa:exponent:isNegative:
+ decimalNumberWithString:
+ decimalNumberWithString:locale:
+ one
+ zero
+ notANumber

Initializing an NSDecimalNumber

– initWithDecimal:
– initWithMantissa:exponent:isNegative:
– initWithString:
– initWithString:locale:

Doing arithmetic

– decimalNumberByAdding:
– decimalNumberBySubtracting:
– decimalNumberByMultiplyingBy:
– decimalNumberByDividingBy:
– decimalNumberByRaisingToPower:
– decimalNumberByMultiplyingByPowerOf10:
– decimalNumberByAdding:withBehavior:
– decimalNumberBySubtracting:withBehavior:
– decimalNumberByMultiplyingBy:withBehavior:
– decimalNumberByDividingBy:withBehavior:
– decimalNumberByRaisingToPower:withBehavior:
– decimalNumberByMultiplyingByPowerOf10:withBehavior:

Rounding off

– decimalNumberByRoundingAccordingToBehavior:

Accessing data

– decimalValue
– doubleValue
– descriptionWithLocale:
– objCType

Modifying behavior

+ defaultBehavior
+ setDefaultBehavior:

Comparing NSDecimalNumbers

– compare:

Getting the highest and lowest possible NSDecimalNumbers

+ maximumDecimalNumber
+ minimumDecimalNumber

Class Methods

decimalNumberWithDecimal:

+ (NSDecimalNumber *)**decimalNumberWithDecimal:**(NSDecimal)*decimal*

Creates and returns an NSDecimalNumber equivalent to *decimal*.

decimal is an NSDecimal structure, which you can initialize by hand, or generate using the **scanDecimal:** method from the NSDecimalNumberScanning category of NSScanner, defined in NSDecimalNumber.h.

decimalNumberWithMantissa:exponent:isNegative:

+ (NSDecimalNumber *)**decimalNumberWithMantissa:**(unsigned long long)*mantissa* **exponent:** (short)*exponent* **isNegative:**(BOOL)*isNegative*

Creates and returns an NSDecimalNumber equivalent to the number specified by the arguments.

The arguments express a number in a kind of scientific notation that requires the mantissa to be an integer. So, for example, if the number to be represented is 1.23, it is expressed as 123x10**-2**—*mantissa* is 123, *exponent* is -2, and *isNegative*, which refers to the sign of the mantissa, is NO.

decimalNumberWithString:

+ (NSDecimalNumber *)**decimalNumberWithString:**(NSString *)*numericString*

Creates and returns an NSDecimalNumber equivalent to *numericString.* Besides digits, *numericString* can include an initial “+” or “-,” a single “E” or “e”, to indicate the exponent of a number in scientific notation, and a single NSDecimalSeparator to divide the fractional from the integral part of the number.

Whether the NSDecimalSeparator is a period (as in the United States) or a comma (as in France) depends on the default locale.

See also: **+ decimalNumberWithString:locale:**

decimalNumberWithString:locale:

+ (NSDecimalNumber *)**decimalNumberWithString:**(NSString *)*numericString* **locale:** (NSDictionary *)*locale*

Creates and returns an NSDecimalNumber equivalent to *numericString.* Besides digits, *numericString* can include an initial “+” or “-,” a single “E” or “e”, to indicate the exponent of a number in scientific notation, and a single NSDecimalSeparator to divide the fractional from the integral part of the number.

locale determines whether the NSDecimalSeparator is a period (as in the United States) or a comma (as in France).

The following strings show examples of acceptable values for *numericString*:

- “2500.6” (or “2500,6”, depending on *locale*)
- “-2500.6” (or “-2500.6”)
- “-2.5006e3” (or “-2,5006e3”)
- “-2.5006E3” (or “-2,5006E3”)

The following are unacceptable:

- “2,500.6”
- “2500 3/5”
- “2.5006x10e3”
- “two thousand five hundred and six tenths”

See also: **+ decimalNumberWithString:**

defaultBehavior

+ (id <NSDecimalNumberBehaviors>)**defaultBehavior**

Returns the way that arithmetic methods, like **decimalNumberByAdding:**, round off and handle error conditions.

By default, the arithmetic methods do not round numbers off. They assume that your need for precision does not exceed 38 significant digits. And they raise exceptions when they try to divide by zero, or when they produce a number that is too big or small to be represented.

If this default behavior doesn’t suit your application, you should use methods that let you specify the behavior, like **decimalNumberByAdding:withBehavior.** If you find yourself using a particular behavior consistently, you can specify a different default behavior with **setDefaultBehavior:**.

maximumDecimalNumber

+ (NSDecimalNumber *)**maximumDecimalNumber**

Returns the largest possible NSDecimalNumber.

See also: + **minimumDecimalNumber**

minimumDecimalNumber

+ (NSDecimalNumber *)**minimumDecimalNumber**

Returns the smallest possible NSDecimalNumber.

See also: + **maximumDecimalNumber**

notANumber

+ (NSDecimalNumber *)**notANumber**

Creates and returns an NSDecimalNumber that specifies no number. Any arithmetic method receiving **notANumber** as an argument returns **notANumber**.

This value can be a useful way of handling non-numeric data in an input file. It can also be a useful response to calculation errors. For more information on calculation errors, see the **exceptionDuringOperation: error:leftOperand:rightOperand:** method description in the NSDecimalNumberBehaviors protocol specification.

one

+ (NSDecimalNumber *)**one**

Creates and returns an NSDecimalNumber equivalent to the number 1.0.

See also: **+ zero**

setDefaultBehavior:

+ (void)**setDefaultBehavior:**(id <NSDecimalNumberBehaviors>)*behavior*

Specifies the way that arithmetic methods, like **decimalNumberByAdding:**, round off and handle error conditions. *behavior* must conform to the NSDecimalNumberBehaviors protocol.

zero

+ (NSDecimalNumber *)**zero**

Returns a newly allocated NSDecimalNumber equivalent to the number 0.0.

See also: **+ one**

Instance Methods

compare:

– (NSComparisonResult)**compare:**(NSNumber *)*decimalNumber*

Compares *decimalNumber* with the receiver. Returns NSOrderedAscending if *decimalNumber*'s value is greater than the receiver's, NSOrderedSame if they're equal, and NSOrderedDescending if *decimalNumber*'s value is less than the receiver's.

decimalNumberByAdding:

– (NSDecimalNumber *)**decimalNumberByAdding:**(NSDecimalNumber *)*decimalNumber*

Adds *decimalNumber* to the receiver, and returns the sum, a newly created NSDecimalNumber. This method uses the default behavior when handling calculation errors and rounding.

See also: **– decimalNumberByAdding:withBehavior:**, **+ defaultBehavior**

decimalNumberByAdding:withBehavior:

– (NSDecimalNumber *)**decimalNumberByAdding:**(NSDecimalNumber *)*decimalNumber*
withBehavior:(id <NSDecimalNumberBehaviors>)*behavior*

Adds *decimalNumber* to the receiver, and returns the sum, a newly created NSDecimalNumber. *behavior* specifies the handling of calculation errors and rounding.

decimalNumberByDividingBy:

– (NSDecimalNumber *)**decimalNumberByDividingBy:**(NSDecimalNumber *)*decimalNumber*

Divides the receiver by *decimalNumber*, and returns the quotient, a newly created NSDecimalNumber. This method uses the default behavior when handling calculation errors and rounding.

See also: **– decimalNumberByDividingBy:withBehavior:**, **+ defaultBehavior**

decimalNumberByDividingBy:withBehavior:

– (NSDecimalNumber *)**decimalNumberByDividingBy:**(NSDecimalNumber *)*decimalNumber* **withBehavior:**(id <NSDecimalNumberBehaviors>)*behavior*

Divides the receiver by *decimalNumber*, and returns the quotient, a newly created NSDecimalNumber. *behavior* specifies the handling of calculation errors and rounding.

decimalNumberByMultiplyingBy:

– (NSDecimalNumber *)**decimalNumberByMultiplyingBy:**(NSDecimalNumber *)*decimalNumber*

Multiplies the receiver by *decimalNumber*, and returns the product, a newly created NSDecimalNumber. This method uses the default behavior when handling calculation errors and when rounding.

See also: **– decimalNumberByMultiplyingBy:withBehavior:**, **+ defaultBehavior**

decimalNumberByMultiplyingBy:withBehavior:

– (NSDecimalNumber *)**decimalNumberByMultiplyingBy:**(NSDecimalNumber *)*decimalNumber* **withBehavior:**(id <NSDecimalNumberBehaviors>)*behavior*

Multiplies the receiver by *decimalNumber*, and returns the product, a newly created NSDecimalNumber. *behavior* specifies the handling of calculation errors and rounding.

decimalNumberByMultiplyingByPowerOf10:

– (NSDecimalNumber *)**decimalNumberByMultiplyingByPowerOf10:**(short)*power*

Multiplies the receiver by 10**power**, and returns the product, a newly created NSDecimalNumber. This method uses the default behavior when handling calculation errors and when rounding.

See also: **– decimalNumberByMultiplyingByPowerOf10:withBehavior:**, **+ defaultBehavior**

decimalNumberByMultiplyingByPowerOf10:withBehavior:

– (NSDecimalNumber *)**decimalNumberByMultiplyingByPowerOf10:**(short)*power* **withBehavior:** (id <NSDecimalNumberBehaviors>)*behavior*

Multiplies the receiver by 10**power**, and returns the product, a newly created NSDecimalNumber. *behavior* specifies the handling of calculation errors and rounding.

decimalNumberByRaisingToPower:

– (NSDecimalNumber *)**decimalNumberByRaisingToPower:**(unsigned)*power*

Raises the receiver to *power*, and returns the result, a newly created NSDecimalNumber. This method uses the default behavior when handling calculation errors and when rounding.

See also: – decimalNumberByRaisingToPower:withBehavior:, **+ defaultBehavior**

decimalNumberByRaisingToPower:withBehavior:

– (NSDecimalNumber *)**decimalNumberByRaisingToPower:**(unsigned)*power* **withBehavior:**
(id <NSDecimalNumberBehaviors>)*behavior*

Raises the receiver to *power*, and returns the result, a newly created NSDecimalNumber. *behavior* specifies the handling of calculation errors and rounding.

decimalNumberByRoundingAccordingToBehavior:

– (NSDecimalNumber *)**decimalNumberByRoundingAccordingToBehavior:**
(id <NSDecimalNumberBehaviors>)*behavior*

Rounds the receiver off in the way specified by behavior, and returns the result, a newly created NSDecimalNumber. For a description of the different ways of rounding, see the **roundingMode** method in the NSDecimalNumberHandler class specification.

decimalNumberBySubtracting:

– (NSDecimalNumber *)**decimalNumberBySubtracting:**(NSDecimalNumber *)*decimalNumber*

Subtracts *decimalNumber* from the receiver, and returns the difference, a newly created NSDecimalNumber. This method uses the default behavior when handling calculation errors and when rounding.

See also: – decimalNumberBySubtracting:withBehavior:, **+ defaultBehavior**

decimalNumberBySubtracting:withBehavior:

– (NSDecimalNumber *)**decimalNumberBySubtracting:**(NSDecimalNumber *)*decimalNumber*
withBehavior:(id <NSDecimalNumberBehaviors>)*behavior*

Subtracts *decimalNumber* from the receiver, and returns the difference, a newly created NSDecimalNumber. *behavior* specifies the handling of calculation errors and rounding.

decimalValue

– (NSDecimal)**decimalValue**

Returns the receiver's value, expressed as an NSDecimal structure.

descriptionWithLocale:

– (NSString *)**descriptionWithLocale:**(NSDictionary *)*locale*

Returns an NSString that represents the contents of the receiver. Whether the separator is a period or a comma depends on *locale*.

doubleValue

– (double)**doubleValue**

Returns the approximate **double** value of the receiver.

initWithDecimal:

– (NSDecimalNumber *)**initWithDecimal:**(NSDecimal)*decimal*

Returns an NSDecimalNumber initialized to represent *decimal*. This method is NSDecimalNumber's designated initializer.

initWithMantissa:exponent:isNegative:

– (NSDecimalNumber *)**initWithMantissa:**(unsigned long long)*mantissa* **exponent:**(short)*exponent* **isNegative:**(BOOL)*isNegative*

Creates and returns an NSDecimalNumber equivalent to the number specified by the arguments.

The arguments express a number in a type of scientific notation that requires the mantissa to be an integer. So, for example, if the number to be represented is 1.23, it is expressed as 123x10**-2**—*mantissa* is 123, *exponent* is -2, and *isNegative*, which refers to the sign of the mantissa, is NO.

initWithString:

– (NSDecimalNumber *)**initWithString:**(NSString *)*numericString*

Returns an NSDecimalNumber equivalent to *numericString. numericString* must be a simple string of digits, possibly including a decimal separator. For a listing of acceptable and unacceptable strings, see the class method **decimalNumberWithString:locale:**.

initWithString:locale:

– (NSDecimalNumber *)**initWithString:**(NSString *)*numericString* **locale:**(NSDictionary *)*locale*

Returns a newly created NSDecimalNumber equivalent to *numericString*. The interpretation of *numericString* depends on *locale*.

See also: **+ decimalNumberWithString:locale:**

objCType

– (const char *)**objCType**

Returns a C string containing the Objective-C type of the data contained in the receiver, which for an NSDecimalNumber is always “d” (for **double**).

NSDecimalNumberHandler

Inherits From: NSObject

Conforms To: NSDecimalNumberBehaviors
NSObject (NSObject)

Declared In: Foundation/NSDecimalNumber.h

Class Description

NSDecimalNumberHandler is a class that adopts the NSDecimalNumberBehaviors protocol. This class allows you to set the way an NSDecimalNumber rounds off and handles errors, without having to create a custom class.

You can use an instance of this class as an argument to any of the NSDecimalNumber methods that end with the word **...Behavior:**. If you don't think you need special behavior, you probably don't need this class. The odds are that NSDecimalNumber's default behavior will suit your needs.

For more information, see the NSDecimalNumberBehaviors protocol specification.

Adopted Protocols

NSDecimalNumberBehaviors

– roundingMode
– scale
– exceptionDuringOperation:error:leftOperand:rightOperand:

Method Types

Creating and initializing an NSDecimalNumberHandler

+ defaultDecimalNumberHandler
+ decimalNumberHandlerWithRoundingMode:scale:
raiseOnExactness:raiseOnOverflow:raiseOnUnderflow:
raiseOnDivideByZero:

Initializing an already-created NSDecimalNumberHandler

– initWithRoundingMode:scale:raiseOnExactness:raiseOnOverflow:
raiseOnUnderflow:raiseOnDivideByZero:

Class Methods

defaultDecimalNumberHandler

+ (id)**defaultDecimalNumberHandler**

Creates and returns the default instance of NSDecimalNumberHandler.

This instance does not round numbers off. It assumes that your need for precision does not exceed 38 significant digits. And it raises an exception when its NSDecimalNumber tries to divide by zero, or when its NSDecimalNumber produces a number that is too big or small to be represented.

decimalNumberHandlerWithRoundingMode:scale:raiseOnExactness: raiseOnOverflow:raiseOnUnderflow:raiseOnDivideByZero:

+ (id)**decimalNumberHandlerWithRoundingMode:**(NSRoundingMode)*roundingMode*
scale:(short)*scale*
raiseOnExactness:(BOOL)*raiseOnExactness*
raiseOnOverflow:(BOOL)*raiseOnOverflow*
raiseOnUnderflow:(BOOL)*raiseOnUnderflow*
raiseOnDivideByZero:(BOOL)*raiseOnDivideByZero*

Creates and returns an NSDecimalNumberHandler with customized behavior.

scale and *roundingMode* affect the way the NSDecimalNumberHandler's NSDecimalNumber rounds off its return value. *scale* sets the number of digits a rounded value should have after its decimal point. *roundingMode* sets the rule by which that NSDecimalNumbers are rounded off; *roundingMode* has four possiblevalues: NSRoundUp, NSRoundDown, NSRoundPlain, and NSRoundBankers.

The *raiseOn...* arguments determine whether the NSDecimalNumberHandler will raise an exception when its NSDecimalNumber notifies it of a certain kind of calculation error. If a *raiseOn...* argument is YES, NSDecimalNumberHandler will raise an exception; if a *raiseOn..* argument is NO, NSDecimalNumber will ignore the error, and return control to the calling method.

See the NSDecimalNumberBehaviors protocol spectification for a complete explanation of these possible behaviors.

Instance Methods

initWithRoundingMode:scale:raiseOnExactness:raiseOnOverflow: raiseOnUnderflow:raiseOnDivideByZero:

- (id)**initWithRoundingMode:**(NSRoundingMode)*roundingMode* **scale:**(short)*scale*
  raiseOnExactness:(BOOL)*raiseOnExactness*
  raiseOnOverflow:(BOOL)*raiseOnOverflow*
  raiseOnUnderflow:(BOOL)*raiseOnUnderflow*
  raiseOnDivideByZero:(BOOL)*raiseOnDivideByZero*

Initializes a previously-allocated NSDecimalNumberHandler so that it behaves as specified by the method's arguments. For an explanation of these arguments, see the **decimalNumberHandlerWithRoundingMode:scale:raiseOnExactness:raiseOnOverflow: raiseOnUnderflow:raiseOnDivideByZero:** class method.

Returns **self**.

NSDeserializer

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	Foundation/NSSerialization.h

Class Description

The NSDeserializer class declares methods that convert an abstract representation of a property list (as contained in an NSData object) into a graph of property list objects in memory. The NSDeserializer class object itself provides these methods; you don't create instances of NSDeserializer. Options to these methods allow you to specify that container objects (arrays or dictionaries) in the resulting graph be mutable or immutable; that deserialization begin at the start of the data or from some position within it; or that deserialization occur lazily, so that a property list is deserialized only if it is actually going to be accessed. See the NSSerializer specification for more information on serialization.

Class Methods

deserializePropertyListFromData:atCursor:mutableContainers:

+ (id)**deserializePropertyListFromData:**(NSData *)*data*
  atCursor:(unsigned int *)*cursor*
  mutableContainers:(BOOL)*mutable*

Returns a property list object corresponding to the abstract representation in *data* at the location *cursor*. If *mutable* is YES and the object is a dictionary or an array, the re-composed object is made mutable. Returns **nil** if the object is not a valid one for property lists.

deserializePropertyListFromData:mutableContainers:

+ (id)**deserializePropertyListFromData:**(NSData *)*serialization* **mutableContainers:**(BOOL)*mutable*

Returns a property list object corresponding to the abstract representation in *serialization* or **nil** if *serialization* doesn't represent a property list. If *mutable* is YES and the object is a dictionary or an array, the re-composed object is made mutable.

deserializePropertyListLazilyFromData:atCursor:length:mutableContainers:

+ (id)**deserializePropertyListLazilyFromData:**(NSData *)*data*
  atCursor:(unsigned int *)*cursor*
  length:(unsigned int)*length*
  mutableContainers:(BOOL)*mutable*

Returns a property list from *data* at location cursor or **nil** if *data* doesn't represent a property list. The deserialization proceeds lazily. That is, if the data at *cursor* has a length greater than *length*, a proxy is substituted for the actual property list as long as the constituent objects of that property list are not being accessed. If *mutable* is YES and the object is a dictionary or an array, the re-composed object is made mutable.

NSDictionary Class Cluster

Class Cluster Description

The NSDictionary and NSMutableDictionary classes declare the programmatic interface for objects that store associations of keys and values. You use these classes when you need a convenient and efficient way to retrieve data associated with an arbitrary key.

Because of the nature of class clusters, the objects you create with this interface are not actual instances of the NSDictionary or NSMutableDictionary classes. Rather, the instances belong to one of their private subclasses. (For convenience, we use the term *dictionary* to refer to any one of these instances without specifying its exact class membership.) Although a dictionary's class is private, its interface is public, as declared by these abstract superclasses, NSDictionary and NSMutableDictionary.

A key-value pair within a dictionary is called an *entry*. Each entry consists of one object that represents the key, and a second object which is that key's value. Within a dictionary, the keys are unique. That is, no two keys in a single dictionary are equal (as determined by **isEqual:**).

You establish an immutable dictionary's entries when it's created, and thereafter the entries can't be modified. A mutable dictionary allows the addition and deletion of entries at any time, automatically allocating memory as needed.

Internally, a dictionary uses a hash table to organize its storage and to provide rapid access to a value given the corresponding key. However, the methods defined in this cluster insulate you from the complexities of working with hash tables, hashing functions, or the hashed value of keys. The methods described below take keys directly, not their hashed form.

Generally, you create a temporary dictionary by sending one of the **dictionary...** messages to either the NSDictionary or NSMutableDictionary class object. These methods return a dictionary containing the associations specified as arguments to the method. Methods that add entries to dictionaries—whether as part of initialization (for all dictionaries) or during modification (for mutable dictionaries)—copy each key argument (keys must conform to the NSCopying protocol) and add the copies to the dictionary. Each corresponding value object receives a **retain** message to ensure that it won't be deallocated before the dictionary is through with it.

The dictionary classes adopt the NSCopying and NSMutableCopying protocols, making it convenient to convert a dictionary of one type to the other.

NSDictionary

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSMutableCopying NSObject (NSObject)
Declared In:	Foundation/NSDictionary.h

Class at a Glance

Purpose

An NSDictionary object stores an immutable set of entries.

Principal Attributes

- A count of the number of entries in the dictionary.
- The set of keys contained in the dictionary.
- The objects that correspond to the keys in the dictionary.

Creation

+ dictionary	Returns an empty dictionary.
+ dictionaryWithContentsOfFile:	Returns a dictionary initialized from the property list stored in the specified file.
+ dictionaryWithDictionary:	Returns a dictionary initialized from an existing dictionary.
+ dictionaryWithObject:forKey:	Returns a dictionary initialized with a single object and key.
+ dictionaryWithObjects:forKeys:	Returns a dictionary of the specified objects and their keys.
+ dictionaryWithObjects:forKeys:count:	Returns a dictionary of the specified objects and their keys.
+ dictionaryWithObjectsAndKeys:	Returns a dictionary of the specified objects and their keys.

Commonly Used Methods

– count	Returns the number of objects currently in the dictionary.
– objectForKey:	Returns the object that corresponds to the specified key.
– keyEnumerator	Returns an enumerator object that lets you access each key in the dictionary.

Primitive Methods

– count
– objectForKey:
– keyEnumerator

Class Description

The NSDictionary class declares the programmatic interface to objects that manage immutable associations of keys and values. NSDictionary's three primitive methods—**count**, **objectForKey:**, and **keyEnumerator**—provide the basis for all of the other methods in its interface. The **count** method returns the number of entries in the dictionary. **objectForKey:** returns the value associated with a given key. **keyEnumerator** returns an object that lets you iterate through each of the keys in the dictionary.

The other methods declared here operate by invoking one or more of these primitives. The non-primitive methods provide convenient ways of accessing multiple entries at once. The **description...** and **writeToFile:atomically:** methods cause a dictionary to write a representation of itself to a string or to a file, respectively.

Adopted Protocols

NSCoding	
	– encodeWithCoder:
	– initWithCoder:
NSCopying	
	– copyWithZone:
NSMutableCopying	
	– mutableCopyWithZone:

Method Types

Creating dictionaries

+ allocWithZone:
+ dictionary
+ dictionaryWithContentsOfFile:
+ dictionaryWithDictionary:
+ dictionaryWithObject:forKey:
+ dictionaryWithObjects:forKeys:
+ dictionaryWithObjects:forKeys:count:
+ dictionaryWithObjectsAndKeys:
– initWithContentsOfFile:
– initWithDictionary:
– initWithObjects:forKeys:
– initWithObjects:forKeys:count:
– initWithObjectsAndKeys:

Counting entries

– count

Accessing keys and values

– allKeys
– allKeysForObject:
– allValues
– description
– descriptionInStringsFileFormat
– descriptionWithLocale:
– descriptionWithLocale:indent:
– keyEnumerator
– keysSortedByValueUsingSelector:
– objectEnumerator
– objectForKey:
– objectsForKeys:notFoundMarker:

Comparing dictionaries

– isEqualToDictionary:

Storing dictionaries

– writeToFile:atomically:

Accessing file attributes

– fileGroupOwnerAccountName
– fileModificationDate
– fileOwnerAccountName
– filePosixPermissions
– fileSize
– fileSystemFileNumber
– fileSystemNumber
– fileType

Class Methods

allocWithZone:

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

Creates and returns an uninitialized dictionary in the specified zone. If the receiver is the NSDictionary class object, an instance of an immutable private subclass is returned; otherwise, an object of the receiver's class is returned.

Typically, you create temporary dictionaries using the **dictionary...** class methods, not the **alloc...** and **init...** methods.

dictionary

+ (id)**dictionary**

Creates and returns an empty dictionary. This method is declared primarily for the use with mutable subclasses of NSDictionary.

Note: If you don't want a temporary object, you can also create an empty dictionary using **alloc...** and **init**.

dictionaryWithContentsOfFile:

+ (id)**dictionaryWithContentsOfFile:**(NSString *)*path*

Allocates and initializes a dictionary using the keys and values found in *path*. *path* can be a full or relative pathname. The file identified by *path* must contain a string representation of a property list whose root object is a dictionary. The dictionary must contain only property list objects (NSString, NSData, NSArray, or NSDictionary objects).

This method returns **nil** if there is a file error or if the contents of the file are an invalid representation of a dictionary.

See also: **– initWithContentsOfFile:**

dictionaryWithDictionary:

+ (id)**dictionaryWithDictionary:**(NSDictionary *)*otherDictionary*

Creates and returns a dictionary containing the keys and values found in *otherDictionary*.

See also: **– initWithDictionary:**

dictionaryWithObject:forKey:

+ (id)**dictionaryWithObject:**(id)*anObject* **forKey:**(id)*aKey*

Creates and returns a dictionary containing a single object, *anObject*, for a single key, *aKey*.

See also: + **dictionaryWithObjects:forKeys:**, + **dictionaryWithObjects:forKeys:count:**, + **dictionaryWithObjectsAndKeys:**

dictionaryWithObjects:forKeys:

+ (id)**dictionaryWithObjects:**(NSArray *)*objects* **forKeys:**(NSArray *)*keys*

Creates and returns a dictionary containing entries constructed from the contents of *objects* and *keys*. This method steps through the *objects* and *keys* arrays, creating entries in the new dictionary as it goes. Each value object receives a **retain** message before being added to the dictionary. In contrast, each key is copied (using **copyWithZone:**; keys must conform to the NSCopying protocol), and the copy is added to the dictionary. An NSInvalidArgumentException is raised if *objects* and *keys* don't have the same number of elements.

See also: – **initWithObjects:forKeys:**, + **dictionaryWithObject:forKey:**, + **dictionaryWithObjects:forKeys:count:**, + **dictionaryWithObjectsAndKeys:**

dictionaryWithObjects:forKeys:count:

+ (id)**dictionaryWithObjects:**(id *)*objects*
forKeys:(id *)*keys*
count:(unsigned int)*count*

Creates and returns a dictionary containing *count* objects from the *objects* array. The objects are associated with keys taken from the *keys* array. For example, this code excerpt creates a dictionary that associates the alphabetic characters with their ASCII values:

```
static const int N_ENTRIES = 26;
NSDictionary *asciiDict;
NSString *keyArray[N_ENTRIES];
NSNumber *valueArray[N_ENTRIES];
int i;

for (i = 0; i < N_ENTRIES; i++) {
    char charValue = 'a' + i;
    keyArray[i] = [NSString stringWithFormat:@"%c", charValue];
    valueArray[i] = [NSNumber numberWithChar:charValue];
}

asciiDict = [NSDictionary dictionaryWithObjects:(id *)valueArray
    forKeys:(id *)keyArray count:N_ENTRIES];
```

See also: – **initWithObjects:forKeys:count:**, + **dictionaryWithObject:forKey:**, + **dictionaryWithObjects:forKeys:**, + **dictionaryWithObjectsAndKeys:**

dictionaryWithObjectsAndKeys:

+ (id)**dictionaryWithObjectsAndKeys:**(id)*object***,** (id)*key*, ...

Creates and returns a dictionary containing entries constructed from the specified set of objects and keys. **initWithObjectsAndKeys:** takes a variable number of arguments: a null-terminated list of alternating objects and keys. If any key is **nil**, an NSInvalidArgumentException is raised.

This method is similar to **initWithObjects:forKeys:**, differing only in the way key-value pairs are specified.

See also: – **initWithObjectsAndKeys:**, + **dictionaryWithObject:forKey:**, + **dictionaryWithObjects:forKeys:**, + **dictionaryWithObjects:forKeys:count:**

Instance Methods

allKeys

– (NSArray *)**allKeys**

Returns a new array containing the dictionary's keys or an empty array if the dictionary has no entries. The order of the elements in the array isn't defined.

See also: – **allValues**, – **allKeysForObject:**

allKeysForObject:

– (NSArray *)**allKeysForObject:**(id)*anObject*

Finds all occurrences of the value *anObject* in the dictionary and returns a new array with the corresponding keys. Each object in the dictionary is sent an **isEqual:** message to determine if it's equal to *anObject*. If no object matching *anObject* is found, this method returns **nil**.

See also: – **allKeys**, – **keyEnumerator**

allValues

– (NSArray *)**allValues**

Returns a new array containing the dictionary's values, or an empty array if the dictionary has no entries. The order of the values in the array isn't defined.

See also: – **allKeys**, – **objectEnumerator**

count

– (unsigned int)**count**

Returns the number of entries in the dictionary.

description

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

Returns a string that represents the contents of the receiver, formatted as a property list. If each key in the dictionary responds to **compare:**, the entries are listed in ascending order, by key. Otherwise, the order in which the entries are listed is undefined.

See also: – **descriptionWithLocale:**, – **descriptionWithLocale:indent:**

descriptionInStringsFileFormat

– (NSString *)**descriptionInStringsFileFormat**

Returns a string that represents the contents of the receiver, formatted in **.strings** file format. The order in which the entries are listed is undefined.

descriptionWithLocale:

– (NSString *)**descriptionWithLocale:**(NSDictionary *)*locale*

Returns a string object that represents the contents of the receiver, formatted as a property list. *locale* specifies options used for formatting each of the receiver's keys and values; specify **nil** if you don't want them formatted.

For a description of how *locale* is applied to each element in the receiver, see **descriptionWithLocale: indent:**.

If each key in the dictionary responds to **compare:**, the entries are listed in ascending order, by key. Otherwise, the order in which the entries are listed is undefined.

See also: – **description**, – **descriptionWithLocale:indent:**

descriptionWithLocale:indent:

– (NSString *)**descriptionWithLocale:**(NSDictionary *)*locale* **indent:**(unsigned int)*level*

Returns a string object that represents the contents of the receiver, formatted as a property list. Use *locale* to specify options to be passed to the methods that format each of the receiver's keys and values; specify **nil** if you don't want them formatted. *level* allows you to specify a level of indent, to make the output more readable: set level to 0 to use four spaces to indent, or 1 to indent the output with a tab character.

The returned NSString contains the string representations of each of the receiver's entries. **descriptionWithLocale:indent:** obtains the string representation of a given key or value as follows:

- If the object is an NSString, it is used as is.
- If the object responds to **descriptionWithLocale:indent:**, that method is invoked to obtain the object's string representation.
- If the object responds to **descriptionWithLocale:**, that method is invoked to obtain the object's string representation.
- If none of the above conditions are met, the object's string representation is obtained by invoking its **description** method.

If each key in the dictionary responds to **compare:**, the entries are listed in ascending order, by key. Otherwise, the order in which the entries are listed is undefined.

See also: – **description**, – **descriptionWithLocale:**

fileGroupOwnerAccountName

– (NSString *)fileGroupOwnerAccountName

Returns the object for the key NSFileGroupOwnerAccountName or **nil** if the receiver doesn't have an entry for the key.

This and the other **file...** methods are for use with a dictionary such as those returned from the methods **fileAttributesAtPath:traverseLink:** (NSFileManager), **directoryAttributes** (NSDirectoryEnumerator) and **fileAttributes** (NSDirectoryEnumerator) that represents the POSIX attributes of a file or directory. This method returns the name of the corresponding file's group.

fileModificationDate

– (NSDate *)fileModificationDate

Returns the object for the key NSFileModificationDate or **nil** if the receiver doesn't have an entry for the key.

This and the other **file...** methods are for use with a dictionary such as those returned from the methods **fileAttributesAtPath:traverseLink:** (NSFileManager), **directoryAttributes** (NSDirectoryEnumerator) and **fileAttributes** (NSDirectoryEnumerator) that represents the POSIX attributes of a file or directory. This method returns the date that the file's data was last modified.

fileOwnerAccountName

– (NSString *)fileOwnerAccountName

Returns the object for the key NSFileOwnerAccountName or **nil** if the receiver doesn't have an entry for the key.

This and the other **file...** methods are for use with a dictionary such as those returned from the methods **fileAttributesAtPath:traverseLink:** (NSFileManager), **directoryAttributes** (NSDirectoryEnumerator) and **fileAttributes** (NSDirectoryEnumerator) that represents the POSIX attributes of a file or directory. This method returns the account name of the file's owner.

filePosixPermissions

– (unsigned long)filePosixPermissions

Returns the object for the key NSFilePosixPermissions or **nil** if the receiver doesn't have an entry for the key.

This and the other **file...** methods are for use with a dictionary such as those returned from the methods **fileAttributesAtPath:traverseLink:** (NSFileManager), **directoryAttributes** (NSDirectoryEnumerator)

and **fileAttributes** (NSDirectoryEnumerator) that represents the POSIX attributes of a file or directory. This method returns the file's permissions.

fileSize

– (unsigned long long)fileSize

Returns the object for the key NSFileSize or **nil** if the receiver doesn't have an entry for the key.

This and the other **file...** methods are for use with a dictionary such as those returned from the methods **fileAttributesAtPath:traverseLink:** (NSFileManager), **directoryAttributes** (NSDirectoryEnumerator) and **fileAttributes** (NSDirectoryEnumerator) that represents the POSIX attributes of a file or directory. This method returns the file's size.

fileSystemFileNumber

– (unsigned long)fileSystemFileNumber

Returns the object for the key NSFileSystemFileNumber or **nil** if the receiver doesn't have an entry for the key.

This and the other **file...** methods are for use with a dictionary such as those returned from the methods **fileAttributesAtPath:traverseLink:** (NSFileManager), **directoryAttributes** (NSDirectoryEnumerator) and **fileAttributes** (NSDirectoryEnumerator) that represents the POSIX attributes of a file or directory. This method returns the file's inode.

This method is not implemented on Windows platforms.

fileSystemNumber

– (unsigned long)fileSystemNumber

Returns the object for the key NSFileSystemNumber or **nil** if the receiver doesn't have an entry for the key.

This and the other **file...** methods are for use with a dictionary such as those returned from the methods **fileAttributesAtPath:traverseLink:** (NSFileManager), **directoryAttributes** (NSDirectoryEnumerator) and **fileAttributes** (NSDirectoryEnumerator) that represents the POSIX attributes of a file or directory. This method returns the ID of the device containing the file.

This method is not implemented on Windows platforms.

fileType

– (NSString *)fileType

Returns the object for the key NSFileType or **nil** if the receiver doesn't have an entry for the key.

This and the other **file...** methods are for use with a dictionary such as those returned from the methods **fileAttributesAtPath:traverseLink:** (NSFileManager), **directoryAttributes** (NSDirectoryEnumerator) and **fileAttributes** (NSDirectoryEnumerator) that represents the POSIX attributes of a file or directory. This method returns the file's type, which is one of the following:

String	Meaning
NSFileTypeUnknown	Unknown file type
NSFileTypeCharacterSpecial	Character special file
NSFileTypeDirectory	Directory
NSFileTypeBlockSpecial	Block special file
NSFileTypeRegular	Regular file
NSFileTypeSymbolicLink	Symbolic link
NSFileTypeSocket	Socket

hash

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

Returns an unsigned integer that can be used as a table address in a hash table structure. For a dictionary, **hash** returns the number of entries in the dictionary. If two dictionaries are equal (as determined by the **isEqual:** method), they will have the same hash value.

See also: – **isEqual:**

initWithContentsOfFile:

– (id)**initWithContentsOfFile:**(NSString *)*path*

Initializes a newly allocated dictionary using the keys and values found in *path*. *path* can be a full or relative pathname. The file identified by *path* must contain a string representation of a property list whose root object is a dictionary. The dictionary must contain only property list objects (NSString, NSData, NSArray, or NSDictionary objects).

After initializing the receiver, this method returns **self**. However, if the new instance can't be initialized (either because of a file error or because the contents of the file is an invalid representation of a dictionary), it's deallocated and **nil** is returned.

See also: + **dictionaryWithContentsOfFile:**

initWithDictionary:

– (id)**initWithDictionary:**(NSDictionary *)*otherDictionary*

Initializes a newly allocated dictionary by placing in it the keys and values contained in *otherDictionary*. Returns **self**.

See also: + **dictionaryWithDictionary:**

initWithDictionary:copyItems:

– (id)initWithDictionary:(NSDictionary *)*otherDictionary* **copyItems:(BOOL)***flag*

Initializes a newly allocated dictionary and, if *flag* is NO, places in it the objects contained in *otherDictionary*. If flag is YES, the members of *otherDictionary* are copied, and the copies are added to the receiver. (Note that **copyWithZone:** is used to make the copies. Thus, the receiver's new member objects may be immutable, even though their counterparts in *otherDictionary* were mutable. Also, members must conform to the NSCopying protocol.) Returns **self**.

See also: – **initWithDictionary:**

initWithObjects:forKeys:

– (id)**initWithObjects:**(NSArray *)*objects* **forKeys:**(NSArray *)*keys*

Initializes a newly allocated dictionary with entries constructed from the contents of the *objects* and *keys* arrays. This method steps through the *objects* and *keys* arrays, creating entries in the new dictionary as it goes. Each value object receives a **retain** message before being added to the dictionary. In contrast, each key object is copied (using **copyWithZone:**), and the copy is added to the dictionary. An NSInvalidArgumentException is raised if the *objects* and *keys* arrays do not have the same number of elements.

See also: + **dictionaryWithObjects:forKeys:**, – **initWithObjects:forKeys:count:**, – **initWithObjectsAndKeys:**

initWithObjects:forKeys:count:

– (id)**initWithObjects:**(id *)*objects* **forKeys:**(id *)*keys* **count:**(unsigned int)*count*

Initializes a newly allocated dictionary with *count* entries. This method steps through the *objects* and *keys* arrays, creating entries in the new dictionary as it goes. Each value object receives a **retain** message before being added to the dictionary. In contrast, each key object is copied (using **copyWithZone:**), and the copy is added to the dictionary. An NSInvalidArgumentException is raised if a key or value object is **nil**.

See also: + **dictionaryWithObjects:forKeys:count:**, – **initWithObjects:forKeys:**, – **initWithObjectsAndKeys:**

initWithObjectsAndKeys:

– (id)**initWithObjectsAndKeys:**(id)*object,* (id)*key, ...*

Initializes a newly allocated dictionary with entries constructed from the specified set of objects and keys. **initWithObjectsAndKeys:** takes a variable number of arguments: a null-terminated list of alternating objects and keys. If a key is found to be **nil**, an NSInvalidArgumentException is raised.

This method is similar to **initWithObjects:forKeys:**, differing only in the way in which the key-value pairs are specified.

See also: + **dictionaryWithObjectsAndKeys:**, – **initWithObjects:forKeys:**, – **initWithObjects:forKeys: count:**

isEqual:

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

Returns YES if the receiver and *anObject* are equal; otherwise returns NO. A YES return value indicates that the receiver and *anObject* are both instances of classes that inherit from NSDictionary and contain the same data (as determined by the **isEqualToDictionary:** method).

See also: – **isEqualToDictionary:**

isEqualToDictionary:

– (BOOL)**isEqualToDictionary:**(NSDictionary *)*otherDictionary*

Compares the receiving dictionary to *otherDictionary.* If the contents of *otherDictionary* are equal to the contents of the receiver, this method returns YES. If not, it returns NO.

Two dictionaries have equal contents if they each hold the same number of entries and, for a given key, the corresponding value objects in each dictionary satisfy the **isEqual:** test.

See also: – **isEqual:** (NSObject)

keyEnumerator

– (NSEnumerator *)**keyEnumerator**

Returns an enumerator object that lets you access each key in the dictionary:

```
NSEnumerator *enumerator = [myDictionary keyEnumerator];
id key;

while ((key = [enumerator nextObject])) {
    /* code that uses the returned key */
}
```

When this method is used with mutable subclasses of NSDictionary, your code shouldn't modify the entries during enumeration. If you intend to modify the entries, use the **allKeys** method to create a "snapshot" of the dictionary's keys. Then use this snapshot to traverse the entries, modifying them along the way.

Note that the **objectEnumerator** method provides a convenient way to access each value in the dictionary.

See also: – **allKeys**, – **allKeysForObject:**, – **objectEnumerator**, – **nextObject** (NSEnumerator protocol)

keysSortedByValueUsingSelector:

– (NSArray *)**keysSortedByValueUsingSelector:**(SEL)*comparator*

Returns an array of the dictionary's keys, in the order they would be in if the dictionary was sorted by its values. Pairs of dictionary values are compared using the comparison method specified by *comparator*; the comparator message is sent to one of the values, and has as its single argument the other value from the dictionary. The comparator method should return NSOrderedAscending if the receiver is smaller than the argument, NSOrderedDescending if the receiver is larger than the argument, and NSOrderedSame if they are equal.

See also: – **allKeys**, **– sortedArrayUsingSelector:** (NSArray)

objectEnumerator

– (NSEnumerator *)**objectEnumerator**

Returns an enumerator object that lets you access each value in the dictionary:

```
NSEnumerator *enumerator = [myDictionary objectEnumerator];
id value;

while ((value = [enumerator nextObject])) {
    /* code that acts on the dictionary's values */
}
```

When this method is used with mutable subclasses of NSDictionary, your code shouldn't modify the entries during enumeration. If you intend to modify the entries, use the **allValues** method to create a "snapshot" of the dictionary's values. Work from this snapshot to modify the values.

See also: – **keyEnumerator**, – **nextObject** (NSEnumerator protocol)

objectForKey:

– (id)**objectForKey:**(id)*aKey*

Returns an entry's value given its key, or **nil** if no value is associated with *aKey.*

See also: – **allKeys**, – **allValues**

objectsForKeys:notFoundMarker:

– (NSArray *)**objectsForKeys:**(NSArray *)*keys* **notFoundMarker:**(id)*anObject*

Returns the set of objects from the receiver that correspond to the specified *keys* as an NSArray. The objects in the returned array and the *keys* array have a one-for-one correspondence, so that the *n*th object in the returned array corresponds to the *n*th key in *keys*. If an object isn't found in the receiver to correspond to a given key, the marker object, specified by *anObject*, is placed in the corresponding element of the returned array.

writeToFile:atomically:

– (BOOL)**writeToFile:**(NSString *)*path*
atomically:(BOOL)*flag*

Writes a textual description of the contents of the dictionary to *path*. If the receiver's contents are all property list objects (NSString, NSData, NSArray, or NSDictionary objects), the file written by this method can be used to initialize a new dictionary with the class method **dictionaryWithContentsOfFile:** or the instance method **initWithContentsOfFile:**.

If *flag* is YES, the dictionary is written to an auxiliary file, and then the auxiliary file is renamed to *path*. If flag is NO, the dictionary is written directly to *path*. The YES option guarantees that *path*, if it exists at all, won't be corrupted even if the system should crash during writing.

If *path* contains a tilde (~) character, you must expand it with **stringByExpandingTildeInPath:** before invoking this method.

This method returns YES if the file is written successfully, and NO otherwise.

NSMutableDictionary

Inherits From: NSDictionary : NSObject

Conforms To: NSCoding
NSCopying
NSMutableCopying (NSDictionary)
NSObject (NSObject)

Declared In: Foundation/NSDictionary.h

Class at a Glance

Purpose

An NSDictionary object stores a mutable set of entries.

Principal Attributes

- A count of the number of entries in the dictionary
- The set of keys contained in the dictionary
- The objects that correspond to the keys in the dictionary

Creation

+ dictionaryWithCapacity:	Returns an empty dictionary with enough allocated space to hold a specified number of objects

Commonly Used Methods

– removeObjectForKey:	Removes the specified entry from the dictionary.
– removeObjectForKeys:	Removes multiple entries from the dictionary.

Primitive Methods

– setObject:forKey:
– removeObjectForKey:

Class Description

The NSMutableDictionary class declares the programmatic interface to objects that manage mutable associations of keys and values. With its two efficient primitive methods—**setObject:forKey:** and **removeObject:forKey:**—this class adds modification operations to the basic operations it inherits from NSDictionary.

The other methods declared here operate by invoking one or both of these primitives. The non-primitive methods provide convenient ways of adding or removing multiple entries at a time.

When an entry is removed from a mutable dictionary, the key and value objects that make up the entry receive **release** messages. If there are no further references to the objects, they're deallocated. Note that if

your program keeps a reference to such an object, the reference will become invalid unless you remember to send the object a **retain** message before it's removed from the dictionary. For example, the third statement below would result in a run-time error if *anObject* was not retained before it was removed:

```
id anObject = [aDictionary objectForKey:theKey] retain];

[aDictionary removeObjectForKey:theKey];
[anObject someMessage];
```

Method Types

Creating an NSMutableDictionary

+ allocWithZone:
+ dictionaryWithCapacity:
– initWithCapacity:

Adding and removing entries

– addEntriesFromDictionary:
– removeAllObjects
– removeObjectForKey:
– removeObjectsForKeys:
– setDictionary:
– setObject:forKey:

Class Methods

allocWithZone:

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

Creates and returns an uninitialized mutable dictionary in the specified zone. If the receiver is this class, an instance of a mutable private subclass is returned; otherwise, an object of the receiver's class is returned.

Typically, you create temporary dictionaries using the **dictionary...** class methods, not the **alloc...** and **init...** methods.

See also: + **dictionary** (NSDictionary), + **dictionaryWithCapacity:**, + **dictionaryWithContentsOfFile:** (NSDictionary), + **dictionaryWithObjects:forKeys:** (NSDictionary), + **dictionaryWithObjects:forKeys:count:** (NSDictionary), + **dictionaryWithObjectsAndKeys:** (NSDictionary)

dictionaryWithCapacity:

+ (id)dictionaryWithCapacity:(unsigned int)*numItems*

Creates and returns an mutable dictionary, giving it enough allocated memory to hold *numItems* entries. Mutable dictionaries allocate additional memory as needed, so *numItems* simply establishes the object's initial capacity.

See also: + **dictionary** (NSDictionary), + **dictionaryWithContentsOfFile:** (NSDictionary), + **dictionaryWithObjects:forKeys:** (NSDictionary), + **dictionaryWithObjects:forKeys:count:** (NSDictionary), + **dictionaryWithObjectsAndKeys:** (NSDictionary), – **initWithCapacity:**

Instance Methods

addEntriesFromDictionary:

– (void)**addEntriesFromDictionary:**(NSDictionary *)*otherDictionary*

Adds the entries from *otherDictionary* to the receiver. Each value object from *otherDictionary* is sent a **retain** message before being added to the receiver. In contrast, each key object is copied (using **copyWithZone:**; keys must conform to the NSCopying protocol), and the copy is added to the receiver.

If both dictionaries contain the same key, the receiver's previous value object for that key is sent a **release** message and the new value object takes its place.

See also: – **setObject:forKey:**

initWithCapacity:

– (id)**initWithCapacity:**(unsigned int)*numItems*

Initializes a newly allocated mutable dictionary, allocating enough memory to hold *numItems* entries. Mutable dictionaries allocate additional memory as needed, so *numItems* simply establishes the object's initial capacity. Returns **self**.

See also: **+ dictionaryWithCapacity:**

removeAllObjects

– (void)**removeAllObjects**

Empties the dictionary of its entries. Each key and corresponding value object is sent a **release** message.

See also: – **removeObjectForKey:**, – **removeObjectsForKeys:**

removeObjectForKey:

– (void)**removeObjectForKey:**(id)*aKey*

Removes *aKey* and its associated value object from the dictionary.

For example, assume you have a, archived dictionary that records the call letters and associated frequencies of radio stations. To remove an entry of a defunct station, you could write code similar to the following:

```
NSMutableDictionary *stations = nil;

stations = [[NSMutableDictionary alloc]
       initWithContentsOfFile: theArchiveFile];
[stations removeObjectForKey:@"KIKT"];
```

See also: – **removeAllObjects**, – **removeObjectsForKeys:**

removeObjectsForKeys:

– (void)**removeObjectsForKeys:**(NSArray *)*keyArray*

Removes one or more entries from the receiver. The entries are identified by the keys in *keyArray.*

See also: – **removeAllObjects**, – **removeObjectForKey:**

setDictionary:

– (void)**setDictionary:**(NSDictionary *)*otherDictionary*

Sets the receiver to entries in *otherDictionary.* setDictionary does this by removing all entries from the receiver (with **removeAllObjects**) then adding each entry from *otherDictionary* into the receiver.

setObject:forKey:

– (void)**setObject:**(id)*anObject* **forKey**:(id)*aKey*

Adds an entry to the receiver, consisting of a*Key* and its corresponding value object *anObject*. The value object receives a **retain** message before being added to the dictionary. In contrast, the key is copied (using **copyWithZone:**; keys must conform to the NSCopying protocol), and the copy is added to the dictionary. An NSInvalidArgumentException is raised if the key or value object is **nil**.

If *aKey* already exists in the receiver, the receiver's previous value object for that key is sent a **release** message and *anObject* takes its place.

See also: – **removeObjectForKey:**

NSDirectoryEnumerator

Inherits From:	NSEnumerator : NSObject
Conforms To:	NSObject (NSObject) NSCopying
Declared In:	Foundation/NSFileManager.h

Class Description

An NSDirectoryEnumerator object enumerates the contents of a directory, returning the pathnames of all files and directories contained within that directory. The pathnames are relative to the directory. This enumeration is recursive, including the files of all subdirectories, and it crosses device boundaries. It does not resolve symbolic links or attempt to traverse symbolic links that point to directories.

NSDirectoryEnumerator is an abstract class, a cover for a private concrete subclass that is tailored to the file system's directory structure. You cannot directly create an instance of NSDirectoryEnumerator—instances are returned only by NSFileManager's **enumeratorAtPath:** method.

To get the next item from the NSDirectoryEnumerator, invoke the NSEnumerator method **nextObject**. The methods declared by NSDirectoryEnumerator return attributes—both of the parent directory and the current file or directory—and allow you to control recursion into subdirectories.

The following example enumerates the contents of a directory and processes files; if, however, it comes across RTFD file packages, it skips recursion into them:

```
NSDirectoryEnumerator *direnum = [[NSFileManager defaultManager]
        enumeratorWithPath:@"/Sales/Reports"];
NSString *pname;
while (pname = [direnum nextObject]) {
    if ([[pname pathExtension] isEqualToString:@"rtfd"]) {
        [direnum skipDescendents]; /* don't enumerate this directory */
    }
    else {
        /* ...process file here... */
    }
}
```

Method Types

Getting attributes

– directoryAttributes
– fileAttributes

Skipping subdirectories

– skipDescendents

Instance Methods

directoryAttributes

– (NSDictionary *)**directoryAttributes**

Returns an NSDictionary that contains the attributes of the directory at which enumeration started. See the description of NSFileManager's **fileAttributesAtPath:traverseLink:** for details on obtaining the attributes from the dictionary.

See also: – **createDirectoryAtPath:attributes:** (NSFileManager)

fileAttributes

– (NSDictionary *)**fileAttributes**

Returns an NSDictionary that contains the attributes of the most recently returned file or subdirectory (as referenced by the path name). See the description of NSFileManager's **fileAttributesAtPath:follow:** for details on extracting the attributes from the dictionary.

skipDescendents

– (void)**skipDescendents**

Causes the NSDirectoryEnumerator to skip recursion into the most recently obtained subdirectory.

NSDistantObject

Inherits From:	NSProxy
Conforms To:	NSCoding NSObject (NSProxy)
Declared In:	Foundation/NSDistantObject.h

Class Description

NSDistantObject is a concrete subclass of NSProxy that defines proxies for objects in other applications or threads. When an NSDistantObject receives a message, in most cases it forwards the message through its NSConnection object to the real object in another application, supplying the return value to the sender of the message if one is forthcoming, and propagating any exception back to the invoker of the method that raised it.

NSDistantObject adds two useful instance methods to those defined by NSProxy. **connectionForProxy** returns the NSConnection that handles the receiver. **setProtocolForProxy:** establishes the set of methods that the real object is known to respond to, saving the network traffic required to determine the argument and return types the first time a particular selector is forwarded to the remote proxy.

There are two kinds of NSDistantObject: local proxies and remote proxies. A local proxy is created by an NSConnection the first time an object is sent to another application. It's used by the NSConnection for bookkeeping purposes and should be considered private. The local proxy is transmitted over the network using the NSCoding protocol to create the remote proxy, which is the object that the other application uses. NSDistantObject defines methods for an NSConnection to create instance, but they're intended only for subclasses to override—you should never invoke them directly. Use NSConnection's **rootProxyForConnectionWithRegisteredName:host:** method, which sets up all the required state for an object-proxy pair.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

Method Types

Creating a local proxy

+ proxyWithLocal:connection:
– initWithLocal:connection:

Creating a remote proxy

+ proxyWithTarget:connection:
– initWithTarget:connection:

Getting a proxy's NSConnection

– connectionForProxy

Setting a proxy's Protocol

– setProtocolForProxy:

Class Methods

proxyWithLocal:connection:

+ (NSDistantObject *)**proxyWithLocal:**(id)*anObject* **connection:**(NSConnection *)*aConnection*

Returns a local proxy for *anObject* and *aConnection*, creating it if necessary. *anObject* is an **id** in the receiver's address space. *connection* is set as the NSConnection for the returned proxy; other applications connect to it using NSConnection's **connectionWithRegisteredName:host:** class method.

Local proxies should be considered private to their NSConnections. Only an NSConnection should use this method to create them, and your code shouldn't retain or otherwise use local proxies.

proxyWithTarget:connection:

+ (NSDistantObject *)**proxyWithTarget:**(id)*remoteObject* **connection:**(NSConnection *)*aConnection*

Returns a remote proxy for *remoteObject* and *aConnection*, creating it if necessary. *remoteObject* is an **id** in another thread or in another application's address space. *aConnection* is set as the NSConnection for the returned proxy; it should have been created using NSConnection's **connectionWithRegisteredName:host:** class method.

A remote proxy can't be used until its NSConnection's peer has a local proxy representing *remoteObject* in the other application.

Instance Methods

connectionForProxy

– (NSConnection *)**connectionForProxy**

Returns the NSConnection used by the receiver.

initWithLocal:connection:

– (id)**initWithLocal:**(id)*anObject* **connection:**(NSConnection *)*aConnection*

Initializes a newly allocated NSDistantObject as a local proxy for *anObject*, which is an **id** in the receiver's address space. *aConnection* is set as the NSConnection for the returned proxy; other applications connect to it using NSConnection's **connectionWithRegisteredName:host:** class method. If a proxy for *anObject* and *aConnection* already exists, the receiver is released and the existing proxy is retained and returned.

Local proxies should be considered private to their NSConnections. Only an NSConnection should use this method to create them, and your code shouldn't retain or otherwise use local proxies.

This is the designated initializer for local proxies. Returns **self**.

initWithTarget:connection:

– (id)**initWithTarget:**(id)*remoteObject* **connection:**(NSConnection *)*aConnection*

Initializes a newly allocated NSDistantObject as a remote proxy for *remoteObject*, which is an **id** in another thread or in another application's address space. *aConnection* is set as the NSConnection for the returned proxy; it should have been created using NSConnection's **connectionWithRegisteredName:host:** class method. If a proxy for *remoteObject* and *aConnection* already exists, the receiver is released and the existing proxy is retained and returned.

A remote proxy can't be used until its NSConnection's peer has a local proxy representing *remoteObject* in the other application.

This is the designated initializer for remote proxies. Returns **self**.

setProtocolForProxy:

– (void)**setProtocolForProxy:**(Protocol *)*aProtocol*

Sets the methods known to be handled by the receiver to those in *aProtocol*. Setting a protocol for a remote proxy reduces network traffic needed to determine method argument and return types.

In order to encode a message's arguments for transmission over the network, the types of those arguments must be known in advance. When they're not known, the distributed objects system must send an initial

message just to get those types, doubling the network traffic for every new message sent. Setting a protocol alleviates this need for the methods defined by that protocol. You can still send messages that aren't declared in *aProtocol*; in this case the initial message is sent to determine the types, and then the real message is sent.

NSDistantObjectRequest

Inherits From: NSObject

Declared In: Foundation/NSConnection.h

Class Description

NSDistantObjectRequest objects are used by the Distributed Objects system to help handle invocations between different processes. You should never create NSDistantObjectRequest objects directly. Unless you are getting involved with the low-level details of Distributed Objects, there should never be a need to access an NSDistantObjectRequest.

Method Types

Getting information about a request
- connection
- invocation

Raising a remote exception
- replyWithException:

Class Methods

Instance Methods

connection

– (NSConnection *)**connection**

Returns the NSConnection involved in the request.

conversation

– (id)**conversation**

<<Description forthcoming.>>

invocation

– (NSInvocation *)**invocation**

Returns the NSInvocation for the request.

replyWithException:

– (void)**replyWithException:**(NSException *)*exception*

<<Description forthcoming.>>

NSDistributedLock

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSDistributedLock.h

Class Description

The NSDistributedLock class defines an object that multiple applications on multiple hosts can use to restrict access to some shared resource, such as a file.

The lock is implemented by an entry (such as a file or directory) in the file system. For multiple applications to use an NSDistributedLock to coordinate their activities, the lock must be writable on a file system accessible to all hosts on which the applications might be running.

Use the **tryLock** method to attempt to acquire a lock. You should generally use the **unlock** method to release the lock rather than **breakLock**.

NSDistributedLock doesn't conform to the NSLocking protocol nor does it have a **lock** method. The protocol's **lock** method is intended to block the execution of the thread until successful. For an NSDistributedLock object, this could mean polling the file system at some predetermined rate. A better solution is to provide the **tryLock** method and let you determine the polling frequency that makes sense for your application.

Method Types

Creating an NSDistributedLock

+ lockWithPath:
– initWithPath:

Acquiring a lock

– tryLock

Relinquishing a lock

– breakLock
– unlock

Getting lock information

– lockDate

Class Methods

lockWithPath:

+ (NSDistributedLock *)**lockWithPath:**(NSString *)*aPath*

Returns an NSDistributedLock object initialized to use the file system entry specified by *aPath* as the locking object. For applications to use the lock, this location in the file system must be accessible—and writable—to all hosts on which the applications might be running.

All of *aPath* up to the last component itself must exist. Use NSFileManager to create (and set permissions) for any nonexistent intermediate directories.

See also: – **initWithPath:**

Instance Methods

breakLock

– (void)**breakLock**

Forces the lock to be relinquished. This method always succeeds unless the lock has been damaged. If another process has already unlocked or broken the lock, this method has no effect. You should generally use **unlock** rather than **breakLock** to relinquish a lock.

Warning: Since **breakLock** can release another process's lock, it should be used with great caution.

Even if you break a lock, there's no guarantee that you will then be able to acquire the lock: Another process might get it before your **tryLock** is invoked.

Raises NSGenericException if the lock could not be removed.

See also: – **unlock**

initWithPath:

– (NSDistributedLock *)**initWithPath:**(NSString *)*aPath*

Initializes a newly allocated NSDistributedLock object to use the file system entry specified by *aPath* as the lock. For applications to use the lock, this location in the file system must be accessible—and writable—to all hosts on which the applications might be running.

All of *aPath* up to the last component itself must exist. Use NSFileManager to create (and set permissions) for any nonexistent intermediate directories.

See also: + **lockWithPath:**

lockDate

– (NSDate *)**lockDate**

Returns the time that the lock occurred.

This method is potentially useful to applications that want to use an age heuristic to decide if a lock is too old and should be broken. Returns **nil** if the lock doesn't exist.

If the creation date on the lock isn't the date on which you locked it, you've lost the lock: It's been broken since you last checked it.

tryLock

– (BOOL)**tryLock**

Attempts to acquire the lock. Returns immediately with a value of YES if successful and NO otherwise.

Raises NSGenericException if a file system error occurs.

See also: – **unlock**

unlock

– (void)**unlock**

Relinquishes the lock. You should generally use the **unlock** method rather than **breakLock** to release a lock.

An NSGenericException is raised if the lock doesn't already exist.

See also: – **breakLock**

NSDistributedNotificationCenter

Inherits From: NSNotificationCenter : NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSDistributedNotificationCenter.h

Class at a Glance

Purpose

NSDistributedNotificationCenter provides a way for objects that don't know about each other to communicate between tasks. It receives NSNotification objects and broadcasts them to all interested objects.

Principal Attributes

A table of objects that want to receive notifications, the notifications they want to receive, and identifying strings they are interested in.

Creation

Each machine has a default distributed notification center. You typically don't create your own.

Commonly Used Methods

+ defaultCenter	Accesses the default notification center.
– addObserver:selector:name:object: suspensionBehavior:	Registers an object to receive a notification with a specified behavior when notification delivery is suspended.
– postNotificationName:object:	Creates and posts a notification.
– removeObserver:name:object:	Specifies that an object no longer wants to receive certain notifications.

Class Description

An NSDistributedNotificationCenter object (or simply, *distributed notification center*) is a notification center that can distributed notifications to tasks other than the one in which the notification was posted.

It notifies all observers of notifications meeting specific criteria. This information is encapsulated in NSNotification objects, also known as *notifications*. Client objects register themselves as observers of specific notifications posted by other objects. When an event occurs, an object posts an appropriate notification to the distributed notification center. (See the NSNotification class specification for more on notifications.) The distributed notification center dispatches a message to each registered observer, passing the notification as the sole argument. It is possible for the posting object and the observing object to be the same.

Each task has a default distributed notification center that you access with the **defaultCenter** class method. There may be different types of distributed notification centers. Right now there is a single type—NSLocalNotificationCenterType. This type of distributed notification center handles notifications that can be sent between tasks on a single machine. For communication between tasks on different machines, use Distributed Objects.

Registering to Receive Notifications

An object registers itself to receive a notification by sending the **addObserver:selector:name:object: suspensionBehavior:** method, specifying the message the notification should send, the name of the notification it wants to receive, the identifying string to match (the **object:** argument), and the behavior to follow if notification delivery is suspended. Because the posting object and the observer may be in different tasks, notifications can't contain pointers to arbitrary objects. Therefore, NSDistributedNotificationCenter requires notifications to use a string as the **object:** argument. Notification matching is done based on this string, rather than an object pointer. You should check the documentation of the object posting the notification to see what it uses as its identifying string.

There are four different types of suspension behavior, each useful in different circumstances.

Suspension Behavior	Description
NSNotificationSuspensionBehaviorDrop	Drops notifications observed with this behavior when the observer suspends notifications.
NSNotificationSuspensionBehaviorCoalesce	Coalesces notifications observed with this behavior, such that only the most recent notification is delivered when notification delivery is resumed.
NSNotificationSuspensionBehaviorHold	Holds all notifications on the notification server for later delivery when notification delivery is resumed.

Suspension Behavior	Description
NSNotificationSuspensionBehaviorDeliverImmediately	Delivers matching notifications immediately. This also flushes the queue of undelivered notifications—the behavior is as if the receiving application had momentarily called **setSuspended:** with an argument of NO, then immediately after all notifications are delivered, called **setSuspended:** again with an argument of YES.

Suspending Notification Delivery

When a task is no longer interested in receiving notifications immediately, it may suspend notification delivery. This is often done when the application is hidden, or is put into the background. You suspend notifications by invoking **setSuspended:** with an argument of YES on the distributed notification center.

While notifications are suspended, the notification server handles notifications destined for the task that suspended notification delivery according to the suspension behavior specified by the observers when they registered to receive notifications. (See the table above.) When the task unsuspends notification delivery, all queued notifications are delivered immediately. Note that a notification destined for an observer that registered with NSNotificationSuspensionBehaviorDeliverImmediately will automatically flush the queue as it is delivered, causing all queued notifications to be delivered at that time as well.

Posting Notifications for Immediate Delivery

If an object posting a notification wants to ensure that all observers receive the notification immediately (for example, if the notification is a warning that a server is about to shut down), it can invoke **postNotificationName:object:userInfo:deliverImmediately:** with a **deliverImmediately:** argument of YES. The notification center will deliver the notification as if the observers had registered with NSNotificationSuspensionBehaviorDeliverImmediately.

Method Types

Accessing distributed notification centers

+ defaultCenter
+ notificationCenterForType:

Adding and removing observers

– addObserver:selector:name:object:
– addObserver:selector:name:object:suspensionBehavior:
– removeObserver:name:object:

Posting notifications

– postNotification:
– postNotificationName:object:
– postNotificationName:object:userInfo:
– postNotificationName:object:userInfo:deliverImmediately:

Suspending and enabling notification delivery

– setSuspended:
– suspended

Class Methods

defaultCenter

+ (NSNotificationCenter *)**defaultCenter**

Returns the default distributed notification center, the local notification center for the machine. Calls **notificationCenterForType:** with an argument of NSLocalNotificationCenterType.

notificationCenterForType:

+ (NSDistributedNotificationCenter *)**notificationCenterForType:**(NSString *)*type*

Returns the distributed notification center for the specified *type*. Currently only one type, NSLocalNotificationCenterType, is supported.

Instance Methods

addObserver:selector:name:object:

– (void)**addObserver:**(id)*anObserver*
selector:(SEL)*aSelector*
name:(NSString *)*notificationName*
object:(NSString *)*anObject*

Registers *anObserver* to receive notifications with the name *notificationName* and/or the identifying string *anObject*.

This method calls **addObserver:selector:name:object:suspensionBehavior:** with a **suspensionBehavior:** argument of NSNotificationSuspensionBehaviorCoalesce.

See also: – **addObserver:selector:name:object:suspensionBehavior:**

addObserver:selector:name:object:suspensionBehavior:

– (void)**addObserver:**(id)*anObserver*
selector:(SEL)*aSelector*
name:(NSString *)*notificationName*
object:(NSString *)*anObject*
suspensionBehavior:(NSNotificationSuspensionBehavior)*suspensionBehavior*

Registers *anObserver* to receive notifications with the name *notificationName* and/or the identifying string *anObject*.

When a notification of name *notificationName* with the identifying string *anObject* is posted, *anObserver* receives an *aSelector* message with this notification as the argument. The method for the selector specified in *aSelector* must have one and only one argument. If *notificationName* is **nil**, the notification center notifies the observer of all notifications with an identifying string matching *anObject*. If *anObject* is **nil**, the notification center notifies the observer of all notifications with the name *notificationName*. The *suspensionBehavior* determines how the notification center handles notifications when notification delivery has been suspended. See “Registering to Receive Notifications” in the class description for more information.

The notification center does not retain *anObserver*. Therefore, you should always send **removeObserver:** or **removeObserver:name:object:** to the notification center before releasing *anObserver*.

See also: – **postNotificationName:object:userInfo:deliverImmediately:**

postNotification:

– (void)**postNotification:**(NSNotification *)*notification*

Posts notification to the notification center. *notification* should have a string as its object instance variable, so the notification can be delivered in another task. This method invokes **postNotificationName:object:userInfo:deliverImmediately:** with the information contained in *notification* and a **deliverImmediately:** argument of NO.

See also: – **postNotificationName:object:**, – **postNotificationName:object:userInfo:**, – **postNotificationName:object:userInfo:deliverImmediately:**

postNotificationName:object:

– (void)**postNotificationName:**(NSString *)*notificationName*
object:(NSString *)*anObject*

Creates a notification with the name *notificationName*, associates it with the string *anObject*, and posts it to the notification center. *anObject* may be **nil**.

This method invokes **postNotificationName:object:userInfo:deliverImmediately:** with a **userInfo:** argument of **nil** and a **deliverImmediately:** argument of NO.

See also: – **postNotification:**

postNotificationName:object:userInfo:

– (void)**postNotificationName:**(NSString *)*notificationName*
object:(NSString *)*anObject*
userInfo:(NSDictionary *)*userInfo*

Creates a notification with the name *notificationName*, associates it with the string *anObject* and dictionary *userInfo*, and posts it to the notification center. *anObject* and *userInfo* may be **nil**.

This method invokes **postNotificationName:object:userInfo:deliverImmediately:** with a **deliverImmediately:** argument of NO.

See also: – **postNotificationName:object:**

postNotificationName:object:userInfo:deliverImmediately:

– (void)**postNotificationName:**(NSString *)*notificationName*
object:(NSString *)*anObject*
userInfo:(NSDictionary *)*userInfo*
deliverImmediately:(BOOL)*deliverImmediately*

Creates a notification with the name *notificationName*, associates it with the string *anObject* and dictionary *userInfo*, and posts it to the notification center with delivery scheduled for *deliverImmediately*, as supplied by the invoker. This method is the preferred method for posting notifications.

The *userInfo* dictionary is serialized using the NSArchiver class, so it can be passed to another task. In the receiving task, it is deserialized using NSUnarchiver. This imposes some restrictions on the objects that can be placed in the userInfo dictionary. See the NSArchiver and NSUnarchiver documentation for details.

Posting with *deliverImmediately* set to NO allows the normal suspension behavior of the observers to take place. If *deliverImmediately* is set to YES, the notification is delivered immediately to all observers, regardless of their suspension behavior or suspension state.

See also: – **postNotificationName:object:userInfo:**, **– encodeRootObject: (NSArchiver)**, **+ unarchiveObjectWithData: (NSUnarchiver)**

removeObserver:name:object:

– (void)**removeObserver:**(id)*anObserver*
name:(NSString *)*notificationName*
object:(NSString *)*anObject*

Removes *anObserver* as the observer of notifications with the name *notificationName* and identifying string *anObject* from the notification center. Be sure to invoke this method (or **removeObserver:**) before deallocating the observer object.

If *anObserver* is **nil**, all objects are removed as observers of *notificationName* with identifying string *anObject*. If *notificationName* is **nil**, *anObserver* is removed as an observer of all notifications with identifying string *anObject*. If *anObject* is **nil**, *anObserver* is removed as an observer of *notificationName* with any identifying string.

setSuspended:

– (void)**setSuspended:**(BOOL)*suspended*

Suspends notification delivery when set to YES, and resumes immediate notification delivery when set to NO. Distributed notification centers enable or suspend notification delivery on a per-task basis. When a task suspends notification delivery, notifications are delivered according to the suspension behavior of the observer. When delivery is not suspended, notifications are always delivered immediately. See “Suspending Notification Delivery” in the class introduction for more information.

See also: **– addObserver:selector:name:object:suspensionBehavior:**,
– postNotificationName:object:userInfo:deliverImmediately:

suspended

– (BOOL)**suspended**

Returns YES if the notification center is delivering notifications for this application according to their suspension behavior, NO if it is delivering them immediately.

NSEnumerator

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSUtilities.h

Class at a Glance

Purpose
An abstract class whose instances enumerate collections of other objects, such as arrays and dictionaries.

Creation
All creation methods are defined in the collection classes such as NSArray and NSDictionary. These methods contain the word "Enumerator," as in NSArray's **objectEnumerator** method or NSDictionary's **keyEnumerator** method.

Commonly Used Methods

– nextObject	Returns the next object in the collection being enumerated.

Primitive Methods
– nextObject

Class Description

NSEnumerator is a simple abstract class whose subclasses enumerate collections of other objects. Collection objects—such as arrays, sets, and dictionaries—provide special NSEnumerator objects with which to enumerate their contents. You send **nextObject** repeatedly to a newly-created NSEnumerator object to have it return the next object in the original collection. When the collection is exhausted, **nil** is returned. You can't "reset" an enumerator after it's exhausted its collection. To enumerate a collection again, you need a new enumerator.

Collection classes such as NSArray, NSSet, and NSDictionary include methods that return an enumerator appropriate to the type of collection. For instance, NSArray has two methods that return an NSEnumerator object: **objectEnumerator** and **reverseObjectEnumerator**. NSDictionary also has two methods that return an NSEnumerator object: **keyEnumerator** and **objectEnumerator**. These methods let you enumerate the contents of an NSDictionary by key or by value, respectively.

Note: It isn't safe to modify a mutable collection while enumerating through it.

Like other collection classes, an NSEnumerator retains the collection over which it's enumerating (unless implemented differently by a custom subclass).

Method Types

Getting the objects

– allObjects
– nextObject

Instance Methods

allObjects

– (NSArray *)**allObjects**

Returns an array of the objects the receiver has yet to enumerate. The array returned by this method does not contain objects that have already been enumerated with previous **nextObject** messages. Invoking this method exhausts the enumerator's collection so that subsequent invocations of **nextObject** return **nil**.

nextObject

– (id)**nextObject**

Returns the next object from the collection being enumerated. When **nextObject** returns **nil**, all objects have been enumerated. The following code illustrates how this works using NSArray:

```
NSEnumerator *enumerator = [anArray objectEnumerator];
id object;

while ((object = [enumerator nextObject])) {
    // do something with object...
}
```

NSException

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSObject (NSObject)
Declared In:	Foundation/NSException.h

Class Description

NSException is used to implement exception handling and contains information about an exception. An *exception* is a special condition that interrupts the normal flow of program execution. Each application can interrupt the program for different reasons. For example, one application might interpret saving a file in a directory that's write-protected as an exception. In this sense, the exception is equivalent to an error. Another application might interpret the user's keypress (i.e., Control-C) as an exception: an indication that a long-running process should be aborted.

Note: The exception handling mechanism uses **longjmp** to control the flow of execution. Any code written for an application that uses exception handling is therefore subject to the restrictions associated with this functionality. See your compiler documentation for more information on the **longjmp** function.

Raising an Exception

Once an exception is detected, it must be propagated to code that will handle it, called the *exception handler*. This entire process of handling an exception is referred to as "raising an exception." Exceptions are raised by instantiating an NSException object and sending it a **raise** message.

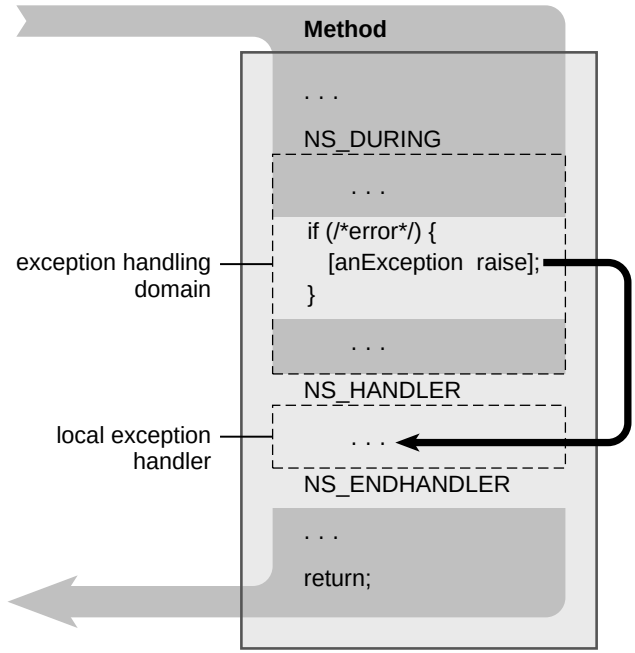
NSException objects provide:

- a name - a short string that is used to uniquely identify the exception.
- a reason - a longer string that contains a "human-readable" reason for the exception.
- userInfo - a dictionary used to supply application-specific data to the exception handler. For example, if the return value of a method causes an exception to be raised, you could pass the return value to the exception handler through **userInfo**.

Handling an Exception

Where and how an exception is handled depends on the context where the exception was raised. In general, a **raise** message is sent to an NSException object within the domain of an exception handler. An exception

handler is contained within a control structure created by the macros NS_DURING, NS_HANDLER, and NS_ENDHANDLER, as shown in the following illustration.



The section of code between NS_DURING and NS_HANDLER is the *exception handling domain*; the section between NS_HANDLER and NS_ENDHANDLER is the *local exception handler*. The normal flow of program execution is marked by the gray arrow; the code within the local exception handler is executed only if an exception is raised. Sending a **raise** message to an exception object causes program control to jump to the first executable line following NS_HANDLER.

Although you can raise an exception directly within the exception handling domain, **raise** is more often invoked indirectly from a method invoked from the domain. No matter how deep in a call sequence the exception is raised, execution jumps to the local exception handler (assuming there are no intervening exception handlers, as discussed in the next section). In this way, exceptions raised at a low level can be caught at a high level.

For example, in the following program excerpt, the local exception handler displays an attention panel after detecting an exception having the name MyAppException. The local exception handler has access to the raised exception object through a local variable **localException**.

```
NS_DURING
    ...
    if (someError)
        [anException raise];
    ...
NS_HANDLER
    if ([[localException name] isEqualToString:MyAppException]) {
        NSRunAlertPanel(@"Error Panel", @"%@", @"OK", nil, nil,
               localException);
    }
    [localException raise]; /* Re-raise the exception. */
NS_ENDHANDLER
```

You may leave the exception handling domain (the section of code between NS_DURING and NS_HANDLER) by:

- Raising an exception.
- Calling **NS_VALUERETURN()**
- Calling **NS_VOIDRETURN**
- “Falling off the end”

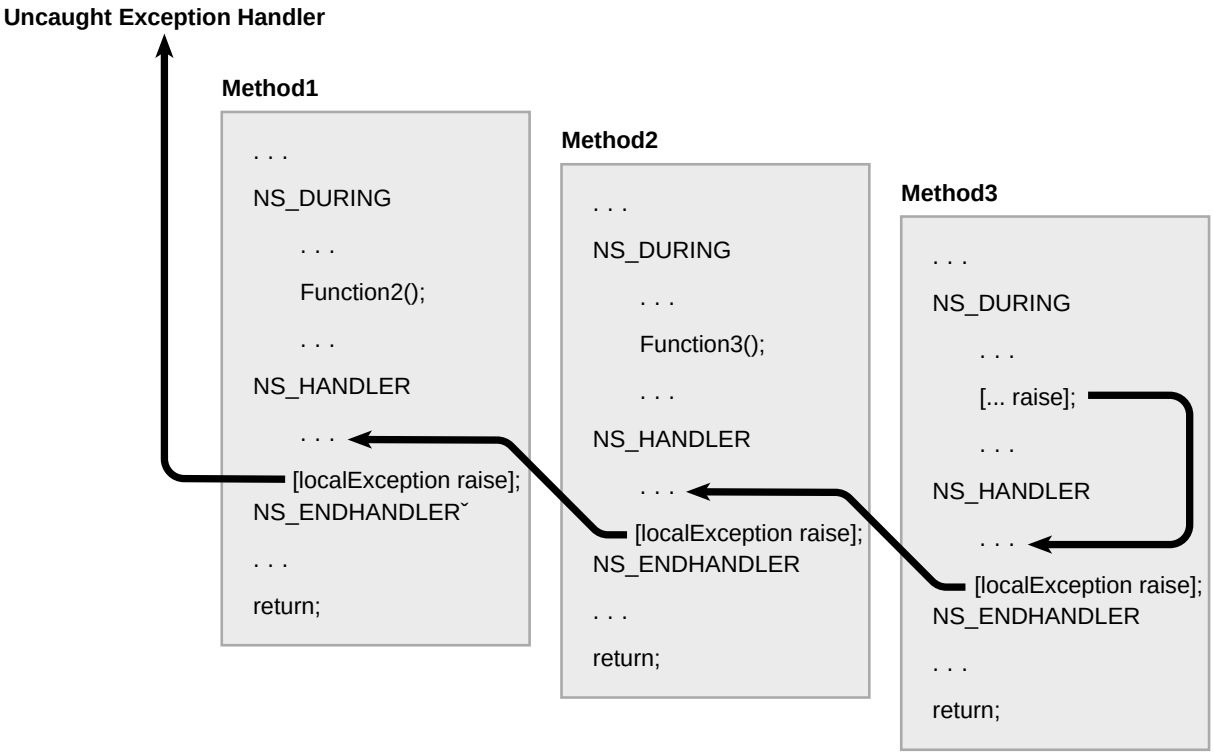
The above example raises an exception when **someError** is YES. Alternatively, you can return control to the caller from within the exception handling domain by calling either **NS_VALUERETURN()** or **NS_VOIDRETURN**. "Falling off the end” is simply the normal path of execution—after all statements in the exception handling domain are executed, execution continues on the line following NS_ENDHANDLER.

Note: You can’t use **goto** or **return** to exit an exception handling domain—errors will result. Nor can you use **setjmp()** and **longjmp()** if the jump entails crossing an NS_DURING statement. Since the code that your program calls may have exception handling domains within it, avoid using **setjmp()** and **longjmp()** in your application.

Similarly, you can leave the local exception handler (the section of code between NS_HANDLER and NS_ENDHANDLER) by raising an exception or simply "falling off the end".

Nested Exception Handlers

In the code example above, the same exception, **localException**, is raised again at the end of the local handler, allowing an encompassing exception handler to take some additional action. Exception handlers can be nested so that an exception raised in an inner domain can be treated by the local exception handler and any number of encompassing exception handlers. The following diagram illustrates the use of nested exception handlers, and is discussed in the text that follows.



An exception raised within Method3's domain causes execution to jump to its local exception handler. In a typical application, this exception handler checks the object **localException** to determine the nature of the exception. For exception types that it recognizes, the local handler responds and then may send **raise** to **localException** to pass notification of the exception to the handler above, the handler in Method2. (An exception that's re-raised appears to the next higher handler just as if the initial exception had been raised within its own exception handling domain.) Method2's exception handler does the same and then re-raises the exception to Method1's handler. Finally, Method1's handler re-raises the exception. Since there's no exception handling domain above Method1, the exception is transferred to the uncaught exception handler as described below.

Uncaught Exceptions

If an exception is not caught by any handler, it's intercepted by the uncaught exception handler, a function set by **NSSetUncaughtExceptionHandler()** and returned by **NSGetUncaughtExceptionHandler()**.

The default uncaught exception handler logs a message in the console and exits the program. However, for Application Kit programs, the message is logged with the Workspace Manager's console window (if the application was launched by the Workspace Manager) or to a Terminal window (if the application was launched from the shell).

You can change the default behavior by changing the uncaught exception handler using **NSSetUncaughtExceptionHandler()**.

Predefined Exceptions

Rhapsody predefines a number of exception names. These exception names are defined in **NSException.h**. For example:

- **NSGenericException**
- **NSRangeException**
- **NSInvalidArgumentException**
- **NSMallocException**

You can catch any of these exceptions from within your exception handler by comparing the exception's name with these predefined names. Note that all predefined exceptions begin with the prefix "NS", so you should avoid using the same prefix when creating new exception names.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

NSCopying

– copyWithZone:

Method Types

Creating and raising an NSException

+ exceptionWithName:reason:userInfo:
+ raise:format:
+ raise:format:arguments:
– initWithName:reason:userInfo:
– raise

Querying an NSException

– name
– reason
– userInfo

Class Methods

exceptionWithName:reason:userInfo:

+ (NSException *)**exceptionWithName:**(NSString *)*name*
reason:(NSString *)*reason*
userInfo:(NSDictionary *)*userInfo*

Creates and returns an exception object using a predefined *name*, a human-readable *reason*, and arbitrary *userInfo*.

See also: **– initWithName:reason:userInfo:**, **– name**, **– reason**, **– userInfo**

raise:format:

+ (void)**raise:**(NSString *)*name* **format:**(NSString *)*format,...*

A convenience method that creates and raises an exception with name *name* and a reason constructed from *format* and the arguments that follow in the manner of **printf()**. The user-defined information is **nil**.

See also: **+ raise:format:arguments:**, **– raise**

raise:format:arguments:

+ (void)**raise:**(NSString *)*name* **format:**(NSString *)*format* **arguments:**(va_list)*argList*

Creates and raises an exception with name *name* and a reason constructed from *format* and the arguments in *argList*, in the manner of **vprintf()**. The user-defined information is **nil**.

See also: **+ raise:format:**, **– raise**

Instance Methods

description

– (NSString *)**description**

Overridden to return the receiver's reason, so that "%@" used in formatted strings produces a meaningful description of the exception.

See also: – **reason**

initWithName:reason:userInfo:

– (id)**initWithName:**(NSString *)*name*
reason:(NSString *)*reason*
userInfo:(NSDictionary *)*userInfo*

Initializes a newly allocated exception object using the predefined *name*, human-readable *reason*, and user-defined *userInfo*. This is the designated initializer.

See also: **+ exceptionWithName:reason:userInfo:**, **– name**, **– reason**, **– userInfo**

name

– (NSString *)**name**

Returns an NSString used to uniquely identify the exception.

See also: **+ exceptionWithName:reason:userInfo:**, **– initWithName:reason:userInfo:**

raise

– (void)**raise**

Raises the exception, causing program flow to jump to the local exception handler. All other methods that raise an exception call this method, so set a breakpoint here if you are debugging exceptions.

See also: **+ raise:format:**, **+ raise:format:arguments:**

reason

– (NSString *)**reason**

Returns an NSString containing a “human-readable” reason for the exception.

See also: **– description**, **+ exceptionWithName:reason:userInfo:**, **– initWithName:reason:userInfo:**

userInfo

– (NSDictionary *)**userInfo**

Returns an NSDictionary that contains application-specific data pertaining to the receiver. Returns **nil** if no application-specific data exists. As an example, if a method’s return value caused the exception to be raised, the return value might be available to the exception handler through this method.

See also: + **exceptionWithName:reason:userInfo:**, **– initWithName:reason:userInfo:**

NSFileHandle Class Cluster

Class Cluster Description

NSFileHandle objects provide an object-oriented wrapper for accessing open files or communications channels.

The objects you create using this class are called *file handle objects*. Because of the nature of class clusters, file handle objects are not actual instances of the NSFileHandle class but of one of its private subclasses. Although a file handle object's class is private, its interface is public, as declared by the abstract superclass NSFileHandle.

Generally, you instantiate a file handle object by sending one of the **fileHandle...** messages to the NSFileHandle class object. These methods return a file handle object pointing to the appropriate file or communications channel. As a convenience, NSFileHandle provides class methods that create objects representing files and devices in the file system and that return objects representing the standard input, standard output, and standard error devices. You can also create file handle objects from file descriptors (such as found on BSD systems) using the **initWithFileDescriptor:** methods. If you create file handle objects with these methods, you "own" the represented descriptor and are responsible for removing it from system tables, usually by sending the file handle object a **closeFile** message .

NSFileHandle

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSFileHandle.h

Class Description

An NSFileHandle is an object that represents an open file or a communications channel. It enables programs to read data from or write data to the represented file or channel. You can use other OpenStep methods for reading from and writing to files—NSFileManager's **contentsAtPath:** and NSData's **writeToFile:atomically:** are but a couple of examples. Why would you use NSFileHandle then? What are its advantages?

- NSFileHandle gives you greater control over input/output operations on files. It allows more manipulatve operations on and within open files, such as seeking, truncating. and reading and writing at an exact position within a file (the file pointer). Other OpenStep methods read or write a file in its entirety; with NSFileHandle, you can range over an open file and insert, extract, and delete data.
- The scope of NSFileHandle is not limited to files. It provides the only OpenStep object that can read and write to communications channels such as those implemented by sockets, pipes, and devices.
- NSFileHandle makes possible asynchronous background communication. With it a program can connect to, and read from, a socket in a separate thread. (See "Background Inter-Process Communication Using Sockets" below for details on how this is done.)
- NSFileHandle enhances application portability. Its API supports I/O operations on file systems on Rhapsody (and supported UNIX systems) as well as on certain other file sysrems such as exist on Windows NT. Because this API is closer in semantics to traditional BSD file I/O, it makes it easier for many programmers to port applications.

Note: Instances of NSPipe, a class closely related to NSFileHandle, represent pipes: unidirectional interprocess communication channels that are found on both Rhapsody (and supported UNIX systems) and on Windows NT. See the NSPipe specification for details.

Background Inter-Process Communication Using Sockets

Sockets are full-duplex communication channels between processes either local to the same host machine or where one process is on a remote host. Unlike pipes, in which data goes in one direction only, sockets allow processes both to send and receive data. NSFileHandle facilitates communciation over stream-type

sockets by providing mechanisms run in background threads that accept socket connections and read from sockets.

Note: NSFileHandle currently handles only communciation through stream-type sockets. If you want to use datagrams or other types of sockets, you must create and manage the connection using native system routines.

The process on one end of the communication channel (the server) starts by creating and preparing a socket using system routines. These routines vary slightly between BSD and non-BSD systems (such as Windows), but consist of the same sequence of steps:

1. Create a stream-type socket of a certain protocol.
2. Bind a name to the socket.
3. Listen for incoming connections to the socket.

Typically the other process (the client) then locates the named socket created by the first process. Instead of accepting a connection to the socket by calling the appropriate system routine, the client performs the following sequence of steps:

1. It creates an NSFileHandle using the socket identifier as argument to **initWithFileDescriptor:**.
2. It adds itself as an observer of NSFileHandleConnectionAcceptedNotification.
3. It sends **acceptConnectionInBackgroundAndNotify** to this NSFileHandle. This method accepts the connection in the background, creates a new NSFileHandle from the new socket descriptor, and posts a NSFileHandleConnectionAcceptedNotification.
4. In a method implemented to respond to this notification, the client extracts the NSFileHandle representing the "near" socket of the connection from the notification's **userInfo** dictionary; it uses the NSFileHandleNotificationFileHandleItem key to do this.

The client can now send data to the other process over the communications channel by sending **writeData:** to the NSFileHandle. (Note that **writeData:** can block.) The client can also read data directly from the NSFileHandle, but this would cause the process to block until the socket connection was closed, so it is usually better to read in the background. To do this, the process must:

5. Add itself as an observer of NSFileHandleReadCompletionNotification or NSFileHandleReadToEndOfFileCompletionNotification.
6. Send **readInBackgroundAndNotify** or **readToEndOfFileInBackgroundAndNotify** to this NSFileHandle. The former method sends a notification after each transmission of data; the latter method accumulates data and sends a notification only after the sending process shuts down its end of the connection.
7. In a method implemented to respond to either of these notifications, the process extracts the transmitted or accumulated data from the notification's **userInfo** dictionary by using the NSFileHandleNotificationDataItem key.

You close the communications channel in both directions by sending **closeFile** to the NSFileHandle; either process can partially or totally close communication across the socket connection with a system-specific shutdown command.

Method Types

Getting an NSFileHandle

+ fileHandleForReadingAtPath:
+ fileHandleForWritingAtPath:
+ fileHandleForUpdatingAtPath:
+ fileHandleWithStandardError
+ fileHandleWithStandardInput
+ fileHandleWithStandardOutput
+ fileHandleWithNullDevice

Creating an NSFileHandle

– initWithFileDescriptor:
– initWithFileDescriptor:closeOnDealloc:

Getting a file descriptor

– fileDescriptor

Reading from an NSFileHandle

– availableData
– readDataToEndOfFile
– readDataOfLength:

Writing to an NSFileHandle

– writeData:

Communicating asynchronously in the background

– acceptConnectionInBackgroundAndNotifyForModes:
– acceptConnectionInBackgroundAndNotify
– readInBackgroundAndNotifyForModes:
– readInBackgroundAndNotify
– readToEndOfFileInBackgroundAndNotifyForModes:
– readToEndOfFileInBackgroundAndNotify

Seeking within a file

– offsetInFile
– seekToEndOfFile
– seekToFileOffset:

Operating on a file

– closeFile
– synchronizeFile
– truncateFileAtOffset:

Class Methods

fileHandleForReadingAtPath:

+ **fileHandleForReadingAtPath:**(NSString *)*path*

Returns an NSFileHandle initialized for reading the file, device, or named socket at *path*. The file pointer is set to the beginning of the file. The returned object responds only to NSFileHandle read messages. If no file exists at *path* the method returns **nil**.

See also: – **availableData**, – **initWithFileDescriptor:**, – **readDataOfLength:**, – **readDataToEndOfFile**

fileHandleForUpdatingAtPath:

+ **fileHandleForUpdatingAtPath:**(NSString *)*path*

Returns an NSFileHandle initialized for reading and writing to the file, device, or named socket at *path*. The file pointer is set to the beginning of the file. The returned object responds to both NSFileHandle read messages and **writeData:**. If no file exists at *path* the method returns **nil**.

See also: – **availableData**, – **initWithFileDescriptor:**, – **readDataOfLength:**, – **readDataToEndOfFile**

fileHandleForWritingAtPath:

+ **fileHandleForWritingAtPath:**(NSString *)*path*

Returns an NSFileHandle initialized for writing to the file, device, or named socket at *path*. The file pointer is set to the beginning of the file. The returned object responds only to **writeData:**. If no file exists at *path* the method returns **nil**.

See also: – **initWithFileDescriptor:**

fileHandleWithNullDevice

+ **fileHandleWithNullDevice**

Returns an NSFileHandle assoicated with a null device. You can use null-device NSFileHandles as "placeholders" for standard-device NSFileHandles or in collection objects to avoid exceptions and other errors resulting from messages being sent to invalid NSFileHandles. Read messages sent to a null-device

NSFileHandle return an end-of-file indicator (an empty NSData) rather than raise an exception. Write messages are no-ops whereas **fileDescriptor** returns an illegal value (as defined by the underlying operating system). Other methods are no-ops or return "sensible" values.

See also: – **initWithFileDescriptor:**

fileHandleWithStandardError

+ **fileHandleWithStandardError**

Returns the NSFileHandle associated with the standard error file, conventionally a terminal device to which error messages are sent. There is one such NSFileHandle per process; it is a shared instance.

See also: + **fileHandleWithNullDevice**, – **initWithFileDescriptor:**

fileHandleWithStandardInput

+ **fileHandleWithStandardInput**

Returns an NSFileHandle associated with the standard input file, conventionally a terminal device on which the user enters a stream of data. There is one such NSFileHandle per process; it is a shared instance.

See also: + **fileHandleWithNullDevice**, – **initWithFileDescriptor:**

fileHandleWithStandardOutput

+ **fileHandleWithStandardOutput**

Returns an NSFileHandle associated with the standard output file, conventionally a terminal device which receives a stream of data from a program. There is one such NSFileHandle per process; it is a shared instance.

See also: + **fileHandleWithNullDevice**, – **initWithFileDescriptor:**

Instance Methods

acceptConnectionInBackgroundAndNotify

– (void)**acceptConnectionInBackgroundAndNotify**

Accepts a socket connection (for stream-type sockets only) in the background and creates a NSFileHandle for the "near" (client) end of the communcations channel. This method is asynchronous. In a separate "safe" thread it accepts a connection, creates an NSFileHandle for the other end of the connection, and returns that object to the client by posting a NSFileHandleConnectionAcceptedNotification in the run loop of the client.

The notification includes as data a **userInfo** dictionary containing the created NSFileHandle; access this object using the NSFileHandleNotificationFileHandleItem key.

The receiver must be created by an **initWithFileDescriptor:** message that takes as an argument a stream-type socket created by the appropriate system routine. The object that will write data to the returned NSFileHandle must add itself as an observer of NSFileHandleConnectionAcceptedNotification.

See also: – **enqueueNotification:postingStyle:coalesceMask:forModes:** (NSNotificationQueue), – **readInBackgroundAndNotify**, – **readToEndOfFileInBackgroundAndNotify**

acceptConnectionInBackgroundAndNotifyForModes:

– (void)**acceptConnectionInBackgroundAndNotifyForModes:**(NSArray *)*modes*

Asynchronously accepts a connection with a stream-type socket in the background and returns (in an NSFileHandleConnectionAcceptedNotification) an NSFileHandle representing the client side of the connection. See **acceptConnectionInBackgroundAndNotify** for details. The method differs from **acceptConnectionInBackgroundAndNotify** in that *modes* specifies the run-loop mode (or modes) in which NSFileHandleConnectionAcceptedNotification can be posted.

See also: – **enqueueNotification:postingStyle:coalesceMask:forModes:** (NSNotificationQueue), – **readInBackgroundAndNotifyForModes:**, – **readToEndOfFileInBackgroundAndNotifyForModes:**

availableData

– (NSData *)**availableData**

If the receiver is a file, returns the data obtained by reading the file from the file pointer to the end of the file. If the receiver is a communications channel, reads up to a buffer of data and returns it (the size of the buffer depends on the operating system); if no data is available, the method blocks. Returns an empty NSData if the end of file is reached. Raises NSFileHandleOperationException if attempts to determine file-handle type fail or if attempts to read from the file or channel fail.

See also: – **readDataOfLength:**, – **readDataToEndOfFile**

closeFile

– (void)**closeFile**

Disallows further access to the represented file or communications channel and signals end of file on communications channels that permit writing. The file or communications channel, however, is available for other uses. Further read and write messages sent to an NSFileHandle to which **closeFile** has been sent will raise an exception.

Note: Sending **closeFile** to an NSFileHandle does not cause its deallocation; for that, you must send it **release**. Deallocation of an NSFileHandle, on the other hand, deletes its descriptor and closes the represented file or channel. Use **closeFile** when you want to close a file immediately and reclaim the descriptor; use **release** when it's acceptable to defer this.

fileDescriptor

– (int)**fileDescriptor**

Returns the file descriptor associated with the receiver. You can send this message to NSFileHandles originating from both file descriptors and file handles and receive a valid file descriptor (unless **closeFile** has been sent to the object, in which case an exception is raised).

See also: – **initWithFileDescriptor:**

initWithFileDescriptor:

– (id)**initWithFileDescriptor:**(int)*fileDescriptor*

Returns an NSFileHandle initialized with the file *descriptor*. If the operating system is Windows, the method converts the file descriptor to a file handle (WIN32 type HANDLE) and initializes the returned object with that as well. You can create an NSFileHandle for a socket on a BSD system by using the result of a **socket** call as *descriptor*. The object creating an NSFileHandle using this method owns *fileDescriptor* and is responsible for its disposition.

See also: – **closeFile**

initWithFileDescriptor:closeOnDealloc:

– (id)**initWithFileDescriptor:**(int)*fileDescriptor* **closeOnDealloc:**(BOOL)*flag*

Same as **initWithFileDescriptor:**, but *flag*, if YES, causes the file descriptor to be closed when the receiver is deallocated.

See also: – **closeFile**

offsetInFile

– (unsigned long long)**offsetInFile**

Returns the position of the file pointer within the file represented by the receiver. Raises an exception if the message is sent to an NSFileHandle representing a pipe or socket or if the file descriptor is closed.

See also: – **seekToEndOfFile**, – **seekToFileOffset:**

readDataOfLength:

– (NSData *)**readDataOfLength:**(unsigned int)*length*

If the receiver is a file, returns the data obtained by reading from the file pointer to *length* or to the end of the file, whichever comes first. If the receiver is a communications channel, the method reads data from the channel up to *length*. Returns an empty NSData if the file is positioned at the end of the file or if an end-of-file indicator is returned on a communications channel. Raises NSFileHandleOperationException if attempts to determine file-handle type fail or if attempts to read from the file or channel fail.

See also: – **availableData**, – **readDataToEndOfFile**

readDataToEndOfFile

– (NSData *)**readDataToEndFile**

Invokes **readDataOfLength:**, reading up to UNIT_MAX bytes (the maximum value for unsigned integers) or, if a communications channel, until an end-of-file indicator is returned.

See also: – **availableData**

readInBackgroundAndNotify

– (void)**readInBackgroundAndNotify**

Performs an asynchronous **availableData** operation on a file or communications channel and posts an NSFileHandleReadCompletionNotification to the client process' run loop. The length of the data is limited to the buffer size of the underlying operating system. The notification includes a **userInfo** dictionary which contains the data read; access this object using the NSFileHandleNotificationDataItem key.

Any object interested in receiving this data asynchronously must add itself as an observer of NSFileHandleReadCompletionNotification. In communication via stream-type sockets, the receiver is often the object returned in the **userInfo** dictionary of NSFileHandleConnectionAcceptedNotification.

See also: – **acceptConnectionInBackgroundAndNotify**, – **enqueueNotification:postingStyle: coalesceMask:forModes:** (NSNotificationQueue)

readInBackgroundAndNotifyForModes:

– (void)**readInBackgroundAndNotifyForModes:**(NSArray *)*modes*

Performs an asynchronous **availableData** operation on a file or communications channel and posts an NSFileHandleReadCompletionNotification to the client process' run loop. See **readInBackgroundAndNotify** for details. The method differs from **readInBackgroundAndNotify** in that

modes specifies the run-loop mode (or modes) in which NSFileHandleReadCompletionNotification can be posted.

See also: – **acceptConnectionInBackgroundAndNotifyForModes:**, – **enqueueNotification: postingStyle:coalesceMask:forModes:** (NSNotificationQueue)

readToEndOfFileInBackgroundAndNotify

– (void)**readToEndOfFileInBackgroundAndNotify**

Performs an asynchronous **readToEndOfFile** operation on a file or communications channel and posts an NSFileHandleReadToEndOfFileCompletionNotification to the client process' run loop. The notification includes a **userInfo** dictionary which contains the data read; access this object using the NSFileHandleNotificationDataItem key.

Any object interested in receiving this data asynchronously must add itself as an observer of NSFileHandleReadToEndOfFileCompletionNotification. In communication via stream-type sockets, the receiver is often the object returned in the **userInfo** dictionary of NSFileHandleConnectionAcceptedNotification.

See also: – **acceptConnectionInBackgroundAndNotify**, – **enqueueNotification:postingStyle: coalesceMask:forModes:** (NSNotificationQueue)

readToEndOfFileInBackgroundAndNotifyForModes:

– (void)**readToEndOfFileInBackgroundAndNotifyForModes:**(NSArray *)*modes*

In a detached "safe" thread, continuously reads data made available across a socket connection and accumulates it until the connection is closed; it then notifies observers. See **readToEndOfFileInBackgroundAndNotify** for details. The method differs from **readToEndOfFileInBackgroundAndNotify** in that *modes* specifies the run-loop mode (or modes) in which NSFileHandleReadToEndOfFileCompletionNotification can be posted.

See also: – **acceptConnectionInBackgroundAndNotifyForModes:**, – **enqueueNotification: postingStyle:coalesceMask:forModes:** (NSNotificationQueue)

seekToEndOfFile

– (unsigned long long)**seekToEndOfFile**

Puts the file pointer at the end of the file referenced by the receiver and returns the new file offset (thus yielding the size of the file). Raises an exception if the message is sent to an NSFileHandle representing a pipe or socket or if the file descriptor is closed.

See also: – **offsetInFile**

seekToFileOffset:

– (void)**seekToFileOffset:**(unsigned long long)*offset*

Moves the file pointer to the specified *offset* within the file represented by the receiver. Raises an exception if the message is sent to an NSFileHandle representing a pipe or socket, if the file descriptor is closed, or if any other error occurs in seeking.

See also: – **offsetInFile**

synchronizeFile

– (void)**synchronizeFile**

Causes all in-memory data and attributes of the file represented by the receiver to be written to permanent storage. This method should be invoked by programs that require the file to always be in a known state. An invocation of this method does not return until memory is flushed; because the way memory is flushed is platform-specific, consult the documentation for your operating system.

truncateFileAtOffset:

– (void)**truncateFileAtOffset:**(unsigned long long)*offset*

Truncates or extends the file represented by the receiver to *offset* within the file and puts the file pointer at that position. It the file is extended, the added characters are null bytes.

waitForDataInBackgroundAndNotify

– (void)**waitForDataInBackgroundAndNotify**

Checks to see if data is available in a background thread. When the data becomes available, the thread notifies all observers with NSFileHandleDataAvailableNotification. After the notification has been posted, the thread is terminated.

See also: **– waitForDataInBackgroundAndNotifyForModes:**

waitForDataInBackgroundAndNotifyForModes:

– (void)**waitForDataInBackgroundAndNotifyForModes:**(NSArray *)*modes*

Checks to see if data is available in a background thread. When the data becomes available, the thread notifies all observers with NSFileHandleDataAvailableNotification. After the notification has been posted,

the thread is terminated. This method differs from **waitForDataInBackgroundAndNotify** in that *modes* specifies the run-loop mode (or modes) in which NSFileHandleDataAvailableNotification can be posted.

See also: **– waitForDataInBackgroundAndNotify**

writeData:

– (void)**writeData:**(NSData *)*data*

Synchronously writes *data* to the file, device, pipe, or socket represented by the receiver. If the receiver is a file, writing takes place at the file pointer's current position. After it writes the data, the method advances the file pointer by the number of bytes written. Raises an exception if the file descriptor is closed or is not valid, if the receiver represents an unconnected pipe or socket end point, if no free space is left on the file system, or if any other writing error occurs.

See also: – **availableData**, – **readDataOfLength:**, – **readDataToEndOfFile**

Notifications

NSFileHandle posts several notifications related to asynchronous background I/O operations. They are set to post when the run loop of the thread that started the asynchronous operation is idle and for a specified set of run-loop modes (see **userInfo** dictionary).

NSFileHandleConnectionAcceptedNotification

This notification contains a notification object and a userInfo dictionary. The notification object is the NSFileHandle that sent the notification. The userInfo dictionary contains these keys and values:

Key	Value
NSFileHandleNotificationFileHandleItem	The NSFileHandle representing the "near" end of a socket connection.
NSFileHandleNotificationMonitorModes	An NSArray containing the run-loop modes in which the notification can be posted.

To cause the posting of this notification, you must send either **acceptConnectionInBackgroundAndNotify** or **acceptConnectionInBackgroundAndNotifyForModes:** to an NSFileHandle representing a server stream-type socket. This notification is posted when NSFileHandle establishes a socket connection between two processes, creates an NSFileHandle for one end of the connection, and makes this object available to observers by putting it in the **userInfo** dictionary.

NSFileHandleDataAvailableNotification

This notification contains a notification object and no userInfo dictionary. The notification object is the NSFileHandle that sent the notification.

To cause the posting of this notification, you must send either **waitForDataInBackgroundAndNotify** or **waitForDataInBackgroundAndNotifyForModes:** to an appropriate NSFileHandle. This notification is posted when the background thread determines that data is currently available for reading in a file or at a communications channel. The observers can then issue the appropriate messages to begin reading the data.

NSFileHandleReadCompletionNotification

This notification contains a notification object and a userInfo dictionary. The notification object is the NSFileHandle that sent the notification. The userInfo dictionary contains these keys and values:

Key	Value
NSFileHandleNotificationDataItem	An NSData containing the available data read from a socket connection.
NSFileHandleNotificationMonitorModes	An NSArray containing the run-loop modes in which the notification can be posted.

To cause the posting of this notification, you must send either **readInBackgroundAndNotify** or **readInBackgroundAndNotifyForModes:** to an appropriate NSFileHandle. This notification is posted when the background thread reads the data currently available in a file or at a communications channel. It makes the data available to observers by putting it in the **userInfo** dictionary.

NSFileHandleReadToEndOfFileCompletionNotification

This notification contains a notification object and a userInfo dictionary. The notification object is the NSFileHandle that sent the notification. The userInfo dictionary contains these keys and values:

Key	Value
NSFileHandleNotificationDataItem	An NSData containing the available data read from a socket connection.
NSFileHandleNotificationMonitorModes	An NSArray containing the run-loop modes in which the notification can be posted.

To cause the posting of this notification, you must send either **readToEndOfFileInBackgroundAndNotify** or **readToEndOfFileInBackgroundAndNotifyForModes:** to an appropriate NSFileHandle. This notification is posted when the background thread reads all data in the file or, if a communications channel, until the other process signals end of data. It makes the data available to observers by putting it in the **userInfo** dictionary.

NSFileManager

Inherits From:	NSObject
Conforms To:	NSObject (NSObject) NSCopying
Declared In:	Foundation/NSFileManager.h

Class Description

NSFileManager enables you to perform many generic file-system operations. With it you can:

- Create directories and files.
- Extract the contents of files (as NSData objects).
- Change your current working location in the file system.
- Copy, move, and link files and directories.
- Remove files, links, and directories.
- Determine the attributes of a file, a directory, or the file system.
- Set the attributes of a file or directory.
- Make and evaluate symbolic links.
- Determine the contents of directories.
- Compare files and directories for equality.

Besides offering a useful range of generic functionality, the NSFileManager API insulates an application from the underlying file system. An important part of this insulation is the encoding of file names (in, for example, Unicode, ISO Latin1, and ASCII). This insulating layer makes it easier to port the application between operating systems with different file systems. There is a default NSFileManager object for the file system; this object responds to all messages that request a operation on the associated file system.

The pathnames specified as arguments to NSFileManager methods can be absolute or relative to the current directory (which you can determine with **currentDirectoryPath** and set with **changeCurrentDirectoryPath:**). However, pathnames cannot include wildcard characters.

Note: On file systems on Rhapsody and supported UNIX systems, an absolute pathname starts with the root directory of the file system, represented by a slash (/), and ends with the file or directory that the pathname identifies. A relative pathname is relative to the *current directory*, the directory in which you are working and in which saved files are currently stored (if no pathname is specified). Relative pathnames start with a subdirectory of the current directory—without an initial slash—and end with the name of the file or directory that the pathname identifies.

Broken Links

Constructing a pathname to a file does not guarantee that the file exists at that path. Specifying a path results in one of the following possibilities:

- A file exists at that path
- A link to a file exists at that path
- A broken link exists at that path
- No file exists at that path

If the pathname specifies a valid file or link, you can obtain information about the file using the methods of this class. If the pathname specifies a broken link, you can still use **fileAttributesAtPath:traverseLink:** to obtain attributes for the link itself (by specifying NO for the traverseLink argument). However, the methods **fileExistsAtPath:** and **fileAttributesAtPath:traverseLink:** (with YES specified for the traverseLink argument) return **nil** when the pathname specifies a broken link. Other methods return appropriate errors; see the method descriptions for specific information. Regardless of whether a link is broken or valid, the link still appears in directory listings.

Path Utilities

NSFileManager methods are commonly used together with path-utility methods implemented as a category on NSString. These methods extract the components of a path (directory, file name, and extension), create paths from those components, "translate" path separators for the given platform, clean up paths containing symbolic links and redundant slashes, and perform similar tasks. Where your code manipulates strings that are part of file-system paths, it should use these methods. See the specification of the NSString class cluster for details.

Method Types

Getting the default manager

+ defaultManager

Directory operations

– changeCurrentDirectoryPath:
– createDirectoryAtPath:attributes:
– currentDirectoryPath

File operations

– copyPath:toPath:handler:
– createFileAtPath:contents:attributes:
– movePath:toPath:handler:
– linkPath:toPath:handler:
– removeFileAtPath:handler:

Getting and comparing file contents
– contentsAtPath:
– contentsEqualAtPath:andPath:

Detemining access to files
– fileExistsAtPath:
– fileExistsAtPath:isDirectory:
– isReadableFileAtPath:
– isWritableFileAtPath:
– isExecutableFileAtPath:
– isDeletableFileAtPath:

Getting and setting attributes
– fileAttributesAtPath:traverseLink:
– fileSystemAttributesAtPath:
– changeFileAttributes:atPath:

Discovering directory contents
– directoryContentsAtPath:
– enumeratorAtPath:
– subpathsAtPath:

Symbolic-link operations
– createSymbolicLinkAtPath:pathContent:
– pathContentOfSymbolicLinkAtPath:

Converting file-system representations
– fileSystemRepresentationWithPath:
– stringWithFileSystemRepresentation:length:

Class Methods

defaultManager

+ (NSFileManager *)**defaultManager**

Returns the default NSFileManager object for the file system. You invoke all NSFileManager instance methods with this object as the receiver.

Instance Methods

changeCurrentDirectoryPath:

– (BOOL)**changeCurrentDirectoryPath:**(NSString *)*path*

Changes the path of the current directory to *path* and returns YES if successful, NO if not successful. All relative pathnames refer implicitly to the current working directory. The current working directory is stored per task.

See also: – **currentDirectoryPath**, – **fileExistsAtPath:isDirectory:**, – **directoryContentsAtPath:**, – **createDirectoryAtPath:attributes:**

changeFileAttributes:atPath:

– (BOOL)**changeFileAttributes:**(NSDictionary *)*attributes* **atPath:**(NSString *)*path*

Changes the attributes of the file or directory specified by *path*. Attributes that you can change are the owner, the group, file permissions, and the modification date. As in the POSIX standard, the application must either own the file or directory or must be running as superuser for attribute changes to take effect. The method attempts to make all changes specified in *attributes* and ignores any rejection of an attempted modification. If all changes succeed, it returns YES. If any change fails, the method returns NO, but it is undefined whether any changes actually occured.

Some useful global keys for identifying object values in the *attributes* dictionary are:

Key	Value Type
NSFileModificationDate	NSDate
NSFilePosixPermissions	NSNumber

The NSFilePosixPermissions value must be initialized with the code representing the POSIX file-permissions bit pattern.

You can change single attributes or any combination of attributes; you need not specify keys for all four attributes.

See also: – **fileAttributesAtPath:traverseLink:**

contentsAtPath:

– (NSData *)**contentsAtPath:**(NSString *)*path*

Returns the contents of the file specified in *path* as an NSData object. If *path* specifies a directory, or if some other error occurs, this method returns **nil**.

See also: – **contentsEqualAtPath:andPath**:, – **createFileAtPath:contents:attributes:**

contentsEqualAtPath:andPath:

– (BOOL)**contentsEqualAtPath:**(NSString *)*path1* **andPath:**(NSString *)*path2*

Compares the file or directory specified in *path1* with that specified in *path2* and returns YES if they have the same contents. If *path1* and *path2* are directories, the contents are the list of files and subdirectories each contain; contents of subdirectories are also compared. If the contents differ, this method returns NO. For files, this method checks to see if they're the same file, then compares their size, and finally compares their contents. It does not traverse symbolic links, but compares the links themselves.

See also: – **contentsAtPath:**

copyPath:toPath:handler:

– (BOOL)**copyPath:**(NSString *)*source*
toPath:(NSString *)*destination*
handler:*handler*

Copies the directory or file specified in path *source* to a different location in the file system identified by pathname *destination*. If *source* is a file, the method creates a file at *destination* that holds the exact contents of the original file (this includes BSD special files). If *source* is a directory, the method creates a new directory at *destination* and recursively populates it with duplicates of the files and directories contained in *source*, preserving all links. The file specified in *source* must exist, while *destination* must not exist prior to the operation. When a file is being copied, the *destination* path must end in a file name; there is no implicit adoption of the *source* file name. Symbolic links are not traversed but are themselves copied.

If the copy operation is successful, the method returns YES. If the operation is not successful, but the callback *handler* of **fileManager:shouldProceedAfterError:** returns YES (see below), **copyPath:toPath:handler:** also returns YES. Otherwise this copy method returns NO. The method also attempts to make the attributes of the directory or file at *destination* identical to *source*, but ignores any failure at this attempt.

The argument *handler* identifies an object that responds to the callback messages **fileManager:willProcessPath:** and **fileManager:shouldProceedAfterError:** (see "Methods Implemented by the CallbackHandler," below). This callback mechanism is similar to delegation. NSFileManager sends the first message when it begins a copy, move, remove, or link operation. It sends the second message when it encounters any error in processing. You can specify **nil** for *handler* if no object responds to the callback messages; if you specify **nil** and an error occurs, the method automatically returns NO.

This code fragment verifies that the file to be copied exists and then copies that file to the user's **~/Library/Reports** directory:

```
NSString *source = @"/tmp/quarterly_report.rtf";
NSString *destination = [[NSHomeDirectory()
        stringByAppendingPathComponent:@"Library"]
        stringByAppendingPathComponent:@"Reports"];
NSFileManager *manager = [NSFileManager defaultManager];

if ([manager fileExistsAtPath:source])
    [manager copyPath:source toPath:destination handler:nil];
```

See also: – **linkPath:toPath:handler:**, – **movePath:toPath:handler:**, – **fileManager: shouldProceedAfterError:**, – **removeFileAtPath:handler:**, – **fileManager:willProcessPath:**

createDirectoryAtPath:attributes:

– (BOOL)**createDirectoryAtPath:**(NSString *)*path*
attributes:(NSDictionary *)*attributes*

Creates a directory (without contents) at *path* that has the specified *attributes*. Returns YES upon success or NO upon failure. The directory to be created must not yet exist, however, its parent directory must exist. The file attributes that you can set are owner and group numbers, file permissions, and modification date. If you specify **nil** for *attributes*, default values for these attributes are set (particularly write access for creator and read access for others). The following table lists the global constants used as keys in the *attributes* NSDictionary and the types of the associated values:

Key	Value Type
NSFileModificationDate	NSDate
NSFileOwnerAccountNumber	NSNumber
NSFileGroupOwnerAccountNumber	NSNumber
NSFilePosixPermissions	NSNumber

See also: – **changeCurrentDirectoryPath:**, – **changeFileAttributes:atPath:**, **– createFileAtPath: contents:attributes:**, **– currentDirectoryPath**

createFileAtPath:contents:attributes:

– (BOOL)**createFileAtPath:**(NSString *)*path*
contents:(NSData *)*contents*
attributes:(NSDictionary *)*attributes*

Creates a file at *path* that contains *contents* and has the specified file *attributes*. Returns YES upon success or NO upon failure. The file attributes that you can set are owner and group numbers, file permissions, and modification date. If you specify **nil** for *attribute*s, the file is given a default set of attributes. The following table summarizes the keys and types to associate with values in the NSDictionary *attributes*.

Key	Value Type
NSFileModificationDate	NSDate
NSFileOwnerAccountNumber	NSNumber
NSFileGroupOwnerAccountNumber	NSNumber
NSFilePosixPermissions	NSNumber

See also: – **contentsAtPath:**, – **changeFileAttributes:atPath:**, – **fileAttributesAtPath:traverseLink:**

createSymbolicLinkAtPath:pathContent:

– (BOOL)**createSymbolicLinkAtPath:**(NSString *)*path*
pathContent:(NSString *)*otherPath*

Creates a symbolic link identified by *path* that refers to the location *otherPath* in the file system. Returns YES if the operation is successful and NO if it is not successful. The method returns NO if a file, directory, or symbolic link identical to *path* already exists.

See also: – **pathContentOfSymbolicLinkAtPath:**, – **linkPath:toPath:handler:**

currentDirectoryPath

– (NSString *)**currentDirectoryPath**

Returns the path of the program's current directory. The string returned by this method is initialized to the current working directory, and can thereafter be changed by invoking **changeCurrentDirectoryPath:**. If the program's current working directory isn't accessible, this method returns **nil**.

Relative pathnames refer implictly to the current directory. For example, if the current directory is **/tmp**, and the relative pathname **reports/info.txt** is specified, the resulting full pathname is **/tmp/reports/info.txt**.

See also: – **createDirectoryAtPath:attributes:**

directoryContentsAtPath:

– (NSArray *)**directoryContentsAtPath:**(NSString *)*path*

Searches the contents of the directory specified by *path* and returns an array of strings identifying the directories and files (including symbolic links) contained in *path*. The search is shallow and therefore does not return the contents of any subdirectories. This returned array does not contain strings for the current directory (".") or parent directory ("..") and does not traverse symbolic links. This method returns **nil** if the directory specified at *path* does not exist or there is some other error accessing it. It returns an empty array if the directory exists, but has no contents.

The following example, generated by NSArray's **description** method, shows the results when this method is invoked on the directory **/System/Developer**.

```
Source,
Makefiles,
ProjectTypes,
Applications,
Java,
PBBundles,
Apps,
Demos,
Headers,
Examples
```

See also: – **currentDirectoryPath**, – **fileExistsAtPath:isDirectory:**, – **enumeratorAtPath:**, – **subpathsAtPath:**

enumeratorAtPath:

– (NSDirectoryEnumerator *)**enumeratorAtPath:**(NSString *)*path*

Creates and returns an autoreleased NSDirectoryEnumerator object that enumerates the contents of the directory specified at *path*.Because the enumeration is deep, that is, it lists the contents of all subdirectories, this enumerator object is useful for preforming actions that involve large file-system subtrees. If the method is passed a directory on which another filesystem is mounted (a mount point), it traverses the mount point. If handed a symbolic link, it evaluates the link and returns an enumerator for the file or directory the link points to. If the link can not be evaluated, the method returns nil. If *path* is a filename, the method returns an enumerator object that enumerates no files—the first call to **nextObject** will return nil.

This code fragment enumerates the subdirectories and files under **/MyAccount/Documents** and processes all files with an extension of **.doc**:

```
NSString *file;
NSDirectoryEnumerator *enumerator = [[NSFileManager defaultManager]
        enumeratorAtPath:@"/MyAccount/Documents"];
while (file = [enumerator nextObject]) {
    if ([[file pathExtension] isEqualToString:@"doc"])
            [self scanDocument:file];
}
```

The NSDirectoryEnumerator class has methods for obtaining the attributes of the existing path and of the parent directory, and for skipping descendents of the existing path.

See also: – **currentDirectoryPath**, – **fileExistsAtPath:isDirectory:**, – **directoryContentsAtPath:**, – **subpathsAtPath:**

fileAttributesAtPath:traverseLink:

– (NSDictionary *)**fileAttributesAtPath:**(NSString *)*path* **traverseLink:**(BOOL)*flag*

Returns an NSDictionary containing various objects that represent the POSIX attributes of the file specified at *path*. You access these objects using the keys:

Key	Value Type
NSFileSize (in bytes)	NSNumber
NSFileModificationDate	NSDate
NSFileOwnerAccountName	NSString
NSFileGroupOwnerAccountName	NSString
NSFileReferenceCount (number of hard links)	NSNumber
NSFileIdentifier	NSNumber
NSFileDeviceIdentifier	NSNumber
NSFilePosixPermissions	NSNumber
NSFileType	NSString

The possible values for the NSFileType key are:

- NSFileTypeDirectory
- NSFileTypeRegular
- NSFileTypeSymbolicLink
- NSFileTypeSocket
- NSFileTypeCharacterSpecial
- NSFileTypeBlockSpecial
- NSFileTypeUnknown

If *flag* is YES and *path* is a symbolic link, the attributes of the linked-to file are returned; if the link points to a non-existant file, this method returns **nil**. If *flag* is NO, the attributes of the symbolic link are returned.

This piece of code gets several attributes of a file and logs them.

```
NSNumber *fsize, *refs, *owner;
NSDate *moddate;

NSDictonary *fattrs =
        [manager fileAttributesAtPath:@"/tmp/List" traverseLink:YES];

if (!fattrs) {
    NSLog(@"Path is incorrect!");
    return;
}

if (fsize = [fattrs objectForKey:NSFileSize])
    NSLog(@"File size: %d\n", [fsize intValue]);

if (refs = [fattrs objectForKey:NSFileReferenceCount])
    NSLog(@"Ref Count: %d\n", [refs intValue]);

if (moddate = [fattrs objectForKey:NSFileModificationDate])
    NSLog(@"Modif Date: %@\n", [moddate description]);
```

As a convenience, NSDictionary provides a set of methods (declared as a category in **NSFileManager.h**) for quickly and efficiently obtaining attribute information from the returned NSDictionary: **fileGroupOwnerAccountName**, **fileModificationDate**, **fileOwnerAccountName**, **filePosixPermissions**, **fileSize**, **fileSystemFileNumber**, **fileSystemNumber**, and **fileType**. For example, you could rewrite the last statement in the code example above as:

```
if (moddate = [fattrs fileModificationDate])
    NSLog(@"Modif Date: %@\n", [moddate description]);
```

See also: – **changeFileAttributes:atPath:**

fileExistsAtPath:

– (BOOL)**fileExistsAtPath:**(NSString *)*path*

Returns YES if the file specified in *path* exists, or NO if it does not. If path specfies a symbolic link, this method traverses the link and returns YES or NO based on the existence of the file at the link destination.

See also: – **fileExistsAtPath:isDirectory:**

fileExistsAtPath:isDirectory:

– (BOOL)**fileExistsAtPath:**(NSString *)*path* **isDirectory:**(BOOL *)*isDirectory*

Returns whether the file specified in *path* exists. If you want to determine if *path* is a directory, specify the address of a boolean variable for *isDirectory*; the method indirectly returns YES if *path* is a directory. The method traverses final symbolic links.

This example gets an NSArray that identifies the fonts in **/System/Library/Fonts**:

```
NSArray *subpaths;
BOOL isDir;
NSString *fontPath = @"/System/Library/Fonts";
NSFileManager *manager = [NSFileManager defaultManager];
if ([manager fileExistsAtPath:fontPath isDirectory:&isDir] && isDir)
    subpaths = [manager subpathsAtPath:fontPath];
```

See also: – **fileExistsAtPath:**

fileSystemAttributesAtPath:

– (NSDictionary *)**fileSystemAttributesAtPath:**(NSString *)*path*

Returns an NSDictionary containing objects that represent attributes of the mounted file system; *path* is any pathname within the mounted file system. You access the attribute objects in the NSDictionary using these global constants as keys:

Key	Value Type
NSFileSystemSize (in an appropriate unit, usually bytes	NSNumber
NSFileSystemFreeSize (in an appropriate unit, usually bytes	NSNumber
NSFileSystemNodes	NSNumber
NSFileSystemFreeNodes	NSNumber
NSFileSystemNumber	NSNumber

The following code example checks to see if there's sufficient space on the file system before adding a new file to it:

```
const char *data = [[customerRec description] cString];
NSData *contents = [NSData dataWithBytes:data length:sizeof(data)];
NSFileManager *manager = [NSFileManager defaultManager];
NSDictionary *fsattrs =
        [manager fileSystemAttributesAtPath:@"/Net/sales/misc"];
if ([[fsattrs objectForKey:NSFileSystemFreeSize] unsignedIntValue] >
        [contents length])
    [manager createFileAtPath:@"/Net/sales/misc/custrec.rtf"
           contents:contents attributes:nil];
```

See also: – **fileAttributesAtPath:traverseLink:**, – **changeFileAttributes:atPath:**

fileSystemRepresentationWithPath:

– (const char *)**fileSystemRepresentationWithPath:**(NSString *)*path*

Returns a C-string representation of *path* that properly encodes Unicode strings for use by the file system. If you need the C string beyond the scope of your autorelease pool, you should copy it. This method raises an exception upon error. Use this method if your code calls system routines that expect C-string path arguments.

See also: – **stringWithFileSystemRepresentation:length:**

isDeletableFileAtPath:

– (BOOL)**isDeletableFileAtPath:**(NSString *)*path*

Returns YES if the invoking object appears able to delete the directory or file specified in *path* and NO if it cannot. To be deletable, either the parent directory of *path* must be writable or its owner must be the same as the owner of the application process. If *path* is a directory, every item contained in *path* must be deletable. This method does not traverse symbolic links.

isExecutableFileAtPath:

– (BOOL)**isExecutableFileAtPath:**(NSString *)*path*

Returns YES if the underlying operating system appears able to execute the file specified in path and NO if it cannot. This method traverses symbolic links.

isReadableFileAtPath:

– (BOOL)**isReadableFileAtPath:**(NSString *)*path*

Returns YES if the invoking object appears able to read the file specified in *path* and NO if it cannot. This method traverses symbolic links.

isWritableFileAtPath:

– (BOOL)**isWritableFileAtPath:**(NSString *)*path*

Returns YES if the invoking object appears able to write to the file specified in *path* and NO if it cannot. This method traverses symbolic links.

linkPath:toPath:handler:

– (BOOL)**linkPath:**(NSString *)*source*
toPath:(NSString *)*destination*
handler:*handler*

If pathname *source* identifies a file, this method hard-links the file specified in *destination* to it. If *destination* is a directory, this method creates a file in *destination* with the same basename (final path element) as *source*, hard-linked to *source*.

If *source* is a directory or symbolic link, this method copies it to *destination* instead of creating a hard link. The file, link, or directory specified in *source* must exist, while *destination* must not yet exist. The *destination* path must end in a file name; there is no implicit adoption of the *source* file name. Symbolic links in *source* are not traversed.

If the link operation is successful, **linkPath:toPath:handler:** returns YES. If the operation is not successful, but the *handler* method **fileManager:shouldProceedAfterError:** returns YES, the method also returns YES. Otherwise it returns NO.

The argument *handler* identifies an object that responds to the callback messages **fileManager: willProcessPath:** and **fileManager:shouldProceedAfterError:** (see “Methods Implemented by the Callback Handler,” below). This callback mechanism is similar to delegation. NSFileManager sends the first message when it begins a copy, move, remove, or link operation. It sends the second message when it encounters any error in processing. You can specify **nil** for *handler* if no object responds to the callback messages; if you specify **nil** and an error occurs, the method automatically returns NO.

This code fragment verifies the pathname typed in a text field (**imageFileField**) and then links the file to the user’s **~/Library/Images** directory:

```
NSString *imageFile = [imageFileField stringValue];
NSString *destination = [[NSHomeDirectory()
    stringByAppendingPathComponent:@"Library"]
    stringByAppendingPathComponent:@"Images"];
NSFileManager *manager = [NSFileManager defaultManager];

if ([manager fileExistsAtPath:source])
    [manager linkPath:source toPath:destination handler:self];
```

See also: – **copyPath:toPath:handler:**, – **createSymbolicLinkAtPath:pathContent:**, – **movePath:toPath:handler:**, – **fileManager:shouldProceedAfterError:,** – **removeFileAtPath:handler:**, – **fileManager:willProcessPath:**

movePath:toPath:handler:

– (BOOL)**movePath:**(NSString *)*source*
toPath:(NSString *)*destination*
handler:*handler*

Moves the directory or file specified in path *source* to a different location in the file system identified by the pathname *destination*. If *source* is a file, the method creates a file at *destination* that holds the exact contents of the original file (including BSD special files) and then deletes the original file. If *source* is a directory, **movePath:toPath:handler:** creates a new directory at *destination* and recursively populates it with duplicates of the files and directories contained in *source*; it then deletes the old directory and its contents. The file specified in *source* must exist, while *destination* must not yet exist. The *destination* path must end in a file name; there is no implicit adoption of the *source* file name. Symbolic links are not traversed; however, links are preserved.

If the move operation is successful, the method returns YES. If the operation is not successful, but the *handler* method **fileManager:shouldProceedAfterError:** returns YES, **movePath:toPath:handler:** also returns YES; otherwise it returns NO. If a failure in a move operation occurs, the pre-existing path or the new path remains intact, but not both.

The argument *handler* identifies an object that responds to the callback messages **fileManager:willProcessPath:** and **fileManager:shouldProceedAfterError:** (see “Methods Implemented by the Callback Handler,” below). This callback mechanism is similar to delegation. NSFileManager sends the first message when it begins a copy, move, remove, or link operation. It sends the second message when it encounters any error in processing. You can specify **nil** for *handler* if no object responds to the callback messages; if you specify **nil** and an error occurs, the method automatically returns NO.

See also: – **copyPath:toPath:handler:**, – **linkPath:toPath:handler:**, – **removeFileAtPath:handler:**, – **fileManager:shouldProceedAfterError:**, – **fileManager:willProcessPath:**

pathContentOfSymbolicLinkAtPath:

– (NSString *)**pathContentOfSymbolicLinkAtPath:**(NSString *)*cStringPath*

Returns the actual path of the directory or file that the symbolic link *cStringPath* refers to. Returns **nil** upon failure.

See also: – **createSymbolicLinkAtPath:pathContent:**

removeFileAtPath:handler:

– (BOOL)**removeFileAtPath:**(NSString *)*path* **handler:***handler*

Deletes the file, link, or directory (including, recusively, all subdirectories, files and links in the directory) identified by *path*. If the removal operation is successful, **removeFileAtPath:handler:** returns YES. If the operation is not successful, but the *handler* method **fileManager:shouldProceedAfterError:** returns YES, **removeFileAtPath:handler:** also returns YES; otherwise it returns NO.

The argument *handler* identifies an object that responds to the callback messages **fileManager: willProcessPath:** and **fileManager:shouldProceedAfterError:** (see "Methods Implemented by the Callback Handler," below). This callback mechanism is similar to delegation. NSFileManager sends the first message when it begins a copy, move, remove, or link operation. It sends the second message when it encounters any error in processing. You can specify **nil** for *handler* if no object responds to the callback messages; if you specify **nil** and an error occurs, the method automatically returns NO.

Since the removal of directory contents is so thorough and final, be careful when using this method. Do not specify "." or ".." for *path*; this will raise the exception NSInvalidArgumentException. This method does not traverse symbolic links.

See also: **– copyPath:toPath:handler:**, – **linkPath:toPath:handler:**, – **movePath:toPath:handler:**, – **fileManager:shouldProceedAfterError:**, – **fileManager:willProcessPath:**

stringWithFileSystemRepresentation:length:

– (NSString *)**stringWithFileSystemRepresentation:**(const char *)*string*
length:(unsigned int)*len*

Returns an NSString object converted from a C-string representation of a path name in the current file system (*string*). Use this method if your code receives paths as C-strings from system routines.

See also: – **fileSystemRepresentationWithPath:**

subpathsAtPath:

– (NSArray *)**subpathsAtPath:**(NSString *)*path*

Returns an NSArray that lists (as NSStrings) the contents of the directory identified by path. This list of directory contents goes very deep and hence is very useful for large file-system subtrees. The method skips "." and "..". If path is a symbolic link, **subpathsAtPath:** traverses the link. The method returns **nil** if it cannot get the device of the linked-to file.

Here is a sample fragment of what **subpathsAtPath:** returns (as the output of NSArray's **description** method) when *path* is **/System/Developer**:

```
Examples/Foundation/ForwardInvocation/PB.project,
Examples/Foundation/ForwardInvocation/Makefile,
Examples/Foundation/ForwardInvocation/TargetProxy.h,
Examples/Foundation/ForwardInvocation/TargetProxy.m,
Examples/Foundation/ForwardInvocation/ForwardInvocation_main.m,
Examples/Foundation/MultiThreadedDO,
Examples/Foundation/MultiThreadedDO/PB.project,
Examples/Foundation/MultiThreadedDO/Makefile,
Examples/Foundation/MultiThreadedDO/ThreadSafeQueue.h,
Examples/Foundation/MultiThreadedDO/ThreadSafeQueue.m,
Examples/Foundation/MultiThreadedDO/MultiThreadedDO_main.m,
```

Notice that this method reveals every element of the subtree at *path*, including the contents of file packages (such as applications, nib files, and RTFD files). This code fragment gets the contents of **/System/Library/Fonts** after verifying that the directory exists:

```
BOOL isDir=NO;
NSArray *subpaths;
NSString *fontPath = @"/System/Library/Fonts";
NSFileManager *manager = [NSFileManager defaultManager];
if ([manager fileExistsAtPath:fontPath isDirectory:&isDir] && isDir)
    subpaths = [manager subpathsAtPath:fontPath];
```

See also: **– directoryContentsAtPath:**, – **enumeratorAtPath:**

Methods Implemented By the Delegate

The methods described in this section are methods to be implemented by the callback handler passed to several methods of NSFileManager.

fileManager:shouldProceedAfterError:

– (BOOL)**fileManager:**(NSFileManager *)*manager*
shouldProceedAfterError:(NSDictionary *)*errorInfo*

NSFileManager sends this message for each error it encounters when copying, moving, removing, or linking files or directories. The NSDictionary object *errorInfo* contains two or three pieces of information (all NSStrings) related to the error:

Key	Value
@"Path"	The path related to the error (usually the source path)
@"Error"	A description of the error
@"ToPath"	The destination path (not all errors)

Return YES if the operation (which is often continuous within a loop) should proceed and NO if it should not; the Boolean value is passed back to the invoker of **copyPath:toPath:handler:**, **movePath:toPath:handler:**, **removeFileAtPath:handler:** or **linkPath:toPath:handler:**. If an error occurs and your handler has not implemented this method, the invoking method automatically returns NO.

The following implementation of **fileManager:shouldProceedAfterError:** displays the error string in an attention panel and leaves it to the user whether to proceed or stop:

```
-(BOOL)fileManager:(NSFileManager *)manager
        shouldProceedAfterError:(NSDictionary *)errorDict
{
    int result;
    result = NSRunAlertPanel(@"Gumby App", @"File operation error:
           %@ with file: %@", @"Proceed", @"Stop", NULL,
           [errorDict objectForKey:@"Error"],
           [errorDict objectForKey:@"Path"]);

    if (result == NSAlertDefaultReturn)
        return YES;
    else
        return NO;
}
```

See also: – **fileManager:willProcessPath:**

fileManager:willProcessPath:

-(BOOL)**fileManager:**(NSFileManager *)*manager*
willProcessPath:(NSString *)*path*

NSFileManager sends this message immediately before attempting to move, copy, or rename *path* or before attempting to link to *path*. You can implement this method in your delegate to monitor file operations. The return value is ignored.

NSFormatter

Inherits From: NSObject

Conforms To: NSObject (NSObject)
NSCoding
NSCopying

Declared In: Foundation/NSFormatter.h

Class Description

NSFormatter is an abstract class that declares an interface for objects that format the textual representation of cell contents. The Foundation framework provides two concrete subclasses of NSFormatter to generate these objects: NSNumberFormatter and NSDateFormatter.

Cells, which are instances of NSCell and its subclasses, can have any arbitrary object as their content. However, when cells are to be displayed or edited, they must convert this object to an NSString. If no formatting object is associated with a cell, the cell displays its content by invoking the localized **description** method of the object it contains. But if the cell has a formatting object, the cell invokes this object's **stringForObjectValue:** method to obtain the correctly formatted string. Conversely, when the user enters text into a cell, the cell needs to convert the text to the underlying object; formatting objects handle this conversion as well.

To use a formatting object, you must create an instance of NSNumberFormatter, NSDateFormatter, or a custom NSFormatter subclass and associate the object with a cell. The cell invokes the formatting behavior of this instance every time it needs to display its object or have it edited, and every time it needs to convert a textual representation to its object. See the class description of NSDateFormatter for the details of using formatting objects.

Instances of NSFormatter subclasses are immutable. In addition, when a cell with a formatter object is copied, the new cell retains the formatter object instead of copying it.

Note: NSCell provides two methods that operate almost the same as instances of NSNumberFormatter. One method, **setEntryType:**, takes a constant that specifies a typical numeric format (integer, float, positive float, double, and so on). With **isEntryAcceptable:**, you can ask a cell for the type of value it expects. Another method, **setFloatingPointFormat:left:right:**, allows you to specify the digits that appear to the left and right of the decimal point. See the NSCell specification for further details.

Delegation Methods for Error Handling

NSControl has delegation methods for handling errors returned in implementations of NSFormatter's **objectValue:forString:errorDescription:** and **isPartialStringValid:newEditingString:**

errorDescription:. These delegation methods are, respectively, **control:didFailToFormatString: errorDescription:** and **control:didFailToValidatePartialString:errorDescription:**.

Making a Subclass of NSFormatter

There are many possibilities for custom subclasses of NSFormatter. You might find use for a custom formatter of telephone numbers, or a custom formatter of part numbers.

To subclass NSFormatter, you must, at the least, override the two primitive methods s**tringForObjectValue:** and **getObjectValue:forString:errorDescription:**. In the first method you convert the cell's object to a string representation; in the second method you convert the string to the object associated with the cell.

Implement **attributedStringForObjectValue:withDefaultAttributes:** in addition to **stringForObjectValue:** when the display string has attributes associated with it. For example, if you want negative financial amounts to appear in red, you would return a string with an attribute of red text. In **attributedStringForObjectValue:withDefaultAttributes:** get the non-attributed NSString by invoking **stringForObjectValue:** and then apply the proper attributes to that NSString.

If the string for editing is different than the string for display—for example, the display version of a currency field should show a dollar sign but the editing version shouldn't—implement **editingStringForObjectValue:** in addition to **stringForObjectValue:**.

The method **isPartialStringValid:newEditingString:errorDescription:** allows you to edit the textual contents of a cell at each key press or to prevent entry of invalid characters. You might apply this on-the-fly editing to things like telephone numbers or social security numbers; the person entering data only needs to enter the number since the formatter automatically inserts the separator characters.

Method Types

Textual representation of cell content

– stringForObjectValue:
– attributedStringForObjectValue:withDefaultAttributes:
– editingStringForObjectValue:

Object equivalent to textual representation

– getObjectValue:forString:errorDescription:

Dynamic cell editing

– isPartialStringValid:newEditingString:errorDescription:

Instance Methods

attributedStringForObjectValue:withDefaultAttributes:

– (NSAttributedString *)**attributedStringForObjectValue:**(id)*anObject*
withDefaultAttributes:(NSDictionary *)*attributes*

The default implementation returns **nil** to indicate that the formatter object does not provide an attributed string. In a subclass implementation, return an NSAttributedString if the string for display should have some attributes. For instance, you might want negative values in a financial application to appear in red text. Invoke your implementation of **stringForObjectValue:** to get the non-attributed string. Then create an NSAttributedString with it. The default attributes for text in the cell is passed in with *attributes*; use this NSDictionary to reset the attributes of the string when a change in value warrants it (for example, a negative value becomes positive). For information on creating attributed strings, see the specification for the NSAttributedString class.

See also: – **editingStringForObjectValue:**

editingStringForObjectValue:

– (NSString *)**editingStringForObjectValue:**(id)*anObject*

The default implementation of this method invokes **stringForObjectValue:**. When implementing a subclass, override this method only when the string that users see and the string that they edit are different. In your implementation, return an NSString that is used for editing, following the logic recommended for implementing **stringForObjectValue:** (see below). As an example, you would implement this method if you want the dollar signs in displayed strings removed for editing.

See also: – **attributedStringForObjectValue:withDefaultAttributes:**

getObjectValue:forString:errorDescription:

– (BOOL)**getObjectValue:**(id *)*anObject*
forString:(NSString *)*string*
errorDescription:(NSString **)*error*

The default implementation of this method raises an exception. In your subclass implementation, return by reference the object *anObject* after creating it from the *string* passed in. Return YES if the conversion from string to cell-content object was successful and NO if any error prevented the conversion. If you return NO, also return by indirection a localized user-presentable NSString (in *error*) that explains the reason why the conversion failed; the delegate (if any) of the NSControl managing the cell can then respond to the failure in **control:didFailToFormatString:errorDescription:**. However, if *error* is NULL, the sender is not interested in the error description, and you should not attempt to assign one.

The following implementation example (which is paired with the **stringForObjectValue:** example below) converts an NSString representation of a dollar amount that includes the dollar sign; it uses an NSScanner to convert this amount to a float after stripping out the initial dollar sign.

```
- (BOOL)getObjectValue:(id *)obj forString:(NSString *)string errorDescription:(NSString
**)error
{
    float floatResult;
    NSScanner *scanner;
    BOOL retval = NO;
    NSString *err = nil;

    scanner = [NSScanner scannerWithString:string];
    if ([string hasPrefix:@"$"]) [scanner setScanLocation:1];
    if ([scanner scanFloat:&floatResult]
        && ([scanner scanLocation] == [string length] )) {
        if (obj) {
            *obj = [NSNumber numberWithFloat:floatResult];
        {
            retval = YES;
        }
        }else {
            err = NSLocalizedString(@"Couldn't convert to float");
        }
    }
    if (error) {
        *error = err;
    }
    return retval;
}
```

See also: – **stringForObjectValue:**

isPartialStringValid:newEditingString:errorDescription:

– (BOOL)**isPartialStringValid:**(NSString *)*partialString*
newEditingString:(NSString **)*newString*
errorDescription:(NSString **)*error*

Since this method is invoked at each key press in the cell, it permits editing or evaluation of cell text as it is typed. The text as currently typed (*partialString*) is passed in. Evaluate this text according to the context, edit the text if necessary, and return by reference any edited NSString in *newString*. Return YES if *partialString* is acceptable and NO if *partialString* is unacceptable. If you return NO and *newString* is **nil**, *partialString* minus the last character typed is displayed. If you return NO, you can also return by indirection an NSString (in *error*) that explains the reason why the validation failed; the delegate (if any)

of the NSControl managing the cell can then respond to the failure in **control: didFailToValidatePartialString:errorDescription:**.

stringForObjectValue:

– (NSString *)**stringForObjectValue:**(id)*anObject*

The default implementation of this method raises an exception. When subclassing, return the NSString that textually represents the cell's object for display and—if **editingStringForObjectValue:** is unimplemented—for editing. First test the passed-in object to see if it's of the correct class. If it isn't, return **nil**; but if it is of the right class return a properly formatted and, if necessary, localized string. (See the specification of the NSString class for formatting and localizing details.)

The following implementation (which is paired with the **getObjectValue: forString: errorDescription:** example above) prefixes a two-digit float representation with a dollar sign:

```
- (NSString *)stringForObjectValue:(id)anObject
{
    if (![anObject isKindOfClass:[NSNumber class]]) {
        return nil;
    }
    return [NSString stringWithFormat:@"$%.2f", [anObject floatValue]];
}
```

See also: – **attributedStringForObjectValue:withDefaultAttributes:**, – **editingStringForObjectValue:**, – **getObjectValue:forString:errorDescription:**

NSHost

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSHost.h

Class Description

An NSHost object holds network name and address information for a host. You use this class to get the current host's name and address and to look up other hosts by name or by address.

To create an NSHost, use the **currentHost**, **hostWithAddress:**, or **hostWithName:** class methods (don't use **alloc** and **init**). These methods use available network administration services (such as NetInfo or the Domain Name Service) to discover all names and addresses for the host requested. They don't attempt to contact the host itself, however. This avoids untimely delays due to a host being unavailable, but it may result in incomplete information about the host.

An NSHost instance contains all of the network addresses and names discovered for a given host by the network administration services. Each NSHost typically contains one unique address, but it may have more than one name. If an NSHost has more than one name, the additional names are variations on the same name, typically the basic host name plus the fully qualified domain name. For example, with a host name "sales" in the domain "anycorp.com", an NSHost can hold both the names "sales" and "sales.anycorp.com".

The NSHost class maintains a cache of previously created instances so that requests for an existing NSHost return that object instead of creating a new one. Use the **setHostCacheEnabled:** method to turn the cache off, forcing lookup of hosts as they're requested. You can also use the **flushHostCache** method to clear the cache of its entries so that subsequent requests look up the host information and create new instances.

Method Types

Creating an NSHost

+ currentHost
+ hostWithAddress:
+ hostWithName:

Getting host names and addresses

– address
– addresses
– name
– names

Comparing instances

– isEqualToHost:

Controlling the cache

+ setHostCacheEnabled:
+ isHostCacheEnabled
+ flushHostCache

Class Methods

currentHost

+ (NSHost *)**currentHost**

Returns an NSHost representing the host that the process is running on.

See also: + **hostWithAddress:**, + **hostWithName:**

flushHostCache

+ (void)**flushHostCache**

Releases the cache of existing NSHost instances so that subsequent requests for NSHost objects create new instances. NSHost instances that were retained before this method was invoked remain valid.

See also: + **isHostCacheEnabled**, + **setHostCacheEnabled:**

hostWithAddress:

+ (NSHost *)**hostWithAddress:**(NSString *)*address*

Returns the NSHost that has the Internet address *address*. If caching is turned on and the cache already contains an NSHost with *address*, returns that object. Otherwise, this method creates a new instance and returns it. *address* should be an NSString containing an Internet address in decimal notation, such as "127.0.0.1".

See also: + **hostWithName:**, + **setHostCacheEnabled:**

hostWithName:

+ (NSHost *)**hostWithName:**(NSString *)*name*

Returns the NSHost named *name*. If caching is turned on and the cache already contains an NSHost with *name*, returns that object. Otherwise, this method creates a new instance and returns it. *name* can be either a simple host name, such as “sales”, or a fully qualified domain name, such as “sales.anycorp.com”.

See also: + **hostWithAddress:**, + **setHostCacheEnabled:**

isHostCacheEnabled

+ (BOOL)**isHostCacheEnabled**

Returns YES if the NSHost class checks the cache before creating new instances, NO if it always creates new instances.

See also: + **setHostCacheEnabled:**, + **flushHostCache**

setHostCacheEnabled:

+ (void)**setHostCacheEnabled:**(BOOL)*flag*

Sets whether the NSHost class caches instances as it creates them to avoid creating duplicate instances. If *flag* is YES, the NSHost class checks the cache before creating new instances. If *flag* is NO, it always creates new instances. Caching is turned on by default.

This method doesn’t flush the cache. If you turn caching off and then back on, new requests for hosts use what was in the cache at the time caching was turned off. However, NSHost instances created while caching is turned off aren’t entered into the cache.

See also: + **isHostCacheEnabled**, + **flushHostCache**

Instance Methods

address

– (NSString *)**address**

Returns an arbitrary Internet address of the receiver as a decimal NSString—for example, “192.42.172.1”.

See also: – **addresses**, – **name**

addresses

– (NSArray *)**addresses**

Returns all of the Internet addresses of the receiver as decimal NSStrings—for example, “192.42.172.1”.

See also: – **address**, – **names**

isEqualToHost:

– (BOOL)**isEqualToHost:**(NSHost *)*aHost*

Returns YES if the receiver represents the same host as *aHost*, NO otherwise. Two NSHost objects represent the same host if they share at least one address.

See also: – **addresses**

name

– (NSString *)**name**

Returns an arbitrary name for the receiver. This name can be a fully qualified domain name such as “sales.anycorp.com” or a simple host name such as “sales”.

See also: – **address**, – **names**

names

– (NSArray *)**names**

Returns all of the names for the receiver as NSStrings. Host names can be fully qualified domain names, such as “sales.anycorp.com”, or simple host names, such as “sales”.

See also: – **addresses**, – **name**

NSInvocation

Inherits From:	NSObject
Conforms To:	NSCoding NSObject (NSObject)
Declared In:	Foundation/NSInvocation.h

Class Description

An NSInvocation is an Objective-C message rendered static, an action turned into an object. NSInvocation objects are used to store and forward messages between objects and between applications, primarily by NSTimers and the distributed objects system. An NSInvocation contains all the elements of an Objective-C message: a target, a selector, arguments, and the return value. Each of these elements can be set directly, and the return value is set automatically when the NSInvocation is dispatched.

An NSInvocation can be repeatedly dispatched to different targets; its arguments can be modified between dispatch for varying results; even its selector can be changed to another with the same method signature (argument and return types). This makes it useful for repeating messages with many arguments and variations; rather than retyping a slightly different expression for each message, you modify the NSInvocation as needed each time before dispatching it to a new target.

Creating an NSInvocation requires several steps. Consider this method of the custom class MyCalendar:

– (BOOL)**updateAppointmentsForDate:**(NSDate *)*aDate*

updateAppointmentsForDate: takes an NSDate as its only argument, and returns YES or NO depending on whether the appointments could be updated without conflicts. The following code fragment sets up an NSInvocation for it:

```
SEL theSelector;
NSMethodSignature *aSignature;
NSInvocation *anInvocation;

theSelector = @selector(updateAppointmentsForDate:);
aSignature = [MyCalendar instanceMethodSignatureForSelector:theSelector]
anInvocation = [NSInvocation invocationWithMethodSignature:aSignature];
[anInvocation setSelector:theSelector];
```

The first two lines get the NSMethodSignature for the **updateAppointmentsForDate:** method. The last two lines actually create the NSInvocation and set its selector. Note that the selector can be set to any selector matching the signature of **updateAppointmentsForDate:**. Any of these methods can be used with **anInvocation**:

– (BOOL)**clearAppointmentsForDate:**(NSDate *)*aDate*
– (BOOL)**isAvailableOnDate:**(NSDate *)*aDate*
– (BOOL)**setMeetingTime:**(NSDate *)*aDate*

Before being dispatched, **anInvocation** must have its target and arguments set:

```
MyCalendar *userDatebook;    /* Assume this exists. */
NSDate *todaysDate;          /* Assume this exists. */

[anInvocation setTarget:userDatebook];
[anInvocation setArgument:&todaysDate atIndex:2];
```

setArgument:atIndex: sets the specified argument to the value supplied. Every method has two hidden arguments, the target and selector (whose indices are 0 and 1), so the first argument that needs to be set is actually at index 2. In this case, **todaysDate** will be the NSDate argument to **updateAppointmentsForDate:**.

To dispatch the NSInvocation, send an **invoke** or **invokeWithTarget:** message. **invoke** only produces a result if the NSInvocation has a target set. Once dispatched, the NSInvocation contains the return value of the message, which **getReturnValue:** produces:

```
BOOL result;

[anInvocation invoke];
[anInvocation getReturnValue:&result];
```

Saving NSInvocations for Later Use

Because an NSInvocation doesn't always need to retain its arguments, by default it doesn't do so. This can cause object arguments as well as the target to become invalid if they're automatically released. If you plan to cache an NSInvocation or dispatch it repeatedly during the execution of your application, you should send it a **retainArguments** message. This method retains the target and all object arguments, and copies C strings so that they're not lost because another object frees them.

Using NSInvocations with NSTimers

Suppose the NSInvocation created above is being used in a time-management application that allows multiple users to set appointments for others, such as group meetings. This application might allow a user's calendar to be automatically updated every few minutes, so that the user always knows what his schedule looks like. Such automatic updating can be accomplished by setting up NSTimer objects with NSInvocations.

Given the NSInvocation above, this is as simple as invoking one NSTimer method:

```
[NSTimer scheduledTimerWithInterval:600
        invocation:anInvocation
        repeats:YES];
```

This line of code sets up an NSTimer to dispatch **anInvocation** every 10 minutes (600 seconds). Note that an NSTimer always instructs its NSInvocation to retain its arguments; thus, you don't need to send **retainArguments** yourself. See the NSTimer class specification for more information on timers.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

Method Types

Creating instances

+ invocationWithMethodSignature:

Accessing message elements

– setArgument:atIndex:
– getArgument:atIndex:
– setReturnValue:
– getReturnValue:
– setSelector:
– selector
– setTarget:
– target

Managing arguments

– argumentsRetained
– retainArguments

Dispatching an invocation

– invoke
– invokeWithTarget:

Getting the method signature

– methodSignature

Class Methods

invocationWithMethodSignature:

+ (NSInvocation *)**invocationWithMethodSignature:**(NSMethodSignature *)*signature*

Returns an NSInvocation object able to construct messages using selectors described by *signature*. The new object must have its selector set with **setSelector:** and its arguments set with **setArgument:atIndex:** before it can be invoked.

Instance Methods

argumentsRetained

– (BOOL)**argumentsRetained**

Returns YES if the NSInvocation has retained its arguments, NO otherwise.

See also: – **retainArguments**

getArgument:atIndex:

– (void)**getArgument:**(void *)*buffer* **atIndex:**(int)*index*

Copies the argument stored at *index* into the storage pointed to by *buffer*. Indices 0 and 1 indicate the hidden arguments **self** and **_cmd**, respectively; these values can be retrieved directly with the **target** and **selector** methods. Use indices 2 and greater for the arguments normally passed in a message.

buffer must be large enough to accommodate the argument value. This method raises NSInvalidArgumentException if *index* is greater than the actual number of arguments for the selector.

See also: – **setArgument:atIndex:**, – **numberOfArguments** (NSMethodSignature)

getReturnValue:

– (void)**getReturnValue:**(void *)*buffer*

Copies the invocation's return value into the storage pointed to by *buffer*, which should be large enough to accommodate the value. Use NSMethodSignature's **methodReturnLength** method to determine the size needed for **buffer**:

```
unsigned int length = [[myInvocation methodSignature] methodReturnLength];
buffer = (void *)malloc(length);
```

If the NSInvocation has never been invoked the result of this method is undefined.

See also: – **setReturnValue:**, – **methodReturnType** (NSMethodSignature)

invoke

– (void)**invoke**

Uses **invokeWithTarget:** to send the NSInvocation's message with arguments to its target. The NSInvocation's target, selector, and argument values must be set before this method is invoked.

See also: – **getReturnValue:**, – **setSelector:**, – **setTarget:**, – **setArgument:atIndex:**

invokeWithTarget:

– (void)**invokeWithTarget:**(id)*anObject*

Sends the NSInvocation's message with arguments to *anObject* and sets the return value. Doesn't set the NSInvocation's target. The NSInvocation's selector and argument values must be set before this method is invoked.

See also: – **getReturnValue:**, – **invoke**, – **setSelector:**, – **setTarget:**, – **setArgument:atIndex:**

methodSignature

– (NSMethodSignature *)**methodSignature**

Returns the invocation's method signature.

retainArguments

– (void)**retainArguments**

If the NSInvocation hasn't already done so, retains the NSInvocation's target and all object arguments, and copies all C string arguments. Before this method is invoked, **argumentsRetained** returns NO; after, it returns YES.

For efficiency, newly created NSInvocations don't retain or copy their arguments, nor do they retain their targets or copy C strings. You should instruct an NSInvocation to retain its arguments if you intend to cache it, since the arguments may otherwise be released before the NSInvocation is invoked. NSTimers always instruct their NSInvocations to retain their arguments, for example, since there's usually a delay before an NSTimer fires.

selector

– (SEL)**selector**

Returns the NSInvocation's selector, or 0 if it hasn't been set.

See also: – **setSelector:**

setArgument:atIndex:

– (void)**setArgument:**(void *)*buffer* **atIndex:**(int)*index*

Copies the contents of *buffer* as the argument at *index*. Indices 0 and 1 indicate the hidden arguments **self** and **_cmd**, respectively; these values should be set directly with the **setTarget:** and **setSelector:** methods. Use indices 2 and greater for the arguments normally passed in a message. The number of bytes copied is determined by the argument size.

This method raises NSInvalidArgumentException if the value of *index* is greater than the actual number of arguments for the selector.

See also: – **getArgument:atIndex:**, – **numberOfArguments** (NSMethodSignature)

setReturnValue:

– (void)**setReturnValue:**(void *)*buffer*

Copies the contents of *buffer* as the NSInvocation's return value. This is normally set when you send an **invoke** or **invokeWithTarget:** message.

See also: – **getReturnValue:**, – **methodReturnLength** (NSMethodSignature),
– **methodReturnType** (NSMethodSignature)

setSelector:

– (void)**setSelector:**(SEL)*selector*

Sets the NSInvocation's selector to *selector*.

See also: – **selector**

setTarget:

– (void)**setTarget:**(id)*anObject*

Sets the NSInvocation's target to *anObject*. The target is the receiver of the message sent by **invoke**.

See also: – **target**, – **invokeWithTarget:**

target

– (id)**target**

Returns the NSInvocation's target, or **nil** if the NSInvocation has no target.

See also: – **setTarget:**

NSLock

Inherits From:	NSObject
Conforms To:	NSLocking NSObject (NSObject)
Declared In:	Foundation/NSLock.h

Class Description

An NSLock object is used to coordinate the operation of multiple threads of execution within the same application. An NSLock object can be used to mediate access to an application's global data or to protect a critical section of code, allowing it to run atomically.

An NSLock object represents a lock that can be acquired by only a single thread at a time. While one thread holds the lock, any other thread is prevented from doing so until the owner relinquishes the lock. An application can have multiple NSLock objects, each protecting different sections of code. It's safest to create all of the locks before the application becomes multi-threaded, to avoid race conditions. If you want to create additional locks after the application becomes multi-threaded, you should create the new lock inside a criticial code section that is itself protected by an existing lock.

The basic interface to NSLock is declared by the NSLocking protocol, which declares the **lock** and **unlock** methods. To this base, NSLock adds the **tryLock** and **lockBeforeDate:** methods. Whereas the **lock** method declared in the protocol doesn't return until it is successful, the methods declared in this class add more flexible means of acquiring a lock.

An NSLock could be used to coordinate the updating of a visual display shared by a number of threads involved in a single calculation:

```
BOOL moreToDo = YES;
NSLock *theLock = [[NSLock alloc] init];
...
while (moreToDo) {
    /* Do another increment of calculation */
    /* until there's no more to do. */
    if ([theLock tryLock]) {
        /* Update display used by all threads. */
        [theLock unlock];
    }
}
```

The NSLock, NSConditionLock, and NSRecursiveLock classes all adopt the NSLocking protocol and offer various additional features and performance characteristics. See the NSConditionLock and NSRecursiveLock class descriptions for more information.

Adopted Protocols

NSLocking

– lock
– unlock

Method Types

Acquiring a lock

– lockBeforeDate:
– tryLock

Instance Methods

lockBeforeDate:

– (BOOL)**lockBeforeDate:**(NSDate *)*limit*

Attempts to acquire a lock before the date represented by *limit*. The thread is blocked until the receiver acquires the lock or *limit* is reached. Returns YES if the lock is acquired within this time limit. Returns NO if the time limit expires before a lock can be acquired.

tryLock

– (BOOL)**tryLock**

Attempts to acquire a lock. Returns immediately, with a value of YES if successful and NO otherwise.

NSMethodSignature

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSMethodSignature.h

Class Description

An NSMethodSignature records type information for the arguments and return value of a method. It's used to forward messages that the receiving object doesn't respond to—most notably in the case of distributed objects. An NSMethodSignature is typically created using NSObject's **methodSignatureForSelector:** instance method. It's then used to create an NSInvocation, which is passed as the argument to a **forwardInvocation:** message to send the invocation on to whatever other object can handle the message. In the default case, NSObject invokes **doesNotRecognizeSelector:**, which raises an exception. For distributed objects, the NSInvocation is encoded using the information in the NSMethodSignature and sent to the real object represented by the receiver of the message.

An NSMethodSignature presents its argument types by index with the **getArgumentTypeAtIndex:** method. The hidden arguments for every method, **self** and **_cmd**, are at indices 0 and 1 respectively. The arguments normally specified in a message invocation follow these. In addition to the argument types, an NSMethodSignature offers the total number of arguments with **numberOfArguments**, the total stack frame length occupied by all arguments with **frameLength** (this varies with hardware architecture), and the length and type of the return value with **methodReturnLength** and **methodReturnType**. Finally, applications using distributed objects can determine if the method is asynchronous with the **isOneway** method.

For more information about the nature of a method, including the hidden arguments, see *Object-Oriented Programming and the Objective-C Language*.

Method Types

Querying attributes

– frameLength
– getArgumentTypeAtIndex:
– isOneway
– numberOfArguments
– methodReturnLength
– methodReturnType

Instance Methods

getArgumentTypeAtIndex:

– (const char *)**getArgumentTypeAtIndex:**(unsigned int)*index*

Returns the type encoding for the argument at *index*. Indices begin with 0. The hidden arguments **self** (of type **id**) and **_cmd** (of type SEL) are at indices 0 and 1; method-specific arguments begin at index 2. Raises NSInvalidArgumentException if *index* is too large for the actual number of arguments.

Argument types are given as C strings with Objective-C type encoding. This encoding is implementation-specific, so OpenStep applications should use it with caution.

frameLength

– (unsigned int)**frameLength**

Returns the number of bytes that the arguments, taken together, occupy on the stack. This number varies with the hardware architecture the application runs on.

isOneway

– (BOOL)**isOneway**

Returns YES if the method is asynchronous when invoked through distributed objects. In this case the sender of the remote message doesn't block awaiting a reply. Returns NO otherwise.

methodReturnLength

– (unsigned int)**methodReturnLength**

Returns the number of bytes required for the return value.

See also: – **methodReturnType**

methodReturnType

– (char *)**methodReturnType**

Returns a C string encoding the return type of the method in Objective-C type encoding. This encoding is implementation-specific, so OpenStep applications should use it with caution.

See also: – **methodReturnLength**

numberOfArguments

– (unsigned int)**numberOfArguments**

Returns the number of arguments recorded in the NSMethodSignature. This is at least 2, since an NSMethodSignature includes the hidden arguments **self** and **_cmd**, which are the first two arguments passed to every method implementation.

NSNotification

Inherits From:	NSObject
Conforms To:	NSCoding, NSCopying NSObject (NSObject)
Declared In:	Foundation/NSNotification.h

Class Description

NSNotification objects encapsulate information so that it can be broadcast to other objects by an NSNotificationCenter object.

Notifications and their Rationale

The standard way to pass information between objects is message passing—one object invokes the method of another object. However, message passing requires that the object sending the message know who the receiver is and what messages it responds to. At times, this tight coupling of two objects is undesirable—most notably because it would join together two otherwise independent subsystems. For these cases, a broadcast model is introduced: An object posts a notification, which is dispatched to the appropriate observers through an NSNotificationCenter object, or simply *notification center*.

An NSNotification object (referred to as a *notification*) contains a name, an object, and an optional dictionary. The name is a tag identifying the notification. The object is any object that the poster of the notification wants to send to observers of that notification (typically, it is the object that posted the notification). The dictionary stores other related objects if any.

Any object may post a notification. Other objects can register themselves as observers to receive notifications when they are posted. The object posting the notification, the object included in the notification, and the observer of the notification may all be different objects or the same object. Objects that post notifications need not know anything about the observers. On the other hand, observers need to know at least the notification name and keys to the dictionary if provided.

NSNotification objects are immutable objects.

Notification Centers

The notification center manages the sending and receiving of notifications. When an object wants to receive a certain notification, it registers itself with the notification center. When an object has a notification to send, it sends it to the notification center. When the notification center receives a notification, it passes that

notification along to all objects registered to receive it. (See the NSNotificationCenter class specification for more on posting notifications.)

This notification model frees an object from concern about what objects it should send information to. Any object may simply post a notification without knowing what objects—if any—are receiving the notification. However, objects receiving notifications do need to know at least the notification name if not the type of information the notification contains. The notification center takes care of broadcasting notifications to registered observers. Another benefit of this model is to allow multiple objects to listen for notifications, which would otherwise be cumbersome.

You can create a notification object with the class methods **notificationWithName:object:** or **notificationWithName:object:userInfo:**. However, you don't usually create your own notifications directly. The NSNotificationCenter methods **postNotificationName:object:** and **postNotificationName: object:userInfo:** allow you to conveniently post a notification without creating it first.

Notification and Delegation

Using the notification system is similar to using delegates, but it has these advantages:

- Any number of objects may receive the notification, not just the delegate object. This precludes returning a value.
- An object may receive any message you like from the notification center, not just the predefined delegate methods.
- The object posting the notification does not even have to know the observer exists.

NSCopying Protocol

The NSNotification class adopts the NSCopying protocol, making it possible to treat notifications as context-independent values that can be copied and reused. You can store a notification for later use or use the Distributed Objects system to send a notification to another process. The NSCopying protocol essentially allows clients to deal with notifications as first class values that can be copied by collections. You can put notifications in an array and send the **copy** message to that array, which recursively copies every item.

Creating Subclasses

You can subclass NSNotification to contain information in addition to the notification name, object, and dictionary. This extra data must be agreed upon between notifiers and observers.

NSNotification is actually a class cluster. As such, it provides no storage for the name, object, and userInfo values. To subclass NSNotification, you must override the primitive methods **name**, **object**, and **userInfo**. Use the **init** method as the designated initializer.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

NSCopying

– copyWithZone:

Method Types

Creating a notification

+ notificationWithName:object:
+ notificationWithName:object:userInfo:

Obtaining information about a notification

– name
– object
– userInfo

Class Methods

notificationWithName:object:

+ (id)**notificationWithName:**(NSString *)*aName*
object:(id)*anObject*

Returns a notification object that associates the name *aName* with the object *anObject*.

This method copies *aName* and retains *anObject*.

See also: **– postNotificationName:object:** (NSNotificationCenter)

notificationWithName:object:userInfo:

+ (id)**notificationWithName:**(NSString *)*aName*
object:(id)*anObject*
userInfo:(NSDictionary *)*userInfo*

Returns a notification object that associates the name *aName* with the object *anObject* and the dictionary of arbitrary data, *userInfo*. The dictionary *userInfo* may be **nil**.

This method copies *aName* and retains both *anObject* and *userInfo*.

See also: **+ notificationWithName:object:, – postNotificationName:object:userInfo: (NSNotificationCenter)**

Instance Methods

name

– (NSString *)**name**

Returns the name of the notification. Examples of this might be “PortIsInvalid” or “PhoneRinging.” Typically, you invoke this method on the notification object passed to your notification-handler method. (You specify a notification-handler method when you register to receive the notification.)

Notification names can be any string. To avoid name collisions, however, you might want to use a prefix that’s specific to your application.

object

– (id)**object**

Returns the object associated with this notification. This is often the object that posted this notification. It may be **nil**.

Typically, you invoke this method on the notification object passed in to your notification-handler method. (You specify a notification-handler method when you register to receive the notification.)

For example, suppose you’ve registered an object to receive the message **handlePortDeath:** when the “PortInvalid” notification is posted to the notification center and that **handlePortDeath:** needs to access the object monitoring the port that is now invalid. **handlePortDeath:** can retrieve that object as shown here:

```
- (void)handlePortDeath:(NSNotification *)notification
{
    ...
    [self reclaimResourcesForPort:[notification object]];
    ...
}
```

userInfo

– (NSDictionary *)**userInfo**

Returns the NSDictionary associated with this notification or **nil** if there is no such object. The NSDictionary stores any additional objects that objects receiving this notification might use. For example

in the Application Kit, NSControl objects post the NSControlTextDidChangeNotification whenever the field editor (an NSText object) changes text inside the NSControl. This notification provides both the NSControl object and the field editor to objects registered to receive it. The field editor is returned when you access the dictionary, as shown here:

```
- (void)controlTextDidBeginEditing:(NSNotification *)notification
{
    NSText *fieldEditor = [[notification userInfo]
        objectForKey:@"NSFieldEditor"];/* the field editor */
    NSControl *postingObject = [notification object];
        /* The object that posted the notification. */
    ...
}
```

NSNotificationCenter

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSNotification.h

Class at a Glance

Purpose

NSNotificationCenter provides a way for objects that don't know about each other to communicate. It receives NSNotification objects and broadcasts them to all interested objects.

Principal Attributes

A table of objects that want to receive notifications, the notifications they want to receive, and about which objects.

Creation

Each task has a default notification center. You typically don't create your own.

Commonly Used Methods

+ defaultCenter	Accesses the default notification center.
– addObserver:selector:name:object:	Registers an object to receive a notification.
– postNotificationName:object:	Creates and posts a notification.
– removeObserver:	Specifies that an object no longer wants to receive notifications.

Class Description

An NSNotificationCenter object (or simply, *notification center*) is essentially a notification dispatch table. It notifies all observers of notifications meeting specific criteria. This information is encapsulated in

NSNotification objects, also known as *notifications*. Client objects register themselves as observers of specific notifications posted by other objects. When an event occurs, an object posts an appropriate notification to the notification center. (See the NSNotification class specification for more on notifications.) The notification center dispatches a message to each registered observer, passing the notification as the sole argument. It is possible for the posting object and the observing object to be the same.

Each task has a default notification center that you access with the **defaultCenter** class method.

Registering to Receive Notifications

An object registers itself to receive a notification by sending the **addObserver:selector:name:object:** method, specifying the message the notification should send, the name of the notification it wants to receive, and about which object. However, the observer need not specify both the name and the object. If it specifies only the object, it will receive *all* notifications containing that object. If the object specifies only a notification name, it will receive that notification every time it's posted, regardless of the object associated with it.

It is possible for an observer to register to receive more than one message for the same notification. In such a case, the observer will receive all messages it is registered to receive for the notification, but the order in which it receives them cannot be determined.

Creating and Posting Notifications

Normally, you create an instance of NSNotification and post it using **postNotification:**. The methods **postNotificationName:object:** and **postNotificationName:object:userInfo:** are convenient ways to post notifications without having to create an NSNotification first.

Example

As an example of using the notification center, suppose your program can perform a number of conversions on text (for instance, RTF to ASCII). You have defined a class of objects that perform those conversions, Converter. Converter objects might be added or removed during program execution. Your program has a client object that wants to be notified when converters are added or removed, allowing the application to reflect the available services in a pop-up list. The client object would register itself as an observer by sending the following messages to the notification center:

```
[[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(objectAddedToConverterList:)
    name:@"ConverterAdded" object:nil];

[[NSNotificationCenter defaultCenter] addObserver:self
    selector:@selector(objectRemovedFromConverterList:)
    name:@"ConverterRemoved" object:nil];
```

When a user installs or removes a Converter, the Converter sends one of the following messages to the notification center:

```
[[NSNotificationCenter defaultCenter]
    postNotificationName:@"ConverterAdded" object:self];
```

or

```
[[NSNotificationCenter defaultCenter]
    postNotificationName:@"ConverterRemoved" object:self];
```

The notification center identifies all observers who are interested in the “ConverterAdded” or “ConverterRemoved” notifications by invoking the method they specified in the selector argument of **addObserver:selector:name:object:**. In the case of our example observer, the selectors are **objectAddedToConverterList:** and **objectRemovedFromConverterList:**. Assume the Converter class has an instance method **converterName** that returns the name of the Converter object. Then the **objectAddedToConverterList:** method might have the following implementation:

```
- (void)objectAddedToConverterList:(NSNotification *)notification
{
    Converter *addedConverter = [notification object];

    // Add this to our popup (it will only be added if not there)...
    [myPopUpButton addItem:[addedConverter converterName]];
}
```

The Converters don’t need to know anything about the pop-up list or any other aspect of the user interface to your program.

If there are other objects of interest to the observer, place them in the notification’s optional dictionary or use **postNotificationName:object:userInfo:**.

Method Types

Accessing the default center

+ defaultCenter

Adding and removing observers

– addObserver:selector:name:object:
– removeObserver:
– removeObserver:name:object:

Posting notifications

– postNotification:
– postNotificationName:object:
– postNotificationName:object:userInfo:

Class Methods

defaultCenter

+ (NSNotificationCenter *)**defaultCenter**

Returns the current task's notification center, which is used for system notifications.

Instance Methods

addObserver:selector:name:object:

– (void)**addObserver:**(id)*anObserver*
selector:(SEL)*aSelector*
name:(NSString *)*notificationName*
object:(id)*anObject*

Registers *anObserver* to receive notifications with the name *notificationName* and/or containing *anObject*. When a notification of name *notificationName* containing the object *anObject* is posted, *anObserver* receives an *aSelector* message with this notification as the argument. The method for the selector specified in *aSelector* must have one and only one argument. If *notificationName* is **nil**, the notification center notifies the observer of all notifications with an object matching *anObject*. If *anObject* is **nil**, the notification center notifies the observer of all notifications with the name *notificationName*.

The notification center does not retain *anObserver* or *anObject*. Therefore, you should always send **removeObserver:** or **removeObserver:name:object:** to the notification center before releasing these objects.

postNotification:

– (void)**postNotification:**(NSNotification *)*notification*

Posts *notification* to the notification center. You can create *notification* with the NSNotification class method **notificationWithName:object:** or **notificationWithName:object:userInfo:**. An exception is raised if *notification* is **nil**.

See also: – **postNotificationName:object:**, – **postNotificationName:object:userInfo:**

postNotificationName:object:

– (void)**postNotificationName:**(NSString *)*notificationName*
object:(id)*anObject*

Creates a notification with the name *notificationName*, associates it with the object *anObject*, and posts it to the notification center. *anObject* is typically the object that is posting the notification. It may be **nil**.

This method invokes **postNotificationName:object:userInfo:** with a **userInfo:** argument of **nil**.

See also: – **postNotification:**

postNotificationName:object:userInfo:

– (void)**postNotificationName:**(NSString *)*notificationName*
object:(id)*anObject*
userInfo:(NSDictionary *)*userInfo*

Creates a notification with the name *notificationName*, associates it with the object *anObject* and dictionary *userInfo*, and posts it to the notification center. This method is the preferred method for posting notifications. *anObject* is typically the object that is posting the notification. It may be **nil**. *userInfo* also may be **nil**.

See also: – **postNotificationName:object:**

removeObserver:

– (void)**removeObserver:**(id)*anObserver*

Removes *anObserver* from all notification associations in the notification center. Be sure to invoke this method (or **removeObserver:name:object:**) before releasing *anObserver* or any object specified in **addObserver:selector:name:object:**.

removeObserver:name:object:

– (void)**removeObserver:**(id)*anObserver*
name:(NSString *)*notificationName*
object:(id)*anObject*

Removes *anObserver* as the observer of notifications with the name *notificationName* and object *anObject* from the notification center. Be sure to invoke this method (or **removeObserver:**) before deallocating the observer object or any object specified in **addObserver:selector:name:object:**.

If *anObserver* is **nil**, all objects are removed as observers of *notificationName* containing *anObject*. (Recall that the object that a notification contains is usually the object that posted the notification.) If *notificationName* is **nil**, *anObserver* is removed as an observer of all notifications containing *anObject*. If *anObject* is **nil**, *anObserver* is removed as an observer of *notificationName* containing any object. For example, if you wanted all objects to stop observing notifications containing the object **aWindow**, you would sent this message:

```
[[NSNotificationCenter defaultCenter] removeObserver:nil name:nil object:aWindow];
```

NSNotificationQueue

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	Foundation/NSNotificationQueue.h

Class Description

NSNotificationQueue objects (or simply, *notification queues*) act as buffers for notification centers (instances of NSNotificationCenter). A notification queue maintains notifications (instances of NSNotification) generally in a First In First Out (FIFO) order. When a notification rises to the front of the queue, the queue posts it to the notification center, which in turn dispatches the notification to all objects registered as observers.

Every thread has a default notification queue, which is associated with the default notification center for the task. You can create your own notification queues and have multiple queues per center and thread.

Coalescing Notifications

NSNotificationQueue contributes two important features to the Foundation Kit's notification mechanism: the coalescing of notifications and asynchronous posting. *Coalescing* is a process that removes notifications in the queue that are similar to the notification just queued. If the new item is similar to a notification already queued, the new one isn't queued, and all similar notifications (except the first one in the queue) are removed. However, you should not depend on this particular coalescing behavior.

You indicate the criteria for similarity by specifying NSNotificationCoalescing constants in the third argument of **enqueueNotification:postingStyle:coalesceMask:forModes:**

- **NSNotificationNoCoalescing**. Do not coalesce notifications in the queue.
- **NSNotificationCoalescingOnName**. Coalesce notifications with the same name.
- **NSNotificationCoalescingOnSender**. Coalesce notifications with the same object.

You can OR the constants together to specify more than one.

Asynchronously Posting Notifications

With NSNotificationCenter's **postNotification:** and its variants, you can post a notification immediately to a notification center. However, the invocation of the method is synchronous: Before the posting object can resume its thread of execution, it must wait until the notification center dispatches the notification to all

observers and returns. With NSNotificationQueue's **enqueueNotification:postingStyle:** and **enqueueNotification:postingStyle:coalesceMask:forModes**:, however, you can post a notification asynchronously by putting it on the queue. These methods immediately return to the invoking object after putting the notification in the queue.

Posting to a notification queue can occur in one of three different styles. The posting style is an argument to both **enqueueNotification:...** methods:

- **NSPostASAP**. The notification is posted at the end of the current notification callout or timer.
- **NSPostWhenIdle**. The notification is posted when the run loop is idle.
- **NSPostNow**. The notification is posted immediately after coalescing.

Note: See "Enqueuing with Different Posting Styles," below, for details on and examples of enqueuing notifications with the three **postingStyle:** constants.

What is the difference between enqueuing notifications with NSPostNow and posting notifications (using NSNotificationCenter's **postNotification...** methods)? Both post notifications immediately (but synchronously) to the notification center. The difference is that **enqueueNotification:...** (with NSPostNow as the posting style) coalesces notifications in the queue before posting while **postNotification:** does not.

enqueueNotification:postingStyle:coalesceMask:forModes: also allows you to control the posting of a notification based on the run loop mode. For example, if you specify NSModalPanelRunLoopMode, the notification will not be posted unless the current run loop is in NSModalPanelRunLoopMode. See the NSRunLoop class specification for more information on run loop modes.

Enqueuing with Different Posting Styles

Any notification queued with the NSPostASAP style is posted to the notification center when the code executing in the current run loop callout completes. Callouts can be Application Kit event messages, file descriptor changes, timers, or other asynchronous notifications. You typically use the NSPostASAP posting style for an expensive resource, such as the Display PostScript server. When many clients draw on the window buffer during a callout, it's expensive to flush the buffer to the Display PostScript server after every draw operation. So in this case, each **draw...** method enqueues some notification such as "FlushTheServer" with coalescing on name and object specified, and a posting style of NSPostASAP. As a result, only one of those notifications is dispatched at the end of the current callout, and the window buffer is flushed only once.

A notification queued with the NSPostWhenIdle style is posted only when the run loop is in a wait state. In this state, there's nothing in the run loop's input channels, be it timers or other asynchronous notifications. A typical example of queuing with the NSPostWhenIdle style occurs when the user types text, and the program displays the size of the text in bytes somewhere. It would be very expensive (and not very useful) to update the text field size after each character the user types, especially if the user types quickly. In this case, the program queues a notification, such as "ChangeTheDisplayedSize," with coalescing turned on and a posting style of NSPostWhenIdle after each character typed. When the user stops typing, the single "ChangeTheDisplayedSize" notification in the queue (due to coalescing) is posted when the run loop is in

a wait state, and the display is updated. Note that a run loop that is about to exit (which occurs when all of the input channels have expired) is not in a wait state and thus will not post a notification.

A notification queued with NSPostNow is posted immediately after coalescing to the notification center. You queue a notification with NSPostNow (or post one with NSNotificationCenter's **postNotification:**) when you do not require asynchronous calling behavior. For many programming situations, synchronous behavior is not only allowable but desirable: You want the notification center to return after dispatching so you can be sure that observing objects have received the notification. Of course, you should use **enqueueNotification...** with NSPostNow rather than use **postNotification:** when there are similar notifications in the queue that you want to remove through coalescing.

Method Types

Creating and Initializing Notification Queues

+ defaultQueue
– init
– initWithNotificationCenter:

Inserting and Removing Notifications From a Queue

– dequeueNotificationsMatching:coalesceMask:
– enqueueNotification:postingStyle:
– enqueueNotification:postingStyle:coalesceMask:forModes:

Class Methods

defaultQueue

+ (NSNotificationQueue *)**defaultQueue**

Returns the default NSNotificationQueue object for the current thread. This object always uses the default notification center object for the same task.

Instance Methods

dequeueNotificationsMatching:coalesceMask:

– (void)**dequeueNotificationsMatching:**(NSNotification *)*notification*
coalesceMask:(unsigned int)*coalesceMask*

Removes all notifications from the queue that match *notification*'s attributes as specified by *coalesceMask*. The mask (set using NSNotificationCoalescing constants) can specify notification name, notification object, or both.

enqueueNotification:postingStyle:

– (void)**enqueueNotification:**(NSNotification *)*notification*
postingStyle:(NSPostingStyle)*postingStyle*

Puts *notification* in the queue. The queue posts *notification* to the notification center at the time indicated by *postingStyle.* The notification queue posts in all run loop modes, and it coalesces only notifications in the queue that match both the notification's name and object.

This method invokes **enqueueNotification:postingStyle:coalesceMask:forModes:**.

enqueueNotification:postingStyle:coalesceMask:forModes:

– (void)**enqueueNotification:**(NSNotification *)*notification*
postingStyle:(NSPostingStyle)*postingStyle*
coalesceMask:(unsigned int)*coalesceMask*
forModes:(NSArray *)*modes*

Puts *notification* in the queue. The queue posts *notification* to the notification center at the time indicated by *postingStyle,* but only if the run loop is in a mode identified by one of the string objects in the *modes* array. The notification queue coalesces related notifications in the queue as specified by *coalesceMask* (set using NSNotificationCoalescing constants). If *modes* is **nil**, all run loop modes are valid for posting.

init

– (id)**init**

Initializes and returns an NSNotificationQueue that uses the default notification center to post notifications. This method invokes **initWithNotificationCenter:** with the default notification center as the argument.

initWithNotificationCenter:

– (id)**initWithNotificationCenter:**(NSNotificationCenter *)*notificationCenter*

Initializes and returns an NSNotificationQueue that uses the notification center specified in *notificationCenter* to post notifications. This method is the designated initializer.

NSNumber

Inherits From: NSValue : NSObject

Conforms To: NSCoding (NSValue)
NSCopying (NSValue)
NSObject (NSObject)

Declared In: Foundation/NSValue.h
Foundation/NSDecimalNumber.h

Class at a Glance

Purpose

An NSNumber object serves as an object wrapper for C numeric data items, allowing them to be stored in collections such as NSArray and NSDictionary objects.

Creation

+ numberWith*Type*:	Returns an initialized NSNumber of the specified type.

Commonly Used Methods

– *type*Value	Returns the value of an NSNumber as a specific numeric type.
– compare:	Compares two NSNumbers.

Class Description

NSNumber is a subclass of NSValue that offers a value as any C scalar (numeric) type. It defines a set of methods specifically for setting and accessing the value as a signed or unsigned **char**, **short int**, **int**, **long**

int, **long long int**, **float**, or **double**, or as a BOOL. It also defines a **compare:** method to determine the ordering of two NSNumber objects.

An NSNumber records the numeric type it's created with, and uses the C rules for numeric conversion when comparing NSNumbers of different numeric types and when returning values as C numeric types. See any standard C reference for information on type conversion.

Creating a Subclass of NSNumber

As with any class cluster, if you create a subclass of NSNumber, you have to override the primitive methods of its superclass, NSValue. Furthermore, there is a restricted set of return values that your implementation of the NSValue method **objCType** can return, in order to take advantage of the abstract implementations of the non-primitive methods. The valid return values are "c", "C", "s", "S", "i", "I", "l", "L", "q", "Q", "f", and "d".

Method Types

Creating an NSNumber

+ numberWithBool:
+ numberWithChar:
+ numberWithDouble:
+ numberWithFloat:
+ numberWithInt:
+ numberWithLong:
+ numberWithLongLong:
+ numberWithShort:
+ numberWithUnsignedChar:
+ numberWithUnsignedInt:
+ numberWithUnsignedLong:
+ numberWithUnsignedLongLong:
+ numberWithUnsignedShort:
– initWithBool:
– initWithChar:
– initWithDouble:
– initWithFloat:
– initWithInt:
– initWithLong:
– initWithLongLong:
– initWithShort:
– initWithUnsignedChar:
– initWithUnsignedInt:
– initWithUnsignedLong:
– initWithUnsignedLongLong:
– initWithUnsignedShort:

Accessing numeric values

– boolValue
– charValue
– decimalValue
– descriptionWithLocale:
– description
– doubleValue
– floatValue
– intValue
– longLongValue
– longValue
– shortValue
– stringValue
– unsignedCharValue
– unsignedIntValue
– unsignedLongLongValue
– unsignedLongValue
– unsignedShortValue

Comparing NSNumbers

– compare:
– isEqualToNumber:

Class Methods

numberWithBool:

+ (NSNumber *)**numberWithBool:**(BOOL)*value*

Creates and returns an NSNumber containing *value*, treating it as a BOOL.

numberWithChar:

+ (NSNumber *)**numberWithChar:**(char)*value*

Creates and returns an NSNumber containing *value*, treating it as a **signed char**.

numberWithDouble:

+ (NSNumber *)**numberWithDouble:**(double)*value*

Creates and returns an NSNumber containing *value*, treating it as a **double**.

numberWithFloat:

+ (NSNumber *)**numberWithFloat:**(float)*value*

Creates and returns an NSNumber containing *value*, treating it as a **float**.

numberWithInt:

+ (NSNumber *)**numberWithInt:**(int)*value*

Creates and returns an NSNumber containing *value*, treating it as a **signed int**.

numberWithLong:

+ (NSNumber *)**numberWithLong:**(long int)*value*

Creates and returns an NSNumber containing *value*, treating it as a **signed long int**.

numberWithLongLong:

+ (NSNumber *)**numberWithLongLong:**(long long int)*value*

Creates and returns an NSNumber containing *value*, treating it as a **signed long long int**.

numberWithShort:

+ (NSNumber *)**numberWithShort:**(short int)*value*

Creates and returns an NSNumber containing *value*, treating it as a **signed short int**.

numberWithUnsignedChar:

+ (NSNumber *)**numberWithUnsignedChar:**(unsigned char)*value*

Creates and returns an NSNumber containing *value*, treating it as an **unsigned char**.

numberWithUnsignedInt:

+ (NSNumber *)**numberWithUnsignedInt:**(unsigned int)*value*

Creates and returns an NSNumber containing *value*, treating it as an **unsigned int**.

numberWithUnsignedLong:

+ (NSNumber *)**numberWithUnsignedLong:**(unsigned long int)*value*

Creates and returns an NSNumber containing *value*, treating it as an **unsigned long int**.

numberWithUnsignedLongLong:

+ (NSNumber *)**numberWithUnsignedLongLong:**(unsigned long long int)*value*

Creates and returns an NSNumber containing *value*, treating it as an **unsigned long long int**.

numberWithUnsignedShort:

+ (NSNumber *)**numberWithUnsignedShort:**(unsigned short int)*value*

Creates and returns an NSNumber containing *value*, treating it as an **unsigned short int**.

Instance Methods

boolValue

– (BOOL)**boolValue**

Returns the receiver's value as a BOOL, converting it as necessary.

Note: The value returned by this method isn't guaranteed to be one of YES or NO. A zero value always means NO or false, but any nonzero value should be interpreted as YES or true.

charValue

– (char)**charValue**

Returns the receiver's value as a char, converting it as necessary.

compare:

– (NSComparisonResult)**compare:**(NSNumber *)*aNumber*

Returns NSOrderedAscending if *aNumber*'s value is greater than the receiver's, NSOrderedSame if they're equal, and NSOrderedDescending if *aNumber*'s value is less than the receiver's.

compare: follows the standard C rules for type conversion. For example, if you compare an NSNumber that has an integer value with an NSNumber that has a floating point value, the integer value is converted to a floating point value for comparison.

decimalValue

– **(NSDecimal)decimalValue**

Returns the receiver's value, expressed as an NSDecimal struct. **The value returned isn't guaranteed to be exact for float and double values.**

description

– **(NSString *)description**

Overrides the NSObject **description** method to invoke **descriptionWithLocale:**. In other words, instead printing the name of the receiver's class and the hexadecimal value of its **id** like the NSObject implementation, returns a string representation of the receiver's value.

descriptionWithLocale:

– (NSString *)**descriptionWithLocale:**(NSDictionary *)*aLocale*

Returns an NSString that represents the contents of the receiver. For example, if you have an NSNumber that has the integer value 522, sending it the **descriptionWithLocale:** message returns the string "522". *aLocale* specifies options used for formatting the description; use **nil** if you don't want the description formatted.

To obtain the string representation, this method invokes NSString's **initWithFormat:locale:** method, supplying the format based on the type the NSNumber was created with:

Data Type	Format Specification
char	%i
double	%0.16g
float	%0.7g
int	%i
long	%li
long long	%li
short	%hi
unsigned char	%u
unsigned int	%u
unsigned long	%lu
unsigned long long	%lu
unsigned short	%hu

See also: – **stringValue**

doubleValue

– (double)**doubleValue**

Returns the receiver's value as a double, converting it as necessary.

floatValue

– (float)**floatValue**

Returns the receiver's value as a float, converting it as necessary.

initWithBool:

– (id)**initWithBool:**(BOOL)*value*

Initializes a newly allocated NSNumber to contain *value*, treated as a BOOL.

initWithChar:

– (id)**initWithChar:**(char)*value*

Initializes a newly allocated NSNumber to contain *value*, treated as a **signed char**.

initWithDouble:

– (id)**initWithDouble:**(double)*value*

Initializes a newly allocated NSNumber to contain *value*, treated as a **double**.

initWithFloat:

– (id)**initWithFloat:**(float)*value*

Initializes a newly allocated NSNumber to contain *value*, treated as a **float**.

initWithInt:

– (id)**initWithInt:**(int)*value*

Initializes a newly allocated NSNumber to contain *value*, treated as a **signed int**.

initWithLong:

– (id)**initWithLong:**(long int)*value*

Initializes a newly allocated NSNumber to contain *value*, treated as a **signed long int**.

initWithLongLong:

– (id)**initWithLongLong:**(long long int)*value*

Initializes a newly allocated NSNumber to contain *value*, treated as a **signed long long int**.

initWithShort:

– (id)**initWithShort:**(short int)*value*

Initializes a newly allocated NSNumber to contain *value*, treated as a **signed short int**.

initWithUnsignedChar:

– (id)**initWithUnsignedChar:**(unsigned char)*value*

Initializes a newly allocated NSNumber to contain *value*, treated as an **unsigned char**.

initWithUnsignedInt:

– (id)**initWithUnsignedInt:**(unsigned int)*value*

Initializes a newly allocated NSNumber to contain *value*, treated as an **unsigned int**.

initWithUnsignedLong:

– (id)**initWithUnsignedLong:**(unsigned long int)*value*

Initializes a newly allocated NSNumber to contain *value*, treated as an **unsigned long int**.

initWithUnsignedLongLong:

– (id)**initWithUnsignedLongLong:**(unsigned long long int)*value*

Initializes a newly allocated NSNumber to contain *value*, treated as an **unsigned long long int**.

initWithUnsignedShort:

– (id)**initWithUnsignedShort:**(unsigned short int)*value*

Initializes a newly allocated NSNumber to contain *value*, treated as an **unsigned short int**.

intValue

– (int)**intValue**

Returns the receiver's value as an int, converting it as necessary.

isEqualToNumber:

– (BOOL)**isEqualToNumber:**(NSNumber *)*aNumber*

Returns YES if the receiver and *aNumber* are equal, otherwise returns NO. Two NSNumbers are considered equal if they have the same **id**s or if they have equivalent values (as determined by the **compare:** method).

longLongValue

– (long long int)**longLongValue**

Returns the receiver's value as a long long int, converting it as necessary.

longValue

– (long int)**longValue**

Returns the receiver's value as a long int, converting it as necessary.

shortValue

– (short int)**shortValue**

Returns the receiver's value as a short int, converting it as necessary.

stringValue

– (NSString *)**stringValue**

Returns the receiver's value as a human-readable NSString, by invoking **descriptionWithLocale:** where locale is **nil.**

unsignedCharValue

– (unsigned char)**unsignedCharValue**

Returns the receiver's value as an unsigned char, converting it as necessary.

unsignedIntValue

– (unsigned int)**unsignedIntValue**

Returns the receiver's value as an unsigned int, converting it as necessary.

unsignedLongLongValue

– (unsigned long long int)**unsignedLongLongValue**

Returns the receiver’s value as an unsigned long long int, converting it as necessary.

unsignedLongValue

– (unsigned long int)**unsignedLongValue**

Returns the receiver’s value as an unsigned long int, converting it as necessary.

unsignedShortValue

– (unsigned short int)**unsignedShortValue**

Returns the receiver’s value as an unsigned short int, converting it as necessary.

NSNumberFormatter

Inherits From:	NSFormatter : NSObject
Conforms To:	NSObject (NSObject) NSCoding NSCopying
Declared In:	Foundation/NSNumberFormatter.h

Class Description

Instances of NSNumberFormatter format the textual representation of cells that contain NSDecimalNumbers and convert textual representations of numeric values into NSDecimalNumbers. The representation encompasses integers, floats, and doubles; floats and doubles can be formatted to a specified decimal position. NSNumberFormatters can also impose ranges on the numeric values that cells can accept.

NSControl provides delegation methods that permit you to validate cell contents and to handle errors in formatting. See the specification of the NSFormatter class for details.

When a cell with a NSNumberFormatter is copied, the new cell retains the NSNumberFormatter object instead of copying it. You remove an NSNumberFormatter from a cell by specifying **nil** as the argument of NSCell's **setFormatter:** method.

Instances of NSNumberFormatter are mutable.

Creating an Instance of NSNumberFormatter

The easiest way to use NSNumberFormatter is to drag a formatter from the palette onto a control in Interface Builder. However, if you're not using Interface Builder to create your user interface or if you simply want more fine-grained control over an NSNumberFormatter (for example, to change the text attributes of the values displayed), you can create and manipulate instances of the class programmatically.

To create an NSNumberFormatter, allocate an instance of NSNumberFormatter and use one or more of NSNumberFormatter's "set format" methods to set its format. You then use NSCell's **setFormatter:** method to associate the NSNumberFormatter instance with a cell.

For example, the following code excerpt creates an instance of NSNumberFormatter, sets its formatting for positive, zero, and negative values, and applies it to the cell of an NSTextField using NSCell's **setFormatter:** method:

```
NSNumberFormatter *numberFormatter =
    [[[NSNumberFormatter alloc] init] autorelease];
[numberFormatter setFormat:@"$#,###.00;0.00;($#,##0.00)"];
[[textField cell] setFormatter:numberFormatter];
```

The value of a cell (NSCell) is represented by an object, which in this case would typically be an NSDecimalNumber object. When this value needs to be displayed or edited, the cell passes its object to the NSNumberFormatter instance, which returns the formatted string. When the user enters a string, or when a string is programmatically written in a cell (using **setStringValue:**), the cell obtains the equivalent NSDecimalNumber object from the NSNumberFormatter.

Using the setFormat: Method

The most common technique for assigning a format to an NSNumberFormatter object is to use the method **setFormat:**, as shown in the preceding section. This method takes as an argument an NSString whose contents can be one of the following:

- *@"positiveFormat"*

  For example, @"$###,##0.00" (the syntax of format strings is discussed in the following section).

- *@"positiveFormat;negativeFormat"*

  For example, @"###,##0.00;(###,##0.00)".

- *@"positiveFormat;zeroFormat;negativeFormat"*

  For example, @"$###,###.00;0.00;($###,##0.00)". Note that zero formats are treated as string constants.

As implied in the above list, you're only required to specify a format for positive values. If you don't specify a format for negative and zero values, a default format based on the positive value format is used. For example, if your positive value format is "#,##0.00", an input value of "0" will be displayed as "0.00".

If you don't specify a format for negative values, the format specified for positive values is used, preceded by a minus sign (-).

If you specify a separate format for negative values, its separators should be parallel to those specified in the positive format string. In NSNumberFormatter, separators are either enabled or disabled for all formats—both your negative and positive formats should therefore use the same approach.

As an alternative to using the **setFormat:** method, you can use the **setPositiveFormat:** and **setNegativeFormat:** methods.

Format String Syntax

Format strings can include the following types of characters:

- Numbers

  Format strings can include numeric characters. Wherever you include a number in a format string, the number is displayed unless an input character in the same relative position overwrites it. For example, suppose you have the positive format string @“9,990.00”, and the value 53.88 is entered into a cell to which the format has been applied. The cell would display the value as 9,953.88.

- Separators

  Format strings can include the period character (.) as a decimal separator, and comma character (,) as a thousand separator. If you want to use different characters as separators, you can set them using the **setDecimalSeparator:** and **setThousandSeparator:** methods. When you enable localization for an NSNumberFormatter object by using the method **setLocalizesFormat:**, separators are converted to characters appropriate to the environment in which the application is running.

- Placeholders

  You use the pound sign character (#) to represent numeric characters that will be input by the user. For example, suppose you have the positive format @”$#,##0.00”. If the characters 76329 were entered into a cell to which the format has been applied, they would be displayed as $76,329.00. Strictly speaking, however, you don’t need to use placeholders. The format strings @”,0.00”, @”#,#0.00, and @”#,##0.00” are functionally equivalent. In other words, including separator characters in a format string signals NSNumberFormatter to use the separators, regardless of whether you use (or where you put) placeholders. The placeholder character’s chief virtue lies in its ability to make format strings more human-readable, which is especially useful if you’re displaying formats in the user interface.

- Spaces

  To include a space in a format string, use the underscore character (_). This character inserts a space if no numeric character has been input to occupy that position.

- Currency

  The dollar sign character ($) is normally treated just like any other character that doesn’t play a special role in NSNumberFormatter. However, when you enable localization for an NSNumberFormatter object by using the method **setLocalizesFormat:**, the dollar sign character is converted to the currency symbol appropriate for the environment in which the application is running.

All other characters specified in a format string are displayed as typed. The following table shows examples of the how the value 1019.55 is displayed for different positive formats:

Format String	Display
@“#,##0.00”	1,019.55
@“$#,##0.00”	$1,019.55
@“___,__0.00”	1,019.55

Working with Values as Attributed Strings

In NSNumberFormatter, positive, negative, zero, **nil**, and “not a number” values are NSAttributedStrings. NSAttributedString objects manage character strings and associated sets of attributes (for example, font and kerning) that apply to individual characters or ranges of characters in the string. An association of characters and their attributes is called an *attributed string*. For more information on NSAttributedString, see the NSAttributedString class cluster specification in the *Foundation Kit Reference*, and the NSAttributedString Class Cluster Additions specification in the *Application Kit Reference*.

Because the values displayed by NSNumberFormatter are attributed strings, you can customize aspects of their appearance, such as their color and font. The NSNumberFormatter methods you use to do this are as follows:

– textAttributesForPositiveValues
– setTextAttributesForPositiveValues:
– textAttributesForNegativeValues
– setTextAttributesForNegativeValues:
– attributedStringForZero
– setAttributedStringForZero:
– allowsFloats
– setAttributedStringForNil:
– attributedStringForNotANumber
– setAttributedStringForNotANumber:

Using Separators

NSNumberFormatter supports two different kinds of separators: thousand and decimal. By default these separators are represented by the comma (,) and period (.) characters respectively, and by default they’re disabled.

All of the following statements have the effect of enabling thousand separators:

```
// use setFormat:
[numberFormatter setFormat:@"#,###"];

// use setHasThousandSeparators:
[numberFormatter setHasThousandSeparators:YES];

// use setThousandSeparator:
[numberFormatter setThousandSeparator:@"_"];
```

If you use the statement `[numberFormatter setHasThousandSeparators:NO]`, it disables thousand separators, even if you've set them through another means.

Both of the following statements have the effect of enabling decimal separators:

```
// use setFormat:
[numberFormatter setFormat:@"0.00"];

// use setDecimalSeparator:
[numberFormatter setDecimalSeparator:@"-"];
```

When you enable or disable separators, it affects both positive and negative formats. Consequently, both formats must use the same separator scheme.

You can use the **thousandSeparator** and **decimalSeparator** methods to return an NSString containing the character the receiver uses to represent each separator. However, this shouldn't be taken as an indication of whether separators are enabled—even when separators are disabled, an NSNumberFormatter still knows the characters it uses to represent separators.

Separators must be single characters. If you specify multiple characters in the arguments to **setThousandSeparator:** and **setDecimalSeparator:**, only the first character is used.

You can't use the same character to represent thousand and decimal separators.

NSCell Methods for Number Formatting

NSCell provides methods that give you almost the same behavior as instances of NSNumberFormatter. Send **setEntryType:** to a cell to associate it with an NSNumberFormatter object; specify the numeric format with one of the constants listed below. The constant is equivalent to an NSNumberFormatter initialized with a certain range and a conversion specifier:

Constant	Range	Specifier
NSIntType	MININT, MAXINT	%d
NSPositiveIntType	1, MAXINT	%d
NSFloatType	-MAXFLOAT, MAXFLOAT	%g

Constant	Range	Specifier
NSPositiveFloatType	MINFLOAT, MAXFLOAT	%g
NSDoubleType	-MAXDOUBLE, MAXDOUBLE	%g
NSPositiveDoubleType	MINDOUBLE, MAXDOUBLE	%g

Send NSCell's **isEntryAcceptable:** to a cell to determine if it can accept a numeric type as indicated by one of the above constants. Send NSCell's **setFloatingPointFormat:left:right:** to specify the number of digits that appear to the left and right of the decimal point. By invoking this method you do not lose any range of values for floats or values set either through **setEntryType:** or by initializing an NSNumberFormatter directly.

Note: The NSNumberFormatter approach is recommended over the NSCell methods because it allows you greater freedom in specifying the representation of numbers. However, NSCell's **setEntryType:**, **isEntryAcceptable:**, and **setFloatingPointFormat:left:right:** are OpenStep-compliant, whereas NSNumberFormatter is an extension to OpenStep.

Method Types

Set formats

– negativeFormat
– setNegativeFormat:
– positiveFormat
– setPositiveFormat:
– format
– setFormat:

Set characteristics for displaying values

– textAttributesForNegativeValues
– setTextAttributesForNegativeValues:
– textAttributesForPositiveValues
– setTextAttributesForPositiveValues:
– attributedStringForZero
– setAttributedStringForZero:
– attributedStringForNil
– setAttributedStringForNil:
– attributedStringForNotANumber
– setAttributedStringForNotANumber:

Set separators
– hasThousandSeparators
– setHasThousandSeparators:
– thousandSeparator
– setThousandSeparator:
– decimalSeparator
– setDecimalSeparator:

Enable localization
– localizesFormat
– setLocalizesFormat:

Set float behavior
– allowsFloats
– setAllowsFloats:

Set rounding behavior
– roundingBehavior
– setRoundingBehavior:

Set minimum and maximum values
– minimum
– setMinimum:
– maximum
– setMaximum:

Instance Methods

allowsFloats

– (BOOL)**allowsFloats**

Returns YES if the receiver allows as input floating point values (that is, values that include the period character (.)), NO otherwise. When this is set to NO, only integer values can be provided as input. The default is YES.

See also: – **setAllowsFloats:**

attributedStringForNil

– (NSAttributedString *)**attributedStringForNil**

Returns the NSAttributedString used to display **nil** values. By default **nil** values are displayed as an empty string.

See also: – **allowsFloats**

attributedStringForNotANumber

– (NSAttributedString *)**attributedStringForNotANumber**

Returns the NSAttributedString used to display “not a number” values. By default “not a number” values are displayed as the string “NaN”.

See also: – **attributedStringForNotANumber**

attributedStringForZero

– (NSAttributedString *)**attributedStringForZero**

Returns the NSAttributedString used to display zero values. By default zero values are displayed according to the format specified for positive values; for more discussion of this subject see the section “Creating an Instance of NSNumberFormatter” in the Class Description.

See also: – **setAttributedStringForZero:**

decimalSeparator

– (NSString *)**decimalSeparator**

Returns an NSString containing the character the receiver uses to represent decimal separators. By default this is the period character (.). Note that the return value doesn’t indicate whether decimal separators are enabled.

See also: – **setDecimalSeparator:**

format

– (NSString *)**format**

Returns an NSString containing the format being used by the receiver.

See also: – **setFormat:**

hasThousandSeparators

– (BOOL)**hasThousandSeparators**

Returns YES to indicate that the receiver’s format includes thousand separators, NO otherwise. The default is NO.

See also: – **setHasThousandSeparators:**

localizesFormat

– (BOOL)**localizesFormat**

Returns YES to indicate that the receiver localizes formats, NO otherwise. The default is NO.

See also: – **setLocalizesFormat:**

maximum

– (NSDecimalNumber *)**maximum**

Returns the highest number that is allowed as input by the receiver.

See also: – **setMaximum:**

minimum

– (NSDecimalNumber *)**minimum**

Returns the lowest number that is allowed as input by the receiver.

See also: – **setMinimum:**

negativeFormat

– (NSString *)**negativeFormat**

Returns an NSString containing the format used by the receiver to display negative numbers.

See also: – **setNegativeFormat:**, – **setFormat:**

positiveFormat

– (NSString *)**positiveFormat**

Returns an NSString containing the format used by the receiver to display positive numbers.

See also: – **setPositiveFormat:**, – **setFormat:**

roundingBehavior

– (NSDecimalNumberHandler *)**roundingBehavior**

Returns an NSDecimalNumberHandler object to indicate the rounding behavior used by the receiver.

See also: – **setRoundingBehavior:**

setAllowsFloats:

– (void)**setAllowsFloats:**(BOOL)*flag*

Sets according to *flag* whether the receiver allows as input floating point values (that is, values that include the period character (.)). By default, floating point values are allowed as input.

See also: – **allowsFloats**

setAttributedStringForNil:

– (void)**setAttributedStringForNil:**(NSAttributedString *)*newAttributedString*

Sets to *newAttributedString* the NSAttributedString the receiver uses to display **nil** values.

See also: – **allowsFloats**

setAttributedStringForNotANumber:

– (void)**setAttributedStringForNotANumber:**(NSAttributedString *)*newAttributedString*

Sets to *newAttributedString* the NSAttributedString the receiver uses to display “not a number” values.

See also: – **attributedStringForNotANumber**

setAttributedStringForZero:

– (void)**setAttributedStringForZero:**(NSAttributedString *)*newAttributedString*

Sets to *newAttributedString* the NSAttributedString the receiver uses to display zero values.

See also: – **attributedStringForZero**

setDecimalSeparator:

– (void)**setDecimalSeparator:**(NSString *)*newSeparator*

Sets to *newSeparator* the character the receiver uses as a decimal separator. If *newSeparator* contains multiple characters, only the first one is used. If you don't have decimal separators enabled through another means (such as **setFormat:**), using this method enables them.

See also: – **decimalSeparator**

setFormat:

– (void)**setFormat:**(NSString *)*aFormat*

Sets the receiver's format to the string *aFormat*. *aFormat* can consist of one, two, or three parts separated by ';'. The first part of the string represents the positive format, the second part of the string represents the zero value, and the last part of the string represents the negative format. If the string just has two parts, the first one becomes the positive format, and the second one becomes the negative format. If the string just has one part, it becomes the positive format, and default formats are provided for zero and negative values based on the positive format. For more discussion of this subject, see the section "Creating an Instance of NSNumberFormatter" in the Class Description.

The following code excerpt shows the three different approaches for setting an NSNumberFormatter object's format using **setFormat:**:

```
NSNumberFormatter *numberFormatter =
    [[[NSNumberFormatter alloc] init] autorelease];

// specify just positive format
[numberFormatter setFormat:@"$#,##0.00"];

// specify positive and negative formats
[numberFormatter setFormat:@"$#,##0.00;($#,##0.00)"];

// specify positive, zero, and negative formats
[numberFormatter setFormat:@"$#,###.00;0.00;($#,##0.00)"];
```

See also: – **format**

setHasThousandSeparators:

– (void)**setHasThousandSeparators:**(BOOL)*flag*

Sets according to *flag* whether the receiver uses thousand separators. When *flag* is NO, thousand separators are disabled for both positive and negative formats (even if you've set them through another means, such as **setFormat:**). When *flag* is YES, thousand separators are used. In addition to using this method to add thousand separators to your format, you can also use it to disable thousand separators if you've set them

using another method. The default is NO (though you in effect change this setting to YES when you set thousand separators through any means, such as **setFormat:**).

See also: – **hasThousandSeparators**

setLocalizesFormat:

– (void)**setLocalizesFormat:**(BOOL)*flag*

Sets according to *flag* whether the dollar sign character ($), decimal separator character (.), and thousand separator character (,) are converted to appropriately localized characters as specified by the user's localization preference. While this feature may be useful in certain types of applications, it's probably more likely that you would tie a particular application to a particular currency (that is, that you would "hard-code" the currency symbol and separators instead of having them dynamically change based on the user's configuration). The reason for this, of course, is that NSNumberFormatter doesn't perform currency conversions, it just formats numeric data. You wouldn't want one user interpreting the value "56324" as US currency and another user who's accessing the same data interpreting it as Japanese currency, simply based on each user's localization preferences.

See also: – **localizesFormat**

setMaximum:

– (void)**setMaximum:**(NSDecimalNumber *)*aMaximum*

Sets to *aMaximum* the highest number the receiver allows as input.

See also: – **maximum**

setMinimum:

– (void)**setMinimum:**(NSDecimalNumber *)*aMinimum*

Sets to *aMinimum* the lowest number the receiver allows as input.

See also: – **setMinimum:**

setNegativeFormat:

– (void)**setNegativeFormat:**(NSString *)*aFormat*

Sets to *aFormat* the format the receiver uses to display negative values.

See also: – **negativeFormat**, – **setFormat:**

setPositiveFormat:

– (void)**setPositiveFormat:**(NSString *)*aFormat*

Sets to *aFormat* the format the receiver uses to display positive values.

See also: – **positiveFormat**, – **setFormat:**

setRoundingBehavior:

– (void)**setRoundingBehavior:**(NSDecimalNumberHandler *)*newRoundingBehavior*

Sets to *newRoundingBehavior* the rounding behavior used by the receiver.

See also: – **roundingBehavior**

setTextAttributesForNegativeValues:

– (void)**setTextAttributesForNegativeValues:**(NSDictionary *)*newAttributes*

Sets to *newAttributes* the text attributes to be used in displaying negative values. For example, this code excerpt causes negative values to be displayed in red:

```
NSNumberFormatter *numberFormatter =
    [[[NSNumberFormatter alloc] init] autorelease];
NSMutableDictionary *newAttrs = [NSMutableDictionary dictionary];

[numberFormatter setFormat:@"$#,##0.00;($#,##0.00)"];
[newAttrs setObject:[NSColor redColor] forKey:@"NSColor"];
[numberFormatter setTextAttributesForNegativeValues:newAttrs];
[[textField cell] setFormatter:numberFormatter];
```

An even simpler way to cause negative values to be displayed in red is to include the constant `[Red]` in your format string, for example:

```
[numberFormatter setFormat:@"$#,##0.00;[Red]($#,##0.00)"];
```

Note: When you set a value's text attributes to use color, the color only appears when the value's cell doesn't have input focus. When the cell has input focus, the value is displayed in standard black.

See also: – **textAttributesForNegativeValues**

setTextAttributesForPositiveValues:

– (void)**setTextAttributesForPositiveValues:**(NSDictionary *)*newAttributes*

Sets to *newAttributes* the text attributes to be used in displaying positive values.

See also: – **textAttributesForPositiveValues**

setThousandSeparator:

– (void)**setThousandSeparator:**(NSString *)*newSeparator*

Sets to *newSeparator* the character the receiver uses as a thousand separator. If *newSeparator* contains multiple characters, only the first one is used. If you don't have thousand separators enabled through any other means (such as **setFormat:**), using this method enables them.

See also: – **thousandSeparator**

textAttributesForNegativeValues

– (NSDictionary *)**textAttributesForNegativeValues**

Returns an NSDictionary containing the text attributes that have been set for negative values.

See also: – **setTextAttributesForNegativeValues:**

textAttributesForPositiveValues

– (NSDictionary *)**textAttributesForPositiveValues**

Returns an NSDictionary containing the text attributes that have been set for positive values.

See also: – **setTextAttributesForPositiveValues:**

thousandSeparator

– (NSString *)**thousandSeparator**

Returns an NSString containing the character the receiver uses to represent thousand separators. By default this is the comma character (,). Note that the return value doesn't indicate whether thousand separators are enabled.

See also: – **setThousandSeparator:**

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.

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: – copy

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. The **load** message is sent to classes and categories that are both dynamically loaded and statically linked, but only if the newly-loaded class or category implements a method that can respond. As an example, when Interface Builder loads a palette, the **load** message 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 sending messages to 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.

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.

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)

Instance 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:**.

NSObject does not itself support the NSCopying protocol. Subclasses must support the protocol and implement the **copyWithZone:** method. A subclass version of the **copyWithZone:** method should pass the message to **super** unless the subclass descends directly from NSObject, in which case the method must not invoke **super**'s method.

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:**

performSelector:withObject:afterDelay:inModes:

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

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.

The *inModes* parameter contains an array of strings that identify the modes to associate with the timer performing the selector. This timer becomes associated with each of the given modes until its **invalidate** method is invoked.

See also: – **performSelector:withObject:afterDelay:**, **– addTimer:forMode:** (NSRunLoop),
– **invalidate** (NSTimer)

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**

NSPipe Class Cluster

Class Cluster Description

NSPipe objects provide an object-oriented interface for accessing pipes. A pipe is a one-way communications channel between related processes; one process writes data while the other process reads the data.

NSPipe

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	Foundation/NSFileHandle.h

Class Description

An NSPipe represents both ends of a pipe and enables communication through the pipe. A pipe is a one-way communications channel between related processes; one process writes data while the other process reads that data. The data that passes through the pipe is buffered; the size of the buffer is determined by the underlying operating system.

Each end point of the pipe is a file descriptor, represented by an NSFileHandle. You thus use NSFileHandle messages to read and write pipe data. A "parent" process creates the NSPipe and holds one end of it. It creates an NSTask for the other process and, before launching it, passes the other end of the pipe to that process; it does this by setting the NSTask's standard input, standard output, or standard error device to be the other NSFileHandle or the NSPipe itself (in the latter case, the type of NSFileHandle—reading or writing—is determined by the NSTask "set" method).

The following example illustrates the above procedure:

```
- (void)readTaskData:(id)sender
{
    NSTask *pipeTask = [[NSTask alloc] init];
    NSPipe *newPipe = [NSPipe pipe];
    NSFileHandle *readHandle = [newPipe fileHandleForReading];
    NSData *inData = nil;

    [pipeTask setStandardOutput:newPipe]; // write handle is closed to this process
    [pipeTask setLaunchPath:[NSHomeDirectory()
           stringByAppendingPathComponent:@"PipeTask.app/PipeTask"]];
    [pipeTask launch];

    while ((inData = [readHandle availableData]) && [inData length]) {
        [inData processData];
    }
}
```

The launched process in this example must get data and write that data, using NSFileHandle's **writeData:** , to its standard output device (obtained NSFileHandle's **fileHandleWithStandardOutput**).

When the processes have no more data to communicate across the pipe, the writing process should simply send **closeFile** to its NSFileHandle end point. This causes the process with the “read” NSFileHandle to receive an empty NSData, signalling end of data. If the “parent” process created the NSPipe with the **init** method, it should then release it.

Method Types

Creating an NSPipe

- init
+ pipe

GettingNSFileHandles for pipe

– fileHandleForReading
– fileHandleForWriting

Class Methods

pipe

+ (id)**pipe**

Returns an NSPipe that’s been sent **autorelease**. Because the returned object will be deallocated at the end of the current loop, retain the object or, better yet, use the **init** method if you intend to keep the object beyond that point. Returns **nil** if the method encounters errors while attempting to create the pipe or the NSFileHandle end points.

Instance Methods

fileHandleForReading

– (NSFileHandle *)**fileHandleForReading**

Returns an NSFileHandle that accepts messages that read pipe data: **availableData**, **readDataToEndOfFile**, **readDataOfLength:**. You don’t need to send **closeFile** to this object or explicitly release after you’re finished using it. This descriptor represented by this object is deleted and the object itself is automatically deallocated when the NSPipe is deallocated.

fileHandleForWriting

– (NSFileHandle *)**fileHandleForWriting**

Returns an NSFileHandle for writing to the pipe using NSFileHandle's **writeData:**. When you are finished writing data to this object, send it **closeFile** to delete the descriptor, which causes the reading process to receive an end-of-data signal (an empty NSData). This object is automatically deallocated when the NSPipe is deallocated.

init

– (id)**init**

Returns an NSPipe. Returns **nil** if the method encounters errors while attempting to create the pipe or the NSFileHandles that serve as end points of the pipe.

See also: + **pipe**

NSPort

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSObject (NSObject)
Declared In:	Foundation/NSPort.h

Class Description

An NSPort represents a communication channel to or from another NSPort, which typically resides in a different thread or task. The distributed objects system uses NSPort objects to send NSPortMessages back and forth. You should implement interapplication communication using distributed objects whenever possible, and use NSPorts only when necessary.

An NSPort is essentially the object form of a Mach port. To use NSPorts effectively you should be familiar with Mach ports, port access rights, and Mach messages. See the Mach OS documentation for more information. When an NSPort receives an NSPortMessage, it forwards the message to its delegate in a **handleMachMessage:** or **handlePortMessage:** message. The delegate should implement only one of these methods to process the incoming message in whatever form desired. **handleMachMessage:** provides a message as a raw Mach message beginning with a **msg_header_t** structure. **handlePortMessage:** provides a message as an NSPortMessage object, which is an object-oriented wrapper for a Mach message.

NSPort is intended to receive incoming messages that need to be added to an NSRunLoop. See the NSRunLoop class specification for more information.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

NSCopying

– copyWithZone:

Method Types

Creating instances

+ port
+ portWithMachPort:
– initWithMachPort:

Getting the Mach port

– machPort

Validation

– invalidate
– isValid

Setting the delegate

– setDelegate:
– delegate

Creating connections

– addConnection:toRunLoop:forMode:
– removeConnection:fromRunLoop:forMode:

Setting information

– sendBeforeDate:components:from:reserved:
– reservedSpaceLength

Class Methods

port

+ (NSPort *)**port**

Creates and returns a new NSPort object capable of both sending and receiving messages.

portWithMachPort:

+ (NSPort *)**portWithMachPort:**(int)*machPort*

Returns an NSPort object that uses the Mach port *machPort* (which should be of type **port_t** cast to **int**). Creates the NSPort object if necessary. Depending on the access rights for *machPort*, the new NSPort may only be able to send messages.

Instance Methods

addConnection:toRunLoop:forMode:

– (void)**addConnection:**(NSConnection *)*connection*
toRunLoop:(NSRunLoop *)*runLoop*
forMode:(NSString *)*mode*

Adds the receiver to the list of ports monitored by *runLoop* for the given input *mode*. You should not call this method directly. The method is provided for subclassers who wish to provide their own custom types of NSPort. NSConnection calls this method at the appropriate times. See the TCP Transport example in **/System/Developer/Examples/Foundation** for an example implementation.

See also: **– addPort:forMode:** (NSRunLoop)

delegate

– (id)**delegate**

Returns the NSPort's delegate.

See also: – **setDelegate:**

initWithMachPort:

– (id)**initWithMachPort:**(int)*machPort*

Initializes a newly allocated NSPort object to use the Mach port *machPort* (which should be of type **port_t** cast to **int**). Depending on the access rights for *machPort*, the new NSPort may only be able to send messages. If an NSPort with *machPort* already exists, deallocates the receiver, then retains and returns the existing NSPort.

This method is the designated initializer for the NSPort class. Returns **self**.

invalidate

– (void)**invalidate**

Marks the NSPort as invalid and posts an NSPortDidBecomeInvalidNotification to the default notification center.

See also: – **isValid**

isValid

– (BOOL)**isValid**

Returns NO if the NSPort is known to be invalid, YES otherwise (an NSPort only notes that it has become invalid when it tries to send or receive a message). An NSPort becomes invalid when its underlying communication resource, which is operating-system dependent, is closed or damaged.

See also: – **invalidate**

machPort

– (int)**machPort**

Returns as an **int** the Mach port used by the NSPort. Cast this value to a **port_t** when using it with Mach system calls.

removeConnection:fromRunLoop:forMode:

– (void)**removeConnection:**(NSConnection *)*connection*
fromRunLoop:(NSRunLoop *)*runLoop*
forMode:(NSString *)*mode*

Removes the receiver from the list of ports monitored by *runLoop* in the given input *mode*. You should not call this method directly. The method is provided for subclassers who wish to provide their own custom types of NSPort. NSConnection calls this method at the appropriate times. See the TCP Transport example in **/System/Developer/Examples/Foundation** for an example implementation.

reservedSpaceLength

– (unsigned int)**reservedSpaceLength**

Returns the amount of space reserved by the port for sending data.

sendBeforeDate:components:from:reserved:

(BOOL)**sendBeforeDate:**(NSDate *)*limitDate*
components:(NSMutableArray *)*components*
from:(NSPort *)*receivePort*
reserved:(unsigned int)*headerSpaceReserved*

You should not call this method directly. The method is provided for subclassers who wish to provide their own custom types of NSPort. NSConnection calls this method at the appropriate times. See the TCP Transport example in **/System/Developer/Examples/Foundation** for an example implementation.

setDelegate:

– (void)**setDelegate:**(id)*anObject*

Sets the NSPort's delegate to *anObject*. Doesn't retain *anObject*.

See also: – **delegate**

Methods Implemented By the Delegate

handleMachMessage:

– (void)**handleMachMessage:**(void *)*machMessage*

Processes *machMessage*, an incoming Mach message cast as a pointer to **void**. The delegate should interpret this data as a pointer to a Mach message beginning with a **msg_header_t** structure and should handle the message appropriately.

The delegate should implement only one of **handleMachMessage:** and **handlePortMessage:**.

handlePortMessage:

– (void)**handlePortMessage:**(NSPortMessage *)*portMessage*

Processes *portMessage*, an incoming message on the NSPort. See the NSPortMessage class specification for more information.

The delegate should implement only one of **handleMachMessage:** and **handlePortMessage:**.

limitDateForMode:

– (NSDate *)limitDateForMode:(NSString *)mode

Returns the port's earliest limit date for the given mode. This limit date represents the timeout value for the port.

See also: **– limitDateForMode:** (NSRunLoop)

Notifications

NSPortDidBecomeInvalidNotification

Posted from the **invalidate** method, which is invoked when the NSPort is deallocated or when it notices that its communication channel has been damaged. This notification contains a notification object but no userInfo dictionary. The notification object is the NSPort object that has become invalid.

The NSPort object posting this notification is no longer useful, so all receivers should unregister themselves for any notifications involving the NSPort. You should ensure that any method that receives this notification checks to ee which port became invalid before attempting to do anything. In particular, observers that receive all NSPortDidBecomeInvalidNotification's should be aware that communication with the window server is handled through an NSPort. If this port becomes invalid,drawing operations will cause a fatal error.

NSPortCoder

Inherits From: NSCoder : NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSPortCoder.h

Class Description

NSPortCoder is a concrete subclass of NSCoder used in the distributed objects system to transmit object proxies (and sometimes objects themselves) between NSConnection objects. An NSPortCoder is always created and used by an NSConnection object; your code should never need to explicitly create or use one.

Making Substitutions During Encoding

Like its abstract superclass, NSCoder, NSPortCoder makes use of substitution methods that allow an object to encode itself as an instance of another class or to replace another object for itself. An object may need to offer a different replacement when being encoded specifically by an NSPortCoder, however, so instead of the generic **classForCoder:** and **replacementObjectForCoder:** methods, NSPortCoder invokes **classForPortCoder:** and **replacementObjectForPortCoder:**. Their default implementations in NSObject fall back to the generic methods, providing reasonable default behavior. (NSPortCoder doesn't use a special substitution method for decoding; it simply uses **awakeAfterUsingCoder:** as NSCoder does.)

The generic **classForCoder:** method is most useful for mapping private subclass hierarchies through a public superclass, which (for example) aids the stability of archives when subclasses are private or subject to change. Since processes communicating at run time typically use the same version of a class library, this mapping is often not needed in distributed objects communication. **classForPortCoder:** allows an object to bypass or override the generic **classForCoder:** behavior, sending its real class (or simply a different one from the generic case) to the communicating process or thread. If you implement a group of classes that use the generic **classForCoder:** method, you should also consider using **classForPortCoder:** to handle the special case of encoding with the distributed objects system.

The generic **replacementObjectForCoder:** method offers a standard way to substitute a different instance at encoding time. **replacementObjectForPortCoder:** specifically allows for the substitution of proxies over a distributed objects connection. The receiver of a **replacementObjectForPortCoder:** message can ask the NSPortCoder whether it should be encoded **bycopy** or not, and return itself or a proxy as appropriate. NSObject's implementation always returns a proxy, so subclasses that allow **bycopy** encoding should override **replacementObjectForPortCoder:** to perform at least as this sample does:

```
- (id)replacementObjectForPortCoder:(NSPortCoder *)encoder
{
    if ([encoder isBycopy]) return self;
    return [super replacementObjectForPortCoder:encoder];
}
```

If the NSPortCoder returns YES when sent an **isBycopy** message, this example method returns **self**, which will result in the receiver being sent an **encodeWithCoder:** message. If the NSPortCoder returns NO, this method invokes the superclass's implementation, which typically returns an NSDistantObject.

Method Types

Creating an NSPortCoder

+ portCoderWithReceivePort:sendPort:components:

Getting the NSConnection

– connection

Encoding NSPort objects

– encodePortObject:
– decodePortObject

Checking for encoding

– isBycopy
– isByref

Dispatching

– dispatch

Class Methods

portCoderWithReceivePort:sendPort:components:

+ **portCoderWithReceivePort:**(NSPort *)*receivePort*
sendPort:(NSPort *)*sendPort*
components:(NSArray *)*components*;

<< Description forthcoming. >>

Instance Methods

connection

– (NSConnection *)**connection**

Returns the NSConnection object that uses the NSPortCoder. In an object's **encodeWithCoder:** method, this is the sending (server) NSConnection. In **initWithCoder:** this is the receiving (client) NSConnection.

decodePortObject

– (NSPort *)**decodePortObject**

Decodes and returns an NSPort object that was previously encoded with any of the general **encode...Object:** messages. This method is primarily for use by NSPort objects themselves; you can always use **decodeObject** to decode any object.

NSPort invokes this method in its **initWithCoder:** method so that the appropriate kernel information for the port can be decoded. A subclass of NSPortCoder shouldn't decode an NSPort by sending it an **initWithCoder:** message. See ""Creating a Subclass of NSCoder"" in the NSCoder class specification for more informaton.

dispatch

– (void)**dispatch**

<< Description forthcoming. >>

encodePortObject:

– (void)**encodePortObject:**(NSPort *)*aPort*

Encodes *aPort* so that it can be properly reconstituted in the receiving process or thread. This method is primarily for use by NSPort objects themselves; you can always use the general **encode...Object:** methods to encode any object.

NSPort invokes this method in its **encodeWithCoder:** method so that the appropriate kernel information for the port can be encoded. A subclass of NSPortCoder shouldn't encode an NSPort by sending it an **encodeWithCoder:** message. See ""Creating a Subclass of NSCoder"" in the NSCoder class specification for more informaton.

isBycopy

– (BOOL)**isBycopy**

Returns YES if the NSPortCoder is encoding an object **bycopy**, NO if it expects a proxy. You typically use this method in **replacementObjectForPortCoder:** to decide whether to substitute a proxy. See ““Making Substitutions During Encoding”” in the class description for more information.

isByref

– (BOOL)**isByref**

<< Description forthcoming. >>

NSPortMessage

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSPortMessage.h

Class Description

An NSPortMessage defines a low level, operating-system independent type for interapplication (and interthread) messages. NSPortMessages are used primarily by the distributed objects system. You should implement interapplication communication using distributed objects whenever possible, and use NSPortMessages only when necessary.

An NSPortMessage has three major parts: the *send* and *receive ports*, which are NSPorts that link the sender of the message to the receiver, and the *components*, which form the body of the message. The components are held as an NSArray of NSData and NSPort objects. NSPortMessage's **sendBeforeDate:** message sends the components out through the send port; any replies to the message arrive on the receive port. See the NSPort class specification for information on handling incoming messages.

An NSPortMessage can be initialized with a Mach message or with a pair of NSPorts and an NSArray of components. In either case, an NSPortMessage's body can contain only the types in the table below. The types listed under **initWithMachMessage:** are converted to the corresponding object type listed under **initWithSendPort:receivePort:components:**. In the distributed objects system the byte/character arrays are usually encoded NSInvocations that are being forwarded from a proxy to the corresponding real object.

initWithMachMessage:	initWithSendPort:receivePort:components:
port_t	NSPort object
MSG_TYPE_BYTE	NSData object
MSG_TYPE_CHAR	
MSG_TYPE_INTEGER_32	

An NSPortMessage also maintains a message identifier, which can be used to indicate the class of a message, such as an Objective-C method invocation, a connection request, an error, and so on. Use the **setMsgid:** and **msgid** methods to access the identifier.

Method Types

Creating instances

– initWithMachMessage:
– initWithSendPort:receivePort:components:

Sending the message

– sendBeforeDate:

Getting the components

– components

Getting the ports

– receivePort
– sendPort

Accessing the message ID

– setMsgid:
– msgid

Instance Methods

components

– (NSArray *)**components**

Returns the data components of the NSPortMessage. See the class description for more information.

initWithMachMessage:

– (id)**initWithMachMessage:**(void *)*buffer*

Initializes a newly allocated NSPortMessage with *buffer*, which is a pointer to a Mach message (beginning with a **msg_header_t** structure) cast as a pointer to **void**. The send and receive ports, message identifier, and components are all extracted from *buffer*. Returns **self**.

See the class description for restrictions on the allowable types in the body of the Mach message.

initWithSendPort:receivePort:components:

– (id)**initWithSendPort:**(NSPort *)*sendPort*
receivePort:(NSPort *)*receivePort*
components:(NSArray *)*components*

Initializes a newly allocated NSPortMessage to send the data in *components* on *sendPort*. Replies to the message will arrive on *receivePort*. An NSPortMessage initialized with this method has a message identifier of 0. This is the designated initializer for the NSPortMessage Class. Returns **self**.

components should contain only NSData and NSPort objects, and the contents of the NSData objects should be in network byte order.

See also: – **setMsgid:**

msgid

– (unsigned int)**msgid**

Returns the identifier for the NSPortMessage. Cooperating applications can use this to define different types of messages, such as connection requests, RPCs, errors, and so on.

See also: – **setMsgid:**

receivePort

– (NSPort *)**receivePort**

For an outgoing message, returns the NSPort that replies to the NSPortMessage will arrive on. For an incoming message, returns the NSPort that the NSPortMessage did arrive on.

See also: – **sendPort**

sendBeforeDate:

– (BOOL)**sendBeforeDate:**(NSDate *)*aDate*

Attempts to send the message before *aDate*, returning YES if successful, NO otherwise. See the NSPort class specification for information on receiving an NSPortMessage.

sendPort

– (NSPort *)**sendPort**

For an outgoing message, returns the NSPort that the NSPortMessage will send itself through when it receives a **sendBeforeDate:** message. For an incoming message, returns the NSPort that replies to the NSPortMessage should be sent through.

See also: – **receivePort**

setMsgid:

– (void)**setMsgid:**(unsigned int)*msgid*

Sets the identifier for the NSPortMessage to *msgid*. Cooperating applications can use this to define different types of messages, such as connection requests, RPCs, errors, and so on.

See also: – **msgid**

NSPortNameServer

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	Foundation/NSPortNameServer.h

Class Description

NSPortNameServer provides an object-oriented interface to the port registration service used by the distributed objects system. NSConnection objects use it to contact each other and to distribute objects over the network; you should rarely need to interact directly with an NSPortNameServer.

You get an NSPortNameServer by using the **defaultPortNameServer** class method; never allocate and initialize an instance directly. With the default server object you can register an NSPort under a given name, making it available on the network, and also unregister it so that it can't be looked up (although other applications that have already looked up the NSPort can still use it until it becomes invalid). See the NSPort class specification for more information.

Method Types

Getting the server object
+ defaultPortNameServer

Looking ports up
– portForName:
– portForName:onHost:

Registering ports
– registerPort:forName:
– removePortForName:

Class Methods

defaultPortNameServer

+ (id)**defaultPortNameServer**

Returns the single instance of NSPortNameServer for the application.

Instance Methods

portForName:

– (NSPort *)**portForName:**(NSString *)*portName*

Looks up and returns the NSPort registered under *portName* on the local host (by invoking **portForName: onHost:** with **nil** as the host name). Returns **nil** if no such NSPort exists.

See also: – **portForName:onHost:**

portForName:onHost:

– (NSPort *)**portForName:**(NSString *)*portName* **onHost:**(NSString *)*hostName*

Looks up and returns the NSPort registered under *portName* on the host named *hostName*, which is an Internet domain name (for example, “sales.anycorp.com”). Returns **nil** if no such NSPort exists.

If *hostName* is **nil** or empty then only the local host is checked for *portName*. If *hostName* is “*” then all hosts on the local subnet are queried for the requested NSPort; where there are duplicates an arbitrary host’s NSPort is returned. That host is then used for every subsequent request by the local host.

registerPort:forName:

– (BOOL)**registerPort:**(NSPort *)*aPort* **forName:**(NSString *)*portName*

Makes *aPort* available on the network under *portName* and the local host’s name (or names). Returns YES if successful, NO otherwise (for example, if another NSPort has already been registered under *portName*). An NSPort can be registered under multiple names; if it is, it must be unregistered for each name with **removePortForName:** to make it completely unavailable.

removePortForName:

– (void)**removePortForName:**(NSString *)*portName*

Unregisters the NSPort for the name *portName* on the local host, so that it can no longer be looked up by that name. Other applications that already have the NSPort can continue to use it until it becomes invalid.

NSProcessInfo

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSProcessinfo.h

Class Description

The NSProcessInfo class provides methods to access process-wide information. An NSProcessInfo object can return such information as the arguments, environment variables, host name, or process name. The **processInfo** class method returns a shared NSProcessInfo object for the process. For example, the following line returns the NSProcessInfo object, which then provides the name of the current process:

```
NSString *processName = [[NSProcessInfo processInfo] processName];
```

NSProcessInfo also includes the **operatingSystem** method, which returns an **enum** constant identifying the operating system on which the process is executing.

Method Types

Getting an NSProcessInfo object

+ processInfo

Returning process information

– arguments
– environment
– hostName
– processName
– globallyUniqueString

Returning the host operating system

– operatingSystem
– operatingSystemName

Specifying a process name

– setProcessName:

Class Methods

processInfo

+ (NSProcessInfo *)**processInfo**

Returns an initialized NSProcessInfo object for the process. An NSProcessInfo object is created the first time this method is invoked, and that same object is returned on each subsequent invocation.

Instance Methods

arguments

– (NSArray *)**arguments**

Returns the command line arguments as an array of NSStrings.

environment

– (NSDictionary *)**environment**

Returns a dictionary of variables for the environment from which the process was launched. The dictionary keys are the environment variable names.

globallyUniqueString

– (NSString *)**globallyUniqueString**

Returns a globally unique string to identify the process. This method uses the host name, process ID, and a time stamp to ensure that the string returned will be unique for the network. This method generates a new string each time it is invoked, so it also uses a counter to guarantee that strings created from the same process will be unique.

See also: – **processName**

hostName

– (NSString *)**hostName**

Returns the name of the host system.

operatingSystem

– (unsigned int)**operatingSystem**

Returns one the contants below to indicate the operating system on which the process is executing:

NSWindowsNTOperatingSystem
NSWindows95OperatingSystem
NSSolarisOperatingSystem
NSHPUXOperatingSystem
NSMACHOperatingSystem
NSSunOSOperatingSystem
NSOSF1OperatingSystem

operatingSystemName

– (NSString *)**operatingSystemName**

<< Description forthcoming. >>

processName

– (NSString *)**processName**

Returns the name of the process. This name is used to register Application defaults and is used in error messages. It does not uniquely identify the process.

See also: – **setProcessName:**

setProcessName:

– (void)**setProcessName:**(NSString *)*newName*

Sets the name of the process to *newName*.

Warning: User defaults and other aspects of the environment might depend on the process name, so be very careful if you change it. Setting the process name in this manner is **not** thread-safe.

See also: – **processName**

NSProtocolChecker

Inherits From: NSObject

Declared In: Foundation/NSProtocolChecker.h

Class Description

The NSProtocolChecker class defines an object that restricts the messages that can be sent to another object (referred to as the checker's delegate). This can be particularly useful when an object with many methods, only a few of which ought to be remotely accessible, is made available using the Distributed Objects system.

A protocol checker acts as a kind of proxy; when it receives a message that is in its designated protocool, it forwards the message to its target, and consequently appears to be the target object itself. However, when it receives a message not in its protocol, it raises an NSInvalidArgumentException exception to indicate that the message isn't allowed, whether or not the target object implements the method.

Typically, an object that is to be distributed (yet must restrict messages) creates an NSProtocolChecker for itself and returns the checker rather than returning itself in response to any messages. The object might also register the checker as the root object of an NSConnection.

The object should be careful about vending references to **self**; the protocol checker will convert a return value of **self** to indicate the checker rather than the object for any messages that were forwarded by the checker, but direct references to the object (bypassing the checker) could be passed around by other objects.

Method Types

Creating a checker

+ protocolCheckerWithTarget:protocol:
– initWithTarget:protocol:

Reimplemented NSObject methods

– forwardInvocation:
– methodDescriptionForSelector:

Getting information

– protocol
– target

Class Methods

protocolCheckerWithTarget:protocol:

+ (id)**protocolCheckerWithTarget:**(NSObject *)*anObject* **protocol:**(Protocol *)*aProtocol*

Allocates and initializes an NSProtocolChecker instance that will forward any messages in the *aProtocol* protocol to *anObject*, its target. Thus, the checker can be vended in lieu of *anObject* to restrict the messages that can be sent to *anObject*. Returns the new instance.

Instance Methods

forwardInvocation:

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

Forwards any message to the delegate if the method is declared in the checker's protocol; otherwise raises an NSInvalidArgumentException.

initWithTarget:protocol:

– (id)**initWithTarget:**(NSObject *)*anObject* **protocol:**(Protocol *)*aProtocol*

Initializes a newly allocated NSProtocolChecker instance that will forward any messages in the *aProtocol* protocol to *anObject*, its delegate. Thus, the checker can be vended in lieu of *anObject* to restrict the messages that can be sent to *anObject*. If *anObject* is allowed to be freed or dereferenced by clients, the **free** method should be included in *aProtocol*. Returns the new instance.

methodDescriptionForSelector:

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

Returns an Objective C description for a method in the checker's protocol, or NULL if *aSelector* isn't declared as an instance method in the protocol.

protocol

– (Protocol *)protocol

Returns the protocol object the checker uses to verify whether a given message should be forwarded to its delegate, or the protocol checker should raise an NSInvalidArgumentException.

target

– (NSObject *)target

Returns the target of the NSProtocolChecker.

NSProxy

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

Conforms To: NSObject

Declared In: Foundation/NSProxy.h

Class Description

NSProxy is an abstract superclass defining an API for objects that act as stand-ins for other objects or for objects that don't exist yet. Typically, a message to a proxy is forwarded to the real object, or causes the proxy to load (or transform itself into) the real object. Subclasses of NSProxy can be used to implement transparent distributed messaging (for example, NSDistantObject) or for lazy instantiation of objects that are expensive to create.

NSProxy implements the basic methods required of a root class, including those defined in the NSObject protocol. However, as an abstract class it doesn't provide an initialization method, and it raises an exception upon receiving any message it doesn't respond to. A concrete subclass must therefore provide an initialization or creation method and override the **forwardInvocation:** and **methodSignatureForSelector:** methods to handle messages that it doesn't implement itself. A subclass's implementation of **forwardInvocation:** should do whatever is needed to process the invocation, such as forwarding the invocation over the network or loading the real object and passing it the invocation. **methodSignatureForSelector:** is required to provide argument type information for a given message; a subclass's implementation should be able to determine the argument types for the messages it needs to forward and should construct an NSMethodSignature accordingly. See the NSDistantObject, NSInvocation, and NSMethodSignature class specifications for more information.

Adopted Protocols

NSObject

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

Method Types

Creating instances

+ alloc
+ allocWithZone:

Deallocating instances

– dealloc

Getting the class

+ class

Handling unimplemented methods

– forwardInvocation:
– methodSignatureForSelector:

Getting a description

– description

Class Methods

alloc

+ (id)**alloc**

Returns a new instance of the receiving class, as described in the NSObject class specification under the **alloc** class method.

allocWithZone:

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

Returns a new instance of the receiving class, as described in the NSObject class specification under the **allocWithZone:** class method.

class

+ (Class)**class**

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

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

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.

Warning: 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.

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

respondsToSelector:

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

Returns YES if the receiving class responds to *aSelector* messages, NO otherwise.

Instance Methods

class

@protocol NSObject
– (Class)**class**

Returns the class of the receiver (not the class of the real object).

conformsToProtocol:

@protocol NSObject
– (BOOL)**conformsToProtocol:**(Protocol)*aProtocol*

Uses **forwardInvocation:** to send the **conformsToProtocol:** message to the real object and returns the result. Note that NSProxy's implementation of **forwardInvocation:** merely raises an exception.

dealloc

– (void)**dealloc**

Deallocates the memory occupied by the receiver, as described in the NSObject class specification under the **dealloc** instance method.

description

– (NSString *)**description**

Returns an NSString containing the real class name and the **id** of the receiver as a hexadecimal number.

forwardInvocation:

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

Passes *anInvocation* on to the real object that the proxy represents. NSProxy's implementation merely raises NSInvalidArgumentException. Override this method in your subclass to handle *anInvocation* appropriately, at the very least by setting its return value.

For example, if your proxy merely forwards messages to an instance variable named **realObject**, it can implement **forwardInvocation:** like this:

```
– (void)forwardInvocation:(NSInvocation *)anInvocation
{
    [anInvocation setTarget:realObject];
    [anInvocation invoke];
    return;
}
```

isKindOfClass:

@protocol NSObject
– (BOOL)**isKindOfClass:**(Class)*aClass*

Uses **forwardInvocation:** to send the **isKindOfClass:** message to the real object and returns the result. Note that NSProxy's implementation of **forwardInvocation:** merely raises an exception.

isMemberOfClass:

@protocol NSObject
– (BOOL)**isMemberOfClass:**(Class)*aClass*

Uses **forwardInvocation:** to send the **isMemberOfClass:** message to the real object and returns the result. Note that NSProxy's implementation of **forwardInvocation:** merely raises an exception.

isProxy

@protocol NSObject
– (BOOL)**isProxy**

Returns YES. Subclasses shouldn't override this method to return NO.

methodSignatureForSelector:

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

Raises NSInvalidArgumentException. Override this method in your concrete subclass to return a proper NSMethodSignature for *aSelector* and the class that your proxy objects stand in for. Be sure to avoid an infinite loop when necessary by checking that *aSelector* isn't the selector for this method itself and by not sending any message that might invoke this method.

For example, if your proxy merely forwards messages to an instance variable named **realObject**, it can implement **methodSignatureForSelector:** like this:

```
- (NSMethodSignature *)methodSignatureForSelector:(SEL)aSelector
{
    return [realObject methodSignatureForSelector:aSelector];
}
```

See also: – **methodSignatureForSelector:** (NSObject)

respondsToSelector:

@protocol NSObject
– (BOOL)**respondsToSelector:**(SEL)*aSelector*

Uses **forwardInvocation:** to send the **respondsToSelector:** message to the real object and returns the result. Note that NSProxy's implementation of **forwardInvocation:** merely raises an exception.

NSRecursiveLock

Inherits From: NSObject

Conforms To: NSLocking
NSObject (NSObject)

Declared In: Foundation/NSLock.h

Class Description

NSRecursiveLock defines a lock that may be acquired multiple times by the same thread without causing a deadlock, a situation where a thread is permanently blocked waiting for itself to relinquish a lock. While the locking thread has one or more locks, all other threads are prevented from accessing the code protected by the lock. Here's an example where a recursive lock functions properly but other lock types would deadlock:

```
NSRecursiveLock  *theLock = [[NSRecursiveLock alloc] init];
...
[theLock lock];
/* lengthy operations involving global data */
[theLock lock];   /* possibly invoked in a subroutine */
...
[theLock unlock]; /* relinquishes most recent lock */
...
[theLock unlock]; /* relinquishes the first lock */
```

Unless **theLock** was an NSRecursiveLock, a deadlock condition would occur at the second **lock** message in the example above.

The NSConditionLock, NSLock, and NSRecursiveLock classes all implement the NSLocking protocol with various features and performance characteristics; see the other class descriptions for more information.

Adopted Protocols

NSLocking

– lock
– unlock

Method Types

Acquiring a lock

– lockBeforeDate:
– tryLock

Instance Methods

lockBeforeDate:

– (BOOL)**lockBeforeDate:**(NSDate *)*limit*

Attempts to acquire a lock before the date represented by *limit*. Returns YES if the lock is acquired within this time limit. Returns NO if the time limit expires before a lock can be acquired.

tryLock

– (BOOL)**tryLock**

Attempts to acquire a lock. Returns immediately with a value of YES if successful and NO otherwise.

NSRunLoop

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSRunLoop.h

Class Description

The NSRunLoop class declares the programmatic interface to objects that manage input sources. An NSRunLoop processes input for sources such as mouse and keyboard events from the window system, NSPorts, NSTimers, and NSConnections.

In general, your application won't need to either create or explicitly manage NSRunLoop objects. Each thread has an NSRunLoop object automatically created for it. Each process begins with a default thread and therefore has a default run loop.

If you do want to perform your own explicit run loop management, you do so by accessing the current thread's run loop (returned by the class method **currentRunLoop**). You must specify two things: the *input sources*, which are the objects from which the run loop will receive information, and an *input mode*, which specifies the type of input to be received. The currently defined input modes are:

Input mode	Description
NSDefaultRunLoopMode	Use this mode to deal with input sources other than NSConnections. Defined in the **Foundation/NSRunLoop.h** header file. This is the most commonly used run loop mode.
NSConnectionReplyMode	Use this mode to indicate NSConnection objects waiting for replies. Defined in the **Foundation/NSConnection.h** header file. You rarely need to use this mode.

In addition, the Application Kit defines these modes:

Additional input modes	Description
NSModalPanelRunLoopMode	Use this mode when waiting for input from a modal panel, such as NSSavePanel or NSOpenPanel. Defined in the **AppKit/NSApplication.h** header file.

Additional input modes	Description
NSEventTrackingRunLoopMode	Use this mode for event tracking loops. Defined in the **AppKit/NSApplication.h** header file.

You associate a list of input sources with each input mode. There are two general types of input sources to a run loop: asynchronous (input arrives at unpredictable intervals) and synchronous (input arrives at regular intervals). NSPort objects represent asynchronous input sources, and NSTimer objects represent synchronous input sources. Each input source has a limit date associated with it. For NSPorts, the limit date is a timeout value, after which input from that port is no longer timely. For NSTimers, the limit date specifies when the timer should fire. (When a timer fires, it sends a specified message to a specified object, and it may be scheduled to fire again later. See the NSTimer class specification for more information.)

When an NSRunLoop runs, it polls each of the sources for the input mode to determine which one has the earliest limit date. During this polling, the input sources may process any input they have queued. Once the NSRunLoop determines the earliest limit date for this input mode, it waits for input from the operating system until that limit date. If input arrives, it is processed. At that point, the NSRunLoop may either return or it may continue, depending on which method was used to run the loop.

For example:

```
NSRunLoop *theLoop = [NSRunLoop currentRunLoop];

[theLoop acceptInputForMode:NSDefaultRunLoopMode beforeDate:[theLoop
    limitDateForMode:NSDefaultRunLoopMode]];
```

The method **limitDateForMode:** returns the earliest limit date of all the input sources for the mode NSDefaultRunLoopMode. **acceptInputForMode:beforeDate:** runs the loop until that date, processing any input it receives until that time. As a convenience, you can use **runMode:beforeDate:** instead. It invokes **acceptInputForMode:beforeDate:** and **limitDateForMode:** with the mode you supply.

To continuously run in NSDefaultRunLoopMode, you can use either of the methods **run** or **runUntilDate:** . To run another mode continuously, invoke **runMode:beforeDate:** in a loop with a date far in the future:

```
while ([[NSRunLoop currentRunLoop] runMode:NSModalPanelRunLoopMode
    beforeDate:[NSDate distantFuture]])
    ;
```

Note: Regardless of the date you specify in **runMode:beforeDate:**, a run loop with nothing to do (that is no sources from which to receive input) will exit immediately. You must add the input sources to the run loop mode before you start the run loop.

Method Types

Accessing the current run loop

+ currentRunLoop
– currentMode
– limitDateForMode:

Adding timers

– addTimer:forMode:

Managing ports

– addPort: forMode:
– removePort:forMode:

Setting up for server processes

– configureAsServer

Running a loop

– run
– runUntilDate:
– runMode:beforeDate:
– acceptInputForMode:beforeDate:

Sending messages

– performSelector:target:argument:order:modes:
– cancelPerformSelector:target:argument:

Class Methods

currentRunLoop

+ (NSRunLoop *)**currentRunLoop**

Returns the NSRunLoop for the current thread.

See also: **– currentMode**

Instance Methods

acceptInputForMode:beforeDate:

– (void)**acceptInputForMode:**(NSString *)*mode* **beforeDate:**(NSDate *)*limitDate*

Blocks awaiting input from the ports in the port list for the input mode *mode* until the time specified by *limitDate*. Use the **limitDateForMode:** method to calculate *limitDate*. If input arrives, it is processed using the NSPort delegates. This method does not check the timers associated with *mode*, thus it does not fire timers even if their scheduled fire dates have passed.

addPort: forMode:

– (void)**addPort:**(NSPort *)*aPort* **forMode:**(NSString *)*mode*

Adds *aPort* to be monitored by the receiver in the input mode *mode*. The receiver maintains a count of the number of ports added, and the same number must be removed.

See also: **– removePort:forMode:**

addTimer:forMode:

– (void)**addTimer:**(NSTimer *)*aTimer*
forMode:(NSString *)*mode*

Registers the timer *aTimer* with input mode *mode*. The run loop causes the timer to fire on or after its scheduled fire date. Timers have an Objective-C message associated with them. When a timer fires, it sends its message to the appropriate object. To remove a timer from a mode, send the **invalidate** message to the timer.

cancelPerformSelector:target:argument:

– (void)**cancelPerformSelector:**(SEL)*aSelector* **target:**(id)*target* **argument:**(id)*anArgument*

Cancels the sending of a message previously scheduled using **performSelector:target:argument:order:modes:**. The *aSelector* message with argument *anArgument* will not be sent to *target*.

configureAsServer

- (void)**configureAsServer**

Performs all necessary configuration to make the run loop suitable for use by a server process. For example, on the Microsoft Windows platform, this method adds a port that receives messages from the WIN32 event queue. This enables the run loop to notice when the user logs out, allowing the server process to exit gracefully.

currentMode

– (NSString *)**currentMode**

Returns the current input mode. The current mode is set by **limitDateForMode:** and **acceptInputForMode:beforeDate:**.

See also: **+ currentRunLoop**

limitDateForMode:

– (NSDate *)**limitDateForMode:**(NSString *)*mode*

Polls *mode*'s input sources for their limit date (if any) and returns the earliest limit date for this mode. Uses the NSPort delegate method **limitDateForMode:** to determine the limit dates of ports. Fires timers if their limit dates have passed. Polls ports for activities appropriate for *mode*. Returns **nil** if there are no input sources for this mode.

performSelector:target:argument:order:modes:

– (void)**performSelector:**(SEL)*aSelector* **target:**(id)*target* **argument:**(id)*anArgument* **order:** (unsigned int)*order* **modes:**(NSArray *)*modes*

Schedules the sending of an *aSelector* message. The *aSelector* message will be sent to *target* with argument *anArgument* after the run loop has completed an iteration in any of the input modes specified in *modes*. *order* assigns a priority to the messages. If multiple messages are scheduled to be sent, the messages with a lower *order* value are sent before messages with a higher *order* value.

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**. The NSRunLoop does not retain the *target* and *anArgument* objects.

Use this method is you want multiple messages to be sent after the current event has been processed and you want to make sure that these messages are sent in a certain order.

See also: – **cancelPerformSelector:target:argument:**

removePort:forMode:

– (void)**removePort:**(NSPort *)*aPort* **forMode:**(NSString *)*mode*

Removes *aPort* from the list of ports being monitored by the receiver in input mode *mode*. The receiver maintains a count of the ports added, and the same number of ports must be removed. Ports are automatically removed from modes if they are detected to be invalid.

See also: **– addPort:forMode:**

run

– (void)**run**

Runs the loop in NSDefaultRunLoopMode by repeatedly invoking **runMode:beforeDate:** until the limit dates for all of the input sources have passed.

See also: – **runUntilDate:**

runMode:beforeDate:

– (BOOL)**runMode:**(NSString *)*mode*
beforeDate:(NSDate *)*limitDate*

Runs the loop once by invoking **acceptInputForMode:beforeDate:**, accepting input for mode *mode* until a limit date. The limit date is determined by using the earliest of *limitDate* and the limit dates set for all input sources in this mode. Returns NO without starting the run loop if the limit dates for all of *mode*'s input sources have passed; otherwise returns YES.

runUntilDate:

– (void)**runUntilDate:**(NSDate *)*limitDate*

Runs the loop in NSDefaultRunLoopMode by repeatedly invoking **runMode:beforeDate:** until *limitDate* or until the limit dates for all of the input sources have passed.

See also: – **run**

NSScanner Class Cluster

Class Cluster Description

An NSScanner object interprets and converts the characters of an NSString into number and string values. You assign the scanner's string on creating it, and the scanner progresses through the characters of that string from beginning to end as you request items. This cluster has a single public class, NSScanner.

The objects you create using this class are referred to as *scanner objects* (and when no confusion will result, merely as *scanners*). Because of the nature of class clusters, scanner objects aren't actual instances of the NSScanner class but one of its private subclasses. Although a scanner object's class is private, its interface is public, as declared by this abstract superclass, NSScanner.

Scanning Behavior

Generally, you instantiate a scanner object by invoking the **scannerWithString:** or **localizedScannerWithString**: class methods. Either method returns a scanner object initialized with the string you pass to it. The newly created scanner starts at the beginning of its string, progressing through the characters as you request values with **scan...** methods. You can change the implicit scan location with the **setScanLocation:** method, to re-scan a portion of the string after an error or to skip ahead a certain number of characters. Scan operations start at the scan location and advance the scanner to just past the last character in the scanned value representation (if any). For example, after scanning an integer from the string "137 small cases of bananas", a scanner's location will be 3, indicating the space immediately after the number.

You can configure a scanner to skip a set of characters with the **setCharactersToBeSkipped:** method. A scanner ignores characters to be skipped at the beginning of any scan operation. Once it finds a scannable character, however, it includes all characters matching the request. Scanners skip whitespace and newline characters by default. If you continue with the previous example's string and use **scanUpToString: intoString:** to find the substring before "of", the scanner skips the space character before the word "small" but includes the space before "of" in its result unless you include a space in the search string:

Search String	Result String
"of" (no space before)	"small cases " (includes the space following)
" of" (space before)	"small cases" (stops before the space)

You can also configure a scanner to consider or ignore case using the **setCaseSensitive:** method. By default a scanner ignores case. Note that case is always considered with regard to characters to be skipped. To skip

all English vowels, for example, you must set the characters to be skipped to those in the string “AEIOUaeiou”.

A scanner bases some of its scanning behavior on a locale, which specifies a language and conventions for value representations. NSScanner uses only the locale’s definition for the decimal separator (given by the key named NSDecimalSeparator). You can create a scanner with the user’s locale by using **localizedScannerWithString:**, or set the locale explicitly using **setLocale:**. If you use a method that doesn’t specify a locale, the scanner assumes the default locale values. See “Locales” in the “Other Features” section of the Foundation Kit documentation for more information on locales.

For an example of using a scanner, suppose you have a string containing lines such as:

> Product: Acme Potato Peeler; Cost: 0.98
> Product: Chef Pierre Pasta Fork; Cost: 0.75
> Product: Chef Pierre Colander; Cost: 1.27

This method scans such a string to extract the product information for each line:

```
- (BOOL)scanProductString:(NSString *)string
{
    NSCharacterSet *semicolonSet;
    NSScanner *theScanner;
    NSString *PRODUCT = @"Product:";
    NSString *COST = @"Cost:";
    NSString *productName;
    float productCost;

    semicolonSet = [NSCharacterSet
           characterSetWithCharactersInString:@";"];
    theScanner = [NSScanner scannerWithString:string];

    while ([theScanner isAtEnd] == NO) {

        if ([theScanner scanString:PRODUCT intoString:NULL] &&
                [theScanner scanUpToCharactersFromSet:semicolonSet
                       ntoString:&productName] &&
                theScanner scanString:@";" intoString:NULL] &&
                [theScanner scanString:COST intoString:NULL] &&
                [theScanner scanFloat:&productCost]) {

            /* Do something with productName and productCost. */
        }
        else return NO;
    }

    return YES;
}
```

This method uses alternating scan operations to skip the expected substrings “Product:” and “Cost:”, as well as the semicolon, and to read the values for the product name and cost (read as a **float** for simplicity’s sake). It returns NO if an error occurs on any scan operation, and YES if it successfully scans and processes all lines. Note that because a scanner skips whitespace and newlines by default, the loop does no special processing for them.

NSScanner

Inherits From:	NSObject
Conforms To:	NSCopying NSObject (NSObject)
Declared In:	Foundation/NSScanner.h Foundation/NSDecimalNumber.h

Class Description

The NSScanner class declares the programmatic interface for an object that scans values from an NSString object. NSScanner's primitive methods are **string** and all of the methods listed below under "Configuring an NSScanner." Using an NSScanner is explained in the class cluster description.

Adopted Protocols

NSCopying	– copyWithZone:

Method Types

Creating an NSScanner

+ scannerWithString:
+ localizedScannerWithString:
– initWithString:

Getting an NSScanner's string

– string

Configuring an NSScanner

– setScanLocation:
– scanLocation
– setCaseSensitive:
– caseSensitive
– setCharactersToBeSkipped:
– charactersToBeSkipped
– setLocale:
– locale

Scanning a string

– scanCharactersFromSet:intoString:
– scanUpToCharactersFromSet:intoString:
– scanDecimal:
– scanDouble:
– scanFloat:
– scanInt:
– scanHexInt:
– scanLongLong:
– scanString:intoString:
– scanUpToString:intoString:
– isAtEnd

Class Methods

localizedScannerWithString:

+ (id)**localizedScannerWithString:**(NSString *)*aString*

Returns an NSScanner that scans *aString* according to the user's default locale (set with **setLocale:**). Sets the string to scan by invoking **initWithString:** with *aString*. See "Locales" in the "Other Features" section of the Foundation Kit documentation for more information on locales.

scannerWithString:

+ (id)**scannerWithString:**(NSString *)*aString*

Returns an NSScanner that scans *aString*. Sets the string to scan by invoking **initWithString:** with *aString*.

Instance Methods

caseSensitive

– (BOOL)**caseSensitive**

Returns YES if the scanner distinguishes case in the characters it scans, NO otherwise. NSScanners are *not* case sensitive by default. Note that case sensitivity doesn't apply to the characters to be skipped.

See also: – **setCaseSensitive:**, – **setCharactersToBeSkipped:**

charactersToBeSkipped

– (NSCharacterSet *)**charactersToBeSkipped**

Returns a character set containing the characters that the scanner ignores when looking for a scannable element. For example, if a scanner ignores spaces and you send it a **scanInt:** message, it skips spaces until it finds a decimal digit or other character. While an element is being scanned, however, no characters are skipped. If you scan for something made of characters in the set to be skipped (for example, using **scanInt:** when the set of characters to be skipped is the decimal digits), the result is undefined.

The default set to skip is the whitespace and newline character set.

See also: – **setCharactersToBeSkipped:**, + **whitespaceAndNewlineCharacterSet** (NSCharacterSet)

initWithString:

– (id)**initWithString:**(NSString *)*aString*

Initializes a newly allocated NSScanner to scan *aString* from the beginning. Returns **self**.

See also: + **localizedScannerWithString:**, + **scannerWithString:**

isAtEnd

– (BOOL)**isAtEnd**

Returns YES if the scanner has exhausted all significant characters in its string, NO if there are characters left to scan. If only characters from the set to be skipped remain, this method returns YES.

See also: – **charactersToBeSkipped**

locale

– (NSDictionary *)**locale**

Returns the scanner's locale, or **nil** if it has none. A scanner's locale affects the way it interprets numeric values from the string. In particular, a scanner uses the locale's decimal separator to distinguish the integer and fractional parts of floating-point representations. A scanner with no locale set uses the default locale values.

See "Locales" in the "Other Features" section of the Foundation Kit documentation for more information on locales.

See also: – **setLocale:**

scanCharactersFromSet:intoString:

– (BOOL)**scanCharactersFromSet:**(NSCharacterSet *)*scanSet*
intoString:(NSString **)*stringValue*

Scans the string as long as characters from *scanSet* are encountered, accumulating characters into a string that's returned by reference in *stringValue*. Returns YES if the scanner scans any characters; otherwise returns NO.

Invoke this method with NULL as *stringValue* to simply scan past a given set of characters.

See also: – **scanUpToCharactersFromSet:intoString:**

scanDecimal:

– (BOOL)**scanDecimal:**(NSDecimal *)*decimalValue*

Scans an NSDecimal value if possible, returning it by reference in *decimalValue*. Returns YES if the scanner finds a valid NSDecimal representation, NO otherwise. See the NSDecimalNumber class specification for more information.

Invoke this method with NULL as *decimalValue* to simply scan past an NSDecimal representation.

scanDouble:

– (BOOL)**scanDouble:**(double *)*doubleValue*

Scans a **double** value if possible, returning it by reference in *doubleValue*. Returns YES if the scanner finds a valid floating-point representation, NO otherwise. Returns HUGE_VAL or –HUGE_VAL by reference in *value* on overflow, or 0.0 on underflow. Skips past excess digits in the case of overflow, so that the scanner's position is past the entire floating-point representation.

Invoke this method with NULL as *doubleValue* to simply scan past a **double** value representation.

Note: Floating-point representations are assumed to be IEEE compliant.

See also: – **doubleValue** (NSString)

scanFloat:

– (BOOL)**scanFloat:**(float *)*floatValue*

Scans a **float** value if possible, returning it by reference in *floatValue*. Returns YES if the scanner finds a valid floating-point representation, NO otherwise. Returns HUGE_VAL or –HUGE_VAL by reference in *floatValue* on overflow, or 0.0 on underflow. Skips past excess digits in the case of overflow, so that the scanner's position is past the entire floating-point representation.

Invoke this method with NULL as *floatValue* to simply scan past a **float** value representation.

Note: Floating-point representations are assumed to be IEEE compliant.

See also: – **floatValue** (NSString)

scanHexInt:

– (BOOL)**scanHexInt:**(unsigned int *)*intValue*

Scans an **int** value from a hexadecimal representation if possible, returning it by reference in *intValue*. The hexadecimal integer representation may optionally be preceded by "0x" or "0X". Returns YES if the scanner finds a valid hexadecimal integer representation, NO otherwise. Returns INT_MAX or INT_MIN by reference in *intValue* on overflow. Skips past excess digits in the case of overflow, so that the scanner's position is past the entire hexadecimal representation.

Invoke this method with NULL as *intValue* to simply scan past a hexadecimal integer representation.

scanInt:

– (BOOL)**scanInt:**(int *)*intValue*

Scans an **int** value from a decimal representation if possible, returning it by reference in *intValue*. Returns YES if the scanner finds a valid decimal integer representation, NO otherwise. Returns INT_MAX or INT_MIN by reference in *intValue* on overflow. Skips past excess digits in the case of overflow, so that the scanner's position is past the entire decimal representation.

Invoke this method with NULL as *intValue* to simply scan past a decimal integer representation.

See also: – **intValue** (NSString)

scanLocation

– (unsigned int)**scanLocation**

Returns the character position at which the scanner begins its next scanning operation.

See also: – **setScanLocation:**

scanLongLong:

– (BOOL)**scanLongLong:**(long long *)*longLongValue*

Scans a **long long int** value from a decimal representation if possible, returning it by reference in *value*. Returns YES if the scanner finds a valid decimal integer representation, NO otherwise. Returns

LONG_LONG_MAX or LONG_LONG_MIN by reference in *longLongValue* on overflow. All overflow digits are skipped. Skips past excess digits in the case of overflow, so that the scanner's position is past the entire decimal representation.

Invoke this method with NULL as *longLongValue* to simply scan past a long decimal integer representation.

scanString:intoString:

– (BOOL)**scanString:**(NSString *)*string*
intoString:(NSString **)*stringValue*

Scans for *string*, and, if a match is found, returns an equivalent string object by reference in *stringValue*. Returns YES if *stringValue* matches the characters at the scan location; otherwise returns NO.

Invoke this method with NULL as *value* to simply scan past a given string.

See also: – **scanUpToString:intoString:**

scanUpToCharactersFromSet:intoString:

– (BOOL)**scanUpToCharactersFromSet:**(NSCharacterSet *)*stopSet*
intoString:(NSString **)*stringValue*

Scans the string until a character from *stopSet* is encountered, accumulating characters into a string that's returned by reference in *stringValue*. Returns YES if the scanner scans any characters; otherwise returns NO.

Invoke this method with NULL as *stringValue* to simply scan up to a given set of characters.

See also: – **scanCharactersFromSet:intoString:**

scanUpToString:intoString:

– (BOOL)**scanUpToString:**(NSString *)*stopString*
intoString:(NSString **)*stringValue*

Scans the string until *stopString* is encountered, accumulating characters into a string that's returned by reference in *stringValue*. Returns YES if the scanner scans any characters; otherwise returns NO.

Invoke this method with NULL as *stringValue* to simply scan up to a given string.

See also: – **scanString:intoString:**

setCaseSensitive:

– (void)**setCaseSensitive:**(BOOL)*flag*

If *flag* is YES, the scanner will distinguish case when scanning characters. If *flag* is NO, it will ignore case distinctions. NSScanners are by default *not* case sensitive. Note that case sensitivity doesn't apply to the characters to be skipped.

See also: – **caseSensitive**, – **setCharactersToBeSkipped:**

setCharactersToBeSkipped:

– (void)**setCharactersToBeSkipped:**(NSCharacterSet *)*skipSet*

Sets the scanner to ignore the characters in *skipSet* when scanning its string for a value representation. For example, if a scanner ignores spaces and you send it a **scanInt:** message, it skips spaces until it finds a decimal digit or other character. While an element is being scanned, however, no characters are skipped. If you scan for something made of characters in the set to be skipped (for example, using **scanInt:** when the set of characters to be skipped is the decimal digits), the result is undefined.

The characters to be skipped are treated literally as single values. A scanner doesn't apply its case sensitivity setting to these characters, and doesn't attempt to match composed character sequences with anything in the set of characters to be skipped (though it does match precomposed characters individually). If you want to skip all vowels while scanning a string, for example, you can set the characters to be skipped to those in the string "AEIOUaeiou" (plus any accented variants with precomposed characters).

The default set of characters to skip is the whitespace and newline character set.

See also: – **charactersToBeSkipped**, + **whitespaceAndNewlineCharacterSet** (NSCharacterSet)

setLocale:

– (void)**setLocale:**(NSDictionary *)*aLocale*

Sets the scanner's locale to *aLocale*. A scanner's locale affects the way it interprets values from the string. In particular, a scanner uses the locale's decimal separator to distinguish the integer and fractional parts of floating-point representations. A new scanner's locale is by default **nil**, which causes it to use the default locale values.

See "Locales" in the "Other Features" section of the Foundation Kit documentation for more information on locales.

See also: – **locale**

setScanLocation:

– (void)**setScanLocation:**(unsigned int)*index*

Sets the location at which the next scan operation begins to *index*. This method is useful for backing up to re-scan after an error. Raises an NSRangeException if *index* is beyond the end of the string being scanned.

Rather than setting the scan location directly to skip known sequences of characters, use **scanString: intoString:** or **scanCharactersFromSet:intoString:**, which allow you to verify that the expected substring (or set of characters) is in fact present.

See also: – **scanLocation**

string

– (NSString *)**string**

Returns the string that the scanner was created or initialized with.

See also: – **locale**

NSSerializer

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	Foundation/NSSerialization.h

Class Description

The NSSerializer class provides a mechanism for creating an abstract representation of a property list. (In OpenStep, property lists are defined to be—and to contain—objects of these classes: NSDictionary, NSArray, NSString, NSData). The NSSerializer class stores this representation in an NSData object in an architecture-independent format, so that property lists can be used with distributed applications. NSSerializer's companion class NSDeserializer declares methods that take the abstract representation and recreate the property list in memory.

In contrast to archiving (see the NSArchiver class specification), the serialization process preserves only structural information, not class information. Thus, if a property list is serialized and then deserialized, the objects in the resulting property list might not be of the same class as the objects in the original property list. However, the structure and interrelationships of the data in the resulting property list are identical to that in the original, with one possible exception.

The exception is that when an object graph is serialized, the mutability of the containers objects (NSDictionary and NSArray objects) is preserved only down to the highest node in the graph that has an immutable container. Thus, if an NSArray contains an NSMutableDictionary, the serialized version of this object graph would not preserve the mutability of the dictionary or any of the mutable objects it contained. Since serialization doesn't preserve class information or—in some cases—mutability, coding (as implemented by NSCoder and NSArchiver) is the preferred way to make object graphs persistent.

The NSSerializer class object provides the interface to the serialization process; you don't create instances of NSSerializer. You might subclass NSSerializer to modify the representation it creates, for example, to encrypt the data or add authentication information.

Other types of data besides property lists can be serialized using methods declared by the NSData and NSMutableData classes (see **serializeDataAt:ofObjCType:context:** and **deserializeDataAt:ofObjCType: atCursor:context:**), allowing these types to be represented in an architecture-independent format. Furthermore, the NSObjCTypeSerializationCallBack protocol allows you to serialize and deserialize objects that aren't property lists.

Class Methods

serializePropertyList:

+ (NSData *)**serializePropertyList:**(id)*aPropertyList*

Creates a data object, serializes *aPropertyList* into it, and returns the data object. *aPropertyList* must be a kind of NSData, NSString, NSArray, or NSDictionary.

serializePropertyList:intoData:

+ (void)**serializePropertyList:**(id)*aPropertyList* **intoData:**(NSMutableData *)*mdata*

Serializes the property list *aPropertyList* in the mutable data object *mdata* . *aPropertyList* must be a kind of NSData, NSString, NSArray, or NSDictionary.

NSSet Class Cluster

Class Cluster Description

The NSSet, NSMutableSet, and NSCountedSet classes declare the programmatic interface for objects that store unordered sets of objects.

Because of the nature of class clusters, the objects you create with the NSSet class cluster are not actual instances of NSSet or NSMutableSet. Rather, the instances belong to one of their private subclasses. (For convenience, we use the term *set* to refer to any one of these instances without specifying its exact class membership.) Although a set's class is private, its interface is public, as declared by the abstract superclasses NSSet and NSMutableSet.Note that NSCountedSet is not part of the class cluster; it is a concrete subclass of NSMutableSet.

NSSet declares the programmatic interface for static sets of objects. You establish a static set's entries when it's created, and thereafter the entries can't be modified. NSMutableSet, on the other hand, declares a programmatic interface for dynamic sets of objects. A dynamic—or mutable—set allows the addition and deletion of entries at any time, automatically allocating memory as needed.

Use sets as an alternative to arrays when the order of elements isn't important and performance in testing whether an object is contained in the set *is* a consideration—while arrays are ordered, testing for membership is slower than with sets.

Objects in a set must respond to the NSObject protocol methods **hash** and **isEqual:**. See the NSObject protocol for more information.

Note: If mutable objects are stored in a set, either the **hash** method of the objects shouldn't depend on the internal state of the mutable objects or the mutable objects shouldn't be modified while they're in the set (note that it can be difficult to know whether or not a given object is in a collection).

Objects added to a set are not copied; rather, each object receives a **retain** message before it's added to a set.

Generally, you create a temporary set by sending one of the **set…** methods to the NSSet class object. These methods return an NSSet object containing the elements (if any) you pass in as arguments. The **set** method is a "convenience" method to create an empty mutable set. Newly created instances of NSSet created by invoking the **allocWithZone:** method can be populated with objects using any of the **init…** methods.

The set classes adopt the NSCopying and NSMutableCopying protocols, making it convenient to convert a set of one type to the other.

NSSet

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSMutableCopying NSObject (NSObject)
Declared In:	Foundation/NSSet.h

Class at a Glance

Purpose

An NSSet object stores an immutable set of objects.

Principal Attributes

- The objects that make up the set.

Creation

+ set	Returns an empty set.
+ setWithArray:	Returns a set containing a number of objects from an array.
+ setWithObject:	Returns a set containing a single object.
+ setWithObjects:	Returns a set containing a number of objects.
+ setWithObjects:count:	Returns a set containing a specified number of objects.
+ setWithSet:	Returns a set containing a number of objects from another set.

Commonly Used Methods

– allObjects	Returns an array containing the set's member objects.
– count	Returns the number of objects in the set.
– containsObject:	Indicates whether a given object is present in the set.

Primitive Methods

– count
– member:
– objectEnumerator

Class Description

The NSSet class declares the programmatic interface to an object that manages an immutable set of objects. NSSet provides support for the mathematical concept of a *set*. A set, both in its mathematical sense and in the implementation of NSSet, is an *unordered* collection of distinct elements. The NSMutableSet and NSCountedSet classes are provided for sets whose contents may be altered.

NSSet provides methods for querying the elements of the set. **allObjects** returns an array containing the objects in a set. **anyObject** returns some object in the set. **count** returns the number of objects currently in the set. **member:** returns the object in the set that is equal to a specified object. Additionally, the **intersectsSet:** tests for set intersection, **isEqualToSet:** tests for set equality, and **isSubsetOfSet:** tests for one set being a subset of another.

The **objectEnumerator** method provides for traversing elements of the set one by one.

NSSet's **makeObjectsPerform:** and **makeObjectsPerform:withObject:** methods provides for sending messages to individual objects in the set.

Exceptions

NSSet implements the **encodeWithCoder:** method, which raises NSInternalInconsistencyException if the number of objects enumerated for encoding turns out to be unequal to the number of objects in the set.

Adopted Protocols

NSCoding
- – encodeWithCoder:
- – initWithCoder:

NSCopying
- – copyWithZone:

NSMutableCopying
- – mutableCopyWithZone:

Method Types

Creating a set

+ allocWithZone:
+ set
+ setWithArray:
+ setWithObject:
+ setWithObjects:
– initWithArray:
– initWithObjects:
– initWithObjects:count:
– initWithSet:
– initWithSet:copyItems:

Counting entries

– count

Accessing the members

– allObjects
– anyObject
– containsObject:
– makeObjectsPerform:
– makeObjectsPerform:withObject:
– member:
– objectEnumerator

Comparing sets

– isSubsetOfSet:
– intersectsSet:
– isEqualToSet:

Describing a set

– description
– descriptionWithLocale:

Class Methods

allocWithZone:

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

Creates and returns an uninitialized set in the specified zone. If the receiver is the NSSet class object, an instance of an immutable private subclass is returned; otherwise, an object of the receiver's class is returned.

Typically, you create temporary sets using the **set...** class methods, not the **allocWithZone:** and **init...** methods. Note that it's your responsibility to free objects created with the **allocWithZone:** method.

See also: **+ set**, **+ setWithObject:**, **+ setWithObjects:**, **+ setWithArray:**

set

+ (id)**set**

Creates and returns an empty set. This method is declared primarily for the use of mutable subclasses of NSSet.

See also: **+ setWithArray:**, **+ setWithObject:**, **+ setWithObjects:**

setWithArray:

+ (id)**setWithArray:**(NSArray *)*anArray*

Creates and returns a set containing those objects contained within the array *anArray.*

See also: **+ set**, **+ setWithObject:**, **+ setWithObjects:**

setWithObject:

+ (id)**setWithObject:**(id)*anObject*

Creates and returns a set containing a single member, *anObject*. *anObject* receives a **retain** message after being added to the set.

See also: **+ setWithArray:**, **+ set**, **+ setWithObjects:**

setWithObjects:

+ (id)**setWithObjects:**(id)*anObject, ...*

Creates and returns a set containing the objects in the argument list. The argument list is a comma-separated list of objects ending with **nil**.

As an example, the following code excerpt creates a set containing three different types of elements (assuming *aPath* exits):

```
NSSet *mySet;
NSData *someData = [NSData dataWithContentsOfFile:aPath];
NSValue *aValue = [NSNumber numberWithInt:5];
NSString *aString = @"a string";

mySet = [NSSet setWithObjects:someData, aValue, aString, nil];
```

See also: + **setWithArray:**, + **set**, + **setWithObject:**

setWithObjects:count:

+ (id)**setWithObjects:**(id *)*objects* **count:**(unsigned int)*count*

Creates and returns a set containing *count* objects from the list of objects specified by *objects*.

setWithSet:

+ (id)**setWithArray:**(NSSet *)*aSet*

Creates and returns a set containing those objects contained within the set *aSet*.

Instance Methods

allObjects

– (NSArray *)**allObjects**

Returns an array containing the receiver's members, or an empty array if the receiver has no members. The order of the objects in the array isn't defined.

anyObject

– (id)**anyObject**

Returns one of the objects in the set (essentially chosen at random), or **nil** if the set contains no objects.

See also: – **allObjects**, – **objectEnumerator**

containsObject:

– (BOOL)**containsObject:**(id)*anObject*

Returns YES if *anObject* is present in the set, NO otherwise.

See also: – **member:**

count

– (unsigned int)**count**

Returns the number of members in the set.

description

– (NSString *)**description**

Returns a string object that represents the contents of the receiver, formatted as a property list.

See also: – **descriptionWithLocale:**

descriptionWithLocale:

– (NSString *)**descriptionWithLocale:**(NSDictionary *)*locale*

Returns a string object that represents the contents of the receiver, formatted as a property list. *locale* specifies options used for formatting each of the receiver's members, each of which is sent **descriptionWithLocale:**with *locale* passed along as the sole parameter. (If the receiver's members do not respond to **descriptionWithLocale:**, this method sends **description** instead.) If you do not want the receiver's members to be formatted, specify **nil** for *locale*.

See also: – **description**

hash

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

Returns an unsigned integer that can be used as a table address in a hash table structure. For a set, **hash** returns the number of members in the set. If two sets are equal (as determined by the **isEqual:** method), they will have the same hash value.

See also: – **isEqual:**

initWithArray:

– (id)**initWithArray:**(NSArray *)*array*

Initializes a newly allocated set with the objects that are contained in *array*. This method steps through *array*, adding members to the new set as it goes. Each object receives a **retain** message as it is added to the set. Returns **self**.

See also: – **initWithObjects:**, – **initWithObjects:count:**, **– initWithSet:, – initWithSet:copyItems:**, +**setWithArray:**

initWithObjects:

– (id)**initWithObjects:**(id)*anObject...*

Initializes a newly allocated set with members taken from the specified list of *objects*. **initWithObjects:** takes a comma-separated list of objects terminated by **nil**. Each object receives a **retain** message as it is added to the set. Returns **self**.

See also: – **initWithArray:**, – **initWithObjects:count:**, **– initWithSet:, – initWithSet:copyItems:**, +**setWithObjects:**

initWithObjects:count:

– (id)**initWithObjects:**(id *)*objects* **count:**(unsigned)*count*

Initializes a newly allocated set with *count* members. This method steps through the *objects* array, creating members in the new set as it goes. Each object receives a **retain** message as it is added to the set. Returns **self**.

See also: – **initWithArray:**, – **initWithObjects:**, **– initWithSet:, – initWithSet:copyItems:**

initWithSet:

– (id)**initWithSet:**(NSSet *)*otherSet*

Initializes a newly allocated set by placing in it the objects contained in *otherSet*. Each object is retained as it is added to the receiver. Returns **self**.

See also: – **initWithArray:**, – **initWithObjects:**, – **initWithObjects:count:**, **– initWithSet:copyItems:**

initWithSet:copyItems:

– (id)**initWithSet:**(NSSet *)*otherSet* **copyItems:**(BOOL)*flag*

Initializes a newly allocated set and, if *flag* is NO, places in it the objects contained in *otherSet*. If *flag* is YES, the members of *otherSet* are copied, and the copies are added to the receiver. (Note that **copyWithZone:** is invoked in making these copies. Thus, the receiver's new member objects may be immutable, even though their counterparts in *otherSet* were mutable. Also, members must conform to the NSCopying protocol)

This method returns **self**.

See also: – **initWithArray:**, – **initWithObjects:**, – **initWithObjects:count:**, **– initWithSet:**

intersectsSet:

– (BOOL)**intersectsSet:**(NSSet *)*otherSet*

Returns YES if at least one object in the receiver is also present in *otherSet*, NO otherwise.

See also: – **isEqualToSet:**, – **isSubsetOfSet:**

isEqual:

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

Returns YES if the receiver and *anObject* are equal; otherwise returns NO. A YES return value indicates that the receiver and *anObject* both inherit from NSSet and contain the same contents (as determined by the **isEqualToSet:** method).

See also: – **isEqualToSet:**

isEqualToSet:

– (BOOL)**isEqualToSet:**(NSSet *)*otherSet*

Compares the receiving set to *otherSet*. If the contents of *otherSet* are equal to the contents of the receiver, this method returns YES. If not, it returns NO.

Two sets have equal contents if they each have the same number of members and if each member of one set is present in the other.

See also: – **intersectsSet:**, – **isEqual:** (NSObject protocol), – **isSubsetOfSet:**

isSubsetOfSet:

– (BOOL)**isSubsetOfSet:**(NSSet *)*otherSet*

Returns YES if every object in the receiver is also present in *otherSet*, NO otherwise.

See also: – **intersectsSet:**, – **isEqualToSet:**

makeObjectsPerform:

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

Sends *aSelector* to each object in the set. The *aSelector* method must be one that takes no arguments. It shouldn't have the side effect of modifying this set. The messages are sent using the **performSelector:** method declared in the NSObject protocol.

See also: – **makeObjectsPerform:withObject:**

makeObjectsPerformSelector:

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

Same as **makeObjectsPerform:**.

makeObjectsPerformSelector:withObject:

– (void)**makeObjectsPerformSelector:**(SEL)*aSelector* **withObject:**(id)*anObject*

Same as **makeObjectsPerform:withObject:**.

makeObjectsPerform:withObject:

– (void)**makeObjectsPerform:**(SEL)*aSelector* **withObject:**(id)*anObject*

Sends *aSelector* to each object in the set. The message is sent each time with *anObject* as the argument, so the *aSelector* method must be one that takes a single argument of type **id**. The *aSelector* method shouldn't, as a side effect, modify this set. The messages are sent using the **performSelector:withObject:** method declared in the NSObject protocol.

See also: – **makeObjectsPerform:**

member:

– (id)**member:**(id)*anObject*

If *anObject* is present in the set (as determined by **isEqual:**), the object in the set is returned. Otherwise, **member:** returns **nil**.

See also: – **containsObject:**

objectEnumerator

– (NSEnumerator *)**objectEnumerator**

Returns an enumerator object that lets you access each object in the set:

```
NSEnumerator *enumerator = [mySet objectEnumerator];
id value;

while ((value = [enumerator nextObject])) {
    /* code that acts on the set's values */
}
```

When this method is used with mutable subclasses of NSSet, your code shouldn't modify the set during enumeration. If you intend to modify the set, use the **allObjects** method to create a "snapshot" of the set's members. Enumerate the snapshot, but make your modifications to the original set.

See also: – **nextObject** (NSEnumerator)

NSMutableSet

Inherits From: NSSet : NSObject

Conforms To: NSCoding
NSCopying
NSMutableCopying (NSSet)
NSObject (NSObject)

Declared In: Foundation/NSSet.h

Class at a Glance

Purpose

An NSMutableSet object stores a modifiable set of objects.

Principal Attributes

- The objects that make up the set.

Creation

+ setWithCapacity:	Returns An empty set with enough allocated memory to hold a specified number of objects.

Commonly Used Methods

– addObject:	Adds an object to the set, if it isn't already a member.
– removeObject:	Removes an object from the set.

Primitive Methods

– addObject:
– removeObject:

Class Description

The NSMutableSet class declares the programmatic interface to an object that manages a mutable set of objects. NSMutableSet provides support for the mathematical concept of a *set*. A set, both in its mathematical sense, and in the NSMutableSet implementation, is an *unordered* collection of distinct elements. The NSCountedSet class, which is a concrete subclass of NSMutableSet, supports mutable sets that can contain multiple instances of the same element. The NSSet class supports creating and managing immutable sets.

Objects are added to an NSMutableSet with **addObject:**, which adds a single object to the set; **addObjectsFromArray:**, which adds all objects from a specified array to the set; or with **unionSet:**, which adds all the objects from another set.

Objects are removed from an NSMutableSet using any of the methods **intersectSet:**, **minusSet:**, **removeAllObjects**, or **removeObject:**.

Method Types

Creating an NSMutableSet

+ allocWithZone:
+ setWithCapacity:
– initWithCapacity:

Adding and removing entries

– addObject:
– removeObject:
– removeAllObjects
– addObjectsFromArray:

Combining and recombining sets

– unionSet:
– minusSet:
– intersectSet:
– setSet:

Class Methods

allocWithZone:

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

Creates and returns an uninitialized mutable set in the specified zone. If the receiver is the NSMutableSet class object, an instance of a mutable private subclass is returned; otherwise, an object of the receiver's class is returned.

Typically, you create temporary sets using the **set...** class methods, not the **allocWithZone:** and **init...** methods.

See also: – **initWithCapacity:**, + **set** (NSSet), + **setWithObjects:count:** (NSSet)

setWithCapacity:

+ (id)**setWithCapacity:**(unsigned)*numItems*

Creates and returns a mutable set, giving it enough allocated memory to hold *numItems* members. Mutable sets allocate additional memory as needed, so *numItems* simply establishes the object's initial capacity.

See also: – **initWithCapacity:**, + **set** (NSSet), + **setWithObjects:count:** (NSSet)

Instance Methods

addObject:

– (void)**addObject:**(id)*anObject*

Adds the specified object to the receiver if it is not already a member. *anObject* is sent a **retain** message as it is added to the receiver. If *anObject* is already present in the set, this method has no effect on either the set or on *anObject*.

See also: – **addObjectsFromArray:**, – **unionSet:**

addObjectsFromArray:

– (void)**addObjectsFromArray:**(NSArray *)*anArray*

Adds each object contained in *anArray* to the receiver, if that object is not already a member. The new member is retained. If a given element of the array is already present in the set, this method has no effect on either the set or on the array element.

See also: – **addObject:**, – **unionSet:**

initWithCapacity:

– (id)**initWithCapacity:**(unsigned)*numItems*

Initializes a newly allocated mutable set, giving it enough allocated memory to hold *numItems* members. Mutable sets allocate additional memory as needed, so *numItems* simply establishes the object's initial capacity. Returns **self**.

See also: **+ setWithCapacity:**

intersectSet:

– (void)**intersectSet:**(NSSet *)*otherSet*

Removes from the receiver each object that isn't a member of *otherSet*. Each object that's removed from the receiver is sent a **release** message.

See also: – **removeObject:**, – **removeAllObjects**, **– minusSet:**

minusSet:

– (void)**minusSet:**(NSSet *)*otherSet*

Removes from the receiver each object contained in *otherSet* that is also present in the receiver. Each object that's successfully removed from the receiver is sent a **release** message. If any member of *otherSet* isn't present in the receiving set, this method has no effect on either the receiver or on the *otherSet* member.

See also: – **removeObject:**, – **removeAllObjects**, **– intersectSet:**

removeAllObjects

– (void)**removeAllObjects**

Empties the set of all of its members. Each member is sent a **release** message.

See also: – **removeObject:**, – **minusSet:**, – **intersectSet:**

removeObject:

– (void)**removeObject:**(id)*anObject*

Removes *anObject* from the set. The removed object is sent a **release** message if it was a member of the receiver.

See also: – **removeAllObjects**, – **minusSet:**, – **intersectSet:**

setSet:

– (void)**setSet:**(NSSet *)*otherSet*

Empties the receiver, then adds each object contained in *otherSet* to the receiver The new member is sent a **retain** message as it is added to the receiver.

unionSet:

– (void)**unionSet:**(NSSet *)*otherSet*

Adds each object contained in *otherSet* to the receiver, if that object is not already a member. The new member is sent a **retain** message as it is added to the receiver. If any member of *otherSet* is already present in the receiver, this method has no effect on either the receiver or on the *otherSet* member.

See also: – **addObject:**, – **addObjectsFromArray:**

NSString Class Cluster

Class Cluster Description

NSString objects represent character strings in OpenStep frameworks. Representing strings as objects allows you to use strings wherever you use other objects. It also provides the benefits of encapsulation, so that string objects can use whatever encoding and storage is needed for efficiency while simply appearing as arrays of characters. The cluster's two public classes, NSString and NSMutableString, declare the programmatic interface for noneditable and editable strings, respectively.

Note: An immutable string is a text string that is defined when it is created and subsequently cannot be changed. An immutable string is implemented as array of Unicode characters (in other words, a text string). To create and manage an immutable string, use the NSString class. To construct and manage a string that can be changed after it has been created, use NSMutableString.

The objects you create using NSString and NSMutableStringare referred to as *string objects* (or, when no confusion will result, merely as *strings*). The term *C string* refers to the standard **char *** type. Because of the nature of class clusters, string objects aren't actual instances of the NSString or NSMutableString classes but of one of their private subclasses. Although a string object's class is private, its interface is public, as declared by these abstract superclasses, NSString and NSMutableString. The string classes adopt the NSCopying and NSMutableCopying protocols, making it convenient to convert a string of one type to the other.

A string object presents itself as an array of Unicode® characters (Unicode is a registered trademark of Unicode, Inc.). You can determine how many characters it contains with the **length** method and can retrieve a specific character with the **characterAtIndex:** method. These two "primitive" methods provide basic access to a string object. Most use of strings, however, is at a higher level, with the strings being treated as single entities: You compare strings against one another, search them for substrings, combine them into new strings, and so on. If you need to access string objects character-by-character, you must understand the Unicode character encoding, specifically issues related to composed character sequences. For details see:

> *The Unicode Standard: Worldwide Character Encoding*, Version 1.0, Volumes 1 and 2. The Unicode Consortium. Addison-Wesley. ISBN 0–201–56788–1 (Volume 1) and 0–201–60845–6 (Volume 2).
>
> Version 2.0, also published by Addison-Wesley, is forthcoming. ISBN 0–201–48345–9.

Creating and Converting String Objects

NSString provides several means for creating instances, most based around the various character encodings it supports. Although string objects always present their own contents as Unicode characters, they can convert their contents to and from many other encodings, such as 7-bit ASCII, ISO Latin 1, EUC, and Shift-JIS. The **availableStringEncodings** class method returns the encodings supported. You can specify an encoding explicitly when converting a C string to or from a string object, or use the *default C string*

encoding, which varies from platform to platform and is returned by the **defaultCStringEncoding** class method.

The simplest way to create a string object in source code is to use either the **stringWithCString:** class method or the **initWithCString:** instance method. Each takes a standard null-terminated C string in the default C string encoding and produces a string object. As a convenience, the Objective-C language also supports the **@"..."** construct to create a string object constant from 7-bit ASCII encoding:

```
NSString *temp = @"/tmp/scratch";
```

Such an object is created at compile time and exists throughout your program's execution. The compiler makes such object constants unique on a per-module basis, and they're never deallocated (though you can retain and release them as you do any other object).

To get a C string from a string object, use the **cString** message. This returns a **char *** in the system's default string encoding, or raises an exception if it can't convert its contents to that encoding. The C string you receive is owned by a temporary object, though, so it will become invalid when automatic deallocation takes place. If you want to get a permanent C string, you must create a buffer and use one of the **getCString:...** methods to fill it. You can find out how large the buffer needs to be with the **cStringLength** method.

Similar methods allow you to create string objects from characters in the Unicode encoding or an arbitrary encoding, and to extract data in these encodings. **initWithData:encoding:** and **dataUsingEncoding:** perform these conversions from and to NSData objects. You can also read a string directly from a file in the Unicode or default C string encoding using the **stringWithContentsOfFile:** class method, and write a string using **writeToFile:atomically:**.

Finally, two types of method allow you to build a string from existing string objects. **localizedStringWithFormat:** and its siblings use a format string as a template into which the values you provide (string and other objects, numerics values, and so on) are inserted. The methods **stringByAppendingString:** and **stringByAppendingFormat:** create a new string by adding one string after another, in the second case using a format string.

In format strings, a '%' character announces a placeholder for a value, with the characters that follow determining the kind of value expected and how to format it. For example, a format string of "%d houses" expects an integer value to be substituted for the format expression "%d". NSString supports the format characters defined for the ANSI C function **printf()**, plus '@' for any object. If the object responds to the **descriptionWithLocale:** message, NSString sends that message to retrieve the text representation, otherwise, it sends a **description** message.

Note: Many compilers perform typecasting of arguments to **printf()**, so that printing an integer into a **%f** (floating-point) placeholder works as expected. This typecasting doesn't occur with NSString's formatted methods, so be sure to cast your values explicitly.

Value formatting is affected by the user's current locale, which is an NSDictionary specifying number, date, and other kinds of formats. NSString uses only the locale's definition for the decimal separator (given by the key named NSDecimalSeparator). If you use a method that doesn't specify a locale, the string assumes the default locale. See "Locales" in the "Other Features" section of the Foundation Kit documentation for more information on locales.

This table summarizes the most common means of creating and converting string objects:

Source	Creation Method	Extraction Method
Default C string encoding	stringWithCString:	getCString: (or cString)
In code	@"..." compiler construct	
Unicode encoding	stringWithCharacters:length:	getCharacters:length:
Arbitrary encoding	initWithData:encoding:	dataUsingEncoding:
File contents	stringWithContentsOfFile:	writeToFile:atomically:
Format string	localizedStringWithFormat:	initWithFormat:locale:
Existing strings	stringByAppendingString:	stringByAppendingFormat:

Working with String Objects

The string classes provide methods for finding characters and substrings within strings and for comparing one string to another. These methods conform to the Unicode standard for determining whether two character sequences are equivalent. The string classes provide comparison methods that handle composed character sequences properly, though you do have the option of specifying a literal search when efficiency is important and you can guarantee some canonical form for composed character sequences.

The search and comparison methods each come in three variants. The simplest version of each searches or compares entire strings. Other variants allow you to alter the way comparison of composed character sequences is performed and to specify a specific range of characters within a string to be searched or compared. You can specify these options (not all options are available for every method):

Search Option	Effect
NSCaseInsensitiveSearch	Ignores case distinctions among characters.
NSLiteralSearch	Performs a byte-for-byte comparison. Differing literal sequences (such as composed character sequences) that would otherwise be considered equivalent are considered not to match. Using this option can speed some operations dramatically.
NSBackwardsSearch	Performs searching from the end of the range toward the beginning.
NSAnchoredSearch	Performs searching only on characters at the beginning or end of the range. No match at the beginning or end means nothing is found, even if a matching sequence of characters occurs elsewhere in the string.

Note: Search and comparison are currently performed as if the NSLiteralSearch option were specified. As the Unicode encoding becomes more widely used, and the need for more flexible comparison increases, the default behavior will be changed accordingly.

Substrings are only found if completely contained within the specified range. If you specify a range for a search or comparison method and don't request NSLiteralSearch, the range must not break composed character sequences on either end; if it does you could get an incorrect result. (See the method description for **rangeOfComposedCharacterSequenceAtIndex:** for a code sample that adjusts a range to lie on character sequence boundaries.)

The basic search and comparison methods are these:

Search methods	Comparison methods
rangeOfString:	compare:
rangeOfString:options:	compare:options:
rangeOfStrings:options:range:	compare:options:range:
rangeOfCharacterFromSet:	
rangeOfCharacterFromSet:options:	
rangeOfCharacterFromSet:options:range:	

The **rangeOfString:...** methods search for a substring within the receiver. The **rangeOfCharacterFromSet:...** methods search for individual characters from a supplied set of characters. The **compare:...** methods return the lexical ordering of the receiver and the supplied string. Several other methods allow you to determine whether two strings are equal or whether one is the prefix or suffix of another, but they don't have variants that allow you to specify search options or ranges.

In addition to searching and comparing strings, you can combine and divide them in various ways. The simplest way to put two strings together is to append one to the other. The **stringByAppendingString:** method returns a string object formed from the receiver and the argument supplied. You can also combine several strings according to a template with the **initWithFormat:**, **stringWithFormat:** and **stringByAppendingFormat:** methods. See "Creating and Converting String Objects" for more information.

You can extract substrings from the beginning or end of a string to a particular index, or from a specific range, with the **substringToIndex:**, **substringFromIndex:**, and **substringWithRange:** methods. You can also split a string into substrings (based on a separator string) with the **componentsSeparatedByString:** method.

Most of the NSString classes' remaining methods are for conveniences such as changing case, quickly extracting numeric values, and working with encodings. There's also a set of methods for treating strings

as file system paths, described below in “Manipulating Paths”. An additional class cluster, NSScanner, allows you to scan a string object for numeric and string values. Both the NSString and the NSScanner class clusters use the NSCharacterSet class cluster for search operations. See the appropriate class specifications for more information.

Manipulating Paths

In addition to all the basic methods for working with character strings merely as strings, NSString also provides a rich set of methods for manipulating strings as file system paths. A string can extract a path’s directory, file name, and extension, expand a tilde expression (such as “~me”) or create one for the user’s home directory, and clean up paths containing symbolic links, redundant slashes, and references to “.” (current directory) and “..” (parent directory). These methods are listed under “Working with paths” in the Method Types section.

NSString represents paths generically with ‘/’ as the path separator and ‘.’ as the extension separator. Methods that accept strings as path arguments convert these generic representations to the proper system-specific form as needed. On systems with an implicit root directory, absolute paths begin with a path separator or with a tilde expression (“~/...” or “~*user*/...”). On systems that require explicit expression of root directories for different devices, such as Microsoft Windows 95, absolute paths begin with the name of the device (for example, “C:/Documents/Paper.doc” to represent the actual path “C:\Documents\Paper.doc”). Where a device must be specified, you can do that yourself—introducing a system dependency—or allow the string object to add a default device.

NSString

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSMutableCopying NSObject (NSObject)
Declared In:	Foundation/NSString.h Foundation/NSPathUtilities.h

Class Description

The NSString class declares the programmatic interface for an object that manages immutable strings. (An immutable string is a text string that is defined when it is created and subsequently cannot be changed. An immutable string is implemented as array of Unicode characters (in other words, a text string). To create and manage a string that can be changed after it has been created, use NSMutableString.)

The NSString class has two primitive methods—**length** and **characterAtIndex:**—that provide the basis for all other methods in its interface. The **length** method returns the total number of Unicode characters in the string. **characterAtIndex:** gives access to each character in the string by index, with index values starting at 0.

NSString declares methods for finding and comparing strings. It also declares methods for reading numeric values from strings, for combining strings in various ways, and for converting a string to different forms (such as encoding and case changes). General use of these methods is presented in the class cluster description under “Working with String Objects”.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

NSCopying

– copyWithZone:

NSMutableCopying

– mutableCopyWithZone:

Method Types

Creating temporary strings

+ stringWithCString:
+ stringWithCString:length:
+ stringWithFormat:
+ localizedStringWithFormat:
+ stringWithContentsOfFile:
+ stringWithCharacters:length:
+ string
+ stringWithString:

Initializing newly allocated strings

– initWithCharactersNoCopy:length:freeWhenDone:
– initWithCharacters:length:
– initWithCStringNoCopy:length:freeWhenDone:
– initWithCString:length:
– initWithCString:
– initWithString:
– initWithFormat:
– initWithFormat:arguments:
– initWithFormat:locale:
– initWithFormat:locale:arguments:
– initWithData:encoding:
– initWithContentsOfFile:
– init

Getting a string's length

– length

Accessing characters

– characterAtIndex:
– getCharacters:
– getCharacters:range:

Combining strings

– stringByAppendingFormat:
– stringByAppendingString:

Dividing strings

– componentsSeparatedByString:
– substringFromIndex:
– substringWithRange:
– substringToIndex:

Finding characters and substrings

– rangeOfCharacterFromSet:
– rangeOfCharacterFromSet:options:
– rangeOfCharacterFromSet:options:range:
– rangeOfString:
– rangeOfString:options:
– rangeOfString:options:range:

Determining line ranges

– getLineStart:end:contentsEnd:forRange:
– lineRangeForRange:

Determining composed character sequences

– rangeOfComposedCharacterSequenceAtIndex:

Converting string contents into a property list

– propertyList
– propertyListFromStringsFileFormat

Identifying and comparing strings

– caseInsensitiveCompare:
– compare:
– compare:options:
– compare:options:range:
– hasPrefix:
– hasSuffix:
– isEqualToString:
– hash

Getting a shared prefix

– commonPrefixWithString:options:

Changing case

– capitalizedString
– lowercaseString
– uppercaseString

Getting C strings

– cString
– lossyCString
– cStringLength
– getCString:
– getCString:maxLength:
– getCString:maxLength:range:remainingRange:

Getting numeric values

– doubleValue
– floatValue
– intValue

Working with encodings

+ availableStringEncodings
+ defaultCStringEncoding
+ localizedNameOfStringEncoding:
– canBeConvertedToEncoding:
– dataUsingEncoding:
– dataUsingEncoding:allowLossyConversion:
– description
– fastestEncoding
– smallestEncoding

Working with paths

+ pathWithComponents:
– pathComponents
– completePathIntoString:caseSensitive:
matchesIntoArray:filterTypes:
– fileSystemRepresentation
– getFileSystemRepresentation:maxLength:
– isAbsolutePath
– lastPathComponent
– pathExtension
– stringByAbbreviatingWithTildeInPath
– stringByAppendingPathComponent:
– stringByAppendingPathExtension:
– stringByDeletingLastPathComponent
– stringByDeletingPathExtension
– stringByExpandingTildeInPath
– stringByResolvingSymlinksInPath
– stringByStandardizingPath
– stringsByAppendingPaths:

Writing to a file

– writeToFile:atomically:

Class Methods

availableStringEncodings

+ (const NSStringEncoding *)**availableStringEncodings**

Returns a zero-terminated list of the encodings that string objects support in the application's environment. Among the more commonly used are:

NSASCIIStringEncoding
NSNEXTSTEPStringEncoding
NSUnicodeStringEncoding
NSISOLatin1StringEncoding
NSISOLatin2StringEncoding
NSSymbolStringEncoding

See the "Types and Constants" section of the Foundation Kit documentation for a complete list and descriptions of supported encodings.

See also: + **localizedNameOfStringEncoding:**

defaultCStringEncoding

+ (NSStringEncoding)**defaultCStringEncoding**

Returns the C string encoding assumed for any method accepting a C string as an argument (these methods use **CString** in the keywords for such arguments; for example, **stringWithCString:**). The default C string encoding is determined from system information, and can't be changed programmatically for an individual process. See the description of NSStringEncoding in the "Types and Constants" section for a full list of supported encodings.

localizedNameOfStringEncoding:

+ (NSString *)**localizedNameOfStringEncoding:**(NSStringEncoding)*encoding*

Returns a human-readable string giving the name of *encoding* in the current locale's language. See "Locales" in the "Other Features" section of the Foundation Kit documentation for more information on locales.

localizedStringWithFormat:

+ (NSString *)**localizedStringWithFormat:**(NSString *)*format, ...*

Returns a string created by using *format* as a template into which the following argument values are substituted according to the formatting information of the current locale. For example, this code excerpt creates a string from another string and an **int**:

```
NSString *myString = [NSString localizedStringWithFormat:@"%@: %d\n",
       @"Cost", 32];
```

The resulting string has the value “Cost: 32\n”.

See “Creating and Converting String Objects” in the class cluster description for more information.

pathWithComponents:

+ (NSString *)**pathWithComponents:**(NSArray *)*components*

Returns a string built from the strings in *components*, by concatenating them with a path separator between each pair. To create an absolute path, use a slash mark “/” as the first component. To include a trailing path divider, use an empty string as the last component. This method doesn’t clean up the path created; use **stringByStandardizingPath** to resolve empty components, references to the parent directory, and so on.

On systems that require an explicit root device for an absolute path, this method add a default device specifier (such as “C:” on Windows systems).

See also: – **pathComponents**

string

+ (id)**string**

Returns an empty string.

See also: – **init**

stringWithCharacters:length:

+ (id)**stringWithCharacters:**(const unichar *)*chars* **length:**(unsigned int)*length*

Returns a string containing *length* characters taken from *chars*, which may not be NULL. This method doesn’t stop short at a zero character.

See also: – **initWithCharacters:length:**

stringWithContentsOfFile:

+ (id)**stringWithContentsOfFile:**(NSString *)*path*

Returns a string created by reading characters from the file named by *path*. If the contents begin with a byte-order mark (U+FEFF or U+FFFE), interprets the contents as Unicode characters; otherwise interprets the contents as characters in the default C string encoding. Returns **nil** if the file can't be opened.

See also: – **initWithContentsOfFile:**, + **defaultCStringEncoding**

stringWithCString:

+ (id)**stringWithCString:**(const char *)*cString*

Returns a string containing derived from the characters in *cString*, which must end with a zero character and which may not be NULL. *cString* should contain characters in the default C string encoding. If the argument passed to **stringWithCString** is not a NULL-terminated C string, the results are undefined.

See also: – **initWithCString:**, + **defaultCStringEncoding**

stringWithCString:length:

+ (id)**stringWithCString:**(const char *)*cString* **length:**(unsigned int)*length*

Returns a string containing characters derived from *cString*, which may not be NULL. *cString* should contain characters in the default C string encoding. This method converts *length* * **sizeof(char)** bytes from *cString*, and doesn't stop short at a zero character.

See also: – **initWithCString:length:**, + **defaultCStringEncoding**

stringWithFormat:

+ (id)**stringWithFormat:**(NSString *)*format, ...*

Returns a string created in the manner of **localizedStringWithFormat:**, but using the default locale to format numbers. See "Creating and Converting String Objects" in the class cluster description for more information.

See also: – **initWithFormat:**

stringWithString:

+ (id)**stringWithString:**(NSString *)*aString*

Returns a string created by copying the characters from *aString*.

See also: – **initWithString:**

Instance Methods

canBeConvertedToEncoding:

– (BOOL)**canBeConvertedToEncoding:**(NSStringEncoding)*encoding*

Returns YES if the receiver can be converted to *encoding* without loss of information. Returns NO if characters would have to be changed or deleted in the process of changing encodings.

If you plan to actually convert a string, the **dataUsingEncoding:...** methods simply return **nil** on failure, so you can avoid the overhead of invoking this method yourself by simply trying to convert the string.

See also: – **dataUsingEncoding:allowLossyConversion:**

capitalizedString

– (NSString *)**capitalizedString**

Returns a string with the first character from each word in the receiver changed to its corresponding uppercase value, and all remaining characters set to their corresponding lowercase values. A “word” here is any sequence of characters delimited by spaces, tabs, or line terminators (listed under **getLineStart:end: contentsEnd:forRange:**). Other common word delimiters such as hyphens and other punctuation aren’t considered, so this method may not generally produce the desired results for multi-word strings.

Case transformations aren’t guaranteed to be symmetrical or to produce strings of the same lengths as the originals. See **lowercaseString** for an example.

See also: – **lowercaseString**, – **uppercaseString**

caseInsensitiveCompare:

– (NSComparisonResult)**caseInsensitiveCompare:**(NSString *)*aString*

This convenience method invokes **compare:options:** with NSCaseInsensitiveSearch as the only option.

characterAtIndex:

– (unichar)**characterAtIndex:**(unsigned int)*index*

Returns the character at the array position given by *index*. Raises an NSRangeException if *index* lies beyond the end of the string.

See also: – **getCharacters:**, – **getCharacters:range:**

commonPrefixWithString:options:

– (NSString *)**commonPrefixWithString:**(NSString *)*aString*
options:(unsigned int)*mask*

Returns a string containing characters that the receiver and *aString* have in common, starting from the beginning of each up to the first characters that aren't equivalent. The returned string is based on the characters of the receiver. For example, if the receiver is "Ma¨dchen" and *aString* is "Mädchenschule", the string returned is "Ma¨dchen", not "Mädchen". The following search options may be specified in *mask* by combining them with the C bitwise OR operator:

NSCaseInsensitiveSearch
NSLiteralSearch

See "Working with String Objects" in the class cluster description for details on these options.

See also: – **hasPrefix:**

compare:

– (NSComparisonResult)**compare:**(NSString *)*aString*

Invokes **compare:options:** with no options.

See also: – **compare:options:range:**, – **caseInsensitiveCompare:**, – **isEqualToString:**

compare:options:

– (NSComparisonResult)**compare:**(NSString *)*aString* **options:**(unsigned int)*mask*

Invokes **compare:options:range:** with *mask* as the options and the receiver's full extent as the range.

See also: – **caseInsensitiveCompare:**, – **isEqualToString:**

compare:options:range:

– (NSComparisonResult)**compare:**(NSString *)*aString*
options:(unsigned int)*mask*
range:(NSRange)*aRange*

Returns NSOrderedAscending if the substring of the receiver given by *aRange* precedes *aString* in lexical ordering, NSOrderedSame if the substring of the receiver and *aString* are equivalent in lexical value, and NSOrderedDescending if the substring of the receiver follows *aString*. You can specify the following options in *mask* by combining them with the C bitwise OR operator:

NSCaseInsensitiveSearch
NSLiteralSearch

See “Working with String Objects” in the class cluster description for details on these options.

Raises an NSRangeException if any part of *aRange* lies beyond the end of the string.

See also: – **caseInsensitiveCompare:**, – **isEqualToString:**

completePathIntoString:caseSensitive:matchesIntoArray:filterTypes:

– (unsigned int)**completePathIntoString:**(NSString **)*outputName*
caseSensitive:(BOOL)*flag*
matchesIntoArray:(NSArray **)*outputArray*
filterTypes:(NSArray *)*filterTypes*

Attempts to perform file- name completion on the receiver, interpreting it as a path in the file system and returning by reference in *outputName* the longest path that matches the receiver. Considers case if *flag* is YES. If *outputArray* is non-NULL, returns all matching file names in an NSArray given by *outputArray*. If an array of strings is provided in *filterTypes*, considers only paths whose extensions (not including the extension separator) match one of those strings.

Returns 0 if no matches are found and 1 if exactly one match is found. In the case of multiple matches, returns the actual number of matching paths if *outputArray* is provided, or simply a positive value if *outputArray* is NULL. Hence, you can check for the existence of matches without retrieving by passing NULL as *outputArray*

componentsSeparatedByString:

– (NSArray *)**componentsSeparatedByString:**(NSString *)*separator*

Returns an NSArray containing substrings from the receiver that have been divided by *separator*. The substrings in the array appear in the order they did in the receiver. If the string begins or ends with the separator, the first or last substring, respectively, is empty. For example, this code excerpt:

```
NSString *list = @"wrenches, hammers, saws";
NSArray *listItems = [list componentsSeparatedByString:@", "];
```

produces an array with these contents:

Index	Substring
0	wrenches
1	hammers
2	saws

If **list** begins with a comma and space—for example, ", wrenches, hammers, saws"—the array has these contents:

Index	Substring
0	*(empty string)*
1	wrenches
2	hammers
3	saws

If **list** has no separators—for example, "wrenches"—the array contains the string itself, in this case "wrenches".

See also: – **componentsJoinedByString:** (NSArray class cluster), – **pathComponents**

cString

– (const char *)**cString**

Returns a representation of the receiver as a C string in the default C string encoding. The returned C string will be automatically freed just as a returned object would be released; your code should copy the C string or use **getCString:** if it needs to store the C string outside of the autorelease context in which the C string is created.

Raises an NSCharacterConversionException if the receiver can't be represented in the default C string encoding without loss of information. Use **canBeConvertedToEncoding:** if necessary to check whether a string can be losslessly converted to the default C string encoding. If it can't, use **lossyCString** or

dataUsingEncoding:allowLossyConversion: to get a C string representation with some loss of information.

See also: – **getCString:**, – **canBeConvertedToEncoding:**, + **defaultCStringEncoding**, – **cStringLength**, – **getCharacters:**

cStringLength

– (unsigned int)**cStringLength**

Returns the length in **char**-sized units of the receiver's C string representation in the default C string encoding.

Raises 0 if the receiver can't be represented in the default C string encoding without loss of information. You can also use **canBeConvertedToEncoding:** to check whether a string can be losslessly converted to the default C string encoding. If it can't, use **lossyCString** to get a C string representation with some loss of information, then check its length explicitly using the ANSI function **strlen()**.

See also: – **cString**, – **canBeConvertedToEncoding:**, + **defaultCStringEncoding**, – **length**

dataUsingEncoding:

– (NSData *)**dataUsingEncoding:**(NSStringEncoding)*encoding*

Invokes **dataUsingEncoding:allowLossyConversion:** with NO as the argument for allowing lossy conversion.

dataUsingEncoding:allowLossyConversion:

– (NSData *)**dataUsingEncoding:**(NSStringEncoding)*encoding*
allowLossyConversion:(BOOL)*flag*

Returns an NSData object containing a representation of the receiver in *encoding*. Returns **nil** if *flag* is NO and the receiver can't be converted without losing some information (such as accents or case). If *flag* is YES and the receiver can't be converted without losing some information, some characters may be removed or altered in conversion. For example, in converting a character from NSUnicodeStringEncoding to NSASCIIStringEncoding, the character 'Á' becomes 'A', losing the accent.

The result of this method, when lossless conversion is made, is the default "plain text" format for *encoding* and is the recommended way to save or transmit a string object.

See also: – **canBeConvertedToEncoding:**

description

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

Returns **self**.

doubleValue

– (double)**doubleValue**

Returns the floating-point value of the receiver's text as a **double**, skipping whitespace at the beginning of the string. Returns HUGE_VAL or –HUGE_VAL on overflow, 0.0 on underflow. Also returns 0.0 if the receiver doesn't begin with a valid text representation of a floating-point number.

This method uses formatting information stored in the default locale; use an NSScanner for localized scanning of numeric values from a string.

See also: – **intValue**, – **floatValue**, – **scanDouble:** (NSScanner)

fastestEncoding

– (NSStringEncoding)**fastestEncoding**

Returns the fastest encoding to which the receiver may be converted without loss of information. "Fastest" applies to retrieval of characters from the string. This encoding may not be space efficient.

See also: – **smallestEncoding**, – **getCharacters:range:**

fileSystemRepresentation

– (const char *)**fileSystemRepresentation**

Returns a file system specific representation of the receiver, as described for **getFileSystemRepresentation:maxLength:**. The returned C string will be automatically freed just as a returned object would be released; your code should copy the representation or use **getFileSystemRepresentation:maxLength:** if it needs to store the representation outside of the autorelease context in which the representation is created.

Raises an NSCharacterConversionException if the receiver can't be represented in the file system's encoding.

floatValue

– (float)**floatValue**

Returns the floating-point value of the receiver's text as a **float**, skipping whitespace at the beginning of the string. Returns HUGE_VAL or –HUGE_VAL on overflow, 0.0 on underflow. Also returns 0.0 if the receiver doesn't begin with a valid text representation of a floating-point number.

This method uses formatting information stored in the default locale; use an NSScanner for localized scanning of numeric values from a string.

See also: – **doubleValue**, – **intValue**, – **scanFloat:** (NSScanner)

getCharacters:

– (void)**getCharacters:**(unichar *)*buffer*

Invokes **getCharacters:range:** with *buffer* and the entire extent of the receiver as the range. *buffer* must be large enough to contain all the characters in the string.

See also: – **length**

getCharacters:range:

– (void)**getCharacters:**(unichar *)*buffer* **range:**(NSRange)*aRange*

Copies characters from *aRange* in the receiver into *buffer*, which must be large enough to contain them. Does *not* add a zero character. Raises an NSRangeException if any part of *aRange* lies beyond the end of the string.

The abstract implementation of this method uses **characterAtIndex:** repeatedly, correctly extracting the characters, though very inefficiently. Subclasses should override it to provide a fast implementation.

getCString:

– (void)**getCString:**(char *)*buffer*

Invokes **getCString:maxLength:range:remainingRange:** with NSMaximumStringLength as the maximum length, the receiver's entire extent as the range, and NULL for the remaining range. *buffer* must be large enough to contain the resulting C string plus a terminating zero character (which this method adds).

Raises an NSCharacterConversionException if the receiver can't be represented in the default C string encoding without loss of information. Use **canBeConvertedToEncoding:** if necessary to check whether a string can be losslessly converted to the default C string encoding. If it can't, use **lossyCString** or

dataUsingEncoding:allowLossyConversion: to get a C string representation with some loss of information.

See also: – **cStringLength**, – **canBeConvertedToEncoding:**, + **defaultCStringEncoding**, – **getCharacters:**

getCString:maxLength:

– (void)**getCString:**(char *)*buffer* **maxLength:**(unsigned int)*maxLength*

Invokes **getCString:maxLength:range:remainingRange:** with *maxLength* as the maximum length in **char**-sized units, the receiver's entire extent as the range, and NULL for the remaining range. *buffer* must be large enough to contain *maxLength* **char**s plus a terminating zero **char** (which this method adds).

Raises an NSCharacterConversionException if the receiver can't be represented in the default C string encoding without loss of information. Use **canBeConvertedToEncoding:** if necessary to check whether a string can be losslessly converted to the default C string encoding. If it can't, use **lossyCString** or **dataUsingEncoding:allowLossyConversion:** to get a C string representation with some loss of information.

See also: – **cStringLength**, – **canBeConvertedToEncoding:**, + **defaultCStringEncoding**, – **getCharacters:**

getCString:maxLength:range:remainingRange:

– (void)**getCString:**(char *)*buffer*
maxLength:(unsigned int)*maxLength*
range:(NSRange)*aRange*
remainingRange:(NSRange *)*leftoverRange*

Converts the receiver's characters to the default C string encoding and stores them in *buffer*. *buffer* must be large enough to contain *maxLength* **char**s plus a terminating zero **char** (which this method adds). Copies and converts as many character as possible from *aRange*, and stores the range of those not converted in the NSRange given by *leftoverRange* (if it's non-NULL). Raises an NSRangeException if any part of *aRange* lies beyond the end of the string.

Raises an NSCharacterConversionException if the receiver can't be represented in the default C string encoding without loss of information. Use **canBeConvertedToEncoding:** if necessary to check whether a string can be losslessly converted to the default C string encoding. If it can't, use **lossyCString** or **dataUsingEncoding:allowLossyConversion:** to get a C string representation with some loss of information.

See also: – **cStringLength**, – **canBeConvertedToEncoding:**, + **defaultCStringEncoding**, – **getCharacters:**

getFileSystemRepresentation:maxLength:

– (BOOL)**getFileSystemRepresentation:**(char *)*buffer* **maxLength:**(unsigned int)*maxLength*

Interprets the receiver as a system-independent path, filling *buffer* with a C string in a format and encoding suitable for use with file system calls. This is done by replacing the abstract path and extension separator characters (‘/’ and ‘.’ respectively) with their equivalents for the operating system. For example, on Microsoft Windows 95 the receiver “C:/Working/Sample.tiff” is returned as the C string “C:\Working\Sample.tiff”.

buffer must be large enough to contain *maxLength* **char**s plus a terminating zero **char** (which this method adds). Returns YES if *buffer* is successfully filled with a file system representation, NO if not (for example, if *maxLength* would be exceeded). Also returns NO if the receiver can’t be represented in the file system’s encoding.

If the system-specific path or extension separator appear in the abstract representation, the characters they’re converted to depend on the system (unless they’re identical to the abstract separators). On Windows 95, for example, a ‘\’ character is converted to ‘/’.

See also: – **fileSystemRepresentation**

getLineStart:end:contentsEnd:forRange:

+ (void)**getLineStart:**(unsigned int *)*startIndex*
end:(unsigned int *)*lineEndIndex*
contentsEnd:(unsigned int *)*contentsEndIndex*
forRange:(NSRange)*aRange*

Returns by reference the indexes of the smallest range of lines containing *aRange*. A line is delimited by any of these characters, the longest possible sequence being preferred to any shorter:

- **U+000D** (**\r** or CR)
- **U+2028** (Unicode line separator)
- **U+000A** (**\n** or LF)
- **U+2029** (Unicode paragraph separator)
- **\r\n**, in that order (also known as CRLF)

When this method returns, *startIndex* contains the index of the first character of the line, which is at or before the location of *aRange; lineEndIndex* contains the index of the first character past the line terminator; and *contentsEndIndex* contains the index of the first character of the line terminator itself. You may pass a NULL pointer for any of these arguments, in which case the work to compute the value isn’t performed.

You can use the results of this method to construct ranges for lines by using the start index as the range’s location and the difference between the end index and the start index as the range’s length.

See also: – **lineRangeForRange:**, – **substringWithRange:**

hash

– (unsigned int)**hash**

Returns an unsigned integer that can be used as a hash table address. If two string objects are equal (as determined by the **isEqualToString:** method), they must have the same hash value. The abstract implementation of this method fulfills this requirement, so subclasses of NSString shouldn't override it.

hasPrefix:

– (BOOL)**hasPrefix:**(NSString *)*aString*

Returns YES if *aString* matches the beginning characters of the receiver, NO otherwise. Returns NO if *aString* is the null string. This method is a convenience for comparing strings using the NSAnchoredSearch option. See "Working with String Objects" in the class cluster description for more information.

See also: – **hasSuffix:**, – **compare:options:range:**

hasSuffix:

– (BOOL)**hasSuffix:**(NSString *)*aString*

Returns YES if *aString* matches the ending characters of the receiver, NO otherwise. Returns NO if *aString* is the null string. This method is a convenience for comparing strings using the NSAnchoredSearch and NSBackwardsSearch options. See "Working with String Objects" in the class cluster description for more information.

See also: – **hasPrefix:**, – **compare:options:range:**

init

– (id)**init**

Initializes the receiver, a newly allocated NSString, to contain no characters. Returns **self**.

See also: + **string**

initWithCharacters:length:

– (id)**initWithCharacters:**(const unichar *)*characters* **length:**(unsigned int)*length*

Initializes the receiver, a newly allocated NSString, by copying *length* characters from *characters*, which may not be NULL. This method doesn't stop short at a zero character. Returns **self**.

See also: + **stringWithCharacters:length:**

initWithCharactersNoCopy:length:freeWhenDone:

– (id)**initWithCharactersNoCopy:**(unichar *)*characters*
length:(unsigned int)*length*
freeWhenDone:(BOOL)*flag*

Initializes the receiver, a newly allocated NSString, to contain *length* characters from *characters*, which may not be NULL. This method doesn't stop short at a zero character. The receiver becomes the owner of *characters;* if *flag* is YES the receiver will free the memory when it no longer needs them, but if *flag* is NO it won't. Returns **self**.

See also: + **stringWithCharacters:length:**

initWithContentsOfFile:

– (id)**initWithContentsOfFile:**(NSString *)*path*

Initializes the receiver, a newly allocated NSString, by reading characters from the file named by *path*. If the contents begin with a byte-order mark (U+FEFF or U+FFFE), interprets the contents as Unicode characters; otherwise interprets the contents as characters in the default C string encoding. Returns **self**, or **nil** if the file can't be opened.

See also: + **stringWithContentsOfFile:**, + **defaultCStringEncoding**

initWithCString:

– (id)**initWithCString:**(const char *)*cString*

Initializes the receiver, a newly allocated NSString, by converting the characters in *cString* from the default C string encoding into the Unicode character encoding. *cString* must be a zero-terminated C string in the default C string encoding, and may not be NULL. Returns **self**.

Note: To create an immutable string from an immutable CString buffer, do not attempt to use this method. Instead, use **initWithCStringNoCopy:length:freeWhenDone:**.

See also: + **stringWithCString:**, **-initWithCStringNoCopy:length:freeWhenDone**, + **defaultCStringEncoding**

initWithCString:length:

– (id)**initWithCString:**(const char *)*cString* **length:**(unsigned int)*length*

Initializes the receiver, a newly allocated NSString, by converting the characters of *cString* from the default C string encoding into the Unicode character encoding. This method converts *length* * **sizeof(char)** bytes

from *cString*, and doesn't stop short at a zero character. *cString* must contain bytes in the default C string encoding, and may not be NULL. Returns **self**.

See also: + **stringWithCString:length:**, + **defaultCStringEncoding**

initWithCStringNoCopy:length:freeWhenDone:

– (id)**initWithCStringNoCopy:**(char *)*cString*
length:(unsigned int)*length*
freeWhenDone:(BOOL)*flag*

Initializes the receiver, a newly allocated NSString, by converting the characters of *cString* from the default C string encoding into the Unicode character encoding. This method converts *length* * **sizeof(char)** bytes from *cString*, and doesn't stop short at a zero character. *cString* must be contain characters in the default C string encoding, and may not be NULL. The receiver becomes the owner of *cString;* if *flag* is YES it will free the memory when it no longer needs it, but if *flag* is NO it won't. Returns **self**.

Note: You can use this method to create an immutable string from an immutable (**const char***) C-string buffer. If you receive a warning message, you can disregard it; its purpose is simply to warn you that the C string passed as the method's first argument may be modified. If you make certain that the **freeWhenDone** argument to **initWithStringNoCopy** is NO, the C string passed as the method's first argument cannot be modified, so you can safely use **initWithStringNoCopy** to create an immutable string from an immutable (**const char***) C-string buffer.

See also: + **stringWithCString:length:**, + **defaultCStringEncoding**

initWithData:encoding:

– (id)**initWithData:**(NSData *)*data* **encoding:**(NSStringEncoding)*encoding*

Initializes the receiver, a newly allocated NSString, by converting the bytes in *data* into Unicode characters. *data* must be an NSData object containing bytes in *encoding* and in the default plain text format (that is, pure content with no attribute or other markup) for that encoding. Returns **self**.

initWithFormat:

– (id)**initWithFormat:**(NSString *)*format*, ...

Invokes **initWithFormat:locale:arguments:** with **nil** as the locale.

See also: + **stringWithFormat:**

initWithFormat:arguments:

– (id)**initWithFormat:**(NSString *)*format* **arguments:**(va_list)*argList*

Invokes **initWithFormat:locale:arguments:** with **nil** as the locale.

See also: + **stringWithFormat:**

initWithFormat:locale:

– (id)**initWithFormat:**(NSString *)*format*
locale:(NSDictionary *)*dictionary*, ...

Invokes **initWithFormat:locale:arguments:** with *dictionary* as the locale.

See also: + **localizedStringWithFormat:**

initWithFormat:locale:arguments:

– (id)**initWithFormat:**(NSString *)*format*
locale:(NSDictionary *)*dictionary*
arguments:(va_list)*argList*

Initializes a newly allocated string object, using *format* as a template into which the following argument values are substituted according to the formatting information of the current locale. For example, this code excerpt creates a string from **myArgs**, which is derived from a string object with the value “Cost:” and an **int** with the value 32:

```
va_list myArgs;
NSDictionary *myLocale;    /* Assume this exists. */

NSString *myString = [[NSString alloc] initWithFormat:@"%@: %d\n",
        locale:[[NSUserDefaults standardUserDefaults] dictionaryRepresentation]
        arguments:myArgs];
```

(Note the message construct for retrieving the user’s locale.). The resulting string has the value “Cost: 32\n”.

See “Creating and Converting String Objects” in the class cluster description for more information. Returns **self**.

See also: – **initWithFormat:arguments:**

initWithString:

– (id)**initWithString:**(NSString *)*aString*

Initializes the receiver, a newly allocated NSString, by copying the characters from *aString*. Returns **self**.

See also: + **stringWithString:**

intValue

– (int)**intValue**

Returns the integer value of the string's text, assuming a decimal representation and skipping whitespace at the beginning of the string. Returns INT_MAX or INT_MIN on overflow. Returns 0 if the receiver doesn't begin with a valid decimal text representation of a number.

This method uses formatting information stored in the default locale; use an NSScanner for localized scanning of numeric values from a string.

See also: – **doubleValue**, – **floatValue**, – **scanInt:** (NSScanner)

isAbsolutePath

– (BOOL)**isAbsolutePath**

Interprets the receiver as a path, returning YES if it represents an absolute path, NO if it represents a relative path. See "Manipulating Paths" in the class description for more information on paths.

isEqualToString:

– (BOOL)**isEqualToString:**(NSString *)*aString*

Returns YES if *aString* is equivalent to the receiver (if they have the same **id** or if they compare as NSOrderedSame), NO otherwise. When you know both objects are strings, this method is a faster way to check equality than **isEqual:**.

See also: – **compare:options:range:**

lastPathComponent

– (NSString *)**lastPathComponent**

Returns the last path component of the receiver. The following table illustrates the effect of **lastPathComponent** on a variety of different paths:

Receiver's String Value	String Returned
"/tmp/scratch.tiff"	"scratch.tiff"
"/tmp/scratch"	"scratch"
"/tmp/"	"tmp"
"scratch"	"scratch"
"/"	"" *(an empty string)*

length

– (unsigned int)**length**

Returns the number of Unicode characters in the receiver. This includes the individual characters of composed character sequences, so you can't use this method to determine if a string will be visible when printed, or how long it will appear.

See also: – **cStringLength**, – **sizeWithAttributes:** (NSString Additions in the Application Kit)

lineRangeForRange:

+ (NSRange)**lineRangeForRange:**(NSRange)*aRange*

Returns the smallest range of lines containing *aRange*, including the characters that terminate the line.

See also: – **getLineStart:end:contentsEnd:forRange:**, – **substringWithRange:**

lossyCString

– (const char *)**lossyCString**

Returns a representation of the receiver as a C string in the default C string encoding, possibly losing information in converting to that encoding (and not raising an exception as **cString** does). The returned C string will be automatically freed just as a returned object would be released; your code should copy the C

string or use **getCString:** if it needs to store the C string outside of the autorelease context in which the C string is created.

See also: – **getCString:**, – **canBeConvertedToEncoding:**, + **defaultCStringEncoding**, – **cStringLength**, – **getCharacters:**

lowercaseString

– (NSString *)**lowercaseString**

Returns a string with each character from the receiver changed to its corresponding lowercase value. Case transformations aren't guaranteed to be symmetrical or to produce strings of the same lengths as the originals. The result of this statement:

```
lcString = [myString lowercaseString];
```

might not be equal to this:

```
lcString = [[myString uppercaseString] lowercaseString];
```

For example, the uppercase form of "ß" in German is "SS", so converting "eßen" to uppercase then lowercase produces this sequence of strings:

"eßen"
"ESSEN"
"essen"

See also: – **capitalizedString**, – **uppercaseString**

pathComponents

– (NSArray *)**pathComponents**

Interprets the receiver as a path, returning an array of strings containing, in order, each path component of the receiver. The strings in the array appear in the order they did in the receiver. If the string begins or ends with the path separator then the first or last component, respectively, is empty. Empty components (caused by consecutive path separators) are deleted. For example, this code excerpt:

```
NSString *path = @"tmp/scratch";
NSArray *pathComponents = [path componentsSeparatedByString:@"/"];
```

produces an array with these contents:

Index	Path Component
0	tmp

Index	Path Component
1	scratch

If the receiver begins with a slash—for example, “/tmp/scratch”—the array has these contents:

Index	Path Component
0	“/”
1	“tmp”
2	“scratch”

If the receiver has no separators—for example, “scratch”—the array contains the string itself, in this case “scratch”.

See also: + **pathWithComponents:**, – **stringByStandardizingPath**, – **componentsSeparatedByString:**

pathExtension

– (NSString *)**pathExtension**

Interprets the receiver as a path, returning the receiver’s extension, if any (not including the extension divider). The following table illustrates the effect of **pathExtension** on a variety of different paths:

Receiver’s String Value	String Returned
“/tmp/scratch.tiff”	“tiff”
“/tmp/scratch”	“” *(an empty string)*
“/tmp/”	“” *(an empty string)*
“/tmp/scratch..tiff”	“tiff”

propertyList

– (id)**propertyList**

Parses the receiver as a text representation of a property list, returning an NSString, NSData, NSArray, or NSDictionary object according to the topmost element. Arrays are delimited by parentheses, with

individual elements separated by commas and optional spaces. Dictionaries are delimited by curly braces, with key-value pairs separated by semicolons, the key and value separated by an equals sign. Strings appear as plain text if they contain no whitespace, or enclosed in straight quotation marks if they do. Data items are delimited by angle brackets and encoded as hexadecimal digits. Here's a short example of a text-format property list:

```
{
    Title = "Star Wars";
    Director = "Lucas, George";
    Cast = (
        "Hamill, Mark",
        "Fisher, Carrie",
        "Ford, Harrison"
    );
    "Thumbnail Image" = <040b7479 70656473 (many more sets of digits) 8484074e>
}
```

See also: – **propertyListFromStringsFileFormat**, + **stringWithContentsOfFile:**

propertyListFromStringsFileFormat

– (NSDictionary *)**propertyListFromStringsFileFormat**

Returns a dictionary object initialized with the keys and values found in the receiver. The receiver must contain text in the format used for **.strings** files. In this format, keys and values are separated by an equals sign, and each key-value pair is terminated with a semicolon. The value is optional, however; if not present, the equals sign is also omitted. The keys and values themselves are always strings enclosed in straight quotation marks. Comments may be included, delimited by **/*** and ***/** as for ANSI C comments. Here's a short example of a strings file:

```
/* Question in confirmation panel for quitting. */
"Confirm Quit" = "Are you sure you want to quit?";

/* Message when user tries to close unsaved document */
"Close or Save" = "Save changes before closing?";

/* Word for Cancel */
"Cancel";
```

See also: – **propertyList**, + **stringWithContentsOfFile:**

rangeOfCharacterFromSet:

– (NSRange)**rangeOfCharacterFromSet:**(NSCharacterSet *)*aSet*

Invokes **rangeOfCharacterFromSet:options:** with no options.

rangeOfCharacterFromSet:options:

– (NSRange)**rangeOfCharacterFromSet:**(NSCharacterSet *)*aSet*
options:(unsigned int)*mask*

Invokes **rangeOfCharacterFromSet:options:range:** with *mask* for the options and the entire extent of the receiver for the range.

rangeOfCharacterFromSet:options:range:

– (NSRange)**rangeOfCharacterFromSet:**(NSCharacterSet *)*aSet*
options:(unsigned int)*mask*
range:(NSRange)*aRange*

Returns the range in the receiver of the first character found from *aSet*. The search is restricted to characters in the receiver within *aRange*. The following options may be specified in *mask* by combining them with the C bitwise OR operator:

NSCaseInsensitiveSearch
NSLiteralSearch
NSBackwardsSearch

See “Working with String Objects” in the class cluster description for details on these options. Raises an NSRangeException if any part of *aRange* lies beyond the end of the string.

Since precomposed characters in *aSet* can match composed characters sequences in the receiver, the length of the returned range can be greater than one. For example, if you search for “ü” in the string “stru¨del”, the returned range is {3,2}.

rangeOfComposedCharacterSequenceAtIndex:

– (NSRange)**rangeOfComposedCharacterSequenceAtIndex:**(unsigned int)*anIndex*

Returns the range in the receiver of the composed character sequence located at *anIndex*. The composed character sequence includes the first base character found at or before *anIndex*, and its length includes the base character and all non-base characters following the base character.

Raises an NSRangeException if *anIndex* lies beyond the end of the string.

If you want to write a method to adjust an arbitrary range so that it includes the composed character sequences on its boundaries, you can create a method such as this:

```
- (NSRange)adjustRange:(NSRange)aRange
{
    unsigned int index, endIndex;
    NSRange newRange, endRange;

    index = aRange.location;
    newRange = [self rangeOfComposedCharacterSequenceAtIndex:index];

    index = aRange.location + aRange.length;
    endRange = [self rangeOfComposedCharacterSequenceAtIndex:index];
    endIndex = endRange.location + endRange.length;

    newRange.length = endIndex - newRange.location;

    return newRange;
}
```

adjustRange: begins by correcting the location for the beginning of *aRange*, storing it in **newRange**. It then works at the end of *aRange*, correcting the location and storing it in **endIndex**. Finally, it sets the length of **newRange** to the difference between **endIndex** and the new range's location.

rangeOfString:

– (NSRange)**rangeOfString:**(NSString *)*aString*

Invokes **rangeOfString:options:** with no options.

rangeOfString:options:

– (NSRange)**rangeOfString:**(NSString *)*aString* **options:**(unsigned int)*mask*

Invokes **rangeOfString:options:range:** with the options specified by *mask* and the entire extent of the receiver as the range.

rangeOfString:options:range:

– (NSRange)**rangeOfString:**(NSString *)*subString*
options:(unsigned int)*mask*
range:(NSRange)*aRange*

Returns an NSRange giving the location and length of the first occurrence of *subString* within *aRange* in the receiver. If *subString* isn't found, the length of the returned NSRange is zero. The length of the returned range and that of *subString* may differ if equivalent composed character sequences are matched. The following options may be specified in *mask* by combining them with the C bitwise OR operator:

NSCaseInsensitiveSearch
NSLiteralSearch
NSBackwardsSearch
NSAnchoredSearch

See “Working with String Objects” in the class cluster description for details on these options. Raises an NSRangeException if any part of *aRange* lies beyond the end of the string. Returns a range of {NSNotFound, 0} if *subString* is the null string.

smallestEncoding

– (NSStringEncoding)**smallestEncoding**

Returns the smallest encoding to which the receiver can be converted without loss of information. This encoding may not be the fastest for accessing characters, but is very space-efficient. This method itself may take some time to execute.

See also: – **fastestEncoding**, – **getCharacters:range:**

stringByAbbreviatingWithTildeInPath

– (NSString *)**stringByAbbreviatingWithTildeInPath**

Returns a string representing the receiver as a path, with a tilde, “~”, substituted for the full path to the current user’s home directory, or “*~user*” for a user other than the current user. Returns the receiver unaltered if it doesn’t begin with the user’s home directory.

See also: – **stringByExpandingTildeInPath**

stringByAppendingFormat:

– (NSString *)**stringByAppendingFormat:**(NSString *)*format, ...*

Returns a string made by appending to the receiver a string constructed from *format* and following arguments in the manner of **stringWithFormat:**.

See also: – **stringByAppendingString:**

stringByAppendingPathComponent:

– (NSString *)**stringByAppendingPathComponent:**(NSString *)*aString*

Returns a string made by appending *aString*, preceded by if necessary by a path separator. The following table illustrates the effect of this method on a variety of different paths, assuming that *aString* is supplied as “scratch.tiff”:

Receiver’s String Value	Resulting String
“/tmp”	“/tmp/scratch.tiff”
“/tmp/”	“/tmp/scratch.tiff”
“/”	“/scratch.tiff”
“” *(an empty string)*	“scratch.tiff”

See also: – **stringsByAppendingPaths:**, – **stringByAppendingPathExtension:**, – **stringByDeletingLastPathComponent**

stringByAppendingPathExtension:

– (NSString *)**stringByAppendingPathExtension:**(NSString *)*string*

Returns a string made by appending to the receiver an extension separator followed by *aString*. The following table illustrates the effect of this method on a variety of different paths, assuming that *aString* is supplied as @“tiff”:

Receiver’s String Value	Resulting String
“/tmp/scratch.old”	“/tmp/scratch.old.tiff”
“/tmp/scratch.”	“/tmp/scratch..tiff”
“/tmp/”	“/tmp/.tiff”
“scratch”	“scratch.tiff”

See also: – **stringByAppendingPathComponent:**, – **stringByDeletingPathExtension**

stringByAppendingString:

– (NSString *)s**tringByAppendingString:**(NSString *)*aString*

Returns a string object made by appending *aString* to the receiver. This code excerpt, for example:

```
NSString *errorTag = @"Error: ";
NSString *errorString = @"premature end of file.";
NSString *errorMessage = [errorTag stringByAppendingString:errorString];
```

produces the string “Error: premature end of file.”.

See also: – **stringByAppendingFormat:**

stringByDeletingLastPathComponent

– (NSString *)**stringByDeletingLastPathComponent**

Returns a string made by deleting the last path component from the receiver, along with any final path separator. If the receiver represents the root path, however, it’s returned unaltered. The following table illustrates the effect of this method on a variety of different paths:

Receiver’s String Value	Resulting String
“/tmp/scratch.tiff”	“/tmp”
“/tmp/lock/”	“/tmp”
“/tmp/”	“/”
“/tmp”	“/”
“/”	“/”
“scratch.tiff”	“” *(an empty string)*

See also: – **stringByDeletingPathExtension**, – **stringByAppendingPathComponent:**

stringByDeletingPathExtension

– (NSString *)**stringByDeletingPathExtension**

Returns a string made by deleting the extension (if any, and only the last) from the receiver. Strips any trailing path separator before checking for an extension. If the receiver represents the root path, however,

it's returned unaltered. The following table illustrates the effect of this method on a variety of different paths:

Receiver's String Value	Resulting String
"/tmp/scratch.tiff"	"/tmp/scratch"
"/tmp/"	"/tmp"
"scratch.bundle/"	"scratch"
"scratch..tiff"	"scratch."
".tiff"	"" *(an empty string)*
"/"	"/"

See also: – **pathExtension**, – **stringByDeletingLastPathComponent**

stringByExpandingTildeInPath

– (NSString *)**stringByExpandingTildeInPath**

Returns a string made by expanding the initial component, if it begins with "~" or "~*user*", to its full path value. Returns the receiver unaltered if that component can't be expanded.

See also: – **stringByAbbreviatingWithTildeInPath**

stringByResolvingSymlinksInPath

– (NSString *)stringByResolvingSymlinksInPath

On Microsoft Windows: Returns **self**.

On Rhapsody and supported systems: Expands an initial tilde expression in the receiving path, then resolves all symbolic links and references to current or parent directories if possible, returning a standardized path. If the original path is absolute, all symbolic links are guaranteed to be removed; if it's a relative path, symbolic links that can't be resolved are left unresolved in the returned string. Returns **self** if an error occurs.

Note: If the name of the receiving path begins with **/private**, the **stringByResolvingSymlinksInPath** method strips off the **/private** designator, provided the result is the name of an existing file.

See also: – **stringByStandardizingPath**, – **stringByExpandingTildeInPath**

stringByStandardizingPath

– (NSString *)stringByStandardizingPath

Returns a string representing the receiving path, with extraneous path components removed. If **stringByStandardizingPath** detects symbolic links in a path name, the **stringByResolvingSymlinksInPath** method is called to resolve them. If an invalid path name is provided, **stringByStandardizingPath** may attempt to resolve it by calling **stringByResolvingSymlinksInPath**, and the results are undefined. If any other kind error is encountered (such as a path component not existing), self is returned**.**

The changes that this method can make in the provided string are:

- An initial tilde expression is expanded using **stringByExpandingTildeInPath**.
- Empty components and references to the current directory (that is, the sequences "//" and "/./") are reduced to single path separators.
- In absolute paths only, references to the parent directory (that is, the component "..") are resolved to the real parent directory if possible using **stringByResolvingSymlinksInPath**, which consults the file system to resolve each potential symbolic link.
- In relative paths, because symbolic links can't be resolved, references to the parent directory are left in place.
- On Mach, an initial component of "/private" is removed from the path if the result still indicates an existing file or directory (checked by consulting the file system).

See also: – **stringByExpandingTildeInPath**, – **stringByResolvingSymlinksInPath**

stringsByAppendingPaths:

– (NSArray *)**stringsByAppendingPaths:**(NSArray *)*paths*

Returns an array of strings made by separately appending each string in *paths* to the receiver, preceded by if necessary by a path separator. See **stringByAppendingPathComponent:** for an individual example.

substringFromIndex:

– (NSString *)**substringFromIndex:**(unsigned int)*anIndex*

Returns a string object containing the characters of the receiver from the one at *anIndex* to the end. Raises an NSRangeException if *anIndex* lies beyond the end of the string.

See also: – **substringWithRange:**, – **substringToIndex:**

substringToIndex:

– (NSString *)**substringToIndex:**(unsigned int)*anIndex*

Returns a string object containing the characters of the receiver up to, but not including, the one at *anIndex*. Raises an NSRangeException if *anIndex* lies beyond the end of the string.

See also: – **substringFromIndex:**, – **substringWithRange:**

substringWithRange:

– (NSString *)**substringWithRange:**(NSRange)*aRange*

Returns a string object containing the characters of the receiver that lie within *aRange*. Raises an NSRangeException if any part of *aRange* lies beyond the end of the string.

See also: – **substringFromIndex:**, – **substringToIndex:**

uppercaseString

– (NSString *)**uppercaseString**

Returns a string with each character from the receiver changed to its corresponding uppercase value. Case transformations aren't guaranteed to be symmetrical or to produce strings of the same lengths as the originals. See **lowercaseString** for an example.

See also: – **capitalizedString**, – **lowercaseString**

writeToFile:atomically:

– (BOOL)**writeToFile:**(NSString *)*path* **atomically:**(BOOL)*flag*

Writes the string's characters to the file named by *path*, returning YES on success and NO on failure. If *flag* is YES, attempts to write the file safely so that an existing file named by *path* is not overwritten, nor does a new file at *path* actually get created, unless the write is successful. The string is written in the default C string encoding if possible (that is, if no information would be lost), in the Unicode encoding otherwise.

If *path* contains a tilde (~) character, you must expand it with **stringByExpandingTildeInPath:** before invoking this method.

See also: + **defaultCStringEncoding**

NSMutableString

Inherits From:	NSString : NSObject
Conforms To:	NSCoding (NSString) NSCopying (NSString) NSMutableCopying (NSString) NSObject (NSObject)
Declared In:	Foundation/NSString.h

Class Description

The NSMutableString class declares the programmatic interface to an object that manages a mutable string—that is, a string whose contents can be edited. To construct and manage an immutable string—or a string that cannot be changed after it has been created—use an object of the NSString class.

An immutable string is implemented as array of Unicode characters (in other words, as a text string). The NSMutableString class adds one primitive method—**replaceCharactersInRange:withString:**—to the basic string-handling behavior inherited from NSString. All other methods that modify a string work through this method. For example, **insertString:atIndex:** simply replaces the characters in a range of zero length, while **deleteCharactersInRange:** replaces the characters in a given range with no characters.

Method Types

Creating temporary strings

+ stringWithCapacity:

Initializing an NSMutableString

– initWithCapacity:

Modifying a string

– appendFormat:
– appendString:
– deleteCharactersInRange:
– insertString:atIndex:
– replaceCharactersInRange:withString:
– setString:

Class Methods

stringWithCapacity:

+ (NSMutableString *)**stringWithCapacity:**(unsigned int)*capacity*

Returns an empty mutable string, using *capacity* as a hint for how much initial storage to reserve.

Instance Methods

appendFormat:

– (void)**appendFormat:**(NSString *)*format*, ...

Adds a constructed string to the receiver. Creates the new string by using NSString's **stringWithFormat:** method with the arguments listed.

See also: – **appendString:**

appendString:

– (void)**appendString:**(NSString *)*aString*

Adds the characters of *aString* to end of the receiver.

See also: – **appendFormat:**

deleteCharactersInRange:

– (void)**deleteCharactersInRange:**(NSRange)*aRange*

Removes the characters in *aRange* from the receiver. Raises an NSRangeException if any part of *aRange* lies beyond the end of the string.

initWithCapacity:

– (id)**initWithCapacity:**(unsigned int)*capacity*

Initializes a newly allocated NSMutableString, using *capacity* as a hint for how much memory to allocate. Returns **self**.

insertString:atIndex:

– (void)**insertString:**(NSString *)*aString* **atIndex:**(unsigned int)*anIndex*

Inserts the characters of *aString* into the receiver, so that the new characters begin at *anIndex* and the existing characters from *anIndex* to the end are shifted by the length of *aString*. Raises an NSRangeException if *anIndex* lies beyond the end of the string.

replaceCharactersInRange:withString:

– (void)**replaceCharactersInRange:**(NSRange)*aRange*
withString:(NSString *)*aString*

Replaces the characters from *aRange* with those in *aString*. Raises an NSRangeException if any part of *aRange* lies beyond the end of the string.

setString:

– (void)**setString:**(NSString *)*aString*

Replaces the characters of the receiver with those in *aString*.

NSTask

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	Foundation/NSTask.h

Class Description

Using NSTask, your program can run another program as a subprocess and can monitor that program's execution. NSTask creates a separate executable entity; it differs from NSThread in that it does not share memory space with the process that creates it.

A task operates within an environment defined by the current values for several items: the current directory, standard input, standard output, standard error, and the values of any environment variables. By default, an NSTask inherits its environment from the process that launches it. If there are any values that should be different for the task, for example, if the current directory should change, you must change the value before you launch the task. A task's environment cannot be changed while it is running.

Creating and Launching an NSTask

There are two ways to create an NSTask. If it's sufficient for the task to run in the environment that it inherits from the process that creates it, use the class method **launchedTaskWithLaunchPath:arguments:**. This method both creates and executes (launches) the task. If you need to change the task's environment, create the task using **alloc** and **init**, use **set...** methods to change parts of the environment, then use the **launch** method to launch the task. For example, the following method runs tasks that take an input file and an output file as arguments. It reads these arguments, the task's executable, and the current directory from text fields before it launches the task:

```
- (void)runTask:(id)sender
{
    NSTask *aTask = [[NSTask alloc] init];
    NSMutableArray *args = [NSMutableArray array];

    /* set arguments */
    [args addObject:[[inputFile stringValue] lastPathComponent]];
    [args addObject:[outputFile stringValue]];
    [aTask setCurrentDirectoryPath:[[inputFile stringValue]
           stringByDeletingLastPathComponent]];
    [aTask setLaunchPath:[taskField stringValue]];
    [aTask setArguments:args];
    [aTask launch];
}
```

If you create an NSTask object in this manner, you must be sure to set the executable name using **setLaunchPath:**. If you don't, an NSInvalidArgumentException is raised.

Ending an NSTask

Normally, you want the task that you've launched to run to completion. When an NSTask exits, it posts an NSTaskDidTerminateNotification to the default notification center. You can add one of the custom objects in your program as an observer of the notification and check the task's exit status (using **terminationStatus**) in the observer method. For example:

```
-(id)init
{
    self = [super init];
    [[NSNotificationCenter defaultCenter] addObserver:self
           selector:@selector(checkATaskStatus:)
           name:NSTaskDidTerminateNotification
           object:nil];
    return self;
}

- (void)checkATaskStatus:(NSNotification *)aNotification
{
    int status = [[aNotification object] terminationStatus];
    if (status == ATASK_SUCCESS_VALUE)
        NSLog(@"Task succeeded.");
    else
        NSLog(@"Task failed.");
}
```

If you need to force a task to end execution, send **terminate** to the NSTask object.

Method Types

Creating and initializing an NSTask

+ launchedTaskWithLaunchPath:arguments:
– init

Returning task information

– arguments
– currentDirectoryPath
– environment
– launchPath
– standardError
– standardInput
– standardOutput

Running and stopping an NSTask

– interrupt
– launch
– terminate
– waitUntilExit

Querying the NSTask state

– isRunning
– terminationStatus

Setting up an NSTask

– setArguments:
– setCurrentDirectoryPath:
– setEnvironment:
– setLaunchPath:
– setStandardError:
– setStandardInput:
– setStandardOutput:

Class Methods

launchedTaskWithLaunchPath:arguments:

+ (NSTask *)**launchedTaskWithLaunchPath:**(NSString *)*path* **arguments:**(NSArray *)*arguments*

Creates and launches a task with the executable specified in *path*, providing the argument in the array *arguments*. The task inherits its environment from the process that invokes this method.

See also: – **init**

Instance Methods

arguments

– (NSArray *)**arguments**

Returns the command arguments that were used when the task was launched.

See also: – **setArguments:**

currentDirectoryPath

– (NSString *)**currentDirectoryPath**

Returns the task's current directory.

See also: – **setCurrentDirectoryPath:**

environment

– (NSDictionary *)**environment**

Returns a dictionary of variables for the environment from which the task was launched. The dictionary keys are the environment variable names.

See also: – **environment** (NSProcessInfo), – **setEnvironment:**

init

– (id)**init**

Returns an initialized NSTask object with the environment of the current process. Usually, if the current process's environment is sufficient, you use the class method **launchedTaskWithLaunchPath: arguments:** to create an run the task. Otherwise, you use **alloc** and **init** and then set up the environment before you launch the task.

interrupt

– (void)interrupt

<< Description forthcoming. >>

isRunning

– (BOOL)**isRunning**

Returns YES if the NSTask is still running, otherwise NO. NO either means the task could not run or it has terminated.

See also: – **launch**, – **terminate**, – **waitUntilExit**

launch

– (void)**launch**

Launches the task represented by the NSTask object. Raises an NSInvalidArgumentException if the launch path has not been set or is invalid or if it fails to create a process.

See also: – **launchPath**, – **setLaunchPath:**, – **terminate**, – **waitUntilExit**

launchPath

– (NSString *)**launchPath**

Returns the path of the NSTask's executable.

See also: + **launchedTaskWithLaunchPath:arguments:**, – **setLaunchPath:**

setArguments:

– (void)**setArguments:**(NSArray *)*arguments*

Sets the command arguments that should be used to launch the path to *arguments*. If this method (or **launchedTaskWithLaunchPath:arguments:**) isn't used, the command is launched with no arguments. You cannot use this method if the task has already been launched. If you do, it raises an NSInvalidArgumentException.

See also: – **arguments**

setCurrentDirectoryPath:

– (void)**setCurrentDirectoryPath:**(NSString *)*path*

Sets the current directory for the task to *path*. If this method isn't used, the current directory is inherited from the process that created the NSTask. You cannot use this method if the task has already been launched. If you do, it raises an NSInvalidArgumentException.

See also: – **currentDirectoryPath**

setEnvironment:

– (void)**setEnvironment:**(NSDictionary *)*environmentDictionary*

Sets the environment for the task to *environmentDictionary*. The environment is a dictionary of environment variable values whose keys are the variable names. If this method isn't used, the environment is inherited from the process that created the NSTask. You cannot use this method if the task has already been launched. If you do, it raises an NSInvalidArgumentException.

See also: – **environment**

setLaunchPath:

– (void)**setLaunchPath:**(NSString *)*path*

Sets the task's executable to path. You must use this method before you send **launch** to launch the task or else use **launchedTaskWithLaunchPath:arguments:**. If you don't, NSTask raises an NSInvalidArgumentException.

See also: – **launchPath**

setStandardError:

– (void)**setStandardError:**(id)*file*

Sets standard error for the task to *file*, which can be either an NSFileHandle or an NSPipe object. If *file* is an NSPipe, launching the NSTask automatically closes the write end of the pipe in the current task. If you're using a pipe for standard error, use an NSPipe instance as the argument to this method. Don't create a handle for the pipe and pass that as the argument. If you do, the write end of the pipe won't be closed automatically.

If this method isn't used, the standard error is inherited from the process that created the NSTask. You cannot use this method if the task has already been launched. If you do, it raises an NSInvalidArgumentException.

See also: – **standardError**

setStandardInput:

– (void)**setStandardInput:**(id)*file*

Sets standard input for the task to *file*, which can be either an NSFileHandle or an NSPipe object. If *file* is an NSPipe, launching the NSTask automatically closes the read end of the pipe in the current task. If you're using a pipe for standard input, use an NSPipe instance as the argument to this method. Don't create a handle for the pipe and pass that as the argument. If you do, the read end of the pipe won't be closed automatically.

If this method isn't used, the standard input is inherited from the process that created the NSTask. You cannot use this method if the task has already been launched. If you do, it raises an NSInvalidArgumentException.

See also: – **standardInput**

setStandardOutput:

– (void)**setStandardOutput:**(id)*file*

Sets standard output for the task to *file*, which can be either an NSFileHandle or an NSPipe object. If *file* is an NSPipe, launching the NSTask automatically closes the write end of the pipe in the current task. If you're using a pipe for standard output, use an NSPipe instance as the argument to this method. Don't create a handle for the pipe and pass that as the argument. If you do, the write end of the pipe won't be closed automatically.

If this method isn't used, the current standard output is inherited from the process that created the NSTask. You cannot use this method if the task has already been launched. If you do, it raises an NSInvalidArgumentException.

See also: – **standardOutput**

standardError

– (id)**standardError**

Returns the standard error file used by the task. Standard error is where all diagnostic messages are sent. The object returned is either an NSFileHandle or an NSPipe instance, depending on what type of object was passed to the **setStandardError:** method.

See also: – **setStandardError:**

standardInput

– (id)**standardInput**

Returns the standard input file used by the task. Standard input is where the task takes its input from unless otherwise specified. The object returned is either an NSFileHandle or an NSPipe instance, depending on what type of object was passed to the **setStandardInput:** method.

See also: – **setStandardInput:**

standardOutput

– (id)**standardOutput**

Returns the standard output file used by the task. Standard output is where the task displays its output. The object returned is either an NSFileHandle or an NSPipe instance, depending on what type of object was passed to the **setStandardOutput:** method.

See also: – **setStandardOutput:**

terminate

– (void)**terminate**

Sends a terminate signal to the NSTask and all of its subtasks, posting a NSTaskDidTerminateNotification to the default notification center. This method has no effect if the NSTask was already launched and has already finished executing. If the task hasn't been launched yet, this method raises an NSInvalidArgumentException.

It is not always possible to terminate the task because the task might be ignoring the terminate signal.

See also: + **launchedTaskWithLaunchPath:arguments:**, – **launch**, – **terminationStatus**, – **waitUntilExit**

terminationStatus

– (int)**terminationStatus**

Returns the value returned by the task's executable. The return value indicates the exit status of the task. Each task defines and documents how its return value should be interpreted. (For example, many commands return 0 if they complete successfully or an error code if they don't. You'll need to look at the documentation for that task to learn what values it returns under what circumstances.) This method raises an NSInvalidArgumentException if the task is still running. Verify that the task is not running before you use it.

```
if (![aTask isRunning]) {
    int return = [aTask terminationStatus];
    if (status == ATASK_SUCCESS_VALUE)
        NSLog(@"Task succeeded.");
    else
        NSLog(@"Task failed.");
}
```

See also: – **terminate**, – **waitUntilExit**

waitUntilExit

– (void)**waitUntilExit**

Suspends your program until the tasks is finished. This method first checks to see if the task is still running using **isRunning**. Then it polls the current run loop using NSDefaultRunLoopMode until the task completes. (See the NSRunLoop class specification for more information on run loops and run loop modes.)

```
int return = [aTask terminationStatus];

[aTask launch];
[aTask waitUntilExit];
return = [aTask terminationStatus];

if (status == ATASK_SUCCESS_VALUE)
    NSLog(@"Task succeeded.");
else
    NSLog(@"Task failed.");
```

See also: – **launch**, – **terminate**

Notifications

NSTaskDidTerminateNotification

Posted when the task has stopped execution. This can be posted either when the task has exited normally or as a result of **terminate** being sent to the NSTask. The observer method can use **terminationStatus** to determine why the task died. See the class description for an example.

This notification contains a notification object but no userInfo dictionary. The notification object is the NSTask that was terminated.

NSThread

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSThread.h

Class Description

An NSThread object controls a thread of execution. Use NSThread when you want to have an Objective-C method run in its own thread of execution or if you need to terminate or delay the current thread.

A *thread* is an executable unit. A *task* is made up of one or more threads. Each thread has its own execution stack and is capable of independent input/output. All threads share the virtual memory address space and communication rights of their task. When a thread is started, it is *detached* from its initiating thread. The new thread runs independently. That is, the initiating thread does not know the new thread's state.

To have an Objective-C message run in its own thread of execution, send the message **detachNewThreadSelector:toTarget:withObject:** to the NSThread class object. This method detaches a new thread from the current thread, and the specified target executes the specified method in that thread. When the target has finished executing the method, the thread exits.

When you use **detachNewThreadSelector:toTarget:withObject:**, your application becomes multithreaded. At any time you can send **isMultiThreaded** to find out if the application is multithreaded, that is, if a thread was ever detached from the main thread. **isMultiThreaded** returns YES even if the detached thread has completed execution.

Note: Do not interchange the use of the **cthreads** functions and NSThread objects within an application. In particular, do not use **cthread_fork()** to create a thread that executes an Objective-C message. **isMultiThreaded** returns YES only if **detachNewThreadSelector:toTarget:withObject:** was used to create the thread.

If you need to terminate the current thread, send the **exit** message to the NSThread class object. Similarly, you send the **sleepUntilDate:** message to the NSThread class object to block the current thread for a period of time.

Method Types

Querying an NSThread

+ isMultiThreaded
+ currentThread
– threadDictionary

Detaching a thread

+ detachNewThreadSelector:toTarget:withObject:

Stopping the Current Thread

+ sleepUntilDate:
+ exit

Class Methods

currentThread

+ (NSThread *)**currentThread**

Returns an object representing the current thread of execution.

See also: + **detachNewThreadSelector:toTarget:withObject:**

detachNewThreadSelector:toTarget:withObject:

+ (void)**detachNewThreadSelector:**(SEL)*aSelector*
toTarget:(id)*aTarget*
withObject:(id)*anArgument*

Detaches a new thread for the message **[***aTarget aSelector***:***anArgument***]**. The method *aSelector* must take only one argument and must not have a return value. The method *aSelector* is responsibile for setting up an autorelease pool for the newly detached thread, and freeing that pool before it exits. The objects *aTarget* and *anArgument* are retained during the execution of the detached thread, then released. The detached thread is exited (using the **exit** class method) as soon as *aTarget* has completed executing the *aSelector* method.

If this is the first thread detached from the current thread, this method posts the NSWillBecomeMultiThreadedNotification with **nil** to the default notification center.

See also: + **currentThread**, + **isMultiThreaded**

exit

+ (void)**exit**

Terminates the current thread, using the **currentThread** class method to access that thread. Before exiting the thread, this method posts the NSThreadWillExitNotification with the thread being exited to the default notification center. Because notifications are delivered synchronously, all observers of NSThreadWillExitNotification are guaranteed to receive the notification before the thread exits.

See also: + **currentThread**, + **sleepUntilDate:**

isMultiThreaded

+ (BOOL)**isMultiThreaded**

Returns YES if the application is multithreaded. An application is considered to be multithreaded if a thread was ever detached from the main thread using **detachNewThreadSelector:toTarget:withObject:**. If you detach a thread using the **cthread_fork()** function, this method returns NO. The detached thread does not have to be running for an application to be considered multithreaded; it may have already exited.

See also: + **detachNewThreadSelector:toTarget:withObject:**

sleepUntilDate:

+ (void)**sleepUntilDate:**(NSDate *)*aDate*

Blocks the current thread until the time specified by *aDate*. No run loop processing occurs while the thread is blocked.

See also: + **currentThread**, + **exit**

Instance Methods

threadDictionary

– (NSMutableDictionary *)**threadDictionary**

Returns the NSThread's dictionary, to which you can add data specific to the receiving NSThread. The thread dictionary is not used during any manipulations of the NSThread; it is simply a place where you can store any interesting attributes of a thread. For example, Foundation uses it to store the thread's NSRunLoop and NSAssertionHandler instances. You may define your own keys for the dictionary. Accessing the thread dictionary may be slower than it usually is to access an NSMutableDictionary.

Notifications

NSWillBecomeMultiThreadedNotification

This notification does not contain a notification object or userInfo dictionary.

Posted when the first thread is detached from the current thread. NSThread posts this notification at most once—the first time a thread is detached using **detachNewThreadSelector:toTarget:withObject:**. Any subsequent invocations of **detachNewThreadSelector:toTarget:withObject:** do not post this notification. The observer methods invoked to receive this notification execute in the main thread, not the new thread, and they execute before the new thread begins executing.

NSThreadWillExitNotification

This notification contains a notification object but no userInfo dictionary. The notification object is the exiting NSThread. NSThread posts this notification when it receives the **exit** message, before the thread exits. Observer methods invoked to receive this notification execute in the exiting thread, before it exits.

NSTimer

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSTimer.h

Class Description

NSTimer creates timer objects, or more simply, *timers*. A timer waits until a certain time interval has elapsed and then fires, sending a specified message to a specified object. For example, you could create an NSTimer that sends a message to a window, telling it to update itself after a certain time interval.

Timers work in conjunction with NSRunLoop objects. NSRunLoops control loops that wait for input, and they use timers to help determine the maximum amount of time they should wait. When the timer's time limit has elapsed, the NSRunLoop fires the timer (causing its message to be sent), then checks for new input. Because of the various input sources a typical run loop manages, the effective resolution of the time interval for an NSTimer is limited to on the order of 50-100 milliseconds.

There are several ways to create a timer. The **scheduledTimerWithTimeInterval**... class methods automatically register the new timer with the current NSRunLoop object in the default mode (NSDefaultRunLoopMode). The **timerWithTimeInterval**... class methods create timers that you may register at a later time by sending the message **addTimer:forMode:** to the NSRunLoop. If you specify that the timer should repeat, it will automatically reschedule itself after it fires.

There is no method that removes the association of a timer from an NSRunLoop—send the timer the **invalidate** message instead. **invalidate** disables the timer, so it will no longer affect the NSRunLoop.

See the NSRunLoop class description for more information on NSRunLoops.

Method Types

Creating a timer

+ scheduledTimerWithTimeInterval:
  invocation:repeats:
+ scheduledTimerWithTimeInterval:target:selector:
  userInfo:repeats:
+ timerWithTimeInterval:invocation:repeats:
+ timerWithTimeInterval:target:selector:
  userInfo:repeats:

Firing a timer

– fire

Stopping a timer

– invalidate

Getting information about a timer

– isValid
– fireDate
– timeInterval
– userInfo

Class Methods

scheduledTimerWithTimeInterval:invocation:repeats:

+ (NSTimer *)**scheduledTimerWithTimeInterval:**(NSTimeInterval)*seconds*
invocation:(NSInvocation *)*invocation*
repeats:(BOOL)*repeats*

Returns a new NSTimer object and registers it with the current NSRunLoop in the default mode. After *seconds* have elapsed, the timer fires, sending *invocation*'s message to its target. If seconds is less than or equal to 0.0, this method chooses a nonnegative interval. If *repeats* is YES, the timer will repeatedly reschedule itself.

scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:

+ (NSTimer *)**scheduledTimerWithTimeInterval:**(NSTimeInterval)*seconds*
target:(id)*target*
selector:(SEL)*aSelector*
userInfo:(id)*userInfo*
repeats:(BOOL)*repeats*

Returns a new NSTimer object and registers it with the current NSRunLoop in the default mode. After *seconds* have elapsed, the timer fires, sending the message *aSelector* to *target*. The *aSelector* method must take only one argument, an NSTimer object. The timer passes itself as the argument to *aSelector*. To pass more information to the target, use *userInfo*. The target gets *userInfo* by sending **userInfo** to the timer.

If *seconds* is less than or equal to 0.0, this method chooses a nonnegative interval. If *repeats* is YES, the timer will repeatedly reschedule itself.

timerWithTimeInterval:invocation:repeats:

+ (NSTimer *)**timerWithTimeInterval:**(NSTimeInterval)*seconds*
invocation:(NSInvocation *)*invocation*
repeats:(BOOL)*repeats*

Returns a new NSTimer that, when registered, will fire after *seconds*. If *seconds* is less than or equal to 0.0, this method chooses a nonnegative interval. Upon firing, the timer sends *invocation*'s message to its target. If *repeats* is YES, the timer will repeatedly reschedule itself after firing.

timerWithTimeInterval:target:selector:userInfo:repeats:

+ (NSTimer *)**timerWithTimeInterval:**(NSTimeInterval)*seconds*
target:(id)*target*
selector:(SEL)*aSelector*
userInfo:(id)*userInfo*
repeats:(BOOL)*repeats*

Returns a new NSTimer that, when registered, will fire after *seconds*. Upon firing, the timer sends *aSelector* to *target*. The *aSelector* method must take only one argument, an NSTimer object. The timer passes itself as the argument to *aSelector*. To pass more information to the target, use *userInfo*. The target gets *userInfo* by sending **userInfo** to the timer.

If *seconds* is less than or equal to 0.0, this method chooses a nonnegative interval. If *repeats* is YES, the timer will repeatedly reschedule itself.

Instance Methods

fire

– (void)**fire**

Causes the receiver's message to be sent to its target.

fireDate

– (NSDate *)**fireDate**

Returns the date at which the receiver will fire. If the timer is no longer valid, this method returns the last date at which the timer fired. Use **isValid** to verify that the timer is valid.

See also: – **isValid**

invalidate

– (void)**invalidate**

Stops the receiver from ever firing again. This is the only way to remove a timer from an NSRunLoop.

isValid

– (BOOL)**isValid**

Returns YES if the timer is currently valid, no otherwise.

timeInterval

– (NSTimeInterval)**timeInterval**

Returns the time interval associated with the receiver.

userInfo

– (id)**userInfo**

Additional data the target may use when the receiver is fired.

See also: + **scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:**,
+ **timerWithTimeInterval:target:selector:userInfo:repeats:**

NSTimeZone Class Cluster

Class Cluster Description

From the NSTimeZone class cluster, you can obtain immutable objects that represent time zones. Time zones are split at lines of longitude and are split latitudinally at political boundaries. For example, US/Pacific and Canada/Pacific are two time zone regions. They are roughly the same time zone, but they differ when Daylight Savings Time is in effect.

The abstract class, NSTimeZone, represents knowledge about a geopolitical time zone region (such as “US/Pacific” or “Canada/Pacific”), its colloquial abbreviation (such as “PST”), its offset from Greenwich Mean Time (GMT), and whether it is a Daylight Savings Time time zone.

NSTimeZone objects are used in conjunction with NSDate objects. NSDate objects represent dates as *universal time*. Universal time treats a date and time value as identical in, for instance, Redwood City and New York City. There is no provision for locale and, consequently, for time zone. Provision for locale is critical for string representations and other expressions of conventional dates and times. NSTimeZone expands the temporal value of date objects so that they reflect locale information related to time zones.

NSTimeZone objects are referred to simply as *time zone objects*. Because of the nature of class clusters, time zone objects returned by this class are *not* instances of the abstract class, but of one of its private subclasses. Although a time zone object’s class is private, its interface is public, as declared by the abstract superclass, NSTimeZone. (See the “Class Clusters” programming topic for more information on class clusters and creating subclasses within a cluster.)

You use the class methods **defaultTimeZone**, **localTimeZone**, **timeZoneWithName:**, **timeZoneWithAbbreviation:**, and **timeZoneForSecondsFromGMT:** to get suitable time zone objects. The instance method **timeZoneForDate:** returns the time zone object for a specific date.

Subclassing NSTimeZone is not recommended.

NSTimeZone

Inherits From:	NSObject
Conforms To:	NSCoding, NSCopying NSObject (NSObject)
Declared In:	Foundation/NSDate.h

Class Description

NSTimeZone is an abstract class that defines the behavior of time zone objects. Time zone objects represent geopolitical regions. Consequently, these objects have names for these regions. Time zone objects also represent a temporal offset, either plus or minus, from Greenwich Mean Time (GMT) and an abbreviation (such as “PST”).

The system supplies many choices for time zones. The system stores time zone information in the directory **System/Library/Frameworks/Foundation.framework/Resources/TimeZoneInfo**. The files in this directory store time zone information. The NSTimeZone class object can provide you with the information in this directory. The class method **abbreviationDictionary** selects a subset of geopolitical regions that provide a non-overlapping set of abbreviations. For example, the geopolitical regions US/Pacific and Canada/Pacific are both in the same longitudinal area and both use the abbreviation “PST.” **abbreviationDictionary** limits the choices for an abbreviation so there is one geopolitical region per abbreviation. Depending on where you are, the following message will return either “US/Pacific” or “Canada/Pacific”.

```
[[NSTimeZone abbreviationDictionary] objectForKey:@"PST"];
```

The **RegionsDictionary** file in the time zone information directory supplies these methods with the information necessary to make such choices.

NSTimeZone provides several class methods to get time zone objects: **timeZoneWithName:**, **timeZoneWithAbbreviation:**, and **timeZoneForSecondsFromGMT:**. The class also permits you to set the default time zone for your locale (**setDefaultTimeZone:**) You can access this default time zone at any time with the **defaultTimeZone** class method, and with the **localTimeZone** class method, you can get a relative time zone object that decodes itself to become the default time zone for any locale in which it finds itself.

Some NSCalendarDate methods return date objects that are automatically bound to time zone objects. These date objects use the functionality of NSTimeZone to adjust dates for the proper locale. Unless you specify otherwise, objects returned from NSCalendarDate are bound to the default time zone for the current locale. A useful instance method is **timeZoneForDate:**, which returns a time zone object associated with a specific date.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

NSCopying

– copyWithZone:

Method Types

Getting time zones

+ timeZoneWithAbbreviation:
+ timeZoneWithName:
+ timeZoneForSecondsFromGMT:
– timeZoneForDate:

Getting the default time zone

+ localTimeZone
+ defaultTimeZone
+ setDefaultTimeZone:

Getting time zone information

+ abbreviationDictionary

Getting information about a specific time zone

– timeZoneName
– timeZoneAbbreviation
– timeZoneSecondsFromGMT

Class Methods

abbreviationDictionary

+ (NSDictionary *)**abbreviationDictionary**

Returns a dictionary holding the mappings of time zone abbreviations to time zone names. More than one abbreviation may map to the same time zone. For example, the “US/Pacific” region is associated with both the abbreviations “PST” and “PDT.”

More than one time zone may have the same abbreviation. For example, US/Pacific and Canada/Pacific both use the abbreviation “PST.” In these cases **abbreviationDictionary** chooses a single name to map the abbreviation to.

defaultTimeZone

+ (NSTimeZone *)**defaultTimeZone**

Returns the default time zone set for the current locale. If no default time zone has been set, this method invokes **systemTimeZone** and returns the system time zone.

See also: **+ localTimeZone**, **+ setDefaultTimeZone:**, **+ systemTimeZone**

localTimeZone

+ (NSTimeZone *)**localTimeZone**

Returns an object that forwards all messages to the default time zone for the current locale. This behavior is particularly useful for NSCalendarDate objects that are archived or sent as Distributed Objects and may be interpreted in different locales.

See also: **+ defaultTimeZone**, **+ setDefaultTimeZone:**

knownTimeZoneNames

+ (NSArray *)**knownTimeZoneNames**

Returns an array of strings listing the names of all the time zones known to the system.

resetSystemTimeZone

+ (void)**resetSystemTimeZone**

Clears the previously determined system time zone, if any. Subsequent calls to **systemTimeZone** will attempt to redetermine the system time zone.

setDefaultTimeZone:

+ (void)**setDefaultTimeZone**:(NSTimeZone *)*aTimeZone*

Sets the time zone appropriate for the current locale. There can be only one default time zone, so by setting a new default time zone, you lose the previous one.

See also: **+ defaultTimeZone**, **+ localTimeZone**

systemTimeZone

+ (NSTimeZone *)**systemTimeZone**

Returns the time zone currently used by the system. If it can't figure out the current time zone, returns the GMT time zone.

timeZoneForSecondsFromGMT:

+ (NSTimeZone *)**timeZoneForSecondsFromGMT**:(int)*seconds*

Returns a time zone object with *seconds* offset from Greenwich Mean Time. The name of the new time zone is GMT +/- the offset, in hours and minutes.

See also: + **timeZoneWithAbbreviation:**, + **timeZoneWithName:**

timeZoneWithAbbreviation:

+ (NSTimeZone *)**timeZoneWithAbbreviation:**(NSString *)*abbreviation*

Returns the time zone object identified by *abbreviation* by resolving the abbreviation to a name using the abbreviation dictionary, and then returning the time zone for that name. Returns **nil** if there is no match for *abbreviation*.

See also: + **abbreviationDictionary**, + **timeZoneForSecondsFromGMT:**, + **timeZoneWithName:**

timeZoneWithName:

+ (NSTimeZone *)**timeZoneWithName:**(NSString *)*aTimeZoneName*

Returns the time zone object identified by the name *aTimeZoneName*. It searches the time zone information directory for matching names. Returns **nil** if there is no match on the name.

See also: + **timeZoneForSecondsFromGMT:**, + **timeZoneWithAbbreviation:**, + **knownTimeZoneNames**

timeZoneWithName:data:

+ (NSTimeZone *)**timeZoneWithName:**(NSString *)*aTimeZoneName* **data:**(NSData *)*data*

Returns the time zone with the name *aTimeZoneName* whose data has been initialized using the contents of *data*. You should not call this method directly—use **timeZoneWithName:** to get the time zone object for a given name.

See also: + **timeZoneWithName:**

Instance Methods

abbreviation

– (NSString *)**abbreviation**

Returns the abbreviation for the time zone, such as “EDT” (Eastern Daylight Time). Invokes **abbreviationForDate:** with the current date as the argument.

abbreviationForDate:

– (NSString *)**abbreviationForDate:**(NSDate *)*aDate*

Returns the abbreviation for the time zone object at the specified date . Note that the abbreviation may be different at different dates. For example, during Daylight Savings Time the US/Eastern time zone has an abbreviation of “EDT.” At other times, its abbreviation is “EST.”

data

– (NSData *)**data**

Returns the data that stores the information used by the time zone. This data should be treated as an opaque object.

description

– (NSString *)**description**

Returns the description of the timezone, including the name, abbreviation, offset from GMT, and whether or not DayligSavings Time is currently in effect.

initWithName:

– (id)**initWithName:**(NSString *)*aName*

Initializes a time zone with *aName*. If *aName* is a known name, this method calls **initWithName:data:** with the appropriate data object.

initWithName:data:

– (id)**initWithName:**(NSString *)*aName* **data:**(NSData *)*data*

Initializes a time zone with *aName* and *data*. You should not call this method directly—use **timeZoneWithName:** to get a time zone object.

isDaylightSavingTime

– (BOOL)**isDaylightSavingTime**

Returns YES if the time zone is currently using Daylight Savings Time. This method invokes **isDaylightSavingTimeForDate:** with the current date as the argument.

isDaylightSavingTimeForDate:

– (BOOL)**isDaylightSavingTimeForDate:**(NSDate *)*aDate*

Returns YES if the time zone uses Daylight Savings Time at *aDate*.

isEqualToTimeZone:

– (BOOL)**isEqualToTimeZone:**(NSTimeZone *)*aTimeZone*

Returns YES if *aTimeZone* and the receiving time zone have the same name and data.

name

– (NSString *)**name**

Returns the name of the time zone as returned by **timeZoneName**.

secondsFromGMT

– (int)**secondsFromGMT**

Returns the current difference in seconds between the time zone and Greenwich Mean Time.

secondsFromGMTForDate:

– (int)**secondsFromGMTForDate:**(NSDate *)*aDate*

Returns the difference in seconds between the time zone and Greenwich Mean Time at *aDate*. This may be different from the current difference if the time zone changes its offset from GMT at different points in the year—for example, the U.S. time zones change with daylight savings time.

timeZoneAbbreviation

– (NSString *)**timeZoneAbbreviation**

Returns the abbreviation for the time zone object, such as “EDT” (Eastern Daylight Time).

timeZoneForDate:

– (NSTimeZone *)**timeZoneForDate:**(NSDate *)*date*

Returns the time-zone object that is associated with the *date*. Invoke this method when a region’s time zone (that is, its offset value from GMT) varies over the year, as happens between Standard Time and Daylight Savings Time.

timeZoneName

– (NSString *)**timeZoneName**

Returns the geopolitical region name that identifies the time zone.

timeZoneSecondsFromGMT

– (int)**timeZoneSecondsFromGMT**

Returns the difference in seconds between the receiver and GMT. This offset can be a positive or negative value.

NSUnarchiver

Inherits From:	NSCoder : NSObject
Conforms To:	NSObject (NSObject)
Declared In:	Foundation/NSArchiver.h

Class Description

NSUnarchiver, a concrete subclass of NSCoder, defines methods for decoding a set of Objective-C objects from an archive. Such archives are produced by objects of the NSArchiver class. See the NSCoder and NSArchiver specifications for an introduction to archiving.

General Exception Conditions

While unarchiving, NSUnarchiver performs a variety of consistency checks on the incoming data stream. NSUnarchiver raises an NSInconsistentArchiveException when:

- A class name is missing where one is expected.
- A class name is found that refers to an unknown class.
- A type code is found that's different from the one expected.
- An unknown type code is found.
- Excess characters are found in a type code, or characters are missing.

For a description of type codes, see the discussion of the **@encode()** compiler directive in *Object-Oriented Programming and the Objective-C Language*.

Invoking inappropriate methods can also lead to errors. NSUnarchiver's superclass, NSCoder, provides methods for both encoding and decoding. However, only the decoding methods are applicable to NSUnarchiver; don't send an NSUnarchiver any **encode...** messages.

Method Types

Initializing an NSUnarchiver

– initForReadingWithData:

Decoding objects

+ unarchiveObjectWithData:
+ unarchiveObjectWithFile:

Managing an NSUnarchiver

– isAtEnd
– objectZone
– setObjectZone:
– systemVersion

Substituting classes or objects

+ classNameDecodedForArchiveClassName:
+ decodeClassName:asClassName:
– classNameDecodedForArchiveClassName:
– decodeClassName:asClassName:
– replaceObject:withObject:

Class Methods

classNameDecodedForArchiveClassName:

+ (NSString *)**classNameDecodedForArchiveClassName:**(NSString *)*nameInArchive*

Returns the name of the class used when instantiating objects whose ostensible class, according to the archived data, is *nameInArchive*. This method returns *nameInArchive* if no substitute name has been specified using the class method (not the instance method) **decodeClassName:asClassName:**.

Note that individual instances of NSUnarchiver can each be given their own class name mappings by invoking the instance method **decodeClassName:asClassName:**. The NSUnarchiver class has no information about these instance-specific mappings, however, so they don't affect the return value of this class method (that is, **classNameDecodedForArchiveClassName:**).

See also: – **classNameDecodedForArchiveClassName:**

decodeClassName:asClassName:

+ (void)**decodeClassName:**(NSString *)*nameInArchive* **asClassName:**(NSString *)*trueName*

Instructs instances of NSUnarchiver to use the class named *trueName* when instantiating objects whose ostensible class, according to the archived data, is *nameInArchive*. This method enables easy conversion of unarchived data when the name of a class has changed since the archive was created.

Note that there's also an instance method of the same name. An instance of NSUnarchiver can maintain its own mapping of class names. However, if both the class method and the instance method have been invoked using an identical value for *nameInArchive*, the class method takes precedence.

See also: + **classNameDecodedForArchiveClassName:**, – **decodeClassName:asClassName:**

unarchiveObjectWithData:

+ (id)**unarchiveObjectWithData:**(NSData *)*data*

Decodes and returns the object archived in *data*. This method invokes **initForReadingWithData:** and **decodeObject** to create a temporary NSUnarchiver that decodes the object. If the archived object is the root of a graph of objects, the entire graph is unarchived.

See also: – **encodeRootObject:** (NSArchiver)

unarchiveObjectWithFile:

+ (id)**unarchiveObjectWithFile:**(NSString *)*path*

Decodes and returns the object archived in the file *path*. This convenience method reads the file by invoking NSData's **dataWithContentsOfFile:** method, and then invokes **unarchiveObjectWithData:**.

Instance Methods

classNameDecodedForArchiveClassName:

– (NSString *)**classNameDecodedForArchiveClassName:**(NSString *)*nameInArchive*

Returns the name of the class that will be used when instantiating objects whose ostensible class, according to the archived data, is *nameInArchive*. This method returns *nameInArchive* unless a substitute name has been specified using the instance method (not the class method) **decodeClassName:asClassName:**.

See also: + **classNameDecodedForArchiveClassName:**

decodeClassName:asClassName:

– (void)**decodeClassName:**(NSString *)*nameInArchive* **asClassName:**(NSString *)*trueName*

Instructs the receiver to use the class named *trueName* when instantiating objects whose ostensible class, according to the archived data, is *nameInArchive*. This method enables easy conversion of unarchived data when the name of a class has changed since the archive was created.

Note that there's also a class method of the same name. The class method has precedence in case of conflicts.

See also: – **classNameDecodedForArchiveClassName:**, + **decodeClassName:asClassName:**

initForReadingWithData:

– (id)**initForReadingWithData:**(NSData *)*data*

Initializes an NSUnarchiver object from the data object *data*, decoding the system version number that was archived in *data* and preparing the NSUnarchiver for a subsequent invocation of **decodeObject**. Raises an NSInvalidArgumentException if *data* is **nil**.

See also: – systemVersion

isAtEnd

– (BOOL)**isAtEnd**

Returns YES if the NSUnarchiver has reached the end of the encoded data while decoding, NO if more data follows. You can invoke this method after invoking **decodeObject** to discover whether the archive contains extra data following the encoded object graph. If it does, you can either ignore this anomaly or consider it an error.

objectZone

– (NSZone *)**objectZone**

Returns the memory zone used to allocate decoded objects.

See also: – setObjectZone:

replaceObject:withObject:

– (void)**replaceObject:**(id)*object* **withObject:**(id)*newObject*

Causes the NSUnarchiver to substitute *newObject* for *object* whenever *object* is extracted from the archive. *newObject* can be of a different class from *object*, and the class mappings set by the two **decodeClassName: asClassName:** methods are ignored.

setObjectZone:

– (void)**setObjectZone:**(NSZone *)*zone*

Set the memory zone used to allocate decoded objects. If *zone* is NULL, or if this method is never invoked, the default zone will be used, as given by **NSDefaultMallocZone()**.

See also: – objectZone

systemVersion

– (unsigned int)**systemVersion**

Returns the system version number that was in effect when the archive was created. This information is available as soon as the NSUnarchiver has been initialized.

The version numbers aren’t the usual release designations (such as 2.0 or 3.1). By convention, version numbers under 1000 refer to early versions of NEXTSTEP that didn’t conform to the OpenStep specification.

NSUndoManager

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	Foundation/NSUndoManager.h

Class Description

NSUndoManager is a general-purpose recorder of operations for undo and redo. You register an undo operation by specifying the object that's changing (or the owner of that object), along with a method to invoke to revert its state, and the arguments for that method. NSUndoManager groups all operations within a single cycle of the run loop, so that performing an undo reverts all changes that occurred during the loop. Also, when performing undo an NSUndoManager saves the operations reverted so that you can redo the undos.

NSUndoManager is implemented as a class of the Foundation framework because executables other than applications might want to revert changes to their states. For example, you might have an interactive command-line tool with undo and redo commands; or there could be Distributed Object implementations that can revert operations "over the wire." However, users typically see undo and redo as application features. The Application Kit implements undo and redo in its NSTextView object and makes it easy to implement it in objects along the responder chain. For more on the role of the Application Kit in undo and redo, see "“Undo in Applications”."

Operations and Groups

An *undo operation* is a method for reverting a change to an object, along with the arguments needed to revert the change (for example, its state before the change). Undo operations are typically collected in *undo groups*, which represent whole revertable actions, and are stored on a stack. *Redo operations* and groups are simply undo operations stored on a separate stack (described below). When an NSUndoManager performs undo or redo, it's actually undoing or redoing an entire group of operations. For example, a user could change the type face and the font size of some text. An application might package both attributed-setting operations as a group, so when the user choses Undo, both type face and font size are reverted. To undo a single operation, it must still be packaged in a group.

NSUndoManager normally creates undo groups automatically during the run loop. The first time it's asked to record an undo operation in the run loop, it creates a new group. Then, at the end of the loop, it closes the group. You can create additional, nested undo groups within these default groups using the **beginUndoGrouping** and **enableUndoRegistration** methods. You can also turn off the default grouping behavior using **setGroupsByEvent:**.

The Undo and Redo Stacks

Undo groups are stored on a stack, with the oldest groups at the bottom and the newest at the top. The undo stack is unlimited by default, but you can restrict it to a maximum number of groups using the **setLevelsOfUndo:** method. When the stack exceeds the maximum, the oldest undo groups are dropped from the bottom.

Initially, both stacks are empty. Recording undo operations adds to the undo stack, but the redo stack remains empty until undo is performed. Performing undo causes the reverting operations in the latest group to be applied to their objects. Since these operations cause changes to the objects' states, the objects presumably register new operations with the NSUndoManager, this time in the reverse direction from the original operations. Since the NSUndoManager is in the process of performing undo, it records these operations as redo operations on the redo stack. Consecutive undos add to the redo stack. Subsequent redo operations pull the operations off the redo stack, apply them to the objects, and push them back onto the undo stack.

The redo stack's contents last as long as undo and redo are performed successively. However, because applying a new change to an object invalidates the previous changes, as soon as a new undo operation is registered, the redo stack is cleared. This prevents redo from returning objects to an inappropriate prior state. You can check for the ability to undo and redo with the **canUndo** and **canRedo** methods.

Registering Undo Operations

To add an undo operation to the undo stack, you must register it with the object that will perform the undo operation. NSUndoManager supports two ways to register undo operations: one based on a simple selector with a single object argument, and one based on a general NSInvocation (which allows any number and type of arguments). In the first type of operation, when an object changes, the object itself (or another object acting on its behalf) records in the object passed as argument the attributes of the object prior to the change. (This argument is frequently an NSDictionary object, but it can be any object.) Performing the undo then involves resetting the object with these attributes. Invocation-based undo is useful for undoing specific state-changing methods, such as a **setFont:color:** method.

In most applications a single instance of NSUndoManager belongs to an object that contains or manages other objects. This is particularly the case with document-based applications, where each NSDocument object is responsible for all undo and redo operations for a document. An object such as this is often called the NSUndoManager's *client*. Each client object has its own NSUndoManager. The client claims exclusive right to alter its undoable objects so that it can record undo operations for all changes. In the specific case of documents, this scheme keeps each pair of undo and redo stacks separate so that when an undo is performed, it applies to the focal document in the application (typically the one displayed in the key window). It also relieves the individual objects in a document from having to know the identity of their NSUndoManager or from having to track changes to themselves.

However, an object that is changed can have its own NSUndoManager and perform its own undo and redo operations. For example, you could have a custom view that displays images dragged into it; with each

successful drag operation, it registers a new undo group. If the view is then selected (that is, made first responder) and the Undo command applied, the previously displayed image would be redisplayed.

Simple Undo

To record a simple undo operation, you need only invoke **registerUndoWithTarget:selector:object:**, giving the object to be sent the undo operation selector, the selector to invoke, and an argument to pass with that message. The target object is usually not the actual object whose state is changing; instead, it's the client object, a document or container that holds many undoable objects. The argument is an object that captures the state of the object before the change is made. Here is a method from the Draw example application (**DrawDocument.m**):

```
- (void)setGridVisible:(NSNumber *)flag {
    BOOL flagValue = [flag boolValue];
    if (gvFlags.showGrid != flagValue) {
        NSNumber *currentValue = [NSNumber numberWithBool:gvFlags.showGrid];
        gvFlags.showGrid = flagValue;
        if (flagValue)
            [graphicView resetGUP];
        [graphicView cache:[graphicView bounds]];
        [undoManager registerUndoWithTarget:self selector:@selector(setGridVisible:
) object:currentValue];
        [undoManager setActionName:GRID_OP];
    }
```

If the user choses Undo, **setGridVisible:** is invoked with the previous value:

Invocation-Based Undo

For other changes involving specific methods or arguments that aren't objects, you can use invocation-based undo, which records an actual message to revert the target object's state. As with simple undo, you record a message that reverts the object to its state before the change. However, in this case you do so by sending the message directly to the NSUndoManager, after preparing it with a special message to note the target:

```
[[myUndoManager prepareWithInvocationTarget:drawObject]
    setFont:[drawObject font] color:[drawObject color]];
[drawObject setFont:newFont color:newColor];
```

The **prepareWithInvocationTarget:** method records the argument as the target of the undo operation about to be established. Following this, you send the message that will revert the target's state—in this case, **setFont:color:**. Because NSUndoManager doesn't respond to this method, **forwardInvocation:** is invoked, which NSUndoManager implements to record the NSInvocation containing the target, selector, and all arguments. Performing undo thus results in **drawObject** being sent a **setFont:color:** message with the old values.

Performing Undo and Redo

Performing undo and redo is usually as simple as sending **undo** and **redo** messages to the NSUndoManager. **undo** closes the last open undo group and then applies all of the undo operations in that group (recording any undo operations as redo operations instead). **redo** likewise applies all of the redo operations on the top redo group.

undo is intended for undoing top-level groups, and shouldn't be used for nested undo groups. If any unclosed, nested undo groups are on the stack when **undo** is invoked, it raises an exception. To undo nested groups, you must use explicitly close the group with an **enableUndoRegistration** message, then use **undoNestedGroup** to undo it. Note also that if you turn off automatic grouping by event with **setGroupsByEvent:**, you must explicitly close the current undo group with **enableUndoRegistration** before invoking either undo method.

Cleaning the Undo Stack

NSUndoManager doesn't retain the targets of undo operations, for several reasons. Foremost is that the client—the object performing undo operations—typically owns the NSUndoManager; thus for the NSUndoManager to retain the target would create cycles. The NSUndoManager does contain references to the targets of undo operations, however, which it uses to send undo messages when undo is performed. If a target object has been deallocated, and an undo message is sent to it, errors will result.

To remedy this, the client must take care to clear undo operations for targets that are being deallocated. This typically occurs in one of three ways:

- The client is the exclusive owner of the NSUndoManager and the target of all undo operations. In this case the client can simply release the NSUndoManager in its **dealloc** method.
- The client shares the NSUndoManager with other clients. To handle this the client should send **removeAllActionsWithTarget:** (argument of **self**) to the NSUndoManager before releasing it in its **dealloc** method.
- The client registers objects other than itself for undo operations. Here either the client must watch for the other objects being deallocated in order to send **removeAllActionsWithTarget:**, or the other objects must do so themselves when deallocated (which requires that they have a reference to the NSUndoManager). This is likely to be needed with invocation-based undo.

In a more general sense, it sometimes makes sense to clear all undo and redo operations. Some applications might want to do this when saving a document, for example. To this end, NSUndoManager defines the **removeAllActions** method, which clears both stacks.

Setting Action Names

NSUndoManager provides methods for qualifying the Undo and Redo command titles in the Edit menu. Take, as an example, a graphics application that allows users to add circles, fill them with a color, and delete

them. With the methods defined here, the application could, upon successive undos, make the titles "Undo Delete," "Undo Fill," and "Undo Add Circle."

These extensions automatically localize Undo and Redo command titles. They also provide two methods that you can override to customize how these titles are localized: **undoMenuTitleForUndoActionName:** and **redoMenuTitleForUndoActionName:**.

Undo Notifications

An NSUndoManager regularly posts *checkpoint notifications* to synchronize the inclusion of undo operations in undo groups. Objects sometimes delay performing changes, for various reasons. This means they may also delay registering undo operations for those changes. Because NSUndoManager collects individual operations into groups, it must be sure to synchronize its client with the creation of these groups so that operations are entered into the proper undo groups. To this end, whenever an NSUndoManager opens or closes a new undo group (except when it opens a top-level group), it posts an “NSUndoManagerCheckpointNotification” so that observers can apply their pending undo operations to the group in effect. The NSUndoManager’s client should register itself as an observer for this notification and record undo operations for all pending changes upon receiving it.

NSUndoManager also posts a number of other notifications at specific intervals: when a group is created, when a group is closed, and just before and just after both undo and redo operations. “”“”

Undo in Applications

The Application Kit supplements the behavior of NSUndoManager in several ways. It offers default undo and redo behavior in text. It includes APIs for managing the action names that appear with "Undo" and "Redo" in an application’s menu. And it establishes a framework for the distribution and selection of NSUndoManagers in an application.

Undo and the Responder Chain

As stated earlier, an application can have one or more clients: objects that register and perform undo operations in their local contexts. Each of these objects has its own NSUndoManger and the associated undo and redo stacks. One example of this scenario involves custom views, each a client of an NSUndoManager. For example, you could have a window with two custom views; each view can display text in changeable attributes (such as font, color, and size) and users can undo (or redo) each change to any attribute in either of the views.NSResponder and NSWindow define methods to help you control the context of undo operations within the view hierarchy.

NSResponder declares the **undoManager** method for most objects that inherit from it (namely, windows and views). When the first responder of an application receives **undo** or **redo** message, NSResponder goes up the responder chain looking for a next responder that implements the **undoManager** by returning an NSUndoManager object. If such an object is returned, it is used for the undo or redo operation. If the **undoManager** message wends its way up the responder chain to the window, the NSWindow object queries

its delegate with **windowWillReturnUndoManager:** to see if the delegater has an NSUndoManager. If the delegate does not implement this method, the NSWindow creates an NSUndoManager for the window and all its views.

Document-based applications often make their NSDocument objects the delegates of their windows and have them respond to the **windowWillReturnUndoManager:** message by returning the NSUndoManager used for the document. These applications can also make each NSWindowController the delegate of its window; the window controller implements **windowWillReturnUndoManager:** to get the NSUndoManager from its document and return it, for example:

```
return [[self document] undoManager];
```

NSTextView

NSTextViews provide undo and redo behavior by default. For your application to take advantage of this feature, however, it must send **setAllowsUndo:** with an argument of YES to the text view. If you want a text view to have its own NSUndoManager (and not the window's), have the text view's delegate implement **undoManagerForTextView:**, to return the NSUndoManager.

The default undo and redo behavior applies to text fields and text in cells as long as the field or cell is the first responder (that is, the focus of keyboard actions). Once the insertion point leaves the field or cell, prior operations cannot be undone.

Method Types

Registering undo operations	– registerUndoWithTarget:selector:object: – prepareWithInvocationTarget: – forwardInvocation:
Checking undo ability	– canUndo – canRedo
Performing undo and redo	– undo – undoNestedGroup – redo
Limiting the undo stack	– setLevelsOfUndo: – levelsOfUndo
Creating undo groups	– beginUndoGrouping – enableUndoRegistration – setGroupsByEvent: – groupingLevel
Disabling undo	– disableUndoRegistration – enableUndoRegistration

Checking whether undo or redo is being performed
– isUndoing
– isRedoing

Clearing undo operations
– removeAllActions
– removeAllActionsWithTarget:

Setting and getting the action name
– setActionName:
– redoActionName
– undoActionName

Getting and localizing menu item title
– redoMenuItemTitle
– undoMenuItemTitle
– redoMenuTitleForUndoActionName:
– undoMenuTitleForUndoActionName:

Instance Methods

beginUndoGrouping

– (void)**beginUndoGrouping**

Marks the beginning of an undo group. All individual undo operations before a subsequent **endUndoGrouping** message are grouped together and reversed by a later **undo** message. By default undo groups are begun automatically at the start of the event loop, but you can begin your own undo groups with this method, and nest them within other groups.

This method posts an "NSUndoManagerCheckpointNotification" unless a top-level undo is in progress. It posts a "NSUndoManagerDidOpenUndoGroupNotification" if a new group was successfully created.

canRedo

– (BOOL)**canRedo**

Returns YES if the receiver has any actions to redo, NO if it doesn't.

Because any undo operation registered clears the redo stack, this method posts an "NSUndoManagerCheckpointNotification" to allow clients to apply their pending operations before testing the redo stack.

See also: – **canUndo**, – **redo**

canUndo

– (BOOL)**canUndo**

Returns YES if the receiver has any actions to undo, NO if it doesn't. This does *not* mean that you can safely invoke **undo** or **undoNestedGroup**; you may have to close open undo groups first.

See also: – **canRedo**, – **enableUndoRegistration**, – **registerUndoWithTarget:selector:object:**

disableUndoRegistration

– (void)**disableUndoRegistration**

Disables the recording of undo operations, whether by **registerUndoWithTarget:selector:object:** or by invocation-based undo. This method can be invoked multiple times by multiple clients; **enableUndoRegistration** must be invoked an equal number of times to re-enable undo registration.

enableUndoRegistration

– (void)**enableUndoRegistration**

Enables the recording of undo operations. Because undo registration is enabled by default, it is often used to balance a prior **disableUndoRegistration** message. Undo registration isn't actually re-enabled until a enable message balances the last disable message in effect. Raises an NSInternalInconsistencyException if invoked while no **disableUndoRegistration** message is in effect.

endUndoGrouping

– (void)**endUndoGrouping**

Marks the end of an undo group. All individual undo operations back to the matching **beginUndoGrouping** message are grouped together and reversed by a later **undo** or **undoNestedGroup** message. Undo groups can be nested, thus providing functionality similar to nested transactions. Raises an NSInternalInconsistencyException if there's no **beginUndoGrouping** message in effect.

This method posts an "NSUndoManagerCheckpointNotification" and an "NSUndoManagerWillCloseUndoGroupNotification" just before the group is closed.

See also: – **levelsOfUndo**

forwardInvocation:

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

Overrides NSObject's implementation to record *anInvocation* as an undo operation. Also clears the redo stack. You can override this method if you want different or supplementary invocation-based behavior. See ""Invocation-Based Undo"" in the class description for more information.

Raises an NSInternalInconsistencyException if **prepareWithInvocationTarget:** wasn't invoked before this method; this method then clears the prepared invocation target. Also raises an NSInternalInconsistencyException if invoked when no undo group has been established using **beginUndoGrouping**. Undo groups are normally set by default, so you should rarely need to begin a top-level undo group explicitly.

See also: – **undoNestedGroup**, – **registerUndoWithTarget:selector:object:**, – **groupingLevel**

groupingLevel

– (int)**groupingLevel**

Returns the number of nested undo groups (or redo groups, if Redo was last invoked) in the current event loop. If zero is returned, there is no open undo or redo group.

See also: – **levelsOfUndo**, – **setLevelsOfUndo:**

groupsByEvent

– (BOOL)**groupsByEvent**

Returns YES if the receiver automatically creates undo groups around each pass of the run loop, NO if it doesn't. The default is YES.

See also: – **beginUndoGrouping**, – **setGroupsByEvent:**

isRedoing

– (BOOL)**isRedoing**

Returns YES if the receiver is in the process of performing its **redo** method, NO otherwise.

See also: – **isUndoing**

isUndoing

– (BOOL)**isUndoing**

Returns YES if the receiver is in the process of performing its **undo** or **undoNestedGroup** method, NO otherwise.

See also: – **isRedoing**

isUndoRegistrationEnabled

– (BOOL)**isUndoRegisgrationEnabled**

Returns whether the recording of undo operations is enabled. Undo registration is enabled by default.

See also: – **disableUndoRegistration**, – **enableUndoRegistration**

levelsOfUndo

– (unsigned int)**levelsOfUndo**

Returns the maximum number of top-level undo groups the receiver will hold. When ending an undo group results in the number of groups exceeding this limit, the oldest groups are dropped from the stack. A limit of zero indicates no limit, so that old undo groups are never dropped. The default is zero.

See also: – **enableUndoRegistration**, – **setLevelsOfUndo:**

prepareWithInvocationTarget:

– (id)**prepareWithInvocationTarget:**(id)*target*

Prepares the receiver for invocation-based undo with *target* as the subject of the next undo operation and returns **self**. See ““Invocation-Based Undo”” in the class description for more information.

See also: – **forwardInvocation:**

redo

– (void)**redo**

Performs the operations in the last group on the redo stack, if there are any, recording them on the undo stack as a single group. Raises an NSInternalInconsistencyException if the method is invoked during an undo operation.

This method posts an “NSUndoManagerCheckpointNotification” and “NSUndoManagerWillRedoChangeNotification” before it performs the redo operation, and it posts the “NSUndoManagerDidRedoChangeNotification” after it performs the redo operation.

See also: – **redo**, – **registerUndoWithTarget:selector:object:**

redoActionName

– (NSString *)**redoActionName**

Returns the name identifying the redo action. For example, if the menu title is "Redo Delete," the string returned is "Delete." Returns an empty string if no action name has been assigned or **nil** if there is nothing to undo.

See also: – **setActionName:**, – **undoActionName**

redoMenuItemTitle

– (NSString *)**redoMenuItemTitle**

Returns the co mplete title of the Redo menu command, for example, "Redo Paste." Returns "Redo" if no action name has been assigned or **nil** if there is nothing to undo.

See also: – **undoMenuItemTitle**

redoMenuTitleForUndoActionName:

– (NSString *)**redoMenuTitleForUndoActionName:**(NSString *)*actionName*

Returns the complete, localized title of the Redo menu command for the action identified by *actionName*. Override this method if you want to customize the localization behavior. This method is invoked by **redoMenuItemTitle**.

See also: – **undoMenuTitleForUndoActionName:**

registerUndoWithTarget:selector:object:

– (void)**registerUndoWithTarget:**(id)*target* **selector:**(SEL)*aSelector* **object:**(id)*anObject*

Records a single undo operation for *target*, so that when undo is performed it’s sent *aSelector* with *anObject* as the sole argument. Also clears the redo stack. Doesn’t retain *target*. See ““Simple Undo”” in the class description for more information.

Raises an NSInternalInconsistencyException if invoked when no undo group has been established using **beginUndoGrouping**. Undo groups are normally set by default, so you should rarely need to begin a top-level undo group explicitly.

See also: – **undoNestedGroup**, – **forwardInvocation:**, – **groupingLevel**

removeAllActions

– (void)**removeAllActions**

Clears the undo and redo stacks and reenables the receiver.

See also: – **enableUndoRegistration**, – **removeAllActionsWithTarget:**

removeAllActionsWithTarget:

– (void)**removeAllActionsWithTarget:**(id)*target*

Clears the undo and redo stacks of all operations involving *target* as the recipient of the undo message. Doesn't re-enable the receiver if it's disabled. An object that shares an NSUndoManager with other clients should invoke this message in its implementation of **dealloc**.

See also: – **enableUndoRegistration**, – **removeAllActions**

runLoopModes

– (NSArray *)**runLoopModes**

Returns the modes determining the types of input handled during a cycle of the run loop. By default, the sole run-loop mode is NSDefaultRunLoopMode (which excludes data from NSConnections).

See also: – **setRunLoopModes:**, – **performSelector:target:argument:order:modes:** (NSRunLoop)

setActionName:

– (void)**setActionName:**(NSString *)*actionName*

Sets the name of the action associated with the Undo or Redo command to *actionName*. If *actionName* is an empty string, the action name currently associated with the menu command is removed. There is no effect if *actionName* is **nil**.

See also: – **redoActionName**, – **undoActionName**

setGroupsByEvent:

– (void)**setGroupsByEvent:**(BOOL)*flag*

Sets whether the receiver automatically groups undo operations during the run loop. If *flag* is YES, the receiver creates undo groups around each pass through the run loop; if *flag* is NO it doesn't. The default is YES.

If you turn automatic grouping off, you must close groups explicitly before invoking either **undo** or **undoNestedGroup**.

See also: – **groupingLevel**, – **groupsByEvent**

setLevelsOfUndo:

– (void)**setLevelsOfUndo:**(unsigned int)*anInt*

Sets the maximum number of top-level undo groups the receiver will hold to *anInt*. When ending an undo group results in the number of groups exceeding this limit, the oldest groups are dropped from the stack. A limit of zero indicates no limit, so that old undo groups are never dropped. The default is zero.

If invoked with a limit below the prior limit, old undo groups are immediately dropped.

See also: – **enableUndoRegistration**, – **levelsOfUndo**

setRunLoopModes:

– (void)**setRunLoopModes:**(NSArray *)*modes*

Sets the modes that determine the types of input handled during a cycle of the run loop. By default, the sole run-loop mode is NSDefaultRunLoopMode (which excludes data from NSConnections). With this method, you could limit the input to data received during a mouse-tracking session by setting the mode to NSEventTrackingRunLoopMode, or you could limit it to data received from a modal panel with NSModalPanelRunLoopMode.

See also: – **runLoopModes**, – **performSelector:target:argument:order:modes:** (NSRunLoop)

undo

– (void)**undo**

Closes the top-level undo group if necessary and invokes **undoNestedGroup**. It also invokes **endUndoGrouping** if the nesting level is 1. Raises an NSInternalInconsistencyException if more than one undo group is open (that is, if the last group isn't at the top level).

This method posts an “NSUndoManagerCheckpointNotification”.

See also: – **enableUndoRegistration**, – **groupingLevel**

undoActionName

– (NSString *)**undoActionName**

Returns the name identifying the undo action. For example, if the menu title is "Undo Delete," the string returned is "Delete." Returns an empty string if no action name has been assigned or **nil** if there is nothing to undo.

See also: – **redoActionName**, – **setActionName:**

undoMenuItemTitle

– (NSString *)**undoMenuItemTitle**

Returns the co mplete title of the Undo menu command, for example, "Undo Paste." If no action name as been assigned, it returns "Undo." Returns "Undo" if no action name has been assigned or **nil** if there is nothing to undo.

See also: – **redoMenuItemTitle**

undoMenuTitleForUndoActionName:

– (NSString *)**undoMenuTitleForUndoActionName:**(NSString *)*actionName*

Returns the complete, localized title of the Undo menu command for the action identified by *actionName*. Override this method if you want to customize the localization behavior. This method is invoked by **undoMenuItemTitle**.

See also: – **redoMenuTitleForUndoActionName:**

undoNestedGroup

– (void)**undoNestedGroup**

Performs the undo operations in the last undo group (whether top-level or nested), recording the operations on the redo stack as a single group. Raises an NSInternalInconsistencyException if any undo operations have been registered since the last **enableUndoRegistration** message.

This method posts an “NSUndoManagerCheckpointNotification” and “NSUndoManagerWillUndoChangeNotification” before it performs the undo operation, and it posts the “NSUndoManagerDidUndoChangeNotification” after it performs the undo operation.

See also: – **undo**

Notifications

NSUndoManagerCheckpointNotification

Posted whenever an NSUndoManager opens or closes an undo group (except when it opens a top-level group), and when checking the redo stack in **canRedo**. The notification contains:

Notification Object The NSUndoManager
Userinfo nil

NSUndoManagerDidOpenUndoGroupNotification

Posted whenever an NSUndoManager opens an undo group, which occurs in the implementation of the **beginUndoGrouping** method. The notification contains:

Notification Object The NSUndoManager
Userinfo nil

NSUndoManagerDidRedoChangeNotification

Posted just after an NSUndoManager performs a redo operation (**redo**). The notification contains:

Notification Object The NSUndoManager
Userinfo nil

NSUndoManagerDidUndoChangeNotification

Posted just after an NSUndoManager performs an undo operation. If you invoke **undo** or **undoNestedGroup**, this notification will be posted. The notification contains:

Notification Object The NSUndoManager
Userinfo nil

NSUndoManagerWillCloseUndoGroupNotification

Posted whenever an NSUndoManager closes an undo group, which occurs in the implementation of the **endUndoGrouping** method.. The notification contains:

Notification Object The NSUndoManager
Userinfonil

NSUndoManagerWillRedoChangeNotification

Posted just before an NSUndoManager performs a redo operation (**redo**).. The notification contains:

Notification Object The NSUndoManager
Userinfonil

NSUndoManagerWillUndoChangeNotification

Posted just after an NSUndoManager performs an undo operation. If you invoke **undo** or **undoNestedGroup**, this notification will be posted. The notification contains:

Notification Object The NSUndoManager
Userinfonil

NSUserDefaults

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSUserDefaults.h

Class at a Glance

Purpose

The NSUserDefaults class provides an programmatic interface for interacting with the OPENSTEP *defaults system*.

Principal Attributes

- A dictionary of defaults.

Creation

+ standardUserDefaults	Returns a shared instance initialized with the current user's defaults.

Commonly Used Methods

– objectForKey:	Returns the default value for the specified key.
– setObject:forKey:	Sets the default value for the specified key.
– removeObjectForKey:	Removes the default entry identified by the specified key.
– registerDefaults:	Adds the specified defaults to the NSRegistrationDomain—a cache of application-provided defaults that are used unless a user overrides them.

Class Description

The NSUserDefaults class provides a programmatic interface for interacting with the OPENSTEP *defaults system*. The defaults system allows an application to customize its behavior to match a user's preferences. For example, you can allow users to determine what units of measurement your application displays or how often documents are automatically saved. Applications record such preferences by assigning values to a set of parameters in a user's defaults database. The parameters are referred to as *defaults* since they're commonly used to determine an application's default state at startup or the way it acts by default.

A defaults database is created automatically for each user. On Unix (Mach, Solaris, and HPUX) platforms, the database is made up of a collection of files located in the **.OpenStep** directory of a user's home directory. On Windows platforms, the defaults database is stored in the Windows registry.

At run time, you use an NSUserDefaults object to read the defaults that your application uses from a user's defaults database. NSUserDefaults caches the information to avoid having to open the user's defaults database each time you need a default value. The **synchronize** method, which is automatically invoked at periodic intervals, keeps the in-memory cache in sync with a user's defaults database.

Warning: User defaults are not thread safe.

Domains

Defaults are grouped in domains. For example, there's a domain for application-specific defaults and another for system-wide defaults that apply to all applications.

Note: All defaults are stored and accessed per user. OpenStep doesn't provide for defaults that affect all users.

Each domain has a name by which it's identified and stores defaults as key-value pairs in an NSDictionary object. Each default is made up of three components:

- The domain in which the default is stored
- The name by which the default is identified (an NSString)
- The default's value, which can be any property-list object (NSData, NSString, NSArray, or NSDictionary)

A domain is either persistent or volatile. Persistent domains are permanent and last past the life of the NSUserDefaults object. Persistent domains are stored in a user's defaults database. If you use NSUserDefaults to make a changes to a default in a persistent domain, the changes are saved in the user's defaults database automatically. On the other hand, volatile domains last only as long as the NSUserDefaults object exists; they aren't saved in the user's defaults database. The standard domains are:

Domain	State
NSArgumentDomain	volatile

Domain	State
Application (Identified by the application's name)	persistent
NSGlobalDomain	persistent
Languages (Identified by the language names)	volatile
NSRegistrationDomain	volatile

A search for the value of a given default proceeds through the domains in an NSUserDefaults object's *search list*. Only domains in the search list are searched. The standard search list contains the domains from the table above, in the order listed. A search ends when the default is found. Thus, if multiple domains contain the same default, only the domain nearest the beginning of the search list provides the default's value. Using the **setSearchList:** method, you can reorder the default search list or set up one that is a subset of all the user's domains.

The following sections describe the purpose of each of the domains.

NSArgumentDomain

Default values can be set from command line arguments (if you start the application from the command line) as well as from a user's *defaults database*. Default values set from the command line go in the NSArgumentDomain. They are set on the command line by preceding the default name with a hyphen and following it with a value. For example, the following command launches Project Builder and sets Project Builder's IndexOnOpen default to NO:

```
localhost> ProjectBuilder.app/ProjectBuilder -IndexOnOpen NO
```

Defaults set from the command line temporarily override values from a user's defaults database. In the example above, Project Builder won't automatically index projects even if the user's IndexOnOpen preference is set to YES in the defaults database.

Application Domain

The application domain contains application-specific defaults that are read from a user's defaults database. The application domain is identified by the name of the application, as returned by NSProcessInfo's **processName** method:

```
NSString *applicationName = [[NSProcessInfo processInfo] processName];
```

NSGlobalDomain

The global domain contains defaults that are read from a user's defaults database and are applicable to all applications that a user runs. Many Application Kit and Foundation objects use default values from the

NSGlobalDomain. For example, NSRulerView objects automatically use a user's preferred measurement units, as stored in the user's defaults database under the key "NSMeasurementUnit." Consequently, ruler views in all applications use the user's preferred measurement units—unless an application overrides the default by creating an NSMeasurementUnit default in its application domain. Another NSGlobalDomain default, NSLanguages, allows users to specify a preference of languages. For example, a user could specify English as the preferred language, followed by Spanish, French, German, Italian, and Swedish.

Languages

If a user has a value for the NSLanguages default, then NSUserDefaults records language-specific default values in domains identified by the language name. The language specific domains contain defaults for a *locale.* Certain classes from the Foundation Framework (NSCalendarDate, NSDate, NSTimeZone, NSString, and NSScanner, for example) use locale defaults to modify their behavior. For example, when you request an NSString representation of an NSCalendarDate, the NSCalendarDate looks at the locale to determine what the months and the days of the week are named in your preferred language. For more information on locale defaults, see the document "Locales" in the *Foundation Reference*.

NSRegistrationDomain

The registration domain is a set of application-provided defaults that are used unless a user overrides them. For example, the first time you run Project Builder, there isn't an IndexOnOpen value saved in your defaults database. Consequently, Project Builder registers a default value for IndexOnOpen in the NSRegistrationDomain as a "catch all" value. Project Builder can thereafter assume that an NSUserDefaults object always has a value to return for the default, simplifying the use of user defaults.

You set NSRegistrationDomain defaults programmatically with the method **registerDefaults:**.

Synchronizing an NSUserDefaults Object with the Defaults Database

Since other applications (and the user) can write to a defaults database, the database and an NSUserDefaults object might not agree on the value of a given default at all times. Using the **synchronize** method, you can update the defaults database with an NSUserDefaults object's new values and update the NSUserDefaults object with any changes that have been made to the database. In applications in which a run-loop is present, **synchronize** is automatically invoked at periodic intervals. Consequently, you might synchronize before exiting a process, but otherwise you shouldn't need to.

Using NSUserDefaults

Typically, you use this class by invoking the **standardUserDefaults** class method to get an NSUserDefaults object. This method returns a global NSUserDefaults object with a search list already initialized. Use the **objectForKey:** and **setObject:forKey:** methods to get and set default values.

For example, suppose that your application needs a default that specifies whether or not to delete backup files. You could use an NSUserDefaults object to manage your default, as follows:

Set a default in the NSRegistrationDomain.

An application can set values for all its defaults in the NSRegistrationDomain. If users specify a different preference in their defaults database, the users' preferences override the values from the NSRegistrationDomain. An NSUserDefaults object only uses values from the NSRegistrationDomain when a user hasn't specified a different preference. So, you need to decide whether or not your application should delete backup files by default.

To register the application's default behavior, you get the application's shared instance of NSUserDefaults and register default values with it. A good place to do this is in the **initialize** method of the class that uses the default. The following example registers the value "YES" for the default named "DeleteBackup".

```
+ (void)initialize{

    NSUserDefaults *defaults = [NSUserDefaults standardUserDefaults];
    NSDictionary *appDefaults = [NSDictionary
        dictionaryWithObject:@"YES" andKey:@"DeleteBackup"];

    [defaults registerDefaults:appDefaults];
}
```

The initialize message is sent to each class before it receives any other message, ensuring that the application's defaults are set before the application needs to read them.

Allow the user to specify a different default behavior.

To allow users to specify a different default behavior for deleting backups, you must provide an interface in which they can express their preference. Most applications provide a Preferences panel for this purpose. When your application detects that a user has specified a new preference, it should save it in the shared instance of NSUserDefaults.

For example, assume that your application has an instance variable called **deleteBackupButton** that is an outlet to an NSButton, and that users toggle this button's state to indicate whether or not the application should delete its backup files. Then you could use the following code to update the user's value for the DeleteBackup default:

```
if ([deleteBackupButton state]) {
    // The user wants to delete backup files.
    [[NSUserDefaults standardUserDefaults]
        setObject:@"YES" forKey:@"DeleteBackup"];
} else {
    // The user doesn't want to delete backup files.
    [[NSUserDefaults standardUserDefaults]
        setObject:@"NO" forKey:@"DeleteBackup"];
}
```

After determining the button's state, **setObject:forKey:** is used to set the value of the specified default in the application domain.

You don’t have to use a Preferences panel to manage all defaults. For example, an NSWindow can store its placement in the user defaults system, so that it appears in the same location the next time the user starts the application.

Use the default value to determine behavior.

To find out whether or not to delete a backup file, you can use the following statement:

```
[[NSUserDefaults standardUserDefaults] boolForKey:@"DeleteBackup"];
```

As a convenience, NSUserDefaults provides **boolForKey:**, **floatForKey:**, and so on. Recall that a default’s value can be only an NSData, NSString, NSArray, or NSDictionary. The **boolForKey:** and similarly named methods attempt to get the value for the specified default and interpret it as a different data type.

Method Types

Getting the shared instance

+ standardUserDefaults

Initializing an NSUserDefaults

– init
– initWithUser:

Getting a default

– arrayForKey:
– boolForKey:
– dataForKey:
– dictionaryForKey:
– floatForKey:
– integerForKey:
– objectForKey:
– stringArrayForKey:
– stringForKey:

Setting and removing defaults

– removeObjectForKey:
– setBool:forKey:
– setFloat:forKey:
– setInteger:forKey:
– setObject:forKey:

Setting and getting the search list

– setSearchList:
– searchList
– dictionaryRepresentation

Maintaining persistent domains
– persistentDomainForName:
– persistentDomainNames
– removePersistentDomainForName:
– setPersistentDomain:forName:
– synchronize

Maintaining volatile domains
– removeVolatileDomainForName:
– setVolatileDomain:forName:
– volatileDomainForName:
– volatileDomainNames

Registering defaults
– registerDefaults:

Class Methods

standardUserDefaults

+ (NSUserDefaults *)**standardUserDefaults**

Returns the shared defaults object. If it doesn't exist yet, it's created with a search list containing the names of the following domains, in this order:

- NSArgumentDomain, consisting of defaults parsed from the application's arguments
- A domain identified by the process (application) name
- NSGlobalDomain, consisting of defaults meant to be seen by all applications
- Separate domains for each of the user's preferred languages
- NSRegistrationDomain, a set of temporary defaults whose values can be set by the application to ensure that searches will always be successful

The defaults are initialized for the current user. Subsequent modifications to the standard search list remain in effect even when this method is invoked again—the search list is guaranteed to be standard only the first time this method is invoked. The shared instance is provided as a convenience; you may create custom instances with **initWithUser:** and **init**.

See also: **– init**, **– initWithUser:**

Instance Methods

arrayForKey:

– (NSArray *)**arrayForKey:**(NSString *)*defaultName*

Invokes **objectForKey:** with key *defaultName*. Returns the value associated with *defaultName* if it's an NSArray object and **nil** otherwise.

See also: –**boolForKey:**, – **dataForKey:**, – **dictionaryForKey:**, – **floatForKey:**, – **integerForKey:**, – **objectForKey:**, **– stringArrayForKey:**, **– stringForKey:**

boolForKey:

– (BOOL)**boolForKey:**(NSString *)*defaultName*

Invokes **stringForKey**: with key *defaultName*. Returns YES if the value associated with *defaultName* is an NSString containing uppercase or lowercase "YES" or responds to the **intValue** message by returning a non-zero value. Otherwise, returns NO.

See also: –**arrayForKey:**, – **dataForKey:**, – **dictionaryForKey:**, – **floatForKey:**, – **integerForKey:**, – **objectForKey:**, **– stringArrayForKey:**, **– stringForKey:**

dataForKey:

– (NSData *)**dataForKey:**(NSString *)*defaultName*

Invokes **objectForKey:** with key *defaultName*. Returns the corresponding value if it's an NSData object and nil otherwise.

See also: – **arrayForKey:**, –**boolForKey:**, – **dictionaryForKey:**, – **floatForKey:**, – **integerForKey:**, – **objectForKey:**, **– stringArrayForKey:**, **– stringForKey:**

dictionaryForKey:

– (NSDictionary *)**dictionaryForKey:**(NSString *)*defaultName*

Invokes **objectForKey:** with key *defaultName*. Returns the corresponding value if it's an NSDictionary object and **nil** otherwise.

See also: – **arrayForKey:**, **–boolForKey:, – dataForKey:**, – **floatForKey:**, – **integerForKey:**, – **objectForKey:**, **– stringArrayForKey:**, **– stringForKey:**

dictionaryRepresentation

– (NSDictionary *)**dictionaryRepresentation**

Returns a dictionary that contains a union of all key-value pairs in the domains in the search list. As with **objectForKey:**, key-value pairs in domains that are earlier in the search list take precedence. The combined result doesn't preserve information about which domain each entry came from.

See also: – **searchList**

floatForKey:

– (float)**floatForKey:**(NSString *)*defaultName*

Invokes **stringForKey:** with key *defaultName*. Returns 0 if no string is returned. Otherwise, the resulting string is sent a **floatValue** message, which provides this method's return value.

See also: – **arrayForKey:**, **–boolForKey:, – dataForKey:**, – **dictionaryForKey:**, – **integerForKey:**, – **objectForKey:**, **– stringArrayForKey:**, **– stringForKey:**

init

– (id)**init**

Initializes defaults for the current user account and returns an NSUserDefaults instance with the argument and registration domains set up. This method doesn't put anything in the search list. Invoke it only if you've allocated your own NSUserDefaults instance instead of using the shared one.

See also: + **standardUserDefaults**

initWithUser:

– (id)**initWithUser:**(NSString *)*username*

Initializes defaults for the user account identified by *username* and returns an NSUserDefaults instance with the argument and registration domains set up. This method doesn't put anything in the search list. Invoke it only if you've allocated your own NSUserDefaults instance instead of using the shared one.

You wouldn't ordinarily use this method to initialize an instance of NSUserDefaults. It is provided for applications intended for use by a superuser who needs to update defaults databases for a number of users. The user who started the application must have appropriate access (read, write, or both) to the defaults database of the new user, or this method returns **nil**.

See also: + **standardUserDefaults**

integerForKey:

– (int)**integerForKey:**(NSString *)*defaultName*

Invokes **stringForKey:** with key *defaultName*. Returns 0 if no string is returned. Otherwise, the resulting string is sent a **intValue** message, which provides this method's return value.

See also: – **arrayForKey:**, **–boolForKey:, – dataForKey:**, – **dictionaryForKey:**, – **floatForKey:**, – **objectForKe**y:, **– stringArrayForKey:**, **– stringForKey:**

objectForKey:

– (id)**objectForKey:**(NSString *)*defaultName*

Returns the value of the first occurrence of the default identified by *defaultName*, searching the domains included in the search list in the order they're listed. Returns **nil** if the default isn't found.

See also: – **arrayForKey:**, **–boolForKey:, – dataForKey:**, – **dictionaryForKey:**, – **floatForKey:**, – **integerForKey:**, **– stringArrayForKey:**, **– stringForKey:**

persistentDomainForName:

– (NSDictionary *)**persistentDomainForName:**(NSString *)*domainName*

Returns a dictionary representing the persistent domain identified by *domainName*. The keys in the dictionary are names of defaults, and the value corresponding to each key is a property-list object (NSString, NSDictionary, NSArray, or NSData).

See also: – **removePersistentDomainForName:**, – **setPersistentDomain:forName:**

persistentDomainNames

– (NSArray *)**persistentDomainNames**

Returns an array containing the names of the current persistent domains. You can get each domain by using the domain names in the array as arguments to **persistentDomainForName:**.

See also: – **removePersistentDomainForName:**, – **setPersistentDomain:forName:**

registerDefaults:

– (void)**registerDefaults:**(NSDictionary *)*dictionary*

Adds the contents of *dictionary* to the registration domain. If there is no registration domain, it's created using dictionary, and NSRegistrationDomain is added to the end of the search list

removeObjectForKey:

– (void)**removeObjectForKey:**(NSString *)*defaultName*

Removes the value for the default identified by *defaultName* in the standard application domain. Removing a default has no effect on the value returned by the **objectForKey:** method if the same key exists in a domain that precedes the standard application domain in the search list.

See also: – setObject:forKey:

removePersistentDomainForName:

– (void)**removePersistentDomainForName:**(NSString *)*domainName*

Removes the persistent domain identified by *domainName* from the user's defaults. The first time that a persistent domain is changed after **synchronize**, an NSUserDefaultsChanged notification is posted.

See also: – **setPersistentDomain:forName:**

removeVolatileDomainForName:

– (void)**removeVolatileDomainForName:**(NSString *)*domainName*

Removes the volatile domain identified by *domainName* from the user's defaults.

See also: – **setVolatileDomain:forName:**

searchList

– (NSArray *)**searchList**

Returns an array of domain names, identifying the domains that **objectForKey:** will search.

See also: – **setSearchList:**, – **dictionaryRepresentation**

setBool:forKey:

– (void)**setBool:**(BOOL)*value* **forKey:**(NSString *)*defaultName*

Sets the value of the default identified by *defaultName* to a string representation of YES or NO, depending on value. Invokes **setObject:forKey:** as part of its implementation.

See also: – **boolForKey:**

setFloat:forKey:

– (void)**setFloat:**(float)*value* **forKey:**(NSString *)*defaultName*

Sets the value of the default identified by *defaultName* to a string representation of value. Invokes **setObject:forKey:** as part of its implementation.

See also: – **floatForKey:**

setInteger:forKey:

– (void)**setInteger:**(int)*value* **forKey:**(NSString *)*defaultName*

Sets the value of the default identified by *defaultName* to a string representation of value. Invokes **setObject: forKey:** as part of its implementation.

See also: – **integerForKey:**

setObject:forKey:

– (void)**setObject:**(id)*value* **forKey:**(NSString *)*defaultName*

Sets the value of the default identified by *defaultName* in the standard application domain. Setting a default has no effect on the value returned by the **objectForKey:** method if the same key exists in a domain that precedes the application domain in the search list.

See also: – **objectForKey:**, – **removeObjectForKey:**

setPersistentDomain:forName:

– (void)**setPersistentDomain:**(NSDictionary *)*domain* **forName:**(NSString *)*domainName*

Sets the dictionary for the persistent domain named *domainName*; raises an NSInvalidArgumentException if a persistent domain with *domainName* already exists. The first time that a persistent domain is changed after synchronize, an NSUserDefaultsChanged notification is posted.

See also: – **persistentDomainForName:**, – **persistentDomainNames**

setSearchList:

– (void)**setSearchList:**(NSArray *)*array*

Sets the domains that **objectForKey:** will search. Domain names in *array* with no corresponding user default domain are ignored.

See also: – **searchList**

setVolatileDomain:forName:

– (void)**setVolatileDomain:**(NSDictionary *)*domain* **forName:**(NSString *)*domainName*

Sets the dictionary to *domain* for the volatile domain named *domainName*. This method raises an NSInvalidArgumentException if a volatile domain with *domainName* already exists.

See also: – **volatileDomainForName:**, – **volatileDomainNames**

stringArrayForKey:

– (NSArray *)**stringArrayForKey:**(NSString *)*defaultName*

Invokes **objectForKey:** with key *defaultName*. Returns the corresponding value if it's an NSArray object containing NSStrings, and **nil** otherwise.

See also: – **arrayForKey:**, **–boolForKey:, – dataForKey:**, – **dictionaryForKey:**, – **floatForKey:**, – **integerForKey:**, – **objectForKey:**, **– stringForKey:**

stringForKey:

– (NSString *)**stringForKey:**(NSString *)*defaultName*

Invokes **objectForKey:** with key *defaultName*. Returns the corresponding value if it's an NSString object and **nil** otherwise.

See also: – **arrayForKey:**, **–boolForKey:, – dataForKey:**, – **dictionaryForKey:**, – **floatForKey:**, – **integerForKey:**, – **objectForKey:**, **– stringArrayForKey:**

synchronize

– (BOOL)**synchronize**

Saves any modifications to the persistent domains and updates all persistent domains that were not modified to what is on disk. Returns NO if it could not save data to disk. Since the **synchronize** method is automatically invoked at periodic intervals, use this method only if you cannot wait for the automatic synchronization (for example if your application is about to exit), or if you want to update user defaults to what is on disk even though you have not made any changes.

See also: – **persistentDomainForName:**, – **persistentDomainName**s, – **removePersistentDomainForName:**, – **setPersistentDomain:forName:**

volatileDomainForName:

– (NSDictionary *)**volatileDomainForName:**(NSString *)*domainName*

Returns a dictionary representing the volatile domain identified by *domainName*. The keys in the dictionary are names of defaults, and the value corresponding to each key is a property-list object (NSString, NSData, NSDictionary, NSArray).

See also: – **removeVolatileDomainForName:**, – **setVolatileDomain:forName:**

volatileDomainNames

– (NSArray *)**volatileDomainNames**

Returns an array containing the names of the current volatile domains. You can get each domain by using the domain names in the array as arguments to **volatileDomainForName:**.

See also: – **removeVolatileDomainForName:**, – **setVolatileDomain:forName:**

Notifications

NSUserDefaultsDidChangeNotification

This notification contains a notification object but no userInfo dictionary. The notification object is the NSUserDefaults instance.

This notification is posted the first time after a **synchronize** when a change is made to defaults in a persistent domain.

NSValue

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSObject (NSObject)
Declared In:	Foundation/NSValue.h Foundation/NSGeometry.h

Class at a Glance

Purpose

An NSValue object serves as an object wrapper for a standard C or Objective-C data item, allowing it to be stored in a collection object such as an NSArray or NSDictionary.

Creation

+ value:withObjCType:	Returns an NSValue containing any C or Objective-C data item.
+ valueWithBytes:objCType:	Returns an NSValue containing any Objective-C data item, which is interpreted as being of the specified Objective-C type.
+ valueWithNonretainedObject:	Returns an NSValue containing an Objective-C object, without retaining the Objective-C object.
+ valueWithPointer:	Returns an NSValue that contains a pointer.

Commonly Used Methods

– objCType	Returns the Objective-C type for the data contained in an NSValue.
– getValue:	Copies an NSValue's contents into a buffer.
– nonretainedObjectValue	Returns an NSValue's contents as an **id**.
– pointerValue	Returns an NSValue's contents as a pointer to **void**.

Primitive Methods

– getValue:
– objCType

Class Description

An NSValue object is a simple container for a single C or Objective-C data item. It can hold any of the scalar types such as **int**, **float**, and **char**, as well as pointers, structures, and object **id**s. The purpose of this class is

to allow items of such data types to be added to collection objects such as NSArrays and NSSets, which require their elements to be objects. NSValue objects are always immutable.

To create an NSValue object with a particular data item, you provide a pointer to the item along with a C string describing the item's type in Objective-C type encoding. You get this string using the **@encode()** compiler directive, which returns the platform-specific encoding for the given type (**See the section "Types Encoding" in the chapter "Objective-C Extensions" in the book *Object-Oriented Programming and the Objective-C Language* for more information about @encode() and a list of type codes**). Fore example, this code excerpt creates **theValue** containing an NSRange:

```
NSRange myRange = {4, 10};
NSValue *theValue = [NSValue valueWithBytes:&myRange objCType:@encode(NSRange)];
```

Note that the type you specify must be of constant length. C strings, variable-length arrays and structures, and other data types of indeterminate length can't be stored in an NSValue. You should use NSString or NSData objects for these. If you must store a variable-length item in an NSValue, you have to store a pointer to the item, not the item itself. This code excerpt incorrectly attempts to place a C string directly into an NSValue object:

```
/* INCORRECT! */
char *myCString = "This is a string.";
NSValue *theValue = [NSValue value:myCString withObjCType:@encode(char *)];
```

In this code excerpt the *contents* of **myCString** are interpreted as a pointer to a **char**, so that the first four bytes contained in the string are treated as a pointer (the actual number of bytes used may vary with the hardware architecture). That is, the sequence "This" is interpreted as a pointer value, which is unlikely to be a legal address. The correct way to store such a data item, short of using an NSString object, is to pass the address of its pointer, not the pointer itself:

```
/* Correct. */
char *myCString = "This is a string.";
NSValue *theValue = [NSValue value:&myCString withObjCType:@encode(char *)];
```

Here the address of **myCString** is passed, so that the address of the first character of the string is stored in **theValue**. Note that the NSValue doesn't copy the contents of the string, but the pointer itself. If you create an NSValue with an allocated data item, don't deallocate its memory while the NSValue object exists.

Adopted Protocols

NSCoding

– encodeWithCoder:
– initWithCoder:

NSCopying

– copyWithZone:

Method Types

Creating an NSValue

– initWithBytes:objCType:
+ valueWithBytes:objCType:
+ value:withObjCType:
+ valueWithNonretainedObject:
+ valueWithPointer:
+ valueWithPoint:
+ valueWithRect:
+ valueWithSize:

Accessing data

– getValue:
– nonretainedObjectValue
– objCType
– pointValue
– pointerValue
– rectValue
– sizeValue

Comparing objects

– isEqualToValue:

Class Methods

value:withObjCType:

+ (NSValue *)**value:**(const void *)*value* **withObjCType:**(const char *)*type*

Creates and returns an NSValue containing *value*, which is interpreted as being of the Objective-C type *type*. *type* should be created with the Objective-C **@encode()** compiler directive; it shouldn't be hard-coded as a C string. See the class description for other considerations in creating an NSValue object and code examples.

See also: **+ valueWithBytes:objCType:**

valueWithBytes:objCType:

+ (NSValue *)**valueWithBytes:**(const void *)*value* **objCType:**(const char *)*type*

Creates and returns an NSValue containing *value*, which is interpreted as being of the Objective-C type *type*. *type* should be created with the Objective-C **@encode()** compiler directive; it shouldn't be hard-coded as a

C string. This method is equivalent to **value:withObjCType:**, which is part of OpenStep. See the class description for other considerations in creating an NSValue object and code examples.

See also: – **initWithBytes:objCType:**

valueWithNonretainedObject:

+ (NSValue *)**valueWithNonretainedObject:**(id)*anObject*

Creates and returns an NSValue containing *anObject*, but doesn't retain it. This method is equivalent to invoking **value:withObjCType:** in this manner:

```
NSValue *theValue = [NSValue value:&anObject withObjCType:@encode(void *)];
```

This method is useful for preventing an object from being retained when it's added to a collection object (such as an NSArray or NSDictionary).

See also: – **nonretainedObjectValue**

valueWithPoint:

+ (NSValue *)**valueWithPoint:**(NSPoint)*aPoint*

Creates and returns an NSValue object that contains the specified NSPoint structure (which represents a geometrical point in two dimensions).

See also: – **pointValue**

valueWithPointer:

+ (NSValue *)**valueWithPointer:**(const void *)*aPointer*

Creates and returns an NSValue object that contains *aPointer*. This method is equivalent to invoking **value: withObjCType:** in this manner:

```
NSValue *theValue = [NSValue value:&aPointer withObjCType:@encode(void *)];
```

This method doesn't copy the contents of *aPointer*, so you should be sure not to deallocate that memory while the NSValue object exists. NSData objects may be more suited for arbitrary pointers than NSValue objects.

See also: – **pointerValue**

valueWithRect:

+ (NSValue *)**valueWithRect:**(NSRect)*rect*

Creates and returns an NSValue object that contains the specified NSRect structure (which represents the coordinates of the rectangle's origin).

See also: – **rectValue**

valueWithSize:

+ (NSValue *)**valueWithSize:**(NSSize)*size*

Creates and returns an NSValue that contains the specified NSSize structure (which represents the width and height of a rectangle).

See also: – **sizeValue**

Instance Methods

getValue:

– (void)**getValue:**(void *)*buffer*

Copies the NSValue's contents into *buffer*. *buffer* should be large enough to hold the value.

initWithBytes:objCType:

– (id)**initWithBytes:**(const void *)*value* **objCType:**(const char *)*type*

Initializes a newly created NSValue to contain *value*, which is interpreted as being of the Objective-C type *type*. *type* should be created with the Objective-C **@encode()** compiler directive; it shouldn't be hard-coded as a C string. See the class description for other considerations in creating an NSValue object.

This is the designated initializer for the NSValue class. Returns **self**.

isEqualToValue:

– **isEqualToValue:**(NSValue *)*aValue*

Returns YES if the receiver and *aValue* are equal, otherwise returns NO. For NSValue objects, the class, type, and contents are compared to determine equality.

nonretainedObjectValue

– (id)**nonretainedObjectValue**

For an NSValue object created to hold a pointer-sized data item, returns that item as an **id.** For any other NSValue object the result is undefined.

See also: – **getValue:**

objCType

– (const char *)**objCType**

Returns a C string containing the Objective-C type of the data contained in the receiver, as encoded by the **@encode()** compiler directive**.**

pointValue

– (NSPoint)**pointValue**

Returns an NSPoint structure (which represents a geometrical point in two dimensions).

See also: – **rectValue,** – **sizeValue**

pointerValue

– (void *)**pointerValue**

For an NSValue object created to hold a pointer-sized data item, returns that item as a pointer to **void**. For any other NSValue object the result is undefined.

See also: – **getValue:**

rectValue

– (NSRect)**rectValue**

Returns an NSRect structure (which represents the coordinates of the rectangle’s origin).

See also: – **pointValue,** – **sizeValue**

sizeValue

– **(NSSize)sizeValue**

Returns an NSSize structure (which represents the width and height of a rectangle).

See also: – pointValue, – rectValue

Functions and Macros

This section describes the functions and macros found in the Foundation Kit.

NSAllHashTableObjects

SUMMARY This function returns all of the elements in the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS NSArray ***NSAllHashTableObjects**(NSHashTable **table*)

DESCRIPTION Returns an array object containing all the elements of *table*. This function should be called only when the table elements are objects, not when they're any other data type.

SEE ALSO **NSCreateHashTable, NSFreeHashTable**

NSAllMapTableKeys

SUMMARY This function returns all of the keys in the specified map table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSArray ***NSAllMapTableKeys**(NSMapTable **table*)

DESCRIPTION **NSAllMapTableKeys** Returns an array object containing all the keys in *table*. This function should be called only when the table keys are objects, not when they're any other type of pointer.

SEE ALSO **NSMapMember, NSMapGet, NSEnumerateMapTable, NSNextMapEnumeratorPair, NSAllMapTableValues**

NSAllMapTableValues

SUMMARY This function returns all of the values in the specified table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSArray ***NSAllMapTableValues**(NSMapTable **table*)

DESCRIPTION **NSAllMapTableValues** Returns an array object containing all the values in *table*. This function should be called only when the table values are objects, not when they're any other type of pointer.

SEE ALSO **NSMapMember, NSMapGet, NSEnumerateMapTable, NSNextMapEnumeratorPair, NSAllMapTableKeys**

NSAllocateMemoryPages

SUMMARY This function allocates a new block of memory.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void ***NSAllocateMemoryPages**(unsigned int *byteCount*)

DESCRIPTION **NSAllocateMemoryPages** allocates the integral number of pages whose total size is closest to, but not less than, *byteCount*. The allocated pages are guaranteed to be filled with zeros.

SEE ALSO **NSCopyMemoryPages, NSDeallocateMemoryPages**

NSAllocateObject

SUMMARY This function creates new objects.

DECLARED IN Foundation/NSObject.h

SYNOPSIS id <NSObject> **NSAllocateObject(**Class *aClass*, unsigned int *extraBytes*, NSZone **zone***)**

DESCRIPTION NSAllocateObject allocates and returns a pointer to an instance of *aClass*, created in the specified *zone* (or in the default zone, if *zone* is NULL). The *extraBytes* argument (usually zero) states the number of extra bytes required for indexed instance variables. Returns **nil** on failure.

SEE ALSO **NSCopyObject, NSDeallocateObject**

NSAssert

SUMMARY This macro generates an assertion if the given condition is false.

DECLARED IN Foundation/NSExceptions.h

SYNOPSIS **NSAssert(***condition***,** NSString *****description***)**

DESCRIPTION Assertions evaluate a condition and, if the condition evaluates to false, call the assertion handler for the current thread, passing it a format string and a variable number of arguments. Each thread has its own assertion handler, which is an object of class NSAssertionHandler. When invoked, an assertion handler prints an error message that includes the method and class names (or the function name). It then raises an exception of type NSInternalInconsistencyException.

The **NSAssert** macro evaluates the condition and serves as a front end to the assertion handler. This macro should be used only within Objective-C methods. **NSAssert** takes no arguments other than the condition and format string.

condition must be an expression that evaluates to true or false. *description* is a **printf**-style format string that contains the error message describing the failure condition.

Assertions are disabled if the preprocessor macro **NS_BLOCK_ASSERTIONS** is defined. All macros return **void**.

SEE ALSO **NSLog, NSLogv, NSAssert1, NSCAssert, NSCParameterAssert, NSParameterAssert**

NSAssert1

SUMMARY NSAssert1 is one of a series of macros, as listed in the SYNOPSIS section, that generate assertions if the given condition is false.

DECLARED IN Foundation/NSExceptions.h

SYNOPSIS
NSAssert1(*condition***,** NSString ******description, arg1* **)**
NSAssert2(*condition***,** NSString ******description, arg1, arg2***)**
NSAssert3(*condition***,** NSString ******description, arg1***,** *arg2***,** *arg3***)**
NSAssert4(*condition***,** NSString ******description, arg1, arg2***,** *arg3***,** *arg4***)**
NSAssert5(*condition***,** NSString ******description, arg1***,** *arg2***,** *arg3***,** *arg4***,** *arg5***)**

DESCRIPTION Assertions evaluate a condition and, if the condition evaluates to false, call the assertion handler for the current thread, passing it a format string and a variable number of arguments. Each thread has its own assertion handler, which is an object of class NSAssertionHandler. When invoked, an assertion handler prints an error message that includes the method and class names (or the function name). It then raises an exception of type NSInternalInconsistencyException.

The **NSAssert*n*** macros evaluate the condition and serve as a front end to the assertion handler. These macros should be used only within Objective-C methods. **NSAssert*n*** takes the number of format-string arguments indicated by *n*.

condition must be an expression that evaluates to true or false. *description* is a **printf**-style format string that contains the error message describing the failure condition. Each *arg* parameter is an argument to be inserted, in place, into the *description*.

Assertions are disabled if the preprocessor macro **NS_BLOCK_ASSERTIONS** is defined. All macros return **void**.

SEE ALSO **NSLog, NSLogv, NSAssert, NSCAssert, NSCParameterAssert, NSParameterAssert**

NSAssert2

SUMMARY See **NSAssert1**.

NSAssert3

SUMMARY See **NSAssert1**.

NSAssert4

SUMMARY See **NSAssert1**.

NSAssert5

SUMMARY See **NSAssert1**.

NSCAssert

SUMMARY This macro generates an assertion if the given condition is false.

DECLARED IN Foundation/NSExceptions.h

SYNOPSIS **NSCAssert(***condition***,** NSString ***description***)**

DESCRIPTION Assertions evaluate a condition and, if the condition evaluates to false, call the assertion handler for the current thread, passing it a format string and a variable number of arguments. Each thread has its own assertion handler, which is an object of class NSAssertionHandler. When invoked, an

assertion handler prints an error message that includes method and class names (or the function name). It then raises an exception of type NSInternalInconsistencyException.

The **NSCAssert** macro evaluates the condition and serves as a front end to the assertion handler. This macro should be used only within C functions. **NSCAssert** takes no arguments other than the condition and format string.

condition must be an expression that evaluates to true or false. *description* is a **printf**-style format string that describes the failure condition.

Assertions are disabled if the preprocessor macro **NS_BLOCK_ASSERTIONS** is defined. All macros return **void**.

SEE ALSO **NSLog, NSLogv, NSAssert, NSCAssert1, NSCParameterAssert, NSParameterAssert**

NSCAssert1

SUMMARY NSCAssert1 is one of a series of macros, as listed in the SYNOPSIS section, that generate assertions if the given condition is false.

DECLARED IN Foundation/NSExceptions.h

SYNOPSIS
NSCAssert1(*condition***,** NSString ***description, arg1***)**
NSCAssert2(*condition***,** NSString ***description, arg1, arg2***)**
NSCAssert3(*condition***,** NSString ***description, arg1***, *arg2*, *arg3***)**
NSCAssert4(*condition***,** NSString ***description, arg1, arg2***, *arg3*, *arg4***)**
NSCAssert5(*condition***,** NSString ***description, arg1***, *arg2*, *arg3*, *arg4*, *arg5***)**

DESCRIPTION Assertions evaluate a condition and, if the condition evaluates to false, call the assertion handler for the current thread, passing it a format string and a variable number of arguments. Each thread has its own assertion handler, which is an object of class NSAssertionHandler. When invoked, an assertion handler prints an error message that includes method and class names (or the function name). It then raises an exception of type NSInternalInconsistencyException.

The **NSCAssert*n*** macros evaluate the condition and serve as a front end to the assertion handler. These macros should be used only within Objective-C methods. **NSCAssert*n*** takes the number of format-string arguments indicated by *n*.

condition must be an expression that evaluates to true or false. *description* is a **printf**-style format string that describes the failure condition. Each *arg* is an argument to be inserted, in place, into the *description*.

Assertions are disabled if the preprocessor macro **NS_BLOCK_ASSERTIONS** is defined. All macros return **void**.

SEE ALSO **NSLog, NSLogv, NSCAssert, NSCParameterAssert, NSParameterAssert**

NSCAssert2

SUMMARY See **NSCAssert1**.

NSCAssert3

SUMMARY See **NSCAssert1**.

NSCAssert4

SUMMARY See **NSCAssert1**.

NSCAssert5

SUMMARY See **NSCAssert1**.

NSClassFromString

SUMMARY This function obtains a class by name.

DECLARED IN Foundation/NSObjCRuntime.h

SYNOPSIS Class **NSClassFromString**(NSString ***aClassName*)

DESCRIPTION **NSClassFromString** returns the class object named by *aClassName*, or **nil** if no class by that name is currently loaded.

SEE ALSO **NSStringFromClass**

NSCompareHashTables

SUMMARY This function compares the elements of two hash tables for equality.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS BOOL **NSCompareHashTables**(NSHashTable ***table1*, NSHashTable ***table2*)

DESCRIPTION Returns YES if the two hash tables are equal—that is, if each element of *table1* is in *table2*, and the two tables are the same size.

SEE ALSO **NSCreateHashTable, NSCreateHashTableWithZone**

NSCompareMapTables

SUMMARY This function compares the elements of two map tables for equality.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS BOOL **NSCompareMapTables**(NSMapTable ***table1*, NSMapTable ***table2*)

DESCRIPTION Returns YES if each key of *table1* is in *table2*, and the two tables are the same size. Note that this function does not compare values, only keys.

SEE ALSO **NSCreateMapTable, NSCreateMapTableWithZone**

NSContainsRect

SUMMARY This function determines whether one rectangle completely encloses another.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSContainsRect**(NSRect *aRect*, NSRect *bRect*)

DESCRIPTION Returns YES if *aRect* completely encloses *bRect*. For this condition to be true, *bRect* cannot be empty and its sides must not touch the sides of *aRect*.

NSConvertHostDoubleToSwapped

SUMMARY This function performs a type conversion.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedDouble **NSConvertHostDoubleToSwapped**(double *x*)

DESCRIPTION Converts the double value in *x* to a value whose bytes can be swapped. This function does not actually swap the bytes of x. You should not need to call this method directly.

SEE ALSO **NSSwapHostDoubleToBig, NSSwapHostDoubleToLittle**

NSConvertHostFloatToSwapped

SUMMARY This function performs a type conversion.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedFloat **NSConvertHostFloatToSwapped**(float *x*)

DESCRIPTION Converts the float value in *x* to a value whose bytes can be swapped. This function does not actually swap the bytes of x. You should not need to call this method directly.

SEE ALSO **NSSwapHostFloatToBig, NSSwapHostFloatToLittle**

NSConvertSwappedDoubleToHost

SUMMARY This function performs a type conversion.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS double **NSConvertSwappedDoubleToHost**(NSSwappedDouble *x*)

DESCRIPTION Converts the value in *x* to a double value. This function does not actually swap the bytes of x. You should not need to call this method directly.

SEE ALSO **NSSwapBigDoubleToHost, NSSwapLittleDoubleToHost**

NSConvertSwappedFloatToHost

SUMMARY This function performs a type conversion.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS float **NSConvertSwappedFloatToHost**(NSSwappedFloat *x*)

DESCRIPTION Converts the value in *x* to a float value. This function does not actually swap the bytes of x. You should not need to call this method directly.

SEE ALSO **NSSwapBigFloatToHost, NSSwapLittleFloatToHost**

NSCopyHashTableWithZone

SUMMARY This function performs a shallow copy of the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS NSHashTable ***NSCopyHashTableWithZone**(NSHashTable **table*, NSZone **zone*)

DESCRIPTION Returns a pointer to a new copy of *table*, created in *zone* and containing pointers to the data elements of *table*. If *zone* is NULL, the new table is created in the default zone.

The new table adopts the callback functions of table and calls the **hash** and **retain** callback functions as appropriate when inserting elements into the new table.

SEE ALSO **NSCreateHashTable, NSCreateHashTableWithZone, NSHashTableCallBacks (structure)**

NSCopyMapTableWithZone

SUMMARY This function performs a shallow copy of the specified map table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSMapTable ***NSCopyMapTableWithZone**(NSMapTable **table*, NSZone **zone*)

DESCRIPTION Returns a pointer to a new copy of *table*, created in *zone* and containing pointers to the keys and values of *table*. If *zone* is NULL, the new table is created in the default zone.

The new table adopts the callback functions of table and calls the **hash** and **retain** callback functions as appropriate when inserting elements into the new table.

SEE ALSO **NSCreateMapTable, NSCreateMapTableWithZone, NSMapTableKeyCallBacks (structure), NSMapTableValueCallBacks (structure)**

NSCopyMemoryPages

SUMMARY This function copies a block of memory.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void **NSCopyMemoryPages**(const void ***source*, void ***destination*, unsigned int *byteCount*)

DESCRIPTION Copies (or copies-on-write) *byteCount* bytes from *source* to *destination*.

SEE ALSO **NSAllocateMemoryPages, NSDeallocateMemoryPages**

NSCopyObject

SUMMARY This function creates exact copies of objects.

DECLARED IN Foundation/NSObject.h

SYNOPSIS id <NSObject> **NSCopyObject(**id <NSObject> *anObject*, unsigned int *extraBytes*, NSZone ***zone***)**

DESCRIPTION Creates and returns a new object that's an exact copy of *anObject*, created in the specified *zone* (or in the default zone, if *zone* is NULL). The *extraBytes* argument (usually zero) states the number of extra bytes required for indexed instance variables. Returns **nil** if *anObject* is **nil** or if *anObject* could not be copied.

SEE ALSO **NSAllocateObject, NSDeallocateObject**

NSCountHashTable

SUMMARY This function returns the number of elements in a hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS unsigned int **NSCountHashTable**(NSHashTable *_table_)

DESCRIPTION Returns the number of elements currently in *table*.

NSCountMapTable

SUMMARY This function returns the number of elements in a map table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS unsigned int **NSCountMapTable**(NSMapTable *_table_)

DESCRIPTION Returns the number of key/value pairs currently in *table*.

NSCParameterAssert

SUMMARY This macro evaluates the specified parameter.

DECLARED IN Foundation/NSExceptions.h

SYNOPSIS **NSCParameterAssert(**_condition_**)**

DESCRIPTION Assertions evaluate a condition and, if the condition evaluates to false, call the assertion handler for the current thread, passing it a format string and a variable number of arguments. Each thread has its own assertion handler, which is an object of class NSAssertionHandler. When invoked, an assertion handler prints an error message that includes method and class names (or the function name). It then raises an exception of type NSInternalInconsistencyException.

This macro validates a parameter for a C-function. Simply provide the parameter as the *condition* argument. The macro evaluates the parameter and, if the parameter evaluates to false, logs an error message which includes the parameter and then raises an exception.

Assertions are disabled if the preprocessor macro **NS_BLOCK_ASSERTIONS** is defined. All macros return **void**.

SEE ALSO **NSLog, NSLogv, NSAssert, NSCAssert, NSParameterAssert**

NSCreateHashTable

SUMMARY This function creates a new hash table in the default zone.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS NSHashTable *__NSCreateHashTable__(NSHashTableCallBacks *callBacks*, unsigned int *capacity*)

DESCRIPTION **NSCreateHashTable** creates and returns a pointer to an NSHashTable in the default zone.

The table's size is dependent on (but generally not equal to) *capacity*. If *capacity* is 0, a small hash table is created. The NSHashTableCallBacks structure *callBacks* has five pointers to functions (documented under "Types and Constants"), with the following defaults: pointer hashing, if **hash** is NULL; pointer equality, if **isEqual** is NULL; no call-back upon adding an element, if **retain** is NULL; no call-back upon removing an element, if **release** is NULL; and a function returning a pointer's hexadecimal value as a string, if **describe** is NULL. The hashing function must be defined such that if two data elements are equal, as defined by the comparison function, the values produced by hashing on these elements must also be equal. Also, data elements must remain invariant if the value of the hashing function depends on them; for example, if the hashing function operates directly on the characters of a string, that string can't change.

SEE ALSO **NSCopyHashTableWithZone, NSCreateHashTableWithZone**

NSCreateHashTableWithZone

SUMMARY This function creates a new hash table in the specified zone.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS NSHashTable ***NSCreateHashTableWithZone**(NSHashTableCallBacks *callBacks*, unsigned int *capacity*, NSZone **zone*)

DESCRIPTION This function creates a new hash table in the specified *zone*. If *zone* is NULL, the hash table is created in the default zone.

The table's size is dependent on (but generally not equal to) *capacity*. If *capacity* is 0, a small hash table is created. The NSHashTableCallBacks structure *callBacks* has five pointers to functions (documented under "Types and Constants"), with the following defaults: pointer hashing, if **hash** is NULL; pointer equality, if **isEqual** is NULL; no call-back upon adding an element, if **retain** is NULL; no call-back upon removing an element, if **release** is NULL; and a function returning a pointer's hexadecimal value as a string, if **describe** is NULL. The hashing function must be defined such that if two data elements are equal, as defined by the comparison function, the values produced by hashing on these elements must also be equal. Also, data elements must remain invariant if the value of the hashing function depends on them; for example, if the hashing function operates directly on the characters of a string, that string can't change.

SEE ALSO **NSCreateHashTable**

NSCreateMapTable

SUMMARY This function creates a new map table in the default zone.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSMapTable ***NSCreateMapTable**(NSMapTableKeyCallBacks *keyCallBacks*, NSMapTableValueCallBacks *valueCallBacks*, unsigned int *capacity*)

DESCRIPTION **NSCreateMapTable** creates, and returns a pointer to, an NSMapTable in the default zone; the table's size is dependent on (but generally not equal to) *capacity*. If *capacity* is 0, a small map table is created. The NSMapTableKeyCallBacks arguments are structures (documented under "Types

and Constants") that are very similar to the call-back structure used by **NSCreateHashTable**; in fact, they have the same defaults as documented for that function.

SEE ALSO **NSCopyMapTableWithZone, NSCreateMapTableWithZone**

NSCreateMapTableWithZone

SUMMARY This function creates a new map table in the specified zone.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSMapTable ***NSCreateMapTableWithZone**(NSMapTableKeyCallBacks *keyCallBacks*, NSMapTableValueCallBacks *valueCallBacks*, unsigned int *capacity*, NSZone **zone*)

DESCRIPTION This function creates a new map table in the specified *zone*. If *zone* is NULL, the hash table is created in the default zone.

The table's size is dependent on (but generally not equal to) *capacity*. If *capacity* is 0, a small map table is created. The NSMapTableKeyCallBacks arguments are structures (documented under "Types and Constants") that are very similar to the call-back structure used by **NSCreateHashTable**; in fact, they have the same defaults as documented for that function.

SEE ALSO **NSCopyMapTableWithZone, NSCreateMapTable**

NSCreateZone

SUMMARY Creates a new zone

DECLARED IN Foundation/NSZone.h

SYNOPSIS NSZone ***NSCreateZone(**unsigned int *startSize*, unsigned int *granularity*, BOOL *canFree***)**

DESCRIPTION Creates and returns a pointer to a new zone of *startSize* bytes, which will grow and shrink by *granularity* bytes. If *canFree* is zero, the allocator will never free memory, and **malloc** will be fast.

SEE ALSO **NSDefaultMallocZone, NSRecycleZone, NSSetZoneName**

NSDeallocateMemoryPages

SUMMARY This function deallocates the specified block of memory.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void **NSDeallocateMemoryPages**(void ***pointer*, unsigned int *byteCount*)

DESCRIPTION This function deallocates memory that was allocated with **NSAllocateMemoryPages**.

SEE ALSO **NSCopyMemoryPages, NSAllocateMemoryPages**

NSDeallocateObject

SUMMARY This function destroys an existing object.

DECLARED IN Foundation/NSObject.h

SYNOPSIS void **NSDeallocateObject(**id <NSObject> *anObject***)**

DESCRIPTION This function deallocates *anObject*, which must have been allocated using NSAllocateObject.

SEE ALSO **NSCopyObject, NSAllocateObject**

NSDecimalAdd

SUMMARY This function adds two decimal values.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError **NSDecimalAdd**(NSDecimal *_result_, const NSDecimal *_leftOperand_, const NSDecimal *_rightOperand_, NSRoundingMode _roundingMode_)

DESCRIPTION Adds _leftOperand_ to _rightOperand_, and stores the sum in _result_.

An NSDecimal can represent a number with up to 38 significant digits. If a number is more precise than that, it must be rounded off. _roundingMode_ determines how to round it off. There are four possible rounding modes:

- NSRoundDown. The number rounds down.
- NSRoundUp. The number rounds up.
- NSRoundPlain. The number rounds to the closest 38-digit approximation. If the number is halfway between two positive numbers, it round up; if it's halfway between two negative numbers, it rounds down.
- NSRoundBankers. The number rounds to the closest 38-digit approximation. If it is caught halfway between two possibilities, it rounds to the one whose last digit is even. In practice, this means that, over the long run, numbers will be rounded up as often as they are rounded down; there will be no systematic bias.

The return value indicates whether any machine limitations were encountered in the addition. If none were encountered, the function returns NSCalculationNoError. Otherwise it may return one of the following values: NSCalculationLossOfPrecision, NSCalculationOverflow or NSCalculationUnderflow. For descriptions of all these error conditions, see **exceptionDuringOperation:error:leftOperand:rightOperand:** in the protocol specification for NSDecimalNumberBehaviors.

NSDecimalCompact

SUMMARY This function compacts the decimal structure for efficiency.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS void **NSDecimalCompact**(NSDecimal *_number_)

DESCRIPTION Formats *number* so that calculations using it will take up as little memory as possible. All the **NSDecimal...** arithmetic functions expect compact NSDecimal arguments.

NSDecimalCompare

SUMMARY This function compares two decimal values.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSComparisonResult **NSDecimalCompare**(const NSDecimal *_leftOperand_, const NSDecimal *_rightOperand_)

DESCRIPTION Compares *leftOperand* to *rightOperand*, with three possible return values.

- If *leftOperand* is bigger, the function returns NSOrderedDescending.
- If *rightOperand* is bigger, the the function returns NSOrdered Ascending.
- If the two operands are equal, the function returns NSOrderedSame.

NSDecimalCopy

SUMMARY This function copies the value of a decimal number.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS void **NSDecimalCopy**(NSDecimal *_destination_, const NSDecimal *_source_)

DESCRIPTION Copies the value in *source* to *destination*.

NSDecimalDivide

SUMMARY This function divides one decimal value by another.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError **NSDecimalDivide**(NSDecimal ***result*, const NSDecimal ***leftOperand*, const NSDecimal ***rightOperand*, NSRoundingMode *roundingMode*)

DESCRIPTION Divides *leftOperand* by *rightOperand*, and stores the quotient, possibly rounded off according to *roundingMode*, in *result*. If *rightOperand* is 0, returns NSDivideByZero.

For explanations of the other possible return values, and of all the possible *roundingMode*'s, see **NSDecimalAdd**, above.

Note that this function can't precisely represent a non-decimal fraction like 1/3.

NSDecimalIsNotANumber

SUMMARY This function determines if the specified decimal contains a valid number.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS BOOL **NSDecimalIsNotANumber**(const NSDecimal ***decimal*)

DESCRIPTION Returns YES if the value in *decimal* represents a valid number, otherwise returns NO.

NSDecimalMultiply

SUMMARY This function multiplies two decimal numbers together.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError **NSDecimalMultiply**(NSDecimal ***result*, const NSDecimal ***leftOperand*, const NSDecimal ***rightOperand*, NSRoundingMode *roundingMode*)

DESCRIPTION Multiplies *rightOperand* by *leftOperand*, and stores the product, possibly rounded off according to *roundingMode*, in *result*.

For explanations of the possible return values and *roundingMode*'s, see **NSDecimalAdd**, above.

NSDecimalMultiplyByPowerOf10

SUMMARY This function multiplies a decimal by the specified power of 10.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError NSDecimalMultiplyByPowerOf10(NSDecimal ***result*, const NSDecimal ***number*, short *power*, NSRoundingMode *roundingMode*)

DESCRIPTION Multiplies *number* by 10*power*, and stores the product, possibly rounded off according to *roundingMode*, in *result*.

For explanations of the possible return values and *roundingMode*'s, see **NSDecimalAdd**, above.

NSDecimalNormalize

SUMMARY This function normalizes the internal format of two decimal numbers to simplify later operations.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError **NSDecimalNormalize**(NSDecimal ***number1*, NSDecimal ***number2*, NSRoundingMode *roundingMode*)

DESCRIPTION An NSDecimal is represented in memory as a mantissa and an exponent, expressing the value mantissa **x** 10exponent. A number can have many NSDecimal representations; for example, the following table lists several valid NSDecimal representations for the number 100:

Mantissa	Exponent
100	0
10	1
1	2

NSDecimalNormalize formats number1 and number2 so that they have equal exponents. This format makes addition and subtraction very convenient. Both **NSDecimalAdd** and **NSDecimalSubtract** call **NSDecimalNormalize**. You may want to use it if you write more complicated addition or subtraction routines.

For explanations of the function's possible return values, see **NSDecimalAdd**, above.

NSDecimalPower

SUMMARY This function raises the decimal value to the specified power.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError **NSDecimalPower**(NSDecimal ***result*, const NSDecimal ***number*, unsigned int *power*, NSRoundingMode *roundingMode*)

DESCRIPTION Raises *number* to *power*, possibly rounding off according to *roundingMode*, and stores the resulting value in *result*.

For explanations of the possible return values and *roundingMode*'s, see **NSDecimalAdd**, above.

NSDecimalRound

SUMMARY This function rounds off the decimal value.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS void **NSDecimalRound**(NSDecimal **result*, const NSDecimal **number*, int *scale*, NSRoundingMode *roundingMode*)

DESCRIPTION Rounds *number* off according to the parameters *scale* and rounding *mode*, and stores the result in *result*.

scale specifies the number of digits result can have after its decimal point. *roundingMode* specifies the way that number is rounded off. There are four possible values for *roundingMode*: NSRoundDown, NSRoundUp, NSRoundPlain, and NSRoundBankers. For thorough discussions of *scale* and *roundingMode*, see the **scale** and **roundingMode** in the protocol specification for NSDecimalNumberBehaviors.

NSDecimalString

SUMMARY This function returns a string representation of the decimal value.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSString ***NSDecimalString**(const NSDecimal **decimal*, NSDictionary **locale*)

DESCRIPTION Returns a string representation of *decimal*. *locale* determines the format of the decimal separator.

NSDecimalSubtract

SUMMARY This function subtracts one decimal value from another.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError **NSDecimalSubtract**(NSDecimal ***result*, const NSDecimal ***leftOperand*, const NSDecimal ***rightOperand*, NSRoundingMode *roundingMode*)

DESCRIPTION Subtracts *rightOperand* from *leftOperand*, and stores the difference, possibly rounded off according to *roundingMode*, in *result*.

For explanations of the possible return values and *roundingMode*'s, see **NSDecimalAdd**, above.

NSDecrementExtraRefCountWasZero

SUMMARY This function decrements the specified object's reference count.

DECLARED IN Foundation/NSObject.h

SYNOPSIS BOOL **NSDecrementExtraRefCountWasZero(**id *anObject***)**

DESCRIPTION This function decrements the "extra reference" count of an object. Newly created objects have only one actual reference, so that a single release message results in the object being deallocated. Extra references are those beyond the single original reference, and are usually created by sending the object a retain message. Your code should generally not use these functions unless it is overriding the retain or release methods.

This function returns NO if *anObject* had an extra reference count. If *anObject* didn't have an extra referenct count, it returns YES, indicating that the object should be deallocated (with **dealloc**).

SEE ALSO **NSExtraRefCount, NSIncrementExtraRefCount**

NSDefaultMallocZone

SUMMARY Returns the default zone

DECLARED IN Foundation/NSZone.h

SYNOPSIS NSZone ***NSDefaultMallocZone(**void**)**

DESCRIPTION Returns the default zone, which is created automatically at startup. This is the zone used by the standard C function **malloc**.

SEE ALSO **NSCreateZone**

NSDivideRect

SUMMARY This function divides a rectangle into two new rectangles.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS void **NSDivideRect**(NSRect *inRect*, NSRect **slice*, NSRect **remainder*, float *amount*, NSRectEdge *edge*)

DESCRIPTION Creates two rectangles, *slice* and *remainder,* from *inRect*, by dividing *inRect* with a line that's parallel to one of *inRect*'s sides (namely, the side specified by edge—either NSMinXEdge, NSMinYEdge, NSMaxXEdge, or NSMaxYEdge). The size of *slice* is determined by *amount,* which measures the distance from *edge*.

SEE ALSO **NSInsetRect, NSIntegralRect, NSOffsetRect**

NSEnumerateHashTable

SUMMARY This function creates an enumerator for the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS NSHashEnumerator **NSEnumerateHashTable**(NSHashTable *_table_)

DESCRIPTION Returns an **NSHashEnumerator** structure that will cause successive elements of *table* to be returned each time this enumerator is passed to **NSNextHashEnumeratorItem**.

SEE ALSO **NSNextHashEnumeratorItem**

NSEnumerateMapTable

SUMMARY This function creates an enumerator for the specified map table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSMapEnumerator **NSEnumerateMapTable**(NSMapTable *_table_)

DESCRIPTION **NSEnumerateMapTable** returns an **NSMapEnumerator** structure that will cause successive key/value pairs of *table* to be visited each time this enumerator is passed to **NSNextMapEnumeratorPair**.

SEE ALSO **NSNextMapEnumeratorPair, NSMapMember, NSMapGet, NSAllMapTableKeys, NSAllMapTableValues**

NSEqualPoints

SUMMARY This function tests the two points for equality.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSEqualPoints**(NSPoint *aPoint*, NSPoint *bPoint*)

DESCRIPTION Returns YES if the two points *aPoint* and *bPoint* are identical, and NO otherwise.

NSEqualRanges

SUMMARY This function tests the two range values for equality.

DECLARED IN Foundation/NSRange.h

SYNOPSIS BOOL **NSEqualRanges**(NSRange *range1*, NSRange *range2*)

DESCRIPTION Returns YES if *range1* and *range2* have the same locations and lengths.

NSEqualRects

SUMMARY This function tests the two rectangles for equality.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSEqualRects**(NSRect *aRect*, NSRect *bRect*)

DESCRIPTION Returns YES if the two rectangles *aRect* and *bRect* are identical, and NOotherwise.

NSEqualSizes

SUMMARY This function tests the two size values for equality.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSEqualSizes**(NSSize *aSize*, NSSize *bSize*)

DESCRIPTION Returns YES if the two sizes *aSize* and *bSize* are identical, and NO otherwise.

NSExtraRefCount

SUMMARY This function returns the specified object's reference count.

DECLARED IN Foundation/NSObject.h

SYNOPSIS unsigned int **NSExtraRefCount**(id *object*)

DESCRIPTION Returns the current reference count of *object*. This function is used in conjunction with **NSIncrementExtraRefCount** and **NSDecrementExtraRefCountWasZero** in situations where you need to override an object's **retain** and **release** methods.

NSFreeHashTable

SUMMARY This function deletes the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void **NSFreeHashTable**(NSHashTable **table*)

DESCRIPTION **NSFreeHashTable** releases each element of the specified hash table and frees the table itself.

SEE ALSO **NSResetHashTable**

NSFreeMapTable

SUMMARY This function deletes the specified map table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void **NSFreeMapTable**(NSMapTable *_table_)

DESCRIPTION **NSFreeMapTable** releases each key and value of the specified map table and frees the table itself.

SEE ALSO **NSResetMapTable**

NSFullUserName

SUMMARY This function returns the full name of the current user.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSString ***NSFullUserName**(void)

DESCRIPTION Returns a string containing the full name of the current user.

SEE ALSO **NSUserName**

NSGetSizeAndAlignment

SUMMARY This function returns the type and size of the specified data type.

DECLARED IN Foundation/NSObjCRuntime.h

SYNOPSIS const char ***NSGetSizeAndAlignment**(const char **typePtr*,
unsigned int **sizep*,
unsigned int **alignp*)

DESCRIPTION Gets the next type code from *typePtr* and returns the size and alignment of that data type in *sizep* and *alignp*, respectively. You can specify 0 for either *sizep* or *alignp* to ignore the corresponding information. This function returns a new pointer to the string specified by *typePtr*; the new pointer points to the character just past the type code that was read.

NSGetUncaughtExceptionHandler

SUMMARY This function returns the top-level error handler.

DECLARED IN Foundation/NSException.h

SYNOPSIS NSUncaughtExceptionHandler ***NSGetUncaughtExceptionHandler(**void**)**

DESCRIPTION **NSGetUncaughtExceptionHandler** returns a pointer to the function serving as the top-level error handler. This handler will process exceptions raised outside of any exception-handling domain.

SEE ALSO **NSSetUncaughtExceptionHandler**

NSHashGet

SUMMARY This function returns an element of the hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void ***NSHashGet**(NSHashTable **table*, const void **pointer*)

DESCRIPTION Returns the pointer in the table that matches *pointer* (as defined by the **isEqual** call-back function). If there is no matching element, the function returns NULL

NSHashInsert

SUMMARY This function adds an element to the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void **NSHashInsert**(NSHashTable *_table_, const void *_pointer_)

DESCRIPTION **NSHashInsert** inserts *pointer*, which must not be NULL, into *table*. If *pointer* matches an item already in the table, the previous pointer is released using the **release** call-back function that was specified when the table was created.

SEE ALSO **NSHashRemove, NSHashInsertKnownAbsent, NSHashInsertIfAbsent**

NSHashInsertIfAbsent

SUMMARY This function adds an element to the specified hash table only if the table does not already contain the element.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void ***NSHashInsertIfAbsent**(NSHashTable *_table_, const void *_pointer_)

DESCRIPTION If *pointer* matches an item already in *table*, **NSHashInsertIfAbsent** returns the pre-existing pointer; otherwise, it adds *pointer* to the table and returns NULL. You must not specify NULL for *pointer*.

SEE ALSO **NSHashRemove, NSHashInsert, NSHashInsertKnownAbsent**

NSHashInsertKnownAbsent

SUMMARY This function adds an element to the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void **NSHashInsertKnownAbsent**(NSHashTable *_table_, const void *_pointer_)

DESCRIPTION **NSHashInsertKnownAbsent** inserts *pointer*, which must not be NULL, into *table*. Unike **NSHashInsert**, this function raises NSInvalidArgumentException if *table* already includes an element that matches *pointer.*

SEE ALSO **NSHashRemove, NSHashInsert, NSHashInsertIfAbsent**

NSHashRemove

SUMMARY This function removes an element from the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void **NSHashRemove**(NSHashTable *_table_, const void *_pointer_)

DESCRIPTION If *pointer* matches an item already in *table*, this function releases the pre-existing item.

SEE ALSO **NSHashInsert, NSHashInsertKnownAbsent, NSHashInsertIfAbsent**

NSHeight

SUMMARY This function returns the height of the specified rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSHeight**(NSRect *aRect*)

DESCRIPTION Returns the height of *aRect*.

SEE ALSO **NSMaxX, NSMaxY, NSMidX, NSMidY, NSMinX, NSMinY, NSWidth**

NSHomeDirectory

SUMMARY This function getsinformation about a user.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSString ***NSHomeDirectory(**void**)**

DESCRIPTION **NSHomeDirectory** returns a path to the current user's home directory.

SEE ALSO **NSFullUserName, NSUserName, NSHomeDirectoryForUser**

NSHomeDirectoryForUser

SUMMARY Get information about a user

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSString ***NSHomeDirectoryForUser(**NSString **userName***)**

DESCRIPTION **NSHomeDirectoryForUser** returns a path to the home directory for the user specified by *userName*.

SEE ALSO **NSFullUserName, NSUserName, NSHomeDirectory**

NSHostByteOrder

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned int **NSHostByteOrder**(void)

DESCRIPTION Returns the endian format, either NSLittleEndian or NSBigEndian, supported by the current platform.

NSIncrementExtraRefCount

SUMMARY This function increments the specified object's reference count.

DECLARED IN Foundation/NSObject.h

SYNOPSIS void **NSIncrementExtraRefCount(**id *anObject***)**

DESCRIPTION This function increments the "extra reference" count of an object. Newly created objects have only one actual reference, so that a single release message results in the object being deallocated. Extra references are those beyond the single original reference, and are usually created by sending the object a retain message. Your code should generally not use these functions unless it is overriding the retain or release methods.

SEE ALSO **NSExtraRefCount, NSDecrementExtraRefCountWasZero**

NSInsetRect

SUMMARY This function insets the rectangle by the specified amount.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSInsetRect**(NSRect *aRect*, float *dX*, float *dY*)

DESCRIPTION **NSInsetRect** returns a copy of the rectangle *aRect*, altered by moving the two sides that are parallel to the y-axis inwards by *dX*, and the two sides parallel to the x-axis inwards by *dY*.

SEE ALSO **NSDivideRect, NSIntegralRect, NSOffsetRect**

NSIntegralRect

SUMMARY This function adjusts the sides of the rectangle to integer values.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSIntegralRect**(NSRect *aRect*)

DESCRIPTION **NSIntegralRect** returns a copy of the rectangle *aRect*, expanded outwards just enough to ensure that none of its four defining values (*x*, *y*, *width*, and *height*) have fractional parts. If *aRect*'s width or height is zero or negative, this function returns a rectangle with origin at (0.0, 0.0) and with zero width and height.

SEE ALSO **NSDivideRect, NSInsetRect, NSOffsetRect**

NSIntersectionRange

SUMMARY This function returns the intersection of the specified ranges.

DECLARED IN Foundation/NSRange.h

SYNOPSIS NSRange **NSIntersectionRange(**NSRange *range1*, NSRange *range2***)**

DESCRIPTION **NSIntersectionRange** returns a range describing the intersection of *range1* and *range2*—that is, a range containing the indices that exist in both ranges. If the returned range's **length** field is zero, then the two ranges don't intersect, and the value of the **location** field is undefined.

SEE ALSO **NSUnionRange**

NSIntersectionRect

SUMMARY This function calculates the intersection of two rectangles.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSIntersectionRect**(NSRect *aRect*, NSRect *bRect*)

DESCRIPTION **NSIntersectionRect** returns the graphic intersection of *aRect* and *bRect*. If the two rectangles don't overlap, the returned rectangle has its origin at (0.0, 0.0) and zero width and height. (This includes situations where the intersection is a point or a line segment.)

SEE ALSO **NSUnionRect**

NSIntersectsRect

SUMMARY This function tests whether two rectangles intersect.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSIntersectsRect**(NSRect *aRect*, NSRect *bRect*)

DESCRIPTION Returns YES if *aRect* intersects *bRect*, otherwise returns NO. This function returns NO if either *aRect* and *bRect* has a width or height that is 0.

SEE ALSO **NSIntersectionRect**

NSIsEmptyRect

SUMMARY This function tests whether the specified rectangle is empty.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSIsEmptyRect**(NSRect *aRect*)

DESCRIPTION Returns YES if the rectangle encloses no area at all—that is, if its width or height is zero or negative.

NSJavaBundleCleanup

SYNOPSIS void **NSJavaBundleCleanup**(NSBundle **bundle*, NSDictionary **plist*)

DESCRIPTION <<Description Forthcoming>>

NSJavaBundleSetup

SYNOPSIS id **NSJavaBundleSetup**(NSBundle **bundle*, NSDictionary **plist*)

DESCRIPTION <<Description Forthcoming>>

NSJavaNeedsToLoadClasses

SYNOPSIS BOOL **NSJavaNeedsToLoadClasses**(NSDictionary **plist*)

DESCRIPTION <<Description Forthcoming>>

NSJavaNeedsVirtualMachine

SYNOPSIS BOOL **NSJavaNeedsVirtualMachine**(NSDictionary *_plist_)

DESCRIPTION <<Description Forthcoming>>

NSJavaProvidesClasses

SYNOPSIS BOOL NSJavaProvidesClasses(NSDictionary *plist)

DESCRIPTION <<Description Forthcoming>>

NSJavaSetup

SYNOPSIS id NSJavaSetup(NSDictionary *plist)

DESCRIPTION <<Description Forthcoming>>

NSJavaVirtualMachineSetup

SYNOPSIS id **NSJavaVirtualMachineSetup**(BOOL *_vmNeeded_)

DESCRIPTION <<Description Forthcoming>>

NSLocationInRange

SUMMARY This function verifies that the specified position is in the given range.

DECLARED IN Foundation/NSRange.h

SYNOPSIS BOOL **NSLocationInRange(**unsigned int *index*, NSRange *aRange***)**

DESCRIPTION **NSLocationInRange** returns YES if the given *index* lies within *aRange*—that is, if it's greater than or equal to *aRange*.location and less than *aRange*.location plus *aRange*.length.

NSLog

SUMMARY This function logs error messages to **stderr**.

DECLARED IN Foundation/NSUtilities.h

SYNOPSIS void **NSLog(**NSString **format*, ...**)**

DESCRIPTION NSLog simply calls NSLogv, passing it a variable number of arguments.

SEE ALSO **NSLogv**

NSLogPageSize

SUMMARY This function returns the binary log of the page size.

DECLARED IN Foundation/NSZone.h

SYNOPSIS unsigned int **NSLogPageSize**(void)

DESCRIPTION **NSLogPageSize** returns the binary log of the current page size.

SEE ALSO **NSRoundDownToMultipleOfPageSize, NSRoundUpToMultipleOfPageSize, NSPageSize**

NSLogv

SUMMARY This function logs error messages to **stderr**.

DECLARED IN Foundation/NSUtilities.h

SYNOPSIS void **NSLogv(**NSString ***format*, va_list *args***)**

DESCRIPTION **NSLogv** logs an error message. The message consists of a timestamp and the process ID prefixed to the string you pass in. You compose this string with a format string , *format*, and one or more arguments to be inserted into the string. The format specification allowed by these functions is that which is understood by NSString's formatting capabilities (which is not necessarily the set of format escapes and flags understood by **printf**).

In general, you should use the **NSLog** function instead of calling this function directly. If you do use this function directly, you must have prepared the the variable argument list in the *args* parameter by calling the standard C macro va_start. Upon completion, you must similarly call the standard C macro va_end for this list.

On HP-UX, Solaris, and Mach, NSLogv writes the log to STDERR_FILENO if the file descriptor is open. If that fails, the message is sent to the syslog subsystem, if it exists on a platform, with the LOG_USER facility (or default facility if LOG_USER doesn't exist on a platform), with priority LOG_ERR (or similar, depending on what the platform supports). If both of these attempts to write the message fail, the message is discarded.

On Windows platforms, the message is written to the STD_ERROR_HANDLE, if that handle is valid, on Windows platforms that support that standard handle. It is also written to the Windows Event Log on Windows platforms that support that, or to a file **c:\fndation.log** on Windows platforms that do not, if that file can be opened. If all of these attempts fail, the message is discarded. On some Windows platforms, the message to the Event Log may be truncated if there is a limit to the size of a message that the Event Log can accept. On Windows platforms that support an application discovering whether or not it's running under a debugger, NSLogv may only send the message to the debugger for its handling, via standard WIN32 mechanisms, and not also write the message to STD_ERROR_HANDLE and the Event Log. Note that a debugger may

choose to not display message thus sent to it, or may choose not to display all of the message—that has nothing to do with **NSLogv**.

Output from **NSLogv** is serialized, in that only one thread in a process can be doing the writing/logging described above at a time. All attempts at writing/logging a message complete before the next thread can begin its attempts.

The effects of **NSLogv** are not serialized with subsystems other than those discussed above (such as the standard I/O package) and do not produce side effects on those subsystems (such as causing buffered output to be flushed, which may be undesirable).

SEE ALSO **NSLog**

NSMakePoint

SUMMARY This function creates a new NSPoint from the specified values.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSPoint **NSMakePoint**(float *x*, float *y*)

DESCRIPTION Creates an NSPoint having the coordinates *x* and *y*.

NSMakeRange

SUMMARY This function creates a new NSRange from the specified values.

DECLARED IN Foundation/NSRange.h

SYNOPSIS NSRange **NSMakeRange**(unsigned int *location*, unsigned int *length*)

DESCRIPTION Creates an NSRange having the specified location and length.

NSMakeRect

SUMMARY This function creates a new NSRect from the specified values.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSMakeRect**(float *x*, float *y*, float *w*, float *h*)

DESCRIPTION Creates an NSRect having the specified origin and size.

NSMakeSize

SUMMARY This function creates a new NSSize from the specified values.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSSize **NSMakeSize**(float *w*, float *h*)

DESCRIPTION Creates an NSSize having the specified width and height.

NSMapGet

SUMMARY This function returns a map-table value for the specified key.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void ***NSMapGet**(NSMapTable **table*, const void **key*)

DESCRIPTION Returns the value that table maps to key, or NULL if the table doesn't contain key.

SEE ALSO **NSMapMember, NSEnumerateMapTable, NSNextMapEnumeratorPair, NSAllMapTableKeys, NSAllMapTableValues**

NSMapInsert

SUMMARY This function inserts a key/value pair into the specified table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void **NSMapInsert**(NSMapTable *_table_, const void *_key_, const void *_value_)

DESCRIPTION **NSMapInsert** inserts _key_ and _value_ into table. If _key_ matches a key already in the table, _value_ is retained and the previous value is released, using the retain and release call-back functions that were specified when the table was created. Raises InvalidArgumentException if _key_ is equal to the **notAKeyMarker** field of the table's NSMapTableKeyCallBacks structure.

SEE ALSO **NSMapRemove, NSMapInsertIfAbsent, NSMapInsertKnownAbsent**

NSMapInsertIfAbsent

SUMMARY This function inserts a key/value pair into the specified table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void ***NSMapInsertIfAbsent**(NSMapTable *_table_, const void *_key_, const void *_value_)

DESCRIPTION If _key_ matches a key already in table, **NSMapInsertIfAbsent** returns the pre-existing key; otherwise, it adds _key_ and _value_ to the table and returns NULL. Raises NSInvalidArgumentException if _key_ is equal to the **notAKeyMarker** field of the table's NSMapTableKeyCallBacks structure.

SEE ALSO **NSMapRemove, NSMapInsert, NSMapInsertKnownAbsent**

NSMapInsertKnownAbsent

SUMMARY This function inserts a key/value pair into the specified table if the pair had not been previously added.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void **NSMapInsertKnownAbsent**(NSMapTable *_table_, const void *_key_, const void *_value_)

DESCRIPTION **NSMapInsertKnownAbsent** inserts *key* (which must not be **notAKeyMarker**) and *value* into table. Unlike **NSMapInsert**, this function raises NSInvalidArgumentException if *table* already includes a key that matches *key*.

SEE ALSO **NSMapRemove, NSMapInsert, NSMapInsertIfAbsent**

NSMapMember

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS BOOL **NSMapMember**(NSMapTable *_table_, const void *_key_, void **_originalKey_, void **_value_)

DESCRIPTION Returns YES if table contains a key equal to *key*. If so, *originalKey* is set to *key*, and *value* is set to the value that the table maps to *key*.

SEE ALSO **NSMapGet, NSEnumerateMapTable, NSNextMapEnumeratorPair, NSAllMapTableKeys, NSAllMapTableValues**

NSMapRemove

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void **NSMapRemove**(NSMapTable *_table_, const void *_key_)

DESCRIPTION If *key* matches a key already in *table*, this function release the pre-existing key and its corresponding value.

SEE ALSO **NSMapInsert, NSMapInsertIfAbsent, NSMapInsertKnownAbsent**

NSMaxRange

DECLARED IN Foundation/NSRange.h

SYNOPSIS unsigned int **NSMaxRange**(NSRange *range*)

DESCRIPTION Returns *range*.location + *range*.length—in other words, the number one greater than the maximum value within the range.

NSMaxX

SUMMARY This function returns the largest x-coordinate of a rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSMaxX**(NSRect *aRect*)

DESCRIPTION **NSMaxX** returns the largest x-coordinate value within *aRect*.

SEE ALSO **NSWidth, NSHeight, NSMaxY**

NSMaxY

SUMMARY This function returns the largest y-coordinate of a rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSMaxY**(NSRect *aRect*)

DESCRIPTION **NSMaxY** returns the largest y-coordinate value within *aRect*.

SEE ALSO **NSWidth, NSHeight, NSMaxX**

NSMidX

SUMMARY This function returns the x-coordinate of a rectangle’s midpoint.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSMidX**(NSRect *aRect*)

DESCRIPTION **NSMidX** returns the x-coordinate of the center of the rectangle.

SEE ALSO **NSWidth, NSHeight, NSMidY**

NSMidY

SUMMARY This function returns the y-coordinate of a rectangle’s midpoint.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSMidY**(NSRect *aRect*)

DESCRIPTION **NSMidY** returns the y-coordinate of the rectangle's center point.

SEE ALSO **NSWidth, NSHeight, NSMidX**

NSMinX

SUMMARY This function returns the smallest x-coordinate of a rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSMinX**(NSRect *aRect*)

DESCRIPTION **NSMinX** returns the smallest x-coordinate value within *aRect*.

SEE ALSO **NSWidth, NSHeight, NSMinY**

NSMinY

SUMMARY This function returns the smallest y-coordinate of a rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSMinY**(NSRect *aRect*)

DESCRIPTION **NSMinY** returns the smallest y-coordinate value within *aRect* .

SEE ALSO **NSWidth, NSHeight, NSMinX**

NSMouseInRect

SUMMARY This function tests whether the point is in the specified rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSMouseInRect**(NSPoint *aPoint*, NSRect *aRect*, BOOL *flipped*)

DESCRIPTION Returns YES if the point represented by *aPoint* is located within the rectangle represented by *aRect*. It assumes an unscaled and unrotated coordinate system; the argument *flipped* should be YES if the coordinate system has been flipped so that the positive y-axis extends downward. This function is used to determine whether the hot spot of the cursor lies inside a given rectangle.

NSNextHashEnumeratorItem

SUMMARY This function returns the next hash-table element in the enumeration.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void ***NSNextHashEnumeratorItem**(NSHashEnumerator **enumerator*)

DESCRIPTION **NSNextHashEnumeratorItem** returns the next element in the table that enumerator is associated with, or NULL if *enumerator* has already iterated over all the elements.

SEE ALSO **NSEnumerateHashTable**

NSNextMapEnumeratorPair

SUMMARY This function returns the next map-table pair in the enumeration

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS BOOL **NSNextMapEnumeratorPair**(NSMapEnumerator *_enumerator_, void **_key_, void **_value_)

DESCRIPTION **NSNextMapEnumeratorPair** returns NO if *enumerator* has already iterated over all the elements in the table that *enumerator* is associated with. Otherwise, this function sets *key* and *value* to match the next key/value pair in the table, and returns YES.

SEE ALSO **NSEnumerateMapTable, NSMapMember, NSMapGet, NSAllMapTableKeys, NSAllMapTableValues**

NSOffsetRect

SUMMARY This function insets the rectangle by the specified amount.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSOffsetRect**(NSRect *aRect*, float *dX*, float *dY*)

DESCRIPTION **NSOffsetRect** returns a copy of the rectangle *aRect*, with its location shifted by *dX* along the x-axis and by *dY* along the y-axis.

SEE ALSO **NSDivideRect, NSInsetRect, NSIntegralRect**

NSOpenStepRootDirectory

SUMMARY This function returns the root directory of the user's system.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSString ***NSOpenStepRootDirectory**(void)

DESCRIPTION Returns a string identifying the root directory of the user's system.

SEE ALSO **NSHomeDirectory, NSHomeDirectoryForUser**

NSPageSize

SUMMARY This function returns the number of bytes in a page.

DECLARED IN Foundation/NSZone.h

SYNOPSIS unsigned int **NSPageSize**(void)

DESCRIPTION **NSPageSize** returns the number of bytes in a page.

SEE ALSO **NSRoundDownToMultipleOfPageSize, NSRoundUpToMultipleOfPageSize, NSLogPageSize**

NSParameterAssert

SUMMARY This macro validates the specified parameter.

DECLARED IN Foundation/NSExceptions.h

SYNOPSIS **NSParameterAssert(***condition***)**

DESCRIPTION Assertions evaluate a condition and, if the condition evaluates to false, call the assertion handler for the current thread, passing it a format string and a variable number of arguments. Each thread has its own assertion handler, which is an object of class NSAssertionHandler. When invoked, an assertion handler prints an error message that includes method and class names (or the function name). It then raises an exception of type NSInternalInconsistencyException.

This macro validates a parameter for an Objective-C method. Simply provide the parameter as the *condition* argument. The macro evaluates the parameter and, if it is false, it logs an error message which includes the parameter and then raises an exception.

Assertions are disabled if the preprocessor macro **NS_BLOCK_ASSERTIONS** is defined.All assertion macros return **void**.

SEE ALSO **NSLog, NSLogv, NSAssert, NSCAssert, NSCParameterAssert**

NSPointFromString

SUMMARY This function returns a point from a text-based representation.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSPoint **NSPointFromString**(NSString ***aString*)

DESCRIPTION Scans *aString* for text of the form "{x=*a*; y=*b*}" and returns the values for *a* and *b* in a new NSPoint object. If *aString* does not contain this text, this function returns an NSPoint object whose x- and y-coordinates are both 0.

SEE ALSO **NSStringFromPoint**

NSPointInRect

SUMMARY This function tests whether the specified point is in the rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSPointInRect**(NSPoint *aPoint*, NSRect *aRect*)

DESCRIPTION Performs the same test as **NSMouseInRect**, but assumes a flipped coordinate system.

NSRangeFromString

SUMMARY This function returns a range from a text-based representation.

DECLARED IN Foundation/NSRange.h

SYNOPSIS NSRange **NSRangeFromString**(NSString **aString*)

DESCRIPTION Returns *aString* for text of the form: "{location = *a*; length = *b*}" and returns the values for *a* and *b* in a new NSRange object. If *aString* does not contain this text, this function returns an NSRange object whose location and length values are both 0.

SEE ALSO **NSStringFromRange**

NSRealMemoryAvailable

SUMMARY This function returns information about the user's system.

DECLARED IN Foundation/NSZone.h

SYNOPSIS unsigned int **NSRealMemoryAvailable**(void)

DESCRIPTION **NSRealMemoryAvailable** returns the number of bytes available in RAM.

NSRectFromString

SUMMARY This function returns a rectangle from a text-based representation.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSRectFromString**(NSString *_aString_)

DESCRIPTION Scans _aString_ for text of the form "{x=_a_; y=_b_; width=_c_; height=_d_}", and returns the values for _a_, _b_, _c_, and _d_ in a new NSRect object. If _aString_ does not contain the specified text, this function returns an NSRect object with a rectangle whose origin is (0, 0) and width and height are both 0.

SEE ALSO **NSStringFromRect**

NSRecycleZone

SUMMARY This function frees memory in a zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void **NSRecycleZone(**NSZone *_zone_**)**
void **NSZoneFree**(NSZone *_zone_, void *_pointer_**)**

DESCRIPTION **NSRecycleZone** frees _zone_ after adding any of its pointers still in use to the default zone. (This strategy prevents retained objects from being inadvertently destroyed.)

NSZoneFree returns the memory indicated by _pointer_ to _zone_. The standard C function **free** does the same, but spends time finding which zone the memory belongs to.

RETURN Both functions return **void**.

SEE ALSO **NSCreateZone, NSZoneMalloc**

NSResetHashTable

SUMMARY This function deletes the elements of the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void **NSResetHashTable**(NSHashTable *_table_)

DESCRIPTION **NSResetHashTable** releases each element but doesn't deallocate the table. This is useful for preserving the table's capacity.

SEE ALSO **NSFreeHashTable**

NSResetMapTable

SUMMARY This function deletes the elements of the specified map table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void **NSResetMapTable**(NSMapTable *_table_)

DESCRIPTION **NSResetMapTable** releases each key and value but doesn’t deallocate the table. This is useful for preserving the table’s capacity.

SEE ALSO **NSFreeMapTable**

NSRoundDownToMultipleOfPageSize

SUMMARY This function returns the number of pages that correspond to the specified number of bytes.

DECLARED IN Foundation/NSZone.h

SYNOPSIS unsigned int **NSRoundDownToMultipleOfPageSize**(unsigned int *byteCount*)

DESCRIPTION **NSRoundDownToMultipleOfPageSize** returns the multiple of the page size that is closest to, but not greater than, *byteCount*.

SEE ALSO **NSPageSize, NSLogPageSize, NSRoundUpToMultipleOfPageSize**

NSRoundUpToMultipleOfPageSize

SUMMARY This function returns the number of pages that correspond to the specified number of bytes.

DECLARED IN Foundation/NSZone.h

SYNOPSIS unsigned int **NSRoundUpToMultipleOfPageSize**(unsigned int *byteCount*)

DESCRIPTION **NSRoundUpToMultipleOfPageSize** returns the multiple of the page size that is closest to, but not less than, *byteCount*.

SEE ALSO **NSPageSize, NSLogPageSize, NSRoundDownToMultipleOfPageSize**

NSSearchPathForDirectoriesInDomains

SYNOPSIS NSArray ***NSSearchPathForDirectoriesInDomains** (NSSearchPathDirectory *directory*, NSSearchPathDomainMask *domainMask*, BOOL *expandTilde*)

DESCRIPTION <<Description forthcoming>>

NSSelectorFromString

SUMMARY This function obtains a selector by name.

DECLARED IN Foundation/NSObjCRuntime.h

SYNOPSIS SEL **NSSelectorFromString(**NSString *_aSelectorName_**)**

DESCRIPTION **NSSelectorFromString** returns the selector named by *aSelectorName*, or zero if none by this name exists.

SEE ALSO **NSStringFromSelector**

NSSetUncaughtExceptionHandler

SUMMARY This function changes the top level error handler.

DECLARED IN Foundation/NSException.h

SYNOPSIS void **NSSetUncaughtExceptionHandler(**NSUncaughtExceptionHandler *_handler_**)**

DESCRIPTION **NSSetUncaughtExceptionHandler** sets the top-level error-handling function to *handler*. If *handler* is NULL or this function is never invoked, the default top-level handler is used.

SEE ALSO **NSGetUncaughtExceptionHandler**

NSSetZoneName

SUMMARY This function sets the name of the specified zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void **NSSetZoneName(**NSZone *_zone_, NSString *_name_**)**

DESCRIPTION **NSSetZoneName** sets the specified *zone*'s name to *name*, which can aid in debugging.

SEE ALSO **NSZoneName**

NSShouldRetainWithZone

SUMMARY This function indicates whether an object should be retained.

DECLARED IN Foundation/NSObject.h

SYNOPSIS BOOL **NSShouldRetainWithZone(**id <NSObject> *anObject*, NSZone **requestedZone***)**

DESCRIPTION Returns YES if *requestedZone* is NULL, the default zone, or the zone in which *anObject* was allocated. This function is typically called from inside an NSObject's **copyWithZone:** method, when deciding whether to retain *anObject* as opposed to making a copy of it.

NSSizeFromString

SUMMARY This function returns an NSSize from a text-based representation.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSSize **NSSizeFromString**(NSString **aString*)

DESCRIPTION Scans *aString* for text of the form "{width=*a*; height=*b*}" and returns the values for *a* and *b* in an NSSize object. If *aString* does not contain the specified text, this function returns an NSSize object whose width and height are both 0.

SEE ALSO **NSStringFromSize**

NSStandardApplicationPaths

SUMMARY This function returns the application paths for the user's system.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSArray* **NSStandardApplicationPaths**(void)

DESCRIPTION Returns an array of strings, each string specifying one of the standard paths to the OpenStep applications.

SEE ALSO **NSStandardLibraryPaths**

NSStandardLibraryPaths

SUMMARY This function returns the library paths for the user's system.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSArray* **NSStandardLibraryPaths**(void)

DESCRIPTION Returns an array of strings, each string specifying one of the standard paths to the OpenStep libraries.

SEE ALSO **NSStandardApplicationPaths**

NSStringFromClass

SUMMARY This function obtains the name of a class

DECLARED IN Foundation/NSObjCRuntime.h

SYNOPSIS NSString ***NSStringFromClass**(Class *aClass*)

DESCRIPTION This function returns an NSString containing the name of *aClass*.

SEE ALSO **NSClassFromString**

NSStringFromHashTable

SUMMARY This function returns a string describing the hash table's contents.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS NSString ***NSStringFromHashTable**(NSHashTable **table*)

DESCRIPTION Returns a string describing the hash table's contents. The function iterates over the table's elements, and for each one appends the string returned by the **describe** call-back function. If NULL was specified for the call-back function, the hexadecimal value of each pointer is added to the string.

NSStringFromMapTable

SUMMARY This function returns a string describing the map table's contents.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSString ***NSStringFromMapTable**(NSMapTable **table*)

DESCRIPTION Returns a string describing the map table's contents. The function iterates over the table's key/value pairs, and for each one appends the string "*a* = *b*;\n", where *a* and *b* are the key and value strings returned by the corresponding **describe** call-back functions. If NULL was specified for the call-back function, a and b are the key and value pointers, expressed as hexadecimal numbers.

NSStringFromPoint

SUMMARY This function returns a string representation of a point.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSString ***NSStringFromPoint**(NSPoint *aPoint*)

DESCRIPTION Returns a string of the form "{x=*a*; y=*b*}", where *a* and *b* are the x- and y-coordinates of *aPoint*.

SEE ALSO **NSPointFromString**

NSStringFromRange

SUMMARY This function returns a string representation of a range.

DECLARED IN Foundation/NSRange.h

SYNOPSIS NSString ***NSStringFromRange**(NSRange *aRange*)

DESCRIPTION Returns a string of the form: "{location = *a*; length = *b*}", where *a* and *b* are non-negative integers representing *aRange*.

NSStringFromRect

SUMMARY This function returns a string representation of a rect.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSString ***NSStringFromRect**(NSRect *aRect*)

DESCRIPTION Returns a string of the form "{x=*a*; y=*b*; width=*c*; height=*d*}", where *a*, *b*, *c*, and *d* are the x- and y-coordinates and the width and height, respectively, of *aRect*.

SEE ALSO **NSRectFromString**

NSStringFromSelector

SUMMARY This function returns the name of a selector.

DECLARED IN Foundation/NSObjCRuntime.h

SYNOPSIS NSString ***NSStringFromSelector(**SEL *aSelector***)**

DESCRIPTION **NSStringFromSelector** returns an NSString containing the name of *aSelector.*

SEE ALSO **NSSelectorFromString**

NSStringFromSize

SUMMARY This function returns a string representation of a size.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSString ***NSStringFromSize**(NSSize *aSize*)

DESCRIPTION Returns a string of the form "{width=*a*; height=*b*}", where *a* and *b* are the width and height of *aSize*.

SEE ALSO **NSSizeFromString**

NSSwapBigDoubleToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS double **NSSwapBigDoubleToHost**(NSSwappedDouble *x*)

DESCRIPTION Converts the big-endian value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapDouble** to perform the swap.

SEE ALSO **NSSwapHostDoubleToBig, NSSwapLittleDoubleToHost**

NSSwapBigFloatToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS float **NSSwapBigFloatToHost**(NSSwappedFloat *x*)

DESCRIPTION Converts the big-endian value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapFloat** to perform the swap.

SEE ALSO **NSSwapHostFloatToBig, NSSwapLittleFloatToHost**

NSSwapBigIntToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned int **NSSwapBigIntToHost**(unsigned int *x*)

DESCRIPTION Converts the big-endian value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapInt** to perform the swap.

SEE ALSO **NSSwapHostIntToBig, NSSwapLittleIntToHost**

NSSwapBigLongLongToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long long **NSSwapBigLongLongToHost**(unsigned long long *x*)

DESCRIPTION Converts the big-endian value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapLongLong** to perform the swap.

SEE ALSO **NSSwapHostLongLongToBig, NSSwapLittleLongLongToHost**

NSSwapBigLongToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long **NSSwapBigLongToHost**(unsigned long *x*)

DESCRIPTION Converts the big-endian value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapLong** to perform the swap.

SEE ALSO **NSSwapHostLongToBig, NSSwapLittleLongToHost**

NSSwapBigShortToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned short **NSSwapBigShortToHost**(unsigned short *x*)

DESCRIPTION Converts the big-endian value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapShort** to perform the swap.

SEE ALSO **NSSwapHostShortToBig, NSSwapLittleShortToHost**

NSSwapDouble

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedDouble **NSSwapDouble**(NSSwappedDouble *x*)

DESCRIPTION Swaps the bytes of *x* and returns the resulting value. Bytes are swapped from each low-order position to the corresponding high-order position and vice versa. For example, if the bytes of *x* are numbered from 1 to 8, this function swaps bytes 1 and 8, bytes 2 and 7, bytes 3 and 6, and bytes 4 and 5.

SEE ALSO **NSSwapLongLong, NSSwapFloat**

NSSwapFloat

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedFloat **NSSwapFloat**(NSSwappedFloat *x*)

DESCRIPTION Swaps the bytes of *x* and returns the resulting value.Bytes are swapped from each low-order position to the corresponding high-order position and vice versa. For example, if the bytes of *x* are numbered from 1 to 4, this function swaps bytes 1 and 4, and swaps bytes 2 and 3.

SEE ALSO **NSSwapLong, NSSwapDouble**

NSSwapHostDoubleToBig

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedDouble **NSSwapHostDoubleToBig**(double *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to big-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapDouble** to perform the swap.

SEE ALSO **NSSwapBigDoubleToHost, NSSwapHostDoubleToLittle**

NSSwapHostDoubleToLittle

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedDouble **NSSwapHostDoubleToLittle**(double *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to little-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapDouble** to perform the swap.

SEE ALSO **NSSwapLittleDoubleToHost, NSSwapHostDoubleToBig**

NSSwapHostFloatToBig

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedFloat **NSSwapHostFloatToBig**(float *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to big-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapFloat** to perform the swap.

SEE ALSO **NSSwapBigFloatToHost, NSSwapHostFloatToLittle**

NSSwapHostFloatToLittle

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedFloat **NSSwapHostFloatToLittle**(float *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to little-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapFloat** to perform the swap.

SEE ALSO **NSSwapLittleFloatToHost, NSSwapHostFloatToBig**

NSSwapHostIntToBig

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned int **NSSwapHostIntToBig**(unsigned int *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to big-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapInt** to perform the swap.

SEE ALSO **NSSwapBigIntToHost, NSSwapHostIntToLittle**

NSSwapHostIntToLittle

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned int **NSSwapHostIntToLittle**(unsigned int *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to little-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapInt** to perform the swap.

SEE ALSO **NSSwapLittleIntToHost, NSSwapHostIntToBig**

NSSwapHostLongLongToBig

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long long **NSSwapHostLongLongToBig**(unsigned long long *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to big-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapLongLong** to perform the swap.

SEE ALSO **NSSwapBigLongLongToHost, NSSwapHostLongLongToLittle**

NSSwapHostLongLongToLittle

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long long **NSSwapHostLongLongToLittle**(unsigned long long *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to little-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapLongLong** to perform the swap.

SEE ALSO **NSSwapLittleLongLongToHost, NSSwapHostLongLongToBig**

NSSwapHostLongToBig

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long **NSSwapHostLongToBig**(unsigned long *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to big-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapLong** to perform the swap.

SEE ALSO **NSSwapBigLongToHost, NSSwapHostLongToLittle**

NSSwapHostLongToLittle

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long **NSSwapHostLongToLittle**(unsigned long *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to little-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapLong** to perform the swap.

SEE ALSO **NSSwapLittleLongToHost, NSSwapHostLongToBig**

NSSwapHostShortToBig

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned short **NSSwapHostShortToBig**(unsigned short *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to big-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapShort** to perform the swap.

SEE ALSO **NSSwapBigShortToHost, NSSwapHostShortToLittle**

NSSwapHostShortToLittle

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned short **NSSwapHostShortToLittle**(unsigned short *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to little-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapShort** to perform the swap.

SEE ALSO **NSSwapLittleShortToHost, NSSwapHostShortToBig**

NSSwapInt

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned int **NSSwapInt** (unsigned int *inv*)

DESCRIPTION Swaps the bytes of *inv* and returns the resulting value. Bytes are swapped from each low-order position to the corresponding high-order position and vice versa. For example, if the bytes of *inv* are numbered from 1 to 4, this function swaps bytes 1 and 4, and swaps bytes 2 and 3.

SEE ALSO **NSSwapShort, NSSwapLong, NSSwapLongLong**

NSSwapLittleDoubleToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS double **NSSwapLittleDoubleToHost**(NSSwappedDouble *x*)

DESCRIPTION Converts the little-endian formatted value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapDouble** to perform the swap.

SEE ALSO **NSSwapHostDoubleToLittle, NSSwapBigDoubleToHost, NSConvertSwappedDoubleToHost**

NSSwapLittleFloatToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS float **NSSwapLittleFloatToHost**(NSSwappedFloat *x*)

DESCRIPTION Converts the little-endian formatted value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapFloat** to perform the swap.

SEE ALSO **NSSwapHostFloatToLittle, NSSwapBigFloatToHost, NSConvertSwappedFloatToHost**

NSSwapLittleIntToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned int **NSSwapLittleIntToHost**(unsigned int *x*)

DESCRIPTION Converts the little-endian formatted value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapInt** to perform the swap.

SEE ALSO **NSSwapHostIntToLittle, NSSwapBigIntToHost**

NSSwapLittleLongLongToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long long **NSSwapLittleLongLongToHost**(unsigned long long *x*)

DESCRIPTION Converts the little-endian formatted value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapLongLong** to perform the swap.

SEE ALSO **NSSwapHostLongLongToLittle, NSSwapBigLongLongToHost**

NSSwapLittleLongToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long **NSSwapLittleLongToHost**(unsigned long *x*)

DESCRIPTION Converts the little-endian formatted value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapLong** to perform the swap.

SEE ALSO **NSSwapHostLongToLittle, NSSwapBigLongToHost, NSSwapLong**

NSSwapLittleShortToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned short **NSSwapLittleShortToHost**(unsigned short *x*)

DESCRIPTION Converts the little-endian formatted value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapShort** to perform the swap.

SEE ALSO **NSSwapHostShortToLittle, NSSwapBigShortToHost**

NSSwapLong

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long **NSSwapLong**(unsigned long *inv*)

DESCRIPTION Swaps the bytes of *inv* and returns the resulting value. Bytes are swapped from each low-order position to the corresponding high-order position and vice versa. For example, if the bytes of *inv* are numbered from 1 to 4, this function swaps bytes 1 and 4, and swaps bytes 2 and 3.

SEE ALSO **NSSwapLongLong, NSSwapInt, NSSwapFloat**

NSSwapLongLong

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long long **NSSwapLongLong**(unsigned long long *inv*)

DESCRIPTION Swaps the bytes of *inv* and returns the resulting value. Bytes are swapped from each low-order position to the corresponding high-order position and vice versa. For example, if the bytes of *inv* are numbered from 1 to 8, this function swaps bytes 1 and 8, bytes 2 and 7, bytes 3 and 6, and bytes 4 and 5.

SEE ALSO **NSSwapLong, NSSwapDouble**

NSSwapShort

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned short **NSSwapShort** (unsigned short *inv*)

DESCRIPTION Swaps the low-order and high-order bytes of *inv* and returns the resulting value.

SEE ALSO **NSSwapInt, NSSwapLong**

NSTemporaryDirectory

SUMMARY This function returns the temporary directory on the user's system.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSString ***NSTemporaryDirectory**(void)

DESCRIPTION Returns a string containing the path of the current temporary directory. If no such directory is currently available, this function returns **nil**.

SEE ALSO **NSStandardApplicationPaths, NSStandardLibraryPaths, NSHomeDirectory**

NSUnionRange

SUMMARY This function returns the intersection of the specified ranges.

DECLARED IN Foundation/NSRange.h

SYNOPSIS NSRange **NSUnionRange(**NSRange *range1*, NSRange *range2***)**

DESCRIPTION **NSUnionRange** returns a range covering all indices in and between *range1* and *range2*. If one range is completely contained in the other, the returned range is equal to the larger range.

SEE ALSO **NSIntersectionRange**

NSUnionRect

SUMMARY This function calculates the union of two rectangles.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSUnionRect**(NSRect *aRect*, NSRect *bRect*)

DESCRIPTION **NSUnionRect** returns the smallest rectangle that completely encloses both *aRect* and *bRect*. If one of the rectangles has zero (or negative) width or height, a copy of the other rectangle is returned; but if both have zero (or negative) width or height, the returned rectangle has its origin at (0.0, 0.0) and has zero width and height.

SEE ALSO **NSIntersectionRect**

NSUserName

SUMMARY This function gets information about a user.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSString ***NSUserName(**void**)**

DESCRIPTION **NSUserName** returns the logon name of the current user.

SEE ALSO **NSFullUserName, NSHomeDirectory, NSHomeDirectoryForUser**

NSWidth

SUMMARY This function returns the width of the specified rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSWidth**(NSRect *aRect*)

DESCRIPTION **NSWidth** returns the width of *aRect*.

SEE ALSO **NSMaxX, NSMaxY, NSMidX, NSMidY, NSMinX, NSMinY, NSHeight**

NSZoneCalloc

SUMMARY This function allocates memory in a zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void ***NSZoneCalloc(**NSZone **zone*, unsigned int *numElems*, unsigned int *byteSize***)**

DESCRIPTION **NSZoneCalloc** allocates enough memory from *zone* for *numElems* elements, each with a size *numBytes* bytes, and returns a pointer to the allocated memory. The memory is initialized with zeros. This function returns **nil** if it was unable to allocate the requested memory.

SEE ALSO **NSDefaultMallocZone, NSRecycleZone, NSZoneFree, NSZoneMalloc, NSZoneRealloc**

NSZoneFree

SUMMARY This function deallocates a block of memory in the specified zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void **NSZoneFree**(NSZone **zone*, void **pointer*)

DESCRIPTION Returns memory to the zone from which it was allocated. The standard C function **free** does the same, but spends time finding which zone the memory belongs to.

SEE ALSO **NSRecycleZone, NSZoneMalloc, NSZoneCalloc, NSZoneRealloc**

NSZoneFromPointer

SUMMARY This function gets the zone for a given block of memory

DECLARED IN Foundation/NSZone.h

SYNOPSIS NSZone ***NSZoneFromPointer**(void **pointer*)

DESCRIPTION Returns the zone for the block of memory indicated by *pointer*, or NULL if the block was not allocated from a zone. The pointer must be one that was returned by a prior call to an allocation function.

RETURN Returns the zone for the indicated block of memory, or NULL if the block was not allocated from a zone.

SEE ALSO **NSZoneCalloc, NSZoneMalloc, NSZoneRealloc**

NSZoneMalloc

SUMMARY This function allocates memory in a zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void ***NSZoneMalloc(**NSZone **zone*, unsigned int *size***)**

DESCRIPTION **NSZoneMalloc** allocates *size* bytes in *zone*, and returns a pointer to the allocated memory. This function returns **nil** if it was unable to allocate the requested memory.

SEE ALSO **NSDefaultMallocZone, NSRecycleZone, NSZoneFree, NSZoneCalloc, NSZoneRealloc**

NSZoneName

SUMMARY This function returns the name of the specified zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS NSString ***NSZoneName(**NSZone **zone***)**

DESCRIPTION Returns a string containing the name associated with *zone*, If *zone* is **nil**, the default zone is used. If no name is associated with *zone*, the returned string is empty.

SEE ALSO **NSSetZoneName**

NSZoneRealloc

SUMMARY This function allocates memory in a zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void ***NSZoneRealloc(**NSZone **zone*, void **ptr*, unsigned int *size***)**

DESCRIPTION **NSZoneRealloc** changes the size of the block of memory pointed to by *ptr* to *size* bytes. It may allocate new memory to replace the old, in which case it moves the contents of the old memory block to the new block, up to a maximum of *size* bytes. *ptr* may be NULL. This function returns **nil** if it was unable to allocate the requested memory.

SEE ALSO **NSDefaultMallocZone, NSRecycleZone, NSZoneFree, NSZoneCalloc, NSZoneMalloc**

NSCoding

Adopted By: Various OpenStep classes

Declared In: Foundation/NSObject.h

Protocol Description

The NSCoding protocol declares the two methods that a class must implement so that instances of that class can be encoded and decoded. This capability provides the basis for archiving (where objects and other structures are stored on disk) and distribution (where objects are copied to different address spaces). See the NSCoder and NSArchiver class specifications for an introduction to coding.

In keeping with object-oriented design principles, an object being encoded or decoded is responsible for encoding and decoding its instance variables. A coder instructs the object to do so by invoking **encodeWithCoder:** or **initWithCoder:**. **encodeWithCoder:** instructs the object to encode its instance variables to the coder provided; an object can receive this method any number of times. **initWithCoder:** instructs the object to initialize itself from data in the coder provided; as such, it replaces any other initialization method and is only sent once per object. Any object class that should be codable must adopt the NSCoding protocol and implement its methods.

When an object receives an **encodeWithCoder:** message, it should encode all of its vital instance variables, after sending a message to **super** if its superclass also conforms to the NSCoding protocol. An object doesn't have to encode all of its instance variables. Some values may not be important to reestablish and others may be derivable from related state upon decoding. Other instance variables should be encoded only under certain conditions (for example, with **encodeConditionalObject:**, as described in the NSArchiver class specification).

For example, suppose you were creating a fictitious MapView class that displays a legend and a map at various magnifications. The MapView class defines several instance variables, including the name of the map and the current magnification. The MapView class also contains instance variables for several related views. The **encodeWithCoder:** method of MapView might look like the following:

```
- (void)encodeWithCoder:(NSCoder *)coder
{
    [super encodeWithCoder:coder];
    [coder encodeValueOfObjCType:@encode(char *) at:mapName];
    [coder encodeValueOfObjCType:@encode(unsigned int) at:&magnification];
    [coder encodeObject:legendView];
    [coder encodeConditionalObject:auxiliaryView];
    return;
}
```

This example assumes that the superclass of MapView also supports the NSCoding protocol. If the superclass of your class does not support NSCoding, you should omit the line that invokes super's **encodeWithCoder:** method.

encodeValueOfObjCType:at: and **encodeObject:** are coder methods that you can use to encode instance variables of your class. You can use these and other methods of the coder to encode id's, scalars, C arrays, structs, strings, and pointers to any of these types. The coder also defines corresponding methods for decoding values. See the NSCoder, NSArchiver, and NSUnarchiver class specifications for a list of methods.

The **@encode()** compiler directive generates an Objective-C type code from a type expression. See *Object-Oriented Programming and the Objective-C Language* for more information.

Similarly, in **initWithCoder:** the object should first send a message to **super** (if appropriate) to initialize inherited instance variables, and then it should decode and initialize its own. MapView's implementation of **initWithCoder:** might look like this:

```
- (id)initWithCoder:(NSCoder *)coder
{
    self = [super initWithCoder:coder];
    [coder decodeValueOfObjCType:@encode(char *) at:mapName];
    [coder decodeValueOfObjCType:@encode(unsigned int) at:&magnification];
    legendView = [[coder decodeObject] retain];
    auxiliaryView = [[coder decodeObject] retain];
    return self;
}
```

Note the assignment of the return value of **initWithCoder:** to **self** in the example above. This is done in the subclass because the superclass, in its implementation of **initWithCoder:**, may decide to return a object other than itself. If the superclass of your class does not support NSCoding, you should invoke super's designated initializer instead of **initWithCoder:**.

Making Substitutions During Coding

During encoding or decoding a coder object invokes methods that allow the object being coded to substitute a replacement class or instance for itself. This allows archives to be shared among implementations with different class hierarchies or simply different versions of a class (for example, class clusters take advantage of this feature). It also allows classes that should maintain unique instances to enforce this policy on decoding (for example, there need only be a single NSFont instance for a given typeface and size).

Substitution methods are declared by NSObject, and come in two flavors: generic and specialized. The generic methods are these:

Method	Typical Use
classForCoder	Allows an object, before being encoded, to substitute a class other than its own. For example, the private subclasses of a class cluster substitute the name of their public superclass when being archived.
replacementObjectForCoder:	Allows an object, before being encoded, to substitute another instance in its place.
awakeAfterUsingCoder:	Allows an object, after being decoded, to substitute another object for itself. For example, an object that represents a font might, upon being decoded, release itself and return an existing object having the same font description as itself. In this way, redundant objects can be eliminated.

The specialized substitution methods are analogous to **classForCoder** and **replacementObjectForCoder:** , but they're designed for (and invoked by) a specific, concrete coder subclass. NSArchiver invokes **classForArchiver:** and **replacementObjectForArchiver:**, while NSPortCoder invokes **classForPortCoder** and **replacementObjectForPortCoder:**. (There isn't a specialized version of **awakeAfterUsingCoder:**.) By implementing these specialized methods, your class can base its coding behavior on the specific coder class being used. For more information on these methods, see their method descriptions in the NSObject class specification, as well as the class description in the NSPortCoder class specification.

Method Types

Encoding and decoding objects

– encodeWithCoder:
– initWithCoder:

Instance Methods

encodeWithCoder:

– (void)**encodeWithCoder:**(NSCoder *)*encoder*

Encodes the receiver using *encoder*.

initWithCoder:

– (id)**initWithCoder:**(NSCoder *)*decoder*

Initializes a newly allocated instance from data in *decoder*. Returns **self**.

NSCopying

Adopted By: Various OpenStep classes

Declared In: Foundation/NSObject.h

Protocol Description

The NSCopying protocol declares a method for providing functional copies of an object. The exact meaning of “copy” can vary from class to class, but a copy must be a functionally independent object with values identical to the original at the time the copy was made. A copy produced with NSCopying is implicitly retained by the sender, who is responsible for releasing it.

NSCopying declares one method, **copyWithZone:**, but copying is commonly invoked with the convenience method **copy**. The **copy** method is defined for all NSObjects and simply invokes **copyWithZone:** with the default zone.

Using NSCopying

NSCopying is frequently used to copy *value* objects—objects that represent attributes. C-type variables can usually be substituted for value objects, but value objects have the advantage of encapsulating convenient utilities for common manipulations. For example, NSString objects are used instead of character pointers because they encapsulate encoding and storage. Despite NSString functionality, the role played by NSStrings parallels the role played by character pointers.

When value objects are passed as method arguments or returned from a method, it is common to use a copy instead of the object itself. For example, consider the following method for assigning a string to an object’s **name** instance variable.

```
- (void)setName:(NSString *)aName
{
    [name autorelease];
    name = [aName copy];
}
```

Storing a copy of **aName** has the effect of producing an object that’s independent of the original, but has the same contents. Subsequent changes to the copy don’t affect the original, and changes to the original don’t affect the copy. Similarly, it is common to return a copy of an instance variable instead of the instance variable itself. For example, this method returns a copy of the **name** instance variable:

```
- (NSString *)name
{
    return [[name copy] autorelease];
}
```

Implementing NSCopying

There are two basic approaches to creating copies. You can use **alloc** and **init...**, or you can use **NSCopyObject()**. To choose the one that's right for your class, you need to consider the following questions:

- "What kind of copying—deep or shallow—does your class need?"
- "Does your class's superclass implement NSCopying?"
- "Are you familiar with the implementations of your class's superclasses?"

These areas are described in the following sections.

What kind of copying—deep or shallow—does your class need?

Generally, copying an object involves creating a new instance and initializing it with the values in the original object. Copying the values for non-pointer instance variables, such as booleans, integers, and floating points, is straightforward. When copying pointer instance variables there are two approaches. One approach, called a *shallow copy,* copies the pointer value from the original object into the copy. Thus, the original and the copy share referenced data. The other approach, called a *deep copy,* duplicates the data referenced by the pointer and assigns it to the copy's instance variable.

The implementation of an instance variable's set method should reflect the kind of copying you need to use. You should deeply copy the instance variable if the corresponding set method copies the new value as in this method:

```
- (void)setMyVariable:(id)newValue
{
    [myVariable autorelease];
    myVariable = [newValue copy];
}
```

You should shallowly copy the instance variable if the corresponding set method retains the new value as in this method:

```
- (void)setMyVariable:(id)newValue
{
    [myVariable autorelease];
    myVariable = [newValue retain];
}
```

Similarly, you should shallowly copy the instance variable if its set method simply assigns the new value to the instance variable without copying or retaining it as in this method:

```
- (void)setMyVariable:(id)newValue
{
    myVariable = newValue;
}
```

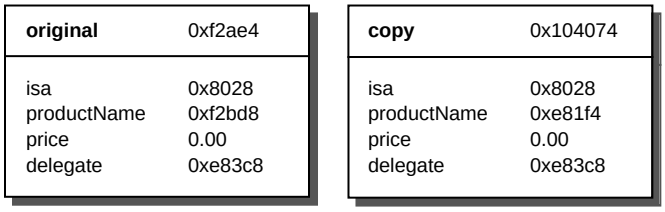
To produce a copy of an object that's truly independent of the original, the entire object must be deeply copied. Every instance variable must be duplicated. If the instance variables themselves have instance variables, those too must be duplicated, and so on. In many cases, a mixed approach is more useful. Pointer instance variables that can be thought of as data containers are generally deeply copied, while more sophisticated instance variables like delegates are shallowly copied.

For example, a Product class adopts NSCopying. Product instances have a name, a price, and a delegate as declared in this interface.

```
@interface Product : NSObject <NSCopying>
{
    NSString *productName;
    float price;
    id delegate;
}

@end
```

Copying a Product instance produces a deep copy of **productName** because it represents a flat data value. On the other hand, the **delegate** instance variable is a more complex object capable of functioning properly for both Products. The copy and the original should therefore share the delegate. The following figure represents the images of a Product instance and a copy in memory.



The different pointer values for **productName** illustrate that the original and the copy each have their own **productName** string object. The pointer values for **delegate** are the same, indicating that the two product objects share the same object as their delegate.

Does your class's superclass implement NSCopying?

If the superclass does not implement NSCopying, your class's implementation will have to copy the instance variables it inherits as well as those declared in your class. Generally, the safest way to do this is by using **alloc**, **init...**, and set methods. On the other hand, if your class inherits NSCopying behavior, its implementation only has to copy instance variables declared in your class. It invokes the superclass's implementation to copy inherited instance variables.

Are you familiar with the implementations of your class's superclasses?

If your class inherits NSCopying behavior, how you handle the new instance variables in **copyWithZone:** depends on your familiarity with the superclass's implementation. There are essentially two ways to make a copy of an object, using **alloc** and **init...** or using the function **NSCopyObject()**. If the superclass used or might have used **NSCopyObject()**, you must handle instance variables differently than you would otherwise.

Using the alloc, init... Approach

If a class does not inherit NSCopying behavior, you should implement **copyWithZone:** using **alloc**, **init...**, and set methods. For example, an implementation of **copyWithZone:** for the Product class described above might be implemented in the following way:

```
- (id)copyWithZone:(NSZone *)zone
{
    Product *copy = [[Product alloc]
           initWithProductName:[self productName]
           price:[self price]];
    [copy setDelegate:[self delegate]];

    return copy;
}
```

Because implementation details associated with inherited instance variables are encapsulated in the superclass, it is generally better to implement NSCopying with the **alloc**, **init...** approach. Doing so uses policy implemented in set methods to determine the kind of copying needed of instance variables.

Using NSCopyObject()

When a class inherits NSCopying behavior, you must consider the possibility that the superclass's implementation uses **NSCopyObject()**. **NSCopyObject()** creates an exact shallow copy of an object by copying instance variable values but not the data they point to. For example, NSCell's implementation of **copyWithZone:** could be defined in the following way.

```
- (id)copyWithZone:(NSZone *)zone
{
    NSCell *cellCopy = NSCopyObject(self, 0, zone);
    /* Assume that other initialization takes place here. */

    cellCopy->image = nil;
    [cellCopy setImage:[self image]];

    return cellCopy;
}
```

In the implementation above, **NSCopyObject()** creates an exact shallow copy of the original cell. This behavior is desirable for copying instance variables that aren't pointers or are pointers to non-retained data that is shallowly copied. Pointer instance variables for retained objects need additional treatment.

In the **copyWithZone:** example above, **image** is a pointer to a retained object. The policy to retain the image is reflected in the following implementation of the **setImage:** accessor method.

```
- (void)setImage:(NSImage *)anImage
{
    [image autorelease];
    image = [anImage retain];
}
```

Notice that **setImage:** autoreleases **image** before it reassigns it. If the above implementation of **copyWithZone:** hadn't explicitly set the copy's **image** instance variable to **nil** before invoking **setImage:**, the image referenced by the copy and the original would be released without a corresponding retain.

Even though **image** points to the right object, it is conceptually uninitialized. Unlike the instance variables that are created with **alloc** and **init...**, these uninitialized variables aren't **nil**-valued. You should explicitly assign initial values to these variables before using them. In this case, **cellCopy**'s **image** instance variable is set to **nil**, then it is set using the **setImage:** method.

The effects of **NSCopyObject()** extend to a subclass's implementation. For example, an implementation of NSSliderCell could copy a new **titleCell** instance variable in the following way.

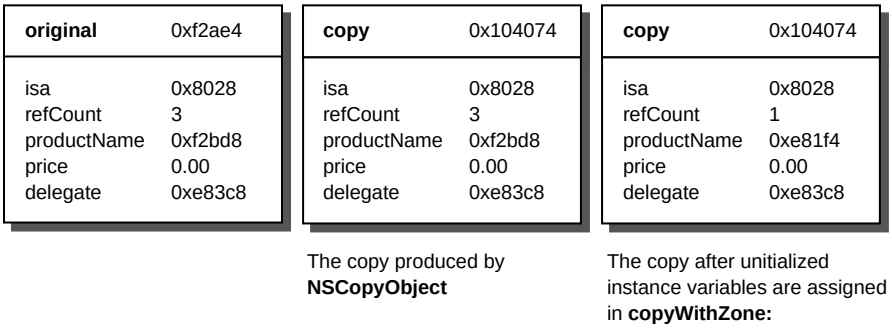
```
- (id)copyWithZone:(NSZone *)zone
{
    NSSliderCell *cellCopy = [super copyWithZone:zone];
    /* Assume that other initialization takes place here. */

    cellCopy->titleCell = nil;
    [cellCopy setTitleCell:[self titleCell]];

    return cellCopy;
}
```

The superclass's **copyWithZone:** method is invoked to copy inherited instance variables. When you invoke a superclass's **copyWithZone:** method, assume that new object instance variables are uninitialized if there's any chance that the superclass implementation uses **NSCopyObject()**. Explicitly assign a value to them before using them. In this example, **titleCell** is explicitly set to **nil** before **setTitleCell:** is invoked.

The implementation of an object's retain count is another consideration when using **NSCopyObject()**. If an object stores its retain count in an instance variable, the implementation of **copyWithZone:** must correctly initialize the copy's retain count. The following figure illustrates the process.



The copy produced by **NSCopyObject**

The copy after unitialized instance variables are assigned in **copyWithZone:**

The first object represents a Product instance in memory. The value in **refCount** indicates that the instance has been retained three times. The second object is a copy of the Product instance produced with **NSCopyObject()**. Its **refCount** value matches the original. The third object represents the copy returned from **copyWithZone:** after **refCount** is correctly initialized. After **copyWithZone:** creates the copy with **NSCopyObject()**, it assigns the value 1 to the **refCount** instance variable. The sender of **copyWithZone:** implicitly retains the copy and is responsible for releasing it.

NSCopying and Immutable Classes

Where the concept “immutable vs. mutable” applies to an object, NSCopying produces immutable copies whether the original is immutable or not. See the NSMutableCopying protocol for details on making mutable copies.

Immutable classes can implement NSCopying very efficiently. Since immutable objects don’t change, there is no need to duplicate them. Instead, NSCopying can be implemented to **retain** the original. For example, **copyWithZone:** for an immutable string class can be implemented in the following way.

```
- (id)copyWithZone:(NSZone *)zone
{
    return [self retain];
}
```

Summary

- Implement NSCopying using **alloc** and **init...** in classes that don’t inherit **copyWithZone:**.
- Implement NSCopying by invoking the superclass’s **copyWithZone:** when NSCopying behavior is inherited. If the superclass implementation might use **NSCopyObject()**, make explicit assignments to pointer instance variables for retained objects.
- Implement NSCopying by retaining the original instead of creating a new copy when the class and its contents are immutable.

Instance Methods

copyWithZone:

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

Returns a new instance that's a copy of the receiver. Memory for the new instance is allocated from *zone*, which may be NULL. If *zone* is NULL, the new instance is allocated from the default zone, which is returned from **NSDefaultMallocZone()**. The returned object is implicitly retained by the sender, who is responsible for releasing it. The copy returned is immutable if the consideration "immutable vs. mutable" applies to the receiving object; otherwise the exact nature of the copy is determined by the class.

See also: – **mutableCopyWithZone:** (NSMutableCopying protocol), – **copy** (NSObject)

NSDecimalNumberBehaviors

Adopted By: NSDecimalNumberHandler

Declared In: Foundation/NSDecimalNumber.h

Protocol Description

The NSDecimalBehaviors protocol declares three methods that control the discretionary aspects of working with NSDecimalNumbers. The **scale** and **roundingMode** methods determine the precision of NSDecimalNumber's return values, and the way in which those values should be rounded to fit that precision. The **exceptionDuringOperation:error:leftOperand:rightOperand:** determines the way in which an NSDecimalNumber should handle different calculation errors.

For an example of a class that adopts the NSDecimalBehaviors protocol, see the specification for NSDecimalNumberHandler.

Method Types

Rounding

– roundingMode
– scale

Handling errors

– exceptionDuringOperation:error:leftOperand:rightOperand:

Instance Methods

exceptionDuringOperation:error:leftOperand:rightOperand:

– (NSDecimalNumber *)**exceptionDuringOperation:**(SEL)*method*
error:(NSCalculationError)*error*
leftOperand:(NSDecimalNumber *)*leftOperand*
rightOperand:(NSDecimalNumber *)*rightOperand*

Specifies what an NSDecimalNumber will do when, in the course of applying *method* to *leftOperand* and *rightOperand*, it encounters *error*.

There are four possible values for *error*. The first three have to do with limits on NSDecimalNumber's ability to represent decimal numbers. An NSDecimalNumber can represent any number that can be

expressed as *mantissa* **x** 10**exponent**, where *mantissa* is a decimal integer up to 38 digits long, and *exponent* is between -256 and 256. If these limits are exceeded, the NSDecimalNumber returns one of the following errors:

- NSCalculationLossOfPrecision. The number can't be represented in 38 significant digits.
- NSCalculationOverflow. The number is too large to represent.
- NSCalculationUnderflow. The number is too small to represent.

The last error is simpler:

- NSCalculationDivideByZero. The caller tried to divide by zero.

In implementing **exceptionDuringOperation:error:leftOperand:rightOperand**, you can handle each of these errors in several ways:

- Raise an exception. For an explantion of exceptions, see the NSException class description in the *Foundation Framework Reference*.
- Return **nil**. The calling *method* will return its value as though no error had occurred. If *error* is NSCalculationLossOfPrecision, *method* will return an imprecise value—that is, one constrained to 38 significant digits. If *error* is NSCalculationUnderflow or NSCalculationOverflow, *method* will return NSDecimalNumber's **notANumber**. You shouldn't return **nil** if *error* is NSDivideByZero.
- Correct the error and return a valid NSDecimalNumber. The calling method will use this as its own return value.

roundingMode

– (NSRoundingMode)**roundingMode**

Returns the way that NSDecimalNumber's **decimalNumberBy...** methods round their return values. There are four possible NSRoundingModes:

- NSRoundDown. The methods round their return values down.
- NSRoundUp. The methods round their return values up.
- NSRoundPlain. The methods round to the closest possible return value. When they are caught halfway between two positive numbers, they round up; when caught between two negative numbers, they round down.
- NSRoundBankers. The methods round to the closest possible return value. When they are caught halfway between two possibilities, they return the possibility whose last digit is even. In practice, this means that, over the long run, numbers will be rounded up as often as they are rounded down; there will be no systematic bias.

The rounding mode only matters if the **scale** method sets a limit on the precision of NSDecimalNumber return values. It has no effect if **scale** returns NSDecimalNoScale.

Assuming that **scale** returns 1, the NSRoundingMode has the following effects on various original values:

Original value	NSRoundDown	NSRoundUp	NSRoundPlain	NSRoundBankers
1.24	1.2	1.3	1.2	1.2
1.26	1.2	1.3	1.3	1.3
1.25	1.2	1.3	1.3	1.2
1.35	1.3	1.4	1.4	1.4
-1.35	-1.4	-1.3	-1.4	-1.4

scale

– (short)**scale**

Limits the precision of the values returned by NSDecimalNumber's **decimalNumberBy...** methods.

Specifically, **scale** returns the number of digits allowed after the decimal separator. If **scale** returns a negative value, it affects the digits *before* the decimal separator as well. If **scale** returns NSDecimalNoScale, the number of digits is unlimited.

Assuming that **roundingMode** returns NSRoundPlain, different values of **scale** have the following effects on the number 123.456:

Scale	Return value
NSDecimalNoScale	123.456
2	123.45
0	123
-2	100

NSLocking

Adopted By: NSConditionLock
NSLock
NSRecursiveLock

Declared In: Foundation/NSLock.h

Protocol Description

The NSLocking protocol declares the elementary methods adopted by classes that define lock objects. A lock object is used to coordinate the actions of multiple threads of execution within a single application. By using a lock object, an application can protect critical sections of code from being executed simultaneously by separate threads, thus protecting shared data and other shared resources from corruption.

For example, consider a multithreaded application in which each thread updates a shared counter. If two threads simultaneously fetch the current value into local storage, increment it, and then write the value back, the counter will be incremented only once, losing one thread's contribution. However, if the code that manipulates the shared data (the *critical section* of code) must be locked before being executed, only one thread at a time can perform the updating operation, and collisions are prevented.

A lock object is either locked or unlocked. You *acquire* a lock by sending the object a **lock** message, thus putting the object in the locked state. You *relinquish* a lock by sending an **unlock** message, and thus putting the object in the unlocked state. (The NEXTSTEP classes that adopt this protocol define additional ways to acquire and relinquish locks.)

The **lock** method as declared in this protocol is *blocking*. That is, the thread that sends a **lock** message is blocked from further execution until the lock is acquired (presumably because some other thread relinquishes its lock). Classes that adopt this protocol typically add methods that offer nonblocking alternatives.

These NEXTSTEP classes conform to the NSLocking protocol:

Class	Adds these features to the basic protocol
NSLock	A nonblocking lock method; the ability to limit the duration of a locking attempt.
NSConditionLock	The ability to postpone entry to a critical section until a condition is met.
NSRecursiveLock	The ability for a single thread to acquire a lock more than once without deadlocking.

The locking mechanism that these classes use causes a thread to sleep while waiting to acquire a lock rather than to poll the system constantly. Thus, lock objects can be used to lock time-consuming operations without causing system performance to degrade. See the class specifications for these classes for further information on their behavior and usage.

There is some performance cost in acquiring a lock, so it's best to avoid the overhead if possible. An application developer has control over whether the application will execute with multiple threads, so it's clear when locking is appropriate. A library developer doesn't necessarily know whether library code will execute in a multithreaded environment. In this case, it's best to test whether the code is executing in a multithreaded environment before attempting to acquire a lock. The following example illustrates how this is done.

Assume your application uses a Counter object to record various operations. Here's one design that lets the Counter know whether it is multithreaded:

```
+ (void)initialize
{
    if ([NSThread isMultiThreaded]) {
        [self taskNowMultiThreaded:nil];
    } else {
        [[NSNotificationCenter defaultCenter] addObserver:self
            selector:@selector(taskNowMultiThreaded:)
            name:NSWillBecomeMultiThreadedNotification object:nil];
    }
}
```

In the initialize method (which, by definition, is invoked before any Counter objects are created), the Counter class object first checks whether the application has already become multithreaded and if so invokes its own **taskNowMultiThreaded:** method. Otherwise, it registers as an observer of NSWillBecomeMultiThreadedNotification so that **taskNowMultiThreaded:** will be invoked when the application becomes multithreaded.

Counter's **taskNowMultiThreaded:** method creates a lock object that the threads use to coordinate their activities:

```
+ (void)taskNowMultiThreaded:(NSNotification *)event
{
    if (!theLock)
        theLock = [[NSLock alloc] init];
}
```

theLock, a static variable declared in the class implementation file, is assigned a value of **nil** until **taskNowMultiThreaded:** is invoked. Since messages sent to **nil** are permitted and have no effect, code within Counter that acts on shared data can be written like this:

```
[theLock lock];
/* Operate on shared data */
[theLock unlock];
```

With this approach, the overhead associated with lock operations is only incurred if the application is multithreaded. This code, however, raises another issue. What happens if one of the statements between the **lock** and **unlock** messages cause the application to become multithreaded? Then the **unlock** message wouldn't be paired with the preceding **lock**.

In normal usage, locking and unlocking messages are paired. However, as in the example above, it might be convenient to unlock a lock object that hasn't yet been locked. This is permitted with two restrictions. First, you can send an unpaired unlocking message to a lock object as long as the object has never before been locked. Second, of the NEXTSTEP classes that conform to the NSLocking protocol, only NSConditionLock and NSLock allow an unpaired unlocking message. NSRecursiveLock requires locking and unlocking messages to be paired.

NEXTSTEP's locking classes are designed to work in a well-behaved, multithreaded environment: The protection they offer can be subverted by the use of signal handlers. A signal handler can interrupt a thread, execute code that affects shared data, and then let the thread resume without alerting the thread that the application has, in effect, become multithreaded. For this reason, it's recommended that you don't use signal handlers in multithreaded NEXTSTEP applications.

Instance Methods

lock

– (void)**lock**

Attempts to acquire a lock. This method blocks a thread's execution until the lock can be acquired.

An application protects a critical section of code by requiring a thread to acquire a lock before executing the code. Once the critical section is past, the thread relinquishes the lock by invoking **unlock**.

unlock

– (void)**unlock**

Relinquishes a previously acquired lock.

NSMutableCopying

Adopted By: Various OpenStep classes

Declared In: Foundation/NSObject.h

Protocol Description

The NSMutableCopying protocol declares a method for providing mutable copies of an object. Only classes that define an "immutable vs. mutable" distinction should adopt this protocol. Classes that don't define such a distinction should adopt NSCopying instead.

NSMutableCopying declares one method, **mutableCopyWithZone:**, but mutable copying is commonly invoked with the convenience method **mutableCopy**. The **mutableCopy** method is defined for all NSObjects and simply invokes **mutableCopyWithZone:** with the default zone.

See the NSCopying protocol for details on implementing copying behavior.

Instance Methods

mutableCopyWithZone:

– **mutableCopyWithZone:**(NSZone *)*zone*

Returns a new instance that's a mutable copy of the receiver. Memory for the new instance is allocated from *zone*, which may be NULL. If *zone* is NULL, the new instance is allocated from the default zone, which is returned by **NSDefaultMallocZone()**. The returned object is implicitly retained by the sender, who is responsible for releasing it. The copy returned is mutable whether the original is mutable or not.

See also: – **copyWithZone:** (NSCopying protocol), – **mutableCopy** (NSObject)

NSObjCTypeSerializationCallBack

Adopted By: No OpenStep classes

Declared In: Foundation/NSSerialization.h

Protocol Description

An object conforms to the NSObjCTypeSerializationCallBack protocol so that it can intervene in the serialization and deserialization process. The primary purpose of this protocol is to allow for the serialization of objects and other data types that aren't directly supported by OpenStep's serialization facility. (See the NSSerializer class specification for information on serialization.)

NSMutableData declares the method that's used to begin the serialization process:

```
- (void)serializeDataAt:(const void *)data
        ofObjCType:(const char *)type
        context:(id <NSObjCTypeSerializationCallBack>)callback
```

This method can serialize all standard Objective C types (**int**, **float**, character strings, and so on) except for objects, **union**, and **void ***. If, during the serialization process, an object is encountered, the object passed as the callback argument above is asked to provide the serialization.

Suppose that the type being serialized is a structure of this description:

```
struct stockRecord {
    NSString *stockName;
    float value;
};
```

The Objective C type code for this structure is {@f}, so the serialization process begins with this message: (Assume that **theData** is the NSMutableData object that's doing the serialization and **helper** is an object that conforms to the NSObjCTypeSerializationCallBack protocol.)

```
struct stockRecord aRecord = {@"aCompany", 34.7};

[theData serializeDataAt:&aRecord ofObjCType:"{@f}" context:helper];
```

Since the first field of the structure is an unsupported type, the helper object is sent a **serializeObjectAt: ofObjCType:intoData:** message, letting it serialize the object. **helper** might implement the method in this way:

```
- (void)serializeObjectAt:(id *)objectPtr
        ofObjCType:(const char *)type
        intoData:(NSMutableData *)theMutableData
{
    NSString *nameObject;
    char *companyName

    nameObject = *objectPtr;
    companyName = [nameObject cString];

    [theData serializeDataAt:&companyName ofObjCType:@encode(typeof(companyName))
           context:nil];
}
```

The callback object is free to serialize the target object as it wishes. In this case, **helper** simply extracts the company name from the NSString object and then has that character string serialized. Once this callback method finishes executing, the original method (**serializeDataAt:ofObjCType:context:**) resumes execution and serializes the second field of the structure. Since this second field contains a supported type (**float**), the callback method is not invoked again.

Deserialization follows a similar pattern, except in this case NSData declares the central method **deserializeDataAt:ofObjCType:atCursor:context:**. The deserialization of the example structure starts with a message to the NSData object that contains the serialized data:

```
(unsigned *)cursor = 0;

[theData deserializeDataAt:&aRecord ofObjCType:"{@f}" cursor:&cursor
        context:helper];
```

(The cursor argument is a pointer to zero since we're starting at the beginning of the data in the NSData object.)

When this method is invoked, the callback object receives a **deserializeObjectAt:ofObjCType:fromData: atCursor:** message, as declared in this protocol. The callback object can then reestablish the first field of the structure. For example, **helper** might implement the method in this way:

```
- (void) deserializeObjectAt:(id *)objectPtr
        ofObjCType:(const char *)type
        fromData:(NSData *)data
        atCursor:(unsigned *)cursor
{
    char *companyName;

    [theData deserializeDataAt:&companyName ofObjCType:"*" atCursor:cursor
           context:nil];
    *objectPtr = [[NSString stringWithCString:companyName] retain];
}
```

Instance Methods

deserializeObjectAt:ofObjCType:fromData:atCursor:

– (void)**deserializeObjectAt:**(id *)*object*
ofObjCType:(const char *)*type*
fromData:(NSData *)*data*
atCursor:(unsigned *)*cursor*

The implementor of this method decodes the referenced *object* (which should always be of *type* "@") located at the *cursor* position in the *data* object. The decoded object is not autoreleased.

See also: – **deserializeDataAt:ofObjCType:atCursor:context:** (NSData)

serializeObjectAt:ofObjCType:intoData:

– (void)**serializeObjectAt:**(id *)*object*
ofObjCType:(const char *)*type*
intoData:(NSMutableData *)*data*

The implementor of this method encodes the referenced *object* (which should always be of *type* "@") in the *data* object.

See also: – **serializeDataAt:ofObjCType:context:** (NSMutableData)

NSObject

Adopted By: NSObject

Declared In: Foundation/NSObject.h

Protocol Description

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

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

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

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

Method Types

Identifying classes

– class
– superclass

Identifying and comparing objects

– isEqual:
– hash
– self

Determining allocation zones

– zone

Managing reference counts

– retain
– release
– autorelease
– retainCount

Testing class functionality

– respondsToSelector:

Testing inheritance relationships

– isKindOfClass:
– isMemberOfClass:

Testing protocol conformance

– conformsToProtocol:

Describing objects

– description

Sending messages

– performSelector:
– performSelector:withObject:
– performSelector:withObject:withObject:

Identifying proxies

– isProxy

Instance Methods

autorelease

– (id)**autorelease**

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

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

class

– (Class)**class**

Returns the class object for the receiver's class.

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

conformsToProtocol:

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

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

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

description

– (NSString *)**description**

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

hash

– (unsigned)**hash**

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

If a mutable object is added to a collection that uses hash values to determine the object's position in the collection, the value returned by the **hash** method of the object must not change while the object is in the collection. To accomplish this, either the **hash** method must not rely on any of the object's internal state information or you must make sure that the object's internal state information does not change while the object is in the collection. (Note that it can be difficult to know whether or not a given object is in a collection.)

isEqual:

– (BOOL)**isEqual:**(id)*anObject*

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

isKindOfClass:

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

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

```
NSMutableData *myData = [NSMutableData dataWithCapacity:30];
id anArchiver = [[NSArchiver alloc] initForWritingWithMutableData:myData];
if ( [anArchiver isKindOfClass:[NSCoder class]] )
    ...
```

Because instance methods defined in the root class can be sent to class objects, when the receiver is a class object, this method returns YES if *aClass* is NSObject, NO otherwise.

See also: – **isMemberOfClass:**

isMemberOfClass:

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

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

```
NSMutableData *myData = [NSMutableData dataWithCapacity:30];
id anArchiver = [[NSArchiver alloc] initForWritingWithMutableData:myData];
if ([anArchiver isMemberOfClass:[NSCoder class]])
    ...
```

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

See also: – **isKindOfClass:**

isProxy

– (BOOL)**isProxy**

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

performSelector:

– (id)**performSelector:**(SEL)*aSelector*

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

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

However, the **performSelector:** 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 performSelector:myMethod];
```

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

See also: – **performSelector:withObject:**, – **performSelector:withObject:withObject:**

performSelector:withObject:

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

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

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

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

performSelector:withObject:withObject:

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

Sends the receiver an *aSelector* message with *anObject* and *anotherObject* as arguments. If *aSelector* is NULL, an NSInvalidArgumentException is raised. This method is the same as **performSelector:** except

that you can supply two arguments for *aSelector*. *aSelector* should identify a method that can take two arguments of type **id**. For methods with other argument types and return values use NSInvocation.

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

release

– (oneway void)**release**

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

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

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

See also: – **retainCount**

respondsToSelector:

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

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

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

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

retain

– (id)**retain**

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

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

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

```
NSString *systemApps = [[NSString
    stringWithCString:"/System/Applications"] retain];
```

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

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

retainCount

– (unsigned)**retainCount**

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

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

self

– (id)**self**

Returns the receiver.

See also: – **class**

superclass

– (Class)**superclass**

Returns the class object for the receiver's superclass.

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

zone

– (NSZone *)**zone**

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

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

Defined Types

NSBTreeComparator

DECLARED IN Foundation/NSByteStore.h

SYNOPSIS

typedef int **NSBTreeComparator**(NSData *, NSData *, const void *);

NSComparisonResult

DECLARED IN Foundation/NSObject.h

SYNOPSIS

typedef enum _NSComparisonResult {
    NSOrderedAscending = -1,
    NSOrderedSame,
    NSOrderedDescending
} **NSComparisonResult**;

DESCRIPTION

An **NSComparisonResult** indicates how items in a request are ordered, from the first one given in a method invocation or function call to the last (that is, left-to-right in code).

NSHashTableCallBacks

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS

```
typedef struct {
        unsigned (*hash)(NSHashTable *table, const void *);
        BOOL (*isEqual)(NSHashTable *table, const void *, const void *);
        void (*retain)(NSHashTable *table, const void *);
        void (*release)(NSHashTable *table, void *);
        NSString *(*describe)(NSHashTable *table, const void *);
} NSHashTableCallBacks;
```

NSMapTableKeyCallBacks

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS

```
typedef struct {
        unsigned (*hash)(NSMapTable *table, const void *);
        BOOL (*isEqual)(NSMapTable *table, const void *, const void *);
        void (*retain)(NSMapTable *table, const void *);
        void (*release)(NSMapTable *table, void *);
        NSString *(*describe)(NSMapTable *table, const void *);
         const void *notAKeyMarker;
} NSMapTableKeyCallBacks;
```

NSMapTableValueCallBacks

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS

```
typedef struct {
        void (*retain)(NSMapTable *table, const void *);
        void (*release)(NSMapTable *table, void *);
        NSString *(*describe)(NSMapTable *table, const void *);
} NSMapTableValueCallBacks;
```

NSNotificationCoalescing

DECLARED IN Foundation/NSNotificationQueue.h

SYNOPSIS

```
typedef enum {
    NSNotificationNoCoalescing = 0,
    NSNotificationCoalescingOnName = 1,
    NSNotificationCoalescingOnSender = 2,
} NSNotificationCoalescing;
```

NSObjCValue

DECLARED IN Foundation/NSInvocation.h

SYNOPSIS

```
typedef struct {
    enum _NSObjCValueType type;
    union {
        char charValue;
        short shortValue;
        long longValue;
        long long longlongValue;
        float floatValue;
        double doubleValue;
        SEL selectorValue;
        id objectValue;
        void *pointerValue;
        void *structLocation;
        char *cStringLocation;
    } value;
} NSObjCValue;
```

NSPoint

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS

```
typedef struct _NSPoint {
    float x;
    float y;
} NSPoint;
```

NSPosixFileActivity

DECLARED IN Foundation/NSPosixFileDescriptor.h

SYNOPSIS

```
typedef enum {
        NSPosixNoActivity = 0,
        NSPosixReadableActivity = 1,
        NSPosixWritableActivity = 2,
        NSPosixExceptionalActivity = 4
} NSPosixFileActivity;
```

NSPostingStyle

DECLARED IN Foundation/NSNotificationQueue.h

SYNOPSIS

```
typedef enum {
        NSPostWhenIdle = 1,
        NSPostASAP = 2,
        NSPostNow = 3
} NSPostingStyle;
```

NSRange

DECLARED IN Foundation/NSRange.h

SYNOPSIS

```
typedef struct _NSRange {
    unsigned int location;
    unsigned int length;
} NSRange;
```

DESCRIPTION An **NSRange** describes a portion of a series—such as characters in a string or objects in an NSArray. Its **location** member gives the start index (0 is the first, as in C arrays), and its **length** member gives the number of items in the range (and can be zero).

NSRect

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS

```
typedef struct _NSRect {
    NSPoint origin;
    NSSize size;
} NSRect;
```

NSRectEdge

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS

```
typedef enum _NSRectEdge {
    NSMinXEdge = 0,
    NSMinYEdge = 1,
    NSMaxXEdge = 2,
    NSMaxYEdge = 3
} NSRectEdge;
```

DESCRIPTION This enumeration identifies the sides of a rectangle represented by an NSRect.

NSSize

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS

```
typedef struct _NSSize {
    float width;
    float height;
} NSSize;
```

DESCRIPTION The values for width and height should never be negative.

NSStringEncoding

DECLARED IN Foundation/NSString.h

SYNOPSIS typedef unsigned **NSStringEncoding**;

NSTimeInterval

DECLARED IN Foundation/NSDate.h

SYNOPSIS typedef double **NSTimeInterval**;

DESCRIPTION Always in seconds; yields sub-millisecond precision over a range of 10000 years.

NSUncaughtExceptionHandler

DECLARED IN Foundation/NSException.h

SYNOPSIS typedef volatile void **NSUncaughtExceptionHandler**(NSException ***exception**);

Enumerations

NSByteOrder

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS

```
enum NSByteOrder {
        NS_UnknownByteOrder,
        NS_LittleEndian,
        NS_BigEndian
};
```

DESCRIPTION

NSNotFound

DECLARED IN Foundation/NSObjCRuntime.h

SYNOPSIS

```
enum {
        NSNotFound = 0x7fffffff
};
```

DESCRIPTION A method or function return value of **NSNotFound** indicates that the item requested couldn't be found or doesn't exist. It's typically used by various methods and functions that search for items in serial data and return indices, such as characters in a string object or **id**s in an NSArray.

NSOpenStepUnicodeReservedBase

DECLARED IN Foundation/NSCharacterSet.h

SYNOPSIS

```
enum {
    NSOpenStepUnicodeReservedBase = 0xF400
};
```

DESCRIPTION

ObjC Types

DECLARED IN Foundation/NSInvocation.h

SYNOPSIS

```
enum _NSObjCValueType {
    NSObjCNoType = 0,
    NSObjCVoidType = 'v',
    NSObjCCharType = 'c',
    NSObjCShortType = 's',
    NSObjCLongType = 'l',
    NSObjCLonglongType = 'q',
    NSObjCFloatType = 'f',
    NSObjCDoubleType = 'd',
    NSObjCSelectorType = ':',
    NSObjCObjectType = '@',
    NSObjCStructType = '{',
    NSObjCPointerType = '^',
    NSObjCStringType = '*',
    NSObjCArrayType = '[',
    NSObjCUnionType = '(',
    NSObjCBitfield = 'b',
};
```

Search Types

DECLARED IN Foundation/NSString.h

SYNOPSIS

```
enum {
    NSCaseInsensitiveSearch = 1,
    NSLiteralSearch = 2,
    NSBackwardsSearch = 4,
    NSAnchoredSearch = 8
};
```

DESCRIPTION These values represent the options available to many of the string classes' searching and comparison methods. See the NSString class cluster description for details on the effects of these options.

String Encodings

DECLARED IN Foundation/NSString.h

SYNOPSIS

```
enum {
    NSASCIIStringEncoding = 1,
    NSNEXTSTEPStringEncoding = 2,
    NSJapaneseEUCStringEncoding = 3,
    NSUTF8StringEncoding = 4,
    NSISOLatin1StringEncoding = 5,
    NSSymbolStringEncoding = 6,
    NSNonLossyASCIIStringEncoding = 7,
    NSShiftJISStringEncoding = 8,
    NSISOLatin2StringEncoding = 9,
    NSUnicodeStringEncoding = 10,
    NSWindowsCP1251StringEncoding = 11,
    NSWindowsCP1252StringEncoding = 12,
    NSWindowsCP1253StringEncoding = 13,
    NSWindowsCP1254StringEncoding = 14,
    NSWindowsCP1250StringEncoding = 15,
    NSISO2022JPStringEncoding = 21
};
```

DESCRIPTION These values represent the various character encodings supported by the NSString classes. These encodings are documented more fully elsewhere; here are some short descriptions:

Encoding	Description
NSASCIIStringEncoding	Strict 7-bit ASCII encoding within 8-bit chars
NSNEXTSTEPStringEncoding	8-bit ASCII encoding with NEXTSTEP extensions (see Appendix C of the NEXTSTEP General Reference)
NSUnicodeStringEncoding	The canonical Unicode encoding for string objects
NSISOLatin1StringEncoding	8-bit ISO Latin 1 encoding
NSISOLatin2StringEncoding	8-bit ISO Latin 2 encoding
NSSymbolStringEncoding	8-bit Adobe Symbol encoding vector
NSJapaneseEUCStringEncoding	8-bit EUC encoding for Japanese text
NSShiftJISStringEncoding	8-bit Shift-JIS encoding for Japanese text
NSUTF8StringEncoding	An 8-bit representation of Unicode characters, suitable for transmission or storage by ASCII-based systems
NSNonLossyASCIIStringEncoding	7-bit verbose ASCII to represent all Unicode characters.
NSWindowsCP1250StringEncoding	Microsoft Windows codepage 1250; equivalent to WinLatin2
NSWindowsCP1251StringEncoding	Microsoft Windows codepage 1251, encoding Cyrillic characters; equivalent to AdobeStandardCyrillic font encoding
NSWindowsCP1252StringEncoding	Microsoft Windows codepage 1252; equivalent to WinLatin1
NSWindowsCP1253StringEncoding	Microsoft Windows codepage 1253, encoding Greek characters
NSWindowsCP1254StringEncoding	Microsoft Windows codepage 1254, encoding Turkish characters
NSISO2022JPStringEncoding	ISO 2022 Japanese encoding for electronic mail

Global Variables

Archiving Exception Names

DECLARED IN Foundation/NSArchiver.h

SYNOPSIS extern NSString ***NSInconsistentArchiveException**;

Bundle Notification Names

DECLARED IN Foundation/NSBundle.h

SYNOPSIS extern NSString ***NSBundleLoaded**;

Byte Store Exception Names

DECLARED IN Foundation/NSByteStore.h

SYNOPSIS extern NSString ***NSByteStoreLockedException**;
extern NSString ***NSByteStoreVersionException**;
extern NSString ***NSBTreeStoreKeyTooLargeException**;
extern NSString ***NSByteStoreDamagedException**;

Connection Exception Names

DECLARED IN Foundation/NSConnection.h

SYNOPSIS extern NSString ***NSFailedAuthenticationException**;

Connection Notification Names

DECLARED IN Foundation/NSConnection.h

SYNOPSIS extern NSString ***NSConnectionDidDieNotification**;

File Attribute Keys

DECLARED IN Foundation/NSFileManager.h

SYNOPSIS
extern NSString ***NSFileType**;
extern NSString ***NSFileTypeDirectory**;
extern NSString ***NSFileTypeRegular**;
extern NSString ***NSFileTypeSymbolicLink**;
extern NSString ***NSFileTypeSocket**;
extern NSString ***NSFileTypeCharacterSpecial**;
extern NSString ***NSFileTypeBlockSpecial**;
extern NSString ***NSFileTypeUnknown**;
extern NSString ***NSFileSize**;
extern NSString ***NSFileModificationDate**;
extern NSString ***NSFileReferenceCount**;
extern NSString ***NSFileDeviceIdentifier**;
extern NSString ***NSFileOwnerAccountNumber**;
extern NSString ***NSFileGroupOwnerAccountNumber**;
extern NSString ***NSFilePosixPermissions**;
extern NSString ***NSFileSystemNumber**;
extern NSString ***NSFileSystemFileNumber**;

DESCRIPTION Keys to access the file attribute values contained in the NSDictionary returned from NSFileManager's **fileAttributesAtPath:follow:**. See the class specification for NSFileManager for details of usage. **NSFileDeviceIdentifier** is used to access the identifier of a remote device.

File Exception Names

DECLARED IN Foundation/NSPosix.h

SYNOPSIS extern NSString ***NSPosixFileOperationException**;

File System Attribute Keys

DECLARED IN Foundation/NSFileManager.h

SYNOPSIS extern NSString ***NSFileSystemSize**;
extern NSString ***NSFileSystemFreeSize**;
extern NSString ***NSFileSystemNodes**;
extern NSString ***NSFileSystemFreeNodes**;

General Exception Names

DECLARED IN Foundation/NSException.h

SYNOPSIS
extern NSString ***NSGenericException**;
extern NSString ***NSRangeException**;
extern NSString ***NSInvalidArgumentException**;
extern NSString ***NSInternalInconsistencyException**;
extern NSString ***NSMallocException**;
extern NSString ***NSObjectInaccessibleException**;
extern NSString ***NSObjectNotAvailableException**;
extern NSString ***NSDestinationInvalidException**;
extern NSString ***NSPortTimeoutException**;
extern NSString ***NSInvalidSendPortException**;
extern NSString ***NSInvalidReceivePortException**;
extern NSString ***NSPortSendException**;
extern NSString ***NSPortReceiveException**;
extern NSString ***NSOldStyleException**;
extern NSString ***NSInvalidSendPort**;
extern NSString ***NSInvalidReceivePort**;
extern NSString ***NSPortSendError**;
extern NSString ***NSPortReceiveError**;
extern NSString ***NSCharacterConversionException**;

Hash Table Call Backs

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS
extern const NSHashTableCallBacks **NSIntHashCallBacks**;
extern const NSHashTableCallBacks **NSNonOwnedPointerHashCallBacks**;
extern const NSHashTableCallBacks **NSOwnedPointerHashCallBacks**;
extern const NSHashTableCallBacks **NSPointerToStructHashCallBacks**;
extern const NSHashTableCallBacks **NSObjectHashCallBacks**;
extern const NSHashTableCallBacks **NSNonRetainedObjectHashCallBacks**;;

DESCRIPTION **NSIntHashCallBacks** are for sets of pointer-sized quantities or samller (for example, ints, longs, or unichars).

NSNonOwnedPointerHashCallBacks are for sets of pointers, hashed by address.

NSOwnedPointerHashCallBacks are for sets of pointers, with transfer of ownership upon insertion.

NSPointerToStructHashCallBacks are for sets of pointers to structs, when the first field of the struct is int-sized.

NSObjectHashCallBacks are for sets of objects (similar to NSSet)

NSNonRetainedObjectHashCallBacks are for sets of objects, but without retaining/releasing.

Note that you can make your own call back by picking fields among the above call backs.

Language-Dependent Information

DECLARED IN Foundation/NSUserDefaults.h

SYNOPSIS

```
extern NSString *NSWeekDayNameArray;
extern NSString *NSShortWeekDayNameArray;
extern NSString *NSMonthNameArray;
extern NSString *NSShortMonthNameArray;
extern NSString *NSTimeFormatString;
extern NSString *NSDateFormatString;
extern NSString *NSTimeDateFormatString;
extern NSString *NSShortTimeDateFormatString;
extern NSString *NSCurrencySymbol;
extern NSString *NSDecimalSeparator;
extern NSString *NSThousandsSeparator;
extern NSString *NSInternationalCurrencyString;
extern NSString *NSCurrencyString;
extern NSString *NSDecimalDigits;
extern NSString *NSAMPMDesignation;
extern NSString *NSHourNameDesignations;
extern NSString *NSYearMonthWeekDesignations;
extern NSString *NSEarlierTimeDesignations;
extern NSString *NSLaterTimeDesignations;
extern NSString *NSThisDayDesignations;
extern NSString *NSNextDayDesignations;
extern NSString *NSNextNextDayDesignations;
extern NSString *NSPriorDayDesignations;
extern NSString *NSDateTimeOrdering;
```

Map Table Key Call Backs

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS
extern const NSMapTableKeyCallBacks **NSIntMapKeyCallBacks**;
extern const NSMapTableKeyCallBacks **NSNonOwnedPointerMapKeyCallBacks**;
extern const NSMapTableKeyCallBacks **NSNonOwnedPointerOrNullMapKeyCallBacks**;
extern const NSMapTableKeyCallBacks **NSOwnedPointerMapKeyCallBacks**;
extern const NSMapTableKeyCallBacks **NSObjectMapKeyCallBacks**;
extern const NSMapTableKeyCallBacks **NSNonRetainedObjectMapKeyCallBacks**;

DESCRIPTION **NSIntMapKeyCallBacks** are for keys that are pointer-sized quantities or smaller (for example, ints, longs, or unichars).

NSNonOwnedPointerMapKeyCallBacks are for keys that are pointers not freed.

NSNonOwnedPointerOrNullMapKeyCallBacks are for keys that are pointers not freed, or NULL.

NSOwnedPointerMapKeyCallBacks are for keys that are pointers, with transfer of ownership upon insertion.

NSObjectMapKeyCallBacks are for keys that are objects

NSNonRetainedObjectMapKeyCallBacks are for sets of objects, but without retaining/releasing.

Note that you can make your own call back by picking fields among the above call backs.

Map Table Value Call Backs

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS
extern const NSMapTableValueCallBacks **NSIntMapValueCallBacks**;
extern const NSMapTableValueCallBacks **NSNonOwnedPointerMapValueCallBacks**;
extern const NSMapTableValueCallBacks **NSOwnedPointerMapValueCallBacks**;
extern const NSMapTableValueCallBacks **NSObjectMapValueCallBacks**;

DESCRIPTION **NSIntMapValueCallBacks** are for values that are pointer-sized quantities, such as ints.

NSNonOwnedPointerMapValueCallBacks are for values that are not owned pointers.

NSOwnedPointerMapValueCallBacks are for values that are owned pointers.

NSObjectMapValueCallBacks are for values that are objects.

Note that you can make your own call back by picking fields among the above call backs.

NSConnectionReplyMode

DECLARED IN **NSConnection.h**

SYNOPSIS extern NSString ***NSConnectionReplyMode**;

NSDefaultRunLoopMode

DECLARED IN **Foundation/NSRunLoop.h**

SYNOPSIS extern NSString ***NSDefaultRunLoopMode**;

Port Notification Names

DECLARED IN Foundation/NSPort.h

SYNOPSIS extern NSString ***NSPortDidBecomeInvalidNotification**;

PPL Notification Names

DECLARED IN Foundation/**NPPL.h**

SYNOPSIS
extern NSString ***NSPPLDidBecomeDirtyNotification**;
extern NSString ***NSPPLDidSaveNotification**;

Thread Notification Names

DECLARED IN Foundation/**NSThread.h**

SYNOPSIS
extern NSString ***NSBecomingMultiThreaded**;
extern NSString ***NSThreadExiting**;

User Defaults

DECLARED IN Foundation/NSUserDefaults.h

SYNOPSIS
extern NSString ***NSGlobalDomain**;
extern NSString ***NSArgumentDomain**;
extern NSString ***NSRegistrationDomain**;

Zero Constants

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS
extern const NSPoint **NSZeroPoint**;
extern const NSSize **NSZeroSize**;
extern const NSRect **NSZeroRect**;

Functions and Macros

This section describes the functions and macros found in the Foundation Kit.

NSAllHashTableObjects

SUMMARY This function returns all of the elements in the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS NSArray ***NSAllHashTableObjects**(NSHashTable **table*)

DESCRIPTION Returns an array object containing all the elements of *table*. This function should be called only when the table elements are objects, not when they're any other data type.

SEE ALSO **NSCreateHashTable, NSFreeHashTable**

NSAllMapTableKeys

SUMMARY This function returns all of the keys in the specified map table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSArray ***NSAllMapTableKeys**(NSMapTable **table*)

DESCRIPTION **NSAllMapTableKeys** Returns an array object containing all the keys in *table*. This function should be called only when the table keys are objects, not when they're any other type of pointer.

SEE ALSO **NSMapMember, NSMapGet, NSEnumerateMapTable, NSNextMapEnumeratorPair, NSAllMapTableValues**

NSAllMapTableValues

SUMMARY This function returns all of the values in the specified table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSArray ***NSAllMapTableValues**(NSMapTable **table*)

DESCRIPTION **NSAllMapTableValues** Returns an array object containing all the values in *table*. This function should be called only when the table values are objects, not when they're any other type of pointer.

SEE ALSO **NSMapMember, NSMapGet, NSEnumerateMapTable, NSNextMapEnumeratorPair, NSAllMapTableKeys**

NSAllocateMemoryPages

SUMMARY This function allocates a new block of memory.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void ***NSAllocateMemoryPages**(unsigned int *byteCount*)

DESCRIPTION **NSAllocateMemoryPages** allocates the integral number of pages whose total size is closest to, but not less than, *byteCount*. The allocated pages are guaranteed to be filled with zeros.

SEE ALSO **NSCopyMemoryPages, NSDeallocateMemoryPages**

NSAllocateObject

SUMMARY This function creates new objects.

DECLARED IN Foundation/NSObject.h

SYNOPSIS id <NSObject> **NSAllocateObject(**Class *aClass*, unsigned int *extraBytes*, NSZone **zone***)**

DESCRIPTION NSAllocateObject allocates and returns a pointer to an instance of *aClass*, created in the specified *zone* (or in the default zone, if *zone* is NULL). The *extraBytes* argument (usually zero) states the number of extra bytes required for indexed instance variables. Returns **nil** on failure.

SEE ALSO **NSCopyObject, NSDeallocateObject**

NSAssert

SUMMARY This macro generates an assertion if the given condition is false.

DECLARED IN Foundation/NSExceptions.h

SYNOPSIS **NSAssert(***condition***,** NSString *****description***)**

DESCRIPTION Assertions evaluate a condition and, if the condition evaluates to false, call the assertion handler for the current thread, passing it a format string and a variable number of arguments. Each thread has its own assertion handler, which is an object of class NSAssertionHandler. When invoked, an assertion handler prints an error message that includes the method and class names (or the function name). It then raises an exception of type NSInternalInconsistencyException.

The **NSAssert** macro evaluates the condition and serves as a front end to the assertion handler. This macro should be used only within Objective-C methods. **NSAssert** takes no arguments other than the condition and format string.

condition must be an expression that evaluates to true or false. *description* is a **printf**-style format string that contains the error message describing the failure condition.

Assertions are disabled if the preprocessor macro **NS_BLOCK_ASSERTIONS** is defined. All macros return **void**.

SEE ALSO **NSLog, NSLogv, NSAssert1, NSCAssert, NSCParameterAssert, NSParameterAssert**

NSAssert1

SUMMARY NSAssert1 is one of a series of macros, as listed in the SYNOPSIS section, that generate assertions if the given condition is false.

DECLARED IN Foundation/NSExceptions.h

SYNOPSIS
NSAssert1(*condition***,** NSString ******description, arg1* **)**
NSAssert2(*condition***,** NSString ******description, arg1, arg2***)**
NSAssert3(*condition***,** NSString ******description, arg1***,** *arg2***,** *arg3***)**
NSAssert4(*condition***,** NSString ******description, arg1, arg2***,** *arg3***,** *arg4***)**
NSAssert5(*condition***,** NSString ******description, arg1***,** *arg2***,** *arg3***,** *arg4***,** *arg5***)**

DESCRIPTION Assertions evaluate a condition and, if the condition evaluates to false, call the assertion handler for the current thread, passing it a format string and a variable number of arguments. Each thread has its own assertion handler, which is an object of class NSAssertionHandler. When invoked, an assertion handler prints an error message that includes the method and class names (or the function name). It then raises an exception of type NSInternalInconsistencyException.

The **NSAssert*n*** macros evaluate the condition and serve as a front end to the assertion handler. These macros should be used only within Objective-C methods. **NSAssert*n*** takes the number of format-string arguments indicated by *n*.

condition must be an expression that evaluates to true or false. *description* is a **printf**-style format string that contains the error message describing the failure condition. Each *arg* parameter is an argument to be inserted, in place, into the *description*.

Assertions are disabled if the preprocessor macro **NS_BLOCK_ASSERTIONS** is defined. All macros return **void**.

SEE ALSO **NSLog, NSLogv, NSAssert, NSCAssert, NSCParameterAssert, NSParameterAssert**

NSAssert2

SUMMARY See **NSAssert1**.

NSAssert3

SUMMARY See **NSAssert1**.

NSAssert4

SUMMARY See **NSAssert1**.

NSAssert5

SUMMARY See **NSAssert1**.

NSCAssert

SUMMARY This macro generates an assertion if the given condition is false.

DECLARED IN Foundation/NSExceptions.h

SYNOPSIS **NSCAssert(***condition***,** NSString ***description***)**

DESCRIPTION Assertions evaluate a condition and, if the condition evaluates to false, call the assertion handler for the current thread, passing it a format string and a variable number of arguments. Each thread has its own assertion handler, which is an object of class NSAssertionHandler. When invoked, an

assertion handler prints an error message that includes method and class names (or the function name). It then raises an exception of type NSInternalInconsistencyException.

The **NSCAssert** macro evaluates the condition and serves as a front end to the assertion handler. This macro should be used only within C functions. **NSCAssert** takes no arguments other than the condition and format string.

condition must be an expression that evaluates to true or false. *description* is a **printf**-style format string that describes the failure condition.

Assertions are disabled if the preprocessor macro **NS_BLOCK_ASSERTIONS** is defined. All macros return **void**.

SEE ALSO **NSLog, NSLogv, NSAssert, NSCAssert1, NSCParameterAssert, NSParameterAssert**

NSCAssert1

SUMMARY NSCAssert1 is one of a series of macros, as listed in the SYNOPSIS section, that generate assertions if the given condition is false.

DECLARED IN Foundation/NSExceptions.h

SYNOPSIS
NSCAssert1(*condition***,** NSString ***description, arg1***)**
NSCAssert2(*condition***,** NSString ***description, arg1, arg2***)**
NSCAssert3(*condition***,** NSString ***description, arg1***, *arg2*, *arg3***)**
NSCAssert4(*condition***,** NSString ***description, arg1, arg2***, *arg3*, *arg4***)**
NSCAssert5(*condition***,** NSString ***description, arg1***, *arg2*, *arg3*, *arg4*, *arg5***)**

DESCRIPTION Assertions evaluate a condition and, if the condition evaluates to false, call the assertion handler for the current thread, passing it a format string and a variable number of arguments. Each thread has its own assertion handler, which is an object of class NSAssertionHandler. When invoked, an assertion handler prints an error message that includes method and class names (or the function name). It then raises an exception of type NSInternalInconsistencyException.

The **NSCAssert*n*** macros evaluate the condition and serve as a front end to the assertion handler. These macros should be used only within Objective-C methods. **NSCAssert*n*** takes the number of format-string arguments indicated by *n*.

condition must be an expression that evaluates to true or false. *description* is a **printf**-style format string that describes the failure condition. Each *arg* is an argument to be inserted, in place, into the *description*.

Assertions are disabled if the preprocessor macro **NS_BLOCK_ASSERTIONS** is defined. All macros return **void**.

SEE ALSO **NSLog, NSLogv, NSCAssert, NSCParameterAssert, NSParameterAssert**

NSCAssert2

SUMMARY See **NSCAssert1**.

NSCAssert3

SUMMARY See **NSCAssert1**.

NSCAssert4

SUMMARY See **NSCAssert1**.

NSCAssert5

SUMMARY See **NSCAssert1**.

NSClassFromString

SUMMARY This function obtains a class by name.

DECLARED IN Foundation/NSObjCRuntime.h

SYNOPSIS Class **NSClassFromString**(NSString ***aClassName*)

DESCRIPTION **NSClassFromString** returns the class object named by *aClassName*, or **nil** if no class by that name is currently loaded.

SEE ALSO **NSStringFromClass**

NSCompareHashTables

SUMMARY This function compares the elements of two hash tables for equality.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS BOOL **NSCompareHashTables**(NSHashTable ***table1*, NSHashTable ***table2*)

DESCRIPTION Returns YES if the two hash tables are equal—that is, if each element of *table1* is in *table2*, and the two tables are the same size.

SEE ALSO **NSCreateHashTable, NSCreateHashTableWithZone**

NSCompareMapTables

SUMMARY This function compares the elements of two map tables for equality.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS BOOL **NSCompareMapTables**(NSMapTable ***table1*, NSMapTable ***table2*)

DESCRIPTION Returns YES if each key of *table1* is in *table2*, and the two tables are the same size. Note that this function does not compare values, only keys.

SEE ALSO **NSCreateMapTable, NSCreateMapTableWithZone**

NSContainsRect

SUMMARY This function determines whether one rectangle completely encloses another.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSContainsRect**(NSRect *aRect*, NSRect *bRect*)

DESCRIPTION Returns YES if *aRect* completely encloses *bRect*. For this condition to be true, *bRect* cannot be empty and its sides must not touch the sides of *aRect*.

NSConvertHostDoubleToSwapped

SUMMARY This function performs a type conversion.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedDouble **NSConvertHostDoubleToSwapped**(double *x*)

DESCRIPTION Converts the double value in *x* to a value whose bytes can be swapped. This function does not actually swap the bytes of x. You should not need to call this method directly.

SEE ALSO **NSSwapHostDoubleToBig, NSSwapHostDoubleToLittle**

NSConvertHostFloatToSwapped

SUMMARY This function performs a type conversion.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedFloat **NSConvertHostFloatToSwapped**(float *x*)

DESCRIPTION Converts the float value in *x* to a value whose bytes can be swapped. This function does not actually swap the bytes of x. You should not need to call this method directly.

SEE ALSO **NSSwapHostFloatToBig, NSSwapHostFloatToLittle**

NSConvertSwappedDoubleToHost

SUMMARY This function performs a type conversion.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS double **NSConvertSwappedDoubleToHost**(NSSwappedDouble *x*)

DESCRIPTION Converts the value in *x* to a double value. This function does not actually swap the bytes of x. You should not need to call this method directly.

SEE ALSO **NSSwapBigDoubleToHost, NSSwapLittleDoubleToHost**

NSConvertSwappedFloatToHost

SUMMARY This function performs a type conversion.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS float **NSConvertSwappedFloatToHost**(NSSwappedFloat *x*)

DESCRIPTION Converts the value in *x* to a float value. This function does not actually swap the bytes of x. You should not need to call this method directly.

SEE ALSO **NSSwapBigFloatToHost, NSSwapLittleFloatToHost**

NSCopyHashTableWithZone

SUMMARY This function performs a shallow copy of the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS NSHashTable ***NSCopyHashTableWithZone**(NSHashTable **table*, NSZone **zone*)

DESCRIPTION Returns a pointer to a new copy of *table*, created in *zone* and containing pointers to the data elements of *table*. If *zone* is NULL, the new table is created in the default zone.

The new table adopts the callback functions of table and calls the **hash** and **retain** callback functions as appropriate when inserting elements into the new table.

SEE ALSO **NSCreateHashTable, NSCreateHashTableWithZone, NSHashTableCallBacks (structure)**

NSCopyMapTableWithZone

SUMMARY This function performs a shallow copy of the specified map table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSMapTable ***NSCopyMapTableWithZone**(NSMapTable **table*, NSZone **zone*)

DESCRIPTION Returns a pointer to a new copy of *table*, created in *zone* and containing pointers to the keys and values of *table*. If *zone* is NULL, the new table is created in the default zone.

The new table adopts the callback functions of table and calls the **hash** and **retain** callback functions as appropriate when inserting elements into the new table.

SEE ALSO **NSCreateMapTable, NSCreateMapTableWithZone, NSMapTableKeyCallBacks (structure), NSMapTableValueCallBacks (structure)**

NSCopyMemoryPages

SUMMARY This function copies a block of memory.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void **NSCopyMemoryPages**(const void ***source*, void ***destination*, unsigned int *byteCount*)

DESCRIPTION Copies (or copies-on-write) *byteCount* bytes from *source* to *destination*.

SEE ALSO **NSAllocateMemoryPages, NSDeallocateMemoryPages**

NSCopyObject

SUMMARY This function creates exact copies of objects.

DECLARED IN Foundation/NSObject.h

SYNOPSIS id <NSObject> **NSCopyObject(**id <NSObject> *anObject*, unsigned int *extraBytes*, NSZone ***zone***)**

DESCRIPTION Creates and returns a new object that's an exact copy of *anObject*, created in the specified *zone* (or in the default zone, if *zone* is NULL). The *extraBytes* argument (usually zero) states the number of extra bytes required for indexed instance variables. Returns **nil** if *anObject* is **nil** or if *anObject* could not be copied.

SEE ALSO **NSAllocateObject, NSDeallocateObject**

NSCountHashTable

SUMMARY This function returns the number of elements in a hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS unsigned int **NSCountHashTable**(NSHashTable *_table_)

DESCRIPTION Returns the number of elements currently in *table*.

NSCountMapTable

SUMMARY This function returns the number of elements in a map table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS unsigned int **NSCountMapTable**(NSMapTable *_table_)

DESCRIPTION Returns the number of key/value pairs currently in *table*.

NSCParameterAssert

SUMMARY This macro evaluates the specified parameter.

DECLARED IN Foundation/NSExceptions.h

SYNOPSIS **NSCParameterAssert(**_condition_**)**

DESCRIPTION Assertions evaluate a condition and, if the condition evaluates to false, call the assertion handler for the current thread, passing it a format string and a variable number of arguments. Each thread has its own assertion handler, which is an object of class NSAssertionHandler. When invoked, an assertion handler prints an error message that includes method and class names (or the function name). It then raises an exception of type NSInternalInconsistencyException.

This macro validates a parameter for a C-function. Simply provide the parameter as the *condition* argument. The macro evaluates the parameter and, if the parameter evaluates to false, logs an error message which includes the parameter and then raises an exception.

Assertions are disabled if the preprocessor macro **NS_BLOCK_ASSERTIONS** is defined. All macros return **void**.

SEE ALSO **NSLog, NSLogv, NSAssert, NSCAssert, NSParameterAssert**

NSCreateHashTable

SUMMARY This function creates a new hash table in the default zone.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS NSHashTable *__NSCreateHashTable__(NSHashTableCallBacks *callBacks*, unsigned int *capacity*)

DESCRIPTION **NSCreateHashTable** creates and returns a pointer to an NSHashTable in the default zone.

The table's size is dependent on (but generally not equal to) *capacity*. If *capacity* is 0, a small hash table is created. The NSHashTableCallBacks structure *callBacks* has five pointers to functions (documented under "Types and Constants"), with the following defaults: pointer hashing, if **hash** is NULL; pointer equality, if **isEqual** is NULL; no call-back upon adding an element, if **retain** is NULL; no call-back upon removing an element, if **release** is NULL; and a function returning a pointer's hexadecimal value as a string, if **describe** is NULL. The hashing function must be defined such that if two data elements are equal, as defined by the comparison function, the values produced by hashing on these elements must also be equal. Also, data elements must remain invariant if the value of the hashing function depends on them; for example, if the hashing function operates directly on the characters of a string, that string can't change.

SEE ALSO **NSCopyHashTableWithZone, NSCreateHashTableWithZone**

NSCreateHashTableWithZone

SUMMARY This function creates a new hash table in the specified zone.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS NSHashTable ***NSCreateHashTableWithZone**(NSHashTableCallBacks *callBacks*, unsigned int *capacity*, NSZone **zone*)

DESCRIPTION This function creates a new hash table in the specified *zone*. If *zone* is NULL, the hash table is created in the default zone.

The table's size is dependent on (but generally not equal to) *capacity*. If *capacity* is 0, a small hash table is created. The NSHashTableCallBacks structure *callBacks* has five pointers to functions (documented under "Types and Constants"), with the following defaults: pointer hashing, if **hash** is NULL; pointer equality, if **isEqual** is NULL; no call-back upon adding an element, if **retain** is NULL; no call-back upon removing an element, if **release** is NULL; and a function returning a pointer's hexadecimal value as a string, if **describe** is NULL. The hashing function must be defined such that if two data elements are equal, as defined by the comparison function, the values produced by hashing on these elements must also be equal. Also, data elements must remain invariant if the value of the hashing function depends on them; for example, if the hashing function operates directly on the characters of a string, that string can't change.

SEE ALSO **NSCreateHashTable**

NSCreateMapTable

SUMMARY This function creates a new map table in the default zone.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSMapTable ***NSCreateMapTable**(NSMapTableKeyCallBacks *keyCallBacks*, NSMapTableValueCallBacks *valueCallBacks*, unsigned int *capacity*)

DESCRIPTION **NSCreateMapTable** creates, and returns a pointer to, an NSMapTable in the default zone; the table's size is dependent on (but generally not equal to) *capacity*. If *capacity* is 0, a small map table is created. The NSMapTableKeyCallBacks arguments are structures (documented under "Types

and Constants") that are very similar to the call-back structure used by **NSCreateHashTable**; in fact, they have the same defaults as documented for that function.

SEE ALSO **NSCopyMapTableWithZone, NSCreateMapTableWithZone**

NSCreateMapTableWithZone

SUMMARY This function creates a new map table in the specified zone.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSMapTable ***NSCreateMapTableWithZone**(NSMapTableKeyCallBacks *keyCallBacks*, NSMapTableValueCallBacks *valueCallBacks*, unsigned int *capacity*, NSZone **zone*)

DESCRIPTION This function creates a new map table in the specified *zone*. If *zone* is NULL, the hash table is created in the default zone.

The table's size is dependent on (but generally not equal to) *capacity*. If *capacity* is 0, a small map table is created. The NSMapTableKeyCallBacks arguments are structures (documented under "Types and Constants") that are very similar to the call-back structure used by **NSCreateHashTable**; in fact, they have the same defaults as documented for that function.

SEE ALSO **NSCopyMapTableWithZone, NSCreateMapTable**

NSCreateZone

SUMMARY Creates a new zone

DECLARED IN Foundation/NSZone.h

SYNOPSIS NSZone ***NSCreateZone(**unsigned int *startSize*, unsigned int *granularity*, BOOL *canFree***)**

DESCRIPTION Creates and returns a pointer to a new zone of *startSize* bytes, which will grow and shrink by *granularity* bytes. If *canFree* is zero, the allocator will never free memory, and **malloc** will be fast.

SEE ALSO **NSDefaultMallocZone, NSRecycleZone, NSSetZoneName**

NSDeallocateMemoryPages

SUMMARY This function deallocates the specified block of memory.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void **NSDeallocateMemoryPages**(void ***pointer*, unsigned int *byteCount*)

DESCRIPTION This function deallocates memory that was allocated with **NSAllocateMemoryPages**.

SEE ALSO **NSCopyMemoryPages, NSAllocateMemoryPages**

NSDeallocateObject

SUMMARY This function destroys an existing object.

DECLARED IN Foundation/NSObject.h

SYNOPSIS void **NSDeallocateObject(**id <NSObject> *anObject***)**

DESCRIPTION This function deallocates *anObject*, which must have been allocated using NSAllocateObject.

SEE ALSO **NSCopyObject, NSAllocateObject**

NSDecimalAdd

SUMMARY This function adds two decimal values.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError **NSDecimalAdd**(NSDecimal *_result_, const NSDecimal *_leftOperand_, const NSDecimal *_rightOperand_, NSRoundingMode _roundingMode_)

DESCRIPTION Adds _leftOperand_ to _rightOperand_, and stores the sum in _result_.

An NSDecimal can represent a number with up to 38 significant digits. If a number is more precise than that, it must be rounded off. _roundingMode_ determines how to round it off. There are four possible rounding modes:

- NSRoundDown. The number rounds down.
- NSRoundUp. The number rounds up.
- NSRoundPlain. The number rounds to the closest 38-digit approximation. If the number is halfway between two positive numbers, it round up; if it's halfway between two negative numbers, it rounds down.
- NSRoundBankers. The number rounds to the closest 38-digit approximation. If it is caught halfway between two possibilities, it rounds to the one whose last digit is even. In practice, this means that, over the long run, numbers will be rounded up as often as they are rounded down; there will be no systematic bias.

The return value indicates whether any machine limitations were encountered in the addition. If none were encountered, the function returns NSCalculationNoError. Otherwise it may return one of the following values: NSCalculationLossOfPrecision, NSCalculationOverflow or NSCalculationUnderflow. For descriptions of all these error conditions, see **exceptionDuringOperation:error:leftOperand:rightOperand:** in the protocol specification for NSDecimalNumberBehaviors.

NSDecimalCompact

SUMMARY This function compacts the decimal structure for efficiency.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS void **NSDecimalCompact**(NSDecimal *_number_)

DESCRIPTION Formats *number* so that calculations using it will take up as little memory as possible. All the **NSDecimal...** arithmetic functions expect compact NSDecimal arguments.

NSDecimalCompare

SUMMARY This function compares two decimal values.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSComparisonResult **NSDecimalCompare**(const NSDecimal *_leftOperand_, const NSDecimal *_rightOperand_)

DESCRIPTION Compares *leftOperand* to *rightOperand*, with three possible return values.

- If *leftOperand* is bigger, the function returns NSOrderedDescending.
- If *rightOperand* is bigger, the the function returns NSOrdered Ascending.
- If the two operands are equal, the function returns NSOrderedSame.

NSDecimalCopy

SUMMARY This function copies the value of a decimal number.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS void **NSDecimalCopy**(NSDecimal *_destination_, const NSDecimal *_source_)

DESCRIPTION Copies the value in *source* to *destination*.

NSDecimalDivide

SUMMARY This function divides one decimal value by another.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError **NSDecimalDivide**(NSDecimal ***result*, const NSDecimal ***leftOperand*, const NSDecimal ***rightOperand*, NSRoundingMode *roundingMode*)

DESCRIPTION Divides *leftOperand* by *rightOperand*, and stores the quotient, possibly rounded off according to *roundingMode*, in *result*. If *rightOperand* is 0, returns NSDivideByZero.

For explanations of the other possible return values, and of all the possible *roundingMode*'s, see **NSDecimalAdd**, above.

Note that this function can't precisely represent a non-decimal fraction like 1/3.

NSDecimalIsNotANumber

SUMMARY This function determines if the specified decimal contains a valid number.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS BOOL **NSDecimalIsNotANumber**(const NSDecimal ***decimal*)

DESCRIPTION Returns YES if the value in *decimal* represents a valid number, otherwise returns NO.

NSDecimalMultiply

SUMMARY This function multiplies two decimal numbers together.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError **NSDecimalMultiply**(NSDecimal ***result*, const NSDecimal ***leftOperand*, const NSDecimal ***rightOperand*, NSRoundingMode *roundingMode*)

DESCRIPTION Multiplies *rightOperand* by *leftOperand*, and stores the product, possibly rounded off according to *roundingMode*, in *result*.

For explanations of the possible return values and *roundingMode*'s, see **NSDecimalAdd**, above.

NSDecimalMultiplyByPowerOf10

SUMMARY This function multiplies a decimal by the specified power of 10.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError NSDecimalMultiplyByPowerOf10(NSDecimal ***result*, const NSDecimal ***number*, short *power*, NSRoundingMode *roundingMode*)

DESCRIPTION Multiplies *number* by 10*power*, and stores the product, possibly rounded off according to *roundingMode*, in *result*.

For explanations of the possible return values and *roundingMode*'s, see **NSDecimalAdd**, above.

NSDecimalNormalize

SUMMARY This function normalizes the internal format of two decimal numbers to simplify later operations.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError **NSDecimalNormalize**(NSDecimal ***number1*, NSDecimal ***number2*, NSRoundingMode *roundingMode*)

DESCRIPTION An NSDecimal is represented in memory as a mantissa and an exponent, expressing the value mantissa **x** 10exponent. A number can have many NSDecimal representations; for example, the following table lists several valid NSDecimal representations for the number 100:

Mantissa	Exponent
100	0
10	1
1	2

NSDecimalNormalize formats number1 and number2 so that they have equal exponents. This format makes addition and subtraction very convenient. Both **NSDecimalAdd** and **NSDecimalSubtract** call **NSDecimalNormalize**. You may want to use it if you write more complicated addition or subtraction routines.

For explanations of the function's possible return values, see **NSDecimalAdd**, above.

NSDecimalPower

SUMMARY This function raises the decimal value to the specified power.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError **NSDecimalPower**(NSDecimal ***result*, const NSDecimal ***number*, unsigned int *power*, NSRoundingMode *roundingMode*)

DESCRIPTION Raises *number* to *power*, possibly rounding off according to *roundingMode*, and stores the resulting value in *result*.

For explanations of the possible return values and *roundingMode*'s, see **NSDecimalAdd**, above.

NSDecimalRound

SUMMARY This function rounds off the decimal value.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS void **NSDecimalRound**(NSDecimal **result*, const NSDecimal **number*, int *scale*, NSRoundingMode *roundingMode*)

DESCRIPTION Rounds *number* off according to the parameters *scale* and rounding *mode*, and stores the result in *result*.

scale specifies the number of digits result can have after its decimal point. *roundingMode* specifies the way that number is rounded off. There are four possible values for *roundingMode*: NSRoundDown, NSRoundUp, NSRoundPlain, and NSRoundBankers. For thorough discussions of *scale* and *roundingMode*, see the **scale** and **roundingMode** in the protocol specification for NSDecimalNumberBehaviors.

NSDecimalString

SUMMARY This function returns a string representation of the decimal value.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSString ***NSDecimalString**(const NSDecimal **decimal*, NSDictionary **locale*)

DESCRIPTION Returns a string representation of *decimal*. *locale* determines the format of the decimal separator.

NSDecimalSubtract

SUMMARY This function subtracts one decimal value from another.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError **NSDecimalSubtract**(NSDecimal ***result*, const NSDecimal ***leftOperand*, const NSDecimal ***rightOperand*, NSRoundingMode *roundingMode*)

DESCRIPTION Subtracts *rightOperand* from *leftOperand*, and stores the difference, possibly rounded off according to *roundingMode*, in *result*.

For explanations of the possible return values and *roundingMode*'s, see **NSDecimalAdd**, above.

NSDecrementExtraRefCountWasZero

SUMMARY This function decrements the specified object's reference count.

DECLARED IN Foundation/NSObject.h

SYNOPSIS BOOL **NSDecrementExtraRefCountWasZero(**id *anObject***)**

DESCRIPTION This function decrements the "extra reference" count of an object. Newly created objects have only one actual reference, so that a single release message results in the object being deallocated. Extra references are those beyond the single original reference, and are usually created by sending the object a retain message. Your code should generally not use these functions unless it is overriding the retain or release methods.

This function returns NO if *anObject* had an extra reference count. If *anObject* didn't have an extra referenct count, it returns YES, indicating that the object should be deallocated (with **dealloc**).

SEE ALSO **NSExtraRefCount, NSIncrementExtraRefCount**

NSDefaultMallocZone

SUMMARY Returns the default zone

DECLARED IN Foundation/NSZone.h

SYNOPSIS NSZone ***NSDefaultMallocZone(**void**)**

DESCRIPTION Returns the default zone, which is created automatically at startup. This is the zone used by the standard C function **malloc**.

SEE ALSO **NSCreateZone**

NSDivideRect

SUMMARY This function divides a rectangle into two new rectangles.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS void **NSDivideRect**(NSRect *inRect*, NSRect **slice*, NSRect **remainder*, float *amount*, NSRectEdge *edge*)

DESCRIPTION Creates two rectangles, *slice* and *remainder,* from *inRect*, by dividing *inRect* with a line that's parallel to one of *inRect*'s sides (namely, the side specified by edge—either NSMinXEdge, NSMinYEdge, NSMaxXEdge, or NSMaxYEdge). The size of *slice* is determined by *amount,* which measures the distance from *edge*.

SEE ALSO **NSInsetRect, NSIntegralRect, NSOffsetRect**

NSEnumerateHashTable

SUMMARY This function creates an enumerator for the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS NSHashEnumerator **NSEnumerateHashTable**(NSHashTable *_table_)

DESCRIPTION Returns an **NSHashEnumerator** structure that will cause successive elements of *table* to be returned each time this enumerator is passed to **NSNextHashEnumeratorItem**.

SEE ALSO **NSNextHashEnumeratorItem**

NSEnumerateMapTable

SUMMARY This function creates an enumerator for the specified map table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSMapEnumerator **NSEnumerateMapTable**(NSMapTable *_table_)

DESCRIPTION **NSEnumerateMapTable** returns an **NSMapEnumerator** structure that will cause successive key/value pairs of *table* to be visited each time this enumerator is passed to **NSNextMapEnumeratorPair**.

SEE ALSO **NSNextMapEnumeratorPair, NSMapMember, NSMapGet, NSAllMapTableKeys, NSAllMapTableValues**

NSEqualPoints

SUMMARY This function tests the two points for equality.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSEqualPoints**(NSPoint *aPoint*, NSPoint *bPoint*)

DESCRIPTION Returns YES if the two points *aPoint* and *bPoint* are identical, and NO otherwise.

NSEqualRanges

SUMMARY This function tests the two range values for equality.

DECLARED IN Foundation/NSRange.h

SYNOPSIS BOOL **NSEqualRanges**(NSRange *range1*, NSRange *range2*)

DESCRIPTION Returns YES if *range1* and *range2* have the same locations and lengths.

NSEqualRects

SUMMARY This function tests the two rectangles for equality.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSEqualRects**(NSRect *aRect*, NSRect *bRect*)

DESCRIPTION Returns YES if the two rectangles *aRect* and *bRect* are identical, and NOotherwise.

NSEqualSizes

SUMMARY This function tests the two size values for equality.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSEqualSizes**(NSSize *aSize*, NSSize *bSize*)

DESCRIPTION Returns YES if the two sizes *aSize* and *bSize* are identical, and NO otherwise.

NSExtraRefCount

SUMMARY This function returns the specified object's reference count.

DECLARED IN Foundation/NSObject.h

SYNOPSIS unsigned int **NSExtraRefCount**(id *object*)

DESCRIPTION Returns the current reference count of *object*. This function is used in conjunction with **NSIncrementExtraRefCount** and **NSDecrementExtraRefCountWasZero** in situations where you need to override an object's **retain** and **release** methods.

NSFreeHashTable

SUMMARY This function deletes the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void **NSFreeHashTable**(NSHashTable **table*)

DESCRIPTION **NSFreeHashTable** releases each element of the specified hash table and frees the table itself.

SEE ALSO **NSResetHashTable**

NSFreeMapTable

SUMMARY This function deletes the specified map table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void **NSFreeMapTable**(NSMapTable *_table_)

DESCRIPTION **NSFreeMapTable** releases each key and value of the specified map table and frees the table itself.

SEE ALSO **NSResetMapTable**

NSFullUserName

SUMMARY This function returns the full name of the current user.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSString ***NSFullUserName**(void)

DESCRIPTION Returns a string containing the full name of the current user.

SEE ALSO **NSUserName**

NSGetSizeAndAlignment

SUMMARY This function returns the type and size of the specified data type.

DECLARED IN Foundation/NSObjCRuntime.h

SYNOPSIS const char ***NSGetSizeAndAlignment**(const char **typePtr*,
unsigned int **sizep*,
unsigned int **alignp*)

DESCRIPTION Gets the next type code from *typePtr* and returns the size and alignment of that data type in *sizep* and *alignp*, respectively. You can specify 0 for either *sizep* or *alignp* to ignore the corresponding information. This function returns a new pointer to the string specified by *typePtr*; the new pointer points to the character just past the type code that was read.

NSGetUncaughtExceptionHandler

SUMMARY This function returns the top-level error handler.

DECLARED IN Foundation/NSException.h

SYNOPSIS NSUncaughtExceptionHandler ***NSGetUncaughtExceptionHandler(**void**)**

DESCRIPTION **NSGetUncaughtExceptionHandler** returns a pointer to the function serving as the top-level error handler. This handler will process exceptions raised outside of any exception-handling domain.

SEE ALSO **NSSetUncaughtExceptionHandler**

NSHashGet

SUMMARY This function returns an element of the hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void ***NSHashGet**(NSHashTable **table*, const void **pointer*)

DESCRIPTION Returns the pointer in the table that matches *pointer* (as defined by the **isEqual** call-back function). If there is no matching element, the function returns NULL

NSHashInsert

SUMMARY This function adds an element to the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void **NSHashInsert**(NSHashTable *_table_, const void *_pointer_)

DESCRIPTION **NSHashInsert** inserts *pointer*, which must not be NULL, into *table*. If *pointer* matches an item already in the table, the previous pointer is released using the **release** call-back function that was specified when the table was created.

SEE ALSO **NSHashRemove, NSHashInsertKnownAbsent, NSHashInsertIfAbsent**

NSHashInsertIfAbsent

SUMMARY This function adds an element to the specified hash table only if the table does not already contain the element.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void ***NSHashInsertIfAbsent**(NSHashTable *_table_, const void *_pointer_)

DESCRIPTION If *pointer* matches an item already in *table*, **NSHashInsertIfAbsent** returns the pre-existing pointer; otherwise, it adds *pointer* to the table and returns NULL. You must not specify NULL for *pointer*.

SEE ALSO **NSHashRemove, NSHashInsert, NSHashInsertKnownAbsent**

NSHashInsertKnownAbsent

SUMMARY This function adds an element to the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void **NSHashInsertKnownAbsent**(NSHashTable *_table_, const void *_pointer_)

DESCRIPTION **NSHashInsertKnownAbsent** inserts *pointer*, which must not be NULL, into *table*. Unike **NSHashInsert**, this function raises NSInvalidArgumentException if *table* already includes an element that matches *pointer.*

SEE ALSO **NSHashRemove, NSHashInsert, NSHashInsertIfAbsent**

NSHashRemove

SUMMARY This function removes an element from the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void **NSHashRemove**(NSHashTable *_table_, const void *_pointer_)

DESCRIPTION If *pointer* matches an item already in *table*, this function releases the pre-existing item.

SEE ALSO **NSHashInsert, NSHashInsertKnownAbsent, NSHashInsertIfAbsent**

NSHeight

SUMMARY This function returns the height of the specified rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSHeight**(NSRect *aRect*)

DESCRIPTION Returns the height of *aRect*.

SEE ALSO **NSMaxX, NSMaxY, NSMidX, NSMidY, NSMinX, NSMinY, NSWidth**

NSHomeDirectory

SUMMARY This function getsinformation about a user.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSString ***NSHomeDirectory(**void**)**

DESCRIPTION **NSHomeDirectory** returns a path to the current user's home directory.

SEE ALSO **NSFullUserName, NSUserName, NSHomeDirectoryForUser**

NSHomeDirectoryForUser

SUMMARY Get information about a user

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSString ***NSHomeDirectoryForUser(**NSString **userName***)**

DESCRIPTION **NSHomeDirectoryForUser** returns a path to the home directory for the user specified by *userName*.

SEE ALSO **NSFullUserName, NSUserName, NSHomeDirectory**

NSHostByteOrder

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned int **NSHostByteOrder**(void)

DESCRIPTION Returns the endian format, either NSLittleEndian or NSBigEndian, supported by the current platform.

NSIncrementExtraRefCount

SUMMARY This function increments the specified object's reference count.

DECLARED IN Foundation/NSObject.h

SYNOPSIS void **NSIncrementExtraRefCount(**id *anObject***)**

DESCRIPTION This function increments the "extra reference" count of an object. Newly created objects have only one actual reference, so that a single release message results in the object being deallocated. Extra references are those beyond the single original reference, and are usually created by sending the object a retain message. Your code should generally not use these functions unless it is overriding the retain or release methods.

SEE ALSO **NSExtraRefCount, NSDecrementExtraRefCountWasZero**

NSInsetRect

SUMMARY This function insets the rectangle by the specified amount.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSInsetRect**(NSRect *aRect*, float *dX*, float *dY*)

DESCRIPTION **NSInsetRect** returns a copy of the rectangle *aRect*, altered by moving the two sides that are parallel to the y-axis inwards by *dX*, and the two sides parallel to the x-axis inwards by *dY*.

SEE ALSO **NSDivideRect, NSIntegralRect, NSOffsetRect**

NSIntegralRect

SUMMARY This function adjusts the sides of the rectangle to integer values.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSIntegralRect**(NSRect *aRect*)

DESCRIPTION **NSIntegralRect** returns a copy of the rectangle *aRect*, expanded outwards just enough to ensure that none of its four defining values (*x*, *y*, *width*, and *height*) have fractional parts. If *aRect*'s width or height is zero or negative, this function returns a rectangle with origin at (0.0, 0.0) and with zero width and height.

SEE ALSO **NSDivideRect, NSInsetRect, NSOffsetRect**

NSIntersectionRange

SUMMARY This function returns the intersection of the specified ranges.

DECLARED IN Foundation/NSRange.h

SYNOPSIS NSRange **NSIntersectionRange(**NSRange *range1*, NSRange *range2***)**

DESCRIPTION **NSIntersectionRange** returns a range describing the intersection of *range1* and *range2*—that is, a range containing the indices that exist in both ranges. If the returned range's **length** field is zero, then the two ranges don't intersect, and the value of the **location** field is undefined.

SEE ALSO **NSUnionRange**

NSIntersectionRect

SUMMARY This function calculates the intersection of two rectangles.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSIntersectionRect**(NSRect *aRect*, NSRect *bRect*)

DESCRIPTION **NSIntersectionRect** returns the graphic intersection of *aRect* and *bRect*. If the two rectangles don't overlap, the returned rectangle has its origin at (0.0, 0.0) and zero width and height. (This includes situations where the intersection is a point or a line segment.)

SEE ALSO **NSUnionRect**

NSIntersectsRect

SUMMARY This function tests whether two rectangles intersect.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSIntersectsRect**(NSRect *aRect*, NSRect *bRect*)

DESCRIPTION Returns YES if *aRect* intersects *bRect*, otherwise returns NO. This function returns NO if either *aRect* and *bRect* has a width or height that is 0.

SEE ALSO **NSIntersectionRect**

NSIsEmptyRect

SUMMARY This function tests whether the specified rectangle is empty.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSIsEmptyRect**(NSRect *aRect*)

DESCRIPTION Returns YES if the rectangle encloses no area at all—that is, if its width or height is zero or negative.

NSJavaBundleCleanup

SYNOPSIS void **NSJavaBundleCleanup**(NSBundle **bundle*, NSDictionary **plist*)

DESCRIPTION <<Description Forthcoming>>

NSJavaBundleSetup

SYNOPSIS id **NSJavaBundleSetup**(NSBundle **bundle*, NSDictionary **plist*)

DESCRIPTION <<Description Forthcoming>>

NSJavaNeedsToLoadClasses

SYNOPSIS BOOL **NSJavaNeedsToLoadClasses**(NSDictionary **plist*)

DESCRIPTION <<Description Forthcoming>>

NSJavaNeedsVirtualMachine

SYNOPSIS BOOL **NSJavaNeedsVirtualMachine**(NSDictionary *_plist_)

DESCRIPTION <<Description Forthcoming>>

NSJavaProvidesClasses

SYNOPSIS BOOL NSJavaProvidesClasses(NSDictionary *plist)

DESCRIPTION <<Description Forthcoming>>

NSJavaSetup

SYNOPSIS id NSJavaSetup(NSDictionary *plist)

DESCRIPTION <<Description Forthcoming>>

NSJavaVirtualMachineSetup

SYNOPSIS id **NSJavaVirtualMachineSetup**(BOOL *_vmNeeded_)

DESCRIPTION <<Description Forthcoming>>

NSLocationInRange

SUMMARY This function verifies that the specified position is in the given range.

DECLARED IN Foundation/NSRange.h

SYNOPSIS BOOL **NSLocationInRange(**unsigned int *index*, NSRange *aRange***)**

DESCRIPTION **NSLocationInRange** returns YES if the given *index* lies within *aRange*—that is, if it's greater than or equal to *aRange*.location and less than *aRange*.location plus *aRange*.length.

NSLog

SUMMARY This function logs error messages to **stderr**.

DECLARED IN Foundation/NSUtilities.h

SYNOPSIS void **NSLog(**NSString **format*, ...**)**

DESCRIPTION NSLog simply calls NSLogv, passing it a variable number of arguments.

SEE ALSO **NSLogv**

NSLogPageSize

SUMMARY This function returns the binary log of the page size.

DECLARED IN Foundation/NSZone.h

SYNOPSIS unsigned int **NSLogPageSize**(void)

DESCRIPTION **NSLogPageSize** returns the binary log of the current page size.

SEE ALSO **NSRoundDownToMultipleOfPageSize, NSRoundUpToMultipleOfPageSize, NSPageSize**

NSLogv

SUMMARY This function logs error messages to **stderr**.

DECLARED IN Foundation/NSUtilities.h

SYNOPSIS void **NSLogv(**NSString ***format*, va_list *args***)**

DESCRIPTION **NSLogv** logs an error message. The message consists of a timestamp and the process ID prefixed to the string you pass in. You compose this string with a format string , *format*, and one or more arguments to be inserted into the string. The format specification allowed by these functions is that which is understood by NSString's formatting capabilities (which is not necessarily the set of format escapes and flags understood by **printf**).

In general, you should use the **NSLog** function instead of calling this function directly. If you do use this function directly, you must have prepared the the variable argument list in the *args* parameter by calling the standard C macro va_start. Upon completion, you must similarly call the standard C macro va_end for this list.

On HP-UX, Solaris, and Mach, NSLogv writes the log to STDERR_FILENO if the file descriptor is open. If that fails, the message is sent to the syslog subsystem, if it exists on a platform, with the LOG_USER facility (or default facility if LOG_USER doesn't exist on a platform), with priority LOG_ERR (or similar, depending on what the platform supports). If both of these attempts to write the message fail, the message is discarded.

On Windows platforms, the message is written to the STD_ERROR_HANDLE, if that handle is valid, on Windows platforms that support that standard handle. It is also written to the Windows Event Log on Windows platforms that support that, or to a file **c:\fndation.log** on Windows platforms that do not, if that file can be opened. If all of these attempts fail, the message is discarded. On some Windows platforms, the message to the Event Log may be truncated if there is a limit to the size of a message that the Event Log can accept. On Windows platforms that support an application discovering whether or not it's running under a debugger, NSLogv may only send the message to the debugger for its handling, via standard WIN32 mechanisms, and not also write the message to STD_ERROR_HANDLE and the Event Log. Note that a debugger may

choose to not display message thus sent to it, or may choose not to display all of the message—that has nothing to do with **NSLogv**.

Output from **NSLogv** is serialized, in that only one thread in a process can be doing the writing/logging described above at a time. All attempts at writing/logging a message complete before the next thread can begin its attempts.

The effects of **NSLogv** are not serialized with subsystems other than those discussed above (such as the standard I/O package) and do not produce side effects on those subsystems (such as causing buffered output to be flushed, which may be undesirable).

SEE ALSO **NSLog**

NSMakePoint

SUMMARY This function creates a new NSPoint from the specified values.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSPoint **NSMakePoint**(float *x*, float *y*)

DESCRIPTION Creates an NSPoint having the coordinates *x* and *y*.

NSMakeRange

SUMMARY This function creates a new NSRange from the specified values.

DECLARED IN Foundation/NSRange.h

SYNOPSIS NSRange **NSMakeRange**(unsigned int *location*, unsigned int *length*)

DESCRIPTION Creates an NSRange having the specified location and length.

NSMakeRect

SUMMARY This function creates a new NSRect from the specified values.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSMakeRect**(float *x*, float *y*, float *w*, float *h*)

DESCRIPTION Creates an NSRect having the specified origin and size.

NSMakeSize

SUMMARY This function creates a new NSSize from the specified values.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSSize **NSMakeSize**(float *w*, float *h*)

DESCRIPTION Creates an NSSize having the specified width and height.

NSMapGet

SUMMARY This function returns a map-table value for the specified key.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void ***NSMapGet**(NSMapTable **table*, const void **key*)

DESCRIPTION Returns the value that table maps to key, or NULL if the table doesn't contain key.

SEE ALSO **NSMapMember, NSEnumerateMapTable, NSNextMapEnumeratorPair, NSAllMapTableKeys, NSAllMapTableValues**

NSMapInsert

SUMMARY This function inserts a key/value pair into the specified table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void **NSMapInsert**(NSMapTable *_table_, const void *_key_, const void *_value_)

DESCRIPTION **NSMapInsert** inserts _key_ and _value_ into table. If _key_ matches a key already in the table, _value_ is retained and the previous value is released, using the retain and release call-back functions that were specified when the table was created. Raises InvalidArgumentException if _key_ is equal to the **notAKeyMarker** field of the table's NSMapTableKeyCallBacks structure.

SEE ALSO **NSMapRemove, NSMapInsertIfAbsent, NSMapInsertKnownAbsent**

NSMapInsertIfAbsent

SUMMARY This function inserts a key/value pair into the specified table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void ***NSMapInsertIfAbsent**(NSMapTable *_table_, const void *_key_, const void *_value_)

DESCRIPTION If _key_ matches a key already in table, **NSMapInsertIfAbsent** returns the pre-existing key; otherwise, it adds _key_ and _value_ to the table and returns NULL. Raises NSInvalidArgumentException if _key_ is equal to the **notAKeyMarker** field of the table's NSMapTableKeyCallBacks structure.

SEE ALSO **NSMapRemove, NSMapInsert, NSMapInsertKnownAbsent**

NSMapInsertKnownAbsent

SUMMARY This function inserts a key/value pair into the specified table if the pair had not been previously added.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void **NSMapInsertKnownAbsent**(NSMapTable *_table_, const void *_key_, const void *_value_)

DESCRIPTION **NSMapInsertKnownAbsent** inserts *key* (which must not be **notAKeyMarker**) and *value* into table. Unlike **NSMapInsert**, this function raises NSInvalidArgumentException if *table* already includes a key that matches *key*.

SEE ALSO **NSMapRemove, NSMapInsert, NSMapInsertIfAbsent**

NSMapMember

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS BOOL **NSMapMember**(NSMapTable *_table_, const void *_key_, void **_originalKey_, void **_value_)

DESCRIPTION Returns YES if table contains a key equal to *key*. If so, *originalKey* is set to *key*, and *value* is set to the value that the table maps to *key*.

SEE ALSO **NSMapGet, NSEnumerateMapTable, NSNextMapEnumeratorPair, NSAllMapTableKeys, NSAllMapTableValues**

NSMapRemove

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void **NSMapRemove**(NSMapTable *_table_, const void *_key_)

DESCRIPTION If *key* matches a key already in *table*, this function release the pre-existing key and its corresponding value.

SEE ALSO **NSMapInsert, NSMapInsertIfAbsent, NSMapInsertKnownAbsent**

NSMaxRange

DECLARED IN Foundation/NSRange.h

SYNOPSIS unsigned int **NSMaxRange**(NSRange *range*)

DESCRIPTION Returns *range*.location + *range*.length—in other words, the number one greater than the maximum value within the range.

NSMaxX

SUMMARY This function returns the largest x-coordinate of a rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSMaxX**(NSRect *aRect*)

DESCRIPTION **NSMaxX** returns the largest x-coordinate value within *aRect*.

SEE ALSO **NSWidth, NSHeight, NSMaxY**

NSMaxY

SUMMARY This function returns the largest y-coordinate of a rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSMaxY**(NSRect *aRect*)

DESCRIPTION **NSMaxY** returns the largest y-coordinate value within *aRect*.

SEE ALSO **NSWidth, NSHeight, NSMaxX**

NSMidX

SUMMARY This function returns the x-coordinate of a rectangle’s midpoint.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSMidX**(NSRect *aRect*)

DESCRIPTION **NSMidX** returns the x-coordinate of the center of the rectangle.

SEE ALSO **NSWidth, NSHeight, NSMidY**

NSMidY

SUMMARY This function returns the y-coordinate of a rectangle’s midpoint.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSMidY**(NSRect *aRect*)

DESCRIPTION **NSMidY** returns the y-coordinate of the rectangle's center point.

SEE ALSO **NSWidth, NSHeight, NSMidX**

NSMinX

SUMMARY This function returns the smallest x-coordinate of a rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSMinX**(NSRect *aRect*)

DESCRIPTION **NSMinX** returns the smallest x-coordinate value within *aRect*.

SEE ALSO **NSWidth, NSHeight, NSMinY**

NSMinY

SUMMARY This function returns the smallest y-coordinate of a rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSMinY**(NSRect *aRect*)

DESCRIPTION **NSMinY** returns the smallest y-coordinate value within *aRect* .

SEE ALSO **NSWidth, NSHeight, NSMinX**

NSMouseInRect

SUMMARY This function tests whether the point is in the specified rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSMouseInRect**(NSPoint *aPoint*, NSRect *aRect*, BOOL *flipped*)

DESCRIPTION Returns YES if the point represented by *aPoint* is located within the rectangle represented by *aRect*. It assumes an unscaled and unrotated coordinate system; the argument *flipped* should be YES if the coordinate system has been flipped so that the positive y-axis extends downward. This function is used to determine whether the hot spot of the cursor lies inside a given rectangle.

NSNextHashEnumeratorItem

SUMMARY This function returns the next hash-table element in the enumeration.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void ***NSNextHashEnumeratorItem**(NSHashEnumerator **enumerator*)

DESCRIPTION **NSNextHashEnumeratorItem** returns the next element in the table that enumerator is associated with, or NULL if *enumerator* has already iterated over all the elements.

SEE ALSO **NSEnumerateHashTable**

NSNextMapEnumeratorPair

SUMMARY This function returns the next map-table pair in the enumeration

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS BOOL **NSNextMapEnumeratorPair**(NSMapEnumerator *_enumerator_, void **_key_, void **_value_)

DESCRIPTION **NSNextMapEnumeratorPair** returns NO if *enumerator* has already iterated over all the elements in the table that *enumerator* is associated with. Otherwise, this function sets *key* and *value* to match the next key/value pair in the table, and returns YES.

SEE ALSO **NSEnumerateMapTable, NSMapMember, NSMapGet, NSAllMapTableKeys, NSAllMapTableValues**

NSOffsetRect

SUMMARY This function insets the rectangle by the specified amount.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSOffsetRect**(NSRect *aRect*, float *dX*, float *dY*)

DESCRIPTION **NSOffsetRect** returns a copy of the rectangle *aRect*, with its location shifted by *dX* along the x-axis and by *dY* along the y-axis.

SEE ALSO **NSDivideRect, NSInsetRect, NSIntegralRect**

NSOpenStepRootDirectory

SUMMARY This function returns the root directory of the user's system.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSString ***NSOpenStepRootDirectory**(void)

DESCRIPTION Returns a string identifying the root directory of the user's system.

SEE ALSO **NSHomeDirectory, NSHomeDirectoryForUser**

NSPageSize

SUMMARY This function returns the number of bytes in a page.

DECLARED IN Foundation/NSZone.h

SYNOPSIS unsigned int **NSPageSize**(void)

DESCRIPTION **NSPageSize** returns the number of bytes in a page.

SEE ALSO **NSRoundDownToMultipleOfPageSize, NSRoundUpToMultipleOfPageSize, NSLogPageSize**

NSParameterAssert

SUMMARY This macro validates the specified parameter.

DECLARED IN Foundation/NSExceptions.h

SYNOPSIS **NSParameterAssert(***condition***)**

DESCRIPTION Assertions evaluate a condition and, if the condition evaluates to false, call the assertion handler for the current thread, passing it a format string and a variable number of arguments. Each thread has its own assertion handler, which is an object of class NSAssertionHandler. When invoked, an assertion handler prints an error message that includes method and class names (or the function name). It then raises an exception of type NSInternalInconsistencyException.

This macro validates a parameter for an Objective-C method. Simply provide the parameter as the *condition* argument. The macro evaluates the parameter and, if it is false, it logs an error message which includes the parameter and then raises an exception.

Assertions are disabled if the preprocessor macro **NS_BLOCK_ASSERTIONS** is defined.All assertion macros return **void**.

SEE ALSO **NSLog, NSLogv, NSAssert, NSCAssert, NSCParameterAssert**

NSPointFromString

SUMMARY This function returns a point from a text-based representation.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSPoint **NSPointFromString**(NSString ***aString*)

DESCRIPTION Scans *aString* for text of the form "{x=*a*; y=*b*}" and returns the values for *a* and *b* in a new NSPoint object. If *aString* does not contain this text, this function returns an NSPoint object whose x- and y-coordinates are both 0.

SEE ALSO **NSStringFromPoint**

NSPointInRect

SUMMARY This function tests whether the specified point is in the rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSPointInRect**(NSPoint *aPoint*, NSRect *aRect*)

DESCRIPTION Performs the same test as **NSMouseInRect**, but assumes a flipped coordinate system.

NSRangeFromString

SUMMARY This function returns a range from a text-based representation.

DECLARED IN Foundation/NSRange.h

SYNOPSIS NSRange **NSRangeFromString**(NSString **aString*)

DESCRIPTION Returns *aString* for text of the form: "{location = *a*; length = *b*}" and returns the values for *a* and *b* in a new NSRange object. If *aString* does not contain this text, this function returns an NSRange object whose location and length values are both 0.

SEE ALSO **NSStringFromRange**

NSRealMemoryAvailable

SUMMARY This function returns information about the user's system.

DECLARED IN Foundation/NSZone.h

SYNOPSIS unsigned int **NSRealMemoryAvailable**(void)

DESCRIPTION **NSRealMemoryAvailable** returns the number of bytes available in RAM.

NSRectFromString

SUMMARY This function returns a rectangle from a text-based representation.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSRectFromString**(NSString *_aString_)

DESCRIPTION Scans _aString_ for text of the form "{x=_a_; y=_b_; width=_c_; height=_d_}", and returns the values for _a_, _b_, _c_, and _d_ in a new NSRect object. If _aString_ does not contain the specified text, this function returns an NSRect object with a rectangle whose origin is (0, 0) and width and height are both 0.

SEE ALSO **NSStringFromRect**

NSRecycleZone

SUMMARY This function frees memory in a zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void **NSRecycleZone(**NSZone *_zone_**)**
void **NSZoneFree**(NSZone *_zone_, void *_pointer_**)**

DESCRIPTION **NSRecycleZone** frees _zone_ after adding any of its pointers still in use to the default zone. (This strategy prevents retained objects from being inadvertently destroyed.)

NSZoneFree returns the memory indicated by _pointer_ to _zone_. The standard C function **free** does the same, but spends time finding which zone the memory belongs to.

RETURN Both functions return **void**.

SEE ALSO **NSCreateZone, NSZoneMalloc**

NSResetHashTable

SUMMARY This function deletes the elements of the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void **NSResetHashTable**(NSHashTable *_table_)

DESCRIPTION **NSResetHashTable** releases each element but doesn't deallocate the table. This is useful for preserving the table's capacity.

SEE ALSO **NSFreeHashTable**

NSResetMapTable

SUMMARY This function deletes the elements of the specified map table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void **NSResetMapTable**(NSMapTable *_table_)

DESCRIPTION **NSResetMapTable** releases each key and value but doesn’t deallocate the table. This is useful for preserving the table’s capacity.

SEE ALSO **NSFreeMapTable**

NSRoundDownToMultipleOfPageSize

SUMMARY This function returns the number of pages that correspond to the specified number of bytes.

DECLARED IN Foundation/NSZone.h

SYNOPSIS unsigned int **NSRoundDownToMultipleOfPageSize**(unsigned int *byteCount*)

DESCRIPTION **NSRoundDownToMultipleOfPageSize** returns the multiple of the page size that is closest to, but not greater than, *byteCount*.

SEE ALSO **NSPageSize, NSLogPageSize, NSRoundUpToMultipleOfPageSize**

NSRoundUpToMultipleOfPageSize

SUMMARY This function returns the number of pages that correspond to the specified number of bytes.

DECLARED IN Foundation/NSZone.h

SYNOPSIS unsigned int **NSRoundUpToMultipleOfPageSize**(unsigned int *byteCount*)

DESCRIPTION **NSRoundUpToMultipleOfPageSize** returns the multiple of the page size that is closest to, but not less than, *byteCount*.

SEE ALSO **NSPageSize, NSLogPageSize, NSRoundDownToMultipleOfPageSize**

NSSearchPathForDirectoriesInDomains

SYNOPSIS NSArray ***NSSearchPathForDirectoriesInDomains** (NSSearchPathDirectory *directory*, NSSearchPathDomainMask *domainMask*, BOOL *expandTilde*)

DESCRIPTION <<Description forthcoming>>

NSSelectorFromString

SUMMARY This function obtains a selector by name.

DECLARED IN Foundation/NSObjCRuntime.h

SYNOPSIS SEL **NSSelectorFromString(**NSString *_aSelectorName_**)**

DESCRIPTION **NSSelectorFromString** returns the selector named by *aSelectorName*, or zero if none by this name exists.

SEE ALSO **NSStringFromSelector**

NSSetUncaughtExceptionHandler

SUMMARY This function changes the top level error handler.

DECLARED IN Foundation/NSException.h

SYNOPSIS void **NSSetUncaughtExceptionHandler(**NSUncaughtExceptionHandler *_handler_**)**

DESCRIPTION **NSSetUncaughtExceptionHandler** sets the top-level error-handling function to *handler*. If *handler* is NULL or this function is never invoked, the default top-level handler is used.

SEE ALSO **NSGetUncaughtExceptionHandler**

NSSetZoneName

SUMMARY This function sets the name of the specified zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void **NSSetZoneName(**NSZone *_zone_, NSString *_name_**)**

DESCRIPTION **NSSetZoneName** sets the specified *zone*'s name to *name*, which can aid in debugging.

SEE ALSO **NSZoneName**

NSShouldRetainWithZone

SUMMARY This function indicates whether an object should be retained.

DECLARED IN Foundation/NSObject.h

SYNOPSIS BOOL **NSShouldRetainWithZone(**id <NSObject> *anObject*, NSZone **requestedZone***)**

DESCRIPTION Returns YES if *requestedZone* is NULL, the default zone, or the zone in which *anObject* was allocated. This function is typically called from inside an NSObject's **copyWithZone:** method, when deciding whether to retain *anObject* as opposed to making a copy of it.

NSSizeFromString

SUMMARY This function returns an NSSize from a text-based representation.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSSize **NSSizeFromString**(NSString **aString*)

DESCRIPTION Scans *aString* for text of the form "{width=*a*; height=*b*}" and returns the values for *a* and *b* in an NSSize object. If *aString* does not contain the specified text, this function returns an NSSize object whose width and height are both 0.

SEE ALSO **NSStringFromSize**

NSStandardApplicationPaths

SUMMARY This function returns the application paths for the user's system.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSArray* **NSStandardApplicationPaths**(void)

DESCRIPTION Returns an array of strings, each string specifying one of the standard paths to the OpenStep applications.

SEE ALSO **NSStandardLibraryPaths**

NSStandardLibraryPaths

SUMMARY This function returns the library paths for the user's system.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSArray* **NSStandardLibraryPaths**(void)

DESCRIPTION Returns an array of strings, each string specifying one of the standard paths to the OpenStep libraries.

SEE ALSO **NSStandardApplicationPaths**

NSStringFromClass

SUMMARY This function obtains the name of a class

DECLARED IN Foundation/NSObjCRuntime.h

SYNOPSIS NSString ***NSStringFromClass**(Class *aClass*)

DESCRIPTION This function returns an NSString containing the name of *aClass*.

SEE ALSO **NSClassFromString**

NSStringFromHashTable

SUMMARY This function returns a string describing the hash table's contents.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS NSString ***NSStringFromHashTable**(NSHashTable **table*)

DESCRIPTION Returns a string describing the hash table's contents. The function iterates over the table's elements, and for each one appends the string returned by the **describe** call-back function. If NULL was specified for the call-back function, the hexadecimal value of each pointer is added to the string.

NSStringFromMapTable

SUMMARY This function returns a string describing the map table's contents.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSString ***NSStringFromMapTable**(NSMapTable **table*)

DESCRIPTION Returns a string describing the map table's contents. The function iterates over the table's key/value pairs, and for each one appends the string "*a* = *b*;\n", where *a* and *b* are the key and value strings returned by the corresponding **describe** call-back functions. If NULL was specified for the call-back function, a and b are the key and value pointers, expressed as hexadecimal numbers.

NSStringFromPoint

SUMMARY This function returns a string representation of a point.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSString ***NSStringFromPoint**(NSPoint *aPoint*)

DESCRIPTION Returns a string of the form "{x=*a*; y=*b*}", where *a* and *b* are the x- and y-coordinates of *aPoint*.

SEE ALSO **NSPointFromString**

NSStringFromRange

SUMMARY This function returns a string representation of a range.

DECLARED IN Foundation/NSRange.h

SYNOPSIS NSString ***NSStringFromRange**(NSRange *aRange*)

DESCRIPTION Returns a string of the form: "{location = *a*; length = *b*}", where *a* and *b* are non-negative integers representing *aRange*.

NSStringFromRect

SUMMARY This function returns a string representation of a rect.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSString ***NSStringFromRect**(NSRect *aRect*)

DESCRIPTION Returns a string of the form "{x=*a*; y=*b*; width=*c*; height=*d*}", where *a*, *b*, *c*, and *d* are the x- and y-coordinates and the width and height, respectively, of *aRect*.

SEE ALSO **NSRectFromString**

NSStringFromSelector

SUMMARY This function returns the name of a selector.

DECLARED IN Foundation/NSObjCRuntime.h

SYNOPSIS NSString ***NSStringFromSelector(**SEL *aSelector***)**

DESCRIPTION **NSStringFromSelector** returns an NSString containing the name of *aSelector.*

SEE ALSO **NSSelectorFromString**

NSStringFromSize

SUMMARY This function returns a string representation of a size.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSString ***NSStringFromSize**(NSSize *aSize*)

DESCRIPTION Returns a string of the form "{width=*a*; height=*b*}", where *a* and *b* are the width and height of *aSize*.

SEE ALSO **NSSizeFromString**

NSSwapBigDoubleToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS double **NSSwapBigDoubleToHost**(NSSwappedDouble *x*)

DESCRIPTION Converts the big-endian value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapDouble** to perform the swap.

SEE ALSO **NSSwapHostDoubleToBig, NSSwapLittleDoubleToHost**

NSSwapBigFloatToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS float **NSSwapBigFloatToHost**(NSSwappedFloat *x*)

DESCRIPTION Converts the big-endian value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapFloat** to perform the swap.

SEE ALSO **NSSwapHostFloatToBig, NSSwapLittleFloatToHost**

NSSwapBigIntToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned int **NSSwapBigIntToHost**(unsigned int *x*)

DESCRIPTION Converts the big-endian value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapInt** to perform the swap.

SEE ALSO **NSSwapHostIntToBig, NSSwapLittleIntToHost**

NSSwapBigLongLongToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long long **NSSwapBigLongLongToHost**(unsigned long long *x*)

DESCRIPTION Converts the big-endian value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapLongLong** to perform the swap.

SEE ALSO **NSSwapHostLongLongToBig, NSSwapLittleLongLongToHost**

NSSwapBigLongToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long **NSSwapBigLongToHost**(unsigned long *x*)

DESCRIPTION Converts the big-endian value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapLong** to perform the swap.

SEE ALSO **NSSwapHostLongToBig, NSSwapLittleLongToHost**

NSSwapBigShortToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned short **NSSwapBigShortToHost**(unsigned short *x*)

DESCRIPTION Converts the big-endian value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapShort** to perform the swap.

SEE ALSO **NSSwapHostShortToBig, NSSwapLittleShortToHost**

NSSwapDouble

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedDouble **NSSwapDouble**(NSSwappedDouble *x*)

DESCRIPTION Swaps the bytes of *x* and returns the resulting value. Bytes are swapped from each low-order position to the corresponding high-order position and vice versa. For example, if the bytes of *x* are numbered from 1 to 8, this function swaps bytes 1 and 8, bytes 2 and 7, bytes 3 and 6, and bytes 4 and 5.

SEE ALSO **NSSwapLongLong, NSSwapFloat**

NSSwapFloat

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedFloat **NSSwapFloat**(NSSwappedFloat *x*)

DESCRIPTION Swaps the bytes of *x* and returns the resulting value.Bytes are swapped from each low-order position to the corresponding high-order position and vice versa. For example, if the bytes of *x* are numbered from 1 to 4, this function swaps bytes 1 and 4, and swaps bytes 2 and 3.

SEE ALSO **NSSwapLong, NSSwapDouble**

NSSwapHostDoubleToBig

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedDouble **NSSwapHostDoubleToBig**(double *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to big-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapDouble** to perform the swap.

SEE ALSO **NSSwapBigDoubleToHost, NSSwapHostDoubleToLittle**

NSSwapHostDoubleToLittle

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedDouble **NSSwapHostDoubleToLittle**(double *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to little-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapDouble** to perform the swap.

SEE ALSO **NSSwapLittleDoubleToHost, NSSwapHostDoubleToBig**

NSSwapHostFloatToBig

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedFloat **NSSwapHostFloatToBig**(float *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to big-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapFloat** to perform the swap.

SEE ALSO **NSSwapBigFloatToHost, NSSwapHostFloatToLittle**

NSSwapHostFloatToLittle

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedFloat **NSSwapHostFloatToLittle**(float *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to little-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapFloat** to perform the swap.

SEE ALSO **NSSwapLittleFloatToHost, NSSwapHostFloatToBig**

NSSwapHostIntToBig

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned int **NSSwapHostIntToBig**(unsigned int *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to big-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapInt** to perform the swap.

SEE ALSO **NSSwapBigIntToHost, NSSwapHostIntToLittle**

NSSwapHostIntToLittle

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned int **NSSwapHostIntToLittle**(unsigned int *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to little-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapInt** to perform the swap.

SEE ALSO **NSSwapLittleIntToHost, NSSwapHostIntToBig**

NSSwapHostLongLongToBig

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long long **NSSwapHostLongLongToBig**(unsigned long long *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to big-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapLongLong** to perform the swap.

SEE ALSO **NSSwapBigLongLongToHost, NSSwapHostLongLongToLittle**

NSSwapHostLongLongToLittle

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long long **NSSwapHostLongLongToLittle**(unsigned long long *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to little-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapLongLong** to perform the swap.

SEE ALSO **NSSwapLittleLongLongToHost, NSSwapHostLongLongToBig**

NSSwapHostLongToBig

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long **NSSwapHostLongToBig**(unsigned long *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to big-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapLong** to perform the swap.

SEE ALSO **NSSwapBigLongToHost, NSSwapHostLongToLittle**

NSSwapHostLongToLittle

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long **NSSwapHostLongToLittle**(unsigned long *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to little-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapLong** to perform the swap.

SEE ALSO **NSSwapLittleLongToHost, NSSwapHostLongToBig**

NSSwapHostShortToBig

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned short **NSSwapHostShortToBig**(unsigned short *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to big-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapShort** to perform the swap.

SEE ALSO **NSSwapBigShortToHost, NSSwapHostShortToLittle**

NSSwapHostShortToLittle

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned short **NSSwapHostShortToLittle**(unsigned short *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to little-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapShort** to perform the swap.

SEE ALSO **NSSwapLittleShortToHost, NSSwapHostShortToBig**

NSSwapInt

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned int **NSSwapInt** (unsigned int *inv*)

DESCRIPTION Swaps the bytes of *inv* and returns the resulting value. Bytes are swapped from each low-order position to the corresponding high-order position and vice versa. For example, if the bytes of *inv* are numbered from 1 to 4, this function swaps bytes 1 and 4, and swaps bytes 2 and 3.

SEE ALSO **NSSwapShort, NSSwapLong, NSSwapLongLong**

NSSwapLittleDoubleToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS double **NSSwapLittleDoubleToHost**(NSSwappedDouble *x*)

DESCRIPTION Converts the little-endian formatted value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapDouble** to perform the swap.

SEE ALSO **NSSwapHostDoubleToLittle, NSSwapBigDoubleToHost, NSConvertSwappedDoubleToHost**

NSSwapLittleFloatToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS float **NSSwapLittleFloatToHost**(NSSwappedFloat *x*)

DESCRIPTION Converts the little-endian formatted value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapFloat** to perform the swap.

SEE ALSO **NSSwapHostFloatToLittle, NSSwapBigFloatToHost, NSConvertSwappedFloatToHost**

NSSwapLittleIntToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned int **NSSwapLittleIntToHost**(unsigned int *x*)

DESCRIPTION Converts the little-endian formatted value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapInt** to perform the swap.

SEE ALSO **NSSwapHostIntToLittle, NSSwapBigIntToHost**

NSSwapLittleLongLongToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long long **NSSwapLittleLongLongToHost**(unsigned long long *x*)

DESCRIPTION Converts the little-endian formatted value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapLongLong** to perform the swap.

SEE ALSO **NSSwapHostLongLongToLittle, NSSwapBigLongLongToHost**

NSSwapLittleLongToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long **NSSwapLittleLongToHost**(unsigned long *x*)

DESCRIPTION Converts the little-endian formatted value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapLong** to perform the swap.

SEE ALSO **NSSwapHostLongToLittle, NSSwapBigLongToHost, NSSwapLong**

NSSwapLittleShortToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned short **NSSwapLittleShortToHost**(unsigned short *x*)

DESCRIPTION Converts the little-endian formatted value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapShort** to perform the swap.

SEE ALSO **NSSwapHostShortToLittle, NSSwapBigShortToHost**

NSSwapLong

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long **NSSwapLong**(unsigned long *inv*)

DESCRIPTION Swaps the bytes of *inv* and returns the resulting value. Bytes are swapped from each low-order position to the corresponding high-order position and vice versa. For example, if the bytes of *inv* are numbered from 1 to 4, this function swaps bytes 1 and 4, and swaps bytes 2 and 3.

SEE ALSO **NSSwapLongLong, NSSwapInt, NSSwapFloat**

NSSwapLongLong

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long long **NSSwapLongLong**(unsigned long long *inv*)

DESCRIPTION Swaps the bytes of *inv* and returns the resulting value. Bytes are swapped from each low-order position to the corresponding high-order position and vice versa. For example, if the bytes of *inv* are numbered from 1 to 8, this function swaps bytes 1 and 8, bytes 2 and 7, bytes 3 and 6, and bytes 4 and 5.

SEE ALSO **NSSwapLong, NSSwapDouble**

NSSwapShort

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned short **NSSwapShort** (unsigned short *inv*)

DESCRIPTION Swaps the low-order and high-order bytes of *inv* and returns the resulting value.

SEE ALSO **NSSwapInt, NSSwapLong**

NSTemporaryDirectory

SUMMARY This function returns the temporary directory on the user's system.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSString ***NSTemporaryDirectory**(void)

DESCRIPTION Returns a string containing the path of the current temporary directory. If no such directory is currently available, this function returns **nil**.

SEE ALSO **NSStandardApplicationPaths, NSStandardLibraryPaths, NSHomeDirectory**

NSUnionRange

SUMMARY This function returns the intersection of the specified ranges.

DECLARED IN Foundation/NSRange.h

SYNOPSIS NSRange **NSUnionRange(**NSRange *range1*, NSRange *range2***)**

DESCRIPTION **NSUnionRange** returns a range covering all indices in and between *range1* and *range2*. If one range is completely contained in the other, the returned range is equal to the larger range.

SEE ALSO **NSIntersectionRange**

NSUnionRect

SUMMARY This function calculates the union of two rectangles.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSUnionRect**(NSRect *aRect*, NSRect *bRect*)

DESCRIPTION **NSUnionRect** returns the smallest rectangle that completely encloses both *aRect* and *bRect*. If one of the rectangles has zero (or negative) width or height, a copy of the other rectangle is returned; but if both have zero (or negative) width or height, the returned rectangle has its origin at (0.0, 0.0) and has zero width and height.

SEE ALSO **NSIntersectionRect**

NSUserName

SUMMARY This function gets information about a user.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSString ***NSUserName(**void**)**

DESCRIPTION **NSUserName** returns the logon name of the current user.

SEE ALSO **NSFullUserName, NSHomeDirectory, NSHomeDirectoryForUser**

NSWidth

SUMMARY This function returns the width of the specified rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSWidth**(NSRect *aRect*)

DESCRIPTION **NSWidth** returns the width of *aRect*.

SEE ALSO **NSMaxX, NSMaxY, NSMidX, NSMidY, NSMinX, NSMinY, NSHeight**

NSZoneCalloc

SUMMARY This function allocates memory in a zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void ***NSZoneCalloc(**NSZone **zone*, unsigned int *numElems*, unsigned int *byteSize***)**

DESCRIPTION **NSZoneCalloc** allocates enough memory from *zone* for *numElems* elements, each with a size *numBytes* bytes, and returns a pointer to the allocated memory. The memory is initialized with zeros. This function returns **nil** if it was unable to allocate the requested memory.

SEE ALSO **NSDefaultMallocZone, NSRecycleZone, NSZoneFree, NSZoneMalloc, NSZoneRealloc**

NSZoneFree

SUMMARY This function deallocates a block of memory in the specified zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void **NSZoneFree**(NSZone **zone*, void **pointer*)

DESCRIPTION Returns memory to the zone from which it was allocated. The standard C function **free** does the same, but spends time finding which zone the memory belongs to.

SEE ALSO **NSRecycleZone, NSZoneMalloc, NSZoneCalloc, NSZoneRealloc**

NSZoneFromPointer

SUMMARY This function gets the zone for a given block of memory

DECLARED IN Foundation/NSZone.h

SYNOPSIS NSZone ***NSZoneFromPointer**(void **pointer*)

DESCRIPTION Returns the zone for the block of memory indicated by *pointer*, or NULL if the block was not allocated from a zone. The pointer must be one that was returned by a prior call to an allocation function.

RETURN Returns the zone for the indicated block of memory, or NULL if the block was not allocated from a zone.

SEE ALSO **NSZoneCalloc, NSZoneMalloc, NSZoneRealloc**

NSZoneMalloc

SUMMARY This function allocates memory in a zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void ***NSZoneMalloc(**NSZone **zone*, unsigned int *size***)**

DESCRIPTION **NSZoneMalloc** allocates *size* bytes in *zone*, and returns a pointer to the allocated memory. This function returns **nil** if it was unable to allocate the requested memory.

SEE ALSO **NSDefaultMallocZone, NSRecycleZone, NSZoneFree, NSZoneCalloc, NSZoneRealloc**

NSZoneName

SUMMARY This function returns the name of the specified zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS NSString ***NSZoneName(**NSZone **zone***)**

DESCRIPTION Returns a string containing the name associated with *zone*, If *zone* is **nil**, the default zone is used. If no name is associated with *zone*, the returned string is empty.

SEE ALSO **NSSetZoneName**

NSZoneRealloc

SUMMARY This function allocates memory in a zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void ***NSZoneRealloc(**NSZone **zone*, void **ptr*, unsigned int *size***)**

DESCRIPTION **NSZoneRealloc** changes the size of the block of memory pointed to by *ptr* to *size* bytes. It may allocate new memory to replace the old, in which case it moves the contents of the old memory block to the new block, up to a maximum of *size* bytes. *ptr* may be NULL. This function returns **nil** if it was unable to allocate the requested memory.

SEE ALSO **NSDefaultMallocZone, NSRecycleZone, NSZoneFree, NSZoneCalloc, NSZoneMalloc**

Locales

A locale is a set of defaults whose values are determined by the user's preferred language. The locale defaults are stored in the NSUserDefault's preferred language domain.

Certain classes from the Foundation Framework (for example, NSCalendarDate, NSDate, NSTimeZone, NSString, and NSScanner) use information from the locale to determine their values. For example, when you request an NSString representation of an NSCalendarDate, the NSCalendarDate looks at the locale to determine what the months and the days of the week are called in the user's preferred language.

The actual attributes for the locale keys are contained in **NextLibrary/Frameworks/Foundation.framework/Resources/Languages**. If a given language does not define one of the locale attributes, the value for that attribute defaults to the one defined for the default language (US English).

The Foundation Framework defines these locale dictionary keys. Other frameworks may add other keys to the locale dictionary.

Key	Description
NSAMPMDesignation	An array of strings that specify how the morning and afternoon designations are printed. The default is AM and PM.
NSCurrencySymbol	A string that specifies the symbol used to denote currency in this language. The default is "$".
NSDateFormatString	A format string that specifies how dates are printed using the date format specifiers. (See the NSCalendarDate class specification for a list of these.) The default is to use weekday names with full month names and full years, as in “Sunday, January 01, 1995.”
NSDateTimeOrdering	A string that specifies how to use ambiguous numbers in date strings. Specify this value as a permutation of the letters M (month), D (day), Y (year), and H (hour). For example, MDYH treats “2/3/95 10” as the 3rd day of February 1995 at 10:00am, whereas DMYH treats the same value as the 2nd day of March 1995 at 10:00am. If fewer numbers are specified than are needed, the numbers are prioritized to satisfy day first, then the month, and then the year. For example, if you supply only the value 12, it means the 12th day of this month in this year because the day must be specified. If you supply “2 12” it means either February 12 or December 2, depending on if the ordering is “MDYH” or “DMYH.”
NSDecimalDigits	Strings that identify the decimal digits in addition to or instead of the ASCII digits.
NSDecimalSeparator	A string that specifies the decimal separator. The decimal separator separates the ones place from the tenths place. The default is “.”.
NSEarlierTimeDesignations	An array of strings that denote a time in the past. These are adjectives that modify values from NSYearMonthWeekDesignations. The defaults are “prior,” “last,” “past,” and “ago.”
NSHourNameDesignations	Strings that identify the time of day. These strings should be bound to an hour. The default is this array of arrays: (0, midnight), (12, noon, lunch), (10, morning), (14, afternoon), (19, dinner).
NSInternationalCurrencyString	A string containing three letter abbreviation for currency, following the ISO 4217 standard.
NSLaterTimeDesignations	An array of strings that denote a time in the future. This is an adjective that modifies a value from NSYearMonthWeekDesignations. The default is “next.”.
NSMonthNameArray	An array that specifies the full names for the months.
NSNextDayDesignations	A string that identifies the day after today. The default is “tomorrow.”
NSNextNextDayDesignations	A string that identifies the day after tomorrow. The default is “nextday”.
NSPriorDayDesignations	A string that identifies the day before today. The default is “yesterday.”
NSShortDateFormatString	A format string that specifies how dates are abbreviated. (See the

	NSCalendarDate class specification for a list of the date format specifiers to use.) The default is to separate the day month and year with slashes and to put the day first, as in 31/10/95.
NSShortMonthNameArray	An array that specifies the abbreviations for the months.
NSShortWeekDayNameArray	An array that specifies the abbreviations for the days of the week. Sunday should be the first day of the week.
NSShortTimeDateFormatString	A format string that specifies how times and dates are abbreviated. (See the NSCalendarDate class specification for a list of the date format specifiers.) The default is to use dashes to separate the day, month, and year and to use a 12-hour clock, as in "31-Jan-95 1:30 PM.
NSThisDayDesignations	A string that identifies what this day is called. The default is "today."
NSThousandsSeparator	A string that specifier the separator character for the thousands place of a decimal number. The default is a comma.
NSTimeDateFormatString	A format string how dates with times are printed.(See the NSCalendarDate class specification for a list of the date format specifiers.) The default is to use abbreviated months and days with a 24 hour clock, as in "Sun Jan 01 23:00:00 +6 2001."
NSTimeFormatString	A format string how dates with times are printed.(See the NSCalendarDate class specification for a list of the date format specifiers.) The default is to use a 24 hour clock, as in 13:30:25.
NSWeekDayNameArray	An array that gives the names for the days of the week. Sunday should be the first day of the week.
NSYearMonthWeekDesignations	An array of strings that specify the word for year, month, and week in the current locale. The defaults are "year," "month," and "week."

NSCoding

Adopted By: Various OpenStep classes

Declared In: Foundation/NSObject.h

Protocol Description

The NSCoding protocol declares the two methods that a class must implement so that instances of that class can be encoded and decoded. This capability provides the basis for archiving (where objects and other structures are stored on disk) and distribution (where objects are copied to different address spaces). See the NSCoder and NSArchiver class specifications for an introduction to coding.

In keeping with object-oriented design principles, an object being encoded or decoded is responsible for encoding and decoding its instance variables. A coder instructs the object to do so by invoking **encodeWithCoder:** or **initWithCoder:**. **encodeWithCoder:** instructs the object to encode its instance variables to the coder provided; an object can receive this method any number of times. **initWithCoder:** instructs the object to initialize itself from data in the coder provided; as such, it replaces any other initialization method and is only sent once per object. Any object class that should be codable must adopt the NSCoding protocol and implement its methods.

When an object receives an **encodeWithCoder:** message, it should encode all of its vital instance variables, after sending a message to **super** if its superclass also conforms to the NSCoding protocol. An object doesn't have to encode all of its instance variables. Some values may not be important to reestablish and others may be derivable from related state upon decoding. Other instance variables should be encoded only under certain conditions (for example, with **encodeConditionalObject:**, as described in the NSArchiver class specification).

For example, suppose you were creating a fictitious MapView class that displays a legend and a map at various magnifications. The MapView class defines several instance variables, including the name of the map and the current magnification. The MapView class also contains instance variables for several related views. The **encodeWithCoder:** method of MapView might look like the following:

```
- (void)encodeWithCoder:(NSCoder *)coder
{
    [super encodeWithCoder:coder];
    [coder encodeValueOfObjCType:@encode(char *) at:mapName];
    [coder encodeValueOfObjCType:@encode(unsigned int) at:&magnification];
    [coder encodeObject:legendView];
    [coder encodeConditionalObject:auxiliaryView];
    return;
}
```

This example assumes that the superclass of MapView also supports the NSCoding protocol. If the superclass of your class does not support NSCoding, you should omit the line that invokes super's **encodeWithCoder:** method.

encodeValueOfObjCType:at: and **encodeObject:** are coder methods that you can use to encode instance variables of your class. You can use these and other methods of the coder to encode id's, scalars, C arrays, structs, strings, and pointers to any of these types. The coder also defines corresponding methods for decoding values. See the NSCoder, NSArchiver, and NSUnarchiver class specifications for a list of methods.

The **@encode()** compiler directive generates an Objective-C type code from a type expression. See *Object-Oriented Programming and the Objective-C Language* for more information.

Similarly, in **initWithCoder:** the object should first send a message to **super** (if appropriate) to initialize inherited instance variables, and then it should decode and initialize its own. MapView's implementation of **initWithCoder:** might look like this:

```
- (id)initWithCoder:(NSCoder *)coder
{
    self = [super initWithCoder:coder];
    [coder decodeValueOfObjCType:@encode(char *) at:mapName];
    [coder decodeValueOfObjCType:@encode(unsigned int) at:&magnification];
    legendView = [[coder decodeObject] retain];
    auxiliaryView = [[coder decodeObject] retain];
    return self;
}
```

Note the assignment of the return value of **initWithCoder:** to **self** in the example above. This is done in the subclass because the superclass, in its implementation of **initWithCoder:**, may decide to return a object other than itself. If the superclass of your class does not support NSCoding, you should invoke super's designated initializer instead of **initWithCoder:**.

Making Substitutions During Coding

During encoding or decoding a coder object invokes methods that allow the object being coded to substitute a replacement class or instance for itself. This allows archives to be shared among implementations with different class hierarchies or simply different versions of a class (for example, class clusters take advantage of this feature). It also allows classes that should maintain unique instances to enforce this policy on decoding (for example, there need only be a single NSFont instance for a given typeface and size).

Substitution methods are declared by NSObject, and come in two flavors: generic and specialized. The generic methods are these:

Method	Typical Use
classForCoder	Allows an object, before being encoded, to substitute a class other than its own. For example, the private subclasses of a class cluster substitute the name of their public superclass when being archived.
replacementObjectForCoder:	Allows an object, before being encoded, to substitute another instance in its place.
awakeAfterUsingCoder:	Allows an object, after being decoded, to substitute another object for itself. For example, an object that represents a font might, upon being decoded, release itself and return an existing object having the same font description as itself. In this way, redundant objects can be eliminated.

The specialized substitution methods are analogous to **classForCoder** and **replacementObjectForCoder:** , but they're designed for (and invoked by) a specific, concrete coder subclass. NSArchiver invokes **classForArchiver:** and **replacementObjectForArchiver:**, while NSPortCoder invokes **classForPortCoder** and **replacementObjectForPortCoder:**. (There isn't a specialized version of **awakeAfterUsingCoder:**.) By implementing these specialized methods, your class can base its coding behavior on the specific coder class being used. For more information on these methods, see their method descriptions in the NSObject class specification, as well as the class description in the NSPortCoder class specification.

Method Types

Encoding and decoding objects

– encodeWithCoder:
– initWithCoder:

Instance Methods

encodeWithCoder:

– (void)**encodeWithCoder:**(NSCoder *)*encoder*

Encodes the receiver using *encoder*.

initWithCoder:

– (id)**initWithCoder:**(NSCoder *)*decoder*

Initializes a newly allocated instance from data in *decoder*. Returns **self**.

NSCopying

Adopted By: Various OpenStep classes

Declared In: Foundation/NSObject.h

Protocol Description

The NSCopying protocol declares a method for providing functional copies of an object. The exact meaning of “copy” can vary from class to class, but a copy must be a functionally independent object with values identical to the original at the time the copy was made. A copy produced with NSCopying is implicitly retained by the sender, who is responsible for releasing it.

NSCopying declares one method, **copyWithZone:**, but copying is commonly invoked with the convenience method **copy**. The **copy** method is defined for all NSObjects and simply invokes **copyWithZone:** with the default zone.

Using NSCopying

NSCopying is frequently used to copy *value* objects—objects that represent attributes. C-type variables can usually be substituted for value objects, but value objects have the advantage of encapsulating convenient utilities for common manipulations. For example, NSString objects are used instead of character pointers because they encapsulate encoding and storage. Despite NSString functionality, the role played by NSStrings parallels the role played by character pointers.

When value objects are passed as method arguments or returned from a method, it is common to use a copy instead of the object itself. For example, consider the following method for assigning a string to an object’s **name** instance variable.

```
- (void)setName:(NSString *)aName
{
    [name autorelease];
    name = [aName copy];
}
```

Storing a copy of **aName** has the effect of producing an object that’s independent of the original, but has the same contents. Subsequent changes to the copy don’t affect the original, and changes to the original don’t affect the copy. Similarly, it is common to return a copy of an instance variable instead of the instance variable itself. For example, this method returns a copy of the **name** instance variable:

```
- (NSString *)name
{
    return [[name copy] autorelease];
}
```

Implementing NSCopying

There are two basic approaches to creating copies. You can use **alloc** and **init...**, or you can use **NSCopyObject()**. To choose the one that's right for your class, you need to consider the following questions:

- "What kind of copying—deep or shallow—does your class need?"
- "Does your class's superclass implement NSCopying?"
- "Are you familiar with the implementations of your class's superclasses?"

These areas are described in the following sections.

What kind of copying—deep or shallow—does your class need?

Generally, copying an object involves creating a new instance and initializing it with the values in the original object. Copying the values for non-pointer instance variables, such as booleans, integers, and floating points, is straightforward. When copying pointer instance variables there are two approaches. One approach, called a *shallow copy,* copies the pointer value from the original object into the copy. Thus, the original and the copy share referenced data. The other approach, called a *deep copy,* duplicates the data referenced by the pointer and assigns it to the copy's instance variable.

The implementation of an instance variable's set method should reflect the kind of copying you need to use. You should deeply copy the instance variable if the corresponding set method copies the new value as in this method:

```
- (void)setMyVariable:(id)newValue
{
    [myVariable autorelease];
    myVariable = [newValue copy];
}
```

You should shallowly copy the instance variable if the corresponding set method retains the new value as in this method:

```
- (void)setMyVariable:(id)newValue
{
    [myVariable autorelease];
    myVariable = [newValue retain];
}
```

Similarly, you should shallowly copy the instance variable if its set method simply assigns the new value to the instance variable without copying or retaining it as in this method:

```
- (void)setMyVariable:(id)newValue
{
    myVariable = newValue;
}
```

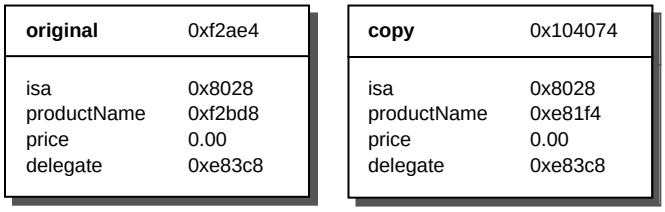
To produce a copy of an object that's truly independent of the original, the entire object must be deeply copied. Every instance variable must be duplicated. If the instance variables themselves have instance variables, those too must be duplicated, and so on. In many cases, a mixed approach is more useful. Pointer instance variables that can be thought of as data containers are generally deeply copied, while more sophisticated instance variables like delegates are shallowly copied.

For example, a Product class adopts NSCopying. Product instances have a name, a price, and a delegate as declared in this interface.

```
@interface Product : NSObject <NSCopying>
{
    NSString *productName;
    float price;
    id delegate;
}

@end
```

Copying a Product instance produces a deep copy of **productName** because it represents a flat data value. On the other hand, the **delegate** instance variable is a more complex object capable of functioning properly for both Products. The copy and the original should therefore share the delegate. The following figure represents the images of a Product instance and a copy in memory.



The different pointer values for **productName** illustrate that the original and the copy each have their own **productName** string object. The pointer values for **delegate** are the same, indicating that the two product objects share the same object as their delegate.

Does your class's superclass implement NSCopying?

If the superclass does not implement NSCopying, your class's implementation will have to copy the instance variables it inherits as well as those declared in your class. Generally, the safest way to do this is by using **alloc**, **init...**, and set methods. On the other hand, if your class inherits NSCopying behavior, its implementation only has to copy instance variables declared in your class. It invokes the superclass's implementation to copy inherited instance variables.

Are you familiar with the implementations of your class's superclasses?

If your class inherits NSCopying behavior, how you handle the new instance variables in **copyWithZone:** depends on your familiarity with the superclass's implementation. There are essentially two ways to make a copy of an object, using **alloc** and **init...** or using the function **NSCopyObject()**. If the superclass used or might have used **NSCopyObject()**, you must handle instance variables differently than you would otherwise.

Using the alloc, init... Approach

If a class does not inherit NSCopying behavior, you should implement **copyWithZone:** using **alloc**, **init...**, and set methods. For example, an implementation of **copyWithZone:** for the Product class described above might be implemented in the following way:

```
- (id)copyWithZone:(NSZone *)zone
{
    Product *copy = [[Product alloc]
           initWithProductName:[self productName]
           price:[self price]];
    [copy setDelegate:[self delegate]];

    return copy;
}
```

Because implementation details associated with inherited instance variables are encapsulated in the superclass, it is generally better to implement NSCopying with the **alloc**, **init...** approach. Doing so uses policy implemented in set methods to determine the kind of copying needed of instance variables.

Using NSCopyObject()

When a class inherits NSCopying behavior, you must consider the possibility that the superclass's implementation uses **NSCopyObject()**. **NSCopyObject()** creates an exact shallow copy of an object by copying instance variable values but not the data they point to. For example, NSCell's implementation of **copyWithZone:** could be defined in the following way.

```
- (id)copyWithZone:(NSZone *)zone
{
    NSCell *cellCopy = NSCopyObject(self, 0, zone);
    /* Assume that other initialization takes place here. */

    cellCopy->image = nil;
    [cellCopy setImage:[self image]];

    return cellCopy;
}
```

In the implementation above, **NSCopyObject()** creates an exact shallow copy of the original cell. This behavior is desirable for copying instance variables that aren't pointers or are pointers to non-retained data that is shallowly copied. Pointer instance variables for retained objects need additional treatment.

In the **copyWithZone:** example above, **image** is a pointer to a retained object. The policy to retain the image is reflected in the following implementation of the **setImage:** accessor method.

```
- (void)setImage:(NSImage *)anImage
{
    [image autorelease];
    image = [anImage retain];
}
```

Notice that **setImage:** autoreleases **image** before it reassigns it. If the above implementation of **copyWithZone:** hadn't explicitly set the copy's **image** instance variable to **nil** before invoking **setImage:**, the image referenced by the copy and the original would be released without a corresponding retain.

Even though **image** points to the right object, it is conceptually uninitialized. Unlike the instance variables that are created with **alloc** and **init...**, these uninitialized variables aren't **nil**-valued. You should explicitly assign initial values to these variables before using them. In this case, **cellCopy**'s **image** instance variable is set to **nil**, then it is set using the **setImage:** method.

The effects of **NSCopyObject()** extend to a subclass's implementation. For example, an implementation of NSSliderCell could copy a new **titleCell** instance variable in the following way.

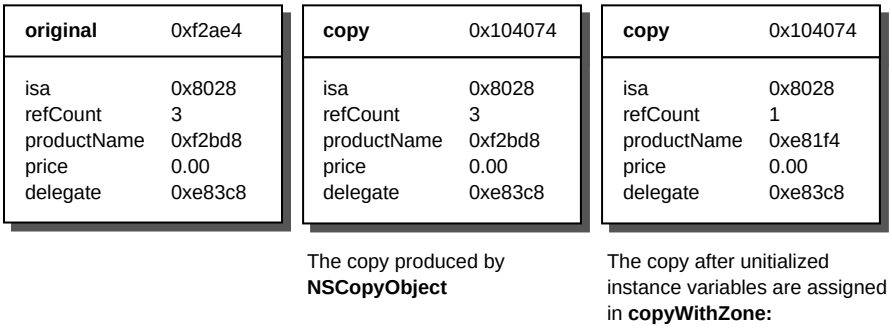
```
- (id)copyWithZone:(NSZone *)zone
{
    NSSliderCell *cellCopy = [super copyWithZone:zone];
    /* Assume that other initialization takes place here. */

    cellCopy->titleCell = nil;
    [cellCopy setTitleCell:[self titleCell]];

    return cellCopy;
}
```

The superclass's **copyWithZone:** method is invoked to copy inherited instance variables. When you invoke a superclass's **copyWithZone:** method, assume that new object instance variables are uninitialized if there's any chance that the superclass implementation uses **NSCopyObject()**. Explicitly assign a value to them before using them. In this example, **titleCell** is explicitly set to **nil** before **setTitleCell:** is invoked.

The implementation of an object's retain count is another consideration when using **NSCopyObject()**. If an object stores its retain count in an instance variable, the implementation of **copyWithZone:** must correctly initialize the copy's retain count. The following figure illustrates the process.



The copy produced by **NSCopyObject**

The copy after unitialized instance variables are assigned in **copyWithZone:**

The first object represents a Product instance in memory. The value in **refCount** indicates that the instance has been retained three times. The second object is a copy of the Product instance produced with **NSCopyObject()**. Its **refCount** value matches the original. The third object represents the copy returned from **copyWithZone:** after **refCount** is correctly initialized. After **copyWithZone:** creates the copy with **NSCopyObject()**, it assigns the value 1 to the **refCount** instance variable. The sender of **copyWithZone:** implicitly retains the copy and is responsible for releasing it.

NSCopying and Immutable Classes

Where the concept “immutable vs. mutable” applies to an object, NSCopying produces immutable copies whether the original is immutable or not. See the NSMutableCopying protocol for details on making mutable copies.

Immutable classes can implement NSCopying very efficiently. Since immutable objects don’t change, there is no need to duplicate them. Instead, NSCopying can be implemented to **retain** the original. For example, **copyWithZone:** for an immutable string class can be implemented in the following way.

```
- (id)copyWithZone:(NSZone *)zone
{
    return [self retain];
}
```

Summary

- Implement NSCopying using **alloc** and **init...** in classes that don’t inherit **copyWithZone:**.
- Implement NSCopying by invoking the superclass’s **copyWithZone:** when NSCopying behavior is inherited. If the superclass implementation might use **NSCopyObject()**, make explicit assignments to pointer instance variables for retained objects.
- Implement NSCopying by retaining the original instead of creating a new copy when the class and its contents are immutable.

Instance Methods

copyWithZone:

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

Returns a new instance that's a copy of the receiver. Memory for the new instance is allocated from *zone*, which may be NULL. If *zone* is NULL, the new instance is allocated from the default zone, which is returned from **NSDefaultMallocZone()**. The returned object is implicitly retained by the sender, who is responsible for releasing it. The copy returned is immutable if the consideration "immutable vs. mutable" applies to the receiving object; otherwise the exact nature of the copy is determined by the class.

See also: – **mutableCopyWithZone:** (NSMutableCopying protocol), – **copy** (NSObject)

NSDecimalNumberBehaviors

Adopted By: NSDecimalNumberHandler

Declared In: Foundation/NSDecimalNumber.h

Protocol Description

The NSDecimalBehaviors protocol declares three methods that control the discretionary aspects of working with NSDecimalNumbers. The **scale** and **roundingMode** methods determine the precision of NSDecimalNumber's return values, and the way in which those values should be rounded to fit that precision. The **exceptionDuringOperation:error:leftOperand:rightOperand:** determines the way in which an NSDecimalNumber should handle different calculation errors.

For an example of a class that adopts the NSDecimalBehaviors protocol, see the specification for NSDecimalNumberHandler.

Method Types

Rounding

– roundingMode
– scale

Handling errors

– exceptionDuringOperation:error:leftOperand:rightOperand:

Instance Methods

exceptionDuringOperation:error:leftOperand:rightOperand:

– (NSDecimalNumber *)**exceptionDuringOperation:**(SEL)*method*
error:(NSCalculationError)*error*
leftOperand:(NSDecimalNumber *)*leftOperand*
rightOperand:(NSDecimalNumber *)*rightOperand*

Specifies what an NSDecimalNumber will do when, in the course of applying *method* to *leftOperand* and *rightOperand*, it encounters *error*.

There are four possible values for *error*. The first three have to do with limits on NSDecimalNumber's ability to represent decimal numbers. An NSDecimalNumber can represent any number that can be

expressed as *mantissa* **x** 10**exponent**, where *mantissa* is a decimal integer up to 38 digits long, and *exponent* is between -256 and 256. If these limits are exceeded, the NSDecimalNumber returns one of the following errors:

- NSCalculationLossOfPrecision. The number can't be represented in 38 significant digits.
- NSCalculationOverflow. The number is too large to represent.
- NSCalculationUnderflow. The number is too small to represent.

The last error is simpler:

- NSCalculationDivideByZero. The caller tried to divide by zero.

In implementing **exceptionDuringOperation:error:leftOperand:rightOperand**, you can handle each of these errors in several ways:

- Raise an exception. For an explantion of exceptions, see the NSException class description in the *Foundation Framework Reference*.
- Return **nil**. The calling *method* will return its value as though no error had occurred. If *error* is NSCalculationLossOfPrecision, *method* will return an imprecise value—that is, one constrained to 38 significant digits. If *error* is NSCalculationUnderflow or NSCalculationOverflow, *method* will return NSDecimalNumber's **notANumber**. You shouldn't return **nil** if *error* is NSDivideByZero.
- Correct the error and return a valid NSDecimalNumber. The calling method will use this as its own return value.

roundingMode

– (NSRoundingMode)**roundingMode**

Returns the way that NSDecimalNumber's **decimalNumberBy...** methods round their return values. There are four possible NSRoundingModes:

- NSRoundDown. The methods round their return values down.
- NSRoundUp. The methods round their return values up.
- NSRoundPlain. The methods round to the closest possible return value. When they are caught halfway between two positive numbers, they round up; when caught between two negative numbers, they round down.
- NSRoundBankers. The methods round to the closest possible return value. When they are caught halfway between two possibilities, they return the possibility whose last digit is even. In practice, this means that, over the long run, numbers will be rounded up as often as they are rounded down; there will be no systematic bias.

The rounding mode only matters if the **scale** method sets a limit on the precision of NSDecimalNumber return values. It has no effect if **scale** returns NSDecimalNoScale.

Assuming that **scale** returns 1, the NSRoundingMode has the following effects on various original values:

Original value	NSRoundDown	NSRoundUp	NSRoundPlain	NSRoundBankers
1.24	1.2	1.3	1.2	1.2
1.26	1.2	1.3	1.3	1.3
1.25	1.2	1.3	1.3	1.2
1.35	1.3	1.4	1.4	1.4
-1.35	-1.4	-1.3	-1.4	-1.4

scale

– (short)**scale**

Limits the precision of the values returned by NSDecimalNumber's **decimalNumberBy...** methods.

Specifically, **scale** returns the number of digits allowed after the decimal separator. If **scale** returns a negative value, it affects the digits *before* the decimal separator as well. If **scale** returns NSDecimalNoScale, the number of digits is unlimited.

Assuming that **roundingMode** returns NSRoundPlain, different values of **scale** have the following effects on the number 123.456:

Scale	Return value
NSDecimalNoScale	123.456
2	123.45
0	123
-2	100

NSLocking

Adopted By: NSConditionLock
NSLock
NSRecursiveLock

Declared In: Foundation/NSLock.h

Protocol Description

The NSLocking protocol declares the elementary methods adopted by classes that define lock objects. A lock object is used to coordinate the actions of multiple threads of execution within a single application. By using a lock object, an application can protect critical sections of code from being executed simultaneously by separate threads, thus protecting shared data and other shared resources from corruption.

For example, consider a multithreaded application in which each thread updates a shared counter. If two threads simultaneously fetch the current value into local storage, increment it, and then write the value back, the counter will be incremented only once, losing one thread's contribution. However, if the code that manipulates the shared data (the *critical section* of code) must be locked before being executed, only one thread at a time can perform the updating operation, and collisions are prevented.

A lock object is either locked or unlocked. You *acquire* a lock by sending the object a **lock** message, thus putting the object in the locked state. You *relinquish* a lock by sending an **unlock** message, and thus putting the object in the unlocked state. (The NEXTSTEP classes that adopt this protocol define additional ways to acquire and relinquish locks.)

The **lock** method as declared in this protocol is *blocking*. That is, the thread that sends a **lock** message is blocked from further execution until the lock is acquired (presumably because some other thread relinquishes its lock). Classes that adopt this protocol typically add methods that offer nonblocking alternatives.

These NEXTSTEP classes conform to the NSLocking protocol:

Class	Adds these features to the basic protocol
NSLock	A nonblocking lock method; the ability to limit the duration of a locking attempt.
NSConditionLock	The ability to postpone entry to a critical section until a condition is met.
NSRecursiveLock	The ability for a single thread to acquire a lock more than once without deadlocking.

The locking mechanism that these classes use causes a thread to sleep while waiting to acquire a lock rather than to poll the system constantly. Thus, lock objects can be used to lock time-consuming operations without causing system performance to degrade. See the class specifications for these classes for further information on their behavior and usage.

There is some performance cost in acquiring a lock, so it's best to avoid the overhead if possible. An application developer has control over whether the application will execute with multiple threads, so it's clear when locking is appropriate. A library developer doesn't necessarily know whether library code will execute in a multithreaded environment. In this case, it's best to test whether the code is executing in a multithreaded environment before attempting to acquire a lock. The following example illustrates how this is done.

Assume your application uses a Counter object to record various operations. Here's one design that lets the Counter know whether it is multithreaded:

```
+ (void)initialize
{
    if ([NSThread isMultiThreaded]) {
        [self taskNowMultiThreaded:nil];
    } else {
        [[NSNotificationCenter defaultCenter] addObserver:self
            selector:@selector(taskNowMultiThreaded:)
            name:NSWillBecomeMultiThreadedNotification object:nil];
    }
}
```

In the initialize method (which, by definition, is invoked before any Counter objects are created), the Counter class object first checks whether the application has already become multithreaded and if so invokes its own **taskNowMultiThreaded:** method. Otherwise, it registers as an observer of NSWillBecomeMultiThreadedNotification so that **taskNowMultiThreaded:** will be invoked when the application becomes multithreaded.

Counter's **taskNowMultiThreaded:** method creates a lock object that the threads use to coordinate their activities:

```
+ (void)taskNowMultiThreaded:(NSNotification *)event
{
    if (!theLock)
        theLock = [[NSLock alloc] init];
}
```

theLock, a static variable declared in the class implementation file, is assigned a value of **nil** until **taskNowMultiThreaded:** is invoked. Since messages sent to **nil** are permitted and have no effect, code within Counter that acts on shared data can be written like this:

```
[theLock lock];
/* Operate on shared data */
[theLock unlock];
```

With this approach, the overhead associated with lock operations is only incurred if the application is multithreaded. This code, however, raises another issue. What happens if one of the statements between the **lock** and **unlock** messages cause the application to become multithreaded? Then the **unlock** message wouldn't be paired with the preceding **lock**.

In normal usage, locking and unlocking messages are paired. However, as in the example above, it might be convenient to unlock a lock object that hasn't yet been locked. This is permitted with two restrictions. First, you can send an unpaired unlocking message to a lock object as long as the object has never before been locked. Second, of the NEXTSTEP classes that conform to the NSLocking protocol, only NSConditionLock and NSLock allow an unpaired unlocking message. NSRecursiveLock requires locking and unlocking messages to be paired.

NEXTSTEP's locking classes are designed to work in a well-behaved, multithreaded environment: The protection they offer can be subverted by the use of signal handlers. A signal handler can interrupt a thread, execute code that affects shared data, and then let the thread resume without alerting the thread that the application has, in effect, become multithreaded. For this reason, it's recommended that you don't use signal handlers in multithreaded NEXTSTEP applications.

Instance Methods

lock

– (void)**lock**

Attempts to acquire a lock. This method blocks a thread's execution until the lock can be acquired.

An application protects a critical section of code by requiring a thread to acquire a lock before executing the code. Once the critical section is past, the thread relinquishes the lock by invoking **unlock**.

unlock

– (void)**unlock**

Relinquishes a previously acquired lock.

NSMutableCopying

Adopted By: Various OpenStep classes

Declared In: Foundation/NSObject.h

Protocol Description

The NSMutableCopying protocol declares a method for providing mutable copies of an object. Only classes that define an "immutable vs. mutable" distinction should adopt this protocol. Classes that don't define such a distinction should adopt NSCopying instead.

NSMutableCopying declares one method, **mutableCopyWithZone:**, but mutable copying is commonly invoked with the convenience method **mutableCopy**. The **mutableCopy** method is defined for all NSObjects and simply invokes **mutableCopyWithZone:** with the default zone.

See the NSCopying protocol for details on implementing copying behavior.

Instance Methods

mutableCopyWithZone:

– **mutableCopyWithZone:**(NSZone *)*zone*

Returns a new instance that's a mutable copy of the receiver. Memory for the new instance is allocated from *zone*, which may be NULL. If *zone* is NULL, the new instance is allocated from the default zone, which is returned by **NSDefaultMallocZone()**. The returned object is implicitly retained by the sender, who is responsible for releasing it. The copy returned is mutable whether the original is mutable or not.

See also: – **copyWithZone:** (NSCopying protocol), – **mutableCopy** (NSObject)

NSObjCTypeSerializationCallBack

Adopted By: No OpenStep classes

Declared In: Foundation/NSSerialization.h

Protocol Description

An object conforms to the NSObjCTypeSerializationCallBack protocol so that it can intervene in the serialization and deserialization process. The primary purpose of this protocol is to allow for the serialization of objects and other data types that aren't directly supported by OpenStep's serialization facility. (See the NSSerializer class specification for information on serialization.)

NSMutableData declares the method that's used to begin the serialization process:

```
- (void)serializeDataAt:(const void *)data
        ofObjCType:(const char *)type
        context:(id <NSObjCTypeSerializationCallBack>)callback
```

This method can serialize all standard Objective C types (**int**, **float**, character strings, and so on) except for objects, **union**, and **void ***. If, during the serialization process, an object is encountered, the object passed as the callback argument above is asked to provide the serialization.

Suppose that the type being serialized is a structure of this description:

```
struct stockRecord {
    NSString *stockName;
    float value;
};
```

The Objective C type code for this structure is {@f}, so the serialization process begins with this message: (Assume that **theData** is the NSMutableData object that's doing the serialization and **helper** is an object that conforms to the NSObjCTypeSerializationCallBack protocol.)

```
struct stockRecord aRecord = {@"aCompany", 34.7};

[theData serializeDataAt:&aRecord ofObjCType:"{@f}" context:helper];
```

Since the first field of the structure is an unsupported type, the helper object is sent a **serializeObjectAt: ofObjCType:intoData:** message, letting it serialize the object. **helper** might implement the method in this way:

```
- (void)serializeObjectAt:(id *)objectPtr
        ofObjCType:(const char *)type
        intoData:(NSMutableData *)theMutableData
{
    NSString *nameObject;
    char *companyName

    nameObject = *objectPtr;
    companyName = [nameObject cString];

    [theData serializeDataAt:&companyName ofObjCType:@encode(typeof(companyName))
           context:nil];
}
```

The callback object is free to serialize the target object as it wishes. In this case, **helper** simply extracts the company name from the NSString object and then has that character string serialized. Once this callback method finishes executing, the original method (**serializeDataAt:ofObjCType:context:**) resumes execution and serializes the second field of the structure. Since this second field contains a supported type (**float**), the callback method is not invoked again.

Deserialization follows a similar pattern, except in this case NSData declares the central method **deserializeDataAt:ofObjCType:atCursor:context:**. The deserialization of the example structure starts with a message to the NSData object that contains the serialized data:

```
(unsigned *)cursor = 0;

[theData deserializeDataAt:&aRecord ofObjCType:"{@f}" cursor:&cursor
        context:helper];
```

(The cursor argument is a pointer to zero since we're starting at the beginning of the data in the NSData object.)

When this method is invoked, the callback object receives a **deserializeObjectAt:ofObjCType:fromData: atCursor:** message, as declared in this protocol. The callback object can then reestablish the first field of the structure. For example, **helper** might implement the method in this way:

```
- (void) deserializeObjectAt:(id *)objectPtr
        ofObjCType:(const char *)type
        fromData:(NSData *)data
        atCursor:(unsigned *)cursor
{
    char *companyName;

    [theData deserializeDataAt:&companyName ofObjCType:"*" atCursor:cursor
           context:nil];
    *objectPtr = [[NSString stringWithCString:companyName] retain];
}
```

Instance Methods

deserializeObjectAt:ofObjCType:fromData:atCursor:

– (void)**deserializeObjectAt:**(id *)*object*
ofObjCType:(const char *)*type*
fromData:(NSData *)*data*
atCursor:(unsigned *)*cursor*

The implementor of this method decodes the referenced *object* (which should always be of *type* "@") located at the *cursor* position in the *data* object. The decoded object is not autoreleased.

See also: – **deserializeDataAt:ofObjCType:atCursor:context:** (NSData)

serializeObjectAt:ofObjCType:intoData:

– (void)**serializeObjectAt:**(id *)*object*
ofObjCType:(const char *)*type*
intoData:(NSMutableData *)*data*

The implementor of this method encodes the referenced *object* (which should always be of *type* "@") in the *data* object.

See also: – **serializeDataAt:ofObjCType:context:** (NSMutableData)

NSObject

Adopted By: NSObject

Declared In: Foundation/NSObject.h

Protocol Description

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

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

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

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

Method Types

Identifying classes

– class
– superclass

Identifying and comparing objects

– isEqual:
– hash
– self

Determining allocation zones

– zone

Managing reference counts

– retain
– release
– autorelease
– retainCount

Testing class functionality

– respondsToSelector:

Testing inheritance relationships

– isKindOfClass:
– isMemberOfClass:

Testing protocol conformance

– conformsToProtocol:

Describing objects

– description

Sending messages

– performSelector:
– performSelector:withObject:
– performSelector:withObject:withObject:

Identifying proxies

– isProxy

Instance Methods

autorelease

– (id)**autorelease**

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

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

class

– (Class)**class**

Returns the class object for the receiver's class.

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

conformsToProtocol:

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

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

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

description

– (NSString *)**description**

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

hash

– (unsigned)**hash**

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

If a mutable object is added to a collection that uses hash values to determine the object's position in the collection, the value returned by the **hash** method of the object must not change while the object is in the collection. To accomplish this, either the **hash** method must not rely on any of the object's internal state information or you must make sure that the object's internal state information does not change while the object is in the collection. (Note that it can be difficult to know whether or not a given object is in a collection.)

isEqual:

– (BOOL)**isEqual:**(id)*anObject*

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

isKindOfClass:

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

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

```
NSMutableData *myData = [NSMutableData dataWithCapacity:30];
id anArchiver = [[NSArchiver alloc] initForWritingWithMutableData:myData];
if ( [anArchiver isKindOfClass:[NSCoder class]] )
    ...
```

Because instance methods defined in the root class can be sent to class objects, when the receiver is a class object, this method returns YES if *aClass* is NSObject, NO otherwise.

See also: – **isMemberOfClass:**

isMemberOfClass:

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

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

```
NSMutableData *myData = [NSMutableData dataWithCapacity:30];
id anArchiver = [[NSArchiver alloc] initForWritingWithMutableData:myData];
if ([anArchiver isMemberOfClass:[NSCoder class]])
    ...
```

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

See also: – **isKindOfClass:**

isProxy

– (BOOL)**isProxy**

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

performSelector:

– (id)**performSelector:**(SEL)*aSelector*

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

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

However, the **performSelector:** 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 performSelector:myMethod];
```

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

See also: – **performSelector:withObject:**, – **performSelector:withObject:withObject:**

performSelector:withObject:

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

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

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

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

performSelector:withObject:withObject:

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

Sends the receiver an *aSelector* message with *anObject* and *anotherObject* as arguments. If *aSelector* is NULL, an NSInvalidArgumentException is raised. This method is the same as **performSelector:** except

that you can supply two arguments for *aSelector*. *aSelector* should identify a method that can take two arguments of type **id**. For methods with other argument types and return values use NSInvocation.

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

release

– (oneway void)**release**

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

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

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

See also: – **retainCount**

respondsToSelector:

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

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

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

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

retain

– (id)**retain**

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

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

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

```
NSString *systemApps = [[NSString
    stringWithCString:"/System/Applications"] retain];
```

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

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

retainCount

– (unsigned)**retainCount**

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

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

self

– (id)**self**

Returns the receiver.

See also: – **class**

superclass

– (Class)**superclass**

Returns the class object for the receiver's superclass.

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

zone

– (NSZone *)**zone**

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

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

Defined Types

NSBTreeComparator

DECLARED IN Foundation/NSByteStore.h

SYNOPSIS

typedef int **NSBTreeComparator**(NSData *, NSData *, const void *);

NSComparisonResult

DECLARED IN Foundation/NSObject.h

SYNOPSIS

typedef enum _NSComparisonResult {
    NSOrderedAscending = -1,
    NSOrderedSame,
    NSOrderedDescending
} **NSComparisonResult**;

DESCRIPTION

An **NSComparisonResult** indicates how items in a request are ordered, from the first one given in a method invocation or function call to the last (that is, left-to-right in code).

NSHashTableCallBacks

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS

```
typedef struct {
        unsigned (*hash)(NSHashTable *table, const void *);
        BOOL (*isEqual)(NSHashTable *table, const void *, const void *);
        void (*retain)(NSHashTable *table, const void *);
        void (*release)(NSHashTable *table, void *);
        NSString *(*describe)(NSHashTable *table, const void *);
} NSHashTableCallBacks;
```

NSMapTableKeyCallBacks

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS

```
typedef struct {
        unsigned (*hash)(NSMapTable *table, const void *);
        BOOL (*isEqual)(NSMapTable *table, const void *, const void *);
        void (*retain)(NSMapTable *table, const void *);
        void (*release)(NSMapTable *table, void *);
        NSString *(*describe)(NSMapTable *table, const void *);
         const void *notAKeyMarker;
} NSMapTableKeyCallBacks;
```

NSMapTableValueCallBacks

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS

```
typedef struct {
        void (*retain)(NSMapTable *table, const void *);
        void (*release)(NSMapTable *table, void *);
        NSString *(*describe)(NSMapTable *table, const void *);
} NSMapTableValueCallBacks;
```

NSNotificationCoalescing

DECLARED IN Foundation/NSNotificationQueue.h

SYNOPSIS

```
typedef enum {
    NSNotificationNoCoalescing = 0,
    NSNotificationCoalescingOnName = 1,
    NSNotificationCoalescingOnSender = 2,
} NSNotificationCoalescing;
```

NSObjCValue

DECLARED IN Foundation/NSInvocation.h

SYNOPSIS

```
typedef struct {
    enum _NSObjCValueType type;
    union {
        char charValue;
        short shortValue;
        long longValue;
        long long longlongValue;
        float floatValue;
        double doubleValue;
        SEL selectorValue;
        id objectValue;
        void *pointerValue;
        void *structLocation;
        char *cStringLocation;
    } value;
} NSObjCValue;
```

NSPoint

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS

```
typedef struct _NSPoint {
    float x;
    float y;
} NSPoint;
```

NSPosixFileActivity

DECLARED IN Foundation/NSPosixFileDescriptor.h

SYNOPSIS

```
typedef enum {
        NSPosixNoActivity = 0,
        NSPosixReadableActivity = 1,
        NSPosixWritableActivity = 2,
        NSPosixExceptionalActivity = 4
} NSPosixFileActivity;
```

NSPostingStyle

DECLARED IN Foundation/NSNotificationQueue.h

SYNOPSIS

```
typedef enum {
        NSPostWhenIdle = 1,
        NSPostASAP = 2,
        NSPostNow = 3
} NSPostingStyle;
```

NSRange

DECLARED IN Foundation/NSRange.h

SYNOPSIS

```
typedef struct _NSRange {
    unsigned int location;
    unsigned int length;
} NSRange;
```

DESCRIPTION An **NSRange** describes a portion of a series—such as characters in a string or objects in an NSArray. Its **location** member gives the start index (0 is the first, as in C arrays), and its **length** member gives the number of items in the range (and can be zero).

NSRect

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS

```
typedef struct _NSRect {
    NSPoint origin;
    NSSize size;
} NSRect;
```

NSRectEdge

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS

```
typedef enum _NSRectEdge {
    NSMinXEdge = 0,
    NSMinYEdge = 1,
    NSMaxXEdge = 2,
    NSMaxYEdge = 3
} NSRectEdge;
```

DESCRIPTION This enumeration identifies the sides of a rectangle represented by an NSRect.

NSSize

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS

```
typedef struct _NSSize {
    float width;
    float height;
} NSSize;
```

DESCRIPTION The values for width and height should never be negative.

NSStringEncoding

DECLARED IN Foundation/NSString.h

SYNOPSIS typedef unsigned **NSStringEncoding**;

NSTimeInterval

DECLARED IN Foundation/NSDate.h

SYNOPSIS typedef double **NSTimeInterval**;

DESCRIPTION Always in seconds; yields sub-millisecond precision over a range of 10000 years.

NSUncaughtExceptionHandler

DECLARED IN Foundation/NSException.h

SYNOPSIS typedef volatile void **NSUncaughtExceptionHandler**(NSException ***exception**);

Enumerations

NSByteOrder

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS

```
enum NSByteOrder {
        NS_UnknownByteOrder,
        NS_LittleEndian,
        NS_BigEndian
};
```

DESCRIPTION

NSNotFound

DECLARED IN Foundation/NSObjCRuntime.h

SYNOPSIS

```
enum {
        NSNotFound = 0x7fffffff
};
```

DESCRIPTION A method or function return value of **NSNotFound** indicates that the item requested couldn't be found or doesn't exist. It's typically used by various methods and functions that search for items in serial data and return indices, such as characters in a string object or **id**s in an NSArray.

NSOpenStepUnicodeReservedBase

DECLARED IN Foundation/NSCharacterSet.h

SYNOPSIS

```
enum {
    NSOpenStepUnicodeReservedBase = 0xF400
};
```

DESCRIPTION

ObjC Types

DECLARED IN Foundation/NSInvocation.h

SYNOPSIS

```
enum _NSObjCValueType {
    NSObjCNoType = 0,
    NSObjCVoidType = 'v',
    NSObjCCharType = 'c',
    NSObjCShortType = 's',
    NSObjCLongType = 'l',
    NSObjCLonglongType = 'q',
    NSObjCFloatType = 'f',
    NSObjCDoubleType = 'd',
    NSObjCSelectorType = ':',
    NSObjCObjectType = '@',
    NSObjCStructType = '{',
    NSObjCPointerType = '^',
    NSObjCStringType = '*',
    NSObjCArrayType = '[',
    NSObjCUnionType = '(',
    NSObjCBitfield = 'b',
};
```

Search Types

DECLARED IN Foundation/NSString.h

SYNOPSIS

```
enum {
    NSCaseInsensitiveSearch = 1,
    NSLiteralSearch = 2,
    NSBackwardsSearch = 4,
    NSAnchoredSearch = 8
};
```

DESCRIPTION These values represent the options available to many of the string classes' searching and comparison methods. See the NSString class cluster description for details on the effects of these options.

String Encodings

DECLARED IN Foundation/NSString.h

SYNOPSIS

```
enum {
    NSASCIIStringEncoding = 1,
    NSNEXTSTEPStringEncoding = 2,
    NSJapaneseEUCStringEncoding = 3,
    NSUTF8StringEncoding = 4,
    NSISOLatin1StringEncoding = 5,
    NSSymbolStringEncoding = 6,
    NSNonLossyASCIIStringEncoding = 7,
    NSShiftJISStringEncoding = 8,
    NSISOLatin2StringEncoding = 9,
    NSUnicodeStringEncoding = 10,
    NSWindowsCP1251StringEncoding = 11,
    NSWindowsCP1252StringEncoding = 12,
    NSWindowsCP1253StringEncoding = 13,
    NSWindowsCP1254StringEncoding = 14,
    NSWindowsCP1250StringEncoding = 15,
    NSISO2022JPStringEncoding = 21
};
```

DESCRIPTION These values represent the various character encodings supported by the NSString classes. These encodings are documented more fully elsewhere; here are some short descriptions:

Encoding	Description
NSASCIIStringEncoding	Strict 7-bit ASCII encoding within 8-bit chars
NSNEXTSTEPStringEncoding	8-bit ASCII encoding with NEXTSTEP extensions (see Appendix C of the NEXTSTEP General Reference)
NSUnicodeStringEncoding	The canonical Unicode encoding for string objects
NSISOLatin1StringEncoding	8-bit ISO Latin 1 encoding
NSISOLatin2StringEncoding	8-bit ISO Latin 2 encoding
NSSymbolStringEncoding	8-bit Adobe Symbol encoding vector
NSJapaneseEUCStringEncoding	8-bit EUC encoding for Japanese text
NSShiftJISStringEncoding	8-bit Shift-JIS encoding for Japanese text
NSUTF8StringEncoding	An 8-bit representation of Unicode characters, suitable for transmission or storage by ASCII-based systems
NSNonLossyASCIIStringEncoding	7-bit verbose ASCII to represent all Unicode characters.
NSWindowsCP1250StringEncoding	Microsoft Windows codepage 1250; equivalent to WinLatin2
NSWindowsCP1251StringEncoding	Microsoft Windows codepage 1251, encoding Cyrillic characters; equivalent to AdobeStandardCyrillic font encoding
NSWindowsCP1252StringEncoding	Microsoft Windows codepage 1252; equivalent to WinLatin1
NSWindowsCP1253StringEncoding	Microsoft Windows codepage 1253, encoding Greek characters
NSWindowsCP1254StringEncoding	Microsoft Windows codepage 1254, encoding Turkish characters
NSISO2022JPStringEncoding	ISO 2022 Japanese encoding for electronic mail

Global Variables

Archiving Exception Names

DECLARED IN Foundation/NSArchiver.h

SYNOPSIS extern NSString ***NSInconsistentArchiveException**;

Bundle Notification Names

DECLARED IN Foundation/NSBundle.h

SYNOPSIS extern NSString ***NSBundleLoaded**;

Byte Store Exception Names

DECLARED IN Foundation/NSByteStore.h

SYNOPSIS extern NSString ***NSByteStoreLockedException**;
extern NSString ***NSByteStoreVersionException**;
extern NSString ***NSBTreeStoreKeyTooLargeException**;
extern NSString ***NSByteStoreDamagedException**;

Connection Exception Names

DECLARED IN Foundation/NSConnection.h

SYNOPSIS extern NSString ***NSFailedAuthenticationException**;

Connection Notification Names

DECLARED IN Foundation/NSConnection.h

SYNOPSIS extern NSString ***NSConnectionDidDieNotification**;

File Attribute Keys

DECLARED IN Foundation/NSFileManager.h

SYNOPSIS
extern NSString ***NSFileType**;
extern NSString ***NSFileTypeDirectory**;
extern NSString ***NSFileTypeRegular**;
extern NSString ***NSFileTypeSymbolicLink**;
extern NSString ***NSFileTypeSocket**;
extern NSString ***NSFileTypeCharacterSpecial**;
extern NSString ***NSFileTypeBlockSpecial**;
extern NSString ***NSFileTypeUnknown**;
extern NSString ***NSFileSize**;
extern NSString ***NSFileModificationDate**;
extern NSString ***NSFileReferenceCount**;
extern NSString ***NSFileDeviceIdentifier**;
extern NSString ***NSFileOwnerAccountNumber**;
extern NSString ***NSFileGroupOwnerAccountNumber**;
extern NSString ***NSFilePosixPermissions**;
extern NSString ***NSFileSystemNumber**;
extern NSString ***NSFileSystemFileNumber**;

DESCRIPTION Keys to access the file attribute values contained in the NSDictionary returned from NSFileManager's **fileAttributesAtPath:follow:**. See the class specification for NSFileManager for details of usage. **NSFileDeviceIdentifier** is used to access the identifier of a remote device.

File Exception Names

DECLARED IN Foundation/NSPosix.h

SYNOPSIS extern NSString ***NSPosixFileOperationException**;

File System Attribute Keys

DECLARED IN Foundation/NSFileManager.h

SYNOPSIS extern NSString ***NSFileSystemSize**;
extern NSString ***NSFileSystemFreeSize**;
extern NSString ***NSFileSystemNodes**;
extern NSString ***NSFileSystemFreeNodes**;

General Exception Names

DECLARED IN Foundation/NSException.h

SYNOPSIS
extern NSString ***NSGenericException**;
extern NSString ***NSRangeException**;
extern NSString ***NSInvalidArgumentException**;
extern NSString ***NSInternalInconsistencyException**;
extern NSString ***NSMallocException**;
extern NSString ***NSObjectInaccessibleException**;
extern NSString ***NSObjectNotAvailableException**;
extern NSString ***NSDestinationInvalidException**;
extern NSString ***NSPortTimeoutException**;
extern NSString ***NSInvalidSendPortException**;
extern NSString ***NSInvalidReceivePortException**;
extern NSString ***NSPortSendException**;
extern NSString ***NSPortReceiveException**;
extern NSString ***NSOldStyleException**;
extern NSString ***NSInvalidSendPort**;
extern NSString ***NSInvalidReceivePort**;
extern NSString ***NSPortSendError**;
extern NSString ***NSPortReceiveError**;
extern NSString ***NSCharacterConversionException**;

Hash Table Call Backs

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS
extern const NSHashTableCallBacks **NSIntHashCallBacks**;
extern const NSHashTableCallBacks **NSNonOwnedPointerHashCallBacks**;
extern const NSHashTableCallBacks **NSOwnedPointerHashCallBacks**;
extern const NSHashTableCallBacks **NSPointerToStructHashCallBacks**;
extern const NSHashTableCallBacks **NSObjectHashCallBacks**;
extern const NSHashTableCallBacks **NSNonRetainedObjectHashCallBacks**;;

DESCRIPTION **NSIntHashCallBacks** are for sets of pointer-sized quantities or samller (for example, ints, longs, or unichars).

NSNonOwnedPointerHashCallBacks are for sets of pointers, hashed by address.

NSOwnedPointerHashCallBacks are for sets of pointers, with transfer of ownership upon insertion.

NSPointerToStructHashCallBacks are for sets of pointers to structs, when the first field of the struct is int-sized.

NSObjectHashCallBacks are for sets of objects (similar to NSSet)

NSNonRetainedObjectHashCallBacks are for sets of objects, but without retaining/releasing.

Note that you can make your own call back by picking fields among the above call backs.

Language-Dependent Information

DECLARED IN Foundation/NSUserDefaults.h

SYNOPSIS

```
extern NSString *NSWeekDayNameArray;
extern NSString *NSShortWeekDayNameArray;
extern NSString *NSMonthNameArray;
extern NSString *NSShortMonthNameArray;
extern NSString *NSTimeFormatString;
extern NSString *NSDateFormatString;
extern NSString *NSTimeDateFormatString;
extern NSString *NSShortTimeDateFormatString;
extern NSString *NSCurrencySymbol;
extern NSString *NSDecimalSeparator;
extern NSString *NSThousandsSeparator;
extern NSString *NSInternationalCurrencyString;
extern NSString *NSCurrencyString;
extern NSString *NSDecimalDigits;
extern NSString *NSAMPMDesignation;
extern NSString *NSHourNameDesignations;
extern NSString *NSYearMonthWeekDesignations;
extern NSString *NSEarlierTimeDesignations;
extern NSString *NSLaterTimeDesignations;
extern NSString *NSThisDayDesignations;
extern NSString *NSNextDayDesignations;
extern NSString *NSNextNextDayDesignations;
extern NSString *NSPriorDayDesignations;
extern NSString *NSDateTimeOrdering;
```

Map Table Key Call Backs

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS
extern const NSMapTableKeyCallBacks **NSIntMapKeyCallBacks**;
extern const NSMapTableKeyCallBacks **NSNonOwnedPointerMapKeyCallBacks**;
extern const NSMapTableKeyCallBacks **NSNonOwnedPointerOrNullMapKeyCallBacks**;
extern const NSMapTableKeyCallBacks **NSOwnedPointerMapKeyCallBacks**;
extern const NSMapTableKeyCallBacks **NSObjectMapKeyCallBacks**;
extern const NSMapTableKeyCallBacks **NSNonRetainedObjectMapKeyCallBacks**;

DESCRIPTION **NSIntMapKeyCallBacks** are for keys that are pointer-sized quantities or smaller (for example, ints, longs, or unichars).

NSNonOwnedPointerMapKeyCallBacks are for keys that are pointers not freed.

NSNonOwnedPointerOrNullMapKeyCallBacks are for keys that are pointers not freed, or NULL.

NSOwnedPointerMapKeyCallBacks are for keys that are pointers, with transfer of ownership upon insertion.

NSObjectMapKeyCallBacks are for keys that are objects

NSNonRetainedObjectMapKeyCallBacks are for sets of objects, but without retaining/releasing.

Note that you can make your own call back by picking fields among the above call backs.

Map Table Value Call Backs

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS
extern const NSMapTableValueCallBacks **NSIntMapValueCallBacks**;
extern const NSMapTableValueCallBacks **NSNonOwnedPointerMapValueCallBacks**;
extern const NSMapTableValueCallBacks **NSOwnedPointerMapValueCallBacks**;
extern const NSMapTableValueCallBacks **NSObjectMapValueCallBacks**;

DESCRIPTION **NSIntMapValueCallBacks** are for values that are pointer-sized quantities, such as ints.

NSNonOwnedPointerMapValueCallBacks are for values that are not owned pointers.

NSOwnedPointerMapValueCallBacks are for values that are owned pointers.

NSObjectMapValueCallBacks are for values that are objects.

Note that you can make your own call back by picking fields among the above call backs.

NSConnectionReplyMode

DECLARED IN **NSConnection.h**

SYNOPSIS extern NSString ***NSConnectionReplyMode**;

NSDefaultRunLoopMode

DECLARED IN **Foundation/NSRunLoop.h**

SYNOPSIS extern NSString ***NSDefaultRunLoopMode**;

Port Notification Names

DECLARED IN Foundation/NSPort.h

SYNOPSIS extern NSString ***NSPortDidBecomeInvalidNotification**;

PPL Notification Names

DECLARED IN Foundation/**NPPL.h**

SYNOPSIS
extern NSString ***NSPPLDidBecomeDirtyNotification**;
extern NSString ***NSPPLDidSaveNotification**;

Thread Notification Names

DECLARED IN Foundation/**NSThread.h**

SYNOPSIS
extern NSString ***NSBecomingMultiThreaded**;
extern NSString ***NSThreadExiting**;

User Defaults

DECLARED IN Foundation/NSUserDefaults.h

SYNOPSIS
extern NSString ***NSGlobalDomain**;
extern NSString ***NSArgumentDomain**;
extern NSString ***NSRegistrationDomain**;

Zero Constants

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS
extern const NSPoint **NSZeroPoint**;
extern const NSSize **NSZeroSize**;
extern const NSRect **NSZeroRect**;

Functions and Macros

This section describes the functions and macros found in the Foundation Kit.

NSAllHashTableObjects

SUMMARY This function returns all of the elements in the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS NSArray ***NSAllHashTableObjects**(NSHashTable **table*)

DESCRIPTION Returns an array object containing all the elements of *table*. This function should be called only when the table elements are objects, not when they're any other data type.

SEE ALSO **NSCreateHashTable, NSFreeHashTable**

NSAllMapTableKeys

SUMMARY This function returns all of the keys in the specified map table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSArray ***NSAllMapTableKeys**(NSMapTable **table*)

DESCRIPTION **NSAllMapTableKeys** Returns an array object containing all the keys in *table*. This function should be called only when the table keys are objects, not when they're any other type of pointer.

SEE ALSO **NSMapMember, NSMapGet, NSEnumerateMapTable, NSNextMapEnumeratorPair, NSAllMapTableValues**

NSAllMapTableValues

SUMMARY This function returns all of the values in the specified table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSArray ***NSAllMapTableValues**(NSMapTable **table*)

DESCRIPTION **NSAllMapTableValues** Returns an array object containing all the values in *table*. This function should be called only when the table values are objects, not when they're any other type of pointer.

SEE ALSO **NSMapMember, NSMapGet, NSEnumerateMapTable, NSNextMapEnumeratorPair, NSAllMapTableKeys**

NSAllocateMemoryPages

SUMMARY This function allocates a new block of memory.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void ***NSAllocateMemoryPages**(unsigned int *byteCount*)

DESCRIPTION **NSAllocateMemoryPages** allocates the integral number of pages whose total size is closest to, but not less than, *byteCount*. The allocated pages are guaranteed to be filled with zeros.

SEE ALSO **NSCopyMemoryPages, NSDeallocateMemoryPages**

NSAllocateObject

SUMMARY This function creates new objects.

DECLARED IN Foundation/NSObject.h

SYNOPSIS id <NSObject> **NSAllocateObject(**Class *aClass*, unsigned int *extraBytes*, NSZone **zone***)**

DESCRIPTION NSAllocateObject allocates and returns a pointer to an instance of *aClass*, created in the specified *zone* (or in the default zone, if *zone* is NULL). The *extraBytes* argument (usually zero) states the number of extra bytes required for indexed instance variables. Returns **nil** on failure.

SEE ALSO **NSCopyObject, NSDeallocateObject**

NSAssert

SUMMARY This macro generates an assertion if the given condition is false.

DECLARED IN Foundation/NSExceptions.h

SYNOPSIS **NSAssert(***condition***,** NSString ***description***)**

DESCRIPTION Assertions evaluate a condition and, if the condition evaluates to false, call the assertion handler for the current thread, passing it a format string and a variable number of arguments. Each thread has its own assertion handler, which is an object of class NSAssertionHandler. When invoked, an assertion handler prints an error message that includes the method and class names (or the function name). It then raises an exception of type NSInternalInconsistencyException.

The **NSAssert** macro evaluates the condition and serves as a front end to the assertion handler. This macro should be used only within Objective-C methods. **NSAssert** takes no arguments other than the condition and format string.

condition must be an expression that evaluates to true or false. *description* is a **printf**-style format string that contains the error message describing the failure condition.

Assertions are disabled if the preprocessor macro **NS_BLOCK_ASSERTIONS** is defined. All macros return **void**.

SEE ALSO **NSLog, NSLogv, NSAssert1, NSCAssert, NSCParameterAssert, NSParameterAssert**

NSAssert1

SUMMARY NSAssert1 is one of a series of macros, as listed in the SYNOPSIS section, that generate assertions if the given condition is false.

DECLARED IN Foundation/NSExceptions.h

SYNOPSIS
NSAssert1(*condition***,** NSString ******description, arg1* **)**
NSAssert2(*condition***,** NSString ******description, arg1, arg2***)**
NSAssert3(*condition***,** NSString ******description, arg1***,** *arg2***,** *arg3***)**
NSAssert4(*condition***,** NSString ******description, arg1, arg2***,** *arg3***,** *arg4***)**
NSAssert5(*condition***,** NSString ******description, arg1***,** *arg2***,** *arg3***,** *arg4***,** *arg5***)**

DESCRIPTION Assertions evaluate a condition and, if the condition evaluates to false, call the assertion handler for the current thread, passing it a format string and a variable number of arguments. Each thread has its own assertion handler, which is an object of class NSAssertionHandler. When invoked, an assertion handler prints an error message that includes the method and class names (or the function name). It then raises an exception of type NSInternalInconsistencyException.

The **NSAssert*n*** macros evaluate the condition and serve as a front end to the assertion handler. These macros should be used only within Objective-C methods. **NSAssert*n*** takes the number of format-string arguments indicated by *n*.

condition must be an expression that evaluates to true or false. *description* is a **printf**-style format string that contains the error message describing the failure condition. Each *arg* parameter is an argument to be inserted, in place, into the *description*.

Assertions are disabled if the preprocessor macro **NS_BLOCK_ASSERTIONS** is defined. All macros return **void**.

SEE ALSO **NSLog, NSLogv, NSAssert, NSCAssert, NSCParameterAssert, NSParameterAssert**

NSAssert2

SUMMARY See **NSAssert1**.

NSAssert3

SUMMARY See **NSAssert1**.

NSAssert4

SUMMARY See **NSAssert1**.

NSAssert5

SUMMARY See **NSAssert1**.

NSCAssert

SUMMARY This macro generates an assertion if the given condition is false.

DECLARED IN Foundation/NSExceptions.h

SYNOPSIS **NSCAssert(***condition***,** NSString ***description***)**

DESCRIPTION Assertions evaluate a condition and, if the condition evaluates to false, call the assertion handler for the current thread, passing it a format string and a variable number of arguments. Each thread has its own assertion handler, which is an object of class NSAssertionHandler. When invoked, an

assertion handler prints an error message that includes method and class names (or the function name). It then raises an exception of type NSInternalInconsistencyException.

The **NSCAssert** macro evaluates the condition and serves as a front end to the assertion handler. This macro should be used only within C functions. **NSCAssert** takes no arguments other than the condition and format string.

condition must be an expression that evaluates to true or false. *description* is a **printf**-style format string that describes the failure condition.

Assertions are disabled if the preprocessor macro **NS_BLOCK_ASSERTIONS** is defined. All macros return **void**.

SEE ALSO **NSLog, NSLogv, NSAssert, NSCAssert1, NSCParameterAssert, NSParameterAssert**

NSCAssert1

SUMMARY NSCAssert1 is one of a series of macros, as listed in the SYNOPSIS section, that generate assertions if the given condition is false.

DECLARED IN Foundation/NSExceptions.h

SYNOPSIS
NSCAssert1(*condition***,** NSString ******description, arg1***)**
NSCAssert2(*condition***,** NSString ******description, arg1, arg2***)**
NSCAssert3(*condition***,** NSString ******description, arg1***,** *arg2***,** *arg3***)**
NSCAssert4(*condition***,** NSString ******description, arg1, arg2***,** *arg3***,** *arg4***)**
NSCAssert5(*condition***,** NSString ******description, arg1***,** *arg2***,** *arg3***,** *arg4***,** *arg5***)**

DESCRIPTION Assertions evaluate a condition and, if the condition evaluates to false, call the assertion handler for the current thread, passing it a format string and a variable number of arguments. Each thread has its own assertion handler, which is an object of class NSAssertionHandler. When invoked, an assertion handler prints an error message that includes method and class names (or the function name). It then raises an exception of type NSInternalInconsistencyException.

The **NSCAssert*n*** macros evaluate the condition and serve as a front end to the assertion handler. These macros should be used only within Objective-C methods. **NSCAssert*n*** takes the number of format-string arguments indicated by *n*.

condition must be an expression that evaluates to true or false. *description* is a **printf**-style format string that describes the failure condition. Each *arg* is an argument to be inserted, in place, into the *description*.

Assertions are disabled if the preprocessor macro **NS_BLOCK_ASSERTIONS** is defined. All macros return **void**.

SEE ALSO **NSLog, NSLogv, NSCAssert, NSCParameterAssert, NSParameterAssert**

NSCAssert2

SUMMARY See **NSCAssert1**.

NSCAssert3

SUMMARY See **NSCAssert1**.

NSCAssert4

SUMMARY See **NSCAssert1**.

NSCAssert5

SUMMARY See **NSCAssert1**.

NSClassFromString

SUMMARY This function obtains a class by name.

DECLARED IN Foundation/NSObjCRuntime.h

SYNOPSIS Class **NSClassFromString**(NSString ***aClassName*)

DESCRIPTION **NSClassFromString** returns the class object named by *aClassName*, or **nil** if no class by that name is currently loaded.

SEE ALSO **NSStringFromClass**

NSCompareHashTables

SUMMARY This function compares the elements of two hash tables for equality.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS BOOL **NSCompareHashTables**(NSHashTable ***table1*, NSHashTable ***table2*)

DESCRIPTION Returns YES if the two hash tables are equal—that is, if each element of *table1* is in *table2*, and the two tables are the same size.

SEE ALSO **NSCreateHashTable, NSCreateHashTableWithZone**

NSCompareMapTables

SUMMARY This function compares the elements of two map tables for equality.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS BOOL **NSCompareMapTables**(NSMapTable ***table1*, NSMapTable ***table2*)

DESCRIPTION Returns YES if each key of *table1* is in *table2*, and the two tables are the same size. Note that this function does not compare values, only keys.

SEE ALSO **NSCreateMapTable, NSCreateMapTableWithZone**

NSContainsRect

SUMMARY This function determines whether one rectangle completely encloses another.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSContainsRect**(NSRect *aRect*, NSRect *bRect*)

DESCRIPTION Returns YES if *aRect* completely encloses *bRect*. For this condition to be true, *bRect* cannot be empty and its sides must not touch the sides of *aRect*.

NSConvertHostDoubleToSwapped

SUMMARY This function performs a type conversion.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedDouble **NSConvertHostDoubleToSwapped**(double *x*)

DESCRIPTION Converts the double value in *x* to a value whose bytes can be swapped. This function does not actually swap the bytes of x. You should not need to call this method directly.

SEE ALSO **NSSwapHostDoubleToBig, NSSwapHostDoubleToLittle**

NSConvertHostFloatToSwapped

SUMMARY This function performs a type conversion.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedFloat **NSConvertHostFloatToSwapped**(float *x*)

DESCRIPTION Converts the float value in *x* to a value whose bytes can be swapped. This function does not actually swap the bytes of x. You should not need to call this method directly.

SEE ALSO **NSSwapHostFloatToBig, NSSwapHostFloatToLittle**

NSConvertSwappedDoubleToHost

SUMMARY This function performs a type conversion.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS double **NSConvertSwappedDoubleToHost**(NSSwappedDouble *x*)

DESCRIPTION Converts the value in *x* to a double value. This function does not actually swap the bytes of x. You should not need to call this method directly.

SEE ALSO **NSSwapBigDoubleToHost, NSSwapLittleDoubleToHost**

NSConvertSwappedFloatToHost

SUMMARY This function performs a type conversion.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS float **NSConvertSwappedFloatToHost**(NSSwappedFloat *x*)

DESCRIPTION Converts the value in *x* to a float value. This function does not actually swap the bytes of x. You should not need to call this method directly.

SEE ALSO **NSSwapBigFloatToHost, NSSwapLittleFloatToHost**

NSCopyHashTableWithZone

SUMMARY This function performs a shallow copy of the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS NSHashTable ***NSCopyHashTableWithZone**(NSHashTable **table*, NSZone **zone*)

DESCRIPTION Returns a pointer to a new copy of *table*, created in *zone* and containing pointers to the data elements of *table*. If *zone* is NULL, the new table is created in the default zone.

The new table adopts the callback functions of table and calls the **hash** and **retain** callback functions as appropriate when inserting elements into the new table.

SEE ALSO **NSCreateHashTable, NSCreateHashTableWithZone, NSHashTableCallBacks (structure)**

NSCopyMapTableWithZone

SUMMARY This function performs a shallow copy of the specified map table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSMapTable ***NSCopyMapTableWithZone**(NSMapTable **table*, NSZone **zone*)

DESCRIPTION Returns a pointer to a new copy of *table*, created in *zone* and containing pointers to the keys and values of *table*. If *zone* is NULL, the new table is created in the default zone.

The new table adopts the callback functions of table and calls the **hash** and **retain** callback functions as appropriate when inserting elements into the new table.

SEE ALSO **NSCreateMapTable, NSCreateMapTableWithZone, NSMapTableKeyCallBacks (structure), NSMapTableValueCallBacks (structure)**

NSCopyMemoryPages

SUMMARY This function copies a block of memory.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void **NSCopyMemoryPages**(const void ***source*, void ***destination*, unsigned int *byteCount*)

DESCRIPTION Copies (or copies-on-write) *byteCount* bytes from *source* to *destination*.

SEE ALSO **NSAllocateMemoryPages, NSDeallocateMemoryPages**

NSCopyObject

SUMMARY This function creates exact copies of objects.

DECLARED IN Foundation/NSObject.h

SYNOPSIS id <NSObject> **NSCopyObject(**id <NSObject> *anObject*, unsigned int *extraBytes*, NSZone ***zone***)**

DESCRIPTION Creates and returns a new object that's an exact copy of *anObject*, created in the specified *zone* (or in the default zone, if *zone* is NULL). The *extraBytes* argument (usually zero) states the number of extra bytes required for indexed instance variables. Returns **nil** if *anObject* is **nil** or if *anObject* could not be copied.

SEE ALSO **NSAllocateObject, NSDeallocateObject**

NSCountHashTable

SUMMARY This function returns the number of elements in a hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS unsigned int **NSCountHashTable**(NSHashTable *_table_)

DESCRIPTION Returns the number of elements currently in *table*.

NSCountMapTable

SUMMARY This function returns the number of elements in a map table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS unsigned int **NSCountMapTable**(NSMapTable *_table_)

DESCRIPTION Returns the number of key/value pairs currently in *table*.

NSCParameterAssert

SUMMARY This macro evaluates the specified parameter.

DECLARED IN Foundation/NSExceptions.h

SYNOPSIS **NSCParameterAssert(**_condition_**)**

DESCRIPTION Assertions evaluate a condition and, if the condition evaluates to false, call the assertion handler for the current thread, passing it a format string and a variable number of arguments. Each thread has its own assertion handler, which is an object of class NSAssertionHandler. When invoked, an assertion handler prints an error message that includes method and class names (or the function name). It then raises an exception of type NSInternalInconsistencyException.

This macro validates a parameter for a C-function. Simply provide the parameter as the *condition* argument. The macro evaluates the parameter and, if the parameter evaluates to false, logs an error message which includes the parameter and then raises an exception.

Assertions are disabled if the preprocessor macro **NS_BLOCK_ASSERTIONS** is defined. All macros return **void**.

SEE ALSO **NSLog, NSLogv, NSAssert, NSCAssert, NSParameterAssert**

NSCreateHashTable

SUMMARY This function creates a new hash table in the default zone.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS NSHashTable ***NSCreateHashTable**(NSHashTableCallBacks *callBacks*, unsigned int *capacity*)

DESCRIPTION **NSCreateHashTable** creates and returns a pointer to an NSHashTable in the default zone.

The table's size is dependent on (but generally not equal to) *capacity*. If *capacity* is 0, a small hash table is created. The NSHashTableCallBacks structure *callBacks* has five pointers to functions (documented under "Types and Constants"), with the following defaults: pointer hashing, if **hash** is NULL; pointer equality, if **isEqual** is NULL; no call-back upon adding an element, if **retain** is NULL; no call-back upon removing an element, if **release** is NULL; and a function returning a pointer's hexadecimal value as a string, if **describe** is NULL. The hashing function must be defined such that if two data elements are equal, as defined by the comparison function, the values produced by hashing on these elements must also be equal. Also, data elements must remain invariant if the value of the hashing function depends on them; for example, if the hashing function operates directly on the characters of a string, that string can't change.

SEE ALSO **NSCopyHashTableWithZone, NSCreateHashTableWithZone**

NSCreateHashTableWithZone

SUMMARY This function creates a new hash table in the specified zone.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS NSHashTable ***NSCreateHashTableWithZone**(NSHashTableCallBacks *callBacks*, unsigned int *capacity*, NSZone **zone*)

DESCRIPTION This function creates a new hash table in the specified *zone*. If *zone* is NULL, the hash table is created in the default zone.

The table's size is dependent on (but generally not equal to) *capacity*. If *capacity* is 0, a small hash table is created. The NSHashTableCallBacks structure *callBacks* has five pointers to functions (documented under "Types and Constants"), with the following defaults: pointer hashing, if **hash** is NULL; pointer equality, if **isEqual** is NULL; no call-back upon adding an element, if **retain** is NULL; no call-back upon removing an element, if **release** is NULL; and a function returning a pointer's hexadecimal value as a string, if **describe** is NULL. The hashing function must be defined such that if two data elements are equal, as defined by the comparison function, the values produced by hashing on these elements must also be equal. Also, data elements must remain invariant if the value of the hashing function depends on them; for example, if the hashing function operates directly on the characters of a string, that string can't change.

SEE ALSO **NSCreateHashTable**

NSCreateMapTable

SUMMARY This function creates a new map table in the default zone.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSMapTable ***NSCreateMapTable**(NSMapTableKeyCallBacks *keyCallBacks*, NSMapTableValueCallBacks *valueCallBacks*, unsigned int *capacity*)

DESCRIPTION **NSCreateMapTable** creates, and returns a pointer to, an NSMapTable in the default zone; the table's size is dependent on (but generally not equal to) *capacity*. If *capacity* is 0, a small map table is created. The NSMapTableKeyCallBacks arguments are structures (documented under "Types

and Constants") that are very similar to the call-back structure used by **NSCreateHashTable**; in fact, they have the same defaults as documented for that function.

SEE ALSO **NSCopyMapTableWithZone, NSCreateMapTableWithZone**

NSCreateMapTableWithZone

SUMMARY This function creates a new map table in the specified zone.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSMapTable *__NSCreateMapTableWithZone__(NSMapTableKeyCallBacks *keyCallBacks*, NSMapTableValueCallBacks *valueCallBacks*, unsigned int *capacity*, NSZone **zone*)

DESCRIPTION This function creates a new map table in the specified *zone*. If *zone* is NULL, the hash table is created in the default zone.

The table's size is dependent on (but generally not equal to) *capacity*. If *capacity* is 0, a small map table is created. The NSMapTableKeyCallBacks arguments are structures (documented under "Types and Constants") that are very similar to the call-back structure used by **NSCreateHashTable**; in fact, they have the same defaults as documented for that function.

SEE ALSO **NSCopyMapTableWithZone, NSCreateMapTable**

NSCreateZone

SUMMARY Creates a new zone

DECLARED IN Foundation/NSZone.h

SYNOPSIS NSZone ***NSCreateZone(**unsigned int *startSize*, unsigned int *granularity*, BOOL *canFree***)**

DESCRIPTION Creates and returns a pointer to a new zone of *startSize* bytes, which will grow and shrink by *granularity* bytes. If *canFree* is zero, the allocator will never free memory, and **malloc** will be fast.

SEE ALSO **NSDefaultMallocZone, NSRecycleZone, NSSetZoneName**

NSDeallocateMemoryPages

SUMMARY This function deallocates the specified block of memory.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void **NSDeallocateMemoryPages**(void ***pointer*, unsigned int *byteCount*)

DESCRIPTION This function deallocates memory that was allocated with **NSAllocateMemoryPages**.

SEE ALSO **NSCopyMemoryPages, NSAllocateMemoryPages**

NSDeallocateObject

SUMMARY This function destroys an existing object.

DECLARED IN Foundation/NSObject.h

SYNOPSIS void **NSDeallocateObject(**id <NSObject> *anObject***)**

DESCRIPTION This function deallocates *anObject*, which must have been allocated using NSAllocateObject.

SEE ALSO **NSCopyObject, NSAllocateObject**

NSDecimalAdd

SUMMARY This function adds two decimal values.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError **NSDecimalAdd**(NSDecimal *_result_, const NSDecimal *_leftOperand_, const NSDecimal *_rightOperand_, NSRoundingMode _roundingMode_)

DESCRIPTION Adds _leftOperand_ to _rightOperand_, and stores the sum in _result_.

An NSDecimal can represent a number with up to 38 significant digits. If a number is more precise than that, it must be rounded off. _roundingMode_ determines how to round it off. There are four possible rounding modes:

- NSRoundDown. The number rounds down.
- NSRoundUp. The number rounds up.
- NSRoundPlain. The number rounds to the closest 38-digit approximation. If the number is halfway between two positive numbers, it round up; if it's halfway between two negative numbers, it rounds down.
- NSRoundBankers. The number rounds to the closest 38-digit approximation. If it is caught halfway between two possibilities, it rounds to the one whose last digit is even. In practice, this means that, over the long run, numbers will be rounded up as often as they are rounded down; there will be no systematic bias.

The return value indicates whether any machine limitations were encountered in the addition. If none were encountered, the function returns NSCalculationNoError. Otherwise it may return one of the following values: NSCalculationLossOfPrecision, NSCalculationOverflow or NSCalculationUnderflow. For descriptions of all these error conditions, see **exceptionDuringOperation:error:leftOperand:rightOperand:** in the protocol specification for NSDecimalNumberBehaviors.

NSDecimalCompact

SUMMARY This function compacts the decimal structure for efficiency.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS void **NSDecimalCompact**(NSDecimal *_number_)

DESCRIPTION Formats *number* so that calculations using it will take up as little memory as possible. All the **NSDecimal...** arithmetic functions expect compact NSDecimal arguments.

NSDecimalCompare

SUMMARY This function compares two decimal values.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSComparisonResult **NSDecimalCompare**(const NSDecimal *_leftOperand_, const NSDecimal *_rightOperand_)

DESCRIPTION Compares *leftOperand* to *rightOperand*, with three possible return values.

- If *leftOperand* is bigger, the function returns NSOrderedDescending.
- If *rightOperand* is bigger, the the function returns NSOrdered Ascending.
- If the two operands are equal, the function returns NSOrderedSame.

NSDecimalCopy

SUMMARY This function copies the value of a decimal number.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS void **NSDecimalCopy**(NSDecimal *_destination_, const NSDecimal *_source_)

DESCRIPTION Copies the value in *source* to *destination*.

NSDecimalDivide

SUMMARY This function divides one decimal value by another.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError **NSDecimalDivide**(NSDecimal **result*, const NSDecimal **leftOperand*, const NSDecimal **rightOperand*, NSRoundingMode *roundingMode*)

DESCRIPTION Divides *leftOperand* by *rightOperand*, and stores the quotient, possibly rounded off according to *roundingMode*, in *result*. If *rightOperand* is 0, returns NSDivideByZero.

For explanations of the other possible return values, and of all the possible *roundingMode*'s, see **NSDecimalAdd**, above.

Note that this function can't precisely represent a non-decimal fraction like 1/3.

NSDecimalIsNotANumber

SUMMARY This function determines if the specified decimal contains a valid number.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS BOOL **NSDecimalIsNotANumber**(const NSDecimal **decimal*)

DESCRIPTION Returns YES if the value in *decimal* represents a valid number, otherwise returns NO.

NSDecimalMultiply

SUMMARY This function multiplies two decimal numbers together.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError **NSDecimalMultiply**(NSDecimal ***result*, const NSDecimal ***leftOperand*, const NSDecimal ***rightOperand*, NSRoundingMode *roundingMode*)

DESCRIPTION Multiplies *rightOperand* by *leftOperand*, and stores the product, possibly rounded off according to *roundingMode*, in *result*.

For explanations of the possible return values and *roundingMode*'s, see **NSDecimalAdd**, above.

NSDecimalMultiplyByPowerOf10

SUMMARY This function multiplies a decimal by the specified power of 10.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError NSDecimalMultiplyByPowerOf10(NSDecimal ***result*, const NSDecimal ***number*, short *power*, NSRoundingMode *roundingMode*)

DESCRIPTION Multiplies *number* by 10*power*, and stores the product, possibly rounded off according to *roundingMode*, in *result*.

For explanations of the possible return values and *roundingMode*'s, see **NSDecimalAdd**, above.

NSDecimalNormalize

SUMMARY This function normalizes the internal format of two decimal numbers to simplify later operations.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError **NSDecimalNormalize**(NSDecimal ***number1*, NSDecimal ***number2*, NSRoundingMode *roundingMode*)

DESCRIPTION An NSDecimal is represented in memory as a mantissa and an exponent, expressing the value mantissa **x** 10exponent. A number can have many NSDecimal representations; for example, the following table lists several valid NSDecimal representations for the number 100:

Mantissa	Exponent
100	0
10	1
1	2

NSDecimalNormalize formats number1 and number2 so that they have equal exponents. This format makes addition and subtraction very convenient. Both **NSDecimalAdd** and **NSDecimalSubtract** call **NSDecimalNormalize**. You may want to use it if you write more complicated addition or subtraction routines.

For explanations of the function's possible return values, see **NSDecimalAdd**, above.

NSDecimalPower

SUMMARY This function raises the decimal value to the specified power.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError **NSDecimalPower**(NSDecimal ***result*, const NSDecimal ***number*, unsigned int *power*, NSRoundingMode *roundingMode*)

DESCRIPTION Raises *number* to *power*, possibly rounding off according to *roundingMode*, and stores the resulting value in *result*.

For explanations of the possible return values and *roundingMode*'s, see **NSDecimalAdd**, above.

NSDecimalRound

SUMMARY This function rounds off the decimal value.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS void **NSDecimalRound**(NSDecimal **result*, const NSDecimal **number*, int *scale*, NSRoundingMode *roundingMode*)

DESCRIPTION Rounds *number* off according to the parameters *scale* and rounding *mode*, and stores the result in *result*.

scale specifies the number of digits result can have after its decimal point. *roundingMode* specifies the way that number is rounded off. There are four possible values for *roundingMode*: NSRoundDown, NSRoundUp, NSRoundPlain, and NSRoundBankers. For thorough discussions of *scale* and *roundingMode*, see the **scale** and **roundingMode** in the protocol specification for NSDecimalNumberBehaviors.

NSDecimalString

SUMMARY This function returns a string representation of the decimal value.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSString ***NSDecimalString**(const NSDecimal **decimal*, NSDictionary **locale*)

DESCRIPTION Returns a string representation of *decimal*. *locale* determines the format of the decimal separator.

NSDecimalSubtract

SUMMARY This function subtracts one decimal value from another.

DECLARED IN Foundation/NSDecimal.h

SYNOPSIS NSCalculationError **NSDecimalSubtract**(NSDecimal ***result*, const NSDecimal ***leftOperand*, const NSDecimal ***rightOperand*, NSRoundingMode *roundingMode*)

DESCRIPTION Subtracts *rightOperand* from *leftOperand*, and stores the difference, possibly rounded off according to *roundingMode*, in *result*.

For explanations of the possible return values and *roundingMode*'s, see **NSDecimalAdd**, above.

NSDecrementExtraRefCountWasZero

SUMMARY This function decrements the specified object's reference count.

DECLARED IN Foundation/NSObject.h

SYNOPSIS BOOL **NSDecrementExtraRefCountWasZero(**id *anObject***)**

DESCRIPTION This function decrements the "extra reference" count of an object. Newly created objects have only one actual reference, so that a single release message results in the object being deallocated. Extra references are those beyond the single original reference, and are usually created by sending the object a retain message. Your code should generally not use these functions unless it is overriding the retain or release methods.

This function returns NO if *anObject* had an extra reference count. If *anObject* didn't have an extra referenct count, it returns YES, indicating that the object should be deallocated (with **dealloc**).

SEE ALSO **NSExtraRefCount, NSIncrementExtraRefCount**

NSDefaultMallocZone

SUMMARY Returns the default zone

DECLARED IN Foundation/NSZone.h

SYNOPSIS NSZone ***NSDefaultMallocZone(**void**)**

DESCRIPTION Returns the default zone, which is created automatically at startup. This is the zone used by the standard C function **malloc**.

SEE ALSO **NSCreateZone**

NSDivideRect

SUMMARY This function divides a rectangle into two new rectangles.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS void **NSDivideRect**(NSRect *inRect*, NSRect **slice*, NSRect **remainder*, float *amount*, NSRectEdge *edge*)

DESCRIPTION Creates two rectangles, *slice* and *remainder,* from *inRect*, by dividing *inRect* with a line that's parallel to one of *inRect*'s sides (namely, the side specified by edge—either NSMinXEdge, NSMinYEdge, NSMaxXEdge, or NSMaxYEdge). The size of *slice* is determined by *amount*, which measures the distance from *edge*.

SEE ALSO **NSInsetRect, NSIntegralRect, NSOffsetRect**

NSEnumerateHashTable

SUMMARY This function creates an enumerator for the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS NSHashEnumerator **NSEnumerateHashTable**(NSHashTable *_table_)

DESCRIPTION Returns an **NSHashEnumerator** structure that will cause successive elements of *table* to be returned each time this enumerator is passed to **NSNextHashEnumeratorItem**.

SEE ALSO **NSNextHashEnumeratorItem**

NSEnumerateMapTable

SUMMARY This function creates an enumerator for the specified map table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSMapEnumerator **NSEnumerateMapTable**(NSMapTable *_table_)

DESCRIPTION **NSEnumerateMapTable** returns an **NSMapEnumerator** structure that will cause successive key/value pairs of *table* to be visited each time this enumerator is passed to **NSNextMapEnumeratorPair**.

SEE ALSO **NSNextMapEnumeratorPair, NSMapMember, NSMapGet, NSAllMapTableKeys, NSAllMapTableValues**

NSEqualPoints

SUMMARY This function tests the two points for equality.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSEqualPoints**(NSPoint *aPoint*, NSPoint *bPoint*)

DESCRIPTION Returns YES if the two points *aPoint* and *bPoint* are identical, and NO otherwise.

NSEqualRanges

SUMMARY This function tests the two range values for equality.

DECLARED IN Foundation/NSRange.h

SYNOPSIS BOOL **NSEqualRanges**(NSRange *range1*, NSRange *range2*)

DESCRIPTION Returns YES if *range1* and *range2* have the same locations and lengths.

NSEqualRects

SUMMARY This function tests the two rectangles for equality.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSEqualRects**(NSRect *aRect*, NSRect *bRect*)

DESCRIPTION Returns YES if the two rectangles *aRect* and *bRect* are identical, and NOotherwise.

NSEqualSizes

SUMMARY This function tests the two size values for equality.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSEqualSizes**(NSSize *aSize*, NSSize *bSize*)

DESCRIPTION Returns YES if the two sizes *aSize* and *bSize* are identical, and NO otherwise.

NSExtraRefCount

SUMMARY This function returns the specified object's reference count.

DECLARED IN Foundation/NSObject.h

SYNOPSIS unsigned int **NSExtraRefCount**(id *object*)

DESCRIPTION Returns the current reference count of *object*. This function is used in conjunction with **NSIncrementExtraRefCount** and **NSDecrementExtraRefCountWasZero** in situations where you need to override an object's **retain** and **release** methods.

NSFreeHashTable

SUMMARY This function deletes the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void **NSFreeHashTable**(NSHashTable **table*)

DESCRIPTION **NSFreeHashTable** releases each element of the specified hash table and frees the table itself.

SEE ALSO **NSResetHashTable**

NSFreeMapTable

SUMMARY This function deletes the specified map table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void **NSFreeMapTable**(NSMapTable *_table_)

DESCRIPTION **NSFreeMapTable** releases each key and value of the specified map table and frees the table itself.

SEE ALSO **NSResetMapTable**

NSFullUserName

SUMMARY This function returns the full name of the current user.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSString ***NSFullUserName**(void)

DESCRIPTION Returns a string containing the full name of the current user.

SEE ALSO **NSUserName**

NSGetSizeAndAlignment

SUMMARY This function returns the type and size of the specified data type.

DECLARED IN Foundation/NSObjCRuntime.h

SYNOPSIS const char ***NSGetSizeAndAlignment**(const char **typePtr*,
unsigned int **sizep*,
unsigned int **alignp*)

DESCRIPTION Gets the next type code from *typePtr* and returns the size and alignment of that data type in *sizep* and *alignp*, respectively. You can specify 0 for either *sizep* or *alignp* to ignore the corresponding information. This function returns a new pointer to the string specified by *typePtr*; the new pointer points to the character just past the type code that was read.

NSGetUncaughtExceptionHandler

SUMMARY This function returns the top-level error handler.

DECLARED IN Foundation/NSException.h

SYNOPSIS NSUncaughtExceptionHandler ***NSGetUncaughtExceptionHandler(**void**)**

DESCRIPTION **NSGetUncaughtExceptionHandler** returns a pointer to the function serving as the top-level error handler. This handler will process exceptions raised outside of any exception-handling domain.

SEE ALSO **NSSetUncaughtExceptionHandler**

NSHashGet

SUMMARY This function returns an element of the hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void ***NSHashGet**(NSHashTable **table*, const void **pointer*)

DESCRIPTION Returns the pointer in the table that matches *pointer* (as defined by the **isEqual** call-back function). If there is no matching element, the function returns NULL

NSHashInsert

SUMMARY This function adds an element to the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void **NSHashInsert**(NSHashTable *_table_, const void *_pointer_)

DESCRIPTION **NSHashInsert** inserts *pointer*, which must not be NULL, into *table*. If *pointer* matches an item already in the table, the previous pointer is released using the **release** call-back function that was specified when the table was created.

SEE ALSO **NSHashRemove, NSHashInsertKnownAbsent, NSHashInsertIfAbsent**

NSHashInsertIfAbsent

SUMMARY This function adds an element to the specified hash table only if the table does not already contain the element.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void ***NSHashInsertIfAbsent**(NSHashTable *_table_, const void *_pointer_)

DESCRIPTION If *pointer* matches an item already in *table*, **NSHashInsertIfAbsent** returns the pre-existing pointer; otherwise, it adds *pointer* to the table and returns NULL. You must not specify NULL for *pointer*.

SEE ALSO **NSHashRemove, NSHashInsert, NSHashInsertKnownAbsent**

NSHashInsertKnownAbsent

SUMMARY This function adds an element to the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void **NSHashInsertKnownAbsent**(NSHashTable *_table_, const void *_pointer_)

DESCRIPTION **NSHashInsertKnownAbsent** inserts *pointer*, which must not be NULL, into *table*. Unike **NSHashInsert**, this function raises NSInvalidArgumentException if *table* already includes an element that matches *pointer.*

SEE ALSO **NSHashRemove, NSHashInsert, NSHashInsertIfAbsent**

NSHashRemove

SUMMARY This function removes an element from the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void **NSHashRemove**(NSHashTable *_table_, const void *_pointer_)

DESCRIPTION If *pointer* matches an item already in *table*, this function releases the pre-existing item.

SEE ALSO **NSHashInsert, NSHashInsertKnownAbsent, NSHashInsertIfAbsent**

NSHeight

SUMMARY This function returns the height of the specified rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSHeight**(NSRect *aRect*)

DESCRIPTION Returns the height of *aRect*.

SEE ALSO **NSMaxX, NSMaxY, NSMidX, NSMidY, NSMinX, NSMinY, NSWidth**

NSHomeDirectory

SUMMARY This function getsinformation about a user.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSString ***NSHomeDirectory(**void**)**

DESCRIPTION **NSHomeDirectory** returns a path to the current user's home directory.

SEE ALSO **NSFullUserName, NSUserName, NSHomeDirectoryForUser**

NSHomeDirectoryForUser

SUMMARY Get information about a user

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSString ***NSHomeDirectoryForUser(**NSString **userName***)**

DESCRIPTION **NSHomeDirectoryForUser** returns a path to the home directory for the user specified by *userName*.

SEE ALSO **NSFullUserName, NSUserName, NSHomeDirectory**

NSHostByteOrder

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned int **NSHostByteOrder**(void)

DESCRIPTION Returns the endian format, either NSLittleEndian or NSBigEndian, supported by the current platform.

NSIncrementExtraRefCount

SUMMARY This function increments the specified object's reference count.

DECLARED IN Foundation/NSObject.h

SYNOPSIS void **NSIncrementExtraRefCount(**id *anObject***)**

DESCRIPTION This function increments the "extra reference" count of an object. Newly created objects have only one actual reference, so that a single release message results in the object being deallocated. Extra references are those beyond the single original reference, and are usually created by sending the object a retain message. Your code should generally not use these functions unless it is overriding the retain or release methods.

SEE ALSO **NSExtraRefCount, NSDecrementExtraRefCountWasZero**

NSInsetRect

SUMMARY This function insets the rectangle by the specified amount.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSInsetRect(**NSRect *aRect*, float *dX*, float *dY*)

DESCRIPTION **NSInsetRect** returns a copy of the rectangle *aRect*, altered by moving the two sides that are parallel to the y-axis inwards by *dX*, and the two sides parallel to the x-axis inwards by *dY*.

SEE ALSO **NSDivideRect, NSIntegralRect, NSOffsetRect**

NSIntegralRect

SUMMARY This function adjusts the sides of the rectangle to integer values.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSIntegralRect**(NSRect *aRect*)

DESCRIPTION **NSIntegralRect** returns a copy of the rectangle *aRect*, expanded outwards just enough to ensure that none of its four defining values (*x*, *y*, *width*, and *height*) have fractional parts. If *aRect*'s width or height is zero or negative, this function returns a rectangle with origin at (0.0, 0.0) and with zero width and height.

SEE ALSO **NSDivideRect, NSInsetRect, NSOffsetRect**

NSIntersectionRange

SUMMARY This function returns the intersection of the specified ranges.

DECLARED IN Foundation/NSRange.h

SYNOPSIS NSRange **NSIntersectionRange(**NSRange *range1*, NSRange *range2***)**

DESCRIPTION **NSIntersectionRange** returns a range describing the intersection of *range1* and *range2*—that is, a range containing the indices that exist in both ranges. If the returned range's **length** field is zero, then the two ranges don't intersect, and the value of the **location** field is undefined.

SEE ALSO **NSUnionRange**

NSIntersectionRect

SUMMARY This function calculates the intersection of two rectangles.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSIntersectionRect**(NSRect *aRect*, NSRect *bRect*)

DESCRIPTION **NSIntersectionRect** returns the graphic intersection of *aRect* and *bRect*. If the two rectangles don't overlap, the returned rectangle has its origin at (0.0, 0.0) and zero width and height. (This includes situations where the intersection is a point or a line segment.)

SEE ALSO **NSUnionRect**

NSIntersectsRect

SUMMARY This function tests whether two rectangles intersect.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSIntersectsRect**(NSRect *aRect*, NSRect *bRect*)

DESCRIPTION Returns YES if *aRect* intersects *bRect*, otherwise returns NO. This function returns NO if either *aRect* and *bRect* has a width or height that is 0.

SEE ALSO **NSIntersectionRect**

NSIsEmptyRect

SUMMARY This function tests whether the specified rectangle is empty.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSIsEmptyRect**(NSRect *aRect*)

DESCRIPTION Returns YES if the rectangle encloses no area at all—that is, if its width or height is zero or negative.

NSJavaBundleCleanup

SYNOPSIS void **NSJavaBundleCleanup**(NSBundle **bundle*, NSDictionary **plist*)

DESCRIPTION <<Description Forthcoming>>

NSJavaBundleSetup

SYNOPSIS id **NSJavaBundleSetup**(NSBundle **bundle*, NSDictionary **plist*)

DESCRIPTION <<Description Forthcoming>>

NSJavaNeedsToLoadClasses

SYNOPSIS BOOL **NSJavaNeedsToLoadClasses**(NSDictionary **plist*)

DESCRIPTION <<Description Forthcoming>>

NSJavaNeedsVirtualMachine

SYNOPSIS BOOL **NSJavaNeedsVirtualMachine**(NSDictionary *_plist_)

DESCRIPTION <<Description Forthcoming>>

NSJavaProvidesClasses

SYNOPSIS BOOL NSJavaProvidesClasses(NSDictionary *plist)

DESCRIPTION <<Description Forthcoming>>

NSJavaSetup

SYNOPSIS id NSJavaSetup(NSDictionary *plist)

DESCRIPTION <<Description Forthcoming>>

NSJavaVirtualMachineSetup

SYNOPSIS id **NSJavaVirtualMachineSetup**(BOOL *_vmNeeded_)

DESCRIPTION <<Description Forthcoming>>

NSLocationInRange

SUMMARY This function verifies that the specified position is in the given range.

DECLARED IN Foundation/NSRange.h

SYNOPSIS BOOL **NSLocationInRange(**unsigned int *index*, NSRange *aRange***)**

DESCRIPTION **NSLocationInRange** returns YES if the given *index* lies within *aRange*—that is, if it's greater than or equal to *aRange*.location and less than *aRange*.location plus *aRange*.length.

NSLog

SUMMARY This function logs error messages to **stderr**.

DECLARED IN Foundation/NSUtilities.h

SYNOPSIS void **NSLog(**NSString **format*, ...**)**

DESCRIPTION NSLog simply calls NSLogv, passing it a variable number of arguments.

SEE ALSO **NSLogv**

NSLogPageSize

SUMMARY This function returns the binary log of the page size.

DECLARED IN Foundation/NSZone.h

SYNOPSIS unsigned int **NSLogPageSize**(void)

DESCRIPTION **NSLogPageSize** returns the binary log of the current page size.

SEE ALSO **NSRoundDownToMultipleOfPageSize, NSRoundUpToMultipleOfPageSize, NSPageSize**

NSLogv

SUMMARY This function logs error messages to **stderr**.

DECLARED IN Foundation/NSUtilities.h

SYNOPSIS void **NSLogv(**NSString ***format*, va_list *args***)**

DESCRIPTION **NSLogv** logs an error message. The message consists of a timestamp and the process ID prefixed to the string you pass in. You compose this string with a format string , *format*, and one or more arguments to be inserted into the string. The format specification allowed by these functions is that which is understood by NSString's formatting capabilities (which is not necessarily the set of format escapes and flags understood by **printf**).

In general, you should use the **NSLog** function instead of calling this function directly. If you do use this function directly, you must have prepared the the variable argument list in the *args* parameter by calling the standard C macro va_start. Upon completion, you must similarly call the standard C macro va_end for this list.

On HP-UX, Solaris, and Mach, NSLogv writes the log to STDERR_FILENO if the file descriptor is open. If that fails, the message is sent to the syslog subsystem, if it exists on a platform, with the LOG_USER facility (or default facility if LOG_USER doesn't exist on a platform), with priority LOG_ERR (or similar, depending on what the platform supports). If both of these attempts to write the message fail, the message is discarded.

On Windows platforms, the message is written to the STD_ERROR_HANDLE, if that handle is valid, on Windows platforms that support that standard handle. It is also written to the Windows Event Log on Windows platforms that support that, or to a file **c:\fndation.log** on Windows platforms that do not, if that file can be opened. If all of these attempts fail, the message is discarded. On some Windows platforms, the message to the Event Log may be truncated if there is a limit to the size of a message that the Event Log can accept. On Windows platforms that support an application discovering whether or not it's running under a debugger, NSLogv may only send the message to the debugger for its handling, via standard WIN32 mechanisms, and not also write the message to STD_ERROR_HANDLE and the Event Log. Note that a debugger may

choose to not display message thus sent to it, or may choose not to display all of the message—that has nothing to do with **NSLogv**.

Output from **NSLogv** is serialized, in that only one thread in a process can be doing the writing/logging described above at a time. All attempts at writing/logging a message complete before the next thread can begin its attempts.

The effects of **NSLogv** are not serialized with subsystems other than those discussed above (such as the standard I/O package) and do not produce side effects on those subsystems (such as causing buffered output to be flushed, which may be undesirable).

SEE ALSO **NSLog**

NSMakePoint

SUMMARY This function creates a new NSPoint from the specified values.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSPoint **NSMakePoint**(float *x*, float *y*)

DESCRIPTION Creates an NSPoint having the coordinates *x* and *y*.

NSMakeRange

SUMMARY This function creates a new NSRange from the specified values.

DECLARED IN Foundation/NSRange.h

SYNOPSIS NSRange **NSMakeRange**(unsigned int *location*, unsigned int *length*)

DESCRIPTION Creates an NSRange having the specified location and length.

NSMakeRect

SUMMARY This function creates a new NSRect from the specified values.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSMakeRect**(float *x*, float *y*, float *w*, float *h*)

DESCRIPTION Creates an NSRect having the specified origin and size.

NSMakeSize

SUMMARY This function creates a new NSSize from the specified values.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSSize **NSMakeSize**(float *w*, float *h*)

DESCRIPTION Creates an NSSize having the specified width and height.

NSMapGet

SUMMARY This function returns a map-table value for the specified key.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void ***NSMapGet**(NSMapTable **table*, const void **key*)

DESCRIPTION Returns the value that table maps to key, or NULL if the table doesn't contain key.

SEE ALSO **NSMapMember, NSEnumerateMapTable, NSNextMapEnumeratorPair, NSAllMapTableKeys, NSAllMapTableValues**

NSMapInsert

SUMMARY This function inserts a key/value pair into the specified table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void **NSMapInsert**(NSMapTable *_table_, const void *_key_, const void *_value_)

DESCRIPTION **NSMapInsert** inserts _key_ and _value_ into table. If _key_ matches a key already in the table, _value_ is retained and the previous value is released, using the retain and release call-back functions that were specified when the table was created. Raises InvalidArgumentException if _key_ is equal to the **notAKeyMarker** field of the table's NSMapTableKeyCallBacks structure.

SEE ALSO **NSMapRemove, NSMapInsertIfAbsent, NSMapInsertKnownAbsent**

NSMapInsertIfAbsent

SUMMARY This function inserts a key/value pair into the specified table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void ***NSMapInsertIfAbsent**(NSMapTable *_table_, const void *_key_, const void *_value_)

DESCRIPTION If _key_ matches a key already in table, **NSMapInsertIfAbsent** returns the pre-existing key; otherwise, it adds _key_ and _value_ to the table and returns NULL. Raises NSInvalidArgumentException if _key_ is equal to the **notAKeyMarker** field of the table's NSMapTableKeyCallBacks structure.

SEE ALSO **NSMapRemove, NSMapInsert, NSMapInsertKnownAbsent**

NSMapInsertKnownAbsent

SUMMARY This function inserts a key/value pair into the specified table if the pair had not been previously added.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void **NSMapInsertKnownAbsent**(NSMapTable *_table_, const void *_key_, const void *_value_)

DESCRIPTION **NSMapInsertKnownAbsent** inserts *key* (which must not be **notAKeyMarker**) and *value* into table. Unlike **NSMapInsert**, this function raises NSInvalidArgumentException if *table* already includes a key that matches *key*.

SEE ALSO **NSMapRemove, NSMapInsert, NSMapInsertIfAbsent**

NSMapMember

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS BOOL **NSMapMember**(NSMapTable *_table_, const void *_key_, void **_originalKey_, void **_value_)

DESCRIPTION Returns YES if table contains a key equal to *key*. If so, *originalKey* is set to *key*, and *value* is set to the value that the table maps to *key*.

SEE ALSO **NSMapGet, NSEnumerateMapTable, NSNextMapEnumeratorPair, NSAllMapTableKeys, NSAllMapTableValues**

NSMapRemove

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void **NSMapRemove**(NSMapTable *_table_, const void *_key_)

DESCRIPTION If *key* matches a key already in *table*, this function release the pre-existing key and its corresponding value.

SEE ALSO **NSMapInsert, NSMapInsertIfAbsent, NSMapInsertKnownAbsent**

NSMaxRange

DECLARED IN Foundation/NSRange.h

SYNOPSIS unsigned int **NSMaxRange**(NSRange *range*)

DESCRIPTION Returns *range*.location + *range*.length—in other words, the number one greater than the maximum value within the range.

NSMaxX

SUMMARY This function returns the largest x-coordinate of a rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSMaxX**(NSRect *aRect*)

DESCRIPTION **NSMaxX** returns the largest x-coordinate value within *aRect*.

SEE ALSO **NSWidth, NSHeight, NSMaxY**

NSMaxY

SUMMARY This function returns the largest y-coordinate of a rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSMaxY**(NSRect *aRect*)

DESCRIPTION **NSMaxY** returns the largest y-coordinate value within *aRect*.

SEE ALSO **NSWidth, NSHeight, NSMaxX**

NSMidX

SUMMARY This function returns the x-coordinate of a rectangle’s midpoint.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSMidX**(NSRect *aRect*)

DESCRIPTION **NSMidX** returns the x-coordinate of the center of the rectangle.

SEE ALSO **NSWidth, NSHeight, NSMidY**

NSMidY

SUMMARY This function returns the y-coordinate of a rectangle’s midpoint.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSMidY**(NSRect *aRect*)

DESCRIPTION **NSMidY** returns the y-coordinate of the rectangle's center point.

SEE ALSO **NSWidth, NSHeight, NSMidX**

NSMinX

SUMMARY This function returns the smallest x-coordinate of a rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSMinX**(NSRect *aRect*)

DESCRIPTION **NSMinX** returns the smallest x-coordinate value within *aRect*.

SEE ALSO **NSWidth, NSHeight, NSMinY**

NSMinY

SUMMARY This function returns the smallest y-coordinate of a rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSMinY**(NSRect *aRect*)

DESCRIPTION **NSMinY** returns the smallest y-coordinate value within *aRect* .

SEE ALSO **NSWidth, NSHeight, NSMinX**

NSMouseInRect

SUMMARY This function tests whether the point is in the specified rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSMouseInRect**(NSPoint *aPoint*, NSRect *aRect*, BOOL *flipped*)

DESCRIPTION Returns YES if the point represented by *aPoint* is located within the rectangle represented by *aRect*. It assumes an unscaled and unrotated coordinate system; the argument *flipped* should be YES if the coordinate system has been flipped so that the positive y-axis extends downward. This function is used to determine whether the hot spot of the cursor lies inside a given rectangle.

NSNextHashEnumeratorItem

SUMMARY This function returns the next hash-table element in the enumeration.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void ***NSNextHashEnumeratorItem**(NSHashEnumerator **enumerator*)

DESCRIPTION **NSNextHashEnumeratorItem** returns the next element in the table that enumerator is associated with, or NULL if *enumerator* has already iterated over all the elements.

SEE ALSO **NSEnumerateHashTable**

NSNextMapEnumeratorPair

SUMMARY This function returns the next map-table pair in the enumeration

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS BOOL **NSNextMapEnumeratorPair**(NSMapEnumerator *_enumerator_, void **_key_, void **_value_)

DESCRIPTION **NSNextMapEnumeratorPair** returns NO if *enumerator* has already iterated over all the elements in the table that *enumerator* is associated with. Otherwise, this function sets *key* and *value* to match the next key/value pair in the table, and returns YES.

SEE ALSO **NSEnumerateMapTable, NSMapMember, NSMapGet, NSAllMapTableKeys, NSAllMapTableValues**

NSOffsetRect

SUMMARY This function insets the rectangle by the specified amount.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSOffsetRect**(NSRect *aRect*, float *dX*, float *dY*)

DESCRIPTION **NSOffsetRect** returns a copy of the rectangle *aRect*, with its location shifted by *dX* along the x-axis and by *dY* along the y-axis.

SEE ALSO **NSDivideRect, NSInsetRect, NSIntegralRect**

NSOpenStepRootDirectory

SUMMARY This function returns the root directory of the user's system.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSString ***NSOpenStepRootDirectory**(void)

DESCRIPTION Returns a string identifying the root directory of the user's system.

SEE ALSO **NSHomeDirectory, NSHomeDirectoryForUser**

NSPageSize

SUMMARY This function returns the number of bytes in a page.

DECLARED IN Foundation/NSZone.h

SYNOPSIS unsigned int **NSPageSize**(void)

DESCRIPTION **NSPageSize** returns the number of bytes in a page.

SEE ALSO **NSRoundDownToMultipleOfPageSize, NSRoundUpToMultipleOfPageSize, NSLogPageSize**

NSParameterAssert

SUMMARY This macro validates the specified parameter.

DECLARED IN Foundation/NSExceptions.h

SYNOPSIS **NSParameterAssert(***condition***)**

DESCRIPTION Assertions evaluate a condition and, if the condition evaluates to false, call the assertion handler for the current thread, passing it a format string and a variable number of arguments. Each thread has its own assertion handler, which is an object of class NSAssertionHandler. When invoked, an assertion handler prints an error message that includes method and class names (or the function name). It then raises an exception of type NSInternalInconsistencyException.

This macro validates a parameter for an Objective-C method. Simply provide the parameter as the *condition* argument. The macro evaluates the parameter and, if it is false, it logs an error message which includes the parameter and then raises an exception.

Assertions are disabled if the preprocessor macro **NS_BLOCK_ASSERTIONS** is defined.All assertion macros return **void**.

SEE ALSO **NSLog, NSLogv, NSAssert, NSCAssert, NSCParameterAssert**

NSPointFromString

SUMMARY This function returns a point from a text-based representation.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSPoint **NSPointFromString**(NSString *_aString_)

DESCRIPTION Scans *aString* for text of the form "{x=*a*; y=*b*}" and returns the values for *a* and *b* in a new NSPoint object. If *aString* does not contain this text, this function returns an NSPoint object whose x- and y-coordinates are both 0.

SEE ALSO **NSStringFromPoint**

NSPointInRect

SUMMARY This function tests whether the specified point is in the rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS BOOL **NSPointInRect**(NSPoint *aPoint*, NSRect *aRect*)

DESCRIPTION Performs the same test as **NSMouseInRect**, but assumes a flipped coordinate system.

NSRangeFromString

SUMMARY This function returns a range from a text-based representation.

DECLARED IN Foundation/NSRange.h

SYNOPSIS NSRange **NSRangeFromString**(NSString **aString*)

DESCRIPTION Returns *aString* for text of the form: "{location = *a*; length = *b*}" and returns the values for *a* and *b* in a new NSRange object. If *aString* does not contain this text, this function returns an NSRange object whose location and length values are both 0.

SEE ALSO **NSStringFromRange**

NSRealMemoryAvailable

SUMMARY This function returns information about the user's system.

DECLARED IN Foundation/NSZone.h

SYNOPSIS unsigned int **NSRealMemoryAvailable**(void)

DESCRIPTION **NSRealMemoryAvailable** returns the number of bytes available in RAM.

NSRectFromString

SUMMARY This function returns a rectangle from a text-based representation.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSRectFromString**(NSString *_aString_)

DESCRIPTION Scans *aString* for text of the form "{x=*a*; y=*b*; width=*c*; height=*d*}", and returns the values for *a*, *b*, *c*, and *d* in a new NSRect object. If *aString* does not contain the specified text, this function returns an NSRect object with a rectangle whose origin is (0, 0) and width and height are both 0.

SEE ALSO **NSStringFromRect**

NSRecycleZone

SUMMARY This function frees memory in a zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void **NSRecycleZone(**NSZone *_zone_**)**
void **NSZoneFree**(NSZone *_zone_, void *_pointer_**)**

DESCRIPTION **NSRecycleZone** frees *zone* after adding any of its pointers still in use to the default zone. (This strategy prevents retained objects from being inadvertently destroyed.)

NSZoneFree returns the memory indicated by *pointer* to *zone*. The standard C function **free** does the same, but spends time finding which zone the memory belongs to.

RETURN Both functions return **void**.

SEE ALSO **NSCreateZone, NSZoneMalloc**

NSResetHashTable

SUMMARY This function deletes the elements of the specified hash table.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS void **NSResetHashTable**(NSHashTable *_table_)

DESCRIPTION **NSResetHashTable** releases each element but doesn't deallocate the table. This is useful for preserving the table's capacity.

SEE ALSO **NSFreeHashTable**

NSResetMapTable

SUMMARY This function deletes the elements of the specified map table.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS void **NSResetMapTable**(NSMapTable *_table_)

DESCRIPTION **NSResetMapTable** releases each key and value but doesn’t deallocate the table. This is useful for preserving the table’s capacity.

SEE ALSO **NSFreeMapTable**

NSRoundDownToMultipleOfPageSize

SUMMARY This function returns the number of pages that correspond to the specified number of bytes.

DECLARED IN Foundation/NSZone.h

SYNOPSIS unsigned int **NSRoundDownToMultipleOfPageSize**(unsigned int *byteCount*)

DESCRIPTION **NSRoundDownToMultipleOfPageSize** returns the multiple of the page size that is closest to, but not greater than, *byteCount*.

SEE ALSO **NSPageSize, NSLogPageSize, NSRoundUpToMultipleOfPageSize**

NSRoundUpToMultipleOfPageSize

SUMMARY This function returns the number of pages that correspond to the specified number of bytes.

DECLARED IN Foundation/NSZone.h

SYNOPSIS unsigned int **NSRoundUpToMultipleOfPageSize**(unsigned int *byteCount*)

DESCRIPTION **NSRoundUpToMultipleOfPageSize** returns the multiple of the page size that is closest to, but not less than, *byteCount*.

SEE ALSO **NSPageSize, NSLogPageSize, NSRoundDownToMultipleOfPageSize**

NSSearchPathForDirectoriesInDomains

SYNOPSIS NSArray ***NSSearchPathForDirectoriesInDomains** (NSSearchPathDirectory *directory*, NSSearchPathDomainMask *domainMask*, BOOL *expandTilde*)

DESCRIPTION <<Description forthcoming>>

NSSelectorFromString

SUMMARY This function obtains a selector by name.

DECLARED IN Foundation/NSObjCRuntime.h

SYNOPSIS SEL **NSSelectorFromString(**NSString *_aSelectorName_**)**

DESCRIPTION **NSSelectorFromString** returns the selector named by *aSelectorName*, or zero if none by this name exists.

SEE ALSO **NSStringFromSelector**

NSSetUncaughtExceptionHandler

SUMMARY This function changes the top level error handler.

DECLARED IN Foundation/NSException.h

SYNOPSIS void **NSSetUncaughtExceptionHandler(**NSUncaughtExceptionHandler *_handler_**)**

DESCRIPTION **NSSetUncaughtExceptionHandler** sets the top-level error-handling function to *handler*. If *handler* is NULL or this function is never invoked, the default top-level handler is used.

SEE ALSO **NSGetUncaughtExceptionHandler**

NSSetZoneName

SUMMARY This function sets the name of the specified zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void **NSSetZoneName(**NSZone *_zone_, NSString *_name_**)**

DESCRIPTION **NSSetZoneName** sets the specified *zone*'s name to *name*, which can aid in debugging.

SEE ALSO **NSZoneName**

NSShouldRetainWithZone

SUMMARY This function indicates whether an object should be retained.

DECLARED IN Foundation/NSObject.h

SYNOPSIS BOOL **NSShouldRetainWithZone(**id <NSObject> *anObject*, NSZone **requestedZone***)**

DESCRIPTION Returns YES if *requestedZone* is NULL, the default zone, or the zone in which *anObject* was allocated. This function is typically called from inside an NSObject's **copyWithZone:** method, when deciding whether to retain *anObject* as opposed to making a copy of it.

NSSizeFromString

SUMMARY This function returns an NSSize from a text-based representation.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSSize **NSSizeFromString**(NSString **aString*)

DESCRIPTION Scans *aString* for text of the form "{width=*a*; height=*b*}" and returns the values for *a* and *b* in an NSSize object. If *aString* does not contain the specified text, this function returns an NSSize object whose width and height are both 0.

SEE ALSO **NSStringFromSize**

NSStandardApplicationPaths

SUMMARY This function returns the application paths for the user's system.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSArray* **NSStandardApplicationPaths**(void)

DESCRIPTION Returns an array of strings, each string specifying one of the standard paths to the OpenStep applications.

SEE ALSO **NSStandardLibraryPaths**

NSStandardLibraryPaths

SUMMARY This function returns the library paths for the user's system.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSArray* **NSStandardLibraryPaths**(void)

DESCRIPTION Returns an array of strings, each string specifying one of the standard paths to the OpenStep libraries.

SEE ALSO **NSStandardApplicationPaths**

NSStringFromClass

SUMMARY This function obtains the name of a class

DECLARED IN Foundation/NSObjCRuntime.h

SYNOPSIS NSString ***NSStringFromClass**(Class *aClass*)

DESCRIPTION This function returns an NSString containing the name of *aClass*.

SEE ALSO **NSClassFromString**

NSStringFromHashTable

SUMMARY This function returns a string describing the hash table's contents.

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS NSString ***NSStringFromHashTable**(NSHashTable **table*)

DESCRIPTION Returns a string describing the hash table's contents. The function iterates over the table's elements, and for each one appends the string returned by the **describe** call-back function. If NULL was specified for the call-back function, the hexadecimal value of each pointer is added to the string.

NSStringFromMapTable

SUMMARY This function returns a string describing the map table's contents.

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS NSString ***NSStringFromMapTable**(NSMapTable **table*)

DESCRIPTION Returns a string describing the map table's contents. The function iterates over the table's key/value pairs, and for each one appends the string *"a = b;*\n", where *a* and *b* are the key and value strings returned by the corresponding **describe** call-back functions. If NULL was specified for the call-back function, a and b are the key and value pointers, expressed as hexadecimal numbers.

NSStringFromPoint

SUMMARY This function returns a string representation of a point.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSString ***NSStringFromPoint**(NSPoint *aPoint*)

DESCRIPTION Returns a string of the form "{x=*a*; y=*b*}", where *a* and *b* are the x- and y-coordinates of *aPoint*.

SEE ALSO **NSPointFromString**

NSStringFromRange

SUMMARY This function returns a string representation of a range.

DECLARED IN Foundation/NSRange.h

SYNOPSIS NSString ***NSStringFromRange**(NSRange *aRange*)

DESCRIPTION Returns a string of the form: "{location = *a*; length = *b*}", where *a* and *b* are non-negative integers representing *aRange*.

NSStringFromRect

SUMMARY This function returns a string representation of a rect.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSString ***NSStringFromRect**(NSRect *aRect*)

DESCRIPTION Returns a string of the form "{x=*a*; y=*b*; width=*c*; height=*d*}", where *a*, *b*, *c*, and *d* are the x- and y-coordinates and the width and height, respectively, of *aRect*.

SEE ALSO **NSRectFromString**

NSStringFromSelector

SUMMARY This function returns the name of a selector.

DECLARED IN Foundation/NSObjCRuntime.h

SYNOPSIS NSString ***NSStringFromSelector(**SEL *aSelector***)**

DESCRIPTION **NSStringFromSelector** returns an NSString containing the name of *aSelector.*

SEE ALSO **NSSelectorFromString**

NSStringFromSize

SUMMARY This function returns a string representation of a size.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSString ***NSStringFromSize**(NSSize *aSize*)

DESCRIPTION Returns a string of the form "{width=*a*; height=*b*}", where *a* and *b* are the width and height of *aSize*.

SEE ALSO **NSSizeFromString**

NSSwapBigDoubleToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS double **NSSwapBigDoubleToHost**(NSSwappedDouble *x*)

DESCRIPTION Converts the big-endian value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapDouble** to perform the swap.

SEE ALSO **NSSwapHostDoubleToBig, NSSwapLittleDoubleToHost**

NSSwapBigFloatToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS float **NSSwapBigFloatToHost**(NSSwappedFloat *x*)

DESCRIPTION Converts the big-endian value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapFloat** to perform the swap.

SEE ALSO **NSSwapHostFloatToBig, NSSwapLittleFloatToHost**

NSSwapBigIntToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned int **NSSwapBigIntToHost**(unsigned int *x*)

DESCRIPTION Converts the big-endian value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapInt** to perform the swap.

SEE ALSO **NSSwapHostIntToBig, NSSwapLittleIntToHost**

NSSwapBigLongLongToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long long **NSSwapBigLongLongToHost**(unsigned long long *x*)

DESCRIPTION Converts the big-endian value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapLongLong** to perform the swap.

SEE ALSO **NSSwapHostLongLongToBig, NSSwapLittleLongLongToHost**

NSSwapBigLongToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long **NSSwapBigLongToHost**(unsigned long *x*)

DESCRIPTION Converts the big-endian value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapLong** to perform the swap.

SEE ALSO **NSSwapHostLongToBig, NSSwapLittleLongToHost**

NSSwapBigShortToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned short **NSSwapBigShortToHost**(unsigned short *x*)

DESCRIPTION Converts the big-endian value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapShort** to perform the swap.

SEE ALSO **NSSwapHostShortToBig, NSSwapLittleShortToHost**

NSSwapDouble

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedDouble **NSSwapDouble**(NSSwappedDouble *x*)

DESCRIPTION Swaps the bytes of *x* and returns the resulting value. Bytes are swapped from each low-order position to the corresponding high-order position and vice versa. For example, if the bytes of *x* are numbered from 1 to 8, this function swaps bytes 1 and 8, bytes 2 and 7, bytes 3 and 6, and bytes 4 and 5.

SEE ALSO **NSSwapLongLong, NSSwapFloat**

NSSwapFloat

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedFloat **NSSwapFloat**(NSSwappedFloat *x*)

DESCRIPTION Swaps the bytes of *x* and returns the resulting value.Bytes are swapped from each low-order position to the corresponding high-order position and vice versa. For example, if the bytes of *x* are numbered from 1 to 4, this function swaps bytes 1 and 4, and swaps bytes 2 and 3.

SEE ALSO **NSSwapLong, NSSwapDouble**

NSSwapHostDoubleToBig

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedDouble **NSSwapHostDoubleToBig**(double *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to big-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapDouble** to perform the swap.

SEE ALSO **NSSwapBigDoubleToHost, NSSwapHostDoubleToLittle**

NSSwapHostDoubleToLittle

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedDouble **NSSwapHostDoubleToLittle**(double *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to little-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapDouble** to perform the swap.

SEE ALSO **NSSwapLittleDoubleToHost, NSSwapHostDoubleToBig**

NSSwapHostFloatToBig

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedFloat **NSSwapHostFloatToBig**(float *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to big-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapFloat** to perform the swap.

SEE ALSO **NSSwapBigFloatToHost, NSSwapHostFloatToLittle**

NSSwapHostFloatToLittle

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS NSSwappedFloat **NSSwapHostFloatToLittle**(float *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to little-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapFloat** to perform the swap.

SEE ALSO **NSSwapLittleFloatToHost, NSSwapHostFloatToBig**

NSSwapHostIntToBig

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned int **NSSwapHostIntToBig**(unsigned int *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to big-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapInt** to perform the swap.

SEE ALSO **NSSwapBigIntToHost, NSSwapHostIntToLittle**

NSSwapHostIntToLittle

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned int **NSSwapHostIntToLittle**(unsigned int *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to little-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapInt** to perform the swap.

SEE ALSO **NSSwapLittleIntToHost, NSSwapHostIntToBig**

NSSwapHostLongLongToBig

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long long **NSSwapHostLongLongToBig**(unsigned long long *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to big-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapLongLong** to perform the swap.

SEE ALSO **NSSwapBigLongLongToHost, NSSwapHostLongLongToLittle**

NSSwapHostLongLongToLittle

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long long **NSSwapHostLongLongToLittle**(unsigned long long *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to little-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapLongLong** to perform the swap.

SEE ALSO **NSSwapLittleLongLongToHost, NSSwapHostLongLongToBig**

NSSwapHostLongToBig

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long **NSSwapHostLongToBig**(unsigned long *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to big-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapLong** to perform the swap.

SEE ALSO **NSSwapBigLongToHost, NSSwapHostLongToLittle**

NSSwapHostLongToLittle

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long **NSSwapHostLongToLittle**(unsigned long *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to little-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapLong** to perform the swap.

SEE ALSO **NSSwapLittleLongToHost, NSSwapHostLongToBig**

NSSwapHostShortToBig

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned short **NSSwapHostShortToBig**(unsigned short *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to big-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapShort** to perform the swap.

SEE ALSO **NSSwapBigShortToHost, NSSwapHostShortToLittle**

NSSwapHostShortToLittle

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned short **NSSwapHostShortToLittle**(unsigned short *x*)

DESCRIPTION Converts the value in *x*, specified in the endian format of the current platform, to little-endian format and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapShort** to perform the swap.

SEE ALSO **NSSwapLittleShortToHost, NSSwapHostShortToBig**

NSSwapInt

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned int **NSSwapInt** (unsigned int *inv*)

DESCRIPTION Swaps the bytes of *inv* and returns the resulting value. Bytes are swapped from each low-order position to the corresponding high-order position and vice versa. For example, if the bytes of *inv* are numbered from 1 to 4, this function swaps bytes 1 and 4, and swaps bytes 2 and 3.

SEE ALSO **NSSwapShort, NSSwapLong, NSSwapLongLong**

NSSwapLittleDoubleToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS double **NSSwapLittleDoubleToHost**(NSSwappedDouble *x*)

DESCRIPTION Converts the little-endian formatted value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapDouble** to perform the swap.

SEE ALSO **NSSwapHostDoubleToLittle, NSSwapBigDoubleToHost, NSConvertSwappedDoubleToHost**

NSSwapLittleFloatToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS float **NSSwapLittleFloatToHost**(NSSwappedFloat *x*)

DESCRIPTION Converts the little-endian formatted value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapFloat** to perform the swap.

SEE ALSO **NSSwapHostFloatToLittle, NSSwapBigFloatToHost, NSConvertSwappedFloatToHost**

NSSwapLittleIntToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned int **NSSwapLittleIntToHost**(unsigned int *x*)

DESCRIPTION Converts the little-endian formatted value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapInt** to perform the swap.

SEE ALSO **NSSwapHostIntToLittle, NSSwapBigIntToHost**

NSSwapLittleLongLongToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long long **NSSwapLittleLongLongToHost**(unsigned long long *x*)

DESCRIPTION Converts the little-endian formatted value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes, this function calls **NSSwapLongLong** to perform the swap.

SEE ALSO **NSSwapHostLongLongToLittle, NSSwapBigLongLongToHost**

NSSwapLittleLongToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long **NSSwapLittleLongToHost**(unsigned long *x*)

DESCRIPTION Converts the little-endian formatted value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapLong** to perform the swap.

SEE ALSO **NSSwapHostLongToLittle, NSSwapBigLongToHost, NSSwapLong**

NSSwapLittleShortToHost

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned short **NSSwapLittleShortToHost**(unsigned short *x*)

DESCRIPTION Converts the little-endian formatted value in *x* to the endian format of the current platform and returns the resulting value. If it is necessary to swap the bytes of *x*, this function calls **NSSwapShort** to perform the swap.

SEE ALSO **NSSwapHostShortToLittle, NSSwapBigShortToHost**

NSSwapLong

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long **NSSwapLong**(unsigned long *inv*)

DESCRIPTION Swaps the bytes of *inv* and returns the resulting value. Bytes are swapped from each low-order position to the corresponding high-order position and vice versa. For example, if the bytes of *inv* are numbered from 1 to 4, this function swaps bytes 1 and 4, and swaps bytes 2 and 3.

SEE ALSO **NSSwapLongLong, NSSwapInt, NSSwapFloat**

NSSwapLongLong

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned long long **NSSwapLongLong**(unsigned long long *inv*)

DESCRIPTION Swaps the bytes of *inv* and returns the resulting value. Bytes are swapped from each low-order position to the corresponding high-order position and vice versa. For example, if the bytes of *inv* are numbered from 1 to 8, this function swaps bytes 1 and 8, bytes 2 and 7, bytes 3 and 6, and bytes 4 and 5.

SEE ALSO **NSSwapLong, NSSwapDouble**

NSSwapShort

SUMMARY This function is a utility for swapping the bytes of a number.

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS unsigned short **NSSwapShort** (unsigned short *inv*)

DESCRIPTION Swaps the low-order and high-order bytes of *inv* and returns the resulting value.

SEE ALSO **NSSwapInt, NSSwapLong**

NSTemporaryDirectory

SUMMARY This function returns the temporary directory on the user's system.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSString ***NSTemporaryDirectory**(void)

DESCRIPTION Returns a string containing the path of the current temporary directory. If no such directory is currently available, this function returns **nil**.

SEE ALSO **NSStandardApplicationPaths, NSStandardLibraryPaths, NSHomeDirectory**

NSUnionRange

SUMMARY This function returns the intersection of the specified ranges.

DECLARED IN Foundation/NSRange.h

SYNOPSIS NSRange **NSUnionRange(**NSRange *range1*, NSRange *range2***)**

DESCRIPTION **NSUnionRange** returns a range covering all indices in and between *range1* and *range2*. If one range is completely contained in the other, the returned range is equal to the larger range.

SEE ALSO **NSIntersectionRange**

NSUnionRect

SUMMARY This function calculates the union of two rectangles.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS NSRect **NSUnionRect**(NSRect *aRect*, NSRect *bRect*)

DESCRIPTION **NSUnionRect** returns the smallest rectangle that completely encloses both *aRect* and *bRect*. If one of the rectangles has zero (or negative) width or height, a copy of the other rectangle is returned; but if both have zero (or negative) width or height, the returned rectangle has its origin at (0.0, 0.0) and has zero width and height.

SEE ALSO **NSIntersectionRect**

NSUserName

SUMMARY This function gets information about a user.

DECLARED IN Foundation/NSPathUtilities.h

SYNOPSIS NSString ***NSUserName(**void**)**

DESCRIPTION **NSUserName** returns the logon name of the current user.

SEE ALSO **NSFullUserName, NSHomeDirectory, NSHomeDirectoryForUser**

NSWidth

SUMMARY This function returns the width of the specified rectangle.

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS float **NSWidth**(NSRect *aRect*)

DESCRIPTION **NSWidth** returns the width of *aRect*.

SEE ALSO **NSMaxX, NSMaxY, NSMidX, NSMidY, NSMinX, NSMinY, NSHeight**

NSZoneCalloc

SUMMARY This function allocates memory in a zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void ***NSZoneCalloc(**NSZone **zone*, unsigned int *numElems*, unsigned int *byteSize***)**

DESCRIPTION **NSZoneCalloc** allocates enough memory from *zone* for *numElems* elements, each with a size *numBytes* bytes, and returns a pointer to the allocated memory. The memory is initialized with zeros. This function returns **nil** if it was unable to allocate the requested memory.

SEE ALSO **NSDefaultMallocZone, NSRecycleZone, NSZoneFree, NSZoneMalloc, NSZoneRealloc**

NSZoneFree

SUMMARY This function deallocates a block of memory in the specified zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void **NSZoneFree**(NSZone **zone*, void ***pointer*)

DESCRIPTION Returns memory to the zone from which it was allocated. The standard C function **free** does the same, but spends time finding which zone the memory belongs to.

SEE ALSO **NSRecycleZone, NSZoneMalloc, NSZoneCalloc, NSZoneRealloc**

NSZoneFromPointer

SUMMARY This function gets the zone for a given block of memory

DECLARED IN Foundation/NSZone.h

SYNOPSIS NSZone *__NSZoneFromPointer__(void **pointer*)

DESCRIPTION Returns the zone for the block of memory indicated by *pointer*, or NULL if the block was not allocated from a zone. The pointer must be one that was returned by a prior call to an allocation function.

RETURN Returns the zone for the indicated block of memory, or NULL if the block was not allocated from a zone.

SEE ALSO **NSZoneCalloc, NSZoneMalloc, NSZoneRealloc**

NSZoneMalloc

SUMMARY This function allocates memory in a zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void ***NSZoneMalloc(**NSZone **zone*, unsigned int *size***)**

DESCRIPTION **NSZoneMalloc** allocates *size* bytes in *zone*, and returns a pointer to the allocated memory. This function returns **nil** if it was unable to allocate the requested memory.

SEE ALSO **NSDefaultMallocZone, NSRecycleZone, NSZoneFree, NSZoneCalloc, NSZoneRealloc**

NSZoneName

SUMMARY This function returns the name of the specified zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS NSString ***NSZoneName(**NSZone **zone***)**

DESCRIPTION Returns a string containing the name associated with *zone*, If *zone* is **nil**, the default zone is used. If no name is associated with *zone*, the returned string is empty.

SEE ALSO **NSSetZoneName**

NSZoneRealloc

SUMMARY This function allocates memory in a zone.

DECLARED IN Foundation/NSZone.h

SYNOPSIS void ***NSZoneRealloc(**NSZone **zone*, void **ptr*, unsigned int *size***)**

DESCRIPTION **NSZoneRealloc** changes the size of the block of memory pointed to by *ptr* to *size* bytes. It may allocate new memory to replace the old, in which case it moves the contents of the old memory block to the new block, up to a maximum of *size* bytes. *ptr* may be NULL. This function returns **nil** if it was unable to allocate the requested memory.

SEE ALSO **NSDefaultMallocZone, NSRecycleZone, NSZoneFree, NSZoneCalloc, NSZoneMalloc**

Locales

A locale is a set of defaults whose values are determined by the user's preferred language. The locale defaults are stored in the NSUserDefault's preferred language domain.

Certain classes from the Foundation Framework (for example, NSCalendarDate, NSDate, NSTimeZone, NSString, and NSScanner) use information from the locale to determine their values. For example, when you request an NSString representation of an NSCalendarDate, the NSCalendarDate looks at the locale to determine what the months and the days of the week are called in the user's preferred language.

The actual attributes for the locale keys are contained in **NextLibrary/Frameworks/Foundation.framework/Resources/Languages**. If a given language does not define one of the locale attributes, the value for that attribute defaults to the one defined for the default language (US English).

The Foundation Framework defines these locale dictionary keys. Other frameworks may add other keys to the locale dictionary.

Key	Description
NSAMPMDesignation	An array of strings that specify how the morning and afternoon designations are printed. The default is AM and PM.
NSCurrencySymbol	A string that specifies the symbol used to denote currency in this language. The default is "$".
NSDateFormatString	A format string that specifies how dates are printed using the date format specifiers. (See the NSCalendarDate class specification for a list of these.) The default is to use weekday names with full month names and full years, as in “Sunday, January 01, 1995.”
NSDateTimeOrdering	A string that specifies how to use ambiguous numbers in date strings. Specify this value as a permutation of the letters M (month), D (day), Y (year), and H (hour). For example, MDYH treats “2/3/95 10” as the 3rd day of February 1995 at 10:00am, whereas DMYH treats the same value as the 2nd day of March 1995 at 10:00am. If fewer numbers are specified than are needed, the numbers are prioritized to satisfy day first, then the month, and then the year. For example, if you supply only the value 12, it means the 12th day of this month in this year because the day must be specified. If you supply “2 12” it means either February 12 or December 2, depending on if the ordering is “MDYH” or “DMYH.”
NSDecimalDigits	Strings that identify the decimal digits in addition to or instead of the ASCII digits.
NSDecimalSeparator	A string that specifies the decimal separator. The decimal separator separates the ones place from the tenths place. The default is “.”.
NSEarlierTimeDesignations	An array of strings that denote a time in the past. These are adjectives that modify values from NSYearMonthWeekDesignations. The defaults are “prior,” “last,” “past,” and “ago.”
NSHourNameDesignations	Strings that identify the time of day. These strings should be bound to an hour. The default is this array of arrays: (0, midnight), (12, noon, lunch), (10, morning), (14, afternoon), (19, dinner).
NSInternationalCurrencyString	A string containing three letter abbreviation for currency, following the ISO 4217 standard.
NSLaterTimeDesignations	An array of strings that denote a time in the future. This is an adjective that modifies a value from NSYearMonthWeekDesignations. The default is “next.”.
NSMonthNameArray	An array that specifies the full names for the months.
NSNextDayDesignations	A string that identifies the day after today. The default is “tomorrow.”
NSNextNextDayDesignations	A string that identifies the day after tomorrow. The default is “nextday”.
NSPriorDayDesignations	A string that identifies the day before today. The default is “yesterday.”
NSShortDateFormatString	A format string that specifies how dates are abbreviated. (See the

	NSCalendarDate class specification for a list of the date format specifiers to use.) The default is to separate the day month and year with slashes and to put the day first, as in 31/10/95.
NSShortMonthNameArray	An array that specifies the abbreviations for the months.
NSShortWeekDayNameArray	An array that specifies the abbreviations for the days of the week. Sunday should be the first day of the week.
NSShortTimeDateFormatString	A format string that specifies how times and dates are abbreviated. (See the NSCalendarDate class specification for a list of the date format specifiers.) The default is to use dashes to separate the day, month, and year and to use a 12-hour clock, as in "31-Jan-95 1:30 PM.
NSThisDayDesignations	A string that identifies what this day is called. The default is "today."
NSThousandsSeparator	A string that specifier the separator character for the thousands place of a decimal number. The default is a comma.
NSTimeDateFormatString	A format string how dates with times are printed.(See the NSCalendarDate class specification for a list of the date format specifiers.) The default is to use abbreviated months and days with a 24 hour clock, as in "Sun Jan 01 23:00:00 +6 2001."
NSTimeFormatString	A format string how dates with times are printed.(See the NSCalendarDate class specification for a list of the date format specifiers.) The default is to use a 24 hour clock, as in 13:30:25.
NSWeekDayNameArray	An array that gives the names for the days of the week. Sunday should be the first day of the week.
NSYearMonthWeekDesignations	An array of strings that specify the word for year, month, and week in the current locale. The defaults are "year," "month," and "week."

NSCoding

Adopted By: Various OpenStep classes

Declared In: Foundation/NSObject.h

Protocol Description

The NSCoding protocol declares the two methods that a class must implement so that instances of that class can be encoded and decoded. This capability provides the basis for archiving (where objects and other structures are stored on disk) and distribution (where objects are copied to different address spaces). See the NSCoder and NSArchiver class specifications for an introduction to coding.

In keeping with object-oriented design principles, an object being encoded or decoded is responsible for encoding and decoding its instance variables. A coder instructs the object to do so by invoking **encodeWithCoder:** or **initWithCoder:**. **encodeWithCoder:** instructs the object to encode its instance variables to the coder provided; an object can receive this method any number of times. **initWithCoder:** instructs the object to initialize itself from data in the coder provided; as such, it replaces any other initialization method and is only sent once per object. Any object class that should be codable must adopt the NSCoding protocol and implement its methods.

When an object receives an **encodeWithCoder:** message, it should encode all of its vital instance variables, after sending a message to **super** if its superclass also conforms to the NSCoding protocol. An object doesn't have to encode all of its instance variables. Some values may not be important to reestablish and others may be derivable from related state upon decoding. Other instance variables should be encoded only under certain conditions (for example, with **encodeConditionalObject:**, as described in the NSArchiver class specification).

For example, suppose you were creating a fictitious MapView class that displays a legend and a map at various magnifications. The MapView class defines several instance variables, including the name of the map and the current magnification. The MapView class also contains instance variables for several related views. The **encodeWithCoder:** method of MapView might look like the following:

```
- (void)encodeWithCoder:(NSCoder *)coder
{
    [super encodeWithCoder:coder];
    [coder encodeValueOfObjCType:@encode(char *) at:mapName];
    [coder encodeValueOfObjCType:@encode(unsigned int) at:&magnification];
    [coder encodeObject:legendView];
    [coder encodeConditionalObject:auxiliaryView];
    return;
}
```

This example assumes that the superclass of MapView also supports the NSCoding protocol. If the superclass of your class does not support NSCoding, you should omit the line that invokes super's **encodeWithCoder:** method.

encodeValueOfObjCType:at: and **encodeObject:** are coder methods that you can use to encode instance variables of your class. You can use these and other methods of the coder to encode id's, scalars, C arrays, structs, strings, and pointers to any of these types. The coder also defines corresponding methods for decoding values. See the NSCoder, NSArchiver, and NSUnarchiver class specifications for a list of methods.

The **@encode()** compiler directive generates an Objective-C type code from a type expression. See *Object-Oriented Programming and the Objective-C Language* for more information.

Similarly, in **initWithCoder:** the object should first send a message to **super** (if appropriate) to initialize inherited instance variables, and then it should decode and initialize its own. MapView's implementation of **initWithCoder:** might look like this:

```
- (id)initWithCoder:(NSCoder *)coder
{
    self = [super initWithCoder:coder];
    [coder decodeValueOfObjCType:@encode(char *) at:mapName];
    [coder decodeValueOfObjCType:@encode(unsigned int) at:&magnification];
    legendView = [[coder decodeObject] retain];
    auxiliaryView = [[coder decodeObject] retain];
    return self;
}
```

Note the assignment of the return value of **initWithCoder:** to **self** in the example above. This is done in the subclass because the superclass, in its implementation of **initWithCoder:**, may decide to return a object other than itself. If the superclass of your class does not support NSCoding, you should invoke super's designated initializer instead of **initWithCoder:**.

Making Substitutions During Coding

During encoding or decoding a coder object invokes methods that allow the object being coded to substitute a replacement class or instance for itself. This allows archives to be shared among implementations with different class hierarchies or simply different versions of a class (for example, class clusters take advantage of this feature). It also allows classes that should maintain unique instances to enforce this policy on decoding (for example, there need only be a single NSFont instance for a given typeface and size).

Substitution methods are declared by NSObject, and come in two flavors: generic and specialized. The generic methods are these:

Method	Typical Use
classForCoder	Allows an object, before being encoded, to substitute a class other than its own. For example, the private subclasses of a class cluster substitute the name of their public superclass when being archived.
replacementObjectForCoder:	Allows an object, before being encoded, to substitute another instance in its place.
awakeAfterUsingCoder:	Allows an object, after being decoded, to substitute another object for itself. For example, an object that represents a font might, upon being decoded, release itself and return an existing object having the same font description as itself. In this way, redundant objects can be eliminated.

The specialized substitution methods are analogous to **classForCoder** and **replacementObjectForCoder:** , but they're designed for (and invoked by) a specific, concrete coder subclass. NSArchiver invokes **classForArchiver:** and **replacementObjectForArchiver:**, while NSPortCoder invokes **classForPortCoder** and **replacementObjectForPortCoder:**. (There isn't a specialized version of **awakeAfterUsingCoder:**.) By implementing these specialized methods, your class can base its coding behavior on the specific coder class being used. For more information on these methods, see their method descriptions in the NSObject class specification, as well as the class description in the NSPortCoder class specification.

Method Types

Encoding and decoding objects

– encodeWithCoder:
– initWithCoder:

Instance Methods

encodeWithCoder:

– (void)**encodeWithCoder:**(NSCoder *)*encoder*

Encodes the receiver using *encoder.*

initWithCoder:

– (id)**initWithCoder:**(NSCoder *)*decoder*

Initializes a newly allocated instance from data in *decoder*. Returns **self**.

NSCopying

Adopted By: Various OpenStep classes

Declared In: Foundation/NSObject.h

Protocol Description

The NSCopying protocol declares a method for providing functional copies of an object. The exact meaning of “copy” can vary from class to class, but a copy must be a functionally independent object with values identical to the original at the time the copy was made. A copy produced with NSCopying is implicitly retained by the sender, who is responsible for releasing it.

NSCopying declares one method, **copyWithZone:**, but copying is commonly invoked with the convenience method **copy**. The **copy** method is defined for all NSObjects and simply invokes **copyWithZone:** with the default zone.

Using NSCopying

NSCopying is frequently used to copy *value* objects—objects that represent attributes. C-type variables can usually be substituted for value objects, but value objects have the advantage of encapsulating convenient utilities for common manipulations. For example, NSString objects are used instead of character pointers because they encapsulate encoding and storage. Despite NSString functionality, the role played by NSStrings parallels the role played by character pointers.

When value objects are passed as method arguments or returned from a method, it is common to use a copy instead of the object itself. For example, consider the following method for assigning a string to an object’s **name** instance variable.

```
- (void)setName:(NSString *)aName
{
    [name autorelease];
    name = [aName copy];
}
```

Storing a copy of **aName** has the effect of producing an object that’s independent of the original, but has the same contents. Subsequent changes to the copy don’t affect the original, and changes to the original don’t affect the copy. Similarly, it is common to return a copy of an instance variable instead of the instance variable itself. For example, this method returns a copy of the **name** instance variable:

```
- (NSString *)name
{
    return [[name copy] autorelease];
}
```

Implementing NSCopying

There are two basic approaches to creating copies. You can use **alloc** and **init...**, or you can use **NSCopyObject()**. To choose the one that's right for your class, you need to consider the following questions:

- "What kind of copying—deep or shallow—does your class need?"
- "Does your class's superclass implement NSCopying?"
- "Are you familiar with the implementations of your class's superclasses?"

These areas are described in the following sections.

What kind of copying—deep or shallow—does your class need?

Generally, copying an object involves creating a new instance and initializing it with the values in the original object. Copying the values for non-pointer instance variables, such as booleans, integers, and floating points, is straightforward. When copying pointer instance variables there are two approaches. One approach, called a *shallow copy*, copies the pointer value from the original object into the copy. Thus, the original and the copy share referenced data. The other approach, called a *deep copy*, duplicates the data referenced by the pointer and assigns it to the copy's instance variable.

The implementation of an instance variable's set method should reflect the kind of copying you need to use. You should deeply copy the instance variable if the corresponding set method copies the new value as in this method:

```
- (void)setMyVariable:(id)newValue
{
    [myVariable autorelease];
    myVariable = [newValue copy];
}
```

You should shallowly copy the instance variable if the corresponding set method retains the new value as in this method:

```
- (void)setMyVariable:(id)newValue
{
    [myVariable autorelease];
    myVariable = [newValue retain];
}
```

Similarly, you should shallowly copy the instance variable if its set method simply assigns the new value to the instance variable without copying or retaining it as in this method:

```
- (void)setMyVariable:(id)newValue
{
    myVariable = newValue;
}
```

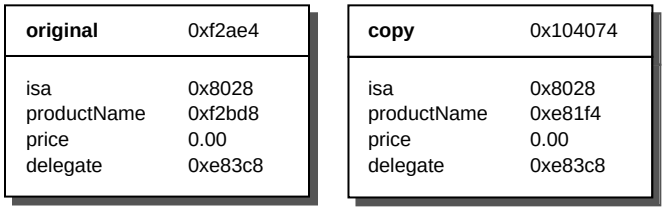
To produce a copy of an object that's truly independent of the original, the entire object must be deeply copied. Every instance variable must be duplicated. If the instance variables themselves have instance variables, those too must be duplicated, and so on. In many cases, a mixed approach is more useful. Pointer instance variables that can be thought of as data containers are generally deeply copied, while more sophisticated instance variables like delegates are shallowly copied.

For example, a Product class adopts NSCopying. Product instances have a name, a price, and a delegate as declared in this interface.

```
@interface Product : NSObject <NSCopying>
{
    NSString *productName;
    float price;
    id delegate;
}

@end
```

Copying a Product instance produces a deep copy of **productName** because it represents a flat data value. On the other hand, the **delegate** instance variable is a more complex object capable of functioning properly for both Products. The copy and the original should therefore share the delegate. The following figure represents the images of a Product instance and a copy in memory.



The different pointer values for **productName** illustrate that the original and the copy each have their own **productName** string object. The pointer values for **delegate** are the same, indicating that the two product objects share the same object as their delegate.

Does your class's superclass implement NSCopying?

If the superclass does not implement NSCopying, your class's implementation will have to copy the instance variables it inherits as well as those declared in your class. Generally, the safest way to do this is by using **alloc**, **init...**, and set methods. On the other hand, if your class inherits NSCopying behavior, its implementation only has to copy instance variables declared in your class. It invokes the superclass's implementation to copy inherited instance variables.

Are you familiar with the implementations of your class's superclasses?

If your class inherits NSCopying behavior, how you handle the new instance variables in **copyWithZone:** depends on your familiarity with the superclass's implementation. There are essentially two ways to make a copy of an object, using **alloc** and **init...** or using the function **NSCopyObject()**. If the superclass used or might have used **NSCopyObject()**, you must handle instance variables differently than you would otherwise.

Using the alloc, init... Approach

If a class does not inherit NSCopying behavior, you should implement **copyWithZone:** using **alloc**, **init...**, and set methods. For example, an implementation of **copyWithZone:** for the Product class described above might be implemented in the following way:

```
- (id)copyWithZone:(NSZone *)zone
{
    Product *copy = [[Product alloc]
           initWithProductName:[self productName]
           price:[self price]];
    [copy setDelegate:[self delegate]];

    return copy;
}
```

Because implementation details associated with inherited instance variables are encapsulated in the superclass, it is generally better to implement NSCopying with the **alloc**, **init...** approach. Doing so uses policy implemented in set methods to determine the kind of copying needed of instance variables.

Using NSCopyObject()

When a class inherits NSCopying behavior, you must consider the possibility that the superclass's implementation uses **NSCopyObject()**. **NSCopyObject()** creates an exact shallow copy of an object by copying instance variable values but not the data they point to. For example, NSCell's implementation of **copyWithZone:** could be defined in the following way.

```
- (id)copyWithZone:(NSZone *)zone
{
    NSCell *cellCopy = NSCopyObject(self, 0, zone);
    /* Assume that other initialization takes place here. */

    cellCopy->image = nil;
    [cellCopy setImage:[self image]];

    return cellCopy;
}
```

In the implementation above, **NSCopyObject()** creates an exact shallow copy of the original cell. This behavior is desirable for copying instance variables that aren't pointers or are pointers to non-retained data that is shallowly copied. Pointer instance variables for retained objects need additional treatment.

In the **copyWithZone:** example above, **image** is a pointer to a retained object. The policy to retain the image is reflected in the following implementation of the **setImage:** accessor method.

```
- (void)setImage:(NSImage *)anImage
{
    [image autorelease];
    image = [anImage retain];
}
```

Notice that **setImage:** autoreleases **image** before it reassigns it. If the above implementation of **copyWithZone:** hadn't explicitly set the copy's **image** instance variable to **nil** before invoking **setImage:**, the image referenced by the copy and the original would be released without a corresponding retain.

Even though **image** points to the right object, it is conceptually uninitialized. Unlike the instance variables that are created with **alloc** and **init...**, these uninitialized variables aren't **nil**-valued. You should explicitly assign initial values to these variables before using them. In this case, **cellCopy**'s **image** instance variable is set to **nil**, then it is set using the **setImage:** method.

The effects of **NSCopyObject()** extend to a subclass's implementation. For example, an implementation of NSSliderCell could copy a new **titleCell** instance variable in the following way.

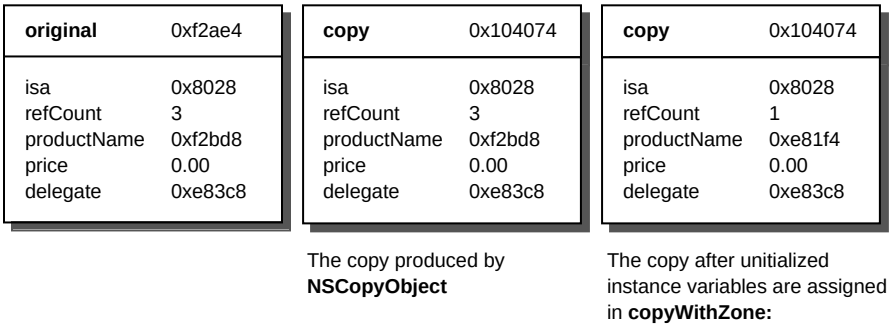
```
- (id)copyWithZone:(NSZone *)zone
{
    NSSliderCell *cellCopy = [super copyWithZone:zone];
    /* Assume that other initialization takes place here. */

    cellCopy->titleCell = nil;
    [cellCopy setTitleCell:[self titleCell]];

    return cellCopy;
}
```

The superclass's **copyWithZone:** method is invoked to copy inherited instance variables. When you invoke a superclass's **copyWithZone:** method, assume that new object instance variables are uninitialized if there's any chance that the superclass implementation uses **NSCopyObject()**. Explicitly assign a value to them before using them. In this example, **titleCell** is explicitly set to **nil** before **setTitleCell:** is invoked.

The implementation of an object's retain count is another consideration when using **NSCopyObject()**. If an object stores its retain count in an instance variable, the implementation of **copyWithZone:** must correctly initialize the copy's retain count. The following figure illustrates the process.



The copy produced by **NSCopyObject**

The copy after unitialized instance variables are assigned in **copyWithZone:**

The first object represents a Product instance in memory. The value in **refCount** indicates that the instance has been retained three times. The second object is a copy of the Product instance produced with **NSCopyObject()**. Its **refCount** value matches the original. The third object represents the copy returned from **copyWithZone:** after **refCount** is correctly initialized. After **copyWithZone:** creates the copy with **NSCopyObject()**, it assigns the value 1 to the **refCount** instance variable. The sender of **copyWithZone:** implicitly retains the copy and is responsible for releasing it.

NSCopying and Immutable Classes

Where the concept “immutable vs. mutable” applies to an object, NSCopying produces immutable copies whether the original is immutable or not. See the NSMutableCopying protocol for details on making mutable copies.

Immutable classes can implement NSCopying very efficiently. Since immutable objects don’t change, there is no need to duplicate them. Instead, NSCopying can be implemented to **retain** the original. For example, **copyWithZone:** for an immutable string class can be implemented in the following way.

```
- (id)copyWithZone:(NSZone *)zone
{
    return [self retain];
}
```

Summary

- Implement NSCopying using **alloc** and **init...** in classes that don’t inherit **copyWithZone:**.
- Implement NSCopying by invoking the superclass’s **copyWithZone:** when NSCopying behavior is inherited. If the superclass implementation might use **NSCopyObject()**, make explicit assignments to pointer instance variables for retained objects.
- Implement NSCopying by retaining the original instead of creating a new copy when the class and its contents are immutable.

nstance Methods

copyWithZone:

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

Returns a new instance that's a copy of the receiver. Memory for the new instance is allocated from *zone*, which may be NULL. If *zone* is NULL, the new instance is allocated from the default zone, which is returned from **NSDefaultMallocZone()**. The returned object is implicitly retained by the sender, who is responsible for releasing it. The copy returned is immutable if the consideration "immutable vs. mutable" applies to the receiving object; otherwise the exact nature of the copy is determined by the class.

See also: – **mutableCopyWithZone:** (NSMutableCopying protocol), – **copy** (NSObject)

NSDecimalNumberBehaviors

Adopted By: NSDecimalNumberHandler

Declared In: Foundation/NSDecimalNumber.h

Protocol Description

The NSDecimalBehaviors protocol declares three methods that control the discretionary aspects of working with NSDecimalNumbers. The **scale** and **roundingMode** methods determine the precision of NSDecimalNumber's return values, and the way in which those values should be rounded to fit that precision. The **exceptionDuringOperation:error:leftOperand:rightOperand:** determines the way in which an NSDecimalNumber should handle different calculation errors.

For an example of a class that adopts the NSDecimalBehaviors protocol, see the specification for NSDecimalNumberHandler.

Method Types

Rounding

– roundingMode
– scale

Handling errors

– exceptionDuringOperation:error:leftOperand:rightOperand:

Instance Methods

exceptionDuringOperation:error:leftOperand:rightOperand:

– (NSDecimalNumber *)**exceptionDuringOperation:**(SEL)*method*
error:(NSCalculationError)*error*
leftOperand:(NSDecimalNumber *)*leftOperand*
rightOperand:(NSDecimalNumber *)*rightOperand*

Specifies what an NSDecimalNumber will do when, in the course of applying *method* to *leftOperand* and *rightOperand*, it encounters *error.*

There are four possible values for *error.* The first three have to do with limits on NSDecimalNumber's ability to represent decimal numbers. An NSDecimalNumber can represent any number that can be

expressed as *mantissa* **x** 10**exponent**, where *mantissa* is a decimal integer up to 38 digits long, and *exponent* is between -256 and 256. If these limits are exceeded, the NSDecimalNumber returns one of the following errors:

- NSCalculationLossOfPrecision. The number can't be represented in 38 significant digits.
- NSCalculationOverflow. The number is too large to represent.
- NSCalculationUnderflow. The number is too small to represent.

The last error is simpler:

- NSCalculationDivideByZero. The caller tried to divide by zero.

In implementing **exceptionDuringOperation:error:leftOperand:rightOperand**, you can handle each of these errors in several ways:

- Raise an exception. For an explantion of exceptions, see the NSException class description in the *Foundation Framework Reference*.
- Return **nil**. The calling *method* will return its value as though no error had occurred. If *error* is NSCalculationLossOfPrecision, *method* will return an imprecise value—that is, one constrained to 38 significant digits. If *error* is NSCalculationUnderflow or NSCalculationOverflow, *method* will return NSDecimalNumber's **notANumber**. You shouldn't return **nil** if *error* is NSDivideByZero.
- Correct the error and return a valid NSDecimalNumber. The calling method will use this as its own return value.

roundingMode

– (NSRoundingMode)**roundingMode**

Returns the way that NSDecimalNumber's **decimalNumberBy...** methods round their return values. There are four possible NSRoundingModes:

- NSRoundDown. The methods round their return values down.
- NSRoundUp. The methods round their return values up.
- NSRoundPlain. The methods round to the closest possible return value. When they are caught halfway between two positive numbers, they round up; when caught between two negative numbers, they round down.
- NSRoundBankers. The methods round to the closest possible return value. When they are caught halfway between two possibilities, they return the possibility whose last digit is even. In practice, this means that, over the long run, numbers will be rounded up as often as they are rounded down; there will be no systematic bias.

The rounding mode only matters if the **scale** method sets a limit on the precision of NSDecimalNumber return values. It has no effect if **scale** returns NSDecimalNoScale.

Assuming that **scale** returns 1, the NSRoundingMode has the following effects on various original values:

Original value	NSRoundDown	NSRoundUp	NSRoundPlain	NSRoundBankers
1.24	1.2	1.3	1.2	1.2
1.26	1.2	1.3	1.3	1.3
1.25	1.2	1.3	1.3	1.2
1.35	1.3	1.4	1.4	1.4
-1.35	-1.4	-1.3	-1.4	-1.4

scale

– (short)**scale**

Limits the precision of the values returned by NSDecimalNumber's **decimalNumberBy...** methods.

Specifically, **scale** returns the number of digits allowed after the decimal separator. If **scale** returns a negative value, it affects the digits *before* the decimal separator as well. If **scale** returns NSDecimalNoScale, the number of digits is unlimited.

Assuming that **roundingMode** returns NSRoundPlain, different values of **scale** have the following effects on the number 123.456:

Scale	Return value
NSDecimalNoScale	123.456
2	123.45
0	123
-2	100

NSLocking

Adopted By: NSConditionLock
NSLock
NSRecursiveLock

Declared In: Foundation/NSLock.h

Protocol Description

The NSLocking protocol declares the elementary methods adopted by classes that define lock objects. A lock object is used to coordinate the actions of multiple threads of execution within a single application. By using a lock object, an application can protect critical sections of code from being executed simultaneously by separate threads, thus protecting shared data and other shared resources from corruption.

For example, consider a multithreaded application in which each thread updates a shared counter. If two threads simultaneously fetch the current value into local storage, increment it, and then write the value back, the counter will be incremented only once, losing one thread's contribution. However, if the code that manipulates the shared data (the *critical section* of code) must be locked before being executed, only one thread at a time can perform the updating operation, and collisions are prevented.

A lock object is either locked or unlocked. You *acquire* a lock by sending the object a **lock** message, thus putting the object in the locked state. You *relinquish* a lock by sending an **unlock** message, and thus putting the object in the unlocked state. (The NEXTSTEP classes that adopt this protocol define additional ways to acquire and relinquish locks.)

The **lock** method as declared in this protocol is *blocking*. That is, the thread that sends a **lock** message is blocked from further execution until the lock is acquired (presumably because some other thread relinquishes its lock). Classes that adopt this protocol typically add methods that offer nonblocking alternatives.

These NEXTSTEP classes conform to the NSLocking protocol:

Class	Adds these features to the basic protocol
NSLock	A nonblocking lock method; the ability to limit the duration of a locking attempt.
NSConditionLock	The ability to postpone entry to a critical section until a condition is met.
NSRecursiveLock	The ability for a single thread to acquire a lock more than once without deadlocking.

The locking mechanism that these classes use causes a thread to sleep while waiting to acquire a lock rather than to poll the system constantly. Thus, lock objects can be used to lock time-consuming operations without causing system performance to degrade. See the class specifications for these classes for further information on their behavior and usage.

There is some performance cost in acquiring a lock, so it's best to avoid the overhead if possible. An application developer has control over whether the application will execute with multiple threads, so it's clear when locking is appropriate. A library developer doesn't necessarily know whether library code will execute in a multithreaded environment. In this case, it's best to test whether the code is executing in a multithreaded environment before attempting to acquire a lock. The following example illustrates how this is done.

Assume your application uses a Counter object to record various operations. Here's one design that lets the Counter know whether it is multithreaded:

```
+ (void)initialize
{
    if ([NSThread isMultiThreaded]) {
        [self taskNowMultiThreaded:nil];
    } else {
        [[NSNotificationCenter defaultCenter] addObserver:self
            selector:@selector(taskNowMultiThreaded:)
            name:NSWillBecomeMultiThreadedNotification object:nil];
    }
}
```

In the initialize method (which, by definition, is invoked before any Counter objects are created), the Counter class object first checks whether the application has already become multithreaded and if so invokes its own **taskNowMultiThreaded:** method. Otherwise, it registers as an observer of NSWillBecomeMultiThreadedNotification so that **taskNowMultiThreaded:** will be invoked when the application becomes multithreaded.

Counter's **taskNowMultiThreaded:** method creates a lock object that the threads use to coordinate their activities:

```
+ (void)taskNowMultiThreaded:(NSNotification *)event
{
    if (!theLock)
        theLock = [[NSLock alloc] init];
}
```

theLock, a static variable declared in the class implementation file, is assigned a value of **nil** until **taskNowMultiThreaded:** is invoked. Since messages sent to **nil** are permitted and have no effect, code within Counter that acts on shared data can be written like this:

```
[theLock lock];
/* Operate on shared data */
[theLock unlock];
```

With this approach, the overhead associated with lock operations is only incurred if the application is multithreaded. This code, however, raises another issue. What happens if one of the statements between the **lock** and **unlock** messages cause the application to become multithreaded? Then the **unlock** message wouldn't be paired with the preceding **lock**.

In normal usage, locking and unlocking messages are paired. However, as in the example above, it might be convenient to unlock a lock object that hasn't yet been locked. This is permitted with two restrictions. First, you can send an unpaired unlocking message to a lock object as long as the object has never before been locked. Second, of the NEXTSTEP classes that conform to the NSLocking protocol, only NSConditionLock and NSLock allow an unpaired unlocking message. NSRecursiveLock requires locking and unlocking messages to be paired.

NEXTSTEP's locking classes are designed to work in a well-behaved, multithreaded environment: The protection they offer can be subverted by the use of signal handlers. A signal handler can interrupt a thread, execute code that affects shared data, and then let the thread resume without alerting the thread that the application has, in effect, become multithreaded. For this reason, it's recommended that you don't use signal handlers in multithreaded NEXTSTEP applications.

nstance Methods

lock

– (void)**lock**

Attempts to acquire a lock. This method blocks a thread's execution until the lock can be acquired.

An application protects a critical section of code by requiring a thread to acquire a lock before executing the code. Once the critical section is past, the thread relinquishes the lock by invoking **unlock**.

unlock

– (void)**unlock**

Relinquishes a previously acquired lock.

NSMutableCopying

Adopted By: Various OpenStep classes

Declared In: Foundation/NSObject.h

Protocol Description

The NSMutableCopying protocol declares a method for providing mutable copies of an object. Only classes that define an "immutable vs. mutable" distinction should adopt this protocol. Classes that don't define such a distinction should adopt NSCopying instead.

NSMutableCopying declares one method, **mutableCopyWithZone:**, but mutable copying is commonly invoked with the convenience method **mutableCopy**. The **mutableCopy** method is defined for all NSObjects and simply invokes **mutableCopyWithZone:** with the default zone.

See the NSCopying protocol for details on implementing copying behavior.

Instance Methods

mutableCopyWithZone:

– **mutableCopyWithZone:**(NSZone *)*zone*

Returns a new instance that's a mutable copy of the receiver. Memory for the new instance is allocated from *zone*, which may be NULL. If *zone* is NULL, the new instance is allocated from the default zone, which is returned by **NSDefaultMallocZone()**. The returned object is implicitly retained by the sender, who is responsible for releasing it. The copy returned is mutable whether the original is mutable or not.

See also: – **copyWithZone:** (NSCopying protocol), – **mutableCopy** (NSObject)

NSObjCTypeSerializationCallBack

Adopted By: No OpenStep classes

Declared In: Foundation/NSSerialization.h

Protocol Description

An object conforms to the NSObjCTypeSerializationCallBack protocol so that it can intervene in the serialization and deserialization process. The primary purpose of this protocol is to allow for the serialization of objects and other data types that aren't directly supported by OpenStep's serialization facility. (See the NSSerializer class specification for information on serialization.)

NSMutableData declares the method that's used to begin the serialization process:

```
- (void)serializeDataAt:(const void *)data
        ofObjCType:(const char *)type
        context:(id <NSObjCTypeSerializationCallBack>)callback
```

This method can serialize all standard Objective C types (**int**, **float**, character strings, and so on) except for objects, **union**, and **void ***. If, during the serialization process, an object is encountered, the object passed as the callback argument above is asked to provide the serialization.

Suppose that the type being serialized is a structure of this description:

```
struct stockRecord {
    NSString *stockName;
    float value;
};
```

The Objective C type code for this structure is {@f}, so the serialization process begins with this message: (Assume that **theData** is the NSMutableData object that's doing the serialization and **helper** is an object that conforms to the NSObjCTypeSerializationCallBack protocol.)

```
struct stockRecord aRecord = {@"aCompany", 34.7};

[theData serializeDataAt:&aRecord ofObjCType:"{@f}" context:helper];
```

Since the first field of the structure is an unsupported type, the helper object is sent a **serializeObjectAt: ofObjCType:intoData:** message, letting it serialize the object. **helper** might implement the method in this way:

```
- (void)serializeObjectAt:(id *)objectPtr
        ofObjCType:(const char *)type
        intoData:(NSMutableData *)theMutableData
{
    NSString *nameObject;
    char *companyName

    nameObject = *objectPtr;
    companyName = [nameObject cString];

    [theData serializeDataAt:&companyName ofObjCType:@encode(typeof(companyName))
            context:nil];
}
```

The callback object is free to serialize the target object as it wishes. In this case, **helper** simply extracts the company name from the NSString object and then has that character string serialized. Once this callback method finishes executing, the original method (**serializeDataAt:ofObjCType:context:**) resumes execution and serializes the second field of the structure. Since this second field contains a supported type (**float**), the callback method is not invoked again.

Deserialization follows a similar pattern, except in this case NSData declares the central method **deserializeDataAt:ofObjCType:atCursor:context:**. The deserialization of the example structure starts with a message to the NSData object that contains the serialized data:

```
(unsigned *)cursor = 0;

[theData deserializeDataAt:&aRecord ofObjCType:"{@f}" cursor:&cursor
        context:helper];
```

(The cursor argument is a pointer to zero since we're starting at the beginning of the data in the NSData object.)

When this method is invoked, the callback object receives a **deserializeObjectAt:ofObjCType:fromData: atCursor:** message, as declared in this protocol. The callback object can then reestablish the first field of the structure. For example, **helper** might implement the method in this way:

```
- (void) deserializeObjectAt:(id *)objectPtr
        ofObjCType:(const char *)type
        fromData:(NSData *)data
        atCursor:(unsigned *)cursor
{
    char *companyName;

    [theData deserializeDataAt:&companyName ofObjCType:"*" atCursor:cursor
            context:nil];
    *objectPtr = [[NSString stringWithCString:companyName] retain];
}
```

nstance Methods

deserializeObjectAt:ofObjCType:fromData:atCursor:

– (void)**deserializeObjectAt:**(id *)*object*
ofObjCType:(const char *)*type*
fromData:(NSData *)*data*
atCursor:(unsigned *)*cursor*

The implementor of this method decodes the referenced *object* (which should always be of *type* "@") located at the *cursor* position in the *data* object. The decoded object is not autoreleased.

See also: – **deserializeDataAt:ofObjCType:atCursor:context:** (NSData)

serializeObjectAt:ofObjCType:intoData:

– (void)**serializeObjectAt:**(id *)*object*
ofObjCType:(const char *)*type*
intoData:(NSMutableData *)*data*

The implementor of this method encodes the referenced *object* (which should always be of *type* "@") in the *data* object.

See also: – **serializeDataAt:ofObjCType:context:** (NSMutableData)

NSObject

Adopted By: NSObject

Declared In: Foundation/NSObject.h

Protocol Description

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

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

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

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

Method Types

Identifying classes

– class
– superclass

Identifying and comparing objects

– isEqual:
– hash
– self

Determining allocation zones

– zone

Managing reference counts

– retain
– release
– autorelease
– retainCount

Testing class functionality

– respondsToSelector:

Testing inheritance relationships

– isKindOfClass:
– isMemberOfClass:

Testing protocol conformance

– conformsToProtocol:

Describing objects

– description

Sending messages

– performSelector:
– performSelector:withObject:
– performSelector:withObject:withObject:

Identifying proxies

– isProxy

nstance Methods

autorelease

– (id)**autorelease**

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

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

class

– (Class)**class**

Returns the class object for the receiver's class.

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

conformsToProtocol:

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

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

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

description

– (NSString *)**description**

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

hash

– (unsigned)**hash**

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

If a mutable object is added to a collection that uses hash values to determine the object's position in the collection, the value returned by the **hash** method of the object must not change while the object is in the collection. To accomplish this, either the **hash** method must not rely on any of the object's internal state information or you must make sure that the object's internal state information does not change while the object is in the collection. (Note that it can be difficult to know whether or not a given object is in a collection.)

isEqual:

– (BOOL)**isEqual:**(id)*anObject*

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

isKindOfClass:

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

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

```
NSMutableData *myData = [NSMutableData dataWithCapacity:30];
id anArchiver = [[NSArchiver alloc] initForWritingWithMutableData:myData];
if ( [anArchiver isKindOfClass:[NSCoder class]] )
    ...
```

Because instance methods defined in the root class can be sent to class objects, when the receiver is a class object, this method returns YES if *aClass* is NSObject, NO otherwise.

See also: – **isMemberOfClass:**

isMemberOfClass:

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

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

```
NSMutableData *myData = [NSMutableData dataWithCapacity:30];
id anArchiver = [[NSArchiver alloc] initForWritingWithMutableData:myData];
if ([anArchiver isMemberOfClass:[NSCoder class]])
    ...
```

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

See also: – **isKindOfClass:**

isProxy

– (BOOL)**isProxy**

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

performSelector:

– (id)**performSelector:**(SEL)*aSelector*

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

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

However, the **performSelector:** 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 performSelector:myMethod];
```

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

See also: – **performSelector:withObject:**, – **performSelector:withObject:withObject:**

performSelector:withObject:

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

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

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

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

performSelector:withObject:withObject:

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

Sends the receiver an *aSelector* message with *anObject* and *anotherObject* as arguments. If *aSelector* is NULL, an NSInvalidArgumentException is raised. This method is the same as **performSelector:** except

that you can supply two arguments for *aSelector*. *aSelector* should identify a method that can take two arguments of type **id**. For methods with other argument types and return values use NSInvocation.

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

release

– (oneway void)**release**

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

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

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

See also: – **retainCount**

respondsToSelector:

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

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

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

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

retain

– (id)**retain**

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

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

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

```
NSString *systemApps = [[NSString
    stringWithCString:"/System/Applications"] retain];
```

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

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

retainCount

– (unsigned)**retainCount**

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

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

self

– (id)**self**

Returns the receiver.

See also: – **class**

superclass

– (Class)**superclass**

Returns the class object for the receiver's superclass.

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

zone

– (NSZone *)**zone**

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

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

NSCoding

Adopted By: Various OpenStep classes

Declared In: Foundation/NSObject.h

Protocol Description

The NSCoding protocol declares the two methods that a class must implement so that instances of that class can be encoded and decoded. This capability provides the basis for archiving (where objects and other structures are stored on disk) and distribution (where objects are copied to different address spaces). See the NSCoder and NSArchiver class specifications for an introduction to coding.

In keeping with object-oriented design principles, an object being encoded or decoded is responsible for encoding and decoding its instance variables. A coder instructs the object to do so by invoking **encodeWithCoder:** or **initWithCoder:**. **encodeWithCoder:** instructs the object to encode its instance variables to the coder provided; an object can receive this method any number of times. **initWithCoder:** instructs the object to initialize itself from data in the coder provided; as such, it replaces any other initialization method and is only sent once per object. Any object class that should be codable must adopt the NSCoding protocol and implement its methods.

When an object receives an **encodeWithCoder:** message, it should encode all of its vital instance variables, after sending a message to **super** if its superclass also conforms to the NSCoding protocol. An object doesn't have to encode all of its instance variables. Some values may not be important to reestablish and others may be derivable from related state upon decoding. Other instance variables should be encoded only under certain conditions (for example, with **encodeConditionalObject:**, as described in the NSArchiver class specification).

For example, suppose you were creating a fictitious MapView class that displays a legend and a map at various magnifications. The MapView class defines several instance variables, including the name of the map and the current magnification. The MapView class also contains instance variables for several related views. The **encodeWithCoder:** method of MapView might look like the following:

```
- (void)encodeWithCoder:(NSCoder *)coder
{
    [super encodeWithCoder:coder];
    [coder encodeValueOfObjCType:@encode(char *) at:mapName];
    [coder encodeValueOfObjCType:@encode(unsigned int) at:&magnification];
    [coder encodeObject:legendView];
    [coder encodeConditionalObject:auxiliaryView];
    return;
}
```

This example assumes that the superclass of MapView also supports the NSCoding protocol. If the superclass of your class does not support NSCoding, you should omit the line that invokes super's **encodeWithCoder:** method.

encodeValueOfObjCType:at: and **encodeObject:** are coder methods that you can use to encode instance variables of your class. You can use these and other methods of the coder to encode id's, scalars, C arrays, structs, strings, and pointers to any of these types. The coder also defines corresponding methods for decoding values. See the NSCoder, NSArchiver, and NSUnarchiver class specifications for a list of methods.

The **@encode()** compiler directive generates an Objective-C type code from a type expression. See *Object-Oriented Programming and the Objective-C Language* for more information.

Similarly, in **initWithCoder:** the object should first send a message to **super** (if appropriate) to initialize inherited instance variables, and then it should decode and initialize its own. MapView's implementation of **initWithCoder:** might look like this:

```
- (id)initWithCoder:(NSCoder *)coder
{
    self = [super initWithCoder:coder];
    [coder decodeValueOfObjCType:@encode(char *) at:mapName];
    [coder decodeValueOfObjCType:@encode(unsigned int) at:&magnification];
    legendView = [[coder decodeObject] retain];
    auxiliaryView = [[coder decodeObject] retain];
    return self;
}
```

Note the assignment of the return value of **initWithCoder:** to **self** in the example above. This is done in the subclass because the superclass, in its implementation of **initWithCoder:**, may decide to return a object other than itself. If the superclass of your class does not support NSCoding, you should invoke super's designated initializer instead of **initWithCoder:**.

Making Substitutions During Coding

During encoding or decoding a coder object invokes methods that allow the object being coded to substitute a replacement class or instance for itself. This allows archives to be shared among implementations with different class hierarchies or simply different versions of a class (for example, class clusters take advantage of this feature). It also allows classes that should maintain unique instances to enforce this policy on decoding (for example, there need only be a single NSFont instance for a given typeface and size).

Substitution methods are declared by NSObject, and come in two flavors: generic and specialized. The generic methods are these:

Method	Typical Use
classForCoder	Allows an object, before being encoded, to substitute a class other than its own. For example, the private subclasses of a class cluster substitute the name of their public superclass when being archived.
replacementObjectForCoder:	Allows an object, before being encoded, to substitute another instance in its place.
awakeAfterUsingCoder:	Allows an object, after being decoded, to substitute another object for itself. For example, an object that represents a font might, upon being decoded, release itself and return an existing object having the same font description as itself. In this way, redundant objects can be eliminated.

The specialized substitution methods are analogous to **classForCoder** and **replacementObjectForCoder:** , but they're designed for (and invoked by) a specific, concrete coder subclass. NSArchiver invokes **classForArchiver:** and **replacementObjectForArchiver:**, while NSPortCoder invokes **classForPortCoder** and **replacementObjectForPortCoder:**. (There isn't a specialized version of **awakeAfterUsingCoder:**.) By implementing these specialized methods, your class can base its coding behavior on the specific coder class being used. For more information on these methods, see their method descriptions in the NSObject class specification, as well as the class description in the NSPortCoder class specification.

Method Types

Encoding and decoding objects

– encodeWithCoder:
– initWithCoder:

Instance Methods

encodeWithCoder:

– (void)**encodeWithCoder:**(NSCoder *)*encoder*

Encodes the receiver using *encoder*.

initWithCoder:

– (id)**initWithCoder:**(NSCoder *)*decoder*

Initializes a newly allocated instance from data in *decoder*. Returns **self**.

NSCopying

Adopted By: Various OpenStep classes

Declared In: Foundation/NSObject.h

Protocol Description

The NSCopying protocol declares a method for providing functional copies of an object. The exact meaning of "copy" can vary from class to class, but a copy must be a functionally independent object with values identical to the original at the time the copy was made. A copy produced with NSCopying is implicitly retained by the sender, who is responsible for releasing it.

NSCopying declares one method, **copyWithZone:**, but copying is commonly invoked with the convenience method **copy**. The **copy** method is defined for all NSObjects and simply invokes **copyWithZone:** with the default zone.

Using NSCopying

NSCopying is frequently used to copy *value* objects—objects that represent attributes. C-type variables can usually be substituted for value objects, but value objects have the advantage of encapsulating convenient utilities for common manipulations. For example, NSString objects are used instead of character pointers because they encapsulate encoding and storage. Despite NSString functionality, the role played by NSStrings parallels the role played by character pointers.

When value objects are passed as method arguments or returned from a method, it is common to use a copy instead of the object itself. For example, consider the following method for assigning a string to an object's **name** instance variable.

```
- (void)setName:(NSString *)aName
{
    [name autorelease];
    name = [aName copy];
}
```

Storing a copy of **aName** has the effect of producing an object that's independent of the original, but has the same contents. Subsequent changes to the copy don't affect the original, and changes to the original don't affect the copy. Similarly, it is common to return a copy of an instance variable instead of the instance variable itself. For example, this method returns a copy of the **name** instance variable:

```
- (NSString *)name
{
    return [[name copy] autorelease];
}
```

Implementing NSCopying

There are two basic approaches to creating copies. You can use **alloc** and **init...**, or you can use **NSCopyObject()**. To choose the one that's right for your class, you need to consider the following questions:

- "What kind of copying—deep or shallow—does your class need?"
- "Does your class's superclass implement NSCopying?"
- "Are you familiar with the implementations of your class's superclasses?"

These areas are described in the following sections.

What kind of copying—deep or shallow—does your class need?

Generally, copying an object involves creating a new instance and initializing it with the values in the original object. Copying the values for non-pointer instance variables, such as booleans, integers, and floating points, is straightforward. When copying pointer instance variables there are two approaches. One approach, called a *shallow copy*, copies the pointer value from the original object into the copy. Thus, the original and the copy share referenced data. The other approach, called a *deep copy*, duplicates the data referenced by the pointer and assigns it to the copy's instance variable.

The implementation of an instance variable's set method should reflect the kind of copying you need to use. You should deeply copy the instance variable if the corresponding set method copies the new value as in this method:

```
- (void)setMyVariable:(id)newValue
{
    [myVariable autorelease];
    myVariable = [newValue copy];
}
```

You should shallowly copy the instance variable if the corresponding set method retains the new value as in this method:

```
- (void)setMyVariable:(id)newValue
{
    [myVariable autorelease];
    myVariable = [newValue retain];
}
```

Similarly, you should shallowly copy the instance variable if its set method simply assigns the new value to the instance variable without copying or retaining it as in this method:

```
- (void)setMyVariable:(id)newValue
{
    myVariable = newValue;
}
```

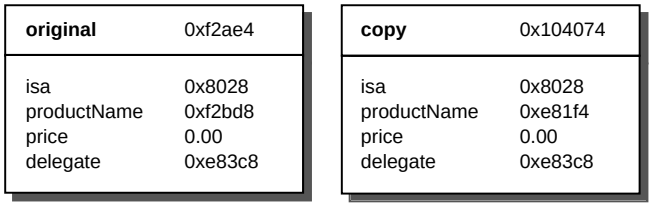
To produce a copy of an object that's truly independent of the original, the entire object must be deeply copied. Every instance variable must be duplicated. If the instance variables themselves have instance variables, those too must be duplicated, and so on. In many cases, a mixed approach is more useful. Pointer instance variables that can be thought of as data containers are generally deeply copied, while more sophisticated instance variables like delegates are shallowly copied.

For example, a Product class adopts NSCopying. Product instances have a name, a price, and a delegate as declared in this interface.

```
@interface Product : NSObject <NSCopying>
{
    NSString *productName;
    float price;
    id delegate;
}

@end
```

Copying a Product instance produces a deep copy of **productName** because it represents a flat data value. On the other hand, the **delegate** instance variable is a more complex object capable of functioning properly for both Products. The copy and the original should therefore share the delegate. The following figure represents the images of a Product instance and a copy in memory.



The different pointer values for **productName** illustrate that the original and the copy each have their own **productName** string object. The pointer values for **delegate** are the same, indicating that the two product objects share the same object as their delegate.

Does your class's superclass implement NSCopying?

If the superclass does not implement NSCopying, your class's implementation will have to copy the instance variables it inherits as well as those declared in your class. Generally, the safest way to do this is by using **alloc**, **init...**, and set methods. On the other hand, if your class inherits NSCopying behavior, its implementation only has to copy instance variables declared in your class. It invokes the superclass's implementation to copy inherited instance variables.

Are you familiar with the implementations of your class's superclasses?

If your class inherits NSCopying behavior, how you handle the new instance variables in **copyWithZone:** depends on your familiarity with the superclass's implementation. There are essentially two ways to make a copy of an object, using **alloc** and **init...** or using the function **NSCopyObject()**. If the superclass used or might have used **NSCopyObject()**, you must handle instance variables differently than you would otherwise.

Using the alloc, init... Approach

If a class does not inherit NSCopying behavior, you should implement **copyWithZone:** using **alloc**, **init...**, and set methods. For example, an implementation of **copyWithZone:** for the Product class described above might be implemented in the following way:

```
- (id)copyWithZone:(NSZone *)zone
{
    Product *copy = [[Product alloc]
           initWithProductName:[self productName]
           price:[self price]];
    [copy setDelegate:[self delegate]];

    return copy;
}
```

Because implementation details associated with inherited instance variables are encapsulated in the superclass, it is generally better to implement NSCopying with the **alloc**, **init...** approach. Doing so uses policy implemented in set methods to determine the kind of copying needed of instance variables.

Using NSCopyObject()

When a class inherits NSCopying behavior, you must consider the possibility that the superclass's implementation uses **NSCopyObject()**. **NSCopyObject()** creates an exact shallow copy of an object by copying instance variable values but not the data they point to. For example, NSCell's implementation of **copyWithZone:** could be defined in the following way.

```
- (id)copyWithZone:(NSZone *)zone
{
    NSCell *cellCopy = NSCopyObject(self, 0, zone);
    /* Assume that other initialization takes place here. */

    cellCopy->image = nil;
    [cellCopy setImage:[self image]];

    return cellCopy;
}
```

In the implementation above, **NSCopyObject()** creates an exact shallow copy of the original cell. This behavior is desirable for copying instance variables that aren't pointers or are pointers to non-retained data that is shallowly copied. Pointer instance variables for retained objects need additional treatment.

In the **copyWithZone:** example above, **image** is a pointer to a retained object. The policy to retain the image is reflected in the following implementation of the **setImage:** accessor method.

```
- (void)setImage:(NSImage *)anImage
{
    [image autorelease];
    image = [anImage retain];
}
```

Notice that **setImage:** autoreleases **image** before it reassigns it. If the above implementation of **copyWithZone:** hadn't explicitly set the copy's **image** instance variable to **nil** before invoking **setImage:**, the image referenced by the copy and the original would be released without a corresponding retain.

Even though **image** points to the right object, it is conceptually uninitialized. Unlike the instance variables that are created with **alloc** and **init...**, these uninitialized variables aren't **nil**-valued. You should explicitly assign initial values to these variables before using them. In this case, **cellCopy**'s **image** instance variable is set to **nil**, then it is set using the **setImage:** method.

The effects of **NSCopyObject()** extend to a subclass's implementation. For example, an implementation of NSSliderCell could copy a new **titleCell** instance variable in the following way.

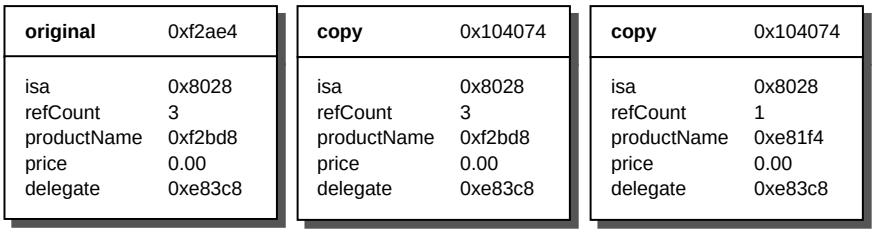
```
- (id)copyWithZone:(NSZone *)zone
{
    NSSliderCell *cellCopy = [super copyWithZone:zone];
    /* Assume that other initialization takes place here. */

    cellCopy->titleCell = nil;
    [cellCopy setTitleCell:[self titleCell]];

    return cellCopy;
}
```

The superclass's **copyWithZone:** method is invoked to copy inherited instance variables. When you invoke a superclass's **copyWithZone:** method, assume that new object instance variables are uninitialized if there's any chance that the superclass implementation uses **NSCopyObject()**. Explicitly assign a value to them before using them. In this example, **titleCell** is explicitly set to **nil** before **setTitleCell:** is invoked.

The implementation of an object's retain count is another consideration when using **NSCopyObject()**. If an object stores its retain count in an instance variable, the implementation of **copyWithZone:** must correctly initialize the copy's retain count. The following figure illustrates the process.



The copy produced by **NSCopyObject**

The copy after unitialized instance variables are assigned in **copyWithZone:**

The first object represents a Product instance in memory. The value in **refCount** indicates that the instance has been retained three times. The second object is a copy of the Product instance produced with **NSCopyObject()**. Its **refCount** value matches the original. The third object represents the copy returned from **copyWithZone:** after **refCount** is correctly initialized. After **copyWithZone:** creates the copy with **NSCopyObject()**, it assigns the value 1 to the **refCount** instance variable. The sender of **copyWithZone:** implicitly retains the copy and is responsible for releasing it.

NSCopying and Immutable Classes

Where the concept “immutable vs. mutable” applies to an object, NSCopying produces immutable copies whether the original is immutable or not. See the NSMutableCopying protocol for details on making mutable copies.

Immutable classes can implement NSCopying very efficiently. Since immutable objects don’t change, there is no need to duplicate them. Instead, NSCopying can be implemented to **retain** the original. For example, **copyWithZone:** for an immutable string class can be implemented in the following way.

```
- (id)copyWithZone:(NSZone *)zone
{
    return [self retain];
}
```

Summary

- Implement NSCopying using **alloc** and **init...** in classes that don’t inherit **copyWithZone:**.
- Implement NSCopying by invoking the superclass’s **copyWithZone:** when NSCopying behavior is inherited. If the superclass implementation might use **NSCopyObject()**, make explicit assignments to pointer instance variables for retained objects.
- Implement NSCopying by retaining the original instead of creating a new copy when the class and its contents are immutable.

Instance Methods

copyWithZone:

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

Returns a new instance that's a copy of the receiver. Memory for the new instance is allocated from *zone*, which may be NULL. If *zone* is NULL, the new instance is allocated from the default zone, which is returned from **NSDefaultMallocZone()**. The returned object is implicitly retained by the sender, who is responsible for releasing it. The copy returned is immutable if the consideration "immutable vs. mutable" applies to the receiving object; otherwise the exact nature of the copy is determined by the class.

See also: – **mutableCopyWithZone:** (NSMutableCopying protocol), – **copy** (NSObject)

NSDecimalNumberBehaviors

Adopted By: NSDecimalNumberHandler

Declared In: Foundation/NSDecimalNumber.h

Protocol Description

The NSDecimalBehaviors protocol declares three methods that control the discretionary aspects of working with NSDecimalNumbers. The **scale** and **roundingMode** methods determine the precision of NSDecimalNumber's return values, and the way in which those values should be rounded to fit that precision. The **exceptionDuringOperation:error:leftOperand:rightOperand:** determines the way in which an NSDecimalNumber should handle different calculation errors.

For an example of a class that adopts the NSDecimalBehaviors protocol, see the specification for NSDecimalNumberHandler.

Method Types

Rounding

– roundingMode
– scale

Handling errors

– exceptionDuringOperation:error:leftOperand:rightOperand:

Instance Methods

exceptionDuringOperation:error:leftOperand:rightOperand:

– (NSDecimalNumber *)**exceptionDuringOperation:**(SEL)*method*
error:(NSCalculationError)*error*
leftOperand:(NSDecimalNumber *)*leftOperand*
rightOperand:(NSDecimalNumber *)*rightOperand*

Specifies what an NSDecimalNumber will do when, in the course of applying *method* to *leftOperand* and *rightOperand*, it encounters *error.*

There are four possible values for *error.* The first three have to do with limits on NSDecimalNumber's ability to represent decimal numbers. An NSDecimalNumber can represent any number that can be

expressed as *mantissa* **x** 10**exponent**, where *mantissa* is a decimal integer up to 38 digits long, and *exponent* is between -256 and 256. If these limits are exceeded, the NSDecimalNumber returns one of the following errors:

- NSCalculationLossOfPrecision. The number can't be represented in 38 significant digits.
- NSCalculationOverflow. The number is too large to represent.
- NSCalculationUnderflow. The number is too small to represent.

The last error is simpler:

- NSCalculationDivideByZero. The caller tried to divide by zero.

In implementing **exceptionDuringOperation:error:leftOperand:rightOperand**, you can handle each of these errors in several ways:

- Raise an exception. For an explantion of exceptions, see the NSException class description in the *Foundation Framework Reference*.
- Return **nil**. The calling *method* will return its value as though no error had occurred. If *error* is NSCalculationLossOfPrecision, *method* will return an imprecise value—that is, one constrained to 38 significant digits. If *error* is NSCalculationUnderflow or NSCalculationOverflow, *method* will return NSDecimalNumber's **notANumber**. You shouldn't return **nil** if *error* is NSDivideByZero.
- Correct the error and return a valid NSDecimalNumber. The calling method will use this as its own return value.

roundingMode

– (NSRoundingMode)**roundingMode**

Returns the way that NSDecimalNumber's **decimalNumberBy...** methods round their return values. There are four possible NSRoundingModes:

- NSRoundDown. The methods round their return values down.
- NSRoundUp. The methods round their return values up.
- NSRoundPlain. The methods round to the closest possible return value. When they are caught halfway between two positive numbers, they round up; when caught between two negative numbers, they round down.
- NSRoundBankers. The methods round to the closest possible return value. When they are caught halfway between two possibilities, they return the possibility whose last digit is even. In practice, this means that, over the long run, numbers will be rounded up as often as they are rounded down; there will be no systematic bias.

The rounding mode only matters if the **scale** method sets a limit on the precision of NSDecimalNumber return values. It has no effect if **scale** returns NSDecimalNoScale.

Assuming that **scale** returns 1, the NSRoundingMode has the following effects on various original values:

Original value	NSRoundDown	NSRoundUp	NSRoundPlain	NSRoundBankers
1.24	1.2	1.3	1.2	1.2
1.26	1.2	1.3	1.3	1.3
1.25	1.2	1.3	1.3	1.2
1.35	1.3	1.4	1.4	1.4
-1.35	-1.4	-1.3	-1.4	-1.4

scale

– (short)**scale**

Limits the precision of the values returned by NSDecimalNumber's **decimalNumberBy...** methods.

Specifically, **scale** returns the number of digits allowed after the decimal separator. If **scale** returns a negative value, it affects the digits *before* the decimal separator as well. If **scale** returns NSDecimalNoScale, the number of digits is unlimited.

Assuming that **roundingMode** returns NSRoundPlain, different values of **scale** have the following effects on the number 123.456:

Scale	Return value
NSDecimalNoScale	123.456
2	123.45
0	123
-2	100

NSLocking

Adopted By: NSConditionLock
NSLock
NSRecursiveLock

Declared In: Foundation/NSLock.h

Protocol Description

The NSLocking protocol declares the elementary methods adopted by classes that define lock objects. A lock object is used to coordinate the actions of multiple threads of execution within a single application. By using a lock object, an application can protect critical sections of code from being executed simultaneously by separate threads, thus protecting shared data and other shared resources from corruption.

For example, consider a multithreaded application in which each thread updates a shared counter. If two threads simultaneously fetch the current value into local storage, increment it, and then write the value back, the counter will be incremented only once, losing one thread's contribution. However, if the code that manipulates the shared data (the *critical section* of code) must be locked before being executed, only one thread at a time can perform the updating operation, and collisions are prevented.

A lock object is either locked or unlocked. You *acquire* a lock by sending the object a **lock** message, thus putting the object in the locked state. You *relinquish* a lock by sending an **unlock** message, and thus putting the object in the unlocked state. (The NEXTSTEP classes that adopt this protocol define additional ways to acquire and relinquish locks.)

The **lock** method as declared in this protocol is *blocking*. That is, the thread that sends a **lock** message is blocked from further execution until the lock is acquired (presumably because some other thread relinquishes its lock). Classes that adopt this protocol typically add methods that offer nonblocking alternatives.

These NEXTSTEP classes conform to the NSLocking protocol:

Class	Adds these features to the basic protocol
NSLock	A nonblocking lock method; the ability to limit the duration of a locking attempt.
NSConditionLock	The ability to postpone entry to a critical section until a condition is met.
NSRecursiveLock	The ability for a single thread to acquire a lock more than once without deadlocking.

The locking mechanism that these classes use causes a thread to sleep while waiting to acquire a lock rather than to poll the system constantly. Thus, lock objects can be used to lock time-consuming operations without causing system performance to degrade. See the class specifications for these classes for further information on their behavior and usage.

There is some performance cost in acquiring a lock, so it's best to avoid the overhead if possible. An application developer has control over whether the application will execute with multiple threads, so it's clear when locking is appropriate. A library developer doesn't necessarily know whether library code will execute in a multithreaded environment. In this case, it's best to test whether the code is executing in a multithreaded environment before attempting to acquire a lock. The following example illustrates how this is done.

Assume your application uses a Counter object to record various operations. Here's one design that lets the Counter know whether it is multithreaded:

```
+ (void)initialize
{
    if ([NSThread isMultiThreaded]) {
        [self taskNowMultiThreaded:nil];
    } else {
        [[NSNotificationCenter defaultCenter] addObserver:self
            selector:@selector(taskNowMultiThreaded:)
            name:NSWillBecomeMultiThreadedNotification object:nil];
    }
}
```

In the initialize method (which, by definition, is invoked before any Counter objects are created), the Counter class object first checks whether the application has already become multithreaded and if so invokes its own **taskNowMultiThreaded:** method. Otherwise, it registers as an observer of NSWillBecomeMultiThreadedNotification so that **taskNowMultiThreaded:** will be invoked when the application becomes multithreaded.

Counter's **taskNowMultiThreaded:** method creates a lock object that the threads use to coordinate their activities:

```
+ (void)taskNowMultiThreaded:(NSNotification *)event
{
    if (!theLock)
        theLock = [[NSLock alloc] init];
}
```

theLock, a static variable declared in the class implementation file, is assigned a value of **nil** until **taskNowMultiThreaded:** is invoked. Since messages sent to **nil** are permitted and have no effect, code within Counter that acts on shared data can be written like this:

```
[theLock lock];
/* Operate on shared data */
[theLock unlock];
```

With this approach, the overhead associated with lock operations is only incurred if the application is multithreaded. This code, however, raises another issue. What happens if one of the statements between the **lock** and **unlock** messages cause the application to become multithreaded? Then the **unlock** message wouldn't be paired with the preceding **lock**.

In normal usage, locking and unlocking messages are paired. However, as in the example above, it might be convenient to unlock a lock object that hasn't yet been locked. This is permitted with two restrictions. First, you can send an unpaired unlocking message to a lock object as long as the object has never before been locked. Second, of the NEXTSTEP classes that conform to the NSLocking protocol, only NSConditionLock and NSLock allow an unpaired unlocking message. NSRecursiveLock requires locking and unlocking messages to be paired.

NEXTSTEP's locking classes are designed to work in a well-behaved, multithreaded environment: The protection they offer can be subverted by the use of signal handlers. A signal handler can interrupt a thread, execute code that affects shared data, and then let the thread resume without alerting the thread that the application has, in effect, become multithreaded. For this reason, it's recommended that you don't use signal handlers in multithreaded NEXTSTEP applications.

Instance Methods

lock

– (void)**lock**

Attempts to acquire a lock. This method blocks a thread's execution until the lock can be acquired.

An application protects a critical section of code by requiring a thread to acquire a lock before executing the code. Once the critical section is past, the thread relinquishes the lock by invoking **unlock**.

unlock

– (void)**unlock**

Relinquishes a previously acquired lock.

NSMutableCopying

Adopted By: Various OpenStep classes

Declared In: Foundation/NSObject.h

Protocol Description

The NSMutableCopying protocol declares a method for providing mutable copies of an object. Only classes that define an “immutable vs. mutable” distinction should adopt this protocol. Classes that don’t define such a distinction should adopt NSCopying instead.

NSMutableCopying declares one method, **mutableCopyWithZone:**, but mutable copying is commonly invoked with the convenience method **mutableCopy**. The **mutableCopy** method is defined for all NSObjects and simply invokes **mutableCopyWithZone:** with the default zone.

See the NSCopying protocol for details on implementing copying behavior.

Instance Methods

mutableCopyWithZone:

– **mutableCopyWithZone:**(NSZone *)*zone*

Returns a new instance that’s a mutable copy of the receiver. Memory for the new instance is allocated from *zone*, which may be NULL. If *zone* is NULL, the new instance is allocated from the default zone, which is returned by **NSDefaultMallocZone()**. The returned object is implicitly retained by the sender, who is responsible for releasing it. The copy returned is mutable whether the original is mutable or not.

See also: – **copyWithZone:** (NSCopying protocol), – **mutableCopy** (NSObject)

NSObjCTypeSerializationCallBack

Adopted By: No OpenStep classes

Declared In: Foundation/NSSerialization.h

Protocol Description

An object conforms to the NSObjCTypeSerializationCallBack protocol so that it can intervene in the serialization and deserialization process. The primary purpose of this protocol is to allow for the serialization of objects and other data types that aren't directly supported by OpenStep's serialization facility. (See the NSSerializer class specification for information on serialization.)

NSMutableData declares the method that's used to begin the serialization process:

```
- (void)serializeDataAt:(const void *)data
       ofObjCType:(const char *)type
       context:(id <NSObjCTypeSerializationCallBack>)callback
```

This method can serialize all standard Objective C types (**int**, **float**, character strings, and so on) except for objects, **union**, and **void ***. If, during the serialization process, an object is encountered, the object passed as the callback argument above is asked to provide the serialization.

Suppose that the type being serialized is a structure of this description:

```
struct stockRecord {
    NSString *stockName;
    float value;
};
```

The Objective C type code for this structure is {@f}, so the serialization process begins with this message: (Assume that **theData** is the NSMutableData object that's doing the serialization and **helper** is an object that conforms to the NSObjCTypeSerializationCallBack protocol.)

```
struct stockRecord aRecord = {@"aCompany", 34.7};

[theData serializeDataAt:&aRecord ofObjCType:"{@f}" context:helper];
```

Since the first field of the structure is an unsupported type, the helper object is sent a **serializeObjectAt: ofObjCType:intoData:** message, letting it serialize the object. **helper** might implement the method in this way:

```
- (void)serializeObjectAt:(id *)objectPtr
        ofObjCType:(const char *)type
        intoData:(NSMutableData *)theMutableData
{
    NSString *nameObject;
    char *companyName

    nameObject = *objectPtr;
    companyName = [nameObject cString];

    [theData serializeDataAt:&companyName ofObjCType:@encode(typeof(companyName))
            context:nil];
}
```

The callback object is free to serialize the target object as it wishes. In this case, **helper** simply extracts the company name from the NSString object and then has that character string serialized. Once this callback method finishes executing, the original method (**serializeDataAt:ofObjCType:context:**) resumes execution and serializes the second field of the structure. Since this second field contains a supported type (**float**), the callback method is not invoked again.

Deserialization follows a similar pattern, except in this case NSData declares the central method **deserializeDataAt:ofObjCType:atCursor:context:**. The deserialization of the example structure starts with a message to the NSData object that contains the serialized data:

```
(unsigned *)cursor = 0;

[theData deserializeDataAt:&aRecord ofObjCType:"{@f}" cursor:&cursor
        context:helper];
```

(The cursor argument is a pointer to zero since we're starting at the beginning of the data in the NSData object.)

When this method is invoked, the callback object receives a **deserializeObjectAt:ofObjCType:fromData: atCursor:** message, as declared in this protocol. The callback object can then reestablish the first field of the structure. For example, **helper** might implement the method in this way:

```
- (void) deserializeObjectAt:(id *)objectPtr
        ofObjCType:(const char *)type
        fromData:(NSData *)data
        atCursor:(unsigned *)cursor
{
    char *companyName;

    [theData deserializeDataAt:&companyName ofObjCType:"*" atCursor:cursor
            context:nil];
    *objectPtr = [[NSString stringWithCString:companyName] retain];
}
```

Instance Methods

deserializeObjectAt:ofObjCType:fromData:atCursor:

– (void)**deserializeObjectAt:**(id *)*object*
ofObjCType:(const char *)*type*
fromData:(NSData *)*data*
atCursor:(unsigned *)*cursor*

The implementor of this method decodes the referenced *object* (which should always be of *type* "@") located at the *cursor* position in the *data* object. The decoded object is not autoreleased.

See also: – **deserializeDataAt:ofObjCType:atCursor:context:** (NSData)

serializeObjectAt:ofObjCType:intoData:

– (void)**serializeObjectAt:**(id *)*object*
ofObjCType:(const char *)*type*
intoData:(NSMutableData *)*data*

The implementor of this method encodes the referenced *object* (which should always be of *type* "@") in the *data* object.

See also: – **serializeDataAt:ofObjCType:context:** (NSMutableData)

NSObject

Adopted By: NSObject

Declared In: Foundation/NSObject.h

Protocol Description

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

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

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

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

Method Types

Identifying classes

– class
– superclass

Identifying and comparing objects

– isEqual:
– hash
– self

Determining allocation zones

– zone

Managing reference counts

– retain
– release
– autorelease
– retainCount

Testing class functionality

– respondsToSelector:

Testing inheritance relationships

– isKindOfClass:
– isMemberOfClass:

Testing protocol conformance

– conformsToProtocol:

Describing objects

– description

Sending messages

– performSelector:
– performSelector:withObject:
– performSelector:withObject:withObject:

Identifying proxies

– isProxy

Instance Methods

autorelease

– (id)**autorelease**

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

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

class

– (Class)**class**

Returns the class object for the receiver's class.

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

conformsToProtocol:

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

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

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

description

– (NSString *)**description**

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

hash

– (unsigned)**hash**

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

If a mutable object is added to a collection that uses hash values to determine the object's position in the collection, the value returned by the **hash** method of the object must not change while the object is in the collection. To accomplish this, either the **hash** method must not rely on any of the object's internal state information or you must make sure that the object's internal state information does not change while the object is in the collection. (Note that it can be difficult to know whether or not a given object is in a collection.)

isEqual:

– (BOOL)**isEqual:**(id)*anObject*

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

isKindOfClass:

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

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

```
NSMutableData *myData = [NSMutableData dataWithCapacity:30];
id anArchiver = [[NSArchiver alloc] initForWritingWithMutableData:myData];
if ( [anArchiver isKindOfClass:[NSCoder class]] )
    ...
```

Because instance methods defined in the root class can be sent to class objects, when the receiver is a class object, this method returns YES if *aClass* is NSObject, NO otherwise.

See also: – **isMemberOfClass:**

isMemberOfClass:

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

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

```
NSMutableData *myData = [NSMutableData dataWithCapacity:30];
id anArchiver = [[NSArchiver alloc] initForWritingWithMutableData:myData];
if ([anArchiver isMemberOfClass:[NSCoder class]])
    ...
```

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

See also: – **isKindOfClass:**

isProxy

– (BOOL)**isProxy**

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

performSelector:

– (id)**performSelector:**(SEL)*aSelector*

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

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

However, the **performSelector:** 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 performSelector:myMethod];
```

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

See also: – **performSelector:withObject:**, – **performSelector:withObject:withObject:**

performSelector:withObject:

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

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

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

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

performSelector:withObject:withObject:

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

Sends the receiver an *aSelector* message with *anObject* and *anotherObject* as arguments. If *aSelector* is NULL, an NSInvalidArgumentException is raised. This method is the same as **performSelector:** except

that you can supply two arguments for *aSelector*. *aSelector* should identify a method that can take two arguments of type **id**. For methods with other argument types and return values use NSInvocation.

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

release

– (oneway void)**release**

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

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

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

See also: – **retainCount**

respondsToSelector:

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

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

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

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

retain

– (id)**retain**

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

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

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

```
NSString *systemApps = [[NSString
    stringWithCString:"/System/Applications"] retain];
```

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

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

retainCount

– (unsigned)**retainCount**

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

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

self

– (id)**self**

Returns the receiver.

See also: – **class**

superclass

– (Class)**superclass**

Returns the class object for the receiver's superclass.

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

zone

– (NSZone *)**zone**

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

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

Defined Types

NSBTreeComparator

DECLARED IN Foundation/NSByteStore.h

SYNOPSIS

typedef int **NSBTreeComparator**(NSData *, NSData *, const void *);

NSComparisonResult

DECLARED IN Foundation/NSObject.h

SYNOPSIS

typedef enum _NSComparisonResult {
    NSOrderedAscending = -1,
    NSOrderedSame,
    NSOrderedDescending
} **NSComparisonResult**;

DESCRIPTION

An **NSComparisonResult** indicates how items in a request are ordered, from the first one given in a method invocation or function call to the last (that is, left-to-right in code).

NSHashTableCallBacks

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS

```
typedef struct {
        unsigned (*hash)(NSHashTable *table, const void *);
        BOOL (*isEqual)(NSHashTable *table, const void *, const void *);
        void (*retain)(NSHashTable *table, const void *);
        void (*release)(NSHashTable *table, void *);
        NSString *(*describe)(NSHashTable *table, const void *);
} NSHashTableCallBacks;
```

NSMapTableKeyCallBacks

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS

```
typedef struct {
        unsigned (*hash)(NSMapTable *table, const void *);
        BOOL (*isEqual)(NSMapTable *table, const void *, const void *);
        void (*retain)(NSMapTable *table, const void *);
        void (*release)(NSMapTable *table, void *);
        NSString *(*describe)(NSMapTable *table, const void *);
         const void *notAKeyMarker;
} NSMapTableKeyCallBacks;
```

NSMapTableValueCallBacks

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS

```
typedef struct {
        void (*retain)(NSMapTable *table, const void *);
        void (*release)(NSMapTable *table, void *);
        NSString *(*describe)(NSMapTable *table, const void *);
} NSMapTableValueCallBacks;
```

NSNotificationCoalescing

DECLARED IN Foundation/NSNotificationQueue.h

SYNOPSIS

```
typedef enum {
    NSNotificationNoCoalescing = 0,
    NSNotificationCoalescingOnName = 1,
    NSNotificationCoalescingOnSender = 2,
} NSNotificationCoalescing;
```

NSObjCValue

DECLARED IN Foundation/NSInvocation.h

SYNOPSIS

```
typedef struct {
    enum _NSObjCValueType type;
    union {
        char charValue;
        short shortValue;
        long longValue;
        long long longlongValue;
        float floatValue;
        double doubleValue;
        SEL selectorValue;
        id objectValue;
        void *pointerValue;
        void *structLocation;
        char *cStringLocation;
    } value;
} NSObjCValue;
```

NSPoint

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS
```
typedef struct _NSPoint {
    float x;
    float y;
} NSPoint;
```

NSPosixFileActivity

DECLARED IN Foundation/NSPosixFileDescriptor.h

SYNOPSIS
```
typedef enum {
        NSPosixNoActivity = 0,
        NSPosixReadableActivity = 1,
        NSPosixWritableActivity = 2,
        NSPosixExceptionalActivity = 4
} NSPosixFileActivity;
```

NSPostingStyle

DECLARED IN Foundation/NSNotificationQueue.h

SYNOPSIS
```
typedef enum {
        NSPostWhenIdle = 1,
        NSPostASAP = 2,
        NSPostNow = 3
} NSPostingStyle;
```

NSRange

DECLARED IN Foundation/NSRange.h

SYNOPSIS

```
typedef struct _NSRange {
    unsigned int location;
    unsigned int length;
} NSRange;
```

DESCRIPTION An **NSRange** describes a portion of a series—such as characters in a string or objects in an NSArray. Its **location** member gives the start index (0 is the first, as in C arrays), and its **length** member gives the number of items in the range (and can be zero).

NSRect

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS

```
typedef struct _NSRect {
    NSPoint origin;
    NSSize size;
} NSRect;
```

NSRectEdge

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS

```
typedef enum _NSRectEdge {
    NSMinXEdge = 0,
    NSMinYEdge = 1,
    NSMaxXEdge = 2,
    NSMaxYEdge = 3
} NSRectEdge;
```

DESCRIPTION This enumeration identifies the sides of a rectangle represented by an NSRect.

NSSize

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS

```
typedef struct _NSSize {
    float width;
    float height;
} NSSize;
```

DESCRIPTION The values for width and height should never be negative.

NSStringEncoding

DECLARED IN Foundation/NSString.h

SYNOPSIS typedef unsigned **NSStringEncoding**;

NSTimeInterval

DECLARED IN Foundation/NSDate.h

SYNOPSIS typedef double **NSTimeInterval**;

DESCRIPTION Always in seconds; yields sub-millisecond precision over a range of 10000 years.

NSUncaughtExceptionHandler

DECLARED IN Foundation/NSException.h

SYNOPSIS typedef volatile void **NSUncaughtExceptionHandler**(NSException ***exception**);

Enumerations

NSByteOrder

DECLARED IN Foundation/NSByteOrder.h

SYNOPSIS

```
enum NSByteOrder {
        NS_UnknownByteOrder,
        NS_LittleEndian,
        NS_BigEndian
};
```

DESCRIPTION

NSNotFound

DECLARED IN Foundation/NSObjCRuntime.h

SYNOPSIS

```
enum {
        NSNotFound = 0x7fffffff
};
```

DESCRIPTION A method or function return value of **NSNotFound** indicates that the item requested couldn't be found or doesn't exist. It's typically used by various methods and functions that search for items in serial data and return indices, such as characters in a string object or **id**s in an NSArray.

NSOpenStepUnicodeReservedBase

DECLARED IN Foundation/NSCharacterSet.h

SYNOPSIS

```
enum {
    NSOpenStepUnicodeReservedBase = 0xF400
};
```

DESCRIPTION

ObjC Types

DECLARED IN Foundation/NSInvocation.h

SYNOPSIS

```
enum _NSObjCValueType {
    NSObjCNoType = 0,
    NSObjCVoidType = 'v',
    NSObjCCharType = 'c',
    NSObjCShortType = 's',
    NSObjCLongType = 'l',
    NSObjCLonglongType = 'q',
    NSObjCFloatType = 'f',
    NSObjCDoubleType = 'd',
    NSObjCSelectorType = ':',
    NSObjCObjectType = '@',
    NSObjCStructType = '{',
    NSObjCPointerType = '^',
    NSObjCStringType = '*',
    NSObjCArrayType = '[',
    NSObjCUnionType = '(',
    NSObjCBitfield = 'b',
};
```

Search Types

DECLARED IN Foundation/NSString.h

SYNOPSIS

```
enum {
    NSCaseInsensitiveSearch = 1,
    NSLiteralSearch = 2,
    NSBackwardsSearch = 4,
    NSAnchoredSearch = 8
};
```

DESCRIPTION These values represent the options available to many of the string classes' searching and comparison methods. See the NSString class cluster description for details on the effects of these options.

String Encodings

DECLARED IN Foundation/NSString.h

SYNOPSIS

```
enum {
    NSASCIIStringEncoding = 1,
    NSNEXTSTEPStringEncoding = 2,
    NSJapaneseEUCStringEncoding = 3,
    NSUTF8StringEncoding = 4,
    NSISOLatin1StringEncoding = 5,
    NSSymbolStringEncoding = 6,
    NSNonLossyASCIIStringEncoding = 7,
    NSShiftJISStringEncoding = 8,
    NSISOLatin2StringEncoding = 9,
    NSUnicodeStringEncoding = 10,
    NSWindowsCP1251StringEncoding = 11,
    NSWindowsCP1252StringEncoding = 12,
    NSWindowsCP1253StringEncoding = 13,
    NSWindowsCP1254StringEncoding = 14,
    NSWindowsCP1250StringEncoding = 15,
    NSISO2022JPStringEncoding = 21
};
```

DESCRIPTION These values represent the various character encodings supported by the NSString classes. These encodings are documented more fully elsewhere; here are some short descriptions:

Encoding	Description
NSASCIIStringEncoding	Strict 7-bit ASCII encoding within 8-bit chars
NSNEXTSTEPStringEncoding	8-bit ASCII encoding with NEXTSTEP extensions (see Appendix C of the NEXTSTEP General Reference)
NSUnicodeStringEncoding	The canonical Unicode encoding for string objects
NSISOLatin1StringEncoding	8-bit ISO Latin 1 encoding
NSISOLatin2StringEncoding	8-bit ISO Latin 2 encoding
NSSymbolStringEncoding	8-bit Adobe Symbol encoding vector
NSJapaneseEUCStringEncoding	8-bit EUC encoding for Japanese text
NSShiftJISStringEncoding	8-bit Shift-JIS encoding for Japanese text
NSUTF8StringEncoding	An 8-bit representation of Unicode characters, suitable for transmission or storage by ASCII-based systems
NSNonLossyASCIIStringEncoding	7-bit verbose ASCII to represent all Unicode characters.
NSWindowsCP1250StringEncoding	Microsoft Windows codepage 1250; equivalent to WinLatin2
NSWindowsCP1251StringEncoding	Microsoft Windows codepage 1251, encoding Cyrillic characters; equivalent to AdobeStandardCyrillic font encoding
NSWindowsCP1252StringEncoding	Microsoft Windows codepage 1252; equivalent to WinLatin1
NSWindowsCP1253StringEncoding	Microsoft Windows codepage 1253, encoding Greek characters
NSWindowsCP1254StringEncoding	Microsoft Windows codepage 1254, encoding Turkish characters
NSISO2022JPStringEncoding	ISO 2022 Japanese encoding for electronic mail

Global Variables

Archiving Exception Names

DECLARED IN Foundation/NSArchiver.h

SYNOPSIS

extern NSString ***NSInconsistentArchiveException**;

Bundle Notification Names

DECLARED IN Foundation/NSBundle.h

SYNOPSIS

extern NSString ***NSBundleLoaded**;

Byte Store Exception Names

DECLARED IN Foundation/NSByteStore.h

SYNOPSIS

extern NSString ***NSByteStoreLockedException**;
extern NSString ***NSByteStoreVersionException**;
extern NSString ***NSBTreeStoreKeyTooLargeException**;
extern NSString ***NSByteStoreDamagedException**;

Connection Exception Names

DECLARED IN Foundation/NSConnection.h

SYNOPSIS extern NSString ***NSFailedAuthenticationException**;

Connection Notification Names

DECLARED IN Foundation/NSConnection.h

SYNOPSIS extern NSString ***NSConnectionDidDieNotification**;

File Attribute Keys

DECLARED IN Foundation/NSFileManager.h

SYNOPSIS
extern NSString ***NSFileType**;
extern NSString ***NSFileTypeDirectory**;
extern NSString ***NSFileTypeRegular**;
extern NSString ***NSFileTypeSymbolicLink**;
extern NSString ***NSFileTypeSocket**;
extern NSString ***NSFileTypeCharacterSpecial**;
extern NSString ***NSFileTypeBlockSpecial**;
extern NSString ***NSFileTypeUnknown**;
extern NSString ***NSFileSize**;
extern NSString ***NSFileModificationDate**;
extern NSString ***NSFileReferenceCount**;
extern NSString ***NSFileDeviceIdentifier**;
extern NSString ***NSFileOwnerAccountNumber**;
extern NSString ***NSFileGroupOwnerAccountNumber**;
extern NSString ***NSFilePosixPermissions**;
extern NSString ***NSFileSystemNumber**;
extern NSString ***NSFileSystemFileNumber**;

DESCRIPTION Keys to access the file attribute values contained in the NSDictionary returned from NSFileManager's **fileAttributesAtPath:follow:**. See the class specification for NSFileManager for details of usage. **NSFileDeviceIdentifier** is used to access the identifier of a remote device.

File Exception Names

DECLARED IN Foundation/NSPosix.h

SYNOPSIS extern NSString ***NSPosixFileOperationException**;

File System Attribute Keys

DECLARED IN Foundation/NSFileManager.h

SYNOPSIS extern NSString ***NSFileSystemSize**;
extern NSString ***NSFileSystemFreeSize**;
extern NSString ***NSFileSystemNodes**;
extern NSString ***NSFileSystemFreeNodes**;

General Exception Names

DECLARED IN Foundation/NSException.h

SYNOPSIS
extern NSString ***NSGenericException**;
extern NSString ***NSRangeException**;
extern NSString ***NSInvalidArgumentException**;
extern NSString ***NSInternalInconsistencyException**;
extern NSString ***NSMallocException**;
extern NSString ***NSObjectInaccessibleException**;
extern NSString ***NSObjectNotAvailableException**;
extern NSString ***NSDestinationInvalidException**;
extern NSString ***NSPortTimeoutException**;
extern NSString ***NSInvalidSendPortException**;
extern NSString ***NSInvalidReceivePortException**;
extern NSString ***NSPortSendException**;
extern NSString ***NSPortReceiveException**;
extern NSString ***NSOldStyleException**;
extern NSString ***NSInvalidSendPort**;
extern NSString ***NSInvalidReceivePort**;
extern NSString ***NSPortSendError**;
extern NSString ***NSPortReceiveError**;
extern NSString ***NSCharacterConversionException**;

Hash Table Call Backs

DECLARED IN Foundation/NSHashTable.h

SYNOPSIS
extern const NSHashTableCallBacks **NSIntHashCallBacks**;
extern const NSHashTableCallBacks **NSNonOwnedPointerHashCallBacks**;
extern const NSHashTableCallBacks **NSOwnedPointerHashCallBacks**;
extern const NSHashTableCallBacks **NSPointerToStructHashCallBacks**;
extern const NSHashTableCallBacks **NSObjectHashCallBacks**;
extern const NSHashTableCallBacks **NSNonRetainedObjectHashCallBacks**;;

DESCRIPTION **NSIntHashCallBacks** are for sets of pointer-sized quantities or samller (for example, ints, longs, or unichars).

NSNonOwnedPointerHashCallBacks are for sets of pointers, hashed by address.

NSOwnedPointerHashCallBacks are for sets of pointers, with transfer of ownership upon insertion.

NSPointerToStructHashCallBacks are for sets of pointers to structs, when the first field of the struct is int-sized.

NSObjectHashCallBacks are for sets of objects (similar to NSSet)

NSNonRetainedObjectHashCallBacks are for sets of objects, but without retaining/releasing.

Note that you can make your own call back by picking fields among the above call backs.

Language-Dependent Information

DECLARED IN Foundation/NSUserDefaults.h

SYNOPSIS

```
extern NSString *NSWeekDayNameArray;
extern NSString *NSShortWeekDayNameArray;
extern NSString *NSMonthNameArray;
extern NSString *NSShortMonthNameArray;
extern NSString *NSTimeFormatString;
extern NSString *NSDateFormatString;
extern NSString *NSTimeDateFormatString;
extern NSString *NSShortTimeDateFormatString;
extern NSString *NSCurrencySymbol;
extern NSString *NSDecimalSeparator;
extern NSString *NSThousandsSeparator;
extern NSString *NSInternationalCurrencyString;
extern NSString *NSCurrencyString;
extern NSString *NSDecimalDigits;
extern NSString *NSAMPMDesignation;
extern NSString *NSHourNameDesignations;
extern NSString *NSYearMonthWeekDesignations;
extern NSString *NSEarlierTimeDesignations;
extern NSString *NSLaterTimeDesignations;
extern NSString *NSThisDayDesignations;
extern NSString *NSNextDayDesignations;
extern NSString *NSNextNextDayDesignations;
extern NSString *NSPriorDayDesignations;
extern NSString *NSDateTimeOrdering;
```

Map Table Key Call Backs

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS
extern const NSMapTableKeyCallBacks **NSIntMapKeyCallBacks**;
extern const NSMapTableKeyCallBacks **NSNonOwnedPointerMapKeyCallBacks**;
extern const NSMapTableKeyCallBacks **NSNonOwnedPointerOrNullMapKeyCallBacks**;
extern const NSMapTableKeyCallBacks **NSOwnedPointerMapKeyCallBacks**;
extern const NSMapTableKeyCallBacks **NSObjectMapKeyCallBacks**;
extern const NSMapTableKeyCallBacks **NSNonRetainedObjectMapKeyCallBacks**;

DESCRIPTION **NSIntMapKeyCallBacks** are for keys that are pointer-sized quantities or smaller (for example, ints, longs, or unichars).

NSNonOwnedPointerMapKeyCallBacks are for keys that are pointers not freed.

NSNonOwnedPointerOrNullMapKeyCallBacks are for keys that are pointers not freed, or NULL.

NSOwnedPointerMapKeyCallBacks are for keys that are pointers, with transfer of ownership upon insertion.

NSObjectMapKeyCallBacks are for keys that are objects

NSNonRetainedObjectMapKeyCallBacks are for sets of objects, but without retaining/releasing.

Note that you can make your own call back by picking fields among the above call backs.

Map Table Value Call Backs

DECLARED IN Foundation/NSMapTable.h

SYNOPSIS
extern const NSMapTableValueCallBacks **NSIntMapValueCallBacks**;
extern const NSMapTableValueCallBacks **NSNonOwnedPointerMapValueCallBacks**;
extern const NSMapTableValueCallBacks **NSOwnedPointerMapValueCallBacks**;
extern const NSMapTableValueCallBacks **NSObjectMapValueCallBacks**;

DESCRIPTION **NSIntMapValueCallBacks** are for values that are pointer-sized quantities, such as ints.

NSNonOwnedPointerMapValueCallBacks are for values that are not owned pointers.

NSOwnedPointerMapValueCallBacks are for values that are owned pointers.

NSObjectMapValueCallBacks are for values that are objects.

Note that you can make your own call back by picking fields among the above call backs.

NSConnectionReplyMode

DECLARED IN **NSConnection.h**

SYNOPSIS extern NSString ***NSConnectionReplyMode**;

NSDefaultRunLoopMode

DECLARED IN **Foundation/NSRunLoop.h**

SYNOPSIS extern NSString ***NSDefaultRunLoopMode**;

Port Notification Names

DECLARED IN Foundation/NSPort.h

SYNOPSIS extern NSString ***NSPortDidBecomeInvalidNotification**;

PPL Notification Names

DECLARED IN Foundation/**NPPL.h**

SYNOPSIS
extern NSString ***NSPPLDidBecomeDirtyNotification**;
extern NSString ***NSPPLDidSaveNotification**;

Thread Notification Names

DECLARED IN Foundation/**NSThread.h**

SYNOPSIS
extern NSString ***NSBecomingMultiThreaded**;
extern NSString ***NSThreadExiting**;

User Defaults

DECLARED IN Foundation/NSUserDefaults.h

SYNOPSIS
extern NSString ***NSGlobalDomain**;
extern NSString ***NSArgumentDomain**;
extern NSString ***NSRegistrationDomain**;

Zero Constants

DECLARED IN Foundation/NSGeometry.h

SYNOPSIS
extern const NSPoint **NSZeroPoint**;
extern const NSSize **NSZeroSize**;
extern const NSRect **NSZeroRect**;