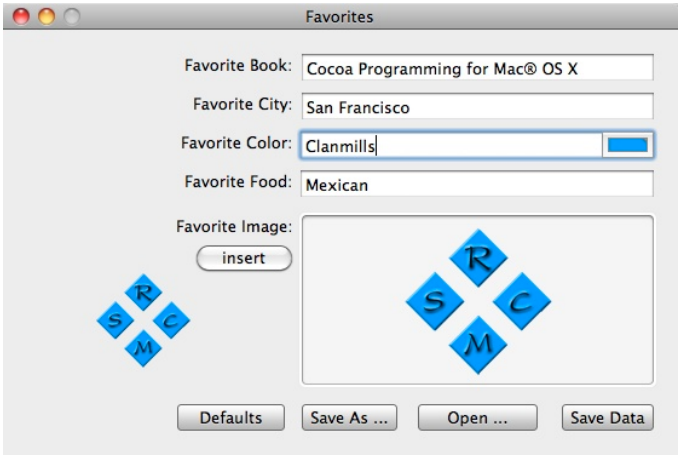
FAVORITE DOCUMENT TUTORIAL

SAVING AND READING DOCUMENTS
INTERACTING WITH THE MENU

This tutorial is a continuation of the FavoritesTutorial which I developed with Mr Prince Kumar, a reader on http://www.cocoaforum.com/ Prince asked for assistance with an application he'd developed as a learning exercise. The application updates some favorites which are stored on disk as ~/Library/Preferences/com.clanmills.Favorites.plist. Prince asked "how would I convert this to a document, so that it reads and writes documents?" This tutorial is my response to that request.

I have used the NSDocument class to enable the data implied by the User Interface to be archived to disk (and unarchived to memory). I hope that by extending the Favorite's Tutorial, the behavior of the NSDocument class will by demystified.



You can download the code from: **http://clanmills.com/files/FavoriteDocument.zip**

All errors are mine and I hope you'll let me know if you are not able to follow the recipe. And of course if you don't understand how/why this all works, I'll be happy to explain a little more. You are also welcome to VNC to my computer for a live demo.

I don't intend to repeat the information in the Favorite tutorial. You can start immediately in this tutorial and download and build the code from: **http://clanmills.com/files/Favorites.zip**
The Favorites tutorial is available from **http://clanmills.com/files/Favorites.pdf**

The subjects which we deal with in this tutorial are:

1) How to interact with the menu (and recursively dump menus to output)

2) How to save and load documents to/from the disk

3) How to use the OpenFile and SaveFile Panels

4) Some tricks in refactoring code

ACKNOWLEDGEMENTS

These notes are published under the GPL License. Mr Kumar (the original author) has kindly agreed to his code being published under GPL. If you are concerned about copyright, please drop me an email.

I'd like to thank Prince for his help in getting this tutorial written. Thanks, Prince - it was fun working with you and talking on Skype.

ADDING OUR NEW BUTTONS

1) Refactoring the Project and Code

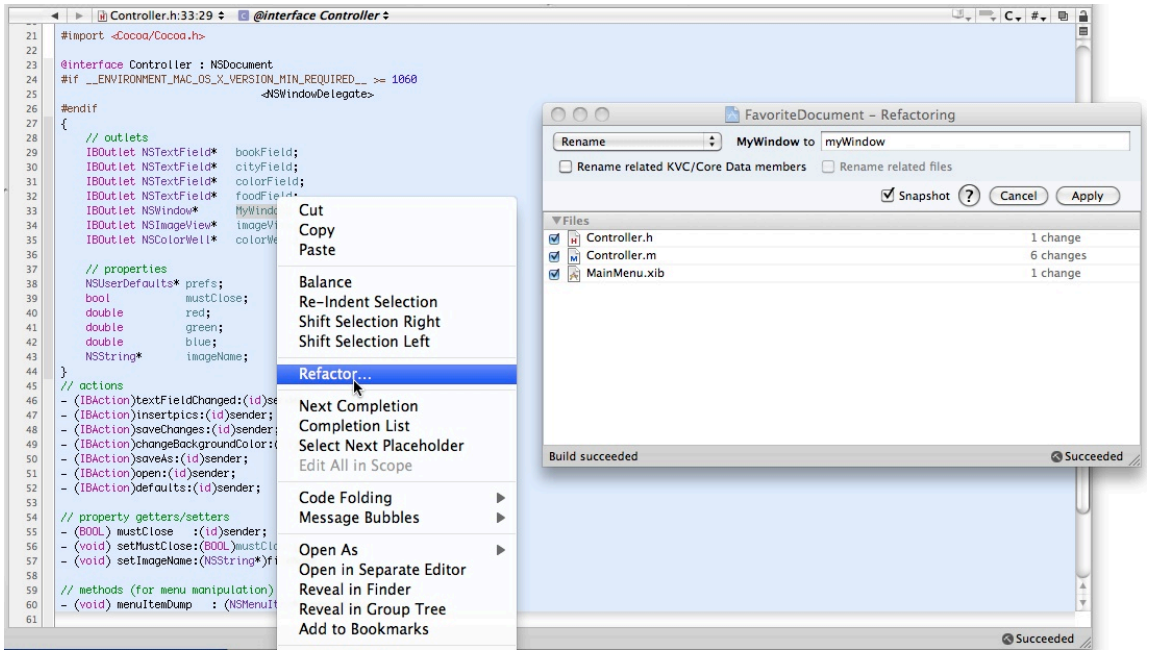
I want to base the new application on Favorites.xcodeproj. However I want to call it FavoriteDocument. So, please do the following:

1) rename the Favorites directory to FavoriteDocument
2) rename Favorites.xcodeproj to FavoriteDocument.xcode
3) rename the file Favorites_Prefix.pch to FavoriteDocument_Prefix.pch
4) edit FavoriteDocument/project.pbxproj and globally replace Favorites with FavoriteDocuments
5) rm FavoriteDocument.xcodeproj/rmills.* FavoriteDocument.xcodeproj/prince.* FavoriteDocument/*.icns
6) Open FavoriteDocument.xcodeproj, built and run.

Congratulations, you've duplicated and created a new project. You should have an application called FavoriteDocument.app

Open the file Controller.h in XCode. Select the variable MyWindows, right click and select 'Refactor" (shift-cmd-J)

Rename this variable as myWindows, select "preview" and "apply". Save and Build. Good isn't it?



2) Update the User Interface (in Interface Builder)

Add three button "Defaults" "Save As..." and "Open..." (you can see the design on page 1 of the tutorial)

3) Update Controller.h and Controller.cpp (in XCode)

Add to Controller.h
- (IBAction) saveAs:(id)sender
- (IBAction) defaults:(id) sender
- (IBAction) open:(id)sender

Add to Controller.cpp
- (IBAction) saveAs:(id) sender { NSLog(@"saveAs...") ; }
- (IBAction) defaults:(id) sender { NSLog(@"defaults") ; }
- (IBAction) open:(id) sender {NSLog(@"open");}

4) Update your outlets in Interface Builder

Right-click on the "Controller" cube and connection your "Defaults" button to defaults:"SaveAs" to saveAs: and "Open" to open:

Save in Interface Builder. Build and Run in XCode. Press your button and verify that your NSLog() messages are in the console.

INTERACTING WITH THE MENU

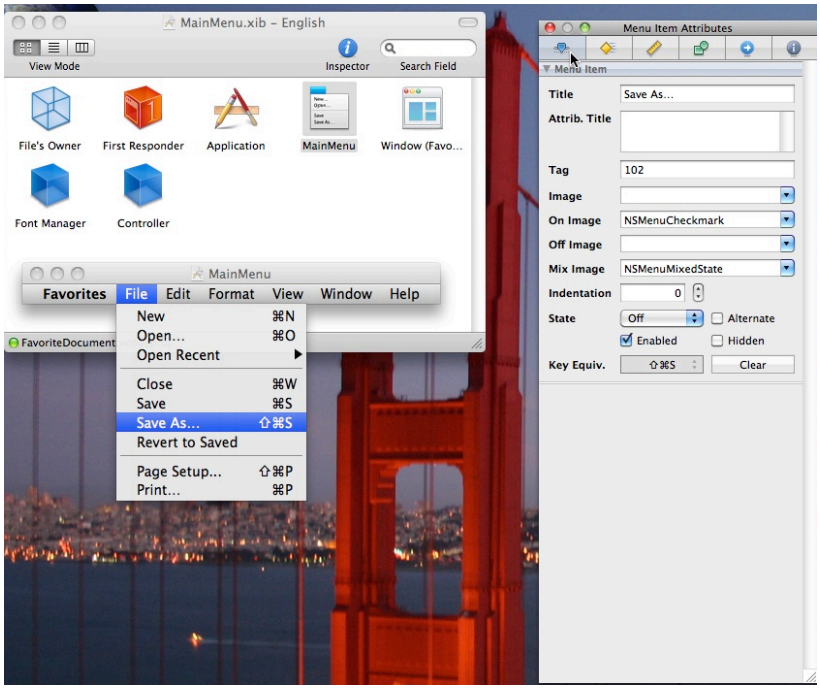
1) Dumping and Enabling Menus

I added code in 4 methods menuDump, menuItemDump, menuEnable and menuItemEnable. These are declared in Controller.h and implemented in Controller.m. Get the code for these function (from clanmills.com/files/FavoriteDocument.zip) and add it. I'll discuss the code later. For the moment, simply cut'n'paste the code from clanmills.com for those functions.

2) Enabling the Open and Save As Menu Items

When you run the application, you'll be rather disappointed to see that although you have a button marked "Open" and one marked "Save As", the corresponding elements in the Menu are grayed out. We have to enable them.

In Interface Builder's main windows (MainMenu.xib), there is a item "MainMenu". Double Click to reveal the Menu. Navigate to the Save As ... item, and inspect its properties with the Inspector (shift-cmd I)



Assign the Menu Item Attribute "Tag" of 101 to "Open" and 102 to "Save As".

Update Controller.mm awakeFromNib with the additional code:

```
NSMenu* mm = [NSApp menu];
[self menuEnable:mm tag:101];
[self menuEnable:mm tag:102];
```

Build and run you application. Save As and Open should not be Enabled (however ineffective).

3) Connect the Open and Save As Menu Items to your controller code

In Interface Builder, Right Click on the Controller "Cube" and connect saveAs: to the Menu Item "Save As" (and open: to "Open"). You may find it more intuitive to ctrl-Drag the Menu Item "Save As" to the Controller "Cube" and release the mouse. You'll be offered a menu and you may select the appropriate method.

Interface Builder offers more that one way to do many things. Which ever makes you feel comfortable is the one to use.

4) Dumping and Enabling Menus

Build and run. All should be well and happy. You should be able to push the buttons, or use the short-cut keys, or use the menus and see you NSLog() messages. Good stuff. Time for a break and some coffee.

A LITTLE MORE REFACTORING

We're going to have to refactor a little more code.

The original application interacts with the preferences using the NSUserDefaults class. However we're going to enable you to save the preferences into a regular file as a document. And we will provide code to read a regular document into memory. The code to display the current preferences will have be refactored to enable it to be called from the open: method. We're going to call that updateUI:

I've also add the "Defaults" button which is going to set our initial values in memory. In the existing code, this is in the init method, so we're going to re-factor that as a a separate method which we will call - (NSDictionary*) defaultPrefs

So here goes. This isn't much fun, however I'm sure you'll be really pleased with the result. And for further motivation, remember that most engineers spend a lot of time refactoring existing code and less time writing new code. So this is a good way to build confidence that code can be quite easily refactored in XCode.

1) We want our Controller to be derived from NSDocument (and not NSObject)

Modify Controller.h to be `@interface Controller : NSDocument`

Add the following methods to Controller.h

```
// methods (for data manipulation)
- (NSDictionary*) defaultPrefs;
- (void) updateUI:(NSDictionary*)dictionary;
```

The XCode short-cut opt-cmd-↑ (uparrow) is very useful for switching between a .h and .m file.

2) defaultPrefs

The defaults code is in the existing init: method. Move it to its own function defaultPrefs.

```
- (NSDictionary*) defaultPrefs
{
      NSMutableDictionary * defaultprefs = [NSMutableDictionary dictionary];

      NSString* favImage = [NSString stringWithFormat:@"%@%s"
                  ,[[NSBundle mainBundle]resourcePath],"/clanmills.gif"];

      [defaultprefs setObject:favImage forKey:FAV_IMAGE];
      [defaultprefs setObject:@"Cocoa Programming for Mac® OS X" forKey:FAV_BOOK];
      [defaultprefs setObject:@"San Francisco" forKey:FAV_CITY];
      [defaultprefs setObject:@"Clanmills" forKey:FAV_COLOR];
      [defaultprefs setObject:@"Mexican" forKey:FAV_FOOD];

      [defaultprefs setObject:[NSNumber numberWithDouble:0.0] forKey:FAV_RED];
      [defaultprefs setObject:[NSNumber numberWithDouble:0.6] forKey:FAV_GREEN];
      [defaultprefs setObject:[NSNumber numberWithDouble:1.0] forKey:FAV_BLUE];

      return defaultprefs;
}
```

You can now simply the init: method by calling defaultsPrefs:

```
- (id) init
{
      [super init];

      prefs = [[NSUserDefaults standardUserDefaults] retain];
      [prefs registerDefaults:[self defaultPrefs]];
      [self setMustClose:NO];
      return self;
}
```

3) updateUI

The code to update the UI is in awakeFromNib: Move it to it's own function updateUI.

When we call updateUI, there are two very different use cases:

In one case (for example when the users presses the button "Defaults", we want to change the values and set them in the UI. The call will be something like [self updateUI : [self defaultPrefs]] A similar use case occurs when we read the document from disk and we will discuss this later.

The other use case is when we simply want the UI to display the current settings. This is the call we make from awakeFromNib when the call is simply [self updateUI:nil];

```
- (IBAction) updateUI:(NSDictionary*) dictionary
{
      NSColor* color;
      if ( dictionary ) {
          for (id key in dictionary) {
                  NSObject* object = [dictionary objectForKey:key];
                  NSLog(@"key: %@, value: %@", key, object);
                  [prefs setObject:object forKey:key];
          }
      }
      [bookField setStringValue:[prefs stringForKey:FAV_BOOK]];
      [cityField setStringValue:[prefs stringForKey:FAV_CITY]];
      [colorField setStringValue:[prefs stringForKey:FAV_COLOR]];
      [foodField setStringValue:[prefs stringForKey:FAV_FOOD]];
      [self setImageName:[prefs stringForKey:FAV_IMAGE]];

      red   = [[prefs stringForKey:FAV_RED]doubleValue];
      green = [[prefs stringForKey:FAV_GREEN]doubleValue];
      blue  = [[prefs stringForKey:FAV_BLUE]doubleValue];

      color = [NSColor colorWithDeviceRed:red
                  green:green
                  blue:blue
                  alpha:1.0
      ];
      [colorWell setColor:color] ;
}
```

You can now update awakeFromNib to call updateUI

```
- (void) awakeFromNib
{
      [NSApp setDelegate:self];
      [MyWindow setDelegate:self];
      [self updateUI:nil];

      [self menuEnable:mm tag:101];
      [self menuEnable:mm tag:102];
}
```

4) Build and run the application.

Finish your coffee, your almost finished.

AND NOW TO READ AND WRITE DOCUMENT

1) How Documents get written to and from disk

They are archived. This is a two step process. All the Foundation classes (such as NSArray, NSDictionary, NSString and so on) know how to write them self to disk (as XML) (and they know how to read themselves). So we have very little to do!

The NSDocument class requires you to implement one several callbacks. Please read the comments in the code which were obtained using the XCode wizard. I actually generated an NSDocument based application and read the code to discover how it did this. I chose to implement dataOfType: error: to write the document and readFromData ofType: error: to read the data.

2) NSDocument dataOfType:error:

This function is called to convert your document into a file. An NSData* is a sequence of bytes. You could return anything. You could for example use 'sprintf' to create a byte sequence in a char buffer[] 'C' structure. And if your corresponding readFromData:ofType:error method knows how to decode it, well you're done.

However I strongly discourage you from doing anything like this. Apple have provide robust solutions to meet your needs and your life will involve less pain when you build on the solutions provided by the Cocoa Framework.

In this case, I simple create a dictionary and copy our preferences to 'toBeSaved' and the call to return [NSKeyedArchiver archievedDataWithRootObject::toBeSaved]; does all the work.

```
- (NSData *)dataOfType:(NSString *)typeName error:(NSError **)outError
{
      // Insert code here to write your document to data of the specified type.
      // If the given outError != NULL, ensure that you set *outError when returning nil.

      // You can also choose to override
      //                  -fileWrapperOfType:error:
      //                  -writeToURL:ofType:error:
      //                  -writeToURL:ofType:forSaveOperation:originalContentsURL:error:
      // instead.

      if ( outError != NULL ) {
                  *outError = [NSError errorWithDomain:NSOSStatusErrorDomain code:unimpErr userInfo:NULL];
      }

      NSMutableDictionary* toBeSaved = [NSMutableDictionary dictionaryWithCapacity:10];
      [toBeSaved setObject:[bookField stringValue]          forKey:FAV_BOOK];
      [toBeSaved setObject:[cityField stringValue]          forKey:FAV_CITY];
      [toBeSaved setObject:[colorField stringValue]         forKey:FAV_COLOR];
      [toBeSaved setObject:[foodField stringValue]          forKey:FAV_FOOD];

      [toBeSaved setObject:imageName
      forKey:FAV_IMAGE];
      [toBeSaved setObject:[NSNumber numberWithDouble:red]   forKey:FAV_RED];
      [toBeSaved setObject:[NSNumber numberWithDouble:green] forKey:FAV_GREEN];
      [toBeSaved setObject:[NSNumber numberWithDouble:blue]  forKey:FAV_BLUE];

      // return nil;
      return [NSKeyedArchiver archivedDataWithRootObject:toBeSaved];
}
```

3) readFromData:typeName:error

This function part of NSDocument and is called to convert your document into a data in memory. An NSData* is a sequence of bytes. So we call [NSKeyedUnarchiver unarchiveObjectWithData] to recover the data.

Notice the beautiful symmetry here. We write the data with NSKeyedArchiver and we read the data with NSKeyedUnarchiver. We don't even know the data format (however it is XML). Beautiful and so simple.

So having recovered the dictionary (favs), I update the UserInterface and return YES.

```
- (BOOL)readFromData:(NSData *)data ofType:(NSString *)typeName error:(NSError **)outError
{
      BOOL result = NO;
      // Insert code here to read your document from the given data of the specified type.
      // If the given outError != NULL, ensure that you set *outError when returning NO.
      // You can also choose to override
      //          -readFromFileWrapper:ofType:error:
      //or        -readFromURL:ofType:error:
      // instead.

      if ( outError != NULL ) {
                  *outError = [NSError errorWithDomain:NSOSStatusErrorDomain code:unimpErr userInfo:NULL];
      }

      NSDictionary* favs =  [NSKeyedUnarchiver unarchiveObjectWithData:data];
      if ( favs ) {
          result = YES ;
         [self updateUI:favs];
      }

      return result ;
}
```

GETTING THE FILE PANELS TO SHOW

1) File Open Panel (so you can read)

This is quite straightforward. You'll probably want to modify open as follows:

```
- (void) openPanelDidEnd:(NSOpenPanel *)op
                          returnCode:(int)returnCode
                         contextInfo:(void *)ci
{
      if (returnCode == NSOKButton) {
                  NSError* error = nil;
                  NSString *path = [op filename];
                  NSURL* url = [NSURL URLWithString:[NSString stringWithFormat:@"file:///%@",path]];
                  NSLog(@"path = %@",path) ;
                  NSData* data = [NSData dataWithContentsOfURL:url options:NSDataReadingUncached error:&error] ;
                  [self readFromData:data ofType:EXT error:&error];
      }
}

- (IBAction)open:(id)sender;
{
      NSLog(@"open");
      [self setFileType:EXT];

      NSOpenPanel *op = [NSOpenPanel openPanel];

      [op beginSheetForDirectory:nil
                            file:nil
                           types:nil
                  modalForWindow:myWindow
                   modalDelegate:self
                  didEndSelector:@selector(openPanelDidEnd:returnCode:contextInfo:)
                     contextInfo:nil];
}
```

You'll appreciate how this works (in open:). You create an NSOpenPanel object, which you attach to your window. You provide a delegate (callback) which will be called when you select a file. I've added the NSLog() code so you can see that this is working.

The code NSData* data = bla bla actually opens a disk file and reads the bytes into memory. From there you only have to call data:ofType:error: to convert the bytes into data.

2) File Save Panel (to write your document to disk)

The NSDocument saveDocumentTo: method takes care of everything and calls you dataOfType:error method

```
- (IBAction) saveAs:(id) sender
{
      NSLog(@"writeFile");
      [self setFileType:EXT];

      [self saveDocumentTo:sender];
}
```

Build and run the application.

Run the application and try your "SaveAs..." button (and MenuItem File/Save As...) (and short-Cut cmd-shift-S).
Inspect your file. It's XML. Have a little read.
Change some values and try your "Open" button (and MenuItem File/Open) (and short-Cut cmd-O)

OTHER MATTERS

1) menuDump and menuItemDump

These function are not required to make FavoriteDocument function. However they will help you understand the nature of a menu. A menu is a collection of course and there are two kinds of objects in a menu. There are MenuItem objects and they are, for the most part, Strings or Menus.

So the menu is a recursive structure - very like the file system. By analogy, a Menu = a directory, a MenuItem = a file. So function menuDump, enumerates the menu and calls menuItemDump on the menuItems, and menuDump on the menus. Please read the code and think a little about how this works.

2) menuEnable and menuItemEnable

After writing menuDump: and menuItemDump, it was of course a quick cut'n'paste job to write those functions. If you've understand menuDump and menuItemDump, you'll appreciate almost instantly how menuEnable does its work.

CHALLENGES

To be written.

LICENSE

Robin Mills
Software Engineer

Windows/Mac/Linux
C++, Web, JavaScript, UI

400 N First St #311
San Jose, CA 95112

T (408) 288 7673
C (408) 394 1534
robin@clanmills.com

http://clanmills.com

Revision History

20100416	Initial version