Working with Property Lists

This chapter describes how to work with property lists in the Yellow Box. It answers the following questions:

- What is a property list?
- What are the different ways of storing property lists?
- How do you implement each of the storage solutions?
- What methods does the Yellow Box provide for working with property lists?

What Is a Property List?

A property list is a mechanism used by the Yellow Box to provide a uniform way of organizing, storing, and accessing data.

Property lists organize data into named values and lists of values using four classes: NSDictionary, NSString, NSData, and NSArray. The four classes used in property lists give you the means to produce data that is meaningfully structured, transportable, storable, and accessible, but still as lightweight as possible. You represent basic data types (such as integers and text) with NSString, and binary data with NSData. You use NSDictionary and NSArray to build complex data structures. In an NSDictionary, data is structured as key-value pairs, where each key is a string and the key's value can be an NSString, an NSArray, an NSData, or another NSDictionary. In an NSArray, data is structured as a collection of objects that can be accessed by index. In a property list, an NSArray can contain NSStrings, NSDatas, NSDictionaries, and other NSArrays.

Note: NSDictionaries and NSArrays can contain data types other than NSStrings, NSDatas, NSDictionaries, and NSArrays, but if they do, they're not property lists—and consequently, you can't use the Yellow Box's property list API to work with them.

Ways of Storing Property Lists

A property list can be stored in one of three different ways:

- As an ASCII file
- In a serialized binary format
- As a persistent property list (NSPPL)

Each of these approaches has its advantages. For example, an ASCII property list is human-readable, but access is slow. Serialization, which stores property lists in a binary format, offers faster access than an ASCII property list and it’s also *lazy*, meaning that you can read parts of files without accessing the whole thing. But if you modify serialized data, you must reserialize the entire file.

Like serialization, a persistent property list (NSPPL) stores data in a binary format, provides fast access, and is lazy. It also allows you to make incremental changes (even when an NSPPL contains tens of megabytes of data), while still ensuring that your data is never corrupted. In this sense, an NSPPL is analogous to a database. Because of their ability to incrementally store and retrieve data, NSPPLs are well-suited for working with large amounts of data—that is, data that has several elements, that occupies a large number of bytes, or both.

The different ways of storing property lists are described in more detail in the following sections.

Working with ASCII Property Lists

The property list data types, NSString, NSArray, NSDictionary, and NSData, all know how read from and write to ASCII representations of themselves. This section describes the ASCII property list syntax, and gives a code example of how to work with ASCII property lists.

ASCII Property List Syntax

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

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 Yellow Box uses Unicode. 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", "London").
```

The items don't all have to be of the same type (for example, all strings)—but they normally should be. 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 = maryg; "error string" = "core dump"; code = <fead0007>; }.
```

Note the omission of quotation marks for single-word alphanumeric strings. Values don't all have to be the same type, since their types are usually defined by whatever program uses them. In this example, the program using the dictionary knows that the value for "user" is a string and the value for "code" is binary data. Dictionaries can contain strings, binary data, arrays, and other dictionaries.

Below is a sample of a more complex property list, taken from a user's defaults system. The property list itself is a dictionary with keys "Clock," "NSGlobalDomain," and so on; each value is also a dictionary, which contains the individual defaults.

```
{
  Clock = {ClockStyle = 3; };
  NSGlobalDomain = {24HourClock = Yes; Language = English; };
  NeXT1 = {Keymap = /NextLibrary/Keyboards/NeXTUSA; };
  Viewer = {NSBrowserColumnWidth = 145;
      "NSWindow Frame Preferences" = "5 197 395 309 "; };
  Workspace = {SelectedTabIndex = 0; WindowOrigin = "-75.000000"; };
  pbs = {};
}
```

Debugging ASCII Property Lists

The property lists you create using Yellow Box property list methods should be free from syntax errors. However, if you manually edit an existing ASCII property list or create one by hand, it's easy to introduce syntax errors that can be hard to track down. You can use the **pl** command line utility to parse an ASCII property list. **pl** converts an ASCII property list into a binary format and vice versa, and as a by-product of this conversion it flags ASCII syntax errors. For example, the following command takes the ASCII property list **myPlist.plist** and writes it to **aBinaryFile**:

```
pl -output aBinaryFile < myPlist.plist
```

If **pl** finds any syntax errors in the supplied ASCII property list, it prints out diagnostic error messages, for example:

```
*** Uncaught exception: <NSParseErrorException> *** Separator ‘}’
expected; Parse error line 4 (position 118) for units: (xObject, Array,
parseArray, xObject, Dictionary, personDictionary); next token is
‘children’
```

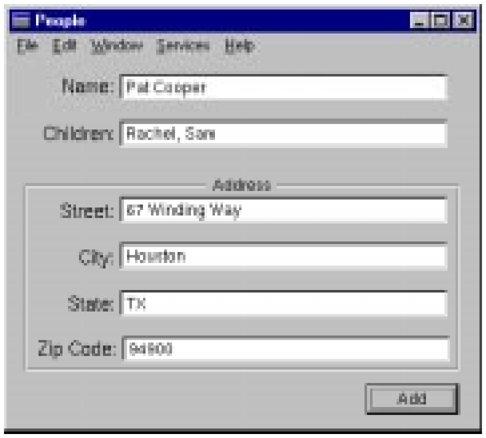
pl is available in ***NEXT_ROOT*/NextLibrary/Executables**.

Example of Using ASCII Property Lists

This section uses an example application, People, to illustrate some of the basic principals involved in working with ASCII property lists. The section shows how to:

- Use the property list data types in combination with each other
- Initialize an array from a string that has property list format
- Read and write ASCII property lists to disk

The user interface of the People application is shown in Figure 1.



Name: Pat Cooper
Children: Rachel, Sara
Street: 67 Winding Way
City: Houston
State: TX
Zip Code: 94900

Users enter data into the fields as shown. Clicking Add adds the new person record to the array of records and saves the array to disk.

Figure 1. *The People Application*

The People application takes user input, uses it to initialize a person record, and then adds the person record to an array. The array of all person records is then stored as an ASCII property list.

More specifically, when a user enters data into the window and clicks Add, the data is used to initialize the NSStrings, NSArrays, and NSDictionaries that constitute the person record. These objects are then placed in an NSDictionary, **personDict**. The array of all person records, **peopleArray**, is initialized from a file that stores data in an

ASCII property list format. The dictionary **personDict** is added to this array, and the array is then saved back out to the file.

This code excerpt shows how to take the user input and use it to initialize a person record:

```
/* Assume these exist. */
id nameField, childrenField,streetField,cityField,
     stateField, zipCodeField;
NSString *name;
NSMutableString *childString;
NSMutableDictionary *personDict, *addressDict;
NSArray *childArray;
NSMutableArray *peopleArray;

name = [nameField stringValue];
personDict = [NSMutableDictionary dictionaryWithCapacity:3];

/* Enclose childString in parentheses to put it in an
 * array property list format so that it can be used to
 * initialize an array.
 */
childString = [NSMutableString stringWithString:
     [childrenField stringValue]];
[childString insertString:@"(" atIndex:0];
[childString appendString:@")"];

/* Now that it’s been enclosed in parentheses, you can
 * use childString to initialize an array.
 */
childArray = [childString propertyList];

addressDict = [NSMutableDictionary dictionaryWithCapacity:4];

/* Add data to addressDict. */
[addressDict setObject:
     [streetField stringValue] forKey:@"street"];
[addressDict setObject:[cityField stringValue] forKey:@"city"];
[addressDict setObject:[stateField stringValue] forKey:@"state"];
[addressDict setObject:
     [zipCodeField stringValue] forKey:@"zipcode"];

[personDict setObject:name forKey:@"name"];
[personDict setObject:addressDict forKey:@"address"];
[personDict setObject:childArray forKey:@"children"];
```

Now that a new person record has been initialized, the record needs to be added to the array of all people records. The following code excerpt shows how to read the

ASCII property list **People.plist** and use it to initialize **peopleArray.** The code then adds the new record to **peopleArray**, and saves **peopleArray** to the file **People.plist**.

```
/* Read People.plist. If it doesn’t exist, create it. */
if(!(peopleArray = [NSMutableArray
     arrayWithContentsOfFile:@"D:/Katie/People.plist"]))
        peopleArray = [NSMutableArray array];

/* Add the dictionary initialized from data entered by the user
 * to peopleArray.
 */
[peopleArray addObject:personDict];

/* Write peopleArray back out to disk. */
[peopleArray writeToFile:@"D:/Katie/People.plist" atomically:YES];
```

The data in **People.plist** is stored in ASCII property list format:

```
(
    {
       address = {city = Napa; state = CA; street = "45 Chatsworth Rd.";
zipcode = 98609; };
        children = (Elise, Timothy, Claire);
        name = "Susanne Beutler";
    },
    {
        address = {city = Yardley; state = PA; street = "67 Sunset Rd.";
zipcode = 19067; };
        children = (Tom, Katie, Sarah);
        name = "Seth McCormick";
    },
    {
       address = {city = Falmouth; state = MA; street = "7 Palmer Ave.";
zipcode = 92300; };
        children = (Hannah);
        name = "Mike Bassett";
    },
     {
       address = {city = Houston; state = TX; street = "67 Winding Way";
zipcode = 94900; };
        children = (Rachel, Sam);
        name = "Pat Cooper";
    }
)
```

Working with Serialized Property Lists

As described in “Ways of Storing Property Lists” on page 1, ASCII property lists have the advantage of being human-readable. However, accessing them is slow. As an alternative to storing property lists in an ASCII format, you can serialize them using the NSSerializer class. The NSSerializer class provides a mechanism for creating an abstract representation of a property list. The NSSerializer class stores this representation in an NSData object in an architecture-independent format, so that property lists can be used with distributed applications. NSSerializer's companion class NSDeserializer declares methods that take the abstract representation and recreate the property list in memory. For more information on serialization, see the NSSerializer and NSDeserializer class specifications in the *Foundation Framework Reference*.

Serialization only works with NSArray, NSDictionary, NSString, and NSData. Any other class of objects (that is, non-property list data) cannot be archived by NSSerializer unless you implement the NSObjCTypeSerializationCallback protocol. For more information, see the NSObjCTypeSerializationCallback protocol specification in the *Foundation Framework Reference*.

NSSerializer vs. NSArchiver

Foundation provides two different ways of “archiving” data: NSArchiver, and NSSerializer. Serialization is generally the most appropriate choice for storage of things naturally structured as a property list. For example, suppose you’re storing time zone data, where each time zone includes the date of daylight savings time transitions and other information. All of this data could be stored in one file: a serialized property list in which the top-level object is a dictionary, the keys are time zone names, and the value for each key is a block of bytes (as an NSData) that contain the data in a standard UNIX format. Serialization would be the best choice in this case because it’s simpler than archiving, and because it’s also less tied to the particular way objects archive themselves. The serialization format is public and documented in the Yellow Box specification.

Other benefits of serialization are:

- It’s generally faster to serialize a tree of property list objects than it is to archive it.
- Serialized property list objects can be decoded lazily (and thus, as in the case of time zone data, if all of the objects aren’t needed, the cost of unarchiving them is not incurred). With archiving, you have to unarchive the whole archive.

Preserving Structural Information

In contrast to archiving (see the NSArchiver class specification), the serialization process preserves only structural information, not class information. Thus, 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. However, the structure and interrelationships of the data in the resulting property list are identical to that in the original, with one possible exception.

The exception is that when an object graph is serialized, the mutability of the container’s objects (NSDictionary and NSArray objects) is preserved only down to the highest node in the graph that has an immutable container. Thus, if an NSArray contains an NSMutableDictionary, the serialized version of this object graph would not preserve the mutability of the dictionary or the mutability of any objects it contains.

If preserving class and mutability information is more important to your application than speed and portability, you may want to use archiving (as implemented by NSCoder and NSArchiver) instead of serialization to make object graphs persistent.

Example of Using Serialization

This section shows you how to rewrite the sample application, People, to use a serialized property list.

```
/* This example adds the variable data */
id data;

/* ...
 * Create person record as shown in the section "Example of
 * Working with ASCII Property Lists."
 * ...
 */
```

```
/* If serialized property list already exists, deserialize
 * it and use it to initialize peopleArray. Otherwise, create
 * peopleArray.
 */
if(!(peopleArray = [NSMutableArray arrayWithArray:
  [NSDeserializer deserializePropertyListFromData:
  [NSData dataWithContentsOfFile:@"D:/Katie/People.splist"]
  mutableContainers:NO]]))
      peopleArray = [NSMutableArray array];

[peopleArray addObject:personDict];

/* Serialize the property list and write it to a file. */
data = [NSSerializer serializePropertyList:peopleArray];
[data writeToFile:@"D:/Katie/People.splist" atomically:NO];
```

Lazy Deserialization

The above example uses the **deserializePropertyListFromData:** method to deserialize an NSData object. You can alternatively use the **deserializePropertyListLazilyFromData:atCursor:length:mutableContainers:** method to lazily deserialize an NSData object:

```
unsigned int cursor = 0;
[NSDeserializer deserializePropertyListLazilyFromData:
      [NSData dataWithContentsOfFile:@"D:/Katie/People.splist"]
      atCursor:&cursor
      length:64
      mutableContainers:YES];
```

This method operates on the range of bytes starting at the specified cursor position, up to the specified length. If this range is longer than the threshold specified by NSDeserializer, a fault (stand-in) object is substituted for the property list represented by the bytes. Once you access the property list and the objects it contains, they're instantiated as "real" objects.

Working with Persistent Property Lists (NSPPLs)

The NSPPL (persistent property list) class allows you to incrementally store property lists in a binary format Like serialization, NSPPL provides fast access and is lazy.

Persistent property lists are atomic, meaning that if a **save** operation fails, the NSPPL reverts to its previously saved state. An NSPPL is never left in an intermediate state. Changes to an NSPPL are applied incrementally (in memory, but not to disk) as you make them. A **save** operation has the effect of committing the changes you've made to disk.

For more information on NSPPL, see the NSPPL class specification in the *Foundation Framework Reference*.

Example of Using NSPPL

This section shows you how to rewrite the sample application, People, to use an NSPPL. Note that you access the data in an NSPPL through a root dictionary.

```
/* This example adds the variables rootDict and ppl */
NSMutableDictionary *rootDict;
NSPPL  *ppl;

/* ...
 * Create person record as shown in the section "Example of
 * Working with ASCII Property Lists."
 * ...
 */

/* Read the NSPPL; if it doesn’t exist, create it. */
ppl = [NSPPL pplWithPath:@"D:/Katie/People.ppl"
         create:YES readOnly:NO];
if (!ppl) {
    NSLog(@"Couldn't open or create %@", pplPath);
    exit(1);
 }

/* Initialize peopleArray from ppl. */
if(!(peopleArray = [NSMutableArray arrayWithArray:
  [[ppl rootDictionary] objectForKey:@"peopleArray"]]))
     peopleArray = [NSMutableArray array];

[peopleArray addObject:personDict];

/* Through the root dictionary, add the updated peopleArray
 * to ppl.
 */
rootDict = [ppl rootDictionary];
[rootDict setObject:peopleArray forKey:@"peopleArray"];
[ppl save];
```

You don’t have to do anything special to access an NSPPL lazily or to save changes to it incrementally—NSPPL handles these things for you. Accessing an NSPPL is always lazy, and sending an NSPPL a **save** message only updates the parts of the NSPPL that have changed.

Yellow Box Property List Methods

This section summarizes the methods Yellow Box uses to operate on property lists.

propertyList Methods

NSString implements the **propertyList** method, which parses the receiver as a text representation of a property list, returning an NSString, NSData, NSArray, or NSDictionary object according to the topmost element. For example, this method is used in the People application to initialize an array from a string:

```
    childArray = [childString propertyList];
```

NSString also implements the **propertyListFromStringsFileFormat** method, which returns a dictionary object initialized with the keys and values found in the receiver. The receiver must contain text in the format used for **.strings** files. NSDictionary implements a complementary method, **descriptionInStringsFileFormat**, which returns a string that represents the contents of the receiver, formatted in **.strings** file format. The **...StringsFileFormat** methods are used for localization. For more information on the **.strings** file format, see the method description for **propertyListFromStringsFileFormat** in the NSString class specification.

description Methods

The four property list types, NSString, NSArray, NSDictionary, and NSData all implement a **description** method, which returns the contents of the receiver formatted as an ASCII property list (the property list format for each data type is described in "ASCII Property List Syntax" on page 2).

In addition, NSArray and NSDictionary implement the **descriptionWithLocale:** method, in which the locale specifies options for formatting each of the receiver's elements. NSArray and NSDictionary also implement a **descriptionWithLocale:indent:** method, which lets you specify indentation options for the returned string.

serialize and deserialize Methods

NSSerializer implements two methods for serializing property lists: **serializePropertyList:intoData:** and **serializePropertyList**. **serializePropertyList:intoData:** serializes a property list into an NSMutableData object. **serializePropertyList** creates an NSData object, serializes a property list into it, and returns the NSData object.

NSDeserializer implements the following methods for deserializing property lists:

- **deserializePropertyListFromData:atCursor:mutableContainers:**

Returns a property list object corresponding to the abstract representation in the NSData at the cursor location. If mutable is YES and the object is a dictionary or an array, the re-composed object is made mutable. Returns **nil** if the object is not a valid one for property lists.

- **deserializePropertyListLazilyFromData:atCursor:length:mutableContainers:**

  Returns a property list from the NSData at the cursor location or **nil** if the NSData doesn't represent a property list. The deserialization proceeds lazily. That is, if the NSData at the cursor location has a length greater than the specified length, a fault is substituted for the actual property list as long as the constituent objects of that property list are not being accessed. If mutable is YES and the object is a dictionary or an array, the re-composed object is made mutable.

- **deserializePropertyListFromData:mutableContainers:**

  Returns a property list object corresponding to the abstract representation in the NSData or **nil** if the NSData doesn't represent a property list. If mutable containers is YES and the object is a dictionary or an array, the re-composed object is made mutable.

encodePropertyList: and decodePropertyList: Methods

The abstract superclass NSCoder implements methods for archiving and unarchiving property lists: **encodePropertyList:** and **decodePropertyList:**. Note that these methods must be used in conjunction with each other—each **encodePropertyList:** message must have a corresponding **decodePropertyList:** message, and vice versa. However, it's likely that instead of using these methods you'd serialize a property list, for the reasons described in "Working with Serialized Property Lists" on page 6.