Property List Programming Guide

Developer

Contents

Contents

Tables and Listings

SwiftObjective-C

Introduction to Property Lists

Property lists organize data into named values and lists of values using several object types. These types give you the means to produce data that is meaningfully structured, transportable, storable, and accessible, but still as efficient as possible. Property lists are frequently used by applications running on both OS X and iOS. The property-list programming interfaces for Cocoa and Core Foundation allow you to convert hierarchically structured combinations of these basic types of objects to and from standard XML. You can save the XML data to disk and later use it to reconstruct the original objects.

This document describes property lists and their various representations, and how to work with them using both certain Foundation classes of Cocoa and Property List Services of Core Foundation.

> **Note:** The user defaults system, which you programmatically access through the `NSUserDefaults` class, uses property lists to store objects representing user preferences. This limitation would seem to exclude many kinds of objects, such as `NSColor` and `NSFont` objects, from the user default system. But if objects conform to the `NSCoding` protocol they can be archived to `NSData` objects, which are property list–compatible objects. For information on how to do this, see "Storing NSColor in User Defaults"; although this article focuses on `NSColor` objects, the procedure can be applied to any object that can be archived.

Organization of This Document

This document consists of the following chapters:

- Quick Start for Property Lists (page 8) is a mini-tutorial on property lists, giving you a "hands-on" familiarity with XML property lists and the Objective-C serialization API.

- About Property Lists (page 15) explains what property lists are and when you should use them.

- Creating Property Lists Programmatically (page 18) shows how you can create hierarchically structured property lists using the APIs of Cocoa and Core Foundation.

- Understanding XML Property Lists (page 25) describes the format of XML property lists.

- Serializing a Property List (page 27) discusses how to serialize and deserialize property lists between their runtime and static representations).

- Reading and Writing Property-List Data (page 37) describes how to save property lists to files or URL resources and how to restore them later.

- Old-Style ASCII Property Lists (page 41) is an appendix that describes the format of old-style (OpenStep) ASCII property lists.
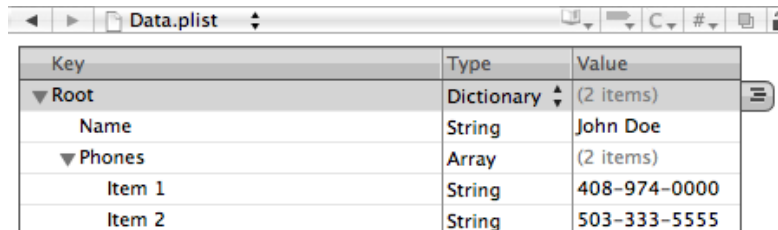
Quick Start for Property Lists

This mini-tutorial gives you a quick, practical introduction to property lists. You start by specifying a short property list in XML. Then you design an application that, when it launches, reads and converts the elements of the XML property list into their object equivalents and stores these objects in instance variables. The application displays these object values in the user interface and allows you to change them. When you quit the application, it writes out the modified property list as XML. When you relaunch the application, the new values are displayed.

Create the XML Property List

In Xcode, create a simple Cocoa application project—call it PropertyListExample. Then select the Resources folder of the project and choose New File from the File menu. In the "Other" template category, select the Property List template and click Next. Name the file "Data.plist".

Double-click the `Data.plist` file in Xcode (you'll find it in the Resources folder). Xcode displays an empty property list in a special editor. Edit the property list so that it looks like the following example:

You can also edit the property list in a text editor such as TextEdit or BBEdit. When you're finished, it should look like the following XML code.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Name</key>
    <string>John Doe</string>
    <key>Phones</key>
    <array>
```

```
        <string>408−974−0000</string>
        <string>503−333−5555</string>
    </array>
</dict>
</plist>
```

Because the property-list file is in the Resources folder, it will be written to the application's main bundle when you build the project.

Define Storage for the Property-List Objects

In this step, you'll add a coordinating controller class to the project and declare properties to hold the property-list objects defined in `Data.plist`. (Note the distinction here between declared *property* and *property*-list object.)

In Xcode, select the Classes folder and choose New File from the File menu. Select the Objective-C Class template and name the files "Controller.h" and "Controller.m". Make the following declarations in `Controller.h`.

```
#import <Cocoa/Cocoa.h>


@interface Controller : NSObject {

    NSString *personName;

    NSMutableArray *phoneNumbers;

}


@property (copy, nonatomic) NSString *personName;

@property (retain, nonatomic) NSMutableArray *phoneNumbers;


@end
```

In `Controller.m`, have the compiler synthesize accessor methods for these properties:

```
@implementation Controller


@synthesize personName;

@synthesize phoneNumbers;

```

```
@end
```

Create the User Interface

Double-click the project's nib file to open it in Interface Builder. Create a simple user interface similar to the following:
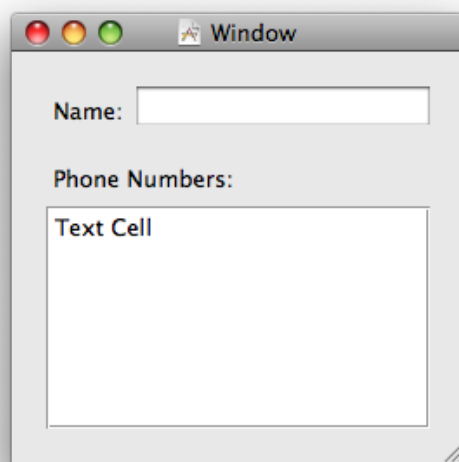
> **Note:** This mini-tutorial shows user-interface techniques and programming interfaces that are specific to OS X. If you want to use an iOS application for this example, you must use techniques and API that are appropriate to that platform. The code specific to property lists is applicable to both platforms.

The table view should have a single column that is editable.

For reasons of simplicity and efficiency, you'll next bind the text field to the `personName` property. But your Controller object will act as a data source for the table. Let's start with the text field.

1. Drag an generic Object proxy from the Library into the nib document window. Select it and, in the Identify pane of the inspector, type or select "Controller" for its class identity.

2. Drag an Object Controller object from the Library into the nib document window. Control-drag (or right-click-drag) a line from Object Controller to Controller and, in the connection window that pops up, select "content".

3. Select the editable text field and, in the Bindings pane of the inspector, bind the value attribute of the text field according to the following:

    • Bind to: Object Controller

    • Controller Key: selection

    • Model Key Path: personName

Next, Control-drag a line from File's Owner in the nib document window (File's Owner in this case represents the global `NSApplication` object) to the Controller object and then select `delegate` in the connection window. As you'll see, the application delegate (Controller) plays a role in saving the property list to its XML representation.

For the table view, Control-drag a line from the table view to the Controller object in the nib document window. Select the `dataSource` outlet in the connection window. Save the nib file. Copy the code in Listing 1-1 to `Controller.m`.

Listing 1-1   Implementation code for table view's data source

```objc
- (NSInteger)numberOfRowsInTableView:(NSTableView *)tableView {

    return self.phoneNumbers.count;

}


- (id)tableView:(NSTableView *)tableView
        objectValueForTableColumn:(NSTableColumn *)tableColumn
        row:(NSInteger)row {

    return [phoneNumbers objectAtIndex:row];

}


- (void)tableView:(NSTableView *)tableView setObjectValue:(id)object
        forTableColumn:(NSTableColumn *)tableColumn row:(NSInteger)row {

    [phoneNumbers replaceObjectsAtIndexes:[NSIndexSet indexSetWithIndex:row]
                withObjects:[NSArray arrayWithObject:object]];

}
```

Note that the last method synchronizes changes to items in the table view with the `phoneNumbers` mutable array that backs it.

Read in the Property List

Now that the necessary user-interface tasks are completed, we can focus on code that is specific to property lists. In its `init` method, the Controller object reads in the initial XML property list from the main bundle when the application is first launched; thereafter, it gets the property list from the user's `Documents` directory. Once it has the property list, it converts its elements into the corresponding property-list objects. Listing 1-2 shows how it does this.

Listing 1-2   Reading in and converting the XML property list

```
- (id) init {


    self = [super init];
    if (self) {
        NSString *errorDesc = nil;
        NSPropertyListFormat format;
        NSString *plistPath;
        NSString *rootPath =
[NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
            NSUserDomainMask, YES) objectAtIndex:0];
        plistPath = [rootPath stringByAppendingPathComponent:@"Data.plist"];
        if (![[NSFileManager defaultManager] fileExistsAtPath:plistPath]) {
            plistPath = [[NSBundle mainBundle] pathForResource:@"Data"
ofType:@"plist"];
        }
      NSData *plistXML = [[NSFileManager defaultManager] contentsAtPath:plistPath];
        NSDictionary *temp = (NSDictionary *)[NSPropertyListSerialization
            propertyListFromData:plistXML
            mutabilityOption:NSPropertyListMutableContainersAndLeaves
            format:&format
            errorDescription:&errorDesc];
        if (!temp) {
            NSLog(@"Error reading plist: %@, format: %d", errorDesc, format);
        }
        self.personName = [temp objectForKey:@"Name"];
        self.phoneNumbers = [NSMutableArray arrayWithArray:[temp
objectForKey:@"Phones"]];
```

```
        }
        return self;
    }
```

This code first gets the file-system path to the file containing the XML property list (`Data.plist`) in the `~/Documents` directory. If there is no file by that name at that location, it obtains the property-list file from the application's main bundle. Then it uses the `NSFileManager` method `contentsAtPath:` to read the property list into memory as an `NSData` object. After that, it calls the `NSPropertyListSerialization` class method `propertyListFromData:mutabilityOption:format:errorDescription:` to convert the static property list into the corresponding property-list objects—specifically, a dictionary containing a string and an array of strings. It assigns the string and the array of strings to the appropriate properties of the Controller object.

Write Out the Property List

When the user quits the application, you want to save the current values of the `personName` and `phoneNumbers` properties in a dictionary object, convert those property-list objects to a static XML representation, and then write that XML data to a file in `~/Documents`. The `applicationShouldTerminate:` delegate method of `NSApplication` is the appropriate place to write the code shown in Listing 1-3.

Listing 1-3   Converting and writing the property list to the application bundle

```
 - (NSApplicationTerminateReply)applicationShouldTerminate:(NSApplication *)sender
 {
    NSString *error;
    NSString *rootPath = [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
 NSUserDomainMask, YES) objectAtIndex:0];
    NSString *plistPath = [rootPath stringByAppendingPathComponent:@"Data.plist"];
    NSDictionary *plistDict = [NSDictionary dictionaryWithObjects:
            [NSArray arrayWithObjects: personName, phoneNumbers, nil]
            forKeys:[NSArray arrayWithObjects: @"Name", @"Phones", nil]];
    NSData *plistData = [NSPropertyListSerialization dataFromPropertyList:plistDict
                        format:NSPropertyListXMLFormat_v1_0
                        errorDescription:&error];
    if(plistData) {
        [plistData writeToFile:plistPath atomically:YES];
    }
```

```
    else {

        NSLog(error);

        [error release];

    }

    return NSTerminateNow;

}
```

This code creates an `NSDictionary` object containing the values of the `personName` and `phoneNumbers` properties and associates these with the keys "Name" and "Phones". Then, using the `dataFromPropertyList:format:errorDescription:` class method of `NSPropertyListSerialization`, it converts this top-level dictionary and the other property-list objects it contains into XML data. Finally, it writes the XML data to `Data.plist` in the user's `Documents` directory.

Run and Test the Application

Build and run the application. The window appears with the name and phone numbers you specified in the XML property list. Modify the name and a phone number and quit the application. Find the `Data.plist` file in `~/Documents` and open it in a text editor. You'll see that the changes you made in the user interface are reflected in the XML property list. If you launch the application again, it displays the changed values.

About Property Lists

Objective-CSwift

A property list is a structured data representation used by Cocoa and Core Foundation as a convenient way to store, organize, and access standard types of data. It is colloquially referred to as a "plist." Property lists are used extensively by applications and other software on OS X and iOS. For example, the OS X Finder—through bundles—uses property lists to store file and directory attributes. Applications on iOS use property lists in their Settings bundle to define the list of options displayed to users. This section explains what property lists are and when you should use them.

> **Note:** Although user and application preferences use property lists to structure and store data, you should not use the property list API of either Cocoa or Core Foundation to read and modify them. Instead, use the programming interfaces provided specifically for this purpose—see *Preferences and Settings Programming Guide* and *Preferences Programming Topics for Core Foundation* for more information.

What is a Property List?

Property lists are based on an abstraction for expressing simple hierarchies of data. The items of data in a property list are of a limited number of types. Some types are for primitive values and others are for containers of values. The primitive types are strings, numbers, binary data, dates, and Boolean values. The containers are arrays—indexed collections of values—and dictionaries—collections of values each identified by a key. The containers can contain other containers as well as the primitive types. Thus you might have an array of dictionaries, and each dictionary might contain other arrays and dictionaries, as well as the primitive types. A root property-list object is at the top of this hierarchy, and in almost all cases is a dictionary or an array. Note, however, that a root property-list object does not have to be a dictionary or array; for example, you could have a single string, number, or date, and that primitive value by itself can constitute a property list.

From the basic abstraction derives both a static representation of the property-list data and a runtime representation of the property list. The static representation of a property list, which is used for storage, can be either XML or binary data. (The binary version is a more compact form of the XML property list.) In XML, each type is represented by a certain element. The runtime representation of a property list is based on objects corresponding to the abstract types. The objects can be Cocoa or Core Foundation objects. Table 2-1 lists the types and their corresponding static and runtime representations.

Table 2-1   Property list types and their various representations

Abstract type	XML element	Cocoa class	Core Foundation type
array	`<array>`	`NSArray`	`CFArray` (`CFArrayRef`)
dictionary	`<dict>`	`NSDictionary`	`CFDictionary` (`CFDictionaryRef`)
string	`<string>`	`NSString`	`CFString` (`CFStringRef`)
data	`<data>`	`NSData`	`CFData` (`CFDataRef`)
date	`<date>`	`NSDate`	`CFDate` (`CFDateRef`)
number - integer	`<integer>`	`NSNumber`(`intValue`)	`CFNumber` (`CFNumberRef`, integer value)
number - floating point	`<real>`	`NSNumber` (`floatValue`)	`CFNumber` (`CFNumberRef`, floating-point value)
Boolean	`<true/>` or `<false/>`	`NSNumber` (`boolValue == YES` or `boolValue == NO`)	`CFBoolean` (`CFBooleanRef`; `kCFBooleanTrue` or `kCFBooleanFalse`)

By convention, each Cocoa and Core Foundation object listed in Table 2-1 is called a *property-list object*. Conceptually, you can think of "property list" as being an abstract superclass of all these classes. If you receive a property list object from some method or function, you know that it must be an instance of one of these types, but *a priori* you may not know which type. If a property-list object is a container (that is, an array or dictionary), all objects contained within it must also be property-list objects. If an array or dictionary contains objects that are not property-list objects, then you cannot save and restore the hierarchy of data using the various property-list methods and functions. And although `NSDictionary` and CFDictionary objects allow their keys to be objects of any type, if the keys are not string objects, the collections are not property-list objects.

Because all these types can be automatically cast to and from their corresponding Cocoa types, you can use the Core Foundation property list API with Cocoa objects. In most cases, however, methods provided by the `NSPropertyListSerialization` class should provide enough flexibility.

When to Use Property Lists

Many applications require a mechanism for storing information that will be needed at a later time. For situations where you need to store small amounts of persistent data—say less than a few hundred kilobytes—property lists offer a uniform and convenient means of organizing, storing, and accessing the data.

In some situations, the property-list architecture may prove insufficient. If you need a way to store large, complex graphs of objects, objects not supported by the property-list architecture, or objects whose mutability settings must be retained, use archiving. See *Archives and Serializations Programming Guide* for more information.

If you are looking for a way to implement user or application preferences, Cocoa provides a class specifically for this purpose. While the user defaults system does use property lists to store information, you do not have to access these plists directly. See *Preferences and Settings Programming Guide* and *Preferences Programming Topics for Core Foundation* for more information.

Note that property lists should be used for data that consists primarily of strings and numbers. They are very inefficient when used with large blocks of binary data.

Property List Representations

A property list can be stored in one of three different ways: in an XML representation, in a binary format, or in an "old-style" ASCII format inherited from OpenStep. You can serialize property lists in the XML and binary formats. The serialization API with the old-style format is read-only.

XML property lists are more portable than the binary alternative and can be manually edited, but binary property lists are much more compact; as a result, they require less memory and can be read and written much faster than XML property lists. In general, if your property list is relatively small, the benefits of XML property lists outweigh the I/O speed and compactness that comes with binary property lists. If you have a large data set, binary property lists, keyed archives, or custom data formats are a better solution.

Creating Property Lists Programmatically

You can create a graph of property-list objects by nesting property-list objects of various types in arrays and dictionaries. The following sections give details on doing this programmatically.

Creating a Property List in Objective-C

You can create a property list in Objective-C if all of the objects in the aggregate derive from the NSDictionary, NSArray, NSString, NSDate, NSData, or NSNumber class. The code in Listing 3-1 creates a property list consisting of an NSDictionary object (the root object) that contains two dictionaries, each containing a string, a date, and an array of numbers.

Listing 3-1   Creating a property list programmatically (Objective-C)

```
        NSMutableDictionary *rootObj = [NSMutableDictionary
dictionaryWithCapacity:2];
        NSDictionary *innerDict;
        NSString *name;
        NSDate *dob;
        NSArray *scores;

        scores = [NSArray arrayWithObjects:[NSNumber numberWithInt:6],
            [NSNumber numberWithFloat:4.6], [NSNumber numberWithLong:6.0000034],
nil];
        name = @"George Washington";
        dob = [NSDate dateWithString:@"1732-02-17 04:32:00 +0300"];
        innerDict = [NSDictionary dictionaryWithObjects:
            [NSArray arrayWithObjects: name, dob, scores, nil]
            forKeys:[NSArray arrayWithObjects:@"Name", @"DOB", @"Scores"]];
        [rootObj setObject:innerDict forKey:@"Washington"];

        scores = [NSArray arrayWithObjects:[NSNumber numberWithInt:8],
            [NSNumber numberWithFloat:4.9],
```

```
        [NSNumber numberWithLong:9.003433], nil];

    name = @"Abraham Lincoln";

    dob = [NSDate dateWithString:@"1809-02-12 13:18:00 +0400"];

    innerDict = [NSDictionary dictionaryWithObjects:

        [NSArray arrayWithObjects: name, dob, scores, nil]

        forKeys:[NSArray arrayWithObjects:@"Name", @"DOB", @"Scores"]];

    [rootObj setObject:innerDict forKey:@"Lincoln"];


    id plist = [NSPropertyListSerialization dataFromPropertyList:(id)rootObj

        format:NSPropertyListXMLFormat_v1_0 errorDescription:&error];
```

Note: The `NSPropertyListSerialization` class method shown in this example serializes the property-list objects into an XML property list. It is described more fully in Serializing a Property List (page 27).

The XML output of the code in Listing 3-1 is shown in Listing 3-2.

Listing 3-2   XML property list produced as output

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">

<plist version="1.0">

<dict>

    <key>Lincoln</key>

    <dict>

        <key>DOB</key>

        <date>1809-02-12T09:18:00Z</date>

        <key>Name</key>

        <string>Abraham Lincoln</string>

        <key>Scores</key>

        <array>

            <integer>8</integer>

            <real>4.9000000953674316</real>

            <integer>9</integer>

        </array>
```

```
        </dict>
        <key>Washington</key>
        <dict>
            <key>DOB</key>
            <date>1732-02-17T01:32:00Z</date>
            <key>Name</key>
            <string>George Washington</string>
            <key>Scores</key>
            <array>
                <integer>6</integer>
                <real>4.5999999046325684</real>
                <integer>6</integer>
            </array>
        </dict>
    </dict>
</plist>
```

Creating a Property List in Core Foundation

The examples in this section demonstrate how to create and work with property lists using Core Foundation functions. The error checking code has been removed for clarity. In practice, it is *vital* that you check for errors because passing bad parameters into Core Foundation routines can cause your application to crash.

Listing 3-3 shows you how to create a very simple property list—an array of CFString objects.

Listing 3-3   Creating a simple property list from an array

```
#include <CoreFoundation/CoreFoundation.h>
#define kNumFamilyMembers 5


void main () {
    CFStringRef names[kNumFamilyMembers];
    CFArrayRef  array;
    CFDataRef   xmlData;

```

```
    // Define the family members.
    names[0] = CFSTR("Marge");
    names[1] = CFSTR("Homer");
    names[2] = CFSTR("Bart");
    names[3] = CFSTR("Lisa");
    names[4] = CFSTR("Maggie");

    // Create a property list using the string array of names.
    array = CFArrayCreate( kCFAllocatorDefault,
            (const void **)names,
            kNumFamilyMembers,
            &kCFTypeArrayCallBacks );

    // Convert the plist into XML data.
    xmlData = CFPropertyListCreateXMLData( kCFAllocatorDefault, array );

    // Clean up CF types.
    CFRelease( array );
    CFRelease( xmlData );
}
```

Note: The `CFPropertyListCreateXMLData` function is discussed in Serializing a Property List (page 27).

Listing 3-4 shows how the contents of `xmlData`, created in Listing 3-3, would look if printed to the screen.

Listing 3-4   XML created by the sample program

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
        "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
    <string>Marge</string>
    <string>Homer</string>
```

```
    <string>Bart</string>

    <string>Lisa</string>

    <string>Maggie</string>

</array>

</plist>
```

About Numbers and Property Lists in Core Foundation

You cannot use C numeric data values directly in Core Foundation property lists. Core Foundation provides the function `CFNumberCreate` to convert C numerical values into CFNumber objects, the form that is required to use numbers in property lists.

A CFNumber object serves simply as a wrapper for C numeric values. Core Foundation includes functions to create a CFNumber, obtain its value, and compare two CFNumber objects. Note that CFNumber objects are immutable with respect to value, but type information may not be maintained. You can get information about a CFNumber object's type, but this is the type the CFNumber object used to store your value and *may not* be the same type as the original C data.

When comparing CFNumber objects, conversion and comparison follow human expectations and not C promotion and comparison rules. Negative zero compares less than positive zero. Positive infinity compares greater than everything except itself, to which it compares equal. Negative infinity compares less than everything except itself, to which it compares equal. Unlike standard practice, if both numbers are NaNs, then they compare equal; if only one of the numbers is a NaN, then the NaN compares greater than the other number if it is negative, and smaller than the other number if it is positive.

Listing 3-5 shows how to create a CFNumber object from a 16-bit integer and then get information about the CFNumber object.

Listing 3-5   Creating a CFNumber object from an integer

```
Int16           sint16val = 276;

CFNumberRef     aCFNumber;

CFNumberType    type;

Int32           size;

Boolean         status;


// Make a CFNumber from a 16-bit integer.
```

```
aCFNumber = CFNumberCreate(kCFAllocatorDefault,

                            kCFNumberSInt16Type,

                            &sint16val);


// Find out what type is being used by this CFNumber.

type = CFNumberGetType(aCFNumber);


// Now find out the size in bytes.

size = CFNumberGetByteSize(aCFNumber);


// Get the value back from the CFNumber.

status = CFNumberGetValue(aCFNumber,

                            kCFNumberSInt16Type,

                            &sint16val);
```

Listing 3-6 creates another CFNumber object and compares it with the one created in Listing 3-5.

Listing 3-6   Comparing two CFNumber objects

```
CFNumberRef         anotherCFNumber;
CFComparisonResult  result;


// Make a new CFNumber.
sint16val = 382;
anotherCFNumber = CFNumberCreate(kCFAllocatorDefault,

                        kCFNumberSInt16Type,

                        &sint16val);


// Compare two CFNumber objects.
result = CFNumberCompare(aCFNumber, anotherCFNumber, NULL);


switch (result) {
    case kCFCompareLessThan:
        printf("aCFNumber is less than anotherCFNumber.\n");
        break;
    case kCFCompareEqualTo:
```

```
        printf("aCFNumber is equal to anotherCFNumber.\n");

        break;

    case kCFCompareGreaterThan:

        printf("aCFNumber is greater than anotherCFNumber.\n");

        break;

}
```

Understanding XML Property Lists

The preferred way to store property lists on OS X and iOS is as an XML file called an *XML property list* or *XML plist*. These files have the advantages of being human-readable and in the standards-based XML format. The `NSArray` and `NSDictionary` classes both have methods for saving themselves as XML plists (for example, `descriptionWithLocale:` and `writeToFile:atomically:`), and also have methods to convert XML property lists back to objects in memory. CFPropertyList provides functions for converting property lists to and from their XML representation.

Core Foundation supports XML as the exclusive medium for the static representation of property lists on disk. Cocoa allows property lists to be stored on disk as XML property lists, in binary form, and as "old-style" property lists. The old-style property lists can only be read, not written; see Old-Style ASCII Property Lists (page 41) for further information.

Generally, there is little need to create or edit XML property yourself, but if you do, use Xcode's built-in property list editor or the Property List Editor application (which is part of the tools package). You should not edit the XML data in a text editor unless you are very familiar with XML syntax and the format of property lists. Moreover, the elements in an XML property list could change in future releases, so keep that in mind.

Even if you don't edit XML property lists, it is useful to understand their structure for design and debugging purposes. Like every XML file, XML plists begin with standard header information, and contain one root object, wrapped with the `<plist>` document type tag. The `<plist>` object also contains exactly one object, denoted by one of the XML elements listed in Table 2-1 (page 16).

Graphs of objects are created by nesting the XML elements listed in Table 2-1. When encoding dictionaries, the element `<key>` is used for dictionary keys, and one of the other property list tags is used for the key's value. Here is an example of XML data generated from a property list:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist SYSTEM "file://localhost/System/Library/DTDs/PropertyList.dtd">
<plist version="1.0">
<dict>
    <key>Author</key>
    <string>William Shakespeare</string>
    <key>Lines</key>
    <array>
```

```
        <string>It is a tale told by an idiot,</string>

        <string>Full of sound and fury, signifying nothing.</string>

    </array>

    <key>Birthdate</key>

    <integer>1564</integer>

</dict>

</plist>
```

Note that data bytes are base-64 encoded between the <data> and </data> tags.

Serializing a Property List

Using the `NSPropertyListSerialization` class or Property List Services (Core Foundation), you can serialize property lists in their runtime (object) form to a static representation that can be stored in the file system; later you can deserialize that static representation back into the original property-list objects. Property-list serialization automatically takes account of endianness on different processor architectures—for example, you can correctly read on an Intel-based Macintosh a binary property list created on a PowerPC-based Macintosh.

The property-list serialization APIs allow you to save graphs of property-list objects as binary data as well as XML property lists. See Property List Representations (page 17) for the relative advantages and disadvantages of XML and binary property lists.

Saving and Restoring a Property List in Objective-C

The `NSPropertyListSerialization` class (available in OS X v10.2 and later) provides methods for saving and restoring property lists from the two major supported formats, XML and binary. To save a property list in XML format, call the `dataFromPropertyList:format:errorDescription:` method, specifying `NSPropertyListXMLFormat_v1_0` as the second parameter; to save in binary format, specify `NSPropertyListBinaryFormat_v1_0` instead.

Listing 5-1 saves an object graph of property-list objects as an XML property list in the application bundle.

Listing 5-1   Saving a property list as an XML property list (Objective-C)

```
id plist;         // Assume this property list exists.
NSString *path = [[NSBundle mainBundle] pathForResource:@"Data" ofType:@"plist"];
NSData *xmlData;
NSString *error;

xmlData = [NSPropertyListSerialization dataFromPropertyList:plist
                                       format:NSPropertyListXMLFormat_v1_0
                                       errorDescription:&error];
if(xmlData) {
    NSLog(@"No error creating XML data.");
```

```
    [xmlData writeToFile:path atomically:YES];
}
else {
    NSLog(error);
    [error release];
}
```

> **Note:** To avoid a memory leak, it is necessary to release the error-description string returned via indirection in the third parameter of `dataFromPropertyList:format:errorDescription:` (as shown in Listing 5-1). This is a rare exception to the standard memory-management rules.

Because you cannot save a property list in the old-style (OpenStep) format, the only valid format parameters for this method are `NSPropertyListXMLFormat_v1_0` and `NSPropertyListBinaryFormat_v1_0`. The `NSData` object returned by `dataFromPropertyList:format:errorDescription:` encapsulates the XML or binary data. You can then call the `writeToFile:atomically:` or `writeToURL:atomically:` method to store the data in the file system.

> **Note:** If the root property-list object is an `NSDictionary` or `NSArray` object (the typical case), you can serialize the property list as an XML property list *and* write it out to disk at the same time using the appropriate methods of those classes. See Reading and Writing Property-List Data (page 37) for details.

To restore a property list from a data object by deserializing it, call the `propertyListFromData:mutabilityOption:format:errorDescription:` class method of the `NSPropertyListSerialization` class, passing in the data object. Listing 5-2 creates an immutable property list from the file at `path`:

Listing 5-2   Restoring a property list (Objective-C)

```
NSString *path = [[NSBundle mainBundle] pathForResource:@"Data" ofType:@"plist"];
NSData *plistData = [NSData dataWithContentsOfFile:path];
NSString *error;
NSPropertyListFormat format;
id plist;
```

```
plist = [NSPropertyListSerialization propertyListFromData:plistData

                             mutabilityOption:NSPropertyListImmutable

                             format:&format

                             errorDescription:&error];

if(!plist){

    NSLog(error);

    [error release];

}
```

The mutability-option parameter of the deserialization method controls whether the deserialized property-list objects are created as mutable or immutable. You can specify that all objects are immutable, that only the container objects are mutable, or that all objects are mutable. Unless you have a specific reason to mutate the collections and other objects in a deserialized property list, use the immutable option. It is faster and uses less memory.

The last two parameters of `propertyListFromData:mutabilityOption:format:errorDescription:` are by-reference. On return from a call, the format parameter holds a constant indicating the on-disk format of the property list: `NSPropertyListXMLFormat_v1_0`, `NSPropertyListBinaryFormat_v1_0`, or `NSPropertyListOpenStepFormat`. You may pass in `NULL` if you are not interested in the format.

If the method call returns `nil`, the final parameter—the error-description string—states the reason the deserialization did not succeed.

> **Note:** In OS X v10.5 (and earlier versions of the operating system), it is necessary to release the error-description string in Listing 5-2.

Saving and Restoring a Property List in Core Foundation

Property List Services of Core Foundation has serialization functions corresponding to the class methods of `NSPropertyListSerialization` described in Saving and Restoring a Property List in Objective-C. To create an XML property list from a property list object, call the `CFPropertyListCreateXMLData` function. To restore a property list object from XML data, call the `CFPropertyListCreateFromXMLData` function.

Listing 5-3 shows you how to create a complex property list, convert it to XML, write it to disk, and then re-create the original data structure using the saved XML. For more information about using CFDictionary objects see *Collections Programming Topics for Core Foundation* .

Listing 5-3   Saving and restoring property list data (Core Foundation)

```c
#include <CoreFoundation/CoreFoundation.h>


#define kNumKids 2

#define kNumBytesInPic 10


CFDictionaryRef CreateMyDictionary( void );

CFPropertyListRef CreateMyPropertyListFromFile( CFURLRef fileURL );

void WriteMyPropertyListToFile( CFPropertyListRef propertyList,

            CFURLRef fileURL );


int main () {

    CFPropertyListRef propertyList;

    CFURLRef fileURL;


    // Construct a complex dictionary object;

    propertyList = CreateMyDictionary();


    // Create a URL that specifies the file we will create to

    // hold the XML data.

    fileURL = CFURLCreateWithFileSystemPath( kCFAllocatorDefault,

                CFSTR("test.txt"),       // file path name

                kCFURLPOSIXPathStyle,    // interpret as POSIX path

                false );                 // is it a directory?


    // Write the property list to the file.

    WriteMyPropertyListToFile( propertyList, fileURL );

    CFRelease(propertyList);


    // Recreate the property list from the file.

    propertyList = CreateMyPropertyListFromFile( fileURL );


    // Release any objects to which we have references.

    CFRelease(propertyList);

    CFRelease(fileURL);
```

```
    return 0;

}


CFDictionaryRef CreateMyDictionary( void ) {

    CFMutableDictionaryRef dict;

    CFNumberRef            num;

    CFArrayRef             array;

    CFDataRef              data;


    int                    yearOfBirth;

    CFStringRef            kidsNames[kNumKids];


    // Fake data to stand in for a picture of John Doe.

    const unsigned char pic[kNumBytesInPic] = {0x3c, 0x42, 0x81,

            0xa5, 0x81, 0xa5, 0x99, 0x81, 0x42, 0x3c};


    // Define some data.

    kidsNames[0] = CFSTR("John");

    kidsNames[1] = CFSTR("Kyra");


    yearOfBirth = 1965;


    // Create a dictionary that will hold the data.

    dict = CFDictionaryCreateMutable( kCFAllocatorDefault,

            0,

            &kCFTypeDictionaryKeyCallBacks,

            &kCFTypeDictionaryValueCallBacks );


    // Put the various items into the dictionary.

    // Because the values are retained as they are placed into the

    //  dictionary, we can release any allocated objects here.


    CFDictionarySetValue( dict, CFSTR("Name"), CFSTR("John Doe") );


    CFDictionarySetValue( dict,
```

```
            CFSTR("City of Birth"),

            CFSTR("Springfield") );


    num = CFNumberCreate( kCFAllocatorDefault,

            kCFNumberIntType,

            &yearOfBirth );
    CFDictionarySetValue( dict, CFSTR("Year Of Birth"), num );

    CFRelease( num );


    array = CFArrayCreate( kCFAllocatorDefault,

              (const void **)kidsNames,

              kNumKids,

              &kCFTypeArrayCallBacks );
    CFDictionarySetValue( dict, CFSTR("Kids Names"), array );

    CFRelease( array );


    array = CFArrayCreate( kCFAllocatorDefault,

              NULL,

              0,

              &kCFTypeArrayCallBacks );
    CFDictionarySetValue( dict, CFSTR("Pets Names"), array );

    CFRelease( array );


    data = CFDataCreate( kCFAllocatorDefault, pic, kNumBytesInPic );

    CFDictionarySetValue( dict, CFSTR("Picture"), data );

    CFRelease( data );


    return dict;

}


void WriteMyPropertyListToFile( CFPropertyListRef propertyList,

          CFURLRef fileURL ) {

    CFDataRef xmlData;

    Boolean status;
```

```
    SInt32 errorCode;


    // Convert the property list into XML data.
    xmlData = CFPropertyListCreateXMLData( kCFAllocatorDefault, propertyList );


    // Write the XML data to the file.
    status = CFURLWriteDataAndPropertiesToResource (
                fileURL,                    // URL to use
                xmlData,                    // data to write
                NULL,
                &errorCode);


    CFRelease(xmlData);
}


CFPropertyListRef CreateMyPropertyListFromFile( CFURLRef fileURL ) {
    CFPropertyListRef propertyList;
    CFStringRef       errorString;
    CFDataRef         resourceData;
    Boolean           status;
    SInt32            errorCode;


    // Read the XML file.
    status = CFURLCreateDataAndPropertiesFromResource(
                kCFAllocatorDefault,
                fileURL,
                &resourceData,              // place to put file data
                NULL,
                NULL,
                &errorCode);


    // Reconstitute the dictionary using the XML data.
    propertyList = CFPropertyListCreateFromXMLData( kCFAllocatorDefault,
                resourceData,
                kCFPropertyListImmutable,
```

```
            &errorString);


    if (resourceData) {

        CFRelease( resourceData );

     } else {

        CFRelease( errorString );

     }

    return propertyList;

 }
```

For a discussion of the mutability-option parameter of `CFPropertyListCreateFromXMLData` see the discussion of the corresponding parameter of the `propertyListFromData:mutabilityOption:format:errorDescription:` method in Saving and Restoring a Property List in Objective-C (page 27).

Listing 5-4 shows how the contents of `xmlData`, created in Listing 5-1 (page 27), would look if printed to the screen.

Listing 5-4   XML file contents created by the sample program

```
<?xml version="1.0" encoding="UTF−8"?>

<!DOCTYPE plist PUBLIC "−//Apple Computer//DTD PLIST 1.0//EN"

        "http://www.apple.com/DTDs/PropertyList−1.0.dtd">

<plist version="1.0">

<dict>

    <key>Year Of Birth</key>

    <integer>1965</integer>

    <key>Pets Names</key>

    <array/>

    <key>Picture</key>

    <data>

        PEKBpYGlmYFCPA==

    </data>

    <key>City of Birth</key>

    <string>Springfield</string>

    <key>Name</key>
```

```
    <string>John Doe</string>

    <key>Kids Names</key>

    <array>

        <string>John</string>

        <string>Kyra</string>

    </array>

</dict>

</plist>
```

Using Property List Services with Cocoa

Cocoa uses the Core Foundation property list API to read and write XML property lists. In some cases, you may wish to access the API directly in an Objective-C Cocoa application. For example, if you want to save an instance of a class other than `NSArray` or `NSDictionary` as the root object of an XML plist, currently the easiest way to do this is through Property List Services. This process is made simple because Cocoa objects can be cast to and from corresponding Core Foundation types. This conversion between Cocoa and Core Foundation object types is known as *toll-free bridging*.

To create an XML property list from a property list object, call the `CFPropertyListCreateXMLData` function. This code fragment saves the property list `plist` into a file at `path`:

```
NSString *path = [NSString stringWithFormat:@"%@/MyData.plist",
NSTemporaryDirectory()];

id plist;        // Assume this is a valid property list.

NSData *xmlData;


xmlData = (NSData *)CFPropertyListCreateXMLData(kCFAllocatorDefault,
                                        (CFPropertyListRef)plist);

[xmlData writeToFile:path atomically:YES];

[xmlData release];
```

To restore a property list object from XML data, call the `CFPropertyListCreateFromXMLData` function. This code fragment restores property list from the XML plist file at `path` with mutable containers but immutable leaves:

```
NSString *path = [NSString stringWithFormat:@"%@/MyData.plist",
NSTemporaryDirectory()];

NSString *errorString;

NSData *xmlData;

id plist;


xmlData = [NSData dataWithContentsOfFile:path];

plist = (id)CFPropertyListCreateFromXMLData(kCFAllocatorDefault,

               (CFDataRef)xmlData, kCFPropertyListMutableContainers,

               (CFStringRef *)&errorString);
```

Reading and Writing Property-List Data

Using Objective-C Methods to Read and Write Property-List Data

You have two major ways to write property-list data to the file system:

- If the root object of the property list is an `NSDictionary` or `NSArray` object—which is almost always the case—you can invoke the `writeToFile:atomically:` or `writeToURL:atomically:` methods of those classes, passing in the root object. These methods save the graph of property-list objects as an XML property list before writing that out as a file or URL resource.

  To read the property-list data back into your program, initialize an allocated collection object by calling the `initWithContentsOfFile:` and `initWithContentsOfURL:` methods or the corresponding class factory methods (for example, `dictionaryWithContentsOfURL:`).

  > **Note:** To function properly, these methods require that *all* objects contained by the `NSDictionary` or `NSArray` root object be property-list objects.

- You can serialize the property-list objects to an `NSData` object using the `dataFromPropertyList:format:errorDescription:` class method and then save that object by calling the `writeToFile:atomically:` or `writeToURL:atomically:` methods of the `NSData` class.

  To read the property-list data back into your program, first initialize an allocated `NSData` object by invoking `initWithContentsOfFile:` or `initWithContentsOfURL:` or call a corresponding class factory method such as `dataWithContentsOfFile:`. Then call the `propertyListFromData:mutabilityOption:format:errorDescription:` class method of `NSPropertyListSerialization`, passing in the data object.

  > **Note:** The code examples in Read in the Property List (page 12) and Write Out the Property List (page 13) of Quick Start for Property Lists illustrate the second approach.

The first approach is simpler—it requires only one method invocation instead of two—but the second approach has its advantages. It allows you to convert the runtime property list to binary format as well as an XML property list. When you convert a static representation of a property list back into a graph of objects, it also lets you specify with more flexibility whether those objects are mutable or immutable.

To expand on this last point, consider this example. You have an XML property list whose root object is an NSArray object containing a number of NSDictionary objects. If you load that property list with this call:

```
NSArray * a = [NSArray arrayWithContentsOfFile:xmlFile];
```

a is an immutable array with immutable dictionaries in each element. Each key and each value in each dictionary are also immutable.

If you load the property list with this call:

```
NSMutableArray * ma = [NSMutableArray arrayWithContentsOfFile:xmlFile];
```

ma is a mutable array with immutable dictionaries in each element. Each key and each value in each dictionary are immutable.

If you need finer-grained control over the mutability of the objects in a property list, use the propertyListFromData:mutabilityOption:format:errorDescription: class method, whose second parameter permits you to specify the mutability of objects at various levels of the aggregate property list. You could specify that all objects are immutable (NSPropertyListImmutable), that only the container (array and dictionary) objects are mutable (NSPropertyListMutableContainers), or that all objects are mutable (NSPropertyListMutableContainersAndLeaves).

For example, you could write code like this:

```
NSMutableArray *dma = (NSMutableArray *)[NSPropertyListSerialization
                   propertyListFromData:plistData
                   mutabilityOption:NSPropertyListMutableContainersAndLeaves
                   format:&format
                   errorDescription:&error];
```

This call produces a mutable array with mutable dictionaries in each element. Each key and each value in each dictionary are themselves also mutable.

Using Core Foundation Functions to Read and Write Property-List Data

To write out XML property lists using Property List Services (Core Foundation), call the function the `CFURLWriteDataAndPropertiesToResource` function, passing the CFData object created through calling `CFPropertyListCreateXMLData`. To read an XML property list from the file system or URL resource, call the function `CFURLCreateDataAndPropertiesFromResource`. Then convert the created CFData object to a graph of property-list objects by calling the `CFPropertyListCreateFromXMLData` function.

Listing 6-1 includes a fragment of the larger code example in Saving and Restoring a Property List in Core Foundation (page 29) that illustrates the use of these functions.

Listing 6-1   Writing and reading property lists using Core Foundation functions

```
void WriteMyPropertyListToFile( CFPropertyListRef propertyList,
            CFURLRef fileURL ) {
    CFDataRef xmlData;
    Boolean status;
    SInt32 errorCode;

    // Convert the property list into XML data.
    xmlData = CFPropertyListCreateXMLData( kCFAllocatorDefault, propertyList );

    // Write the XML data to the file.
    status = CFURLWriteDataAndPropertiesToResource (
                fileURL,                    // URL to use
                xmlData,                    // data to write
                NULL,
                &errorCode);

    CFRelease(xmlData);
}


CFPropertyListRef CreateMyPropertyListFromFile( CFURLRef fileURL ) {
    CFPropertyListRef propertyList;
    CFStringRef       errorString;
    CFDataRef         resourceData;
    Boolean           status;
```

```
    SInt32           errorCode;


    // Read the XML file.
    status = CFURLCreateDataAndPropertiesFromResource(
            kCFAllocatorDefault,
            fileURL,
            &resourceData,          // place to put file data
            NULL,
            NULL,
            &errorCode);


    // Reconstitute the dictionary using the XML data.
    propertyList = CFPropertyListCreateFromXMLData( kCFAllocatorDefault,
            resourceData,
            kCFPropertyListImmutable,
            &errorString);


    if (resourceData) {
        CFRelease( resourceData );
     else {
        CFRelease( errorString );
     }
    return propertyList;
}
```

You may also write and read property lists to the file system using the functions
CFPropertyListWriteToStream and CFPropertyListCreateFromStream. These functions require that
you open and configure the read and write streams yourself.

Old-Style ASCII Property Lists

The OpenStep frameworks, which Cocoa is derived from, used an ASCII format for storing property lists. These files store information equivalent to the information in XML property lists, and are still supported by Cocoa, but for reading only. Old-style plist support remains primarily for legacy reasons. You read old-style property lists by calling the `propertyListFromData:mutabilityOption:format:errorDescription:` method of `NSPropertyListSerialization`.

ASCII property lists support the four primary property list data types: NSString, NSData, NSArray, and NSDictionary. The following sections describe the ASCII syntax for each of these types.

> **Important:** Cocoa allows you to read old-style ASCII property lists only. You may not write them.

NSString

A string is enclosed in double quotation marks, for example:

```
"This is a string"
```

The quotation marks can be omitted if the string is composed strictly of alphanumeric characters and contains no white space (numbers are handled as strings in property lists). Though the property list format uses ASCII for strings, note that Cocoa uses Unicode. Since string encodings vary from region to region, this representation makes the format fragile. You may see strings containing unreadable sequences of ASCII characters; these are used to represent Unicode characters.

NSData

Binary data is enclosed in angle brackets and encoded in hexadecimal ASCII. Spaces are ignored. For example:

```
<0fbd777 1c2735ae>
```

NSArray

An array is enclosed in parentheses, with the elements separated by commas. For example:

```
("San Francisco", "New York", "Seoul", "London", "Seattle", "Shanghai")
```

The items don't all have to be of the same type (for example, all strings)—but they normally are. Arrays can contain strings, binary data, other arrays, or dictionaries.

NSDictionary

A dictionary is enclosed in curly braces, and contains a list of keys with their values. Each key-value pair ends with a semicolon. For example:

```
{ user = wshakesp; birth = 1564; death = 1616; }
```

Note the omission of quotation marks for single-word alphanumeric strings. Values don't all have to be the same type, because their types are usually defined by whatever program uses them. Dictionaries can contain strings, binary data, arrays, and other dictionaries.

Below is a sample of a more complex property list. The property list itself is a dictionary with keys "AnimalSmells," "AnimalSounds," and so on; each value is also a dictionary, with key-value pairs.

```
{
    AnimalSmells = { pig = piggish; lamb = lambish; worm = wormy; };
    AnimalSounds = { pig = oink; lamb = baa; worm = baa;
                     Lisa = "Why is the worm talking like a lamb?"; };
    AnimalColors = { pig = pink; lamb = black; worm = pink; };
}
```

Document Revision History

This table describes the changes to *Property List Programming Guide*.

Date	Notes
2010-03-24	Update example code to new initializer pattern.
2009-07-30	Added links to inline Cocoa concepts and made minor modifications.
2008-11-19	Combined Cocoa and Core Foundation documents on property lists and reorganized material. Added a "Quick Start" mini-tutorial and a chapter on programmatically creating property lists. Mentioned iOS as a platform on which property lists are important.
2006-11-07	Updated deprecated method in code listing.
2006-06-28	Added an article describing how element mutability is determined when a property list is loaded from a file.
2005-08-11	Made minor revisions. Changed title from "Property Lists."
2003-06-30	Corrected error in restore property list sample code in "Using Old-Style Property Lists" and "Using XML Property Lists".
2003-05-02	"Serialized" property lists renamed "Binary" to better reflect their purpose.
2003-04-21	Corrected error in restore property list sample code in "Binary Property Lists".
2003-03-21	Enumerated NSArray and NSDictionary methods available for writing XML plists in "XML Property Lists". Added link to Core Foundation Property List documentation, and removed references to Property List "Services".

Date	Notes
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.