Key-Value Observing Programming Guide

Data Management: Event Handling

Contents

Listings

Introduction to Key-Value Observing Programming Guide

Key-value observing is a mechanism that allows objects to be notified of changes to specified properties of other objects.

In order to understand key-value observing, you should first read *Key-Value Coding Programming Guide*.

> **Note:** Key-value observing is not supported in Java applications.

Organization of This Document

These concepts are covered in this programming topic:

- "What is Key-Value Observing?" (page 9) provides an overview of the functionality provided by key-value observing.
- "Registering for Key-Value Observing" (page 11) describes how to register and receive observation notifications.
- "Automatic Versus Manual Support" (page 15) describes the difference between automatic and manual key-value observing, and how to implement both.
- "Registering Dependent Keys" (page 19) explains how to specify that the value of a key is dependent on the value of another key.
- "Ensuring KVO Compliance" (page 23) describes what your classes must implement to be key-value observing compliant.
- "Key-Value Observing Implementation Details " (page 25) describes how key-value observing is implemented.

Organization of This Document

What is Key-Value Observing?

Key-value observing provides a mechanism that allows objects to be notified of changes to specific properties of other objects.

The controller layer binding technology relies heavily on key-value observing to be notified of changes in the model and controller layers. For applications that don't rely on the controller layer classes, key-value observing provides simplified methods for implementing inspectors and updating your user interface values.

Unlike NSNotification, there is no central object that provides change notification for all observers. Instead, notifications are sent directly to the observing objects when changes are made. NSObject provides this base implementation of key-value observing, and you should rarely need to override these methods.

You can observe any object properties including simple attributes, to-one relationships, and to-many relationships. Observers of to-many relationships are informed of the type of change made — as well as which objects are involved in the change.

Key-value observing provides an automatic observing capability for all objects. You can further refine notifications by disabling automatic observer notifications and implementing manual notifications.

Registering for Key-Value Observing

In order to receive key-value observing notifications for a property, three things are required:

- The observed class must be key-value observing compliant for the property that you wish to observe, as discussed in "Ensuring KVO Compliance" (page 23).

- You must register the observing object with the observed object, using the method `addObserver:forKeyPath:options:context:`.

- The observing class must implement `observeValueForKeyPath:ofObject:change:context:`.

Registering as an Observer

In order to be notified of changes to a property, an observing object must first register with the object to be observed by sending it an `addObserver:forKeyPath:options:context:` message, passing the observer object and the key path of the property to be observed. The options parameter specifies the information that is provided to the observer when a change notification is sent. Using the option `NSKeyValueObservingOptionOld` specifies that the original object value is provided to the observer as an entry in the change dictionary. Specifying the `NSKeyValueObservingOptionNew` option provides the new value as an entry in the change dictionary. To receive both values, you would bitwise `OR` the option constants.

The example in Listing 1 demonstrates registering an inspector object for the property `openingBalance`.

Listing 1   Registering the inspector as an observer of the openingBalance property

```
- (void)registerAsObserver
{
    // register "inspector" to receive change notifications
    // for the "openingBalance" property of the "account" object
    // and that both the old and new values of "openingBalance"
    // should be provided to the observer
    [account addObserver:inspector
            forKeyPath:@"openingBalance"
               options:(NSKeyValueObservingOptionNew |
                            NSKeyValueObservingOptionOld)
                  context:NULL];
}
```

When you register an object as an observer, you can also provide a context pointer. The context pointer is provided to the observer when `observeValueForKeyPath:ofObject:change:context:` is invoked. The context pointer can be a C pointer or an object reference. The context pointer can be used as a unique identifier to determine the change that is being observed, or to provide some other data to the observer. The context pointer is not retained, and it is the responsibility of the application to ensure that it is not released before the observing object is removed as an observer.

> **Note:** The key-value observing `addObserver:forKeyPath:options:context:` method does not retain the observing object or the observed objects. You need to review your application's requirements and manage retain and release for the observing, and observed, objects.

Receiving Notification of a Change

When the value of an observed property of an object changes, the observer receives an `observeValueForKeyPath:ofObject:change:context:` message. All observers must implement this method.

The observer is provided the object and key path that triggered the observer notification, a dictionary containing details about the change, and the context pointer that was provided when the observer was registered.

The change dictionary entry `NSKeyValueChangeKindKey` provides information about the type of change that occurred. If the value of the observed object has changed, the `NSKeyValueChangeKindKey` entry returns `NSKeyValueChangeSetting`. Depending on the options specified when the observer was registered, the `NSKeyValueChangeOldKey` and `NSKeyValueChangeNewKey` entries in the change dictionary contain the values of the property before, and after, the change.

If the observed property is a to-many relationship, the `NSKeyValueChangeKindKey` entry also indicates whether objects in the relationship were inserted, removed, or replaced by returning `NSKeyValueChangeInsertion`, `NSKeyValueChangeRemoval`, or `NSKeyValueChangeReplacement`, respectively.

The change dictionary entry for `NSKeyValueChangeIndexesKey` is an `NSIndexSet` object specifying the indexes in the relationship that changed. If `NSKeyValueObservingOptionNew` or `NSKeyValueObservingOptionOld` are specified as options when the observer is registered, the `NSKeyValueChangeOldKey` and `NSKeyValueChangeNewKey` entries in the change dictionary are arrays containing the values of the related objects before, and after, the change.

The example in Listing 2 shows the `observeValueForKeyPath:ofObject:change:context:` implementation for an inspector that reflects the old and new values of the property `openingBalance`, as registered in Listing 1 (page 11).

Listing 2   Implementation of observeValueForKeyPath:ofObject:change:context:

```
- (void)observeValueForKeyPath:(NSString *)keyPath
            ofObject:(id)object
                  change:(NSDictionary *)change
                  context:(void *)context
{
    if ([keyPath isEqual:@"openingBalance"]) {
    [openingBalanceInspectorField setObjectValue:
        [change objectForKey:NSKeyValueChangeNewKey]];
    }
    // be sure to call the super implementation
    // if the superclass implements it
    [super observeValueForKeyPath:keyPath
                ofObject:object
                  change:change
```

```
                    context:context];
    }
```

Removing an Object as an Observer

You remove a key-value observer by sending the observed object a `removeObserver:forKeyPath:` message, specifying the observing object and the key path. The example in Listing 3 removes the inspector as an observer of `openingBalance`.

Listing 3   Removing the inspector as an observer of openingBalance

```
- (void)unregisterForChangeNotification
{
    [observedObject removeObserver:inspector
                  forKeyPath:@"openingBalance"];
}
```

If the context specified when the observer was registered is an object, it can be safely released only after removing the observer. After receiving a `removeObserver:forKeyPath:` message, the observing object will no longer receive any `observeValueForKeyPath:ofObject:change:context:` messages for the specified key path and object.

Automatic Versus Manual Support

There are two techniques for making a class's properties observable using key-value observing. Automatic observing is provided by NSObject and is available for all properties of a class that are key-value coding compliant. Manual observing provides additional control over when observations are noted, and requires additional coding.

Automatic Key-Value Observing

NSObject provides a basic implementation of automatic key-value observing. Using automatic observer notifications, it is not necessary to bracket changes to a property with invocations of `willChangeValueForKey:` and `didChangeValueForKey:` when mutating properties via key-value coding and key-value coding compliant methods. Automatic observer notification is controlled by the class method `automaticallyNotifiesObserversForKey:`. The default implementation returns `YES` for all keys.

Automatic key-value observing informs observers of changes made using key-value compliant accessors, as well as the key-value coding methods. The examples shown in Listing 1 result in any observers of the property `name` to be notified of the change.

Listing 1 Methods of invoking key-value observing

```
// calling the accessor method
[self setName:@"Savings"];

// using setValue:forKey:
[self setValue:@"Savings" forKey:@"name"];

// using a key path, where account is a kvc-compliant property
// of "document"
[document setValue:@"Savings" forKeyPath:@"account.name"]
```

Automatic notification is also supported for the collection proxy objects returned by `mutableArrayValueForKey:` and `mutableSetValueForKey:`. This works for to-many relationships that support the indexed accessor methods `insertObject:in<Key>AtIndex:`, `replaceObjectIn<Key>AtIndex:`, and `removeObjectFrom<Key>AtIndex:`.

You can control automatic observer notifications for properties of your subclass by implementing the class method `automaticallyNotifiesObserversForKey:`. Subclasses can test the key passed as the parameter and return `YES` if automatic notification should be enabled, `NO` if it should be disabled.

Manual Observer Notification

Manual key-value observer notification provides more granular control over how and when notifications are sent to observers. This can be useful to help minimize triggering notifications that are unnecessary, or to group a number of changes into a single notification.

A class that implements manual observer notification must override the NSObject implementation of `automaticallyNotifiesObserversForKey:`. It is possible to use both automatic and manual observer notifications in the same class. For properties that perform manual observer notification, the subclass implementation of `automaticallyNotifiesObserversForKey:` should return `NO`. A subclass implementation should invoke `super` for any unrecognized keys. The example in Listing 2 enables manual notification for the `openingBalance` property allowing the superclass to determine the notification for all other keys.

Listing 2   Example implementation of automaticallyNotifiesObserversForKey:

```
+ (BOOL)automaticallyNotifiesObserversForKey:(NSString *)theKey {
BOOL automatic = NO;

    if ([theKey isEqualToString:@"openingBalance"]) {
        automatic=NO;
    } else {
        automatic=[super automaticallyNotifiesObserversForKey:theKey];
    }
    return automatic;
}
```

To implement manual observer notification, you must invoke `willChangeValueForKey:` before changing the value, and `didChangeValueForKey:` after changing the value. The example in Listing 3 implements manual observer notifications for the `openingBalance` property.

Listing 3   Example accessor method implementing manual observer notification

```
- (void)setOpeningBalance:(double)theBalance {
    [self willChangeValueForKey:@"openingBalance"];
    openingBalance=theBalance;
    [self didChangeValueForKey:@"openingBalance"];
}
```

You can minimize sending unnecessary notifications by first checking if the value has changed. The example in Listing 4 tests the value of `openingBalance` and only provides the notification if it has changed.

Listing 4   Testing the value for change before providing notification

```
- (void)setOpeningBalance:(double)theBalance {
    if (theBalance != openingBalance) {
        [self willChangeValueForKey:@"openingBalance"];
        openingBalance=theBalance;
        [self didChangeValueForKey:@"openingBalance"];
    }
}
```

If a single operation causes multiple keys to change you must nest the change notifications as shown in Listing 5.

Listing 5   Nesting change notifications for multiple keys

```
- (void)setOpeningBalance:(double)theBalance {
    [self willChangeValueForKey:@"openingBalance"];
    [self willChangeValueForKey:@"itemChanged"];
    openingBalance=theBalance;
    itemChanged=itemChanged+1;
    [self didChangeValueForKey:@"itemChanged"];
    [self didChangeValueForKey:@"openingBalance"];
}
```

In the case of a to-many relationship, you must specify not only the key that changed, but also the type of change and the indexes of the objects involved. The type of change is an `NSKeyValueChange` that specifies `NSKeyValueChangeInsertion`, `NSKeyValueChangeRemoval`, or `NSKeyValueChangeReplacement`. The indexes of the affected objects are passed as an NSIndexSet.

The code fragment in Listing 6 demonstrates how to wrap a deletion of objects in the to-many relationship `transactions`.

Listing 6   Implementation of manual observer notification in a to-many relationship

```
- (void)removeTransactionsAtIndexes:(NSIndexSet *)indexes {
    [self willChange:NSKeyValueChangeRemoval
     valuesAtIndexes:indexes forKey:@"transactions"];

    // remove the transaction objects at the specified indexes here

    [self didChange:NSKeyValueChangeRemoval
    valuesAtIndexes:indexes forKey:@"transactions"];
}
```

Note: Care should be taken that you do not release the values that will change, before sending a `willChange` message.

Registering Dependent Keys

There are many situations in which the value of one property depends on that of one or more other attributes in another entity. If the value of one attribute changes, then the value of the derived property should also be flagged for change. How you ensure that key-value observing notifications are posted for these dependent properties depends on which version of Mac OS X you're using and the cardinality of the relationship.

Mac OS X v10.5 and later for a to-one relationship

If you are targeting Mac OS X v10.5 and later, and there is a to-one relationship to the related entity, then to trigger notifications automatically you should either override `keyPathsForValuesAffectingValueForKey:` or implement a suitable method that follows the pattern it defines for registering dependent keys.

For example, the full name of a person is dependent on both the first and last names. A method that returns the full name could be written as follows:

```
- (NSString *)fullName {
    return [NSString stringWithFormat:@"%@ %@",firstName, lastName];
}
```

An application observing the `fullName` property must be notified when either the `firstName` or `lastName` properties change, as they affect the value of the property.

One solution is to override `keyPathsForValuesAffectingValueForKey:` specifying that the `fullName` property of a person is dependent on the `lastName` and `firstName` properties. Listing 1 (page 19) shows an example implementation of such a dependency:

Listing 1   Example implementation of `keyPathsForValuesAffectingValueForKey:`

```
+ (NSSet *)keyPathsForValuesAffectingValueForKey:(NSString *)key
{
    NSSet *keyPaths = [super keyPathsForValuesAffectingValueForKey:key];

    if ([key isEqualToString:@"fullName"])
    {
        NSSet *affectingKeys = [NSSet setWithObjects:@"lastName",
@"firstName",nil];
        keyPaths = [keyPaths setByAddingObjectsFromSet:affectingKeys];
    }
    return keyPaths;
}
```

Your override should typically invoke super and return a set that includes any members in the set that result from doing that (so as not to interfere with overrides of this method in superclasses).

You can also achieve the same result by implementing a class method that follows the naming convention `keyPathsForValuesAffecting<Key>`, where `<Key>` is the name of the attribute (first letter capitalized) that is dependent on the values. Using this pattern the code in Listing 1 (page 19) could be rewritten as a class method named `keyPathsForValuesAffectingFullName` as shown in Listing 2 (page 20).

Listing 2   Example implementation of the `keyPathsForValuesAffecting<Key>` naming convention

```
+ (NSSet *)keyPathsForValuesAffectingFullName
{
    return [NSSet setWithObjects:@"lastName", @"firstName", nil];
}
```

You can't override the `keyPathsForValuesAffectingValueForKey:` method when you add a computed property to an existing class using a category, because you're not supposed to override methods in categories. In that case, implement a matching `keyPathsForValuesAffecting<Key>` class method to take advantage of this mechanism.

> **Note:** You cannot set up dependencies on to-many relationships by implementing `keyPathsForValuesAffectingValueForKey:`. Instead, you must observe the appropriate attribute of each of the objects in the to-many collection and respond to changes in their values by updating the dependent key yourself. The following section shows a strategy for dealing with this situation.

Mac OS X v10.4 and to-many relationships on Mac OS X v10.5

If you are targeting Mac OS X v10.4, `setKeys:triggerChangeNotificationsForDependentKey:` does not allow key-paths, so you cannot follow the pattern described above.

If you are targeting Mac OS X v10.5, `keyPathsForValuesAffectingValueForKey:` does not allow key-paths that include a to-many relationship. For example, suppose you have an Department entity with a to-many relationship (`employees`) to a Employee, and Employee has a salary attribute. You might want the Department entity have a `totalSalary` attribute that is dependent upon the salaries of all the Employees in the relationship. You can not do this with, for example, `keyPathsForValuesAffectingTotalSalary` and returning `employees.salary` as a key.

There are two possible solutions in both situations:

1. You can use key-value observing to register the parent (in this example, Department) as an observer of the relevant attribute of all the children (Employees in this example). You must add and remove the parent as an observer as child objects are added to and removed from the relationship (see "Registering for Key-Value Observing" (page 11)). In the `observeValueForKeyPath:ofObject:change:context:` method you update the dependent value in response to changes, as illustrated in the following code fragment:

    ```
    - (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object
    change:(NSDictionary *)change context:(void *)context
    {
        if (context == totalSalaryContext) {
            [self updateTotalSalary];
        }
        else
        // deal with other observations and/or invoke super...
    ```

```
    }
- (void)updateTotalSalary
{
    [self setTotalSalary:[self valueForKeyPath:@"employees.@sum.salary"]];
}
- (void)setTotalSalary:(NSNumber *)newTotalSalary
{
    if (totalSalary != newTotalSalary) {
        [self willChangeValueForKey:@"totalSalary"];
        [totalSalary release];
        totalSalary = [newTotalSalary retain];
        [self didChangeValueForKey:@"totalSalary"];
    }
}
- (NSNumber *)totalSalary
{
    return totalSalary;
}
```

2. If you're using Core Data, you can register the parent with the application's notification center as an observer of its managed object context. The parent should respond to relevant change notifications posted by the children in a manner similar to that for key-value observing.

Ensuring KVO Compliance

In order to be considered KVO-compliant for a specific property, a class must ensure the following;

- The class must be key-value coding compliant for the property as specified in Ensuring KVC Compliance.

- The class must allow automatic observer notifications for the property, or implement manual key-value observing for the property.

Key-Value Observing Implementation Details

Automatic key-value observing is implemented using a technique called isa-swizzling.

The isa pointer, as the name suggests, points to the object's class which maintains a dispatch table. This dispatch table essentially contains pointers to the methods the class implements, among other data.

When an observer is registered for an attribute of an object the isa pointer of the observed object is modified, pointing to an intermediate class rather than at the true class. As a result the value of the isa pointer does not necessarily reflect the actual class of the instance.

Instead of relying on the isa pointer your application should use the `class` method to determine the class of an object instance.

Document Revision History

This table describes the changes to *Key-Value Observing Programming Guide*.

Date	Notes
2009-08-14	Added links to some key Cocoa definitions.
2009-05-09	Corrected minor typo.
2009-05-06	Clarified Core Data requirement in Registering Dependent Keys.
2009-03-04	Updated Registering Dependent Keys chapter.
2006-06-28	Updated code examples.
2005-07-07	Clarified that you should not release objects before calling willChangeValueForKey: methods. Noted that Java is not supported.
2004-08-31	Corrected minor typos.
	Clarified the need to nest manual key-value change notifications in "Automatic Versus Manual Support" (page 15).
2004-03-20	Modified source example in "Registering Dependent Keys" (page 19).
2004-02-22	Corrected source example in "Registering for Key-Value Observing" (page 11). Added article "Key-Value Observing Implementation Details " (page 25).
2003-11-11	Corrected reference to instance variables in "Automatic Versus Manual Support" (page 15)
2003-10-15	Initial publication of *Key-Value Observing*.