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 dynamic 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, respectively, NSArchiver and 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*.

Coder objects operate on values of any Objective-C type except **union** and **void ***. They 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 the 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 mutableData];
archiver = [[NSArchiver alloc] initForWritingWithMutableData:data];
[archiver encodeRootObject:myMapView];
result = [data writeToFile:@"/tmp/MapArchive" atomically:YES];
```

NSArchiver also provides a convenience method for archiving immediately 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, creating a graph of references that may contain cycles, objects that must be shared upon decoding, and inessential objects. NSCoder declares methods that allow a concrete subclass to manage these cases: **encodeRootObject:**, **encodeObject:**, and **encodeConditionalObject:**. As implemented by a subclass, **encodeRootObject:** should encode the given object along with any objects it contains references to, and so on recursively, keeping track of multiple references to each object to avoid redundancy. To allow one part of a graph to be encoded without the rest**,**

encodeConditionalObject:should encode an object only if it's unconditionally encoded elsewhere in the graph.

However, 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

NSCoder's abstract implementation is based on these methods: **encodeValueOfObjCType:at:**, **decodeValueOfObjCType:at:**, **encodeDataObject:**, **decodeDataObject:**, and **versionForClassName:**. To create a functional coder subclass, you must implement at least these methods. Other methods that can be overridden for more specialized behavior are:

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

See the individual method descriptions for more information, and 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.

In general, the object being coded is fully responsible for coding itself. A few classes, however, push responsibility back on the coder, whether 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 always asks its coder to handle its contents directly using the **encodeDataObject:** and **decodeDataObject:** methods described in this class specification. Similarly, an NSPort 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:
	– encodeConditionalObject: – encodeDataObject: – encodeObject: – encodePropertyList: – encodePoint: – encodeRect: – encodeRootObject: – encodeSize: – encodeValueOfObjCType:at: – encodeValuesOfObjCTypes:
Decoding data	– decodeArrayOfObjCType:count:at:
	– 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 values are decoded into a buffer beginning at *address*, which must be large enough to contain them all. *itemType* must contain exactly one type code. If you use this method to decode Objective-C objects, you are responsible for releasing them.

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: – **decodeValueOfObjCType:at:**, – **decodeValuesOfObjCTypes:**

decodeDataObject

– (NSData *)**decodeDataObject**

Must be overridden by subclasses to decode and return an NSData object. This method matches an **encodeDataObject:** message used during encoding.

decodeNXObject

– (Object *)**decodeNXObject**

Decodes and returns an object descended from the Object class of NEXTSTEP Release 3 or earlier.

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.

decodePoint

– (NSPoint)**decodePoint**

Decodes and returns a point structure that was previously encoded with **encodePoint:**.

decodePropertyList

– (id)**decodePropertyList**

Decodes a property list that was previously encoded with **encodePropertyList:**. See the NSPPL class specification for information on property lists.

decodeRect

– (NSRect)**decodeRect**

Decodes and returns a rectangle structure that was previously encoded with **encodeRect:**.

decodeSize

– (NSSize)**decodeSize**

Decodes and returns a size structure that was previously encoded with **encodeSize:**.

decodeValueOfObjCType:at:

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

Must be overridden by subclasses to decode a single value, whose Objective-C type is given by *valueType*. The value is decoded into a buffer beginning at *address*, which must be large enough to contain the value. *valueType* must contain exactly one type code. If you use this method to decode an Objective-C object, you are responsible for releasing it.

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

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: – **decodeArrayOfObjCType:count:at:**, – **decodeValuesOfObjCTypes:**

decodeValuesOfObjCTypes:

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

Decodes a series of values of differing Objective-C types, as given by *valueTypes*. The values are decoded into buffers given by pointer arguments following *valueTypes*, which must each be large enough to hold their respective values. *valueTypes* may contain any number of type codes, so long as each one has a corresponding buffer following. 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.

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:**, – **decodeValueOfObjCType: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 a buffer beginning at *address*. *itemType* must contain exactly one type code.

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:**

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:**, – **encodeObject:**, – **encodeNXObject:**

encodeConditionalObject:

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

Can be overridden by subclasses to conditionally encode *object*, preserving common references to that object. *object* should normally be encoded only if it's unconditionally encoded elsewhere (with any other **encode...Object:** method). NSCoder's implementation simply invokes **encodeObject:**.

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

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.

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

encodeObject:

– (void)**encodeObject:**(id)*object*

Encodes *object*, possibly only creating a reference if *object* was already encoded by the receiver. This method must be matched by a subsequent **decodeObject** message.

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

encodePoint:

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

Encodes *point*. 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.

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

encodeRect:

– (void)**encodeRect:**(NSRect)*rect*

Encodes *point*. 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*. 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 differing Objective-C types, as given by *valueTypes*. The values are encoded from the arguments following *valueTypes*. *valueTypes* may contain any number of type codes, so long as each one has a corresponding value following.

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

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()**.

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.

See also: – **objectZone**

systemVersion

– (unsigned int)**systemVersion**

Returns the system version currently in effect during encoding, or during decoding, the version that was in effect when the data was encoded.

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 *className* exists in the encoding.

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