Key-Value Observing Programming Guide

Developer

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.

Important In order to understand key-value observing, you must first understand key-value coding.

At a Glance

Key-value observing provides a mechanism that allows objects to be notified of changes to specific properties of other objects. It is particularly useful for communication between model and controller layers in an application. (In OS X, the controller layer binding technology relies heavily on key-value observing.) A controller object typically observes properties of model objects, and a view object observes properties of model objects through a controller. In addition, however, a model object may observe other model objects (usually to determine when a dependent value changes) or even itself (again to determine when a dependent value changes).

You can observe 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.

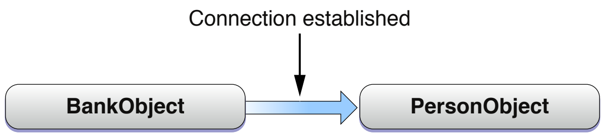
There are three steps to setting up an observer of a property. Understanding these three steps provides a clear illustration of how KVO works.

1. First, see whether you have a scenario where key-value observing could be beneficial, for example, an object that needs to be notified when any changes are made to a specific property in another object.

   BankObject
   @property int accountBalance
   PersonObject

   For example, a `PersonObject` will want to be aware of any changes made to their `accountBalance` in the `BankObject`.

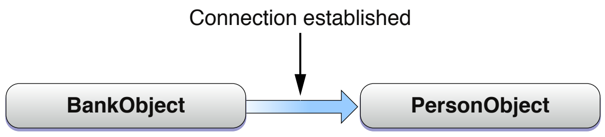
2. The `PersonObject` must register as an observer of the `BankObject`’s `accountBalance` property by sending an `addObserver:forKeyPath:options:context:` message.



```
[bankInstance addObserver:personInstance
    forKeyPath:@"accountBalance"
       options:NSKeyValueObservingOptionNew
           context:NULL];
```

> **Note** The `addObserver:forKeyPath:options:context:` method establishes a connection between the instances of the objects that you specify. A connection is not established between the two classes, but rather between the two specified instances of the objects.

3. In order to respond to change notifications, the observer must implement the `observeValueForKeyPath:ofObject:change:context:` method. This method implementation defines how the observer responds to change notifications. It is in this method that you can customize your response to a change in one of the observed properties.

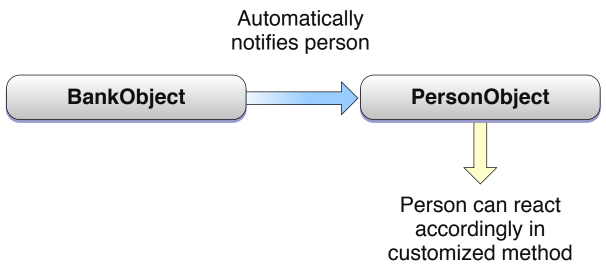


```
-(void) observeValueForKeyPath:(NSString *)keyPath
    ofObject:(id)object
        change:(NSDictionary *)change
            context:(void *)context
{

//custom implementation
//be sure to call the superclass’ implementation
//if the superclass implements it

}
```

“Registering for Key-Value Observing” (page 7) describes how to register and receive observation notifications.

4. The `observeValueForKeyPath:ofObject:change:context:` method is automatically invoked when the value of an observed property is changed in a KVO-compliant manner, or if a key upon which it depends is changed.

```
[bankInstance setAccountBalance:50];
```


Automatically notifies person
BankObject
PersonObject
Person can react accordingly in customized method

"Registering Dependent Keys" (page 15) explains how to specify that the value of a key is dependent on the value of another key.

KVO's primary benefit is that you don't have to implement your own scheme to send notifications every time a property changes. Its well-defined infrastructure has framework-level support that makes it easy to adopt—typically you do not have to add any code to your project. In addition, the infrastructure is already full-featured, which makes it easy to support multiple observers for a single property, as well as dependent values.

"KVO Compliance" (page 11) describes the difference between automatic and manual key-value observing, and how to implement both.

Unlike notifications that use `NSNotificationCenter`, 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.

"Key-Value Observing Implementation Details " (page 19) describes how key-value observing is implemented.

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.
- 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:`.

Important Not all classes are KVO-compliant for all properties. You can ensure your own classes are KVO-compliant by following the steps described in "KVO Compliance" (page 11). Typically properties in Apple-supplied frameworks are only KVO-compliant if they are documented as such.

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 specify that both the old and new values of
"openingBalance"
     should be provided in the observe… method.
     */
    [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.

> **Note** The key-value observing `addObserver:forKeyPath:options:context:` method does not maintain strong references to the observing object, the observed objects, or the context. You should ensure that you maintain strong references to the observing, and observed, objects, and the context as necessary.

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 property is an object, the value is provided directly. If the property is a scalar or a C structure, the value is wrapped in an `NSValue` object (as with key-value coding).

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 7).

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 superclass's implementation *if it implements it*.
     NSObject does not implement the method.
     */
    [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 is an object, you must keep a strong reference to it until 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.

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".

  KVO supports the same data types as KVC.
- The class emits KVO change notifications for the property.
- Dependent keys are registered appropriately (see "Registering Dependent Keys" (page 15)).

There are two techniques for ensuring the change notifications are emitted. Automatic support is provided by `NSObject` and is by default available for all properties of a class that are key-value coding compliant. Typically, if you follow standard Cocoa coding and naming conventions, you can use automatic change notifications—you don't have to write any additional code.

Manual change notification provides additional control over when notifications are emitted, and requires additional coding. You can control automatic notifications for properties of your subclass by implementing the class method `automaticallyNotifiesObserversForKey:`.

Automatic Change Notification

`NSObject` provides a basic implementation of automatic key-value change notification. Automatic key-value change notification informs observers of changes made using key-value compliant accessors, as well as the key-value coding methods. Automatic notification is also supported by the collection proxy objects returned by, for example, `mutableArrayValueForKey:`.

The examples shown in Listing 1 result in any observers of the property `name` to be notified of the change.

Listing 1 Examples of method calls that cause KVO change notifications to be emitted

```
// Call the accessor method.
[account setName:@"Savings"];

// Use setValue:forKey:.
[account setValue:@"Savings" forKey:@"name"];
```

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

// Use mutableArrayValueForKey: to retrieve a relationship proxy object.
Transaction *newTransaction = <#Create a new transaction for the account#>;
NSMutableArray *transactions = [account mutableArrayValueForKey:@"transactions"];
[transactions addObject:newTransaction];
```

Manual Change Notification

Manual change 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 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 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 invoke `willChangeValueForKey:` before changing the value, and `didChangeValueForKey:` after changing the value. The example in Listing 3 implements manual notifications for the `openingBalance` property.

Listing 3 Example accessor method implementing manual 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 an ordered 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` object.

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.

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

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 object. 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 the cardinality of the relationship.

To-one Relationships

To trigger notifications automatically for a to-on relationship 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 15) 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"]) {
        NSArray *affectingKeys = @[@"lastName", @"firstName"];
        keyPaths = [keyPaths setByAddingObjectsFromArray: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 15) could be rewritten as a class method named `keyPathsForValuesAffectingFullName` as shown in Listing 2 (page 16).

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.

To-many Relationships

The `keyPathsForValuesAffectingValueForKey:` method does not support key-paths that include a to-many relationship. For example, suppose you have a Department object with a to-many relationship (`employees`) to a Employee, and Employee has a salary attribute. You might want the Department object 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 7)). 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 = newTotalSalary;

        [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.

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.

You should never rely on the `isa` pointer to determine class membership. Instead, you 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
2012-07-17	Updated to use new Objective-C features. ARCification
2011-03-08	Clarified terminology in "Registering Dependent Keys."
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.
2004-03-20	Modified source example in "Registering Dependent Keys" (page 15).
2004-02-22	Corrected source example in "Registering for Key-Value Observing" (page 7). Added article "Key-Value Observing Implementation Details " (page 19).
2003-10-15	Initial publication of *Key-Value Observing*.