Pasteboard Programming Guide

Developer

Contents

Figures

SwiftObjective-C

Introduction

A pasteboard is a standardized mechanism for exchanging data within applications or between applications. The most familiar use for pasteboards is handling copy and paste operations. A number of operations are supported by pasteboards, including dragging and application Services.

You should read this document to understand how to use pasteboards.

Note: This document describes how to use pasteboards on OS X v10.6 and later. To understand how to use pasteboards on OS X v10.5 and earlier, read *Pasteboard Programming Topics for Cocoa*.

Important: To use pasteboards effectively, you must have a working knowledge of property lists. If you don't know what a property list is, or how to use property list items, you should first read *Property List Programming Guide*.

Pasteboards frequently use archived representations of objects and object graphs. If you don't understand what a Cocoa archive is and how to create or use one, you should first read *Archives and Serializations Programming Guide*.

Organization of This Document

This document contains the following articles:

Getting Started with Pasteboards

This tutorial offers a quick, practical, introduction to using pasteboards. It doesn't provide in-depth explanation of how pasteboards work, or details of the methods used. These are discussed in later articles in this document.

Introduction

This tutorial introduces you to pasteboards on OS X. The project is a simple document-based application that manages a window containing an image view. You can copy and paste images between documents in your application, or to and from another application. The application will not support saving and opening documents, although you can easily add this functionality.

On completion of this tutorial, you should be able to:

- Copy objects to and retrieve objects from a pasteboard
- Understand how to validate user interface items based on the content of the pasteboard

Before You Start

This tutorial assumes you already have familiarity with the fundamentals of Cocoa development. Concepts you must understand include:

- How to create a new Xcode project
- How to configure a user interface in Interface Builder
- How to define and implement a simple class

You must have at least either worked through *Start Developing Mac Apps Today* or gained equivalent experience using other examples.

It is also helpful to understand:

- The document architecture

You don't need to understand the document architecture in detail—the application will not support saving and opening documents, although you can easily add these features. The goal is simply to provide an application that supports copy and paste between multiple windows, and this is most easily achieved using the document architecture. If you want to know more about the document architecture, read *Mac App Programming Guide*.

- Toolbars

  The document makes use of a toolbar to illustrate user interface validation.

- User interface validation

  User interface validation provides a standard way to set the state of interface items as appropriate for the current application context. In this tutorial, it is used to disable the Paste toolbar item if there is no suitable data on the pasteboard. You don't need to know more than this, but for more details see *User Interface Validation*.

Summary

The main stages of the tutorial are as follows:

- Create a simple document-based application
- Configure the document class to interact with a simple user interface
- Configure the document user interface
- Add copy and paste methods to the document
- Add user interface validation to the application

Tutorial

Create a new Xcode project

Create a new Xcode Cocoa document-based application. Call your new project CopyImage, or something like that.

Update the document class

The document maintains a simple user interface that contains an image view. It knows how to copy an image from the image view, and paste an image into the image view.

Update MyDocument.h to this:

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


@interface MyDocument : NSDocument
{
    NSImageView *imageView;
}
@property (nonatomic, retain) IBOutlet NSImageView *imageView;
- (IBAction)copy:sender;
- (IBAction)paste:sender;
@end
```

Update MyDocument.m. *In addition to the methods provided by the template*, synthesize the property and add stub implementations of the `copy:` and `paste:` methods: methods:

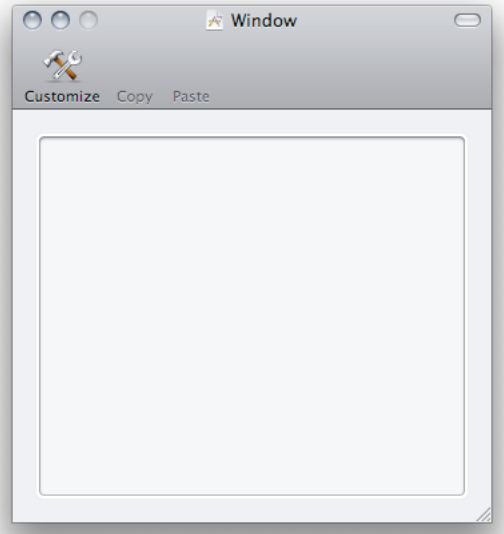
```
@synthesize imageView;
- (IBAction)copy:sender {
}
- (IBAction)paste:sender {
}
```

Configure the user interface

To the document window:

- Add an image view
- Connect the document's (File's Owner's) `imageView` outlet to the image view
- Add a toolbar
- Add two toolbar items to the toolbar—one labeled "Copy", the other "Paste"
- Connect the "Copy" and "Paste" toolbar items' actions to File's Owner's `copy:` and `paste:` actions respectively

You should end up with a document window that looks like this:



Add copy support to the document

There are three steps to writing to a pasteboard:

1. Get a pasteboard
2. Clear the pasteboard's contents
3. Write an array of objects to the pasteboard

Objects you write to the pasteboard must adopt the `NSPasteboardWriting Protocol Reference` protocol. Several of the common Foundation and Application Kit classes implement the protocol including `NSString`, `NSImage`, `NSURL`, and `NSColor`. (If you want to write an instance of a custom class, either it must adopt the `NSPasteboardWriting` protocol or you can wrap it in an instance of an `NSPasteboardItem`—see Custom Data (page 25).) Since `NSImage` adopts the `NSPasteboardWriting` protocol, you can write an instance directly to a pasteboard.

In the MyDocument class, complete the implementation of `copy:` as follows:

```
- (IBAction)copy:sender {
    NSImage *image = [imageView image];
    if (image != nil) {
        NSPasteboard *pasteboard = [NSPasteboard generalPasteboard];
```

```
        [pasteboard clearContents];
        NSArray *copiedObjects = [NSArray arrayWithObject:image];
        [pasteboard writeObjects:copiedObjects];
    }
}
```

Test the application

Build and run your application.

- Drag an image from Finder or from another application into a document's image view
- Press Copy in the toolbar

You should find that you can paste the image into another application. (For example, in Text Edit, create a new Rich Text document then select Edit > Paste.)

Add paste support to the document

Before you try to read from a pasteboard, you need to check that it contains data you want.

You check that the pasteboard contains objects you're interested in by sending it a `canReadObjectForClasses:options:` message. The first argument is an array that tell the pasteboard what classes object you're interested in.

Classes you ask to read from the pasteboard must adopt the `NSPasteboardReading` protocol. Like writing, several of the common Foundation and Application Kit classes implement the protocol, again including `NSString`, `NSImage`, `NSURL`, and `NSColor`. (Similarly, if you want to read an instance of a your own class class, either it must adopt the `NSPasteboardReading` protocol or, when you write it to the pasteboard, you can wrap it in an instance of an `NSPasteboardItem` and retrieve that—see Custom Data (page 25).) Since `NSImage` adopts the `NSPasteboardReading` protocol, you can read an instance directly from the pasteboard

If the pasteboard does contain objects you're interested in, you can retrieve them by sending the pasteboard a `readObjectsForClasses:options:`. message. The pasteboard determines which objects it contains that can be represented using the classes you specify, and returns an array of the best matches (if any).

In the MyDocument class, complete the implementation of `paste:` as follows:

```
- (IBAction)paste:sender {
    NSPasteboard *pasteboard = [NSPasteboard generalPasteboard];
    NSArray *classArray = [NSArray arrayWithObject:[NSImage class]];
```

```
    NSDictionary *options = [NSDictionary dictionary];


    BOOL ok = [pasteboard canReadObjectForClasses:classArray options:options];
    if (ok) {
        NSArray *objectsToPaste = [pasteboard readObjectsForClasses:classArray
options:options];
        NSImage *image = [objectsToPaste objectAtIndex:0];
        [imageView setImage:image];
    }
}
```

Test the application

Build and run your application:

- Drag an image from Finder or from another application into the document window
- Press Copy in the toolbar
- Create a new document (select File > New)
- In the new document, press Paste in the toolbar

You should find that you can paste the image from the first document into the second. If you have another application that allows you to copy and paste images, you should also find that you can copy and paste between that application and the tutorial application.

Extra Credit: Menu validation

It's good practice to restrict the user to performing actions that will have an effect. In the current application, you can press the Paste toolbar item even if there is no image on the pasteboard. It would be better to disable the item if there isn't anything on the pasteboard that can be pasted.

User interface validation—supported by the `NSUserInterfaceValidations` protocol—provides a standard way to set the state of interface items as appropriate for the current application context. The protocol contains a single method—`validateUserInterfaceItem:`—that returns a Boolean which specifies whether or not the user interface element passed as the argument should be enabled.

When you implement `validateUserInterfaceItem:`, you typically first check the action associated with the user interface element (you don't want to enable or disable every item on the basis of a single test). In this case, you're only interested if the associated action is `paste:`. If it is, you then need to check whether there's anything on the pasteboard that can be pasted. You can use `canReadObjectForClasses:options:` to ask the pasteboard if it contains any data that can be converted into an `NSImage` object.

In your document class, implement `validateUserInterfaceItem:` as follows:

```
- (BOOL)validateUserInterfaceItem:(id < NSValidatedUserInterfaceItem >)anItem {

    if ([anItem action] == @selector(paste:)) {
        NSPasteboard *pasteboard = [NSPasteboard generalPasteboard];
        NSArray *classArray = [NSArray arrayWithObject:[NSImage class]];
        NSDictionary *options = [NSDictionary dictionary];
        return [pasteboard canReadObjectForClasses:classArray options:options];
    }
    return [super validateUserInterfaceItem:anItem];
}
```

If you build and run your application, you should find that you can copy and paste images as before. You should also, though, find that if you haven't copied an image, the Paste toolbar item is disabled.

Pasteboard Concepts

On OS X, a number of operations are supported by a pasteboard server process. The most obvious is copy and paste, however dragging and Services operations are also mediated using pasteboards. In Cocoa, you access the pasteboard server through an `NSPasteboard` object. This article describes how the pasteboard process works.

What's a Pasteboard?

A pasteboard is a standardized mechanism for exchanging data within an application or between applications. The most familiar use for pasteboards is handling copy and paste operations:

- When a user selects data in an application and chooses the Copy (or Cut) menu item, the selected data is placed onto a pasteboard.
- When the user chooses the Paste menu item (either in the same or a different application), the data on a pasteboard is copied to the current application.

Perhaps less obviously, Find operations are also supported by pasteboards, as are dragging and Services operations:

- When a user begins a drag, the drag data is added to a pasteboard. If the drag ends with a drop action, the receiving application retrieves the drag data from the pasteboard.
- If a translation service is requested, the requesting application places the data to be translated onto a pasteboard. The service retrieves this data, performs the translation, and places the translated data back onto the pasteboard.

Because they may be used to transfer data between applications, pasteboards exist in a special global memory area separate from application processes—this is described in more detail in The Pasteboard Server (page 17). This implementation detail, though, is abstracted away by the `NSPasteboard` class and its methods. All you have to do is interact with the pasteboard object.

The basic tasks you want to perform, regardless of the operation, are to (a) write data to a pasteboard and (b) read data from a pasteboard. These tasks are conceptually very simple, but mask a number of important details. In practical terms, the main underlying issue that adds complexity is that there may be a number of ways to represent data—this leads in turn to considerations of efficiency. Furthermore, from the system's perspective, there are additional issues to consider. These are discussed in the following sections.

Named Pasteboards

Pasteboards may be public or private, and may be used for a variety of purposes. There are several standard pasteboards provided for well-defined operations system-wide:

- `NSGeneralPboard`—for cut, copy, and paste
- `NSRulerPboard`—for copy and paste of rulers
- `NSFontPboard`—for cut, copy, and paste of `NSFont` objects
- `NSFindPboard`—application-specific find panels can share a sought after text value
- `NSDragPboard`—for graphical drag and drop operations

Typically you use one of the system-defined pasteboards, but if necessary you can create your own pasteboard for exchanges that fall outside the predefined set using `pasteboardWithName:` If you invoke `pasteboardWithUniqueName`, the pasteboard server will provide you with a uniquely-named pasteboard.

Pasteboard Items

Each piece of data placed onto a pasteboard is considered a pasteboard *item*. The pasteboard can hold multiple items. Applications can place or retrieve as many items as they wish. For example, say a user selection in a browser window contains both text and an image. The pasteboard lets you copy the text and the image to the pasteboard as separate items. An application pasting multiple items can choose to take only those that is supports (the text, but not the image, for example).

Representations and UTIs

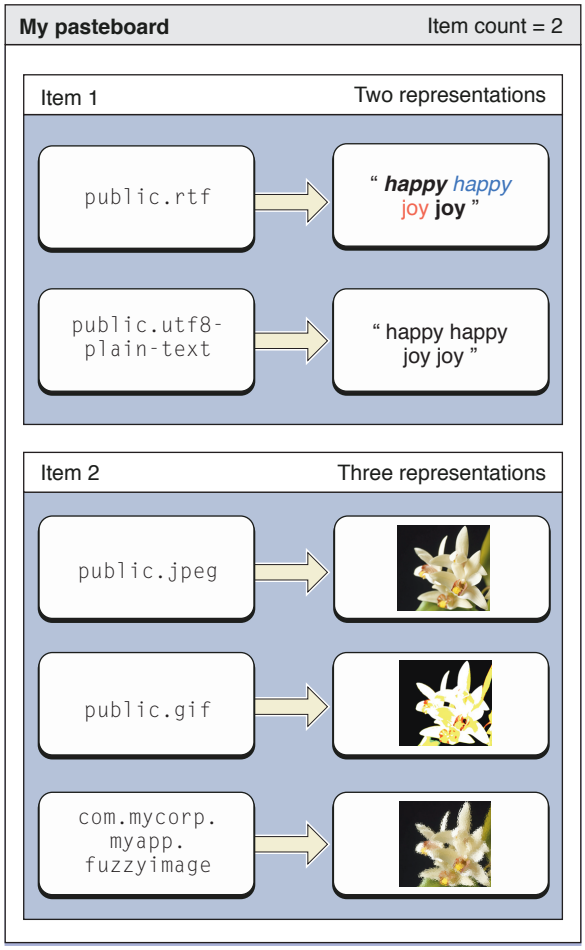
Pasteboard operations are often carried out between two different applications. Neither application has knowledge about the other and the kinds of data each can handle. To maximize the potential for sharing, a pasteboard can hold multiple representations of the same pasteboard *item* . For example, a rich text editor might provide RTFD, RTF, and `NSString` representations of the copied data. An item that is added to a pasteboard specifies what representations it is able to provide.

Each representation of an item is identified by a different Unique Type Identifier (UTI). (A UTI is simply a string that uniquely identifies a particular data type. For more information, see *Uniform Type Identifiers Overview*.) The UTI provides a common means to identify to identify data types.

For example, suppose an application supported selection of rich text and images. It may want to place on a pasteboard both rich text and Unicode versions of a text selection and different representations of an image selection. Each representation of each item is stored with its own data, as shown in Figure 1 (page 16). You can declare your own UTIs to support your own custom data types.

Important: If you formally declare a pasteboard type as a UTI, it must conform to `public.data`.

Figure 1 Pasteboard items and representations



In general, to maximize the potential for sharing, pasteboard items should provide as many different representations as possible (see Custom Data (page 25)). Theoretically this may lead to concerns regarding efficiency, in practice, however, these are mitigated by the way the items provide different representations—see Promised Data (page 17).

A pasteboard reader must find the data type that best suits its capabilities (if any). Typically, this means selecting the richest type available. In the previous example, a rich text editor might provide RTFD, RTF, and `NSString` representations of copied data. An application that supports rich text but without images should retrieve the RTF representation; an application that only supports plain text should retrieve the `NSString` object, whereas an image-editing application might not be able to use the text at all.

Promised Data

If a pasteboard item supports multiple representations, it is typically impractical or time- and resource-consuming to put the data for each representation onto the pasteboard. For example, say your application places an image on the pasteboard. For maximum compatibility, it might be useful for the image to provide a number of different representations, including PNG, JPEG, TIFF, GIF, and so on. Creating each of these representations, however, would require time and memory.

Rather than requiring an item to provide all of the representations it offers, a pasteboard only asks for the first representation in the list of representations an item offers. If a paste recipient wants a different representation, the item can generate it when it's requested.

You can also place an item on a pasteboard and specify that one or more representations of the item be provided by some other object. To do this, you specify a data provider for a particular type on the pasteboard item. The data provider must conform to the `NSPasteboardItemDataProvider Protocol` protocol to provide the corresponding data on demand.

Change Count

The change count is a computer-wide variable that increments every time the contents of the pasteboard changes (a new owner is declared). An independent change count is maintained for each named pasteboard. By examining the change count, an application can determine whether the current data in the pasteboard is the same as the data it last received. The `changeCount` and `clearContents` methods return the change count.

The Pasteboard Server

Whether the data is transferred between objects in the same application or two different applications, in a Cocoa application the interface is the same—an `NSPasteboard` object accesses a shared repository where writers and readers meet to exchange data. The writer, referred to as the pasteboard owner, deposits data on

a pasteboard instance and moves on. The reader then accesses the pasteboard asynchronously, at some unspecified point in the future. By that time, the writer object may not even exist anymore. For example, a user may have closed the source document or quit the application.

Consequently, when moving data between two different applications, and therefore two different address spaces, a third memory space gets involved so the data persists even in the absence of the source. `NSPasteboard` provides access to a third address space—a pasteboard server process (`pbs`)—that is always running in the background. The pasteboard server maintains an arbitrary number of individual pasteboards to distinguish among several concurrent data transfers.

Errors

Except where errors are specifically mentioned in the `NSPasteboard` method descriptions, any communications error with the pasteboard server raises an`NSPasteboardCommunicationException`.

Copying to a Pasteboard

Objective-CSwift

You perform a copy operation by first clearing the existing contents of, then writing the copied objects to, a pasteboard.

There are three steps to performing a copy operation:

1. Get a pasteboard.

   Typically, you just use the general pasteboard:

   ```
   NSPasteboard *pasteboard = [NSPasteboard generalPasteboard];
   ```

2. Clear the contents of the pasteboard.

   Typically, you just use the general pasteboard:

   ```
   NSInteger changeCount = [pasteboard clearContents];
   ```

   The method returns the change count of the pasteboard; you usually don't need this value.

3. Write the objects being copied to the pasteboard.

   You pass the objects to write in an array—objects in the array must adopt the `NSPasteboardWriting Protocol Reference` protocol:

   ```
   NSArray *objectsToCopy = <#An array of objects#>;
   BOOL OK = [pasteboard writeObjects:objectsToCopy];
   ```

   The method returns `NO` if the items were not successfully added to the pasteboard.

In many cases, this is as much as you need to do. Note, though, the important prerequisite that objects you write to the pasteboard must adopt the `NSPasteboardWriting` protocol. Classes that implement the protocol include `NSString`, `NSImage`, `NSURL`, `NSColor`, `NSAttributedString`, and `NSPasteboardItem`. If you want to write an instance of a custom class, either it must adopt the `NSPasteboardWriting` protocol or you can wrap it in an instance of an `NSPasteboardItem`—see Custom Data (page 25).

Reading from a Pasteboard

This article describes how to read data from a pasteboard.

Simple Reading

You read data from the pasteboard using the method, `readObjectsForClasses:options:`.

The method takes two arguments:

1. An array of the classes, instances of which you're interested in reading from the pasteboard.

   It's important that you order this array such that your preferred class is at the beginning of the array, with any remaining classes ordered in decreasing preference. (This will be explained in more detail in a moment.)

2. A dictionary containing options associated with the request.

   Using the options dictionary, you can place additional constraints on the such as restricting the search to file URLs with particular content types. If you don't want to add further constraints, provide an empty dictionary.

The method returns an array of items that meet the criteria specified by the arguments. (If there is an error in retrieving the requested items from the pasteboard or if no objects of the specified classes can be created, the method returns `nil`.) For every *item* on the pasteboard:

1. Each class in the classes array is queried for the types it can read (using `readableTypesForPasteboard:`).
2. An instance is created of the *first* class found in the classes array whose readable types match a conforming type contained in the pasteboard item

   This is why it's important to order the classes in the classes array according to your preference. (If no class is found, then no instance is created.)

Any instances that could be created from pasteboard item data are returned in the array. The returned array therefore contains at most the same number of elements as there are items on the pasteboard.

The following example shows how you can read strings from the pasteboard.

```
NSPasteboard *pasteboard = <#Get a pasteboard#>;
```

```
NSArray *classes = [[NSArray alloc] initWithObjects:[NSString class], nil];
NSDictionary *options = [NSDictionary dictionary];
NSArray *copiedItems = [pasteboard readObjectsForClasses:classes options:options];
if (copiedItems != nil) {
    // Do something with the contents...
```

Requesting Multiple Types

Sometimes you may be willing to handle multiple representations. There are at least two strategies for retrieving items from the pasteboard.

1. Make a single invocation of `readObjectsForClasses:options:` passing an array containing the classes you wish to handle.
2. Make multiple invocations of `readObjectsForClasses:options:`, each time passing a different array of classes.

The following examples illustrate the two approaches.

In this example, for any item that can provide both a rich text and a string representation, the rich text is preferred:

```
NSPasteboard *pasteboard = <#Get a pasteboard#>;
NSArray *classes = [[NSArray alloc] initWithObjects:[NSAttributedString class],
[NSString class], nil];
NSDictionary *options = [NSDictionary dictionary];
NSArray *copiedItems = [pasteboard readObjectsForClasses:classes options:options];
if (copiedItems != nil) {
    // Do something with the contents...
}
```

In this example, the pasteboard is searched first for string objects, and then for attributed strings. This sequence may result in pasteboard items being "used" twice if they can provide both string and attributed string representations.

```
NSPasteboard *pasteboard = <#Get a pasteboard#>;
NSDictionary *options = [NSDictionary dictionary];
NSArray *classes;
```

```
classes = [[NSArray alloc] initWithObjects:[NSString class], nil];
NSArray *strings = [pasteboard readObjectsForClasses:classes options:options];
if (strings != nil) {
    // Do something with the strings...
}

classes = [[NSArray alloc] initWithObjects:[NSAttributedString class], nil];
NSArray *attributedStrings = [pasteboard readObjectsForClasses:classes
options:options];
if (copiedItems != nil) {
    // Do something with the attributed strings...
}
```

Validation

Sometimes you want to know whether a pasteboard contains data that you can use, but without actually retrieving the data. For example, you might want to know whether a Paste menu item should be enabled, or whether a drop operation (in a drag and drop sequence) is valid.

To determine whether a pasteboard contains data that can create instances of classes you're interested in, you can send the pasteboard a `canReadObjectForClasses:options:` message.

```
NSPasteboard *pasteboard = <#Get a pasteboard#>;
NSArray *classes = [[NSArray alloc] initWithObjects:[NSAttributedString class],
[NSString class], nil];
NSDictionary *options = [NSDictionary dictionary];
BOOL ok = [pasteboard canReadObjectForClasses:classes options:options];
if (ok) {
    // ...
}
```

URL Reading Options

Sometimes you may want to retrieve URLs from a pasteboard, but only if they meet certain constraints—for example, you may only be interested in file URLs, or URLs that point to data of a particular type. You can impose such constraints using the options argument when you validate the contents of a pasteboard using `canReadObjectForClasses:options:`, or read from a pasteboard using `readObjectsForClasses:options:`.

You use the `NSPasteboardURLReadingFileURLsOnlyKey` option to constrain results to file URLs, as shown in this example.

```
NSPasteboard *pasteboard = <#Get a pasteboard#>;
NSArray *classes = [NSArray arrayWithObject:[NSURL class]];

NSDictionary *options = [NSDictionary dictionaryWithObject:
    [NSNumber numberWithBool:YES] forKey:NSPasteboardURLReadingFileURLsOnlyKey];

NSArray *fileURLs =
    [pasteboard readObjectsForClasses:classes options:options];
```

You use the `NSPasteboardURLReadingContentsConformToTypesKey` option to constrain results to URLs that point to particular types of data. The following example illustrates how you can retrieve only URLs that point to image data.

```
NSPasteboard *pasteboard = <#Get a pasteboard#>;
NSArray *classes = [NSArray arrayWithObject:[NSURL class]];

NSDictionary *options = [NSDictionary dictionaryWithObject:
    [NSImage imageTypes], forKey:NSPasteboardURLReadingContentsConformToTypesKey];

NSArray *imageURLs =
    [pasteboard readObjectsForClasses:classes options:options];
```

You can combine the two options; to retrieve only file URLs that point to image data, you would create the options dictionary as follows:

```
NSDictionary *options = [NSDictionary dictionaryWithObjectsAndKeys:
    [NSNumber numberWithBool:YES], NSPasteboardURLReadingFileURLsOnlyKey,
```

```
    [NSImage imageTypes], NSPasteboardURLReadingContentsConformToTypesKey, nil];
```

Custom Data

To be used with a pasteboard, an object must conform to the `NSPasteboardWriting` and/or `NSPasteboardReading` protocols. You can make your own custom objects conform to these protocols, or you can use instances of `NSPasteboardItem` to contain custom data.

Overview

Any object that you put on a pasteboard must conform to the `NSPasteboardWriting` protocol; to retrieve an instance of an object from the pasteboard, it must adopt the `NSPasteboardReading` protocol. Several of the common Foundation and Application Kit classes implement both of these protocols, including `NSString`, `NSImage`, `NSURL`, `NSColor`, `NSAttributedString`, and `NSPasteboardItem`.

If you want to be able to write your custom object to a pasteboard, or initialize an instance of your custom class from a pasteboard, your class must also adopt the corresponding protocol. In some situations, it may be appropriate to use `NSPasteboardItem`.

Custom Class

For the examples that follow, consider a simple model class that represents a bookmark as you might find in a web browser:

```
@interface Bookmark : NSObject <NSCoding, NSPasteboardWriting, NSPasteboardReading>
 {
}
@property (nonatomic, copy) NSString *title;
@property (nonatomic, copy) NSString *notes;
@property (nonatomic, retain) NSDate *date;
@property (nonatomic, retain) NSURL *url;
@property (nonatomic, retain) NSColor *color;
@end
```

Notice that the class adopts the `NSCoding` protocol so that it can be archived and unarchived.

Writing

To support writing to a pasteboard, your class must conform to the `NSPasteboardWriting` protocol. It has three methods—one of which is optional—as described below. The pasteboard sends these messages to your object when you add your object to the pasteboard, so that it can determine what types of data it will hold and to get the first data representation of your object. The last method may be invoked additional times if your object provides multiple representations.

`– (NSArray *)writableTypesForPasteboard:(NSPasteboard *)pasteboard`

This method returns an array of UTIs for the data types your object can write to the pasteboard.

The order in the array is the order in which the types should be added to the pasteboard—this is important as only the first type is written initially, the others are provided lazily (see Promised Data (page 17)).

The method provides the pasteboard argument so that you can return different arrays for different pasteboards. You might, for example, put different data types on the dragging pasteboard than you do on the general pasteboard, or you might put on the same data types but in a different order. You might add to the dragging pasteboard a special representation that indicates the indexes of items being dragged so that they can be reordered.

`– (NSPasteboardWritingOptions)writingOptionsForType:(NSString *)type pasteboard:(NSPasteboard *)pasteboard`

This method is optional. It returns the options for writing data of a type to a pasteboard.

The only option currently supported is "promised". You can use this if you want to customize the behavior—for example, to ensure that one particular type is only promised.

`– (id)pasteboardPropertyListForType:(NSString *)type`

This method returns the appropriate property list object representation of your object for the specified type.

In most cases, you just implement the two required methods—`writableTypesForPasteboard:` and `pasteboardPropertyListForType:`.

In the example of the Bookmark class, you could implement `writableTypesForPasteboard:` as follows:

```
- (NSArray *)writableTypesForPasteboard:(NSPasteboard *)pasteboard {
    static NSArray *writableTypes = nil;

    if (!writableTypes) {
        writableTypes = [[NSArray alloc] initWithObjects:BOOKMARK_UTI,
    (NSString *)kUTTypeURL, NSPasteboardTypeString, nil];
```

```
    }
    return writableTypes;
}
```

This implementation ignores the pasteboard type, however you might choose to return different types for say, a general and for a dragging pasteboard.

You could implement `pasteboardPropertyListForType:` as follows:

```
- (id)pasteboardPropertyListForType:(NSString *)type {

    if ([type isEqualToString:BOOKMARK_UTI]) {
        return [NSKeyedArchiver archivedDataWithRootObject:self];
    }

    if ([type isEqualToString:(NSString *)kUTTypeURL]) {
        return [url pasteboardPropertyListForType:(NSString *)kUTTypeURL];
    }

    if ([type isEqualToString:NSPasteboardTypeString]) {
        return [NSString stringWithFormat:@"<a href=\"%@\">%@</a>",
    [url absoluteString], title];
    }

    return nil;
}
```

In the example, the method returns either an `NSData` or an `NSString` object. The pasteboard accepts data values directly, and automatically converts the string to the appropriate data representation. If the method returned any other property list type, the pasteboard would automatically convert it to the appropriate data representation.

Reading

To support reading from a pasteboard, your class must conform to the `NSPasteboardReading` protocol. It has three methods, two of which are optional.

`+ (NSArray *)readableTypesForPasteboard:(NSPasteboard *)pasteboard`

This *class* method returns an array of UTIs for the data types your object can read from the pasteboard.

The method provides the pasteboard argument so that you can return for different arrays for different pasteboards. As with reading, you might put different data types on the general pasteboard than you do on the dragging pasteboard, or you might put on the same data types but in a different order.

`+ (NSPasteboardReadingOptions)readingOptionsForType:(NSString *)type pasteboard:(NSPasteboard *)pasteboard`

This *class* method is optional. It allows you to specify the options for reading from a pasteboard.

The options are expressed using a `NSPasteboardReadingOptions` bit field. You can specify that the corresponding data should be read as *one* of:

- An `NSData` object (`NSPasteboardReadingAsData`)

  This simply returns the pasteboard data as-is.

- A string (`NSPasteboardReadingAsString`)

  The pasteboard data is returned as an instance of `NSString`.

- A property list (`NSPasteboardReadingAsPropertyList`)

  The data on the pasteboard is unserialized as a property list.

- An archive (`NSPasteboardReadingAsKeyedArchive`)

  The data on the pasteboard is treated as a keyed archive.

`– (id)initWithPasteboardPropertyList:(id)propertyList ofType:(NSString *)type`

This method is optional. You implement this method if it is possible to initialize an instance of your class using a property list.

> **Important:** This method is optional because: if your implementation of `readableTypesForPasteboard:` returns just a single type, and that type uses the `NSPasteboardReadingAsKeyedArchive` reading option, then `initWithCoder:` is called instead of this method.

The property list object is the `NSData` for that type on the pasteboard, but by specifying an `NSPasteboardReading` option for a type, the data can be automatically retrieved as a string or property list.

In the example of the Bookmark class, you could implement the protocol as follows

First, `readableTypesForPasteboard:` specifies that you can initialize a Bookmark using your own custom type and an URL:

```
+ (NSArray *)readableTypesForPasteboard:(NSPasteboard *)pasteboard {


    static NSArray *readableTypes = nil;

    if (!readableTypes) {

        readableTypes = [[NSArray alloc] initWithObjects:BOOKMARK_UTI, (NSString
*)kUTTypeURL, nil];

    }

    return readableTypes;

}
```

Next, in this case provide an implementation of `readingOptionsForType:pasteboard:` to specify that if the type is your custom type you only want to treat the data as a keyed archive, and if the type is an URL type then support the same options as `NSURL`.

```
+ (NSPasteboardReadingOptions)readingOptionsForType:(NSString *)type
pasteboard:(NSPasteboard *)pboard {

    if ([type isEqualToString:BOOKMARK_UTI]) {

        /*

         This means you don't need to implement code for this

         type from initWithPasteboardPropertyList:ofType:

        */

        return NSPasteboardReadingAsKeyedArchive;

    }

    else if ([type isEqualToString: (NSString *)kUTTypeURL]) {

        return [NSURL readingOptionsForType:type pasteboard:pboard];

    }

    return 0;

}
```

Finally, implement `initWithPasteboardPropertyList:ofType:` as follows:

```
- (id)initWithPasteboardPropertyList:(id)propertyList ofType:(NSString *)type {


    if (self = [self init]) {

        if ([type isEqualToString:(NSString *)kUTTypeURL]) {

            [url release];
```

```
            url = [[NSURL alloc] initWithPasteboardPropertyList:propertyList
ofType:type];
            [title release];
            title = [[url absoluteString] retain];
        } else {
            [self release];
            return nil;
        }
    }
    return self;
}
```

Notice two things:

1. The method calls the designated initializer, in this case `init`.
2. The method does not test for the custom Bookmark type. Recall that `readableTypesForPasteboard:` returned just a single type, and that `readingOptionsForType:pasteboard:` specified `NSPasteboardReadingAsKeyedArchive` as the sole reading option; thus for the custom Bookmark type `initWithCoder:` is called instead of `initWithPasteboardPropertyList:ofType:`.

NSPasteboardItem

Sometimes you want to write items to the pasteboard but you don't have a convenient wrapper object, or you may want to provide data in a common format but only on demand. For example, you may want to be able to provide a string as an attributed string, a simple string, or as tabular text, where the tabular text is formatted differently from the simple string (so you can't just write the attributed string directly to the pasteboard).

For these situations, you can use `NSPasteboardItem` objects. In general, `NSPasteboardItem` objects provide you with fine-grained control over what you might put on the pasteboard. They're designed to be a temporary objects—they're only associated with a single pasteboard, and only with one change count, so you shouldn't maintain them after you've created them and put them on the pasteboard.

Writing

The following example illustrates how you might use an `NSPasteboardItem` object to create a pasteboard item with three representations—string, attributed string, and tabular text. The data for these representations is promised by a data provider—in this case, `self`. The data provider must conform to the `NSPasteboardItemDataProvider Protocol` protocol. In its implementation of `pasteboard:item:provideDataForType:`, it generates and returns the data requested.

```
NSPasteboardItem *pasteboardItem = [[NSPasteboardItem alloc] init];
NSArray *types = [[NSArray alloc] initWithObjects:
    [NSPasteboardTypeRTF, NSPasteboardTypeString, NSPasteboardTypeTabularText,
nil];
BOOL ok = [pasteboardItem setDataProvider:self forTypes:types];

if (ok) {
    NSPasteboard *pasteboard = [NSPasteboard generalPasteboard];
    [pasteboard clearContents];
    ok = [pasteboard writeObjects:[NSArray arrayWithObject:pasteboardItem]];
}
if (ok) {
    // Maintain the information required to provide the data if requested.
}
```

Reading

Suppose there are five items on the pasteboard, two contain TIFF data, two contain RTF data, one contains a private data type. Calling `readObjectsForClasses:options:` with just the `NSImage` class will return an array containing two image objects. Calling with just the `NSAttributedString` class will return an array containing two attributed strings. Calling with both classes (`NSImage` and `NSAttributedString`) will return two image objects and two attributed strings.

Notice that in the previous examples, the count of objects returned is less than the number of items on the pasteboard. Only objects of the requested classes are returned. If you add the `NSPasteboardItem` class to the array, then you will always get back an array that contains the same number of objects as there are items on the pasteboard. Since you provide the elements in the classes array in your preferred order, you should add the `NSPasteboardItem` class to the end of the array.

```
// NSPasteboard *pasteboard = <#Get a pasteboard#>;
NSArray *classes = [[NSArray alloc] initWithObjects:
```

```
                    [NSImage class], [NSAttributedString class], [NSPasteboardItem
 class], nil];
NSDictionary *options = [NSDictionary dictionary];
NSArray *copiedItems = [pasteboard readObjectsForClasses:classes options:options];
if (copiedItems != nil) {
    // Do something with the contents.
```

In this case, the method will return an array with two images, two attributed strings, and one pasteboard item containing the private data type.

Integrating with OS X v10.5 and Earlier

This article describes the interoperation of pasteboard API from OS X v10.5 and earlier with the APIs introduced in OS X v10.6.

Dealing with Multiple Items

On OS X v10.6, `NSPasteboard` allows you to write multiple items to the pasteboard. If you take advantage of this ability, you may need to consider how your application will interact with others that have not yet adopted this feature.

If you place multiple items on the pasteboard, an application that uses the OS X v10.5 and earlier API will typically retrieve just the best representation of the first item on the pasteboard (using `dataForType:`, `-stringForType:`, or `propertyListForType:`). You should therefore typically ensure that the first item you place on the pasteboard is that likely to be "most interesting" to clients that use the 10.5 and earlier API.

The exception to this pattern is text. If a 10.6-based application writes to the pasteboard multiple items that provide text (strings and attributed strings), then if a 10.5-based application reads text from the pasteboard the text from the items is combined using return characters. (You can think of this as akin to the pasteboard putting the individual strings from the items into an array then joining them with `componentsJoinedByString:"@\r"`.)

Methods

You can typically use API from OS X v10.5 and earlier with the APIs introduced in OS X v10.6 together in the same application. This allows you to migrate your code to the new API in a gradual fashion. With any given sequence of interaction, however, you should be consistent in using either the API from OS X v10.5 and earlier or the APIs introduced in OS X v10.6. For example, the following combination will *not* work:

```
NSArray *fileURLs = <#An array of file URLs#>;
NSPasteboard *pboard = [NSPasteboard generalPasteboard];
NSArray *typeArray = [NSArray arrayWithObject:NSURLPboardType];
[pboard declareTypes:typeArray owner:nil]; // 10.5
[pboard writeObjects:fileURLs]; // 10.6
```

Backwards Compatibility

If your application currently defines a public pasteboard type that other applications rely on, you need to perform the following steps to ensure that existing applications will continue to see your pasteboard type even after your application has moved from pboard types to UTIs:

1. Formally declare the new pasteboard type UTI as an exported type declaration.
2. Ensure the declared UTI conforms to the `public.data` type.
3. Add a tag specification for the `com.apple.nspboard-type` tag to the type declaration. The value you provide should be the literal string value of the pboard type, not the name of the constant. The following example shows how to create a tag specification to associate a UTI with a pboard type:

```
<key>UTTypeTagSpecification</key>
    <dict>
        <key>com.apple.nspboard-type</key>
        <string>MyCustomPboardType</string>
    </dict>
```

Pasteboard Types and UTIs

The `NSPasteboardItem`, `NSPasteboardReading`, and `NSPasteboardWriting` APIs introduced in OS X v10.6 use UTIs to specify representation types. Over time, Cocoa will be moving away from the old pboard types, and towards using UTIs exclusively on the pasteboard.

On OS X v10.6 and later, you should use UTIs to identify pasteboard types. Most of the existing pasteboard type constants are not deprecated in OS X v10.6 (the exceptions are PICT and file content types), however they will be deprecated in the future.

Pasteboard Types with New UTIs

The following table shows pasteboard types for which there are new UTIs. The table shows the OS X v10.5 and earlier constant, the OS X v10.6 constant, and the UTI string.

Old Constant	New Constant	UTI String
NSColorPboardType	NSPasteboardTypeColor	com.apple.cocoa.pasteboa
NSSoundPboardType	NSPasteboardTypeSound	com.apple.cocoa.pasteboa

Old Constant	New Constant	UTI String
NSFontPboardType	NSPasteboardTypeFont	com.apple.cocoa.pasteboa
NSRulerPboardType	NSPasteboardTypeRuler	com.apple.cocoa.pasteboa
NSTabularTextPboardType	NSPasteboardTypeTabularText	com.apple.cocoa.pasteboa
NSMultipleTextSelectionPboardType	NSPasteboardTypeMultipleTextSelection	com.apple.cocoa.pasteboa
NSFindPanelSearchOptionsPboardType	NSPasteboardTypeFindPanelSearchOptions	com.apple.cocoa.pasteboa

Constants for Common Pasteboard Types with Existing UTIs

The following table shows pasteboard types for which there are existing UTIs. The table shows the OS X v10.5 and earlier constant (where there is one), the OS X v10.6 constant, and the UTI string.

Old Constant	New Constant	UTI String
NSStringPboardType	NSPasteboardTypeString	public.utf8-plain-text
NSPDFPboardType	NSPasteboardTypePDF	com.adobe.pdf
NSRTFPboardType	NSPasteboardTypeRTF	public.rtf
NSRTFDPboardType	NSPasteboardTypeRTFD	com.apple.flat-rtfd
NSTIFFPboardType	NSPasteboardTypeTIFF	public.tiff
	NSPasteboardTypePNG	public.png
NSHTMLPboardType	NSPasteboardTypeHTML	public.html

Pasteboard Types Without Direct Replacement

Some OS X v10.5 constants do not have corresponding constant definitions on OS X v10.6. The following table shows either their deprecation status or what you should use instead.

Constant	Replacement/Comments
NSPICTPboardType	Deprecated in OS X v10.6
NSFilesPromisePboardType	Use `(NSString *)kPasteboardTypeFileURLPromise` instead.
NSVCardPboardType	Use `(NSString *)kUTTypeVCard` instead.

Constant	Replacement/Comments
NSPostScriptPboardType	Use `@"com.adobe.encapsulated-postscript"` instead.
NSInkTextPboardType	Use `(NSString *)kUTTypeInkText` instead.
NSURLPboardType	Use `writeObjects:` to write `NSURL` objects directly to pasteboard.
NSFilenamesPboardType	Use `writeObjects:` to write file `NSURL` objects directly to pasteboard.

Document Revision History

This table describes the changes to *Pasteboard Programming Guide*.

Date	Notes
2010-09-01	Corrected initialization methods described in a code sample in "Custom Data".
2009-05-28	New document that describes the pasteboard programming interfaces for Mac OS x v10.6 and later.

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