NSCoder

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

Class Description

NSCoder is an abstract class that declares the interface used by subclasses to take data from memory and code them into and out of some other format. 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: – encodeNXObject: – encodeObject: – encodePropertyList: – encodePoint: – encodeRect: – encodeRootObject: – encodeSize: – encodeValueOfObjCType:at: – encodeValuesOfObjCTypes:
Decoding data	– decodeArrayOfObjCType:count:at: – decodeBytesWithReturnedLength: – decodeDataObject – decodeNXObject – decodeObject – decodePropertyList – decodePoint – decodeRect – decodeSize – decodeValueOfObjCType:at: – decodeValuesOfObjCTypes:

Managing zones	– objectZone – setObjectZone:
Getting version information	– systemVersion – versionForClassName:

nstance 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 description of the **@encode()** compiler directive in *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.

decodeNXObject

– (Object *)**decodeNXObject**

Decodes and returns an object descended from the Object class of NEXTSTEP Release 3 or earlier. Clients do not need to retain the returned object.

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

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 description of the **@encode()** compiler directive in *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:**, – **encodeNXObject:,**
– **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:**.

See also: This method must be matched by a subsequent **decodeObject** message.– **encodeByCopyObject:,** – **replacementObjectForPortCoder:**

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:**, – **encodeNXObject:**, – **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:**

encodeNXObject:

– (void)**encodeNXObject:**(Object *)*nxobject*

Encodes *nxobject*, an object descended from the Object class of NEXTSTEP Release 3 or earlier. This method must be matched by a subsequent **decodeNXObject** message.

Objects encoded more than once by successive invocations of this method are written in their entirety on each invocation. Applied to an object graph, the behavior of this method effectively flattens the graph structure, creating a new copy of the object in place of a reference to the original. This differs from the

behavior of **encodeObject:**, which encodes an object once and replaces subsequent occurences of the object with a reference to the original.

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

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:**, – **encodeNXObject:**

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:**, – **encodeNXObject:**, **– 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