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).

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"];
}
```