

# NSCoder

Inherits From:	NSObject
Conforms To:	NSCoding NSObject
Declared In:	foundation/NSCoder.h

## Class Description

NSCoder is an abstract class that declares the interface used by subclasses to take Objective C objects from dynamic memory and code them into and out of some other format. 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 NSArchiver and NSUnarchiver class specifications for more information on archiving.

NSCoder operates on the basic C and Objective C types—**int**, **float**, **id**, and so on (but excluding **void \*** and **union**)—as well as on user-defined structures and pointers to these types.

NSCoder declares methods that a subclass can override if it wants:

- To encode or decode an object only under certain conditions, such as it being an intrinsic part of a larger structure (**encodeRootObject:** and **encodeConditionalObject:**).
- To allow decoded objects to be allocated from a specific memory zone (**setObjectZone:**).
- To allow system versioning (**systemVersion**)

## Encoding and Decoding Objects

In NEXTSTEP, coding is facilitated by methods declared in several places, most notably the NSCoder class, the NSObject class, and the NSCoding protocol.

The NSCoding protocol declares the two methods (**encodeWithCoder:** and **initWithCoder:**) that a class must implement so that objects of that class can be encoded and decoded. When an object receives an **encodeWithCoder:** message, it should send a message to **super** to encode inherited instance variables before it encodes the instance variables that it's class declares. For example, the fictitious MapView class, that displays a

legend and a map at various magnifications, might implement **encodeWithCoder:** like this:

```
- (void)encodeWithCoder:(NSCoder *)coder
{
    [super encodeWithCoder:coder];
    [coder encodeValuesOfObjCTypes:"si@", &mapName, &magnification,
        &legendView];
}
```

Objects are decoded in two steps. First, an object of the appropriate class is allocated and then it's sent an **initWithCoder:** message to allow it to initialize its instance variables. Again, the object should first send a message to **super** to initialize inherited instance variables, and then it should initialize its own. `MapView`'s implementation of this method looks like this:

```
- (id)initWithCoder:(NSCoder *)coder
{
    self = [super initWithCoder:coder];
    [coder decodeValuesOfObjCTypes:"si@", &mapName, &magnification,
        &legendView];
    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.

There are other methods that allow an object to customize its response to encoding or decoding. `NSObject` declares these methods:

Method	Typical Use
<code>classForCoder:</code>	Allows an object, when 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.
<code>replacementObjectForCoder:</code>	Allows an object, when being encoded, to substitute another object for itself. For example, an object might encode itself into an archive, but encode a proxy for itself if it's being encoded for distribution.
<code>awakeAfterUsingCoder:</code>	Allows an object, when 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.

See the `NSObject` class specification for more information.

## Instance Variables

None declared in this class.

## Method Types

Encoding Data	<ul style="list-style-type: none"> <li>– <code>encodeArrayOfObjCType:count:at:</code></li> <li>– <code>encodeBycopyObject:</code></li> <li>– <code>encodeConditionalObject:</code></li> <li>– <code>encodeDataObject:</code></li> <li>– <code>encodeNXObject:</code></li> <li>– <code>encodeObject:</code></li> <li>– <code>encodePropertyList:</code></li> <li>– <code>encodeRootObject:</code></li> <li>– <code>encodeValueOfObjCType:at:</code></li> <li>– <code>encodeValuesOfObjCTypes:</code></li> </ul>
Decoding Data	<ul style="list-style-type: none"> <li>– <code>decodeArrayOfObjCType:count:at:</code></li> <li>– <code>decodeDataObject</code></li> <li>– <code>decodeNXObject</code></li> <li>– <code>decodeObject</code></li> <li>– <code>decodePropertyList</code></li> </ul>

	– decodeValueOfObjCType:at:
	– decodeValuesOfObjCTypes:
Managing Zones	– objectZone
	– setObjectZone:
Getting a Version	– systemVersion
	– versionForClassName:

## Instance Methods

### **decodeArrayOfObjCType:count:at:**

– (void)**decodeArrayOfObjCType:(const char \*)type count:(unsigned)count at:(void \*)address**

Decodes data of Objective C types listed in *type* having *count* elements residing at *address*.

### **decodeDataObject**

– (NSData \*)**decodeDataObject**

Decodes and returns an NSData object.

### **decodeNXObject**

– (Object \*)**decodeNXObject**

Decodes and returns an object that descends from Object.

### **decodeObject**

– (id)**decodeObject**

Decodes an Objective C object.

## **decodePropertyList**

– (id)**decodePropertyList**

Decodes a property list (NSData, NSArray, NSDictionary, or NSString objects).

## **decodeValueOfObjCType:at:**

– (void)**decodeValueOfObjCType:(const char \*)type at:(void \*)address**

Decodes data of the specified Objective C *type* residing at *address*. You are responsible for releasing the resulting objects.

## **decodeValuesOfObjCTypes:**

– (void)**decodeValuesOfObjCTypes:(const char \*)types,...**

Decodes values corresponding to the Objective C types listed in *types* argument list. You are responsible for releasing the resulting objects.

## **encodeArrayOfObjCType:count:at:**

– (void)**encodeArrayOfObjCType:(const char \*)type count:(unsigned)count  
at:(const void \*)array**

Encodes data of Objective C types listed in *type* having *count* elements residing at address *array*.

## **encodeBycopyObject:**

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

Overridden by subclasses to encode the supplied Objective C object so that a copy rather than a proxy of *anObject* is created upon decoding. NSCoder's implementation simply invokes **encodeObject:**.

**encodeConditionalObject:**

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

Overridden by subclasses to conditionally encode the supplied Objective C object. The object should be encoded only if it is an intrinsic member of the larger data structure. NSCoder’s implementation simply invokes **encodeObject:**.

**encodeDataObject:**

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

Encodes the NSData object *data*.

**encodeNXObject:**

– (void)**encodeNXObject:(Object \*)***object*

Encodes an object that descends from Object.

**encodeObject:**

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

Encodes the supplied Objective C object.

**encodePropertyList:**

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

Encodes the supplied property list (NSData, NSArray, NSDictionary, or NSString objects).

**encodeRootObject:**

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

Overridden by subclasses to start encoding an interconnected group of Objective C objects, starting with *rootObject*. NSCoder’s implementation simply invokes **encodeObject:**.

### **encodeValueOfObjCType:**

– (void)**encodeValueOfObjCType:**(const char \*)*type* **at:**(const void \*)*address*

Encodes data of the specified Objective C *type* residing at *address*.

### **encodeValuesOfObjCTypes:**

– (void)**encodeValuesOfObjCTypes:**(const char \*)*types*,...

Encodes values corresponding to the Objective C types listed in *types* argument list.

### **objectZone**

– (NSZone \*)**objectZone**

Returns the memory zone used by decoded objects. For instances of NSCoder, this is the default memory zone, the one returned by **NSDefaultMallocZone()**.

### **setObjectZone:**

– (void)**setObjectZone:**(NSZone \*)*zone*

Sets the memory zone used by decoded objects. Instances of NSCoder always use the default memory zone, the one returned by **NSDefaultMallocZone()**, and so ignore this method.

### **systemVersion**

– (unsigned int)**systemVersion**

Returns the system version number as of the time the archive was created.

### **versionForClassName:**

– (unsigned int)**versionForClassName:**(NSString \*)*className*

Returns the version number of the class *className* as of the time it was archived.