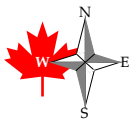


FarWest Software, Inc.

Geoff's Primer On iOS Observers

Table of Contents

Introduction

Observers in iOS provide a powerful way for objects in an iOS app to monitor the state or value of data inside a model or other objects. It acts as a very simple publish-and-subscribe system, but without much in the way of overhead. There is minimal work involved to make an object observer-friendly, and only a modest amount of housekeeping for objects that are observing properties on other objects.

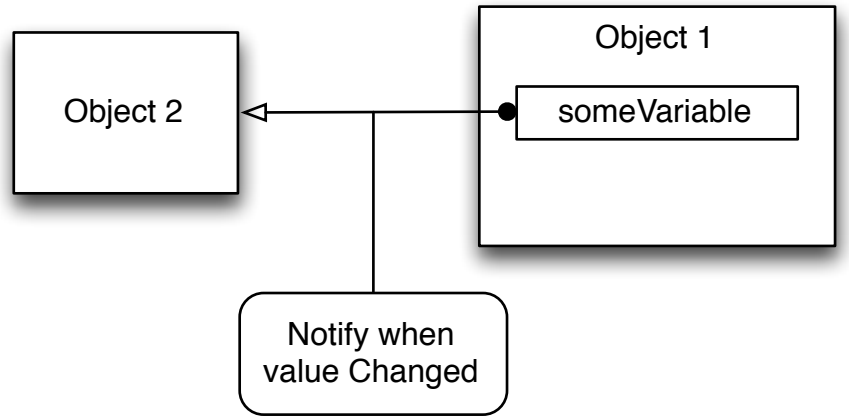
This primer will show you the basics of how observers work in practice, and includes a reference to an iOS project that you can build and run that shows observers in action.

This document is not a substitute for official Apple documentation, and does not attempt to delve under the covers to explain how observers work "behind the scenes". As always, when in doubt refer to Apple's documentation at http://developer.apple.com for the official API's and the latest information on this technology.

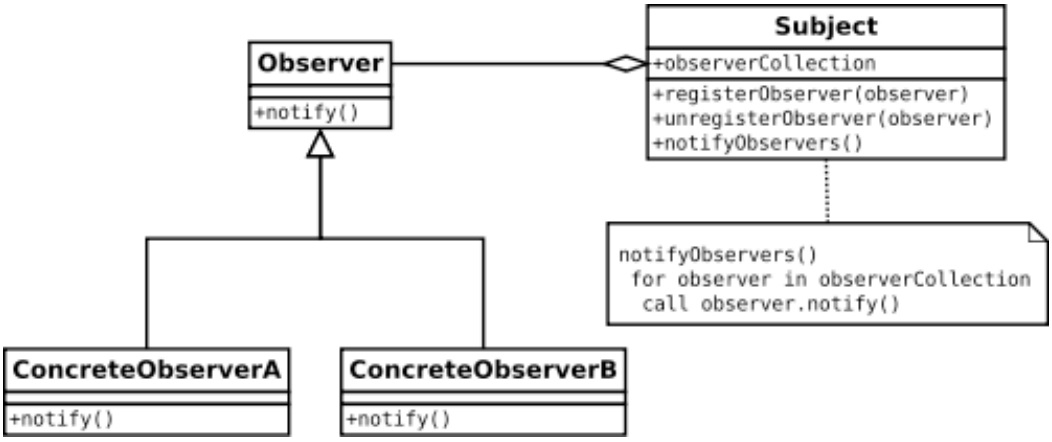
As a side-note, the observers in MacOS work in the exact same way, conforming to the same message signatures and having the same functionality. If you know observers on iOS, then you know observers on MacOS, and vice versa.

What Is An Observer?

An Observer in iOS follows the basic Observer pattern as documented in ***Design Patterns***[1]. The basic pattern is meant to allow one or more objects (observers) to monitor or observe changes in state on another object (the subject). The diagram below illustrates the relationship in its simplest form.



The formal description of the pattern shows that the subject maintains a list of observers, and notifies them when appropriate. The UML for the Observer Pattern is shown below.



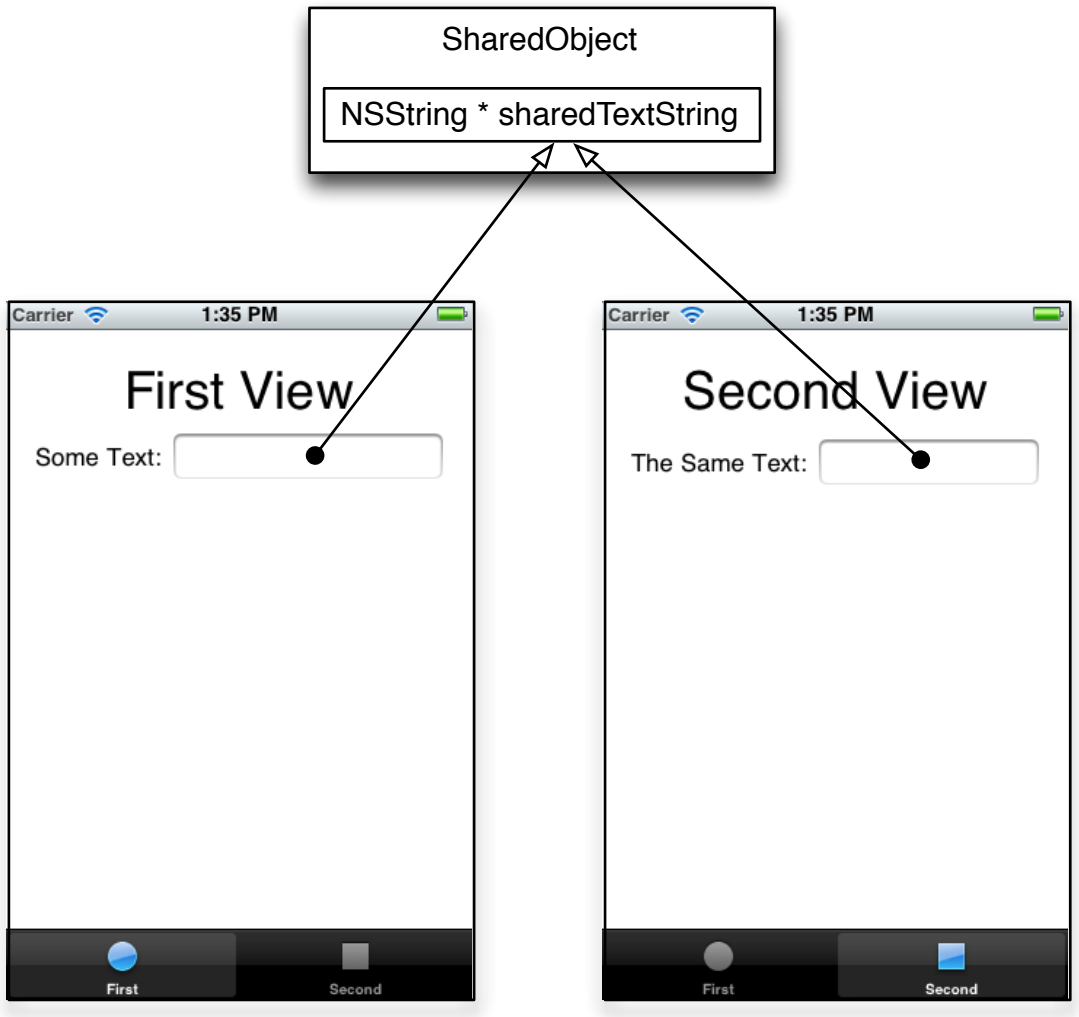
[1] *Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma, Helm, Johnson and Vlissides, Addison-Wesley, 1994, ISBN 978-0201633610.

How the observer pattern actually works is dependent on the implementation. You can build your own Observer pattern in Objective-C, and not use the built-in observer functionality in iOS. However, the ready-made observer implementation in iOS is lightweight and flexible, and using it is straightforward.

Why Use Observers

So what are observers good for? Where observers come in handy is helping to conform to the *model-view-controller* paradigm for iOS apps. It means that different views can all monitor the same, common model, and update their own state, without the monitored object having to know these views are watching, or requiring that the views that alter the model notify all the other views that a change has occurred.

Take an example where two views can display and edit a text value in a shared model.



Without using the observer pattern (either the built-in version in iOS, or one of your own construction), what would typically happen is that the views would have to be made aware of each other. Each time a view changed a property, it would have to notify all the other views that also use the property that the value has changed. In the example above, it is pretty straightforward, since each view only needs to know about one other view. But if a 3rd view is added, then the original 2 views would have to be changed, and all 3 views would have to know about each other. Adding a 4th view and so on makes it harder from a maintenance point of view, and increases the possibility of introducing an error because a view was overlooked.

The Observer pattern allows you to abstract the views from the data, and allows the views to act independently of each other, while still being dependent on the data they are interested in. This, in turn, makes it easier to add new views on the same data, because the other views don't have to change.

Another use for the Observer pattern is simple notification: by monitoring a variable, one view or component can be notified (by a change in the variable's value) that work should be performed (or work in progress should stop). The actual value of the variable may not matter, it's the notification that is the trigger. This can be a simpler way of passing notifications between components, rather than using the Notification Center technology.

The iOS Observer API

The *NSKeyValueObserving* protocol provides for a very rich method for monitoring object properties. The basics, though, only require that you use 2 methods on a subject object, and implement one method in the observing class to be notified of changes. A sample application is described later in this primer showing these methods in context.

- addObserver:forKeyPath:options:context

This method is used to add an observer to an object. The *keyPath* is the name of the property that the observing objects wants to monitor. The options allow you to specify whether you want the pre-change value or the new value after the change, as well as when to be notified. The context allows you to provide your own data, which is passed along when the observing object is notified of a change.

- removeObserver:forKeyPath

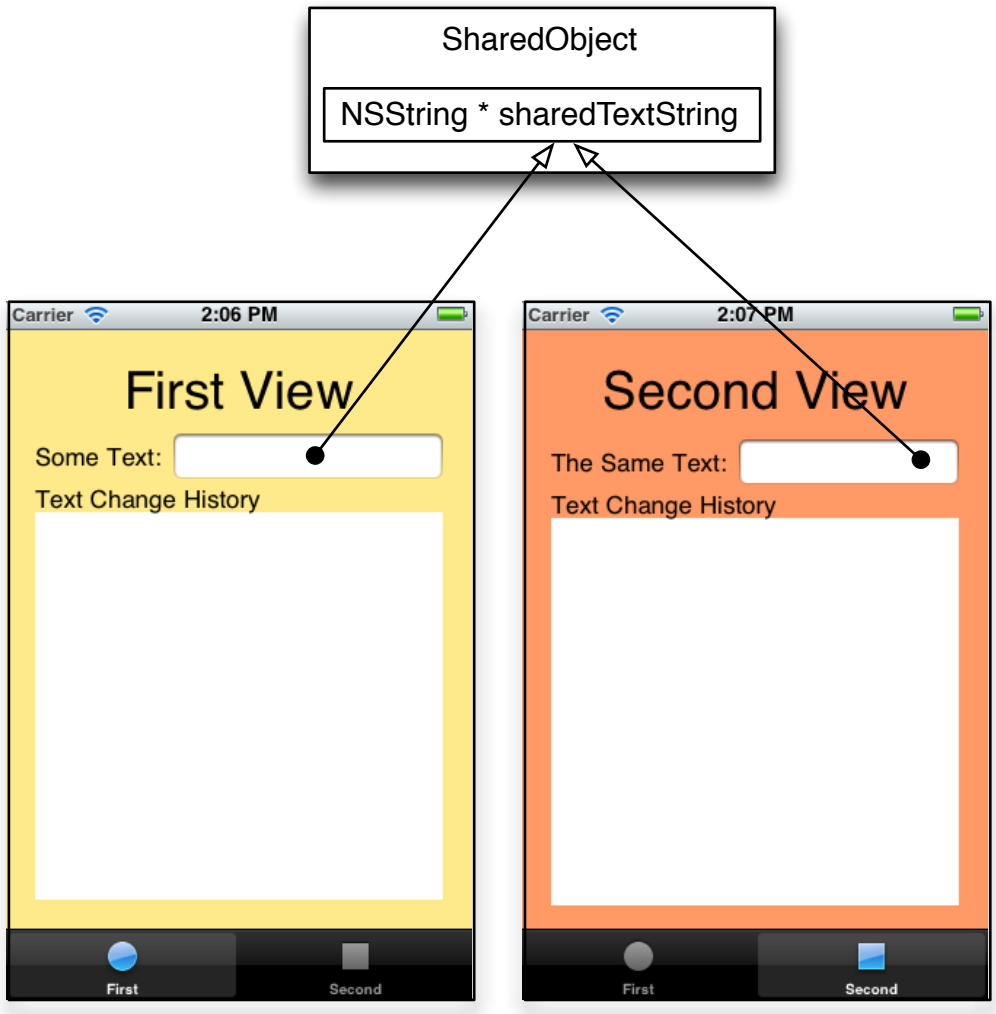
This removes the observer from the observed object's list of listeners. This is a necessary step to ensure that an object doesn't get notified more than once of a change, and to prevent views from being held when they are no longer in use. This second reason prevents memory leaks, since the view instance won't be garbage collected and the memory associated with it freed up if it is still referenced as an observer.

- observeValueForKeyPath:ofObject:change:context

The observer must implement this to be notified of any changes made to properties. The *keyPath* specified when you called *addObserver*, the object that changed and the context you passed during addObserver are all included.

An Example App

The example below shows a simple iOS app that has 2 views sharing a text property on a singleton.



A tab bar is used to manage moving between screens. The iOS storyboard feature was used to build the sample application.

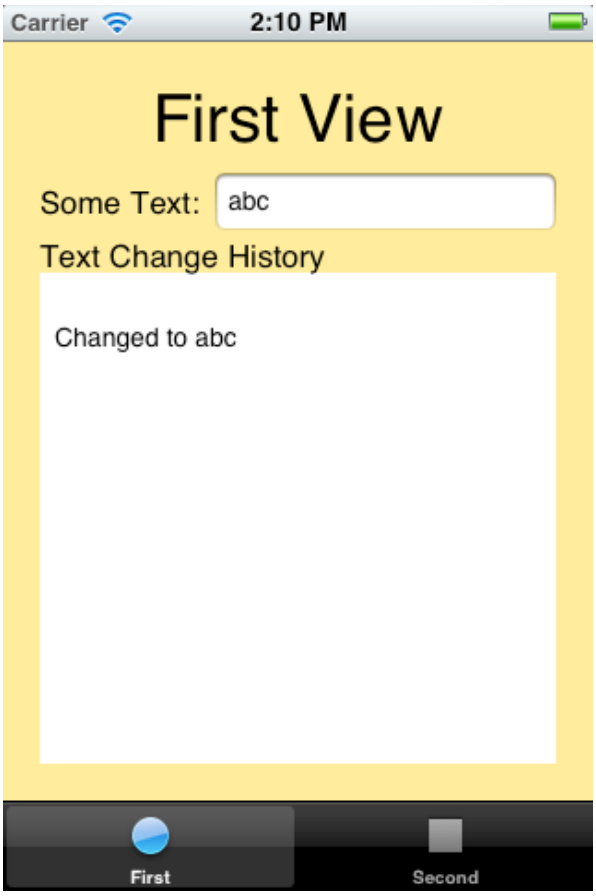
The sample project is available at http://www.farwest.ca/ObserverSample.zip. The project was created using XCode 4.2.1 and the app was built for iOS 5 on iPhone. Ignore the iPad part of the project, as that wasn't implemented or used.

Entering Text

In the sample app, you change the text by performing the following steps in either screen:

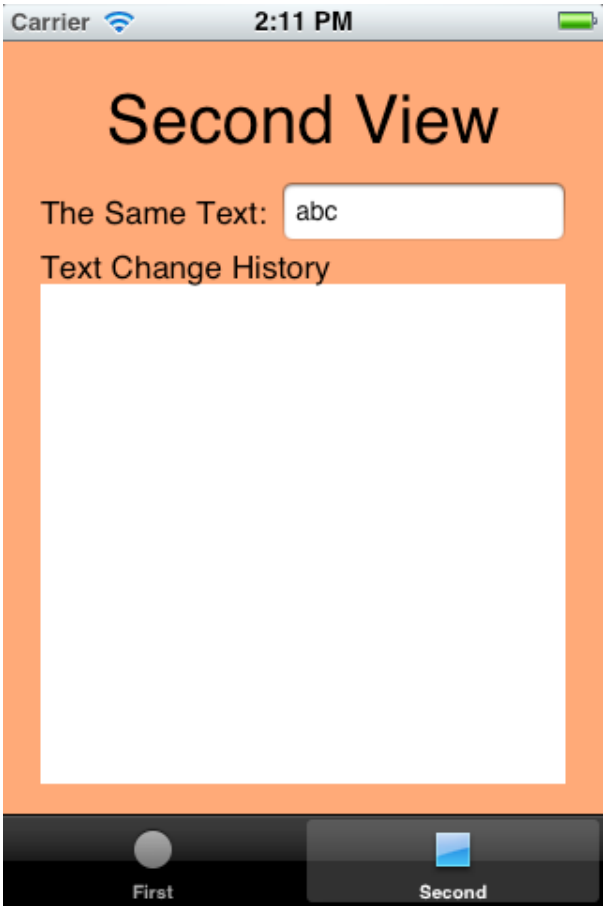
1. Tap on the text field. This brings up the keyboard.
2. Using the keyboard, type the new text value.
3. When you are finishing typing, tap the *return* key. This hides the keyboard, and changes the value of the *sharedTextString* in the shared object instance.

When you run the app for the first time, the First View screen is visible. Changing the text results in an entry in the text view of the form "Change to ..." with the new text as the last part of the entry. For example, entering "abc" into the text field in the first view appears as follows:



But What About The Second View?

When we switch to the second view, the screen appears as follows.



So why doesn't the change history field contain anything? After all, the text field has the current value. Some might expect that the history field should contain something as well.

There is a simple explanation for this: the second view wasn't loaded at startup. Until the user taps on the tab for the second view, the second view doesn't exist yet. Once loaded, it remains loaded and active until the operating system decides it needs to reclaim space, and one way to do that is to reclaim views that aren't visible. Because the view wasn't loaded yet, it had't registered any observers, and was therefore not notified of the changes to the shared text property. The text field has the current string because it obtains that data when it is created and set up in the *viewDidLoad:* method..

A Look At The Code

The First View's Definition

The first view's definition in *FirstViewController.h* is as follows:

```
//
//  FirstViewController.h
//  ObserverSample
//
//  Created by Geoff Kratz on 12-01-03.
//  Copyright (c) 2012 FarWest Software, Inc. All rights reserved.
//

#import <UIKit/UIKit.h>

@interface FirstViewController : UIViewController<UITextFieldDelegate>

@property (weak, nonatomic) IBOutlet UITextField *sharedTextField;
@property (weak, nonatomic) IBOutlet UITextView *textHistoryView;

@end
```

The view contains two outlets (the text field where text is entered, and a text view to display activity related to the shared value). The view controller is also a text field delegate, to allow it to know that the return key was pressed and act accordingly.

Configuring The View

The method where the important pieces are configured is in the *viewDidLoad:* method.

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    sharedTextField.text = [SharedObject getSharedObject].sharedTextString;
    [[SharedObject getSharedObject] addObserver:self forKeyPath:@"sharedTextString"
                                        options:NSKeyValueObservingOptionNew
                                        context:NULL];
}
```

The text field is set up to contain the current value of the shared text property. The instance also adds itself as an observer on the singleton containing the shared property.

Notification From The *return* Key

There are two methods on this object that are also important. The first is the delegate method that is called when the “return” key is tapped.

```
-(BOOL)textFieldShouldReturn:(UITextField *)textField
{
    NSString *txt = sharedTextField.text;
    [SharedObject getSharedObject].sharedTextString = txt;
    [sharedTextField resignFirstResponder];
    return YES;
}
```

This method simply gets the value of the text from the text field, sets that value to the shared property on the *SharedObject*, and then resigns the first responder, which hides the keyboard.

Notification The Shared Property Changed

The next important method is the method called when the shared object’s value has changed.

```
- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context
{
    NSLog(@"First view notified that shared text changed");
    NSString *sharedText = [SharedObject getSharedObject].sharedTextString;
    sharedTextField.text = sharedText;
    textHistoryView.text = [NSString stringWithFormat:@"%@\nChanged to %@",
                            textHistoryView.text, sharedText];
}
```

In this method, we first get the value of the shared property and keep a local reference to it. While it might seem that simply calling ***[SharedObject getSharedObject].sharedTextString*** in the other parts of the code would yield the same results, that isn’t necessarily the case. Keep in mind that some of the activity in iOS is multithreaded, so it is possible that the value of the property will change from one call to another. By keeping a local reference for the lifespan of the method, we are assured that the value we are working with remains unchanged for the time we need it.

Performing Cleanup Activities

An important thing you should do is remove observers when they are no longer necessary. This is typically done when a view is unloaded.

```
- (void)viewDidUnload
{
    [[SharedObject getSharedObject] removeObserver:self
                                         forKeyPath:@"sharedTextString"];
    [self setSharedTextField:nil];
    [self setTextHistoryView:nil];
    [super viewDidUnload];
}
```

In this implementation, we remove the observer before reclaiming the outlet objects.This is to avoid any changes occurring on the outlets if the property we are monitoring changes, and prevents references through a *nil* pointer.

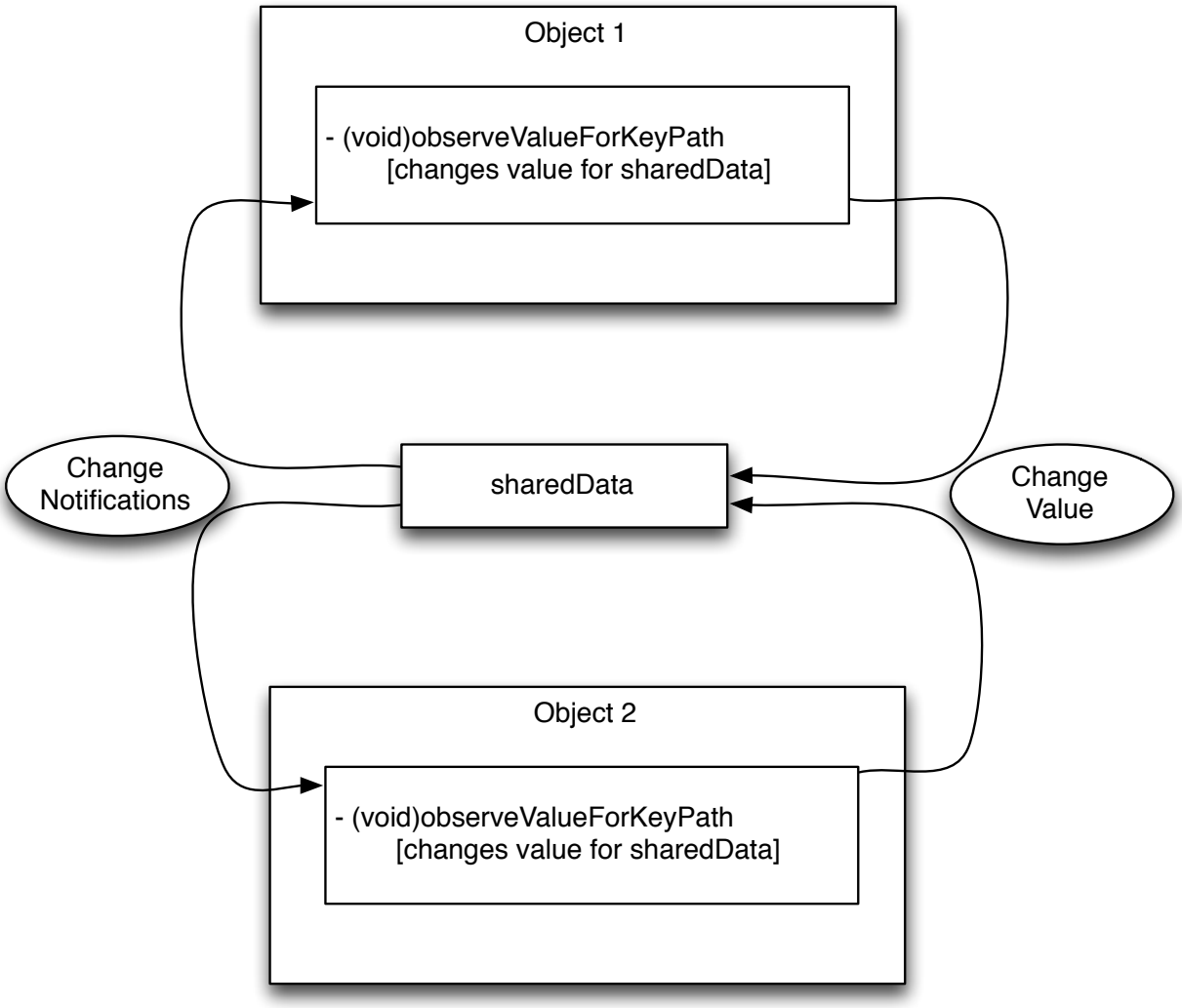
Second View Similar To First

The second view basically implements the same logic and the same methods. You can see this by reviewing the sample project. The view is a *UITextField* delegate configured to respond to the *return* key, registers itself as an observer on the same shared property, and update its history field when changes are detected.

Pitfalls

Beware The Circular Reference Trap

As with anything there are pitfalls and traps that you should be aware of when working with observers. The most common is a form of circular references, where an object changes the value of a shared property while responding to notification that the property was changed.



When *Object 1* changes the value of *sharedData*, *Object 2* is notified. But since *Object 2* changes the value inside it's observe method, *Object 1* is notified again. This can continue indefinitely. As well, the objects not only notify each other, they notify themselves, causing another infinite loop.

How can this happen? Shouldn't I just avoid changing the values of properties I'm observing inside the *observeValueForKeyPath* method? That is certainly good practice. But sometimes one of these can sneak in without you knowing it. Sometimes it can happen because some other object referenced in the *observeValueForKeyPath* method changes the shared property. At other times, it can be because actions in the *observeValueForKeyPath* method trigger messages to other objects, which in turn alter shared data.

Circular References With UITextField and Editing Changed Notification

An example of this is using a *UITextField* and responding to the *Editing Changed* notification. The *Editing Changed* notification is called each time a letter is typed in a text field, as well as if the user backspaces over text, or deletes all the text at once. But, anytime the *text* property on the *UITextField* is changed programmatically, *Editing Changed* is fired, as if the user was doing the typing.

How is this a problem? Consider a typical code structure: each time *Editing Changed* is received, the object updates a shared text field. Consider a *UITextField* that has the action listed below bound to *Editing Changed*.

```
-(IBAction)fieldChanged:(id)sender {
     sharedObject.sharedText = textControl.text;
 }
```

It is pretty straightforward. It sets the value of the *sharedText* property to the current text in the field.

At the same time, the observer notification method will typically set the value on the *UITextField* when it is notified the shared value has changed.

```
-(void) observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context
 {
```

```
            textControl.text = sharedObject.sharedText
        }
```

This certainly seems innocuous enough. But, channeling our best Admiral Ackbar, “It’s a trap!”.

The problem arises as soon as the user types their first character in the text field. First, the *fieldChanged* method is called. As soon as it sets the value of the *sharedText* property, that results in the observers being notified the value has changed. That means that the *observerValueForKeyPath* method on the same object is called. In that method, we set the *UITextField’s* text property. That results in an *Editing Changed* notification, which calls *fieldChanged*. And around we go again, ad infinitum.

One way to avoid this it to determine who the first responder is in the *observeValueForKeyPath* method. If it is the *UITextField* associated with the shared text property in question, then don’t change the text property of the *UITextField*. How do you tell? Basically, you would have to test every control that is associated with a specific shared value, and see if *isFirstResponder* is *TRUE*. Given that most views are fairly simple, this shouldn’t cause an inordinate amount of overhead.

Conclusion

As this document shows, the iOS implementation of observers is quite simple and straightforward. It easily allows objects to monitor the value of properties on other objects. The framework takes care of the overhead of notifying the appropriate objects when changes are made. Your objects don't have to manage this, which makes life simpler in your own apps as well.

Your only job as a developer is to register an observer when appropriate, and remove any observers when an object no longer cares about a particular property. You do need to be careful about infinite loops and circular references where methods that respond to an observer notification end up causing an observer notification. With careful thought and planning, this can generally be avoided.

More questions? I can usually be reached via Twitter through @farwestab.