Chapter 2

DEFAULTS AND USER PREFERENCES

CHAPTER 2 DEFAULTS AND USER PREFERENCES

Goal

To explore the Foundation classes, user interface objects and designs for building a multi-view preferences component with persistent user defaults.

Prerequisites

Familiarity with user preference interfaces of some typical applications.

Objectives

At the end of this section, you will be able to:

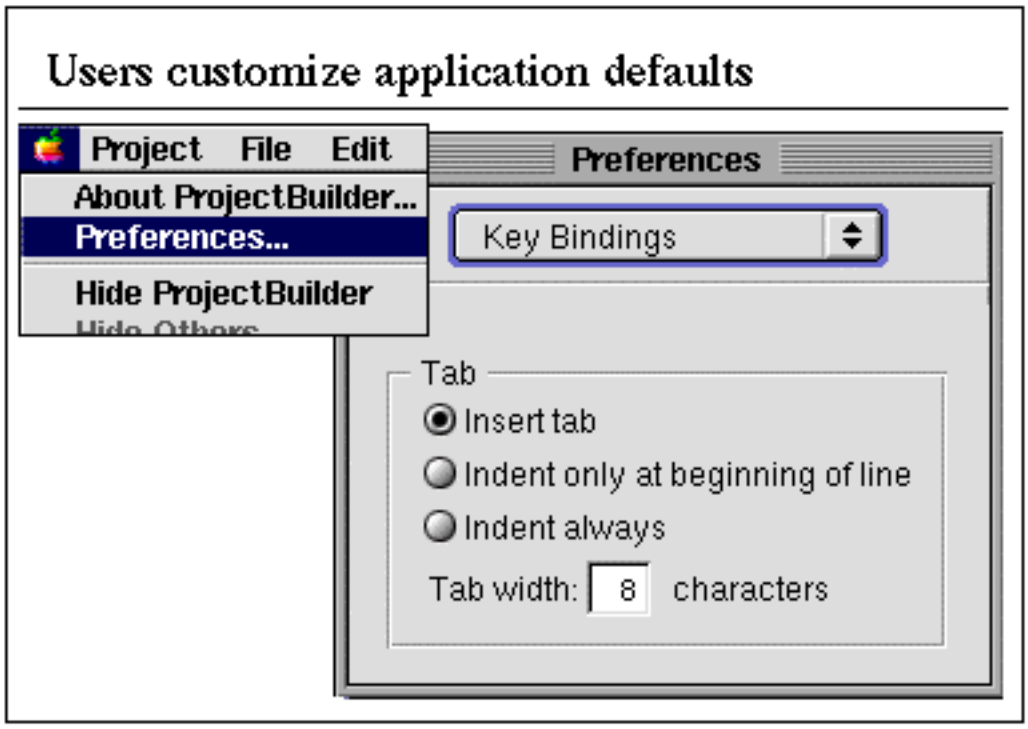
- Register application defaults and dynamically get and set user default values.
- Describe how NSDictionary stores key-value pairs.
- Implement a view switching panel

Reading

NSUserDefaults class reference in the Foundation.

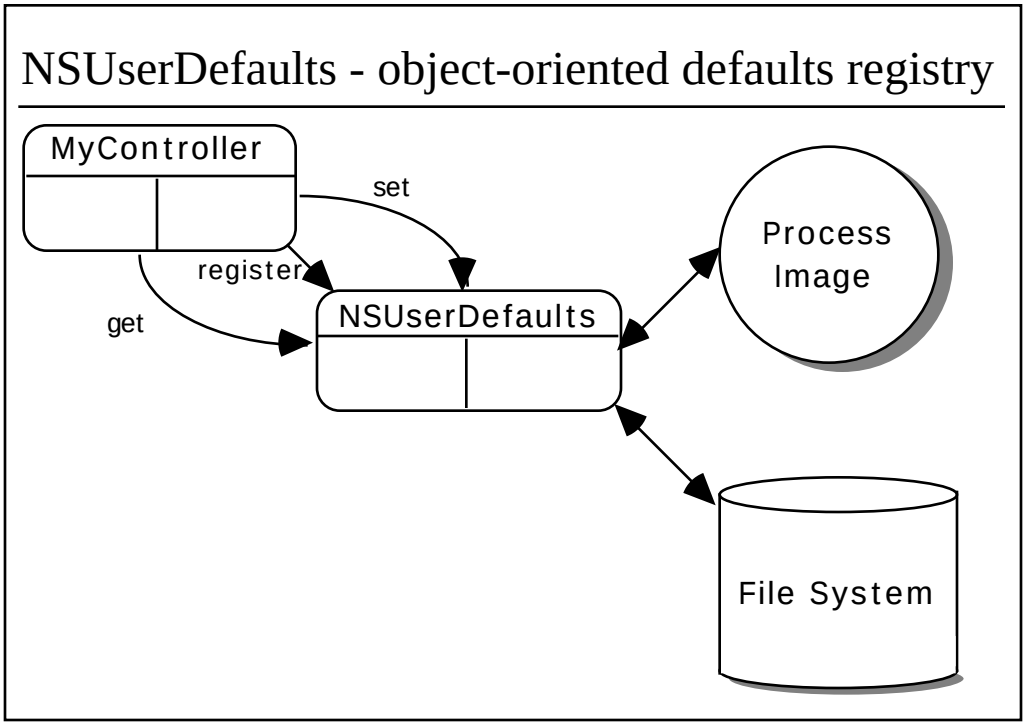
NSDictionary class reference in the Foundation.

NSPopUpButton class reference in the Application Kit.



Users customize application defaults

Most applications provide customizable options to suit the preferences of individual users. Examples include things like default fonts, auto-save features, dynamically loadable application extensions and the ability to enable or disable elaborate or time-consuming features. The application has a set of “factory settings” that provide default choices in lieu of a user’s particular preference. Once the user has modified these, the application must remember the new values and put them into effect every time it is launched by that user. The Foundation provides an object for managing this and the Application Kit provides some convenient features for implementing a typical graphical component with a flexible interface.



NSUserDefaults—object-oriented defaults registry

The NSUserDefault object provides API for managing persistent as well as dynamically changing application parameters. It stores values in the file system for future use and maintains a collection of current values in the application's memory space so that values can be queried at run time. Your application can communicate with user defaults in the following general manner:

- register—during application start-up, register the default "factory settings" for your application
- get—when the application needs a particular value to configure its user interface or its behavior, it retrieves it from the repository
- set—if and when a user configures a non-default setting, the application sets this value in the user defaults repository where it will be saved and returned for future get actions

In this way, the application gets a value when it needs it, unaware of whether the value is a registered default, a value customized by the user or even a temporary setting just for the current invocation of the application.

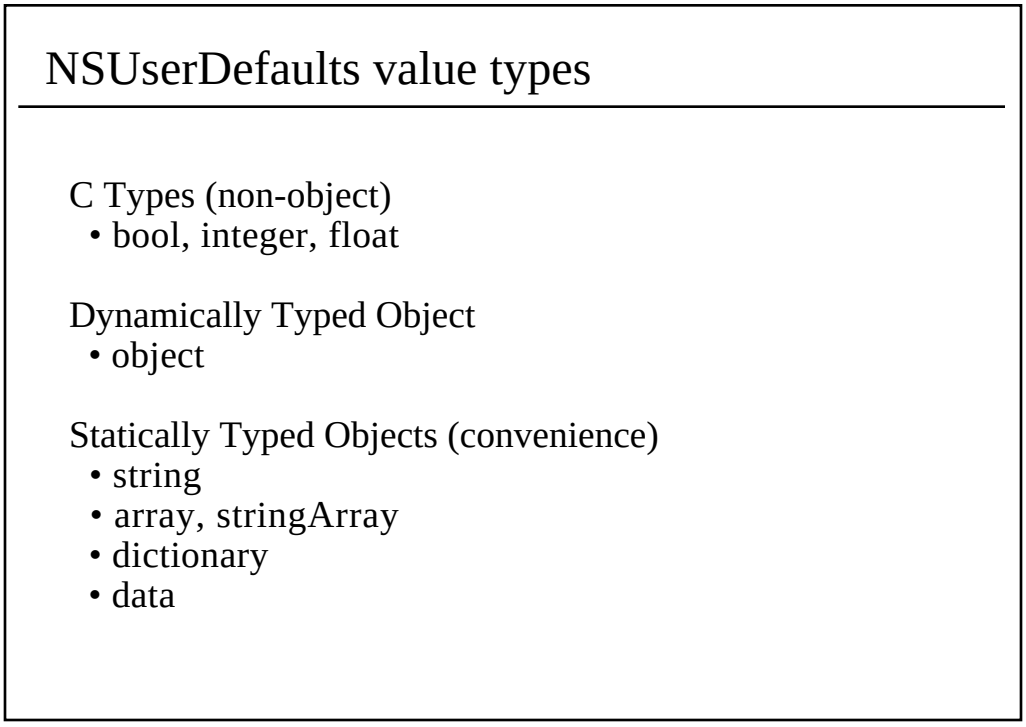
Domains - properties and defaults search order

DOMAIN	IDENTIFIER	PERSISTENT?
Argument	NSArgumentDomain	NO
Application	*Application Name*	YES
Global	NSGlobalDomain	YES
Language	*Prefered Language Name*	NO
Registration	NSRegistrationDomain	NO

Domains—properties and default search order

Default values may be specified in several different layers that form a hierarchy. Each layer is called a domain which has a name and an search order relative to the other domains. Some domains store values that will persist while others are said to be volatile—their values only live for this invocation of the application. Together, they form a coherent hierarchy where one domain can override the other. When a value is sought by the application, the domains are consulted in a fixed order, one at a time. The search stops as soon as one of the domains provides a value or until all domains are exhausted.

- Argument—the highest priority, this represents values specified on the command line that launched the application. They apply strictly to this one-time invocation of the application
- Application—this layer provides the persistent values customized by the user for this particular application
- Global—these are persistent values, customized by the user, that potentially apply to all applications unless overridden by a particular application's domain
- Language—these apply to a user's language preferences and relate to features that are locale-specific
- Registration—the lowest level in the hierarchy, these are the factory settings provided by the application to be used in lieu of any customizations in the higher domains



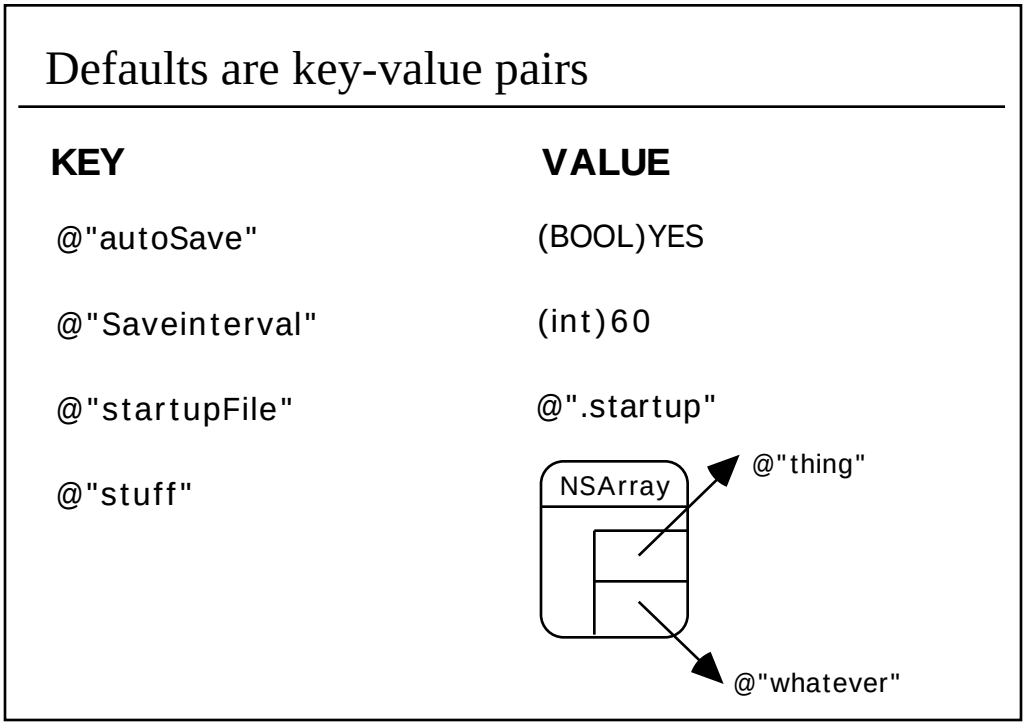
NSUserDefaults value types

What types of values can the NSUserDefault object deal with? It has methods for handling the full array of values your application is likely to need:

- C types—not objects, these are the basic C primitives
- Dynamically typed object—that is, type id. Note, it must be a kind of value object, capable of archiving and unarchiving and copying itself. It must conform to the NSCoding and NSCopying protocols
- Statically typed objects—these specific object types are used to build property lists and so have explicit API in the NSUserDefault object. It is likely that you will use these most frequently

As listed here, each type has two corresponding accessor methods in the NSUserDefault object. Using string as an example, you can get and set a string value with the following methods:

```
NSUserDefaults *defaults = [NSUserDefaults
standardUserDefaults];

[defaults setString: aString] // set

aString = [defaults string] // get
```

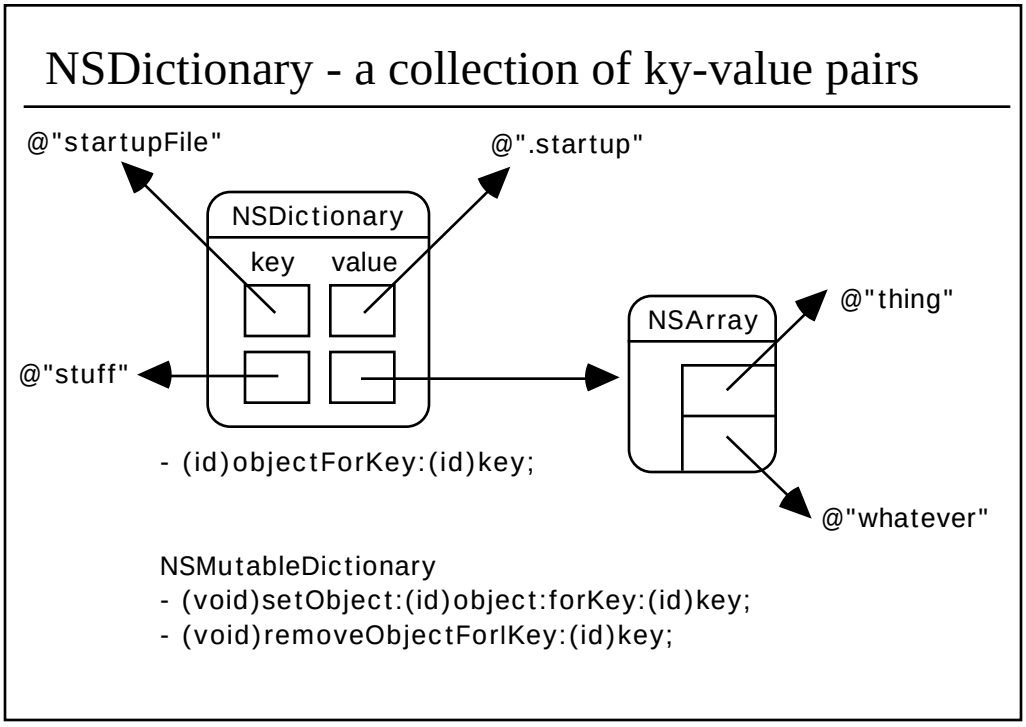


Defaults are key-value pairs

Values stored by NSUserDefaults must have two pieces:

- key—the name of the parameter used to distinguish it from all others
- value—the current value of the parameter

Keys are typically strings you choose to be meaningful to your application code and possibly to a user who wishes to modify the value directly from the command line. A key can actually be any value object class—hashable, copiabled, and archivable. The value can be any one of several value or collection object classes including arbitrarily complex objects like arrays of strings.



NSDictionary—a collection of key-value pairs

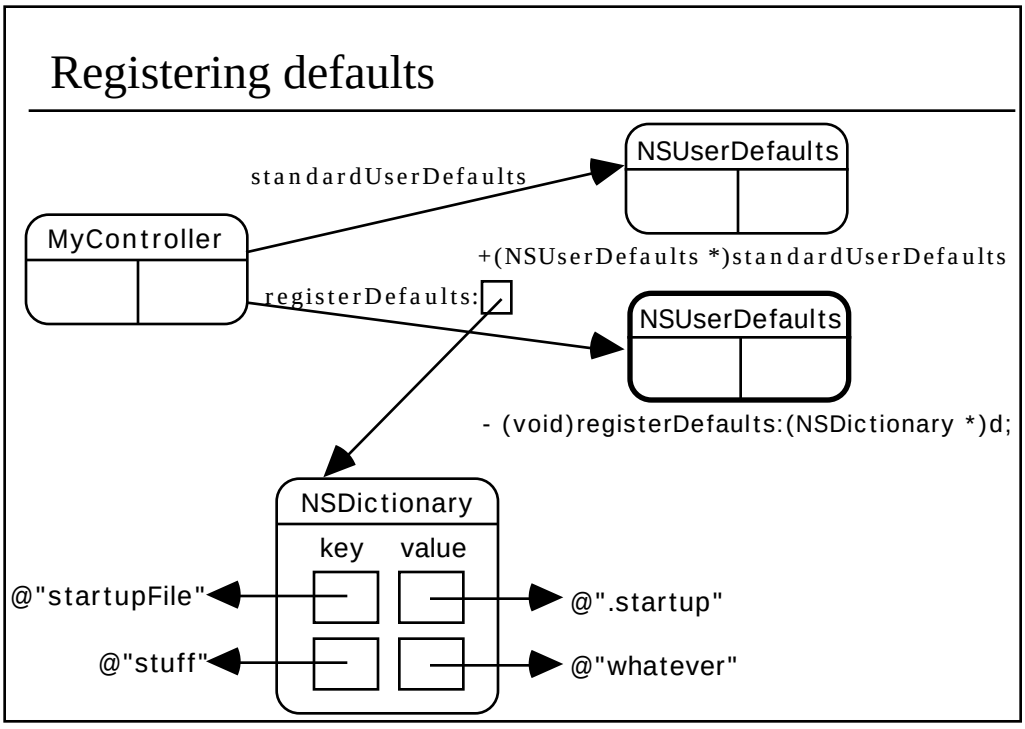
Representing parameters as key-value pair is so common that the Foundation provides an object to managing collections of them. NSDictionary stores a set of key-value pairs much like what are generally called associative arrays. You can think of NSDictionary as an object-oriented hash table. Instead of accessing a value with a numerical index, you access it with a meaningful string—the value's name.

Like many Foundation classes, NSDictionaries come in two forms:

- Immutable—a read-only object efficient for sharing
- Mutable—an instance you can modify by adding or deleting values

Value objects can be retrieved using **objectForKey:**, set using **setObject:forKey:** and removed altogether with **removeObjectForKey:**. There are a number of ways to instantiate a dictionary, including loading one from a property list file—a text-based representation of a dictionary saved in the file system. You should familiarize yourself with the power and flexibility of this useful Foundation class.

objectForKey: returns nil to indicate that there is not a value for the specified key in the dictionary. To avoid any possible ambiguity, you cannot use nil for either a key or a value—NSDictionary raises an exception if you do.



Registering defaults

The first step your application takes when it starts is to register the “factory settings” as the initial application defaults. Like most NSUserDefault interactions, it takes two steps:

- Get an NSUserDefault instance – **standardUserDefaults** will return the standard shared instance. You can allocate custom instances for more exotic application needs
- Register the defaults—pass an NSDictionary of key-value pairs using the message **registerDefaults:**.

When should you register defaults?

```
+ (void)initialize
{
    NSUserDefaults *defaults;
    NSMutableDictionary *values;

    if (self == [MyController class]) {
        // get standard UserDefaults instance
        defaults = [NSUserDefaults standardUserDefaults];

        // build dictionary of registered defaults
        values = [[NSMutableDictionary alloc] init];
        [values setObject:@".startup" forKey:@"startupFile"];

        // Register
        [defaults registerDefaults: values];
        [values release];
    }
}
```

When should you register defaults?

You need to register a default value before any part of your application attempts to get it. Otherwise, it may not be any of the domains in which case it is undefined. Objective-C provides for a class to initialize itself before any other messages are sent, all other factory or instance methods included. Your class can implement + (void)**initialize**, a factory method, and perform its UserDefault registration there. **initialize** is sent to the class object just before it is sent its first message from within the program. This implies that the class must receive a message before any other application object tries to get a UserDefaults value.

Which class should handle this responsibility? It might be your application controller or a controller object that manages your application's preference component. You might want to split your defaults into subsets that are managed by different objects. The important thing is you must message this class and provoke **initialize** before any other UserDefaults activity. Often this is accomplished by instantiating the relevant controller in the main nib so that its class is messaged early during application start-up.

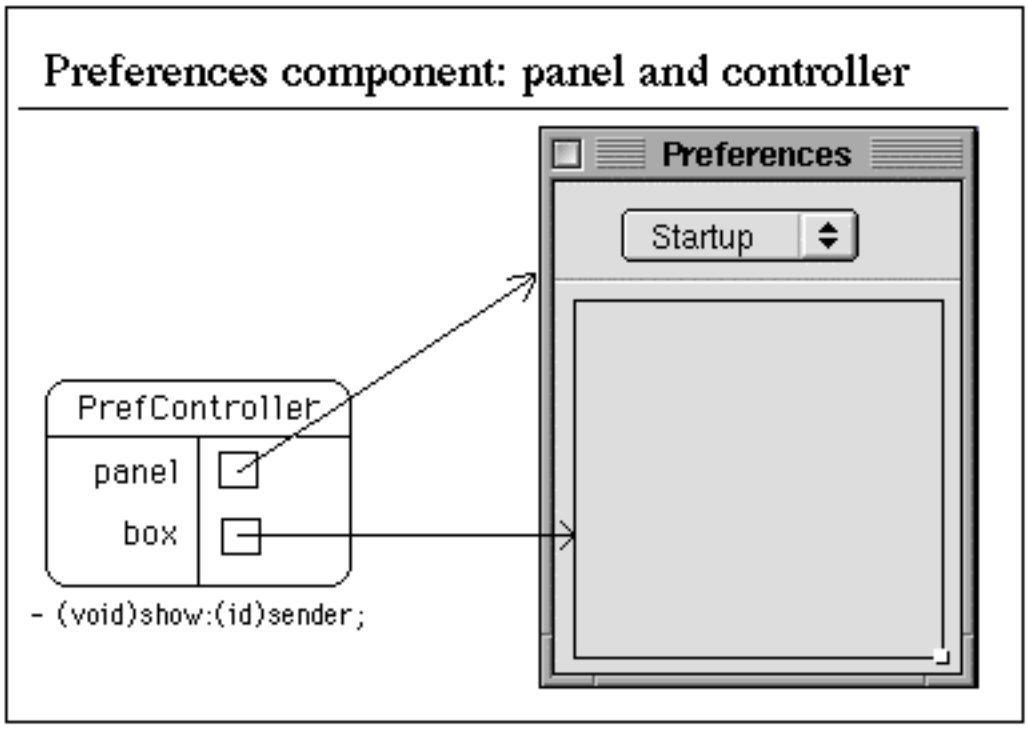
This example demonstrates how you might register your defaults. Instead of hardcoding your default values, you might read them from a separate data file located in your main bundle and accessible through NSBundle. See NSDictionary and the method **+ dictionaryWithContentsOfFile:**.

Getting and setting defaults

```
NSUserDefaults *defaults;
NSString *file;
int time;

// Get standard defaults instance
defaults = [NSUserDefaults standardUserDefaults];

// Getting values
time = [defaults integerForKey:@"interval"];
file = [defaults stringForKey:@"startupFile"];

// Setting values
[defaults setInteger:time forKey:@"interval"];
[defaults setObject:file forKey:@"startupFile"];
```

Getting and setting defaults

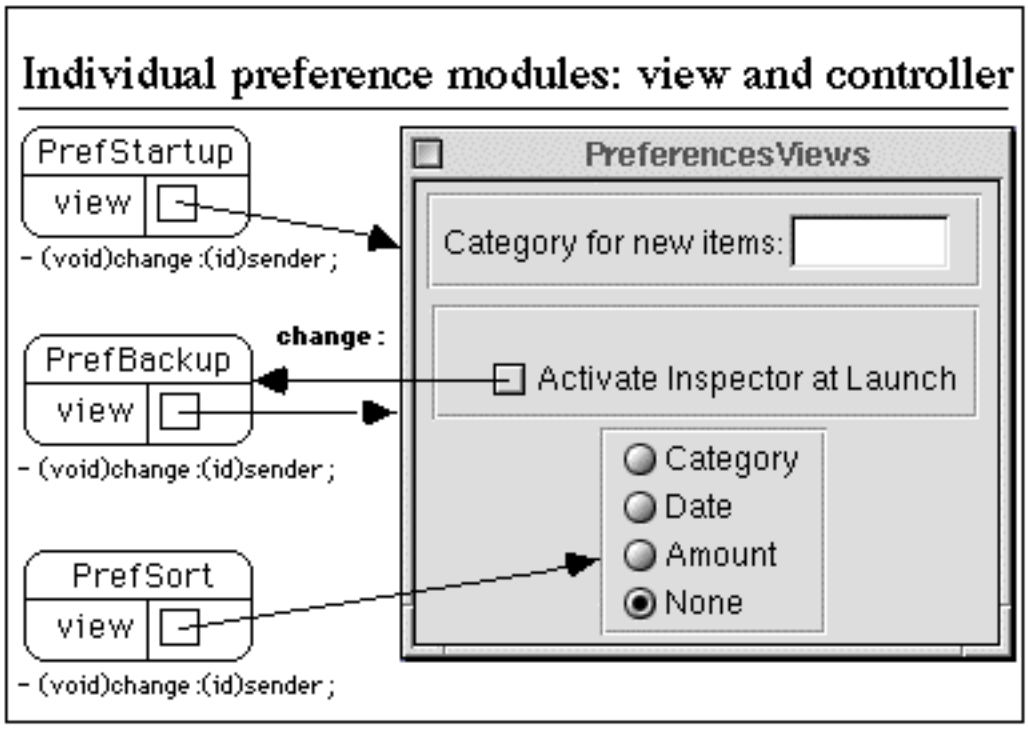
You get NSUserDefault values exactly when you need them—during application start-up, after loading a separate nib file based component, during the course of a procedure, when the user asks to see the current settings. Setting default values is almost always driven by the user's interaction with your preferences user interface.



Preferences component: panel and controller

Applications typically have many different configurable preferences that together require a flexible user interface for management. A common way of designing for this is to provide a panel that can dynamically switch among different views within the panel, one per logical group of configuration parameters. You might have one view for start-up options, another for backup options and so on. View-switching within a single window is a useful feature and it can be used in a variety of places throughout your application—preferences panel, inspectors, tab views and so forth. The next few pages present one of many approaches for implementing this feature with a hypothetical preferences component in mind.

A familiar basic design—a component controller with a separate nib-based panel. In this case, the panel is rather bare. It contains simply an instance of NSBox, a place holder where any one of several different views will appear. Although the picture shows the outline of box, you would typically make its border invisible.To manage the switching of views, the controller needs an outlet to the box and well as to the panel itself.



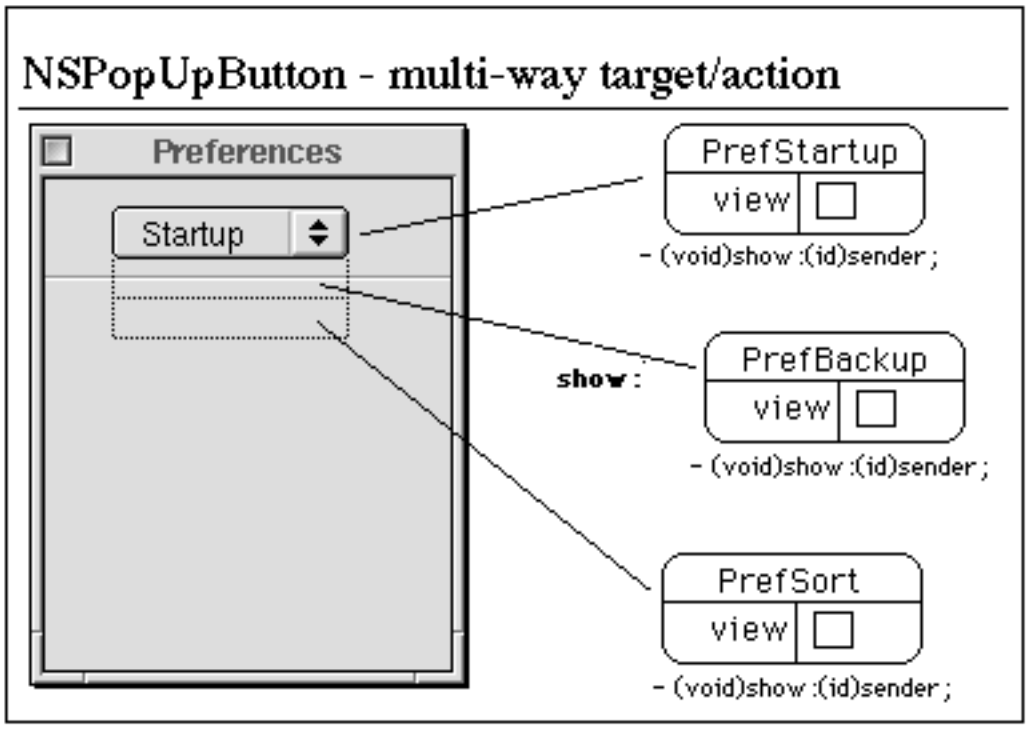
Individual preference modules: view and controller

Within the same nib file, a second window is used to build each different view. In this design, each view is placed in its own NSBox instance. The NSBox border is visible for convenience when working with Interface Builder. The design is recursive—each view has its corresponding controller much like the entire preferences panel has its controller. Each controller has the following attributes and methods:

» **view**—an outlet to the view that it owns

» additional outlets—not shown but typically required. Each controller will have to configure the values and appearance of the controls in its view to reflect UserDefault settings. The PrefBackup object, for example, is likely to have an outlet to the check button, another for the slider and the read-only textfield next to it

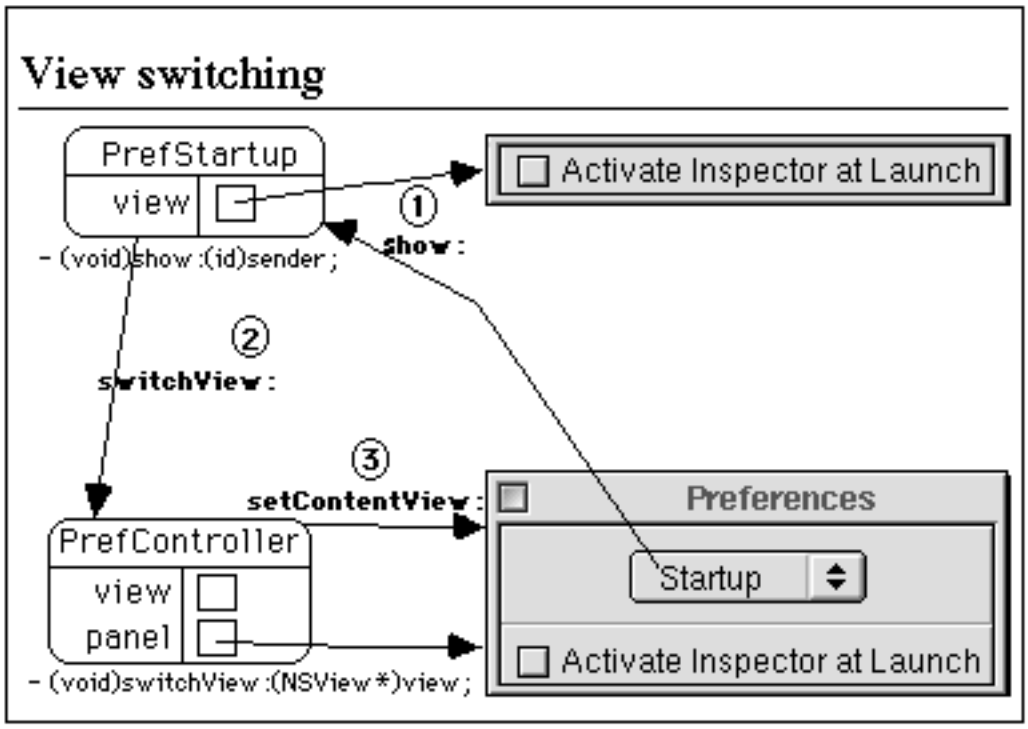
» **change:**—a message interface for the controls to tell the controller that the user has changed something. This enables the controller to read the new settings from the user interface objects and set them in the User defaults registry.



NSPopUpButton—multi-way target/action

Each preference view controller, like full panel controllers, provides a method for displaying itself. NSPopUpButton provides a perfect multi-way target/action control for messaging one of many such controllers. The user presses the desired button, it messages the corresponding view controller and it switches itself into view, filling the previously empty box on the main panel.

In many ways, NSPopUpButton is like a menu. It might even be that like menu items, NSPopUpButton items need dynamic updating to enable or disable choices as your application state changes. NSPopUpButton items do in fact conform to the NSMenuItem protocol and will be auto enabled—this is turned on by default. They will interact with a target that conforms to the NSMenuActionResponder protocol.



View switching

Like NSWindow, NSBox has a **contentView**, the top of an arbitrarily complex hierarchy of views. The preference panel has an NSBox instance as does each individual preference object. The idea is simple: when its time, set the contentView of the NSBox on the main panel to be that of the desired preference view. In this design it follows these steps:

- NSPopUpButton item activated. Sends **show:** to a preference view controller
- The view controller sends a message to the panel controller telling it to **switchView:** and passes its NSBox's **contentView** as a parameter
- The panel controller sends **setContentView:** to the main panel's NSBox, passing the preference view as a parameter

There is an aesthetic concern: what if the preference view is not the same size as the NSBox on the preference panel? Of course, it must at least be smaller to work You will find that, by default, a smaller view will end up in the lower left corner since the 0,0 origins of each are matched. Better to have it centered, possibly expanding to fill the NSBox. You can do this programmatically by adjusting the view's frame or, more conveniently, with Interface Builder's size inspector. Set each preference view so that the box and everything within expands when resized. You may have to polish the effect through trial and error.

View switching code: preferences controller

```
- (void)switchView:(NSView *)view
{
    // put the new view in place
    [box setContentView:view];

    // display it
    [box setNeedsDisplay:YES];
}
```

View switching code —preferences controller

Here is a possible **switchView:** method for the panel controller. the box outlet is type cast here because it needs to be declared as “id” for Interface Builder connections but two different Application Kit objects respond to **setContentView:** with conflicting prototypes. By explicitly declaring which of those we expect to use here, we avoid a compiler warning.

Individual preference controller

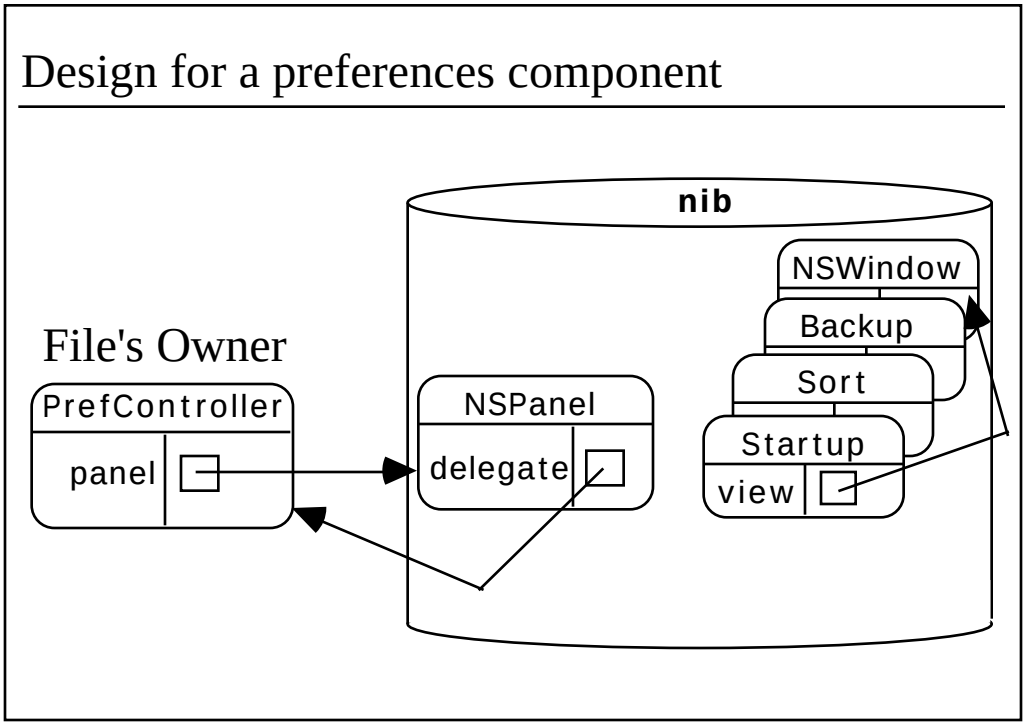
```
- (void)awakeFromNib
{
    view = [[view contentView] retain];
    // Read defaults and set panel view controls
}

- (void)show:(id)sender
{
    [[NSApp delegate] perferences] switchView:view];
}

- (void)change:(id)sender
{
    // Write defaults from panel view controls
}
```

Individual preference controller

This code belongs to the individual view controllers.

- **awakeFromNib**—set the **view** instance variable to point to the contentView of the box and retain it. By default, the view is retained by the NSBox that contains it. When it is set as the contentView of the other NSBox, it is retained there and released here (automatically by the NSBox implementation). When another view is switched in, this view is released again and no one is retaining it unless the controller retains it once here permanently. The more elaborate responsibility of **awakeFromNib** would is to read the relevant NSUserDefaults parameters and configure the user interface to reflect them
- **show:**—the view controller can get to the panel controller in a variety of ways depending on outlets and connections. This code assumes that AppController has a method for dispensing the preferences controller
- **change:**—the user modified a value in the view. Fetch it and write it back to the NSUserDefaults object

Since each preference view controller has to implement the common code shown above, it might be great opportunity to factor out an abstract superclass from which each of the view controller can subclass.



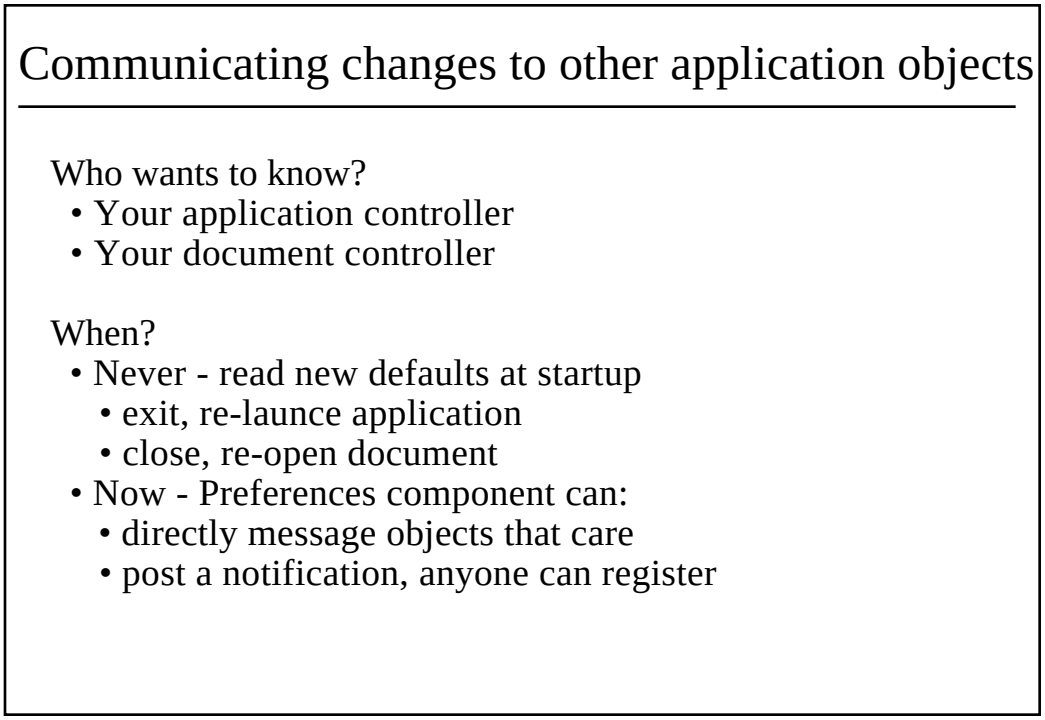
Design for a preferences component

Where should the NSUserDefault registration go? Remember, it should be in the **+initialize** method of some class so that preferences are registered before anyone asks.

This all depends on how dynamic your application is—whether objects are instantiated in the main nib or allocated lazily. You might even be bundle-loading some components so that the controller classes are not even defined until that point. With the class code linked in, you can always send it **+initialize** regardless of whether you have any object instances. Which object does the work? The possibilities are various:

- Application controller
- Preferences controller
- Individual preference view controllers

The last is arguably the most dynamic and the most object-oriented in style. This design would support dynamically loadable preference objects which manage their own default parameters. But implementing this approach is more complicated. You should remember that other application objects will need to get at these parameters, the reason they are parameters in the first place. Such extreme dynamism, while interesting, is not always the most practical design approach.



Communicating changes to other application objects

Once the user has established a new preference by changing a default value, the application must decide how to update the relevant parts of itself. This might be updating the appearance of one or multiple windows, starting or cancelling a timer, or simply that subsequent actions will retrieve the new value and take effect at that time in the future.

Communicating these changes from your preferences component to other application objects requires additional API, additional dependencies between objects. There are a variety of ways to accomplish the task from a tightly coupled direct message to the concerned party, to a more loosely coupled notification. If possible, you should avoid forcing a user to stop and restart your entire application just to have the new settings take effect.

NSUserDefaults automatically posts a notification every time a value is modified—NSUserDefaulsDidChangeNotification. Application components—such as the application controller—can register for this notification and re-fetch the default values when they change.

Important ideas from this section

» NSUserDefault provides an object-oriented defaults registry.

» It features several domains arranged in a priority hierarchy and with some domains persisting and others volatile, living only for the life of the application in memory

» The basic NSUserDefault API provides for getting a standard shared instance and messaging it to affect values:

register

get

set

» NSDictionary stores key-value pairs. The key is the index for retrieving the corresponding value

» View switching is a convenient user interface technique for multiplexing several different views on to a single panel. It is convenient to implement it with NSBox since it has a content view. Complex preference panels are typically implemented with view switching.

Classes featured in this section

» NSUserDefaults

» NSDictionary

» NSPopUpButton

REVIEW DEFAULTS AND USER PREFERENCES

1. What's the purpose of the NSUserDefaults object?

2. Explain the idea behind the domain concept when designing a user preferences system.

3. Which class should handle the responsibility of registering preference defaults in an application?

4. What happens if a requested user default key is not in any of the domains? Why might this happen?

5. Give some examples where view switching might be used?

EXERCISE 2.1 VIEW SWITCHING PREFERENCES PANEL

In this exercise, you add the preferences panel shown below to the Expense Report application. It lets the user switch between views—different sets of preferences stored and retrieved from the user defaults registry. The user can configure the desired default value and the application will respond accordingly.

Objectives

After completing this exercise, you'll be able to:

- Configure an application to use defaults to control its behavior
- Create user interfaces employing switchable views

Exercise—Stage 1

1. Make a copy of the Expenses project which as found in the folder S**uppExercises/Provided/Supp_2.1_Prefs**. (This is the Expenses project as it would appear after completion of Chapter 14 of the main Programming Yellow Box text.).

2. Although the preferences panel we will create in this exercise is specific to this application, making a preferences panel subproject for the sake of organization is a good idea.

   Create a new Component subproject named **Preferences**. Drag the following files from **SuppExercises/Provided/Supp_2.1_Files** into your Preferences subproject project:

   - Into the Preferences subproject Classes suitcase: **PrefController.m**, **PrefViewController.m** (the corresponding **.h** files are automatically copied for you).
   - Into the Preferences subproject Other Resources suitcase: **Defaults.plist**. This file contains the defaults as a property list. It is convenient to have them in a separate file so that they can be modified without recompiling the application.
   - Into the Preferences subproject Interfaces suitcase: **Preferences.nib**. The preferences main panel is already assembled for you. Your job is to implement the individual preference views and their corresponding controller objects.

3. The PrefController class is a controller for the preferences component. Its job is to manage the nib file, the panel, and to handle the view switching. It also registers the application defaults using **Defaults.plist**. Your first task is to equip your application with the preferences panel via its controller.

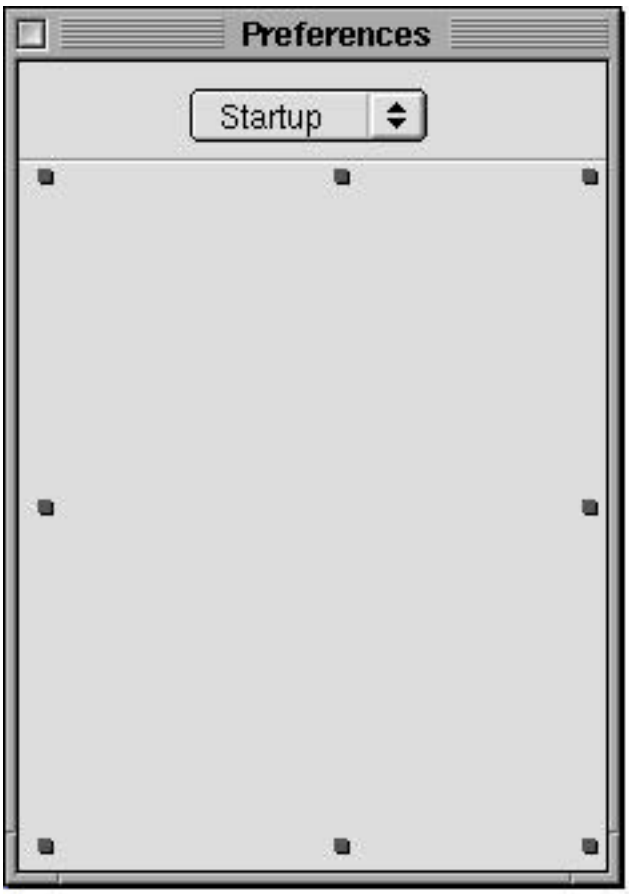
   - Open the main application nib file. Drag the P**refController.h** file into the class browser.
   - Instantiate a PrefController object in the main nib.
   - Add a **prefController** outlet to AppController and an accessor method with the same name—(id)**prefController;**. This is how each individual preference object will find the main controller. Add these to **AppController.h** directly and then re-read it into Interface Builder. Implement the accessor in **AppController.m.**
   - Connect the **prefController** outlet to the Preference instance.
   - Add a Preferences menu item to the Tools or Help menu. Enable it.
   - Connect the Preferences main menu item to the **show:** action of the PrefController instance.
   - Save the main nib.

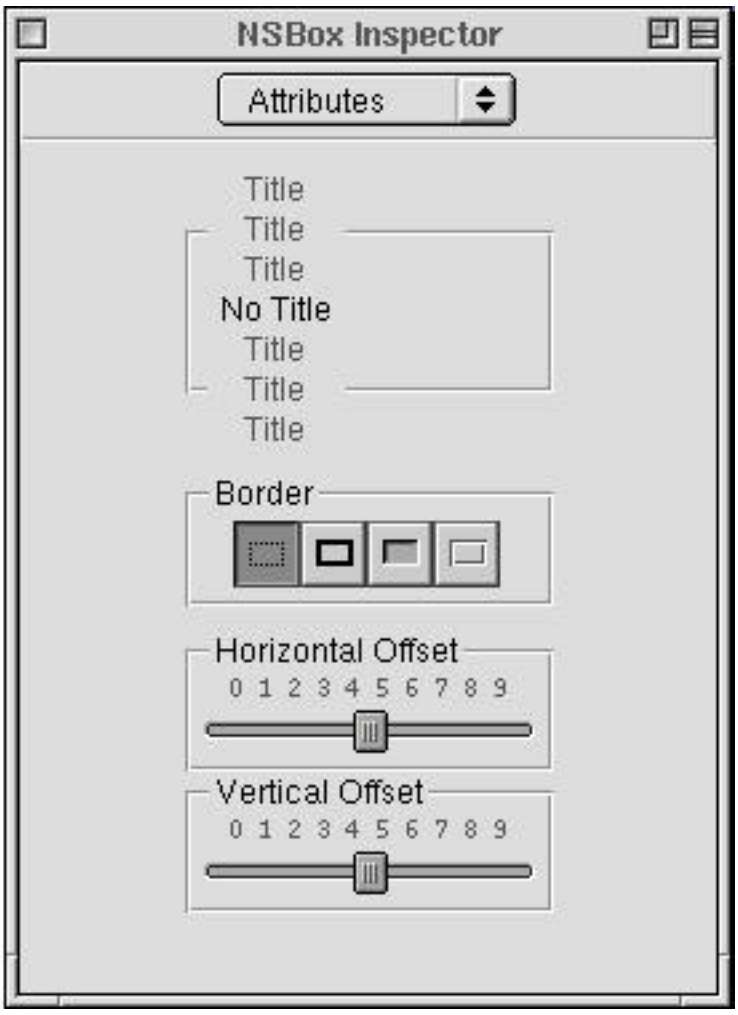
4. Open the **Preferences.nib** file and study what is already in place:

» File's Owner is set to the class PrefController.

» Its **panel** outlet is connected to the panel instance.

» The panel contains a popup button with the three different preference view labels. They will eventually connect to your individual view controllers. Sorting is covered in Exercise 2.2. PrefController has a popup outlet which is connected to the popup button.





» The panel also contains a line—a very thin box. Beneath the line, the panel contains an invisible box. This is where the individual preference views will be switched into place. PrefController has a **box** outlet which is already connected.

5. Read the **PrefController.m** file to understand how this controller works. It is fully implemented for you.

» **initialize**—a class method. Locates the **Defaults.plist** file in the main bundle, instantiates and autoreleased dictionary from the file and registers it with the user defaults object. Since PrefController is instantiated in the main nib, this method is invoked when your application starts up.

» **show:**—loads the nib and makes the panel key and front. The first time around, it must programmatically switch the first view into place. It determines the selected popup button item—configured in Interface Builder—and invokes its target/action. The individual preference view sends a **switchView:** message back to PrefController and it becomes visible.

» **switchView:**—sets the **contentView** of the box on the main panel to the view passed in by one of the individual preference view controllers and marks the box for redisplay.

6. At this point, build the project and check that when you select the Preferences menu option, the Preferences panel appears.

Stage 2

1. The interface now requires separate preference views, one of which will be switched into the box on the main panel at any given time. To save you some time, two views have been preconstructed on a second window inside the nib file. Open the window labelled “Views” and examine the them, each of which is a simple set of controls inside a box. Each will require a separate controller instance, specialized for its particular user interface and the user default value it reflects.

2. Study the PrefViewController class. It provides a simple abstract superclass that implements common functionality. You need to subclass this for each particular type of default—in this case, three different types. In the subclass, you provide a way of retrieving and updating the user default(s) associated with the individual view.

   Note: You do not need to switch the correct view into the preferences panel as that is done in the superclass (Preference) **show:** method, provided everything is connected correctly.

3. Implement the PrefViewController subclass for the "Categories" default. Start with the Preferences nib file in Interface Builder.

   - Drag the PrefViewController header file into Interface Builder's class browser.
   - Subclass PrefViewController and call the subclass **Categories**.
   - Add a **category** outlet.
   - Add a **change:** action.
   - Create the files and add them to the project.
   - Instantiate a Categories object.
   - Connect its **view** outlet to the box and the **category** outlet to the text field that you placed in the view.
   - Connect the text field target/action to the Categories instance using its **change:** action.

4. Connect the Categories menu item in the Preferences pop-up button to the **show:** action in the Categories object. The PrefViewController superclass implements **show:** by messaging the PrefController to switch its view on to the panel. Notice that Categories does not need to be connected to the PrefController instance—it locates it via the application delegate. (See **awakeFromNib** in **PrefViewController.m**)

5. At this point, build the application and verify that the view switching is working correctly. The initial view switched in place corresponds to the first or "selected" item in the popup button. If the panel is initially blank, select the Categories item in the popup button. If everything is in order, the Categories view should appear.

Stage 3

With the user interface working, you can now interact with the user defaults registry. The Categories object must implement **awakeFromNib** to fetch the current user default and set the text field's value to reflect it. The user default key is @"defaultNewCategory". Since the PrefViewController superclass implements **awakeFromNib**, its subclasses must incorporate a call to super:

```
- (void)awakeFromNib
{
   // fetch user defaults value and configure user
interface
   [super awakeFromNib];
}
```

1. Complete the **change:** method in the Categories class. It should set the appropriate user default value to the value of the text field when the user changes the selection. Be sure to use the same user default's key.

   Hint: To save the changed the user defaults to persistent store, use the **synchronize** method—refer to the NSUserDefaults documentation for more information.

2. Look in **Expense.m** in the Documents subproject—you can see that the class already checks the user defaults for the value it should use for new Expense instances—you need not change anything. Once again, make sure you are using the same key for the default value—it must match in three places: **Defaults.plist**, **Expense.m** and **Categories.m.**

3. Try out the application with this new preference view. Create new Expense items and change the default to see if the new items use the new default. Quit and restart the application and verify that the changes persisit. Once you are happy the application is working correctly, move on to the next part of the exercise where you will add a second view.

Stage 4

1. To make the view switching worthwhile, you need to add at least one more view and PrefViewController subclass to coordinate the view and the default. Repeat the procedure, this time for the "Startup" defaults:

   - Return to the Preferences nib and subclass PrefViewController— name the class **Startup**.
   - Add an outlet for a switch and a **change:** action.
   - Create the files and add them to the project.
   - Instantiate a Startup object.
   - In the preferences views window, add a switch button and label it "Show Inspector at Startup".
   - Group the switch and label in a box.
   - Connect the Startup object outlets to the box and the switch, and connect the switch to the **change:** method.
   - Implement **awakeFromNib**—the view displays the current default value when it first appears. The default key is @"showInspector". Remember to call super's **awakeFromNib** as well.
   - Implement **change:**—when the switch is activated, the user default value is updated with the user's preference—the state of the check.

2. Your application should now use the showInspector parameter and show the inspector at startup if its value is YES. Add the relevant code to AppController to **show:** the inspector at startup if this default is set. A logical place is the NSApplication delegate method **applicationDidFinishLaunching:**.

3. Save all the files, build the project and see that it functions as you expect.

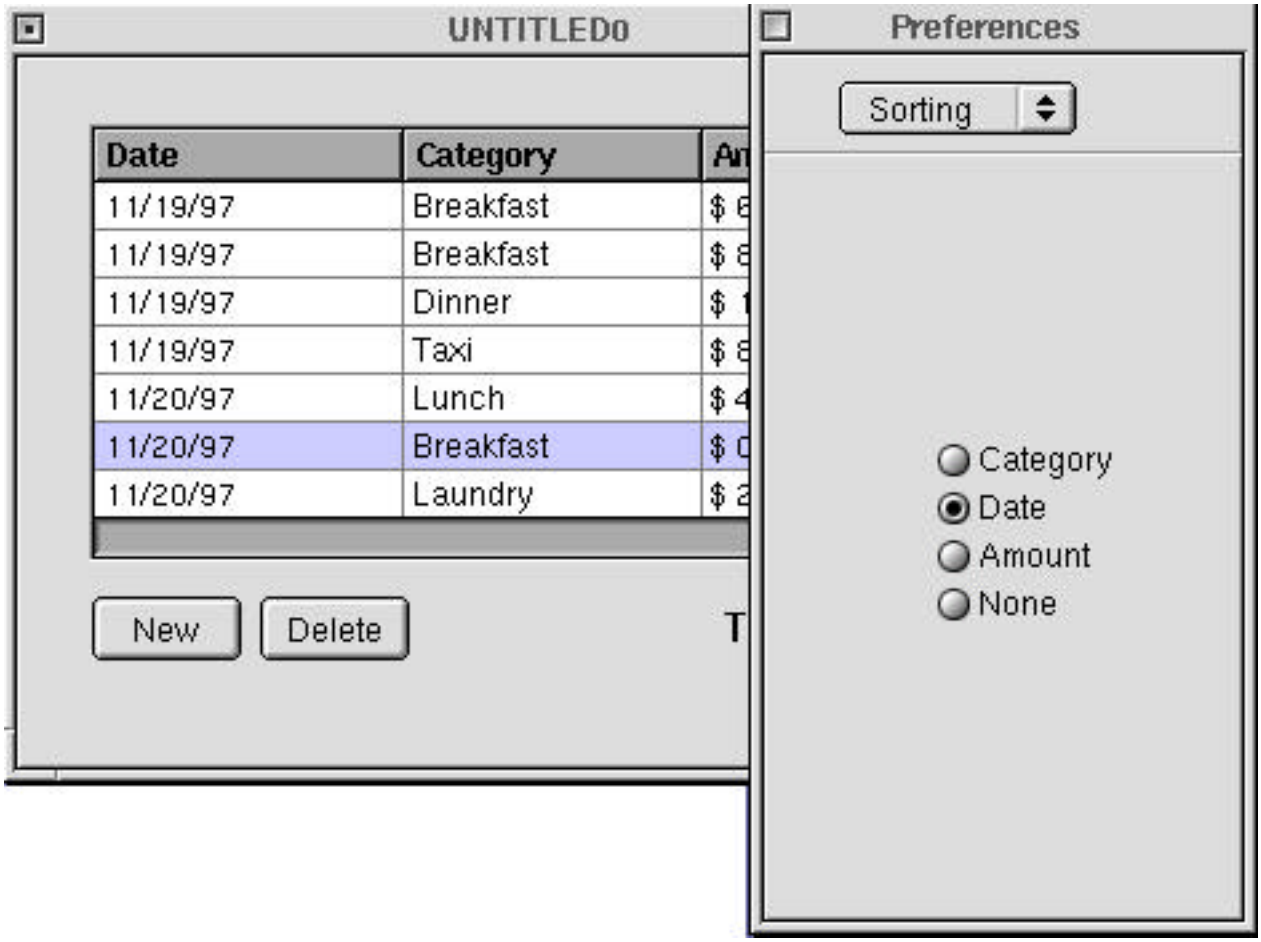
EXERCISE 2.2 ADDING A SORTING PREFERENCE

This exercise demonstrates how to add sorting capability to the table view in the application, and provides a preferences panel view for the user to specify the sort attribute. Each document will sort its table view accordingly.

SuppExercises provides an object which you can insert into your nib file to perform the sorting. You then create a Sort PrefViewController subclass to save the user's sort category in the user defaults registry.

Once the user can configure a sort key, the change needs to be propagated to each document. It is possible to walk through each document in turn and tell it explicitly to re-sort, but it is more elegant to use notifcations. A change to the sort attribute broadcasts a message to any observer objects that wants to know.

The sorting object provided in **SuppExercises** listens for the notification NSUserDefaultsDidChangeNotification. This is a standard notification provided as part of the NSUserDefaults class for this very purpose.



Objective

After completing this exercise, you will be able to use notifications to signal changes to observing objects.

Exercise

1. Drag **SortingDataSource.m** and **SortableTableRow.m** from **SuppExercises/Provided/Supp_2.2_Files** and add them to the Classes suitcase of the documents subproject. These provide the sorting features configurable in the preferences panel. You can study them to see how the SortingDataSource instance handles notification, but this is not necessary for this exercise. **SortingDataSource.h** defines the user default key for sorting preferences—@"sortByColumn".

2. Open the Document nib in Interface Builder. Drag the **SortingDataSource.h** header file into the class browser and instantiate a SortingDataSource object. This object acts as a filter between the ExpenseDataSource and the table view to provide sorting capabilities. To use it, you insert it between the ExpenseDataSource and the table view. To the table view, it looks like a data source; to the data source, it looks like a table view.

   - Disconnect the ExpenseDataSource from the table view and the table view from the ExpenseDataSource.
   - Connect the SortingDataSource's **tableView** outlet to the table view.
   - Connect the SortingDataSource as the dataSource and delegate of the table view.
   - Connect the ExpensesDataSource as the data source of the SortingDataSource.
   - Connect the SortingDataSource to the ExpenseDataSource as though it were the table view. The SortingDataSource will pass on any table view requests to the table view.
   - Save the nib.

3. Return to the Preferences nib and add the Sort PrefViewController:

   - Drag the **Sort.h** and **Sort.m** files from **SuppExercises/Provided/Supp_2.2_Files** into Project Builder, in the Classes suitcase of the Preferences subproject.
   - Drag the **Sort.h** file into Preferences nib as well and instantiate a Sort instance.
   - Add a matrix of radio buttons to the set of preferences views to allow the user to choose among the different sorting attributes. Resize the matrix using the Alt key to add, or delete, buttons to the matrix. These should be labelled the same as the columns of the table view. Add a fourth button, at the bottom, with the label **None**. Set the tags for each radio button to number from 0 to 3. Be sure to press Return after typing in the tag value in the inspector.
   - Group this matrix in a box.
   - Connect the Sort instance to the radio matrix andthe box and connect the radio matrix to Sort's **change:** action.
   - Connect the Sorting popup menu item to the Sort object using the **show:** action.

4. The Sort class already contains the code to interpret the user choices and convert them to the column identifiers the sorter is expecting in its defaults set. Study the methods if you have time.

5. By default, a cell's object value is stored as a string. Without a formatter, you will find that your Expense objects are being filled with strings instead of NSCalendarDates. The SortingDataSource expects Number and CalendarDate objects. To avoid this problem, you will make use of formatters for the columns. Formatters will be explained in a future section, so don't worry if you don't understand how they work.

   - Select the Text palatte in the Interface Builder palette window
   - Drag a Date formatter onto the Date column header in your Document nib tableview
   - Drag a Money formatter on to the Amount column header
   - Save the nib

6. Rebuild the project and run the application. Select one of the sort categories from the Preferences panel and verify that sorting works as expected.

Enhancements

Currently, sorting is initiated by selecting a different sort attribute. However, the SortingDataSource can dynamically re-sort if requested to. In Document nib, connect the SortingDataSource as the target for the table view, using **sortBy:** as the action. The SortingDataSource programmatically changes this to become the table view's double action. Thus the user can double-click on a column title to invoke the dynamic re-sort. This does not change the default. Try it.