NSPPL

Inherits From: NSObject

Conforms To: NSObject (NSObject)

Declared In: Foundation/NSPPL.h

Class Description

The NSPPL (Persistent Property List) class allows you to incrementally store property lists to and retrieve them from disk. A *property list* organizes data into named values and lists of values using four classes: NSDictionary, NSString, NSData, and NSArray. A property list is an abstract concept whose implementation can vary depending on context. For example, while NSPPL stores property lists in a binary format, in other parts of NEXTSTEP (such as the defaults system) property lists are represented in an ASCII format. NSPPL is only one mechanism for creating and storing property lists; for a discussion of other approaches, see the section ““Alternatives to NSPPL”.”

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 the key is a string, and the 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. An NSArray in an NSPPL can contain NSStrings, NSDatas, NSDictionaries, and other NSArrays.

For example, this code excerpt creates an NSPPL, **PersonalInfo.ppl**, that stores a person’s name, address, and children’s names. The example uses combinations of NSString, NSArray, and NSDictionary objects. All data is accessed through a root NSDictionary in which the person’s name is an NSString, the children’s names are stored in an NSArray, and the address is stored in an NSDictionary containing NSStrings.

```
NSPPL *ppl;
NSString *pplPath;
NSMutableDictionary *root;
NSString *name = @"Chris Smith";
NSArray *childArray;
NSMutableDictionary *addressDict;

childArray = [NSArray arrayWithObjects:@"Sam", @"Bettina",
    @"Eloise", nil];

addressDict = [NSMutableDictionary dictionaryWithCapacity:4];
/* Add data to addressDict. */
```

```
[addressDict setObject:@"955 Elm Street" forKey:@"street"];
[addressDict setObject:@"Midland" forKey:@"city"];
[addressDict setObject:@"Kansas" forKey:@"state"];
[addressDict setObject:@"19067" forKey:@"zipcode"];

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

/* Get ppl's root dictionary. */
root = [ppl rootDictionary];
/* Through the root dictionary, add data to ppl. */
[root setObject:name forKey:@"name"];
[root setObject:addressDict forKey:@"address"];
[root setObject:childArray forKey:@"children"];
/* Save ppl. */
[ppl save];
```

Structuring a Persistent Property List

As the above code excerpt illustrates, an NSPPL maintains a root dictionary that's used as the point of departure for whatever other objects or data structures you attach to it. How you structure your data within that dictionary is a matter of personal preference. For example, you can set up an NSDictionary of NSDictionaries, an NSDictionary containing NSArrays, and so on. The solution you choose is dependent only on what works best for your data.

How Persistent Property Lists Work

When you instantiate an NSPPL object, a directory is created that has the name you specify. This directory contains two files: **store** and **log**. **store** and **log** work in conjunction to ensure that any changes you make to your NSPPL can be rolled back in the case of a failure. While you're modifying a persistent property list, the changes are recorded in **store** and added to **log**. **log** maintains the property list's old state so that if a failure occurs, the changes made in **store** can be rolled back.

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 and are reflected in the **store** and **log** files. A **save** operation has the effect of committing the changes you've made to disk.

If an NSPPL becomes damaged (for example, if there's a bad bit on the disk), attempting to access it raises the exception NSInternalInconsistencyException. If this happens, you can retrieve the NSPPL by catching the exception and using the NSException **userInfo** method to get the userInfo NSDictionary. The userInfo

NSDictionary usually has the NSPPL available under the key “NSPPL”, which you can use to rebuild your persistent property list (when possible).

Alternatives to NSPPL

You can store a property list in three different ways: as an ASCII file, in a serialized binary format, and 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 serialization doesn’t allow you to modify your data and then only re-serialize the part that changed.

Like serialization, a persistent property list stores data in a binary format, provides fast access, and is lazy. It also allows you to make incremental changes to an NSPPL (even one that 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 particularly 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).

Overview of Methods

The methods in NSPPL break down into three general categories:

- Basic methods
- PPLData methods
- Performance tuning methods

You use the basic methods to perform most NSPPL operations, such as creating an NSPPL, changing its contents, and saving your changes. Basic methods include **pplWithPath:create:readOnly:**, **initWithPath:create:readOnly:**, **rootDictionary**, and **save**. These are the methods you use the majority of the time.

The PPLData methods provide read-only snapshots of an entire NSPPL as an NSData object. This allows you to pass NSPPLs across processes; passing NSPPLs as binary data is very fast. Wherever a method has the keyword **...PPLData** (for example, **pplWithPath:fromPPLData:readOnly:**), you use the **contentsAsData** method to return the contents of an NSPPL as an NSData object. For example:

```
NSPPL *originalPPL;   /* Assume this exists. */
NSString *aPath;      /* Assume this exists. */
NSPPL *newPPL = [NSPPL pplWithPath:aPath
    fromPPLData:[originalPPL contentsAsData] readOnly:NO];
```

The PPLData methods include **pplWithPath:fromPPLData:readOnly:**, **initWithPath:fromPPLData:readOnly:**, and **propertyListWithPPLData:**. These methods provide a fast mechanism for reading and passing read-only copies of an NSPPL. For example, you can have one

process continually updating the contents of an NSPPL as multiple read-only copies of the NSPPL are distributed.

The performance tuning methods, **flush**, **pushChangesToDisk**, **detachFromFile** and **setCacheHalfLife:**, let you exercise finer-grained control over the behavior of an NSPPL. You can use **flush** and **pushChangesToDisk** in conjunction with **save** to speed up save operations. **detachFromFile** ensures that no more changes are written to disk; this is useful if certain error conditions occur such as running out of disk space or if an NSPPL becomes damaged. **setCacheHalfLife:** lets you fine-tune the trade-offs between memory usage and speed.

Method Types

Creating an NSPPL object	+ pplWithPath:create:readOnly: + pplWithPath:fromPPLData:readOnly: – initWithPath:create:readOnly: – initWithPath:fromPPLData:readOnly:
Passing and accessing data	+ propertyListWithPPLData: – contentsAsData – rootDictionary
Saving changes to an NSPPL	– flush – pushChangesToDisk – save
Tuning NSPPL performance	– detachFromFile – setCacheHalfLife:

Class Methods

pplWithPath:create:readOnly:

+ (NSPPL *)**pplWithPath:**(NSString *)*path*
create:(BOOL)*createFlag*
readOnly:(BOOL)*readOnlyFlag*

Given a *path*, reads the NSPPL stored in it or creates a new one, where *readOnlyFlag* indicates whether the NSPPL can be changed, and *createFlag* indicates whether the NSPPL should be created if it doesn't already exist. Returns the new NSPPL object or **nil** if the file can't be read.

See also: + **pplWithPath:fromPPLData:readOnly:**, + **propertyListWithPPLData:**

pplWithPath:fromPPLData:readOnly:

+ (NSPPL *)**pplWithPath:**(NSString *)*path*
fromPPLData:(NSData *)*pplData*
readOnly:(BOOL)*readOnlyFlag*

Given a *path*, reads or creates an NSPPL that is identical in every way, except for mutability, to the NSPPL represented by *pplData*. *readOnlyFlag* indicates whether the new NSPPL can be modified; the new NSPPL doesn't have to have the same mutability or immutability as the NSPPL specified in the *pplData* argument. Returns the new NSPPL object or **nil** if the file can't be read.

An NSPPL is saved when it's created; you don't need to explicitly save it. You can use this method to make local back ups of a remote NSPPL.

When you use this method or any of the other NSPPL methods that take NSData arguments, you use **contentsAsData** to return a read-only snapshot of an NSPPL as an NSData object, for example:

```
NSPPL *originalPPL;  /* Assume this exists. */
NSString *aPath;     /* Assume this exists. */
NSPPL *pplCopy = [NSPPL pplWithPath:aPath
    fromPPLData:[originalPPL contentsAsData] readOnly:NO];
```

See also: + **pplWithPath:create:readOnly:**, + **propertyListWithPPLData:**

propertyListWithPPLData:

+ (NSDictionary *)**propertyListWithPPLData:**(NSData *)*pplData*

Returns an immutable root dictionary that gives you access to a snapshot of the NSPPL specified in the *pplData* argument.

This method is similar to **rootDictionary**, in that both methods return a root dictionary object that gives you access to an NSPPL. However, while **rootDictionary** returns an NSMutableDictionary that allows you to modify the associated NSPPL (so long as the NSPPL itself is mutable), **propertyListWithPPLData:** always returns an immutable NSDictionary whose associated NSPPL can't be modified. This method is a very fast way of getting a read-only snapshot of an NSPPL.

See also: + **pplWithPath:fromPPLData:readOnly:**, + **pplWithPath:create:readOnly:**, – **rootDictionary**

nstance Methods

contentsAsData

– (NSData *)**contentsAsData**

Returns an NSData object containing a read-only snapshot of an entire NSPPL, that can then be passed to another process. Passing NSPPLs as binary data is very fast; see the section “Overview of Methods” in the class description for more information.

See also: – **rootDictionary**, – **propertyListWithPPLData:**, – **initWithPath:fromPPLData:readOnly:**

detachFromFile

– (void)**detachFromFile**

Guarantees that no more changes will be written to disk. After this method is invoked, proxies can be released but not accessed. This method is invoked by **dealloc**. You can use this method to intervene in certain error conditions: For example, if your NSPPL becomes damaged, you can use this method to disable writing to disk. You can then delete the damaged NSPPL and regenerate it (when possible). You should only use this method if you’re sure where proxies are when the file is being detached.

flush

– (void)**flush**

Flushes changes to disk. You can use **flush** and **save** in combination to achieve maximum performance. **save** is a costly operation that writes the current state of the NSPPL to disk and makes the necessary adjustments to the **log** and **store** files; **flush** is faster and it accomplishes much of the work that would otherwise have to be performed by **save**. If you use **flush** at regular intervals and save only as needed, your **save** operations will be faster since **flush** does much of the work.

The primary difference between **flush** and **save** is that if your program fails, your NSPPL is guaranteed only to revert to the previous **save**, not to the previous **flush**.

Like **save**, **flush** guarantees that nothing is done if no change was made to your NSPPL. All mutable container objects from the NSPPL that you have not retained are invalid after this method is invoked.

See also: – **pushChangesToDisk**

initWithPath:create:readOnly:

– (id)**initWithPath:**(NSString *)*path*
create:(BOOL)*createFlag*
readOnly:(BOOL)*readOnlyFlag*

Given a *path*, reads the NSPPL stored in it or creates a new one, where *readOnlyFlag* indicates whether the NSPPL can be changed, and *createFlag* indicates whether the NSPPL should be created if it doesn't already exist. Returns the new NSPPL or **nil** if the file can't be read.

See also: – **initWithPath:fromPPLData:readOnly:**

initWithPath:fromPPLData:readOnly:

– (id)**initWithPath:**(NSString *)*path*
fromPPLData:(NSData *)*pplData*
readOnly:(BOOL)*readOnlyFlag*

Given a *path*, reads or creates an NSPPL that is identical in every way except for mutability to the NSPPL represented by *pplData*. *readOnlyFlag* indicates whether the new NSPPL can be modified; the new NSPPL doesn't have to have the same mutability or immutability as the NSPPL specified in the *pplData* argument. Returns **nil** if the file can't be read. An NSPPL is saved when it's created; you don't need to explicitly save it.

See also: – **initWithPath:create:readOnly:**

pushChangesToDisk

– (void)**pushChangesToDisk**

Ensures that the disk image reflects the current state of the NSPPL. Unlike **flush** and **save** this method has no effect on non-retained objects, and no notification is posted.

rootDictionary

– (NSMutableDictionary *)**rootDictionary**

Returns an NSPPL's root dictionary, through which you can modify the NSPPL (so long as the NSPPL itself is mutable; if it isn't, this method returns an immutable NSDictionary). You interact with an NSPPL through its root dictionary, which provides access to all of the NSPPL's other data structures.

Whenever the root dictionary is first modified after a **flush** or a **save** operation, this method posts the NSPPLBecameDirty notification with the NSPPL that was modified to the default notification center.

NSMutableDictionaries returned from this method have the same validity as if the dictionary were in memory, except that they do not survive beyond a **save** or **flush** operation unless they have been retained or copied.

If you store an NSMutableDictionary, release all references to it, and retrieve it, a proxy is returned rather than the original dictionary. However, if an NSMutableDictionary is stored and retained, you don't get a proxy when you retrieve it—you get the actual object.

See also: – **contentsAsData**

save

– (void)**save**

Saves any changes you made to the NSPPL to disk and makes the necessary adjustments to the **log** and **store** files. If **save** returns, it means that the changes have been reflected in the NSPPL. This method guarantees that nothing is done if no change was made. If a **save** operation succeeds, this method posts the NSPPLSaved notification with the NSPPL that was saved to the default notification center.

You can use **save** in combination with **flush** to achieve maximum performance. Use **flush** at regular intervals and **save** as needed. **flush** is a faster and less comprehensive operation than **save**, and it also makes subsequent **save** operations faster since **flush** does much of the work that would otherwise have to be performed by **save**. The primary difference between **flush** and **save** is that if your program fails, your NSPPL is guaranteed only to revert to the previous **save**, not to the previous **flush**.

As with **flush**, all mutable objects that you have not retained are invalid after this operation.

See also: – **flush**, – **pushChangesToDisk**

setCacheHalfLife:

– (void)**setCacheHalfLife:**(NSTimeInterval)*halfLife*

Sets the time interval after which cached items are written to disk. Cached items have a 50% chance of being written out after *halfLife* seconds. If *halfLife* is zero, cached items are always flushed when the cache is refreshed. If *halfLife* is less than zero, caching is disabled and can't be re-enabled. This method allows you to fine-tune the trade-offs between memory usage and speed.