Issue 7: Foundation · December 2013

Browse Issue ↓

Key-Value Coding and Observing

By Daniel Eggert

Key-value coding and key-value obse
mechanisms that allow us to simplify
dynamic and introspective properties
article, we'll take a look at some exar

In this article

Observing Changes to

In Cocoa, the Model-View-Controller
is to keep the view and the model syr

this: when the model object changes
reflect this change, and when the use
has to be updated accordingly.

Key-Value Observing helps us update
model objects. The controller can obs
values that the views depend on.

Let's look at a sample: Our model cla
color space where the components a
and blue). We want sliders to change
shows the color.

Our model class will have three prope

OBJECTIVE-C

```
@property (nonatomic) double l
@property (nonatomic) double aComponent;
@property (nonatomic) double bComponent;
```


Dependent Properties

We need to create a `UIColor` from this that we can use to display the color. We'll add three additional properties for the red, green, and blue components and another property for the `UIColor`:

OBJECTIVE-C SELECT ALL

```
@property (nonatomic, readonly) double redComponent;
@property (nonatomic, readonly) double greenComponent;
@property (nonatomic, readonly) double blueComponent;

@property (nonatomic, strong, readonly) UIColor *color;
```

With this, we have all we need for our class interface:

OBJECTIVE-C SELECT ALL

```
@interface LabColor : NSObject

@property (nonatomic) double lComponent;
@property (nonatomic) double aComponent;
@property (nonatomic) double bComponent;

@property (nonatomic, readonly) double redComponent;
@property (nonatomic, readonly) double greenComponent;
@property (nonatomic, readonly) double blueComponent;

@property (nonatomic, strong, readonly) UIColor *color;

@end
```

The math for calculating the red, green, and blue components is outlined on Wikipedia. It looks something like this:

OBJECTIVE-C SELECT ALL

```
- (double)greenComponent;
{
    return D65TristimulusValues[1] * inverseF(1./116. * (self
}

[...]

- (UIColor *)color
{
    return [UIColor colorWithRed:self.redComponent * 0.01 gre
}
```

Nothing too exciting here. What's interesting to us is that this `greenComponent` property depends on the `lComponent` and `aComponent` properties. This is important to key-value observing; whenever we set the `lComponent` property we want anyone interested in either of the red-green-blue components or the `color` property to be notified.

The mechanism that the Foundation framework provides for expressing dependencies is:

OBJECTIVE-C SELECT ALL

```
+ (NSSet *)keyPathsForValuesAffectingValueForKey:(NSString *)
```

and more specifically:

OBJECTIVE-C SELECT ALL

```
+ (NSSet *)keyPathsForValuesAffecting<Key>
```

In our concrete case, that'll look like so:

OBJECTIVE-C SELECT ALL

```
+ (NSSet *)keyPathsForValuesAffectingRedComponent
{
    return [NSSet setWithObject:@"lComponent"];
}

+ (NSSet *)keyPathsForValuesAffectingGreenComponent
{
    return [NSSet setWithObjects:@"lComponent", @"aComponent'
}

+ (NSSet *)keyPathsForValuesAffectingBlueComponent
{
    return [NSSet setWithObjects:@"lComponent", @"bComponent'
}

+ (NSSet *)keyPathsForValuesAffectingColor
{
    return [NSSet setWithObjects:@"redComponent", @"greenComp
}
```

We have now fully expressed the dependencies. Note that we're able to do chaining of these dependencies. For example, this would allow us to

safely subclass and override the `redComponent` method and the dependency would continue to work.

Observing Changes

Let's turn toward the view controller. The `NSViewController` subclass owns the model object `LabColor` as a property:

OBJECTIVE-C SELECT ALL

```
@interface ViewController ()

@property (nonatomic, strong) LabColor *labColor;

@end
```

We want to register the view controller to receive key-value observation notifications. The method on `NSObject` to do this is:

OBJECTIVE-C SELECT ALL

```
- (void)addObserver:(NSObject *)anObserver
         forKeyPath:(NSString *)keyPath
            options:(NSKeyValueObservingOptions)options
            context:(void *)context
```

This will cause:

OBJECTIVE-C SELECT ALL

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

to get called on `anObserver` whenever the value of `keyPath` changes. This API can seem a bit daunting. To make things worse, we have to remember to call:

OBJECTIVE-C SELECT ALL

```
- (void)removeObserver:(NSObject *)anObserver
            forKeyPath:(NSString *)keyPath
```

to remove the observer, otherwise our app will crash in strange ways.

For most intents and purposes, *key-value observing* can be done in a much simpler and more elegant way by using a helper class. We'll add a so-called *observation token* property to our view controller:

OBJECTIVE-C SELECT ALL

```
@property (nonatomic, strong) id colorObserveToken;
```

and when the `labColor` gets set on the view controller, we'll simply observe changes to its `color` by overriding the setter for `labColor`, like so:

OBJECTIVE-C SELECT ALL

```
- (void)setLabColor:(LabColor *)labColor
{
    _labColor = labColor;
    self.colorObserveToken = [KeyValueObserver observeObject:
                                                      keyPath:
                                                       target:
                                                     selector:
                                                      options:
}

- (void)colorDidChange:(NSDictionary *)change;
{
    self.colorView.backgroundColor = self.labColor.color;
}
```

The KeyValueObserver helper class simply wraps the calls to `-addObserver:forKeyPath:options:context:`, `-observeValueForKeyPath:ofObject:change:context:` and `-removeObserverForKeyPath:` and keeps the view controller code free of clutter.

Tying it Together

The view controller finally needs to react to changes of the *L*, *a*, and *b* sliders:

OBJECTIVE-C SELECT ALL

```
- (IBAction)updateLComponent:(UISlider *)sender;
{
    self.labColor.lComponent = sender.value;
}

- (IBAction)updateAComponent:(UISlider *)sender;
{
    self.labColor.aComponent = sender.value;
}

- (IBAction)updateBComponent:(UISlider *)sender;
{
    self.labColor.bComponent = sender.value;
}
```

The entire code is available as a sample project on our GitHub repository.

Manual vs. Automatic Notification

What we did above may seem a bit like magic, but what happens is that calling `-setLComponent:` etc. on a `LabColor` instance will automatically cause:

OBJECTIVE-C SELECT ALL

```
- (void)willChangeValueForKey:(NSString *)key
```

and:

OBJECTIVE-C SELECT ALL

```
- (void)didChangeValueForKey:(NSString *)key
```

to get called prior to or after running the code inside the `-setLComponent:` method. This happens both if we implement `-setLComponent:` and if we (as in our case) choose to auto-synthesize the accessors for `lComponent`.

There are cases when we want or need to override `-setLComponent:` and control whether change notifications are sent out, like so:

OBJECTIVE-C SELECT ALL

```
+ (BOOL)automaticallyNotifiesObserversForLComponent;
{
    return NO;
}

- (void)setLComponent:(double)lComponent;
{
    if (_lComponent == lComponent) {
        return;
    }
    [self willChangeValueForKey:@"lComponent"];
    _lComponent = lComponent;
    [self didChangeValueForKey:@"lComponent"];
}
```

We disable automatic invocation of `-willChangeValueForKey:` and `-didChangeValueForKey:`, and then call it ourselves. We should only call `-willChangeValueForKey:` and `-didChangeValueForKey:` inside the setter if we've disabled automatic invocation. And in most cases, this optimization doesn't buy us much.

If we modify the instance variables (e.g. `_lComponent`) outside the accessor, we need to be careful to similarly wrap those changes in `-willChangeValueForKey:` and `-didChangeValueForKey:`. But in most cases, the code stays simpler if we make sure to always use the accessors.

Key-Value Observing and the Context

There may be reasons why we don't want to use the `KeyValueObserver` helper class. There's a slight overhead of creating another object. If we're observing a lot of keys, that might be noticeable, however unlikely that is.

If we're implementing a class that registers itself as an observer with:

OBJECTIVE-C SELECT ALL

```
- (void)addObserver:(NSObject *)anObserver
         forKeyPath:(NSString *)keyPath
            options:(NSKeyValueObservingOptions)options
            context:(void *)context
```

it is *very* important that we pass a `context` that's unique to this class. We recommend putting:

OBJECTIVE-C SELECT ALL

```
static int const PrivateKVOContext;
```

at the top of the class' `.m` file and then calling the API with a pointer to this `PrivateKVOContext` as the context, like so:

OBJECTIVE-C SELECT ALL

```
[otherObject addObserver:self forKeyPath:@"someKey" options:s
```

and then implement the `-observeValueForKeyPath:...` method like so

OBJECTIVE-C SELECT ALL

```
- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context
{
    if (context == &PrivateKVOContext) {
        // Observe values here
    } else {
        [super observeValueForKeyPath:keyPath ofObject:object
    }
}
```

This ensures that subclassing works. With this pattern, both superclasses and subclasses can safely observe the same keys on the same objects without clashing. Otherwise, we'll end up running into odd behavior that's very difficult to debug.

Advanced Key-Value Observing

We often want to update some UI when a value changes, but we also need to initially run the code to update the UI once. We can use KVO to do both by specifying the `NSKeyValueObservingOptionInitial`. That will cause the KVO notification to trigger during the call to `-addObserver:forKeyPath:...`.

Before and After the Fact

When we register for KVO, we can also specify `NSKeyValueObservingOptionPrior`. This allows us to be notified before the value is changed. This directly corresponds to the point in time when `-willChangeValueForKey:` gets called.

If we register with `NSKeyValueObservingOptionPrior` we will receive two notifications: one before the change and one after the change. The first one will have another key in the `change` dictionary, and we can test if it's the notification prior to the change or the one after, like so:

OBJECTIVE-C SELECT ALL

```
if ([change[NSKeyValueChangeNotificationIsPriorKey] boolValue
    // Before the change
} else {
    // After the change
}
```

Values

If we need either the old or the new value (or both) of the key, we can ask KVO to pass those as part of the notification by specifying `NSKeyValueObservingOptionNew` and/or `NSKeyValueObservingOptionOld`.

This is often easier and better that using `NSKeyValueObservingOptionPrior`. We would extract the old and new values with:

OBJECTIVE-C SELECT ALL

```
id oldValue = change[NSKeyValueChangeOldKey];
id newValue = change[NSKeyValueChangeNewKey];
```

KVO basically stores the values for the corresponding key at the point in time where `-willChangeValueForKey:` and `-didChangeValueForKey:` were called, respectively.

Indexes

KVO also has very powerful support for notifying about changes to collections. These are returned for collection proxy objects returned by:

OBJECTIVE-C SELECT ALL

```
-mutableArrayValueForKey:
-mutableSetValueForKey:
-mutableOrderedSetValueForKey:
```

We'll describe how these work further below. If you use these, the change dictionary will contain information about the change kind (insertion, removal, or replacement) and for ordered relations, the change dictionary also contains information about the affected indexes.

The combination of collection proxy objects and these detailed change notifications can be used to efficiently update the UI when presenting large collections, but they require quite a bit of work.

Key-Value Observing and Threading

It is important to note that KVO happens synchronously and on the same thread as the actual change. There's no queuing or run-loop magic going on. The manual or automatic call to `-didChange...` will trigger the KVO notification to be sent out.

Hence, we must be very careful about not making changes to properties from another thread unless we can be sure that everybody observing that key can handle the change notification in a thread-safe manner. Generally speaking, we cannot recommend mixing KVO and

multithreading. If we are using multiple queues or threads, we should not be using KVO between queues or threads.

The fact that KVO happens synchronously is very powerful. As long as we're running on a single thread (e.g. the main queue) KVO ensures two important things.

First, if we call a KVO compliant setter, like:

```
OBJECTIVE-C                SELECT ALL
self.exchangeRate = 2.345;
```

we are guaranteed that all observers of `exchangeRate` have been notified by the time the setter returns.

Second, if the key path is observed with `NSKeyValueObservingOptionPrior`, someone accessing the `exchangeRate` property will stay the same until the `-observe...` method is called.

Key-Value Coding

Key-value coding in its simplest form allows us to access a property like:

```
OBJECTIVE-C                                    SELECT ALL
@property (nonatomic, copy) NSString *name;
```

through:

OBJECTIVE-C SELECT ALL

```
NSString *n = [object valueForKey:@"name"]
```

and:

OBJECTIVE-C SELECT ALL

```
[object setValue:@"Daniel" forKey:@"name"]
```

Note that this works for properties with object values, as well as scalar types (e.g. `int` and `CGFloat`) and structs (e.g. `CGRect`). Foundation will automatically do the wrapping and unwrapping for us. For example, if the property is:

OBJECTIVE-C SELECT ALL

```
@property (nonatomic) CGFloat height;
```

we can set it with:

OBJECTIVE-C SELECT ALL

```
[object setValue:@(20) forKey:@"height"]
```

Key-value coding allows us to access properties using strings to identify properties. These strings are called *keys*. In certain situations, this will give us a lot of flexibility which we can use to simplify our code. We'll look at an example in the next section, *Simplifying Form-Like User Interfaces*.

But there's more to key-value coding. Collections (`NSArray`, `NSSet`, etc.) have powerful collection operators which can be used with key-value

coding. And finally, an object can support key-value coding for keys that are not normal properties e.g. through proxy objects.

Simplifying Form-Like User Interfaces

Let's say we have an object:

OBJECTIVE-C SELECT ALL

```
@interface Contact : NSObject

@property (nonatomic, copy) NSString *name;
@property (nonatomic, copy) NSString *nickname;
@property (nonatomic, copy) NSString *email;
@property (nonatomic, copy) NSString *city;

@end
```

and a detail view controller that has four corresponding `UITextField` properties:

OBJECTIVE-C SELECT ALL

```
@interface DetailViewController ()

@property (weak, nonatomic) IBOutlet UITextField *nameField;
@property (weak, nonatomic) IBOutlet UITextField *nicknameFie
@property (weak, nonatomic) IBOutlet UITextField *emailField;
@property (weak, nonatomic) IBOutlet UITextField *cityField;

@end
```

We can now simplify the update logic. First, we need two methods that return us all model keys of interest to use and that map those keys to the keys of their corresponding text field respectively:

OBJECTIVE-C SELECT ALL

```
- (NSArray *)contactStringKeys;
{
    return @[@"name", @"nickname", @"email", @"city"];
}

- (UITextField *)textFieldForModelKey:(NSString *)key;
{
    return [self valueForKey:[key stringByAppendingString:@"F
}
```

With this, we can update the text field from the model, like so:

OBJECTIVE-C SELECT ALL

```
- (void)updateTextFields;
{
    for (NSString *key in self.contactStringKeys) {
        [self textFieldForModelKey:key].text = [self.contact
    }
}
```

We can also use a single-action method for all four text fields to update the model:

OBJECTIVE-C SELECT ALL

```
- (IBAction)fieldEditingDidEnd:(UITextField *)sender
{
    for (NSString *key in self.contactStringKeys) {
        UITextField *field = [self textFieldForModelKey:key];
        if (field == sender) {
            [self.contact setValue:sender.text forKey:key];
            break;
        }
    }
}
```

Note: We will add validation to this later, as pointed out at *Key-Value Validation*.

Finally, we need to make sure the text fields get updated when needed:

OBJECTIVE-C SELECT ALL

```
- (void)viewWillAppear:(BOOL)animated;
{
    [super viewWillAppear:animated];
    [self updateTextFields];
}

- (void)setContact:(Contact *)contact
{
    _contact = contact;
    [self updateTextFields];
}
```

And with this, our detail view controller is working.

Check out the entire project on our GitHub repository. It also uses *Key-Value Validation* as discussed further below.

Key Paths

Key-value coding also allows you to go through relations, e.g. if `person` is an object that has a property called `address`, and `address` in turn has a property called `city`, we can retrieve that through:

OBJECTIVE-C SELECT ALL

```
[person valueForKeyPath:@"address.city"]
```

Note that we're calling `-valueForKeyPath:` instead of `-valueForKey:` in this case.

Key-Value Coding Without `@property`

We can implement a key-value coding-compliant attribute without `@property` and `@synthesize` / auto-synthesize. The most straightforward example would be to simply implement the `-<key>` and `-set<Key>:` methods. For example, if we want to support setting `name`, we would implement:

OBJECTIVE-C SELECT ALL

```
- (NSString *)name;
- (void)setName:(NSString *)name;
```

This is straightforward, and identical to how `@property` works.

One thing to be aware of, though, is how `nil` is handled for scalar and struct values. Let's say we want to support key-value coding for `height` by implementing:

OBJECTIVE-C SELECT ALL

```
- (CGFloat)height;
- (void)setHeight:(CGFloat)height;
```

When we call:

OBJECTIVE-C SELECT ALL

```
[object setValue:nil forKey:@"height"]
```

this would throw an exception. In order to be able to handle `nil` values, we need to make sure to override `-setNilValueForKey:`, like so:

OBJECTIVE-C SELECT ALL

```
- (void)setNilValueForKey:(NSString *)key
{
    if ([key isEqualToString:@"height"]) {
        [self setValue:@0 forKey:key];
    } else
        [super setNilValueForKey:key];
}
```

We can make a class support key-value coding by overriding:

OBJECTIVE-C SELECT ALL

```
- (id)valueForUndefinedKey:(NSString *)key;
- (void)setValue:(id)value forUndefinedKey:(NSString *)key;
```

This may seem odd, but it allows a class to dynamically support certain keys. Using these two methods comes with a performance hit, though.

As a side note, it is worth mentioning that Foundation supports accessing instance variables directly. Use that feature sparingly. Check the documentation for `+accessInstanceVariablesDirectly`. It defaults to `YES`, which causes Foundation to look for an instance variable called `_<key>`, `_is<Key>`, `<key>`, or `is<Key>`, in that order.

Collection Operators

An oft-overlooked feature of key-value coding is its support for collection operators. For example, we can get the maximum value from an array with:

OBJECTIVE-C SELECT ALL

```
NSArray *a = @[@4, @84, @2];
NSLog(@"max = %@", [a valueForKeyPath:@"@max.self"]);
```

or, if we have an array of `Transaction` objects that have an `amount` property, we can get the maximum `amount` with:

OBJECTIVE-C SELECT ALL

```
NSArray *a = @[transaction1, transaction2, transaction3];
NSLog(@"max = %@", [a valueForKeyPath:@"@max.amount"]);
```

When we call `[a valueForKeyPath:@"@max.amount"]`, this will call `-valueForKey:@"amount"` on each element in the array `a` and then return the maximum of those.

Apple's documentation for key-value coding has a section called Collection Operators that describes this in detail.

Key-Value Coding Through Collection Proxy Objects

We can expose collections (`NSArray`, `NSSet`, etc.) in the same way as normal objects. But key-value coding also allows us to implement a key-value coding-compliant collection though proxy objects. This is an advanced technique. We'll rarely find use for it, but it's a powerful trick to have in the tool chest.

When we call `-valueForKey:` on an object, that object can return collection proxy objects for an `NSArray`, an `NSSet`, or an `NSOrderedSet`. The class doesn't implement the normal `-<Key>`

method but instead implements a number of methods that the proxy uses.

If we want the class to be able to support returning an `NSArray` through a proxy object for the key `contacts`, we could implement:

OBJECTIVE-C SELECT ALL

```
- (NSUInteger)countOfContacts;
- (id)objectInContactsAtIndex:(NSUInteger)idx;
```

Doing so, when we call `[object valueForKey:@"contacts"]`, this will return an `NSArray` that proxies all calls to those two methods. But the array will support *all* methods on `NSArray`. The proxying is transparent to the caller. In other words, the caller doesn't know if we return a normal `NSArray` or a proxy.

We can do the same for an `NSSet` and `NSOrderedSet`. The methods that we have to implement are:

NSArray	NSSet	NSOrderedSet
`-countOf<Key>`	`-countOf<Key>`	`-countOf<Key>`
	`-enumeratorOf<Key>`	`-indexIn<Key>OfObj`
One of	`-memberOf<Key>:`	
`-objectIn<Key>AtIndex:`		One of
`-<key>AtIndexes:`		`-objectIn<Key>AtIn`
		`-<key>AtIndexes:`
Optional (performance)		
`-get<Key>:range:`		Optional (performan

NSArray	NSSet	NSOrderedSet
		-get<Key>:range:

The *optional* methods can improve performance of the proxy object.

Using these proxy objects only makes sense in special situations, but in those cases it can be very helpful. Imagine that we have a very large existing data structure and the caller doesn't need to access all elements (at once).

As a (perhaps contrived) example, we could write a class that contains a huge list of primes, like so:

OBJECTIVE-C SELECT ALL

```
@interface Primes : NSObject

@property (readonly, nonatomic, strong) NSArray *primes;

@end


@implementation Primes

static int32_t const primes[] = {
    2, 101, 233, 383, 3, 103, 239, 389, 5, 107, 241, 397, 7,
    251, 401, 11, 113, 257, 409, 13, 127, 263, 419, 17, 131,
    421, 19, 137, 271, 431, 23, 139, 277, 433, 29, 149, 281,
    31, 151, 283, 443, 37, 157, 293, 449, 41, 163, 307, 457,
    167, 311, 461, 47, 173, 313, 463, 53, 179, 317, 467, 59,
    331, 479, 61, 191, 337, 487, 67, 193, 347, 491, 71, 197,
    499, 73, 199, 353, 503, 79, 211, 359, 509, 83, 223, 367,
    89, 227, 373, 523, 97, 229, 379, 541, 547, 701, 877, 1049
    557, 709, 881, 1051, 563, 719, 883, 1061, 569, 727, 887,
    1063, 571, 733, 907, 1069, 577, 739, 911, 1087, 587, 743,
    919, 1091, 593, 751, 929, 1093, 599, 757, 937, 1097, 601,
    761, 941, 1103, 607, 769, 947, 1109, 613, 773, 953, 1117,
    617, 787, 967, 1123, 619, 797, 971, 1129, 631, 809, 977,
    1151, 641, 811, 983, 1153, 643, 821, 991, 1163, 647, 823,
    997, 1171, 653, 827, 1009, 1181, 659, 829, 1013, 1187, 66
    839, 1019, 1193, 673, 853, 1021, 1201, 677, 857, 1031,
    1213, 683, 859, 1033, 1217, 691, 863, 1039, 1223, 1229,
```

```
};

- (NSArray *)primes;
{
    return [self valueForKey:@"backingPrimes"];
}

- (NSUInteger)countOfBackingPrimes;
{
    return (sizeof(primes) / sizeof(*primes));
}

- (id)objectInBackingPrimesAtIndex:(NSUInteger)idx;
{
    NSParameterAssert(idx < sizeof(primes) / sizeof(*primes));
    return @(primes[idx]);
}

@end
```

We would be able to run:

OBJECTIVE-C SELECT ALL

```
Primes *primes = [[Primes alloc] init];
```


This would call `-countOfPrimes` once and then `-objectInPrimesAtIndex:` once with `idx` set to the last index. It would *not* have to wrap all the integers into an `NSNumber` first and then wrap all those into an `NSArray`, only to then extract the last object.

The *Contacts Editor* sample app uses the same method to wrap a C++ `std::vector` – in a contrived example. But it illustrates how this method can be used.

Mutable Collections

We can even use collection proxies for mutable collections, i.e. `NSMutableArray`, `NSMutableSet`, and `NSMutableOrderedSet`.

Accessing such a mutable collection works slightly differently. The caller now has to call one of these methods:

OBJECTIVE-C SELECT ALL

```
- (NSMutableArray *)mutableArrayValueForKey:(NSString *)key;
- (NSMutableSet *)mutableSetValueForKey:(NSString *)key;
- (NSMutableOrderedSet *)mutableOrderedSetValueForKey:(NSStri
```

As a trick, we can have the class return a mutable collection proxy through:

OBJECTIVE-C SELECT ALL

```
- (NSMutableArray *)mutableContacts;
{
    return [self mutableArrayValueForKey:@"wrappedContacts"];
}
```


We would have to implement both the method listed above for the immutable collection, as well as these:

NSMutableArray / NSMutableOrderedSet	NSMutableSet
At least 1 insertion and 1 removal method	At least 1 addition and removal method
`-insertObject:in<Key>AtIndex:`	`-add<Key>Object:`
`-removeObjectFrom<Key>AtIndex:`	`-remove<Key>Object`
`-insert<Key>:atIndexes:`	`-add<Key>:`

NSMutableArray / NSMutableOrderedSet	NSMutableSet
`-remove<Key>AtIndexes:`	`-remove<Key>:`
Optional (performance) one of	Optional (performance
`-replaceObjectIn<Key>AtIndex:withObject:`	`-intersect<Key>:`
`-replace<Key>AtIndexes:with<Key>:`	`-set<Key>:`

As noted above, these mutable collection proxy objects are also very powerful in combination with key-value observing. The KVO mechanism will put detailed change information into the change dictionary when these collections are mutated.

There are batch-change methods (taking multiple objects) and ones that only take a single object. We recommend picking the one that's the easiest to implement for the given task – with a slight favor for the batch update ones.

notifications. If we choose to implement the fine-grained notifications ourselves through:

OBJECTIVE-C SELECT ALL

```
-willChange:valuesAtIndexes:forKey:
-didChange:valuesAtIndexes:forKey:
```

or:

OBJECTIVE-C SELECT ALL

```
-willChangeValueForKey:withSetMutation:usingObjects:
-didChangeValueForKey:withSetMutation:usingObjects:
```

we need to make sure to turn automatic notifications off, otherwise KVO will send out two notifications for every change.

Common Key-Value Observing Mistakes

First and foremost, KVO compliance is part of an API. If the owner of a class doesn't promise that the property is KVO compliant, we cannot make any assumption about KVO to work. Apple does document which properties are KVO compliant. For example, the `NSProgress` class lists most of its properties to be KVO compliant.

Sometimes people try to trigger KVO by putting `-willChange` and `-didChange` pairs with nothing in between *after* a change was made. This

`NSKeyValueObservingOld` property to support observing key paths.

We would also like to point out that collections as such are not observable. KVO is about observing *relationships* rather than collections. We cannot observe an `NSArray`; we can only observe a property on an object – and that property may be an `NSArray`. As an example, if we have a `ContactList` object, we can observe its `contacts` property, but we cannot pass an `NSArray` to `-addObserver:forKeyPath:...` as the object to be observed.

Likewise, observing `self` doesn't always work. It's probably not a good design pattern, either.

Debugging Key-Value Observing

Inside `lldb` you can dump the observation info of an observed object, like so:

OBJECTIVE-C SELECT ALL

```
(lldb) po [observedObject observationInfo]
```

This prints lots of information about who's observing what.

The format is private and we mustn't rely on anything about it – Apple is free to change it at any point in time. But it's a very powerful debugging tool.

Key-Value Validation

provides any logic or functionality on its own.

But if we're writing model classes that can validate values, we should implement the API the way set forward by key-value validation to make sure it's consistent. Key-value validation is the Cocoa convention for validating values in model classes.

Let us stress this again: key-value coding will not do any validation, and it will not call the key-value validation methods. Your controller will have to do that. Implementing your validation methods according to key-value validation will make sure they're consistent, though.

A simple example would be:

OBJECTIVE-C SELECT ALL

```
- (IBAction)nameFieldEditingDidEnd:(UITextField *)sender;
{
    NSString *name = [sender text];
    NSError *error = nil;
    if ([self.contact validateName:&name error:&error]) {
        self.contact.name = name;
    } else {
        // Present the error to the user
    }
    sender.text = self.contact.name;
}
```

The powerful thing is that we're asking the model class (`Contact` in this case) to validate the `name` , and at the same time we're giving the model class an opportunity to sanitize the name.

If we want to make sure the name doesn't have any leading white space, that's logic which should live inside the model object. The `Contact` class would implement the key-value validation method for the `name` property


```
- (BOOL)validateName:(NSString **)nameP error:(NSError * __au
{
    if (*nameP == nil) {
        *nameP = @"";
        return YES;
    } else {
        *nameP = [*nameP stringByTrimmingCharactersInSet:[NSC
        return YES;
    }
}
```

The *Contact Editor* sample illustrates this in the `DetailViewController` and `Contact` class.

CONTINUE READING ISSUE 7

Foundation

December 2013