Archives and Serializations Programming Guide

Developer

Contents

Figures

Introduction

Important: This is a preliminary document for an API or technology in development. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein for use on Apple-branded products. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future betas of the API or technology.

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 6) introduces the concept of an object graph and discusses the two techniques for turning objects into byte streams: archives and serializations.
- Archives (page 9) describes the different types of archive and archiver classes.
- Creating and Extracting Archives (page 13) describes how to create and extract an archive.
- Encoding and Decoding Objects (page 16) 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 21) 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 24) provides some tips on how to make your classes more compatible with previous and future versions of your classes in keyed archives.
- Subclassing NSCoder (page 29) provides some tips on how to create your own coder classes.
- Serializing Property Lists (page 31) describes how to create and read serialized representations of a property list.

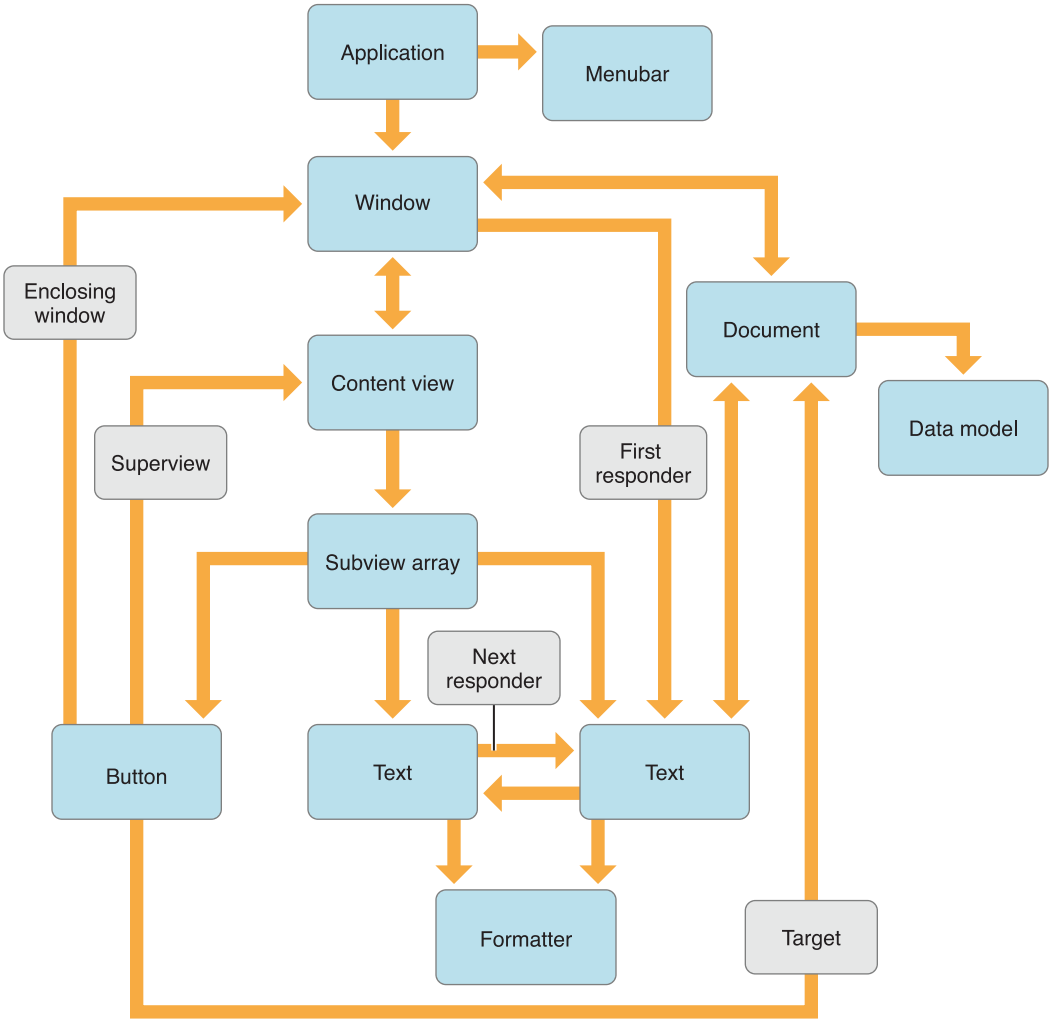
Object Graphs

Object-oriented applications contain complex webs of interrelated objects. Objects are linked to each other by one object either owning or containing another object or holding a reference to another object to which it sends messages. This web of objects is called an object graph.

Even with very few objects, an application's object graph becomes very entangled with circular references and multiple links to individual objects. Figure 1 (page 7) shows an incomplete object graph for a simple Cocoa application in OS X. (Many more connections exist than are shown in this figure.) Consider the window's view hierarchy portion of the object graph. This hierarchy is described by each view containing a list of all of its

immediate subviews. However, views also have links to each other to describe the responder chain and the keyboard focus loop. Views also link to other objects in the application for target-action messages, contextual menus, and much more.

Figure 1 Partial object graph of an application



There are situations where you may want to convert an object graph, usually just a section of the full object graph in the application, into a form that can be saved to a file or transmitted to another process or machine and then reconstructed. Nib files and property lists are two examples in OS X where object graphs are saved to a file. Nib files are archives that represent the complex relationships within a user interface, such as 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

An archive can 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.

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.

To support archiving, an object must adopt 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 objects (`NSString`, `NSArray`, `NSNumber`, and so on) and most of the Application Kit and UIKit user interface objects adopt `NSCoding` and can be put into an archive. Each class's reference document identifies whether they adopt `NSCoding`.

Serializations

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.

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 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 18).

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.

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 7), 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 `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 references to objects.

`NSCoder` does not implement support for conditional objects; `NSCoder`'s implementation of `encodeConditionalObject:forKey:` simply encodes the object by invoking `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.

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.

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 Objects

The `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.

`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 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`. Use of the non-keyed methods within keyed coding is discouraged.

Creating and Extracting Archives

This chapter describes how to create and extract an archive. To understand how to make your custom objects support archival, see Encoding and Decoding Objects (page 16).

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 `aPerson` directly to a file.

```
Person *aPerson = <#Get a Person#>;
NSString *archivePath = <Path for the archive#>;
BOOL success = [NSKeyedArchiver archiveRootObject:aPerson 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 `initForWritingWithMutableData:`.

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, you must invoke `finishEncoding` before accessing the archive data. The following sample code fragment archives a custom object called `aPerson` similar to the above code, but allows for customization.

```
Person *aPerson = <#Get a Person#>;
NSString *archivePath = <Path for the archive#>;
NSMutableData *data = [NSMutableData data];
NSKeyedArchiver *archiver = [[NSKeyedArchiver alloc]
initForWritingWithMutableData:data];
[archiver encodeObject:aPerson forKey:ASCPersonKey];
[archiver finishEncoding];
```

```
NSURL *archiveURL = <URL for the archive#>;
BOOL result = [data writeToURL:archiveURL atomically:YES];
```

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 `encodeRootObject:`.

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 `aPerson` directly from a file.

```
NSString *archivePath = <Path for the archive#>;
Person *aPerson = [NSKeyedUnarchiver unarchiveObjectWithFile:archivePath];
```

However, if you want to customize the unarchiving process (for example, by substituting certain classes for others), you typically 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 `initForReadingWithData:`.

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 `myMap` similar to the above code sample, but allows for customization.

```
NSURL *archiveURL = <URL for the archive#>;
NSData *data = [NSData dataWithContentsOfURL:archiveURL];


NSKeyedUnarchiver *unarchiver = [[NSKeyedUnarchiver alloc]
initForReadingWithData:data];
// Customize the unarchiver.
Person *aPerson = [unarchiver decodeObjectForKey:ASCPersonKey];
```

```
[unarchiver finishDecoding];
```

You can 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.

To customize the unarchiving process for an archive previously created using `archiveRootObject:toFile:`, use `decodeObjectForKey:` and the key “root” to identify the root object in the archive:

```
id object = [unarchiver decodeObjectForKey:@"root"];
```

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 state. A coder instructs the object to do so by invoking `encodeWithCoder:` or `initWithCoder:`. The `encodeWithCoder:` method instructs the object to encode its state 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 state (often represented by its properties or 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 state. Some values may not be important to reestablish and others may be derivable from related state upon decoding. Other state should be encoded only under certain conditions (for example, with `encodeConditionalObject:`, as described in Conditional Objects (page 10)).

You use the coding methods, such as `encodeObject:forKey:`, to encode id's, scalars, C arrays, structures, strings, and pointers to any of these types. See the `NSKeyedArchiver` class specification for a list of methods. For example, suppose you created a Person class that has properties for first name, last name, and height; you might implement `encodeWithCoder:` as follows

```
- (void)encodeWithCoder:(NSCoder *)coder {
    [coder encodeObject:self.firstName forKey:ASCPersonFirstName];
    [coder encodeObject:self.lastName forKey:ASCPersonLastName];
    [coder encodeFloat:self.height forKey:ASCPersonHeight];
}
```

This example assumes that the superclass of Person does not adopt the `NSCoding` protocol. If the superclass of your class does adopt `NSCoding`, you should invoke the superclass's `encodeWithCoder:` method before invoking any of the other encoding methods:

```
- (void)encodeWithCoder:(NSCoder *)coder {
    [super encodeWithCoder:coder];
    // Implementation continues.
```

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

In the implementation of an `initWithCoder:` method, the object should first invoke its superclass's designated initializer to initialize inherited state, and then it should decode and initialize its state. The keys can be decoded in any order. Person's implementation of `initWithCoder:` might look like this:

```
- (id)initWithCoder:(NSCoder *)coder {
    self = [super init];
    if (self) {
        _firstName = [coder decodeObjectForKey:ASCPersonFirstName];
        _lastName = [coder decodeObjectForKey:ASCPersonLastName];
        _height = [coder decodeFloatForKey:ASCPersonHeight];
    }
    return self;
}
```

If the superclass adopts the `NSCoding` protocol, you start by assigning of the return value of `initWithCoder:` to `self`:

```
- (id)initWithCoder:(NSCoder *)coder {
    self = [super initWithCoder:coder];
    if (self) {
        // Implementation continues.
```

This is done in the subclass because the superclass, in its implementation of `initWithCoder:`, may decide to return an object other than itself.

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 22)).

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 typically 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.

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 makes archiving slower.

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
`classForCoder`	Allows an object, before being encoded, to substitute a class other than its own. For example, the private subclasses of a class cluster substitute the name of their public superclass when being archived.
`replacementObject-ForCoder:`	Allows an object, before being encoded, to substitute another instance in its place.
`awakeAfterUsing-Coder:`	Allows an object, after being decoded, to substitute another object for itself. For example, an object that represents a font might, upon being decoded, 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.

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 25)) and make the old key obsolete (see Removing or Retiring Keys (page 26)).

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 26)).

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 `NSImage`.) Suppose the Image class is able to encode its instances as an 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

In some situations, you may have code and archives that you use on 32- and 64-bit platforms (for example, you might have iOS and OS X versions of an application that share data via iCloud). In these cases, you need to take care to ensure that integer values are treated correctly.

Encoding what is a 32-bit integer 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 is easy to implement.

The generic `encodeInt:forKey:` and `decodeIntForKey:` methods 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, 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 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.

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 18)) 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 10)), 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.

Serializing Property Lists

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.

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.

The `NSPropertyListSerialization` class provides the serialization methods that convert property list objects to and from either an XML or an optimized binary format. The `NSPropertyListSerialization` class object provides the interface to the serialization process; you don't create instances of `NSPropertyListSerialization`.

The following code sample shows how to serialize a simple property list into an XML format.

```
NSDictionary *propertyList= @{ @"FirstNameKey" : @"Edmund",
                               @"LastNameKey" : @"Blackadder" };
NSString *errorStr;
NSData *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.

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

Document Revision History

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

Date	Notes
2012-07-17	Removed legacy information about non-keyed archiving.
2010-09-01	Added note for customizing unarchival of archives created using archiveRootObject:toFile:.
2009-08-18	Added links to related concepts.
2009-02-04	Replaced instances of "/tmp" with NSTemporaryDirectory().
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 `cString`, which will eventually be deprecated, to `UTF8String` in sample code in Serializing Property Lists (page 31).
2002-11-12	Revision history was added to existing topic.

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