
Archives and Serializations Programming Guide for Cocoa

Data Management: File Management



2009-08-18



Apple Inc.
© 2002, 2009 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, Cocoa, Mac, Mac OS, and Objective-C are trademarks of Apple Inc., registered in the United States and other countries.

Java and all Java-based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS

PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction 7

Organization of This Document 7

Object Graphs 9

Archives 10

Serializations 10

Archives 11

Coders 11

 Root Object 12

 Conditional Objects 12

Keyed Archives 13

 Naming Values 13

 Return Values for Missing Keys 14

 Type Coercions 14

 Class Versioning 14

 Root Object 14

 Delegates 15

 Non-Keyed Coding Methods 15

Sequential Archives 15

Creating an Archive 17

Decoding an Archive 19

Encoding and Decoding Objects 21

Encoding an Object 21

Decoding an Object 22

Performance Considerations 22

Making Substitutions During Coding 23

When to Retain a Decoded Object 24

Restricting Coder Support 24

Encoding and Decoding C Data Types 25

Pointers 25

Arrays of Simple Types 25

Arrays of Objects 26

Structures and Bit Fields 26
More Complex Data Types 26

Forward and Backward Compatibility for Keyed Archives 27

Benefits of Keyed Archiving 27
General Tips on Maintaining Compatibility 27
Adding New Values to Keys 28
Adding New Keys 28
Removing or Retiring Keys 28
Changing Bit Sizes of Values 29

Legacy Support for Non-Keyed Archives 31

Updating a Class to Use Keyed Coding 31
Supporting Keyed and Non-Keyed Archiving 32
Converting Coding Methods After a Class Has Been Archived to Keyed Archives 33
Distinguishing an NSArchiver Archive From an NSKeyedArchiver Archive 33

Subclassing NSCoder 35

Serializations 37

Property Lists 37
 NSPropertyListSerialization 37
 Issues with Property List Serialization 37
Deprecated Classes and Methods 38
 NSSerializer and NSDeserializer 38
 Primitive Data 38

Serializing Objects 39

Serializing Property Lists 39
Serializing Primitive Data 40
 Serializing 40
 Deserializing 41

Document Revision History 43

Figures

Object Graphs 9

Figure 1 Partial object graph of an application 9

Archives 11

Figure 1 Class hierarchy for coders 11

Introduction

Archives and serializations are two ways in which you can create architecture-independent byte streams of hierarchical data. Byte streams can then be written to a file or transmitted to another process, perhaps over a network. When the byte stream is decoded, the hierarchy is regenerated. Archives provide a detailed record of a collection of interrelated objects and values. Serializations record only the simple hierarchy of property-list values.

You should read this document to learn how to create and extract archived representations of object graphs.

Organization of This Document

This programming topic contains the following articles:

- [“Object Graphs”](#) (page 9) introduces the concept of an object graph and discusses the two techniques for turning objects into byte streams: archives and serializations.
- [“Archives”](#) (page 11) describes the different types of archive and archiver classes.
- [“Creating an Archive”](#) (page 17) describes how to create an archive.
- [“Decoding an Archive”](#) (page 19) describes how to decode an archive.
- [“Encoding and Decoding Objects”](#) (page 21) describes how to implement the methods that allow an object to be encoded in and decoded from archives.
- [“Encoding and Decoding C Data Types”](#) (page 25) describes how to encode and decode C data types that do not have convenience methods defined in the archive classes.
- [“Forward and Backward Compatibility for Keyed Archives”](#) (page 27) provides some tips on how to make your classes more compatible with previous and future versions of your classes in keyed archives.
- [“Legacy Support for Non-Keyed Archives”](#) (page 31) describes how to convert older classes to use keyed coding instead of non-keyed coding.
- [“Subclassing NSCoder”](#) (page 35) provides some tips on how to create your own coder classes.
- [“Serializations”](#) (page 37) discusses the behavior of serializations.
- [“Serializing Objects”](#) (page 39) describes how to create and read serializations.

a window's view hierarchy. Property lists are serializations that store the simple hierarchical relationship of basic value objects. More details on archives and serializations, and how you can use them, are described in the following sections.

Archives

Mac OS X archives store an arbitrarily complex object graph. The archive preserves the identity of every object in the graph and all the relationships it has with all the other objects in the graph. When unarchived, the rebuilt object graph should, with few exceptions, be an exact copy of the original object graph.

Interface Builder uses archives (nib file) to store the objects and relationships that make up a user interface. A Cocoa application loads the nib archive to reconstruct a window, menu, or view that was designed in Interface Builder.

Your application can use an archive as the storage medium of your data model. Instead of designing (and maintaining) a special file format for your data, you can leverage Cocoa's archiving infrastructure and store the objects directly into an archive. With minimal effort, you can implement Save and Open in your application.

To support archiving, an object must implement the `NSCoding` protocol, which consists of two methods. One method encodes the object's important instance variables into the archive and the other decodes and restores the instance variables from the archive.

All of the Foundation value objects (`NSString`, `NSArray`, `NSNumber`, and so on) and most of the Application Kit user interface objects implement `NSCoding` and can be put into an archive. Each class's reference document identifies whether they implement `NSCoding`.

Serializations

Mac OS X serializations store a simple hierarchy of value objects, such as dictionaries, arrays, strings, and binary data. The serialization only preserves the values of the objects and their position in the hierarchy. Multiple references to the same value object might result in multiple objects when deserialized. The mutability of the objects is not maintained.

Property lists are examples of serializations. Application attributes (the `Info.plist` file) and user preferences are stored as property lists.

Arbitrary objects cannot be serialized. Only instances of `NSArray`, `NSDictionary`, `NSString`, `NSDate`, `NSNumber`, and `NSData` (and some of their subclasses) can be serialized. The contents of array and dictionary objects must also contain only objects of these few classes.

Note: In Mac OS X version 10.1 and earlier, Cocoa supported the serialization of arbitrary Objective-C data types, but this feature is now deprecated. Documentation of this feature, however, remains in [“Serializations”](#) (page 37) and [“Serializing Objects”](#) (page 39) for maintaining backward compatibility.

Archives

Archives provide a means to convert objects and values into an architecture-independent stream of bytes that preserves the identity of and the relationships between the objects and values.

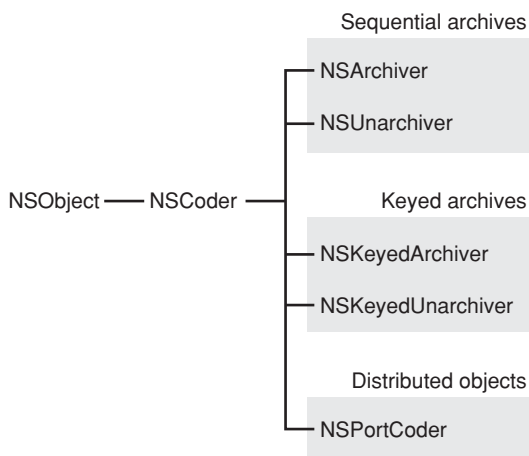
Cocoa archives can hold Objective-C and Java objects, scalars, arrays, structures, and strings. They do not hold types whose implementation varies across platforms, such as `union`, `void *`, function pointers, and long chains of pointers.

Archives store 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. Exceptions to this rule are described in [“Making Substitutions During Coding”](#) (page 23).

Coders

Objects are written to and read from archives with coder objects. Coder objects are instances of concrete subclasses of the abstract class `NSCoder`. `NSCoder` declares an extensive interface for taking the information stored in an object and putting it into another format suitable for writing to a file, transmitting between processes or across a network, or performing other types of data exchange. `NSCoder` also declares the interface for reversing the process, taking the information stored in a byte stream and converting it back into an object. Subclasses implement the appropriate portions of this interface to support a specific archiving format. [Figure 1](#) (page 11) shows the class hierarchy of the coders provided by Cocoa.

Figure 1 Class hierarchy for coders



As shown in [Figure 1](#) (page 11), there are three types of coders: sequential archive coders, keyed archive coders, and distributed object coders. Sequential archive coders read and write sequential archives, which must be processed linearly. In other words, objects must be decoded in the same sequence in which they were encoded in the archive. Keyed archive coders read and write keyed archives. The objects in a keyed

archive are assigned names, or keys, allowing random access to archived objects. Sequential and keyed archives are described in more detail below in “[Keyed Archives](#)” (page 13) and “[Sequential Archives](#)” (page 15). The distributed object coder is used only by the distributed objects architecture in Cocoa and is not described here; `NSPortCoder` is instead described in the Programming Topic *Distributed Objects Programming Topics*.

`NSCoder` defines separate methods for reading and writing objects with and without keys. The coding methods that encode and decode objects without keys are named like `encodeObject:` and `decodeObject:`. The coding methods that use keys are named like `encodeObject:forKey:` and `decodeObjectForKey:`. Sequential archive coders only support the methods without keys. Coders indicate whether they support the keyed coding methods by returning `YES` from its `allowsKeyedCoding` method. Objects being encoded or decoded invoke this method to determine which coding methods to use. If the coder returns `NO`, the object must encode or decode itself using only non-keyed coding methods.

Coder objects read and write objects by sending one of two messages to the objects to be encoded or decoded. A coder sends `encodeWithCoder:` to objects when creating an archive and `initWithCoder:` when reading an archive. These messages are defined by the `NSCoding` protocol. Only objects whose class conforms to the `NSCoding` protocol can be written to an archive. (The reference for each Cocoa class indicates whether the class adopts the `NSCoding` protocol.) When an object receives one of these messages, the object sends messages back to the coder to tell the coder which objects or values, usually instance variables, to read or write next. When encoding objects, the coder records in the archive the class identity of the objects (or type of Objective-C values) and their position in the hierarchy.

Object graphs, such as the one shown in [Figure 1](#) (page 9), pose two problems for a coder: redundancy and constraint. To solve these problems, `NSCoder` introduces the concepts of root objects and conditional objects, which are described in the following sections.

Root Object

An object graph is not necessarily a simple tree structure. Two objects can contain references to each other, for example, creating a cycle. If a coder follows every link and blindly encodes each object it encounters, this circular reference will generate an infinite loop in the coder. Also, a single object can be referenced by several other objects. The coder must be able to recognize and handle multiple and circular references so that it does not encode more than one copy of each object, but still regenerate all the references when decoding.

To solve this problem, `NSCoder` introduces the concept of a root object. The root object is the starting point of an object graph. To encode an object graph, you invoke the `NSCoder` method `encodeRootObject:`, passing in the first object to encode. Every object encoded within the context of this invocation is tracked. If the coder is asked to encode an object more than once, the coder encodes a reference to the first encoding instead of encoding the object again.

`NSCoder` does not implement support for root objects; `NSCoder`'s implementation of `encodeRootObject:` simply encodes the object by invoking `encodeObject:`. It is the responsibility of its concrete subclasses to keep track of multiple references to objects, thus preserving the structure of any object graphs.

Conditional Objects

Another problem presented by object graphs is that it is not always appropriate to archive the entire graph. For example, when you encode an `NSView` object, the view can have many links to other objects: subviews, superviews, formatters, targets, windows, menus, and so on. If a view encoded all of its references to these objects, the entire application would get pulled in. Some objects are more important than others, though.

A view's subviews always should be archived, but not necessarily its superview. In this case, the superview is considered an extraneous part of the graph; a view can exist without its superview, but not its subviews. A view, however, needs to keep a reference to its superview, if the superview is also being encoded in the archive.

To solve this dilemma, `NSCoder` introduces the concept of a conditional object. A conditional object is an object that should be encoded only if it is being encoded unconditionally elsewhere in the object graph. A conditional object is encoded by invoking the `NSCoder` methods `encodeConditionalObject:` and `encodeConditionalObject:forKey:`. If all requests to encode an object are made with these conditional methods, the object is not encoded and references to it decode to `nil`. If the object is encoded elsewhere, all the conditional references decode to the single encoded object.

Typically, conditional objects are used to encode weak, or non-retained, references to objects.

`NSCoder` does not implement support for conditional objects; `NSCoder`'s implementations of `encodeConditionalObject:` and `encodeConditionalObject:forKey:` simply encode the object by invoking `encodeObject:` or `encodeObject:forKey:`. It is the responsibility of its concrete subclasses to keep track of conditional objects and to not encode objects unless they are needed. Both `NSArchiver` and `NSKeyedArchiver` provide full support for conditional objects.

Keyed Archives

Keyed archives are created by `NSKeyedArchiver` objects and decoded by `NSKeyedUnarchiver` objects. Keyed archives differ from sequential archives in that every value encoded in a keyed archive is given a name, or key. When decoding the archive, the values can be requested by name, allowing the values to be requested in any order or not at all. This freedom enables greater flexibility for making your classes forward and backward compatible.

Keyed archives are the preferred archive type in Mac OS X version 10.2 and later.

The following sections describe how to use keyed archives.

Naming Values

Values that an object encodes to a keyed archive can be individually named with an arbitrary string. Archives are hierarchical with each object defining a separate name space for its encoded values, similar to the object's instance variables. Therefore, keys must be unique only within the scope of the current object being encoded. The keys used by object A to encode its instance variables do not conflict with the keys used by object B, even if A and B are instances of the same class. Within a single object, however, the keys used by a subclass can conflict with keys used in its superclasses.

Public classes, such as those in a framework, which can be subclassed, should add a prefix to the name string to avoid collisions with keys that may be used now or in the future by the subclasses of the class. A reasonable prefix is the full name of the class. Cocoa classes use the prefix "NS" in their keys, the same as the API prefix, and carefully makes sure that there are no collisions in the class hierarchy. Another possibility is to use the same string as the bundle identifier for the framework.

You should avoid using "\$" as a prefix for your keys. The keyed archiver and unarchiver use keys prefixed with "\$" for internal values. Although they test for and mangle user-defined keys that have a "\$" prefix, this overhead slows down archiving performance.

Subclasses also need to be somewhat aware of the prefix used by superclasses to avoid accidental collisions on key names. Subclasses of Cocoa classes should avoid unintentionally starting their key names with “NS”. For example, don’t name a key “NSString search options”.

Return Values for Missing Keys

While decoding, if you request a keyed value that does not exist, the unarchiver returns a default value based on the return type of the decode method you invoked. The default values are the equivalent of zero for each data type: `nil` for objects, `NO` for booleans, `0.0` for reals, `NSZeroSize` for sizes, and so on. If you need to detect the absence of a keyed value, use the `NSKeyedUnarchiver` instance method `containsValueForKey:`, which returns `NO` if the supplied key is not present. For performance reasons, you should avoid explicitly testing for keys when the default values are sufficient.

Type Coercions

`NSKeyedUnarchiver` supports limited type coercion. A value encoded as any type of integer, be it a standard `int` or an explicit 32-bit or 64-bit integer, can be decoded using any of the integer decode methods. Likewise, a value encoded as a `float` or `double` can be decoded as either a `float` or a `double` value. When decoding a `double` value as a `float`, though, the decoded value loses precision. If an encoded value is too large to fit within the coerced decoded type, the decoding method throws an `NSRangeException`. Further, when trying to coerce a value to an incompatible type, such as decoding an `int` as a `float`, the decoding method throws an `NSInvalidUnarchiveOperationException`.

Class Versioning

Versioning of encoded data is not handled through class versioning with keyed coding as it is in sequential archives. In fact, no automatic versioning is done for a class; this allows a class to at least get a look at the encoded values without the unarchiver deciding on its own that the versions fatally mismatch. A class is free to decide to encode some type of version information with its other values if it wishes, and this information can be of any type or quantity.

Root Object

For keyed archives, the distinction between the root object and other objects does not apply. The regular `encodeObject:` and `encodeObject:forKey:` methods are able to track multiple references to objects in multiple object graphs. As many object graphs or values as desired may be encoded at the top level of a keyed archive.

To be consistent with the methods of `NSArchiver`, `NSKeyedArchiver` implements `archiveRootObject:toFile:` and `archivedDataWithRootObject:` to produce archives with a single object graph. These archives, however, have to be unarchived using the `NSKeyedUnarchiver` methods `unarchiveObjectWithFile:` and `unarchiveObjectWithData:`.

Delegates

`NSKeyedArchiver` and `NSKeyedUnarchiver` objects, unlike their non-keyed equivalents, can have delegate objects. The delegates are notified as each object is encoded or decoded. You can use the delegates to perform substitutions, replacing one object for another, if desired.

Non-Keyed Coding Methods

Keyed archivers do not have to provide names for every value encoded in the archive. The `NSKeyedArchiver` and `NSKeyedUnarchiver` classes implement the non-keyed encoding and decoding methods that they inherit from `NSCoder`. This way, a class written to work with sequential archiver objects should continue to work properly when encoded by a keyed archiver object.

The use of these non-keyed coding methods is subject to the same restrictions they have for sequential archives (see [“Sequential Archives”](#) (page 15)): (1) you must use the proper decode method for the encode method that was used, and (2) the decoding of items must be in the same order in which they were encoded. Also, even if all the values in the archive are encoded without keys, the archive is still a keyed archive and cannot be decoded by an `NSUnarchiver` object.

With `NSKeyedArchiver` and `NSKeyedUnarchiver`, you can intermix keyed and non-keyed methods. On decode you can ask for the keyed values in any order, but the non-keyed values must be requested in the same order. If you attempt to decode non-keyed values out of order, the unarchiver may give you the wrong values or throw exceptions.

Use of the non-keyed methods within keyed coding is discouraged. Essentially, keyed coding with the non-keyed methods doesn't solve any of the problems or limitations of non-keyed coding—it's just writing the results to a different format.

Sequential Archives

Sequential archives were the only available archive type in Mac OS X version 10.0 and version 10.1. They are deprecated in Mac OS X v10.2 in favor of keyed archives described in [“Keyed Archives”](#) (page 13). Use sequential archives only for backward compatibility with earlier systems.

Sequential archives are created by `NSArchiver` objects and decoded by `NSUnarchiver` objects. Because different classes are used to encode and decode the archives, the instances are referred to as encoder and decoder objects. The sequential archive classes implement methods to create and decode archives stored in either files or `NSData` objects. `NSArchiver` and `NSUnarchiver` objects, unlike their non-keyed equivalents, can *not* have delegate objects.

Sequential archives are characterized as such because the values in the archive must be decoded in the same order in which they were encoded. You cannot skip values nor request them out of order. You must also specify the exact data type of Objective-C values when decoding the value. The decoder objects do not perform any type coercions; if a value was originally encoded as a short integer, it has to be decoded as a short integer. If you attempt to read values out of order or as the wrong data type, you will get incorrect results or cause an exception to be raised.

All the objects to be placed in a single sequential archive must be interconnected members of a single graph. In other words, there can be only one root object per archive. The only recommended way to archive objects is to send an `NSArchiver` object a single `encodeRootObject:` message, whether directly, or indirectly by invoking `archiveRootObjectToFile:` or `archiveDataWithRootObject:`. 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 is part of the graph.

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 is different from the one expected
- an unknown type code is found
- excess characters are found in a type code, or characters are missing

To aid in backward compatibility, sequential archives have built-in support for class versioning. When an object is encoded, its class version, which is set with the `NSObject` class method `setVersion:`, is recorded in the archive. When the object is decoded, you can ask for the version number of its class in the archive. You can then decode the appropriate values for each version of the class. Typically, you have an if-else block containing separate decode sequences for each version number.

Because of the need to decode archives in a specific order and with specific data types, forward and backward compatibility becomes increasingly difficult to maintain as the application evolves. Each time the class of an archived object changes, you have to add special code to handle each version of the class that may be encountered in a pre-existing archive. If new versions of the class must create archives that can be read by older versions, you have to create the older archive, leaving out newer data. Keyed archives solve many of these problems.

Creating an Archive

The easiest way to create an archive of an object graph is to invoke a single class method—either `archiveRootObject:toFile:` or `archivedDataWithRootObject:`—on the archiver class. These convenience methods create a temporary archiver object that encodes a single object graph; you need do no more. The following code fragment, for example, archives a custom object called *myMapView* directly to a file.

```
MapView *myMapView; // Assume this exists
NSString *archivePath = [NSTemporaryDirectory()
stringByAppendingPathComponent:@"Map.archive"];
result = [NSKeyedArchiver archiveRootObject:myMapView
toFile:archivePath];
```

However, if you want to customize the archiving process (for example, by substituting certain classes for others), you must instead create an instance of the archiver yourself, configure it as desired, and send it an `encode` message explicitly. `NSCoder` itself defines no particular method for creating a coder; this typically varies with the subclass. `NSKeyedArchiver` defines `initWithWritingWithMutableData:`.

Once you have the configured coder object, to encode an object or data item, use any `encode` method for an `NSKeyedArchiver` coder. When finished encoding a keyed archive, you must invoke `finishEncoding` before accessing the archive data. The following sample code fragment archives a custom object called *myMapView* similar to the above code, but allows for customization.

```
MapView *myMapView; // Assume this exists
NSMutableData *data;
NSString *archivePath = [NSTemporaryDirectory()
stringByAppendingPathComponent:@"Map.archive"];
NSKeyedArchiver *archiver;
BOOL result;

data = [NSMutableData data];
archiver = [[NSKeyedArchiver alloc] initWithWritingWithMutableData:data];
// Customize archiver here
[archiver encodeObject:myMapView forKey:@"MVMapView"];
[archiver finishEncoding];
result = [data writeToFile:archivePath atomically:YES];
[archiver release];
```

It is possible to create an archive that does not contain any objects. To archive other data types, invoke one of the type-specific methods, such as `encodeInteger:forKey:` or `encodeDouble:forKey:` directly for each data item to be archived, instead of using `archiveRootObject:`.

Legacy Note: Creating a non-keyed archive (using `NSArchiver`) by encoding item by item should not be used to archive objects. Use `archiveRootObject:` instead, to avoid the problems mentioned in “[Root Object](#)” (page 12) and to simplify unarchiving. A keyed archive can have multiple objects encoded with different keys at the root, or top, of the archive.

Decoding an Archive

The easiest way to decode an archive of an object is to invoke a single class method—either `unarchiveObjectWithFile:` or `unarchiveObjectWithData:`—on the unarchiver class. These convenience methods create a temporary unarchiver object that decodes and returns a single object graph; you need do no more. `NSKeyedUnarchiver` requires that the object graph in the archive was encoded with one of `NSKeyedArchiver`'s convenience class methods, such as `archiveRootObject:toFile:.` The following code fragment, for example, unarchives a custom object called *myMapView* directly from a file.

```
MapView *myMapView;
NSString *archivePath = [NSTemporaryDirectory()
stringByAppendingPathComponent:@"Map.archive"];
myMapView = [NSKeyedUnarchiver unarchiveObjectWithFile:archivePath];
```

However, if you want to customize the unarchiving process (for example, by substituting certain classes for others), you must instead create an instance of the unarchiver class yourself, configure it as desired, and send it a `decode` message explicitly. `NSCoder` itself defines no particular method for creating a coder; this typically varies with the subclass. `NSKeyedUnarchiver` defines `initWithReadingWithData:.`

Once you have the configured decoder object, to decode an object or data item, use the `decodeObjectForKey:` method. When finished decoding a keyed archive, you should invoke `finishDecoding` before releasing the unarchiver. The following sample code fragment unarchives a custom object called *myMapView* similar to the above code sample, but allows for customization.

```
MapView *myMapView;
NSData *data;
NSKeyedUnarchiver *unarchiver;
NSString *archivePath = [NSTemporaryDirectory()
stringByAppendingPathComponent:@"Map.archive"];

data = [NSData dataWithContentsOfFile:archivePath];
unarchiver = [[NSKeyedUnarchiver alloc] initWithReadingWithData:data];
// Customize unarchiver here
myMapView = [unarchiver decodeObjectForKey:@"MVMapView"];
[unarchiver finishDecoding];
[unarchiver release];
```

It is possible to create an archive that does not contain any objects. To unarchive non-object data types, simply use the `decode...` method (such as `decodeIntForKey:` or `decodeDoubleForKey:.`) corresponding to the original `encode...` method for each data item to be unarchived.

Encoding and Decoding Objects

To support encoding and decoding of instances, a class must adopt the `NSCoding` protocol and implement its methods. This protocol declares two methods that are sent to the objects being encoded or decoded.

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:`. The `encodeWithCoder:` message instructs the object to encode its instance variables with the provided coder; an object can receive this method any number of times. The `initWithCoder:` message instructs the object to initialize itself from data in the provided coder; as such, it replaces any other initialization method and is sent only once per object.

Encoding an Object

When an object receives an `encodeWithCoder:` message, it should encode all of its vital instance variables, after forwarding the message to its superclass if its superclass also conforms to the `NSCoding` protocol. An object does not 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 “[Conditional Objects](#)” (page 12)).

For example, suppose you were creating a `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 encodeObject:mapName forKey:@"MVMapName"];
    [coder encodeFloat:magnification forKey:@"MVMagnification"];
    [coder encodeObject:legendView forKey:@"MVLegend"];
    [coder encodeConditionalObject:auxiliaryView forKey:@"MVAuxView"];
}
```

Note: This example uses only keyed archiving. If instances of your class can be encoded to a keyed or sequential archive, see “[Supporting Keyed and Non-Keyed Archiving](#)” (page 32).

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 the superclass’s `encodeWithCoder:` method.

The `auxiliaryView` object is encoded conditionally in this example because `MapView` objects do not “own” their auxiliary view. `MapView` objects hold only a weak, non-retained, reference to the auxiliary view. Any weak references held by your objects will normally be encoded conditionally, too.

The `encodeValueOfObjCType:at:` and `encodeObject:` methods are coder methods that you can use to encode instance variables of your class. Use the keyed coding methods, such as `encodeObject:forKey:`, only if the coder supports keyed coding. You can use these and other methods of the coder to encode id's, scalars, C arrays, structures, strings, and pointers to any of these types. See the `NSCoder`, `NSArchiver`, `NSUnarchiver`, `NSKeyedArchiver`, and `NSKeyedUnarchiver` class specifications for a list of methods.

The `@encode()` compiler directive generates an Objective-C type code from a type expression that can be used as the first argument of `encodeValueOfObjCType:at:`. See “Type Encodings” in *The Objective-C Programming Language* for more information.

Decoding an Object

When an object receives an `initWithCoder:` message, the object should first send a message to its superclass (if appropriate) to initialize inherited instance variables, and then it should decode and initialize its own instance variables. For non-keyed coding, the sequence of decode messages must be identical to the sequence of encode messages used in `encodeWithCoder:`. For keyed coding, the keys can be decoded in any order. `MapView`'s implementation of `initWithCoder:` might look like this:

```
- (id)initWithCoder:(NSCoder *)coder {
    self = [super initWithCoder:coder];
    mapName = [[coder decodeObjectForKey:@"MVMapName"] retain];
    legendView = [[coder decodeObjectForKey:@"MVLegend"] retain];
    auxiliaryView = [[coder decodeObjectForKey:@"MVAuxView"] retain];
    magnification = [coder decodeFloatForKey:@"MVMagnification"];
    return self;
}
```

Note: This example uses only keyed archiving. If instances of your class can be encoded to a keyed or sequential archive, see “Supporting Keyed and Non-Keyed Archiving” (page 32).

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 an object other than itself. If the superclass of your class does not support `NSCoding`, you should invoke the superclass's designated initializer instead of `initWithCoder:`.

Also note that the `auxiliaryView` object is not retained after it is decoded. The `MapView` class encoded the object conditionally, because it only had a weak reference to the auxiliary view. If the auxiliary view does not decode to `nil`, some other object in the archive had to encode the view object unconditionally and, presumably, will retain the object when it is decoded. Therefore, by not retaining it here, `MapView` objects restore their weak reference to the auxiliary view.

Performance Considerations

The less you encode for an object, the less you have to decode, and both writing and reading archives becomes faster. Stop writing out items which are no longer pertinent to the class (but which you may have had to continue writing under non-keyed coding).

Encoding and decoding booleans is faster and cheaper than encoding and decoding individual 1-bit bit fields as integers. However, encoding many bit fields into a single integer value can be cheaper than encoding them individually, but that can also complicate compatibility efforts later (see “Structures and Bit Fields” (page 26)).

Don’t read keys you don’t need. You aren’t required to read all the information from the archive for a particular object, as you were with non-keyed coding, and it is much cheaper not to. However, unread data still contributes to the size of an archive, so stop writing it out, too, if you don’t need to read it.

It is faster to just decode a value for a key than to check if it exists and then decode it if it exists. Invoke `containsValueForKey:` only when you need to distinguish the default return value (due to non-existence) from a real value that happens to be the same as the default.

It is more valuable for decoding to be fast than for encoding to be fast. If there is some trade-off you can make that improves decoding performance at the cost of lower encoding performance, the trade-off is usually reasonable to make.

You should avoid using “\$” as a prefix for your keys. The keyed archiver and unarchiver use keys prefixed with “\$” for internal values. Although they test for and mangle user-defined keys that have a “\$” prefix, this overhead slows down archiving performance.

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. Class clusters, for example, take advantage of this feature. This feature 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. These are the generic methods:

Method	Typical use
<code>classForCoder</code>	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.
<code>replacementObjectForCoder:</code>	Allows an object, before being encoded, to substitute another instance in its place.
<code>awakeAfterUsingCoder:</code>	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 are designed for (and invoked by) a specific, concrete coder subclass. For example, `classForArchiver` and `replacementObjectForPortCoder:` are used by `NSArchiver` and `NSPortCoder`,

respectively. 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.

In addition to the methods just discussed, `NSKeyedArchiver` and `NSKeyedUnarchiver` allow a delegate object to perform a final substitution before encoding and after decoding objects. The delegate for an `NSKeyedArchiver` object can implement `archiver:willEncodeObject:` and the delegate for an `NSKeyedUnarchiver` object can implement `unarchiver:didDecodeObject:` to perform the substitutions.

When to Retain a Decoded Object

You can decode an object value in two ways. The first is explicitly, using the `decodeObject` or `decodeObjectForKey:` 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 the coder is responsible for releasing the object.

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 decode values directly into memory that you provide. In the case of objects, the value is the object pointer. As this memory is already owned by you, you are responsible for releasing the objects decoded into it. This behavior can prove useful for optimizing large decoding operations, as it obviates the need for sending a `retain` message to each decoded object.

Restricting Coder Support

In some cases, a class may implement the `NSCoding` protocol, but not support one or more coder types. For example, the classes `NSDistantObject`, `NSInvocation`, `NSPort`, and their subclasses adopt `NSCoding` only for use by `NSPortCoder` within the distributed objects system; they cannot be encoded into an archive. In these cases, a class can test whether the coder is of a particular type and raise an exception if it isn't supported. If the restriction is just to limit a class to sequential or keyed archives, you can send the message `allowsKeyedCoding` to the coder; otherwise, you can test the class identity of the coder as shown in the following sample.

```
- (void)encodeWithCoder:(NSCoder *)coder {
    if ([coder isKindOfClass:[NSKeyedArchiver class]]) {
        // encode object
    }
    else {
        [NSException raise:NSInvalidArchiveOperationException
                 format:@"Only supports NSKeyedArchiver coders"];
    }
}
```

In other situations, a class may inherit `NSCoding` from a superclass, but the subclass may not want to support coding. For example, `NSWindow` inherits `NSCoding` from `NSResponder`, but it does not support coding. In these cases, the class should override the `initWithCoder:` and `encodeWithCoder:` methods so that they raise exceptions if they are ever invoked.

Encoding and Decoding C Data Types

`NSKeyedArchiver` and `NSKeyedUnarchiver` provide a number of methods for handling non-object data. Integers can be encoded with `encodeInt:forKey:`, `encodeInt32:forKey:`, or `encodeInt64:forKey:`. Likewise, real values can be encoded with `encodeFloat:forKey:` or `encodeDouble:forKey:`. Other methods encode booleans and byte arrays. The classes also provide several convenience methods for handling special data types used in Cocoa, such as `NSPoint`, `NSSize`, and `NSRect`.

`NSKeyedArchiver` and `NSKeyedUnarchiver` do not provide methods for encoding and decoding aggregate types, such as structures, arrays, and bit fields. The following sections provide suggestions on how to handle unsupported data types.

Pointers

You can't encode a pointer and get back something useful at decode time. You have to encode the information to which the pointer is pointing. This is true in non-keyed coding as well.

Pointers to C strings (`char *`) are a special case because they can be treated as a byte array which can be encoded using `encodeBytes:length:forKey:`. You can also wrap C strings with a temporary `NSString` object and archive the string. Reverse the process when decoding. Be sure to keep in mind the character set encoding of the string when creating the `NSString` object, and chose an appropriate creation method.

Arrays of Simple Types

If you are encoding an array of bytes, you can just use the provided methods to do so.

For other arithmetic types, create an `NSData` object with the array. *Note that in this case, dealing with platform endianness issues is your responsibility.* Platform endianness can be handled in two general ways. The first technique is to convert the elements of the array (or rather, a temporary copy of the array) into a canonical endianness, either big or little, one at a time with the functions discussed in *Swapping Bytes in Universal Binary Programming Guidelines, Second Edition* (see also "Byte Ordering" in *Foundation Functions Reference*) and give that result to the `NSData` as the buffer. (Or, you can write the bytes directly with `encodeBytes:length:forKey:`.) At decode time, you have to reverse the process, converting from the big or little endian canonical form to the current host representation. The other technique is to use the array as-is and record in a separate keyed value (perhaps a boolean) which endianness the host was when the archive was created. During decoding, read the endian key and compare it to the endianness of the current host and swap the values only if different.

Alternatively, you can archive each array element separately as their native type, perhaps by using key names inspired by the array syntax, like "theArray[0]", "theArray[1]", and so on. This is not a terribly efficient technique, but you can ignore endian issues.

Arrays of Objects

The simplest thing to do for a C array of objects is to temporarily wrap the array in an `NSArray` object with `initWithObjects:count:`, encode the array object, then get rid of the object. Because objects contain other information that has to be encoded, you can't just embed the array of pointers in an `NSData` object; each object must be individually archived. During decoding, use `getObjects:` on the retrieved array to get the objects back out into an allocated C array (of the correct size).

Structures and Bit Fields

The best technique for archiving a structure or a collection of bit fields is to archive the fields independently and choose the appropriate type of encoding/decoding method for each. The key names can be composed from the structure field names if you wish, like “theStruct.order”, “theStruct.flags”, and so on. This creates a slight dependency on the names of the fields in the source code, which over time may get renamed, but the archiving keys cannot change if you want to maintain compatibility.

You should not wrap a structure with an `NSData` object and archive that. If the structure contains object or pointer fields, the data object isn't going to archive them correctly. You also create a dependence on how the compiler decides to lay out the structure, which can change between versions of the compiler and may depend on other factors. A compiler is not constrained to organize a structure just as you've specified it in the source code—there may be arbitrary internal and invisible padding bytes between fields in the structure, for example, and the amount of these can change without notice and on different platforms. In addition, any fields that are multiple bytes in width aren't going to get treated correctly with respect to endianness issues. You will cause yourself all sorts of compatibility trouble.

Likewise, bit fields should never be encoded by reading the raw bits of several bit fields as an integer and encoding the integer. (Encoding an integer that you construct manually from several bit fields, using bit shifts and OR operations, however, avoids most of the pitfalls that follow.) Although there are some requirements on compilers specified in the C standard, a compiler still has some freedom in how things are actually organized and which bits it chooses to store where, and what bits it may choose not to use (inter-field padding bits). The location of those bits could differ between compilers or change as a particular compiler evolves. On top of this, you also have to deal with endianness issues. The order of the bits within an integer could be different for the machine that encodes the archive and the machine that decodes it. Finally, by encoding the raw bits, you constrain future development of your class to use the same bit field sizes as the oldest archive you need to support. Otherwise, you have to be able to parse the old bit stream and initialize the new bit stream yourself, handling the compiler and platform issues appropriately.

As is the case for an object's instance variables in general, it is not necessary to archive every field of a structure or bit field. You only need to encode and decode the fields required to preserve a structure's state. Fields that are calculated or otherwise derived by other means should not be archived.

More Complex Data Types

More complex data types, such as arrays of aggregates, can generally be handled using the techniques for simple data types and combining them with custom logic for your particular application.

Forward and Backward Compatibility for Keyed Archives

Keyed archiving gives you plenty of flexibility to make your classes forward and backward compatible. The following sections describe some general tips on how you can implement compatibility and then some guidelines for maintaining compatibility with specific types of changes.

Benefits of Keyed Archiving

The principal benefit of keyed coding is that it makes it easier to be backward and forward compatible. The ability to read keyed values from the archive in any order, ignore keys you don't need, and add new keys without disrupting older versions of the class is the foundation for implementing backward and forward compatibility with keyed coding.

For maximum compatibility, you need to be able to do the following:

- Read archives created by older versions of your class.
- Create archives that can be read by older versions of your class.
- Read archives created by future versions of your class.
- Create archives that can be read by future versions of your class.

The first two items provide full backward compatibility: the old and current versions of the class can read each others archives. To achieve this capability, it is essential that you know what values were encoded by all the previous versions of your class that you need to support as well as how previous versions decode themselves. If you don't have this information, you may be able to deduce some things from existing archives and the existing implementations of the `NSCoding` methods.

The last two items provide full forward compatibility: the current and future versions of the class can read each others archives. To achieve this capability, you need to anticipate the types of changes you may make in the future and code your current `NSCoding` methods appropriately.

General Tips on Maintaining Compatibility

To easily identify the version of the class being decoded, you can add some version info to the archive. This can be any type of information you want, not just an integer (such as the class version) as it was with non-keyed coding. You may just encode a "version" integer or string with some key or in some rare cases you may want a dictionary object full of goodies. Of course, adding some version information today presumes that you also have a plan for dealing with different versions in your `initWithCoder:` today. If not, changing the version info in the future will not do the present version of the class any good.

Remember to keep your `NSCoding` implementations synchronized. Whenever you change how you write out an objects' state in the class's `encodeWithCoder:` method, you need to update your `initWithCoder:` method to understand the new keys. Because information in a keyed archive can be encoded and decoded in any order, the two `NSCoding` methods don't need to process keys in the same sequence. Use whatever sequences is most convenient for each method.

Adding New Values to Keys

Some of the values a class encodes may have a particular set of possible values. For example, a button can be a checkbox, a radio button, a push button, and so on. In the future, your set of values may expand; you may create a button that has another type of behavior and need to have a new value for the button's type.

To prepare for this change in future archives, you can test whether the decoded value for the key is one of the allowed values. If it is not, you can assign a default value to it. Then, the future version of the class can just assign the new value to the old key and the current class will behave reasonably well.

If you are making this change and a previous version did not make allowances for the change or the allowances are insufficient or unacceptable, you probably have to create a whole new key for the new state (see [“Adding New Keys”](#) (page 28)) and make the old key obsolete (see [“Removing or Retiring Keys”](#) (page 28)).

Adding New Keys

As a class evolves, you may need to add information to the class to describe its new features. For example, a button has a label and a style. Later you may allow the button to have a custom color. You need to create a new key in the archive to hold the color data.

Because you do not need to decode every value in a keyed archive, new keyed values are harmless to old versions of the class, as long as it is OK for them not to be initialized with such state. You can safely add as many new keys as necessary without affecting older versions; old versions automatically ignore those values.

When decoding older archives, you must be prepared to handle the absence of the new key. If appropriate, you can still attempt to decode the new key and just accept the default value for the missing key (`nil`, `0`, `NSZeroPoint`, and so on). The coder's default value may not be valid for every key, however. In that case, you should detect the default value and substitute a more reasonable default value of your own. If the new key is a replacement for an older key, the appropriate substitution should come from the old key, which may require mapping the old value to one of the allowed values for the new key. If you must distinguish between the default value for a missing key and the same value for an existing key, use the `NSCoder` method `containsValueForKey:`.

If the new key is replacing an older key, you need to properly handle the obsolete key (see [“Removing or Retiring Keys”](#) (page 28)).

Removing or Retiring Keys

As a class evolves, some information may become obsolete or replaced by a newer implementation.

Because you do not need to decode every value in a keyed archive, when decoding older archives, you can just ignore keys you no longer need. The decoding will be slightly faster, too.

When decoding future archives, you must be prepared to handle missing keys. If appropriate, you can simply accept the default decode value for the missing keys (`nil`, `0`, `NSZeroPoint`, and so on). If the coder's default value is not valid for a particular key, you should detect the default value and substitute a more reasonable default value of your own. If you must distinguish between the default value for a missing key and the same value for an existing key, use the `NSCoder` method `containsValueForKey:`. In this way, you give yourself the flexibility to stop encoding certain values later.

In cases where you need to abandon an old key for a newer one, but an old class cannot handle a missing key appropriately, you need to keep writing some value for the old key as well as the newer key. The value should be something the old class can understand and should probably be as close a simulation of the new state as possible. For example, consider a class that originally came in “vanilla,” “chocolate,” and “butter pecan” flavors and now has additional “double chocolate” and “caramel” flavors. To encode a value for the old key, you can map “double chocolate” to the value for “chocolate” in the old class, but you may have to map “caramel” to “vanilla.” Of course, you write the entire new set of values with the new key and your `initWithCoder:` method should prefer to use the new key if available.

In some cases it may also be useful to build in fallback handling. Fallback handling is useful when one of a set of possible keys for a value is encoded. The set of supported keys may evolve over time, with newer keys being preferred in future versions of your class. Fallback handling defines a fundamental key that must be readable forever, but is used only when no other recognized keys are present. Future versions can then write a value using both a new key and the fallback key. Older versions of the class will not see the new key, but can still read the value with the fallback key.

Consider as an example a class named `Image` that represents images. (This example does not necessarily reflect the actual behavior of any image class, like `UIImage`.) Suppose the `Image` class is able to encode its instances as a URL, JPEG, or GIF, depending on whichever is most convenient for the particular instance. An encoded `Image` object, therefore, contains only one of the following keys: `@“URL”`, `@“JPEG”`, `@“GIF”`. The `initWithCoder:` method checks for the keys in the order `@“URL”`, `@“JPEG”`, `@“GIF”`, and initializes itself with the first representation that it finds. In the future it might be that none of these are easy or convenient to archive (for example, taking whatever data the `Image` instance does have and converting it to JPEG might be fairly expensive).

An example of fallback handling in this case would be to allow for an additional key (or group of keys), like `@“rawdata”`, that is understood and used by `Image`'s `initWithCoder:` method if none of the other keys for this value (the image data) are present. The value of the `@“rawdata”` key might be defined, for example, to be an `NSData` object containing 32-bit RGBA pixels. There might also be auxiliary keys like `@“pixelshigh”` and `@“pixelswide”` that `initWithCoder:` would look for to get a minimal set of information needed to produce an `Image` instance from the archived information. In the future, the encoding process for an `Image` might write out the convenient information, whatever that is at that time, and would also have to write out the `@“rawdata”` and other keys to allow old decoders to read the object.

Changing Bit Sizes of Values

As technology evolves, the native word size of a computer's processor tends to increase. Someday, your class's code may run on a computer with a processor with a 64-bit native word size.

Encoding what is a 32-bit integer today as a 64-bit integer isn't necessarily the best solution. The extra high-order zero bits you're giving to the value as you give it to the archiver are wasted. On the other hand, it isn't harmful either, and writing the code to understand 64-bit today while you're making changes may allow you to avoid upgrading it in the future (as long as you deal with it properly at decode time).

If you are using the generic `encodeInt(forKey:)` and `decodeIntForKey:` methods, these read and write whatever the native `int` size is on the computer. On a 64-bit computer, `int` may be 64 bits wide (or it might not be; the C language is flexible in this regard). Therefore, in the future it's possible that values requiring more than 32 bits to represent may be written by an `encodeInt(forKey:)` method. If such an archive is transported to a 32-bit computer, the `decodeIntForKey:` method may be unable to represent that integer in the `int` return value, and have to throw an `NSRangeException`.

Whether or not it is useful to attempt to handle this now by always decoding such integers as 64-bit is debatable. If the integer is a "count" of something, for example, it may be physically impossible to have more than 2^{32} of whatever it is on a 32-bit computer, so further attempting to unarchive the file is probably a waste of time, and an exception is reasonable. Alternatively, you might want to either catch the exception or perform your own bounds checking on a 64-bit decoded value and return `nil` from `initWithCoder:`. However, the caller of the `decodeObjectForKey:` method that is unpacking an instance of your class may not like the `nil` any more than the exception, and might end up raising an exception of its own that is less intelligible as to the cause of the problem than the range exception might have been.

Legacy Support for Non-Keyed Archives

Keyed archives are available only in Mac OS X version 10.2 and later. If your application must run in earlier versions, you cannot make use of keyed archives. There should typically be no need to retrofit support for non-keyed archiving to a class originally written to support keyed archiving. Since non-keyed archiving is deprecated, you should add support for keyed archiving to classes that currently use only non-keyed archiving.

Updating a Class to Use Keyed Coding

Updating your classes to use keyed coding is not difficult. In most cases, your `NSCoding` methods need to assign keys only to the values being encoded and decoded. Special issues come up if you still need to handle sequential archives. Here are recommendations for changing your class to do keyed coding:

- Preserve current non-keyed coding behavior in public classes. `NSArchiver` and `NSUnarchiver` still exist to allow applications to read and write old-style archives. If other developers may have used your class, or subclassed it, you should keep the old encoding and decoding code for non-keyed coding, but execute the keyed coding code if the coder supports keyed coding. (See [“Encoding and Decoding Objects”](#) (page 21).)
- Reassess whether private classes need to be codable. If a class doesn't need to be codable, or shouldn't be codable, but is currently implementing `NSCoding`, you should remove the `NSCoding` conformance from the interface declaration and remove the methods. If the class is encoded or decoded by other classes outside your control, however, you need to preserve the old behavior. You don't have to support new keyed coding behavior, though. You can prevent the class from ever being encoded into a keyed archive by raising an exception if the coder supports keyed coding. Conversely, you may not want new classes to support non-keyed coding, in which case you can raise an exception if the coder does not support keyed coding.
- Reassess what values are being encoded. If the old code was archiving stuff that is no longer needed, just for the sake of compatibility, now is a good time to stop doing that. Nominally, an `initWithCoder:` method might become much simpler without the need to do old class version checking.
- Convert old methods to new methods. There may not be exact matches for the old methods in the set of new keyed methods—choose something reasonable. Decide on names for each value, and don't forget to put some sort of prefix or other “uniqueness guarantor” into the key strings. Do not forget to retain the return value of `decodeObjectForKey:`, when converting code that used to decode the objects with the `decodeValueOfObjCType:at:`, `decodeValuesOfObjCTypes:`, and `decodeArrayOfObjCType:count:at:` methods. Also note that the `decodeBytesForKey:returnedLength:` method returns a pointer to bytes that cannot be mutated—you must make a copy if you want to change the bytes.
- Freeze the old encoding and decoding methods along with the class version number. Any `setVersion:` calls should be left in, so that the class version remains at its current value. Of course, you can keep updating the old non-keyed coding algorithms and changing the class version, as before, but users of

your updated class should also be updating to keyed coding. The Cocoa classes, for example, will not be updating their non-keyed coding algorithms, even when new features and state are added—non-keyed archives will simply not save or get the new features.

Supporting Keyed and Non-Keyed Archiving

The following example shows how you can implement archiving for a class "MapView" to support keyed and non-keyed archiving. 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 lines that invoke the superclass's `encodeWithCoder:` and `initWithCoder:` methods.

```
- (void)encodeWithCoder:(NSCoder *)coder {
    [super encodeWithCoder:coder];
    if ([coder allowsKeyedCoding]) {
        [coder encodeObject:mapName forKey:@"MVMapName"];
        [coder encodeFloat:magnification forKey:@"MVMagnification"];
        [coder encodeObject:legendView forKey:@"MVLegend"];
        [coder encodeConditionalObject:auxiliaryView forKey:@"MVAuxView"];
    }
    else {
        [coder encodeObject:mapName];
        [coder encodeValueOfObjCType:@encode(float) at:&magnification];
        [coder encodeObject:legendView];
        [coder encodeConditionalObject:auxiliaryView];
    }
}

- (id)initWithCoder:(NSCoder *)coder {
    self = [super initWithCoder:coder];
    if ([coder allowsKeyedCoding]) {
        // Can decode keys in any order
        mapName = [[coder decodeObjectForKey:@"MVMapName"] retain];
        legendView = [[coder decodeObjectForKey:@"MVLegend"] retain];
        auxiliaryView = [[coder decodeObjectForKey:@"MVAuxView"] retain];
        magnification = [coder decodeFloatForKey:@"MVMagnification"];
    }
    else {
        // Must decode keys in same order as encodeWithCoder:
        mapName = [[coder decodeObject] retain];
        [coder decodeValueOfObjCType:@encode(float) at:&magnification];
        legendView = [[coder decodeObject] retain];
        auxiliaryView = [coder decodeObject];
    }
    return self;
}
```

When unarchiving data from a sequential archive, the corresponding unarchiving code must follow exactly the same sequence of data types. Matching these is important, as the method originally used determines the format of the encoded data.

Converting Coding Methods After a Class Has Been Archived to Keyed Archives

Because `NSKeyedArchiver` also implements the non-keyed coding methods that it inherits from `NSCoder`, a class that has not been updated for keyed coding can still be encoded into a keyed archive. This can happen if the application creating the archive has been updated to use `NSKeyedArchiver`, but a class has not, so it still uses the old style encoding methods. The class's instance variables are written to the archive without keys, just like for a sequential archive. If this may have happened for one of your classes, when you update your class, you must be able to handle the case where although the unarchiver supports keyed coding, the object's instance variables were not encoded with keys. If this occurs, you must decode the values as if they are coming from a non-keyed archive. In other words, you must decode the values in the same sequence and with the matching non-keyed decoding methods as when encoded.

The simplest technique is to use a version key of some sort. When (after conversion), the object encodes itself, it needs to write a special keyed value which indicates that the object was encoded using keyed coding methods. At decoding time, if the coder allows keyed coding and this special key exists, then `initWithCoder:` knows that not only is this a keyed archive, but keyed coding methods were also used. If the key does not exist, `initWithCoder:` must still use the old decoding algorithm.

```
- (id)initWithCoder:(NSCoder *)coder {
    if ([coder allowsKeyedCoding]
        && [coder containsValueForKey:@"UsesKeyedCoding"] ) {
        // Use keyed coding methods
    }
    else {
        // Use non-keyed coding methods
    }
    return self;
}
```

Distinguishing an NSArchiver Archive From an NSKeyedArchiver Archive

Ideally, you should use a different file extension for a new document format which is keyed-archiving-based rather than non-keyed-archiving-based. If this is not possible, then you can look at the first few bytes of the archived data (the "magic number"). If the data is at least 13 bytes long, and the 2nd-13th bytes are `\00btypedstream` or `\00bstreamtyped` then you have an old archive. A suitable test is illustrated in the following code fragment:

```
if (13 <= dataLength &&
    ((databytes[1] == 0xb && 0 == memcmp(databytes + 2, "typedstream", 11))
    ||
    (databytes[1] == 0xb && 0 == memcmp(databytes + 2, "streamtyped", 11))))
{
    // non-keyed archive ...
}
```


Subclassing NSCoder

`NSCoder`'s interface is quite general and extensive, declaring methods to encode and decode objects and values with and without keys. Concrete subclasses are not required to properly implement all of `NSCoder`'s methods and may explicitly restrict themselves to certain types of operations. For example, `NSArchiver` does not implement the `decode...` methods, and `NSUnarchiver` does not implement the `encode...` methods. In addition, neither class implements the keyed coding methods for encoding and decoding keyed archives. Invoking a `decode` method on `NSArchiver` or an `encode` method on `NSUnarchiver` raises an `NSInvalidArgumentException`.

If you define a subclass of `NSCoder` that does not support keyed coding, at a minimum your subclass must override the following methods:

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

If your subclass supports keyed coding, you must override the above methods as well as the `allowsKeyedCoding` method (to return `YES`) and all of the keyed coding methods defined by `NSCoder`. In both cases, if you are creating separate classes for encoding and decoding, you do not need to override the `encode` methods in the decoder class nor the `decode` methods in the encoder class.

Note that `encodeObject:` and `decodeObject:` are not among the basic methods. They are 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 "[Making Substitutions During Coding](#)" (page 23)) to the object before encoding it and after decoding it.

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

```
encodeRootObject:
encodeConditionalObject:
encodeBycopyObject:
encodeByrefObject:
```

See the individual method descriptions for more information on their required behavior. The default `NSCoder` implementations of these methods just invoke `encodeObject:`.

If you override `encodeConditionalObject:` to support conditional objects (see "[Conditional Objects](#)" (page 12)), be aware that the first unconditional encoding may occur after any number of conditional encoding requests, so your coder will not know which conditional objects to encode until all the other objects have been encoded.

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 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 when it receives the `encodeWithCoder:` and `initWithCoder:` messages. 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 implement the appropriate custom methods to encode and decode `NSData` and (if relevant) `NSPort` objects itself.

Serializations

Serialization converts Objective-C types to and from an architecture-independent byte stream. In contrast to archiving, basic serialization does not record the data type of the values nor the relationships between them; only the values themselves are recorded. It is your responsibility to deserialize the data in the proper order. Several convenience classes, however, do provide the ability to serialize property lists, recording their structure along with their values.

Property Lists

In Cocoa, property lists are defined as simple object graphs consisting of nothing but the following types: `NSDictionary`, `NSArray`, `NSString`, `NSDate`, `NSNumber`, Java number objects (such as `Integer`), and `NSData`. Several classes are defined that can serialize these property list objects directly. These classes define special formats for the data stream which record the contents of the objects and their hierarchical relationship.

For more details on property lists, see the Programming Topic *Property List Programming Guide*.

NSPropertyListSerialization

The `NSPropertyListSerialization` class provides the serialization methods that convert property list objects to and from either an XML or an optimized binary format. This class can also read the old-style OpenStep string format, although numbers and dates are interpreted as strings. Container objects are always created immutable when deserializing in Java, but can be mutable or immutable when deserializing in Objective-C.

The `NSPropertyListSerialization` class object provides the interface to the serialization process; you don't create instances of `NSPropertyListSerialization`.

Issues with Property List Serialization

Property list serialization does not preserve the full class identity of the objects, only its general kind—a dictionary, an array, and so on. As a result, 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. In particular, when a property list is serialized, the mutability of the container objects (`NSDictionary` and `NSArray` objects) is not preserved. When deserializing, though, you can choose to have all container objects created mutable or immutable.

Serialization also does not track the presence of objects referenced multiple times. Each reference to an object within the property list is serialized separately, resulting in multiple instances when deserialized.

Because serialization does not preserve class information or mutability, nor handles multiple references, coding (as implemented by `NSCoder` and its subclasses) is the preferred way to make object graphs persistent.

Deprecated Classes and Methods

Note: `NSSerializer` and `NSDeserializer` were deprecated in Mac OS X version 10.2 and are obsolete. Use `NSPropertyListSerialization` instead.

`NSData`'s and `NSMutableData`'s serialization methods were deprecated in Mac OS X version 10.2 and are obsolete. Use the `NSPropertyListSerialization` class instead.

NSSerializer and NSDeserializer

The `NSSerializer` and `NSDeserializer` classes, available only in Objective-C, provide mechanisms for converting between a property list and an architecture-independent representation of the property list. The `NSSerializer` class stores the property list representation in an `NSData` object that can be written to a file or transmitted to another process or machine. Conversely, the `NSDeserializer` class converts a representation of a property list (as contained in an `NSData` object) back into a collection of property list objects in memory. Options to these `NSDeserializer` methods allow you to specify that container objects (arrays or dictionaries) in the resulting object 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 a property list is deserialized only if it is actually going to be accessed.

The `NSSerializer` and `NSDeserializer` class objects provide the interface to the serialization process; you don't create instances of `NSSerializer` or `NSDeserializer`. You might create subclasses to modify the representation it creates, for example, to encrypt the data or add authentication information.

Primitive Data

`NSData` and `NSMutableData` define the methods `serializeDataAt:ofObjCType:context:` and `deserializeDataAt:ofObjCType:atCursor:context:` to serialize and deserialize Objective-C types. These methods can be used to serialize the basic types, such as integers, floats, character strings, and so on, as well as structures and arrays.

These methods do not directly support the serialization of objects, but by implementing the `NSObjCTypeSerializationCallback` protocol you can provide a helper object (via the context argument) that can serialize and deserialize the object in a non-object form. For example, an `NSString` object can be converted to a C string and then serialized; when deserializing, the helper object can read the C string and convert it back into an `NSString` object. See [“Serializing Objects”](#) (page 39) for details.

Serializing Objects

Serialization comes in two types: property list serialization and primitive data serialization. Examples of each are shown in the following sections

Serializing Property Lists

Property lists are object graphs consisting exclusively of instances of `NSArray`, `NSDictionary`, `NSString`, `NSData`, `NSDate`, `NSNumber`, and their mutable subclasses. The class methods of the `NSPropertyListSerialization` class handle the conversion between the `NSData` byte stream and the object graph. The following code sample shows how to serialize a simple property list into an XML format.

```
NSData *dataRep;
NSString *errorStr = nil;
NSDictionary *propertyList;

propertyList = [NSDictionary dictionaryWithObjectsAndKeys:
                @"Javier", @"FirstNameKey",
                @"Alegria", @"LastNameKey", nil];
dataRep = [NSPropertyListSerialization dataFromPropertyList: propertyList
                                       format: NSPropertyListXMLFormat_v1_0
                                       errorDescription: &errorStr];
if (!dataRep) {
    // Handle error
}
```

The following code sample converts the XML data from above back into an object graph.

```
NSData *dataRep; // Assume this exists
NSString *errorStr = nil;
NSDictionary *propertyList;
NSPropertyListFormat format;

propertyList = [NSPropertyListSerialization propertyListFromData: dataRep
                                       mutabilityOption: NSPropertyListImmutable
                                       format: &format
                                       errorDescription: &errorStr];
if (!propertyList) {
    // Handle error
}
```

Serializing Primitive Data

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, which is not directly supported by Cocoa's serialization facility.

Note: The serialization methods described in this section are obsolete and have been deprecated. Use `NSPropertyListSerialization` instead.

Serializing

`NSMutableData` declares the method that is 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 *callback* object passed to the method is asked to provide the serialization.

Suppose 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 code fragment:

```
MyHelperObject *helper; // assume exists and conforms to protocol
NSMutableData *theData = [NSMutableData data];
struct stockRecord aRecord = @{@"aCompany", 34.7};

[theData serializeDataAt:&aRecord ofObjCType:@"{@f}" context:helper];
```

Because 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 *)theData {
    NSString *nameObject;
    char *companyName;

    nameObject = *objectPtr;
    companyName = [nameObject UTF8String];

    [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.

The above implementation assumes the object sent to it is always an `NSString` object. If the helper object is to support serializing objects of different types (for example, if the serialized structure contains multiple objects of different types), you need to inspect `objectPtr` to identify what type of object is being serialized at each invocation. This can be done with the `isKindOfClass:` method. For example, the implementation of `serializeObjectAt:ofObjCType:intoData:` could be expanded to contain the following code fragment:

```
if ([*objectPtr isKindOfClass:[NSString class]]) {
    // Record the object type for deserialization
    char classType = ENCODE_STRING;
    [theData serializeDataAt:&classType
                ofObjCType:@encode(typeof(classType))
                context:nil];
    // Now encode the C string version of *objectPtr
}
else if (/* Test other types */) {
    // ...
}
```

Because the helper is serializing multiple object types, the object type being recorded must be stored within the byte stream so that the helper object can know what type of object to create when deserializing the byte stream. `ENCODE_STRING` is assumed to be a macro that holds a numerical value identifying each object type the helper supports.

Deserializing

Deserialization follows a similar pattern, except in this case `NSData` declares the central method that is used to begin the deserialization process:

```
- (void)deserializeDataAt:(void *)data
    ofObjCType:(const char *)type
    atCursor:(unsigned *)cursor
    context:(id <NSObjCTypeSerializationCallback>)callback
```

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 identifies where within `theData` to read the data. It is initialized to zero to start deserializing at the beginning of the data. The `cursor` argument is incremented by the length of the data processed.

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 *)theData
    atCursor:(unsigned *)cursor {
    char *companyName;

    [theData deserializeDataAt:&companyName ofObjCType:@"*" atCursor:cursor
        context:nil];
    *objectPtr = [[NSString stringWithCString:companyName] retain];
}
```

If `helper` supports multiple object types, as described in [“Serializing”](#) (page 40), you need to read the type code first and create the appropriate object:

```
char classType;

[theData deserializeDataAt:&classType ofObjCType:@encode(sizeof(classType))
    atCursor:cursor context:nil];
switch (classType) {
    case ENCODE_STRING:
        // Read the C string and create an NSString
        break;
    /* ... */
}
```

Document Revision History

This table describes the changes to *Archives and Serializations Programming Guide for Cocoa*.

Date	Notes
2009-08-18	Added links to related concepts.
2009-02-04	Replaced instances of <code>"/tmp"</code> with <code>NSTemporaryDirectory()</code> .
2007-10-31	Added discussion of how to distinguish old archives from keyed archives.
2007-07-09	Removed pointers to deprecated reference.
2006-11-07	Corrected inaccurate description of code sample in "Encoding an Object."
2006-02-07	Clarified preference of keyed over non-keyed archiving. Changed title from "Archives and Serializations."
2003-04-02	Changed use of <code>cString</code> , which will eventually be deprecated, to <code>UTF8String</code> in sample code in " Serializing Objects " (page 39).
2002-11-12	Revision history was added to existing topic.

