

IOS

Understanding Key-Value Observing and Coding



GABRIEL THEODOROPOULOS



13TH AUG '14



37

In programming, one of the most commonly accepted facts is that the flow of a program depends on the value of the various variables and properties you use. You just have to think how many times during an application development you have to check for your properties' values, and based on them to drive the execution of the code to one or another way. Even more, think of how many times you need to check if an object has been added to an array, or if it has been removed from it. So, being aware of the changes that happen to the properties of your classes is a vital part of the programming process.

There are various ways that let us be notified when a property gets changed. For example, if you have a method to set a property's value, or you just override a setter method, you can send a *notification* (*NSNotification*) to notify an observer about that change, and then perform the proper actions based on the value that was set. If you are familiarized with notifications, then using them would not be a problem at all if you would want to use them for being aware about changes in properties, but the truth is that doing so for a great number of properties would require to send and observe for a great number of notifications as well, and that could lead to a lot of

extra code writing.

Nevertheless, there is a much better way to observe for changes in properties, and Apple uses it a lot in its own software. That way is not used a lot by programmers because it seems hard to be learnt, however this is not true. Once you get to know it, you'll probably love it and you'll see that it's much effortless and easier to track down changes on single properties or collections, such as arrays. This method is called **Key-Value Observing**, but is mostly known as **KVO**.

Key-Value Observing (KVO) is related directly to another powerful and important mechanism, named **Key-Value Coding**, or **KVC**. Actually, any property you want to observe for changes must be Key-Value Coding compliant, but we will talk more about that later. The important thing for now is to make clear that both of them provide a really powerful and efficient way to write code, and knowing how to handle them can definitely turn you into a more skilled and advanced programmer.

My goal in this tutorial is to walk you through the basics of both of them, and to show you step by step how they can be used. Covering all the aspects regarding these topics in one tutorial is not possible to happen, however after you have finished this one you will be able to manage them efficiently, and you will know what to look for if you need more details on them.

Before we take off and start examining their insides, I should necessarily mention that we will begin our journey from the Key-Value Coding mechanism. It's necessary to know what it is and how it is used before we proceed to the Key-Value Observing mechanism, as the first is prerequisite for the second. So, if you are ready, let's get started!

Demo App Overview

As always, this section is dedicated to the description of what the demo app is all about. So, I will start talking about it, highlighting a special characteristic the app will have. That is the fact that we won't do any visual work at all, neither we will work with the Simulator or a device. Actually, the results of our implementation will be displayed on the console only.

Being more detailed, the contents of this tutorial are best suited to a *command line tool* application, as our topic concerns a clearly programming concept, for which we won't create one sample app from scratch. Instead, we will be moving by implementing many small examples that demonstrate exactly the point of each discussed aspect of our topic, and we will use *NSLog* commands to output the results of our work. Nevertheless, we are still going to create a single view iOS application, but we'll use no views or subviews to input and output any kind of data. The default view controller's view will just remain empty.

The reason that made me choose an empty iOS application instead of a command line application is simple: I want you to feel comfortable by working to a familiar environment. KVC and KVO are surely new concepts for many of you, so I would like you to focus on these only. I wouldn't like to distract you by making you to work for first time to a command line application. Methods like *viewDidLoad*, *viewWillAppear:*, etc., are definitely much more familiar to you (even though I must admit that I would prefer to create a command line app).

So, as I said we will be using only *NSLog* commands to output the results of our work. Besides that, we are going to create a couple of classes for the sake of the tutorial. The first one will exist for demonstration reasons only, while the second class can be used as a reusable code to your own apps. Generally speaking, by keeping things simple, it will be easier for you to get the point of the Key-Value

Coding and Key-Value Observing.

Creating the Sample App

So, the first step is to create a new project, therefore go and launch Xcode and select to create a new project. To the guide that appears, select the **Single View Application** template, in the **Application** category under the **iOS** section.

Proceed to the next step, and in the **Product Name** field set the **KVCO Demo** name. Leave everything else as it is, and go to the last step, where you should choose a directory to save the project, and you're done.

The project is now ready!

Key-Value Coding

Here is the definition of the Key-Value Coding according to Apple's official documentation:

Key-value coding is a mechanism for accessing an object's properties indirectly, using strings to identify properties, rather than through invocation of an accessor method or accessing them directly through instance variables.

What does this mean? Well, let's see it through some simple examples:

Let's suppose that we have a property named *firstname*, and we want to assign the

value *John* to it. Normally, what we would write in code to do it is this:

```
self.firstname = @"John";
```

or this:

```
_firstname = @"John";
```

Quite familiar, right? Now, using the KVC mechanism the above assignment would look like the next one:

```
[self setValue:@"John" forKey:@"firstname"];
```

If you look closely, this looks similar to the way we set values to dictionaries, or when converting scalar values and structs to *NSValue* objects. As you see, we set the value *John* for the key *firstname*. One more example:

```
[someObject.someProperty setText:@"This is a text"];
```

Using KVC:

```
[self setValue:@"This is a text" forKeyPath:@"someObject.somePr
```

In both of these examples, instead of directly setting the value (first example) to the

property or use the setter method (second example) of the property, we simply match values to keys or keypaths (more about keys and keypaths in just a moment). As you assume, because we use keys and values, the above technique is called Key-Value Coding.

So, returning to Apple's definition, you notice that we use strings (the keys) to indirectly access the properties we want to assign values to, instead of doing so directly. On the other hand now, to get values from KVC properties, we would write something like that:

```
NSLog(@"%@", [self valueForKey:@"firstname"]);
```

Now that you have started getting into the point, let's discuss about some important stuff before we see an actual example on Xcode. First of all, there is an informal protocol, named *NSKeyValueCoding*, and your class (or classes) must comply with it if you want to use KVC. If your class is a subclass of the *NSObject* though, then you are already set, because *NSObject* complies to that protocol.

Besides that, as you saw in the above two examples we used two different methods to set the values to properties. The first one is the *setValue:forKey:* and the second is the *setValue:forKeyPath:*. The difference is that the first method expects a key (which is a string value), while the second expects a keypath (also a string value). So, what is a key and what a keypath?

Simply put, the *key* specifies a single property, the one we want to set a value to or get one from, so its name should be similar to the property's one. In our example, we had the *firstname* property, and that was exactly the key we used when we set its value using the KVC.

A *keypath* is always formed using the dot syntax, so it's not a single word, and represents all the properties of an object that should be traversed until the desired one is found (speaking totally non-technically so as to make things clear, I would say that the keypath looks for the subproperty of the subproperty of the ... (and so on) ... of an object). For example, the *someObject.someProperty.text* keypath we set to the second example above refers to the *text* property of the *someProperty* of the *someObject* object.

With all that said, it's time to go to Xcode and see some more examples there. Don't worry if things look still obscure to you, everything will become lighter as we move on.

On Xcode, begin by creating a new class. To do so, open the **File > New > File...** menu, and in the window that appears select the **Objective-C class** as the template for our new file.

Next, go to the second step and set the **NSObject** value to the **Subclass of** field. In the **Class** textfield, specify the **Children** value as the name of the new class.

Get finished with the guide, and let Xcode to create the new class and add the couple new files to the project. In the class we just created we are going to add properties regarding the name and the age of a hypothetical person's children, and the way we will use it is perfect to demonstrate all the major aspects of the Key-Value Coding.

Now, open the *Children.h* file and modify it as shown next. As you'll see, we add only two properties, one for the name and one for the age of the child represented by an instance of that class.

```
@interface Children : NSObject

@property (nonatomic, strong) NSString *name;

@property (nonatomic) NSUInteger age;

@end
```

Next, open the *Children.m* file, and initialise the above two properties:

```
@implementation Children

- (instancetype)init
{
    self = [super init];
    if (self) {
        self.name = @"";
        self.age = 0;
    }
    return self;
}

@end
```

It is time to see KVC in action now. Open the *ViewController.m* file and at the top of it import the header of the *Children* class:

```
#import "Children.h"
```

Next, declare three objects of the *Children* class to the private class section as it's shown below:


```
@interface ViewController ()

@property (nonatomic, strong) Children *child1;

@property (nonatomic, strong) Children *child2;

@property (nonatomic, strong) Children *child3;

@end
```

Then, head to the *viewDidLoad* method, where we will initialise the first object:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically

    self.child1 = [[Children alloc] init];
}
```

If we assign values to the *name* and *age* properties of the *child1* object, and then show them on the debugger as follows...

```
- (void)viewDidLoad
{
    ...

    self.child1.name = @"George";
    self.child1.age = 15;

    NSLog(@"%@, %d", self.child1.name, self.child1.age);
}
```

... then we will get the expected output:

Now, let's try to use the Key-Value Coding method to do the same thing. If you want, either comment out or delete the previous lines in the *viewDidLoad*, just make sure to leave the *child1* object initialisation intact.

```
- (void)viewDidLoad
{
    ...

    [self.child1 setValue:@"George" forKey:@"name"];
    [self.child1 setValue:[NSNumber numberWithInt:15] forKey:@"age"];

    NSString *childName = [self.child1 valueForKey:@"name"];
    NSUInteger childAge = [[self.child1 valueForKey:@"age"] integerValue];

    NSLog(@"%@, %d", childName, childAge);
}
```

In the first couple of rows we set the desired values to both properties using the *setValue:forKey:* method. Pay attention to the fact that the age is a number, therefore it cannot be passed directly as an argument to that method. Instead, we must convert it to a *NSNumber* object first. Besides that, watch that the key strings are the same to the properties's names.

Next, we perform the exact opposite task. We extract the values out of the properties using the *valueForKey:* method, and we assign them to two local variables. Focusing on the age again, notice that we want to get an unsigned integer value, so we convert the returned data type from *NSNumber* to *NSUInteger*.

Finally, we display everything to the debugger. The output in this case is similar to

the previous one:

George, 15

That's great, but you may ask yourself if it actually worths it to write the extra lines. Well, as you'll find out next when we'll talk about the Key-Value Observing, yes it is.

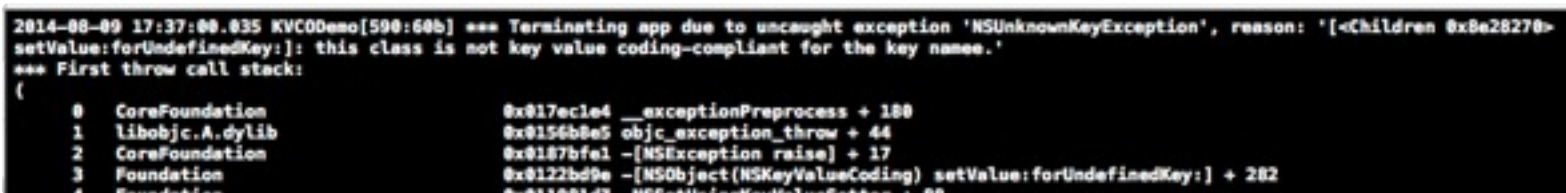
Now, replace this line...

```
[self.child1 setValue:@"George" forKey:@"name"];
```

... with this one:

```
[self.child1 setValue:@"George" forKey:@"namee"];
```

In case you read too fast and you don't see the modification, there's an extra “;e”; letter to the key string. If you run the app once again then... Boom! The app crashes! Here's a screenshot from the debugger:



```
2014-08-09 17:37:00.035 KVCODemo[590:60b] *** Terminating app due to uncaught exception 'NSUnknownKeyException', reason: '[<Children 0x8e28270>
setValue:forUndefinedKey:]: this class is not key value coding-compliant for the key namee.'
*** First throw call stack:
(
  0 CoreFoundation 0x17ec1e4 __exceptionPreprocess + 180
  1 libobjc.A.dylib 0x156b8e5 objc_exception_throw + 44
  2 CoreFoundation 0x187bfe1 -[NSException raise] + 17
  3 Foundation      0x122bd9e -[NSObject(NSKeyValueCoding) setValue:forUndefinedKey:] + 282
  4 Foundation      0x1168147 NSSetUserKeyValuesForKey + 88
```

The actual message before all that bunch with the technical information says:

Terminating app due to uncaught exception 'NSUnknownKeyExcepti

This crash message is quite explanatory. In simple words, it's telling us that the app crashed because no property found matching to the key *namee*. Okay, no big deal as we can go back to our code and fix this error. However, let's see the bottom line here: When writing KVC-compliant code it's really important to take care so the key strings actually match to the property names, otherwise the app will simply fall apart. This is not a case when dealing directly with properties, as there's no chance to do any mistake on their names; the compiler would throw an error in such a case which would drive us to fix it.

With all the above, we have managed to see how to write KVC-styled code, and how to set and get values using keys. Also, we have seen and underlined what happens if a key name is mistyped. Now, let's extend our example a bit more, and let's see how we can work with keypaths. For starters, go to the *Children.h* file, and add the next property to the class:

```
@interface Children : NSObject

...

@property (nonatomic, strong) Children *child;

@end
```

With this, we can represent in code the child of a child (and so on). Return to the *ViewController.m* file, in the *viewDidLoad* method. Now, add the next lines that initialize the related objects and assign their initial values:

```
- (void)viewDidLoad
```

```

{
    ...

    self.child2 = [[Children alloc] init];

    [self.child2 setValue:@"Mary" forKey:@"name"];
    [self.child2 setValue:[NSNumber numberWithInt:35] forKey:@"age"];
    self.child2.child = [[Children alloc] init];
}

```

Nothing new here, we just initialize the *child2* object of our custom class, we set the values of the name and the age properties, and we then initialize its *child* object.

The point now is how we can set values to the properties of the *child* object using the KVC style. According to what we said in the introduction of this tutorial, we must use the *setValue:forKeyPath:* method to do so. The key that we will provide is going to be a string with the dot syntax to point out the property of the object we want to change. In code:

```

- (void)viewDidLoad
{
    ...

    [self.child2 setValue:@"Andrew" forKeyPath:@"child.name"];
    [self.child2 setValue:[NSNumber numberWithInt:5] forKeyPath:@"child.age"];
}

```

It's not difficult, right? Now, simply by adding the next *NSLog* command, let's see if the assignment was successful:

```

- (void)viewDidLoad
{
    ...

```

```
        NSLog(@"%@", %d", self.child2.child.name, self.child2.child.c  
    }
```

Output:

Andrew, 5

Perfect! Now, what if the child of the child has a child too? Let's see this case too (the commands are given altogether):

```
- (void)viewDidLoad  
{  
    ...  
  
    self.child3 = [[Children alloc] init];  
    self.child3.child = [[Children alloc] init];  
    self.child3.child.child = [[Children alloc] init];  
  
    [self.child3 setValue:@"Tom" forKeyPath:@"child.child.name"]  
    [self.child3 setValue:[NSNumber numberWithInt:2] forKeyPath:@"child.child.age"]  
  
    NSLog(@"%@", %d", self.child3.child.child.name, self.child3.child.child.age)  
}
```

The output is:

Tom, 2

Great! So, now you know how to deal with both keys and keypaths, and how to write

KVC compliant code. If you want, feel free to write more examples and see what results you get back.

Observing for Property Changes

Now that you know the basics of the Key-Value Coding let's move on and let's focus on the Key-Value Observing (KVO). Here we will see what actions should be taken in order to be able to track down changes on properties. First of all, let me introduce you as a list the steps needed to implement KVO:

1. The class of which you want to observe its properties must be KVO compliant. That means:
 - The class must be KVC compliant according to what we have already seen in the introduction and in the previous section.
 - The class must be able to send notifications either automatically, or manually (we will see more about that later).
2. The class that will be used to observe the property of another class should be set as *observer*.
3. A special method named *observeValueForKeyPath:ofObject:change:context:* should be implemented to the observing class.

Let's see everything one by one. The most important thing when we want to observe for changes of a property, is to *make* our class observe for these changes. This is done more or less with as with the casual notifications (*NSNotification*), but using another method. This method is the *addObserver:forKeyPath:options:context:*. In order to become more specific, I will

use our previous example where we used the *Children* class. In this class we had three properties, the *name*, the *age* and the *child* which was of type *Children* as well. To observe for the first two of them, here's what is only needed (add the *viewWillAppear:* method and then write the next lines to it, in the *ViewController.m* file):

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

    [self.child1 addObserver:self forKeyPath:@"name" options:NS
    [self.child1 addObserver:self forKeyPath:@"age" options:NS
}
}
```

The parameters the above method accepts are:

- *addObserver:* This is the observing class, usually the *self* object.
- *keyPath:* I guess you can understand what's this for. It is the string you used as a key or a key path and matches to the property you want to observe. Note that you specify here either a single key, or a key path.
- *options:* By setting a value other than 0 (zero) to this parameter, you specify what the notification should contain. You can set a single value, or a combination of *NSKeyValueObservingOptions* values, combining them using the logical or (*|*). In the above example, we ask from the notifications to contain both the old and the new value of the properties we observe.
- *context:* This is a pointer that can be used as a unique identifier for the change of the property we observe. Usually this is set to *nil* or *NULL*. We'll see more about this later.

Now that we have made our class able to observe for any changes in the above two properties, we must implement the *observeValueForKeyPath:ofObject:change:context:* method. Its implementation is mandatory, and it has one great disadvantage. That is the fact that is called for every KVO change, and if you observe many properties then you must write a lot of *if* statements so as to take the proper actions for each property. However, this can be easily overlooked as the benefits of the KVO are greater than this limitation.

Now, let's implement it.

```
-(void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)ofObject
forKeyPath:(NSString *)keyPath {
    if ([keyPath isEqualToString:@"name"]) {
        NSLog(@"The name of the child was changed.");
        NSLog(@"%@", change);
    }

    if ([keyPath isEqualToString:@"age"]) {
        NSLog(@"The age of the child was changed.");
        NSLog(@"%@", change);
    }
}
```

As you see, we do nothing special here (for the sake of the demonstration). What we only do, is to determine the property that was changed based on the *keyPath* parameter, and then to display just a single message along with the dictionary containing the changes that took place. In a real application, you apply all the needed logic to handle all the changes in each case.

Time to test it, don't you think? In the *viewWillAppear:* method, add the next line:

```
-(void)viewWillAppear:(BOOL)animated{
    ...

    [self.child1 setValue:@"Michael" forKey:@"name"];
}
```

Let's run the app and let's see what will be displayed on the debugger:

```
The name of the child was changed.
{
    kind = 1;
    new = Michael;
    old = George;
}
```

Super! After we have set a new value to the *name* property of the *child1* object, we received the notification, and the messages we asked for to be displayed were shown on the debugger. As you see, both the previous and the new value are included in the dictionary.

Now, let's change the *age* property of the *child1* object to see if the notification will arrive:

```
-(void)viewWillAppear:(BOOL)animated{
    ...

    [self.child1 setValue:[NSNumber numberWithInt:20] forKey:@"age"];
}
```

And when we run it:

The age of the child was changed.

```
{
    kind = 1;
    new = 20;
    old = 15;
}
```

So, everything seems to be working great. From the *change* dictionary you can extract any value you want (if needed), but the most important of all is that it's super-easy to be notified about changes in properties. If all these look new to you, don't worry. It's all just a matter of habit!

We are going to make everything much more interesting now, simply by adding one more observer for the *age* property of the *child2* object. After that, we will assign a new value to that property.

```
-(void)viewWillAppear:(BOOL)animated{
    ...

    [self.child2 addObserver:self forKeyPath:@"age" options:NS
    [self.child2 setValue:[NSNumber numberWithInt:45] forKeyPath:@"age"]
}
```

If we run the app, the output will be this:

The name of the child was changed.

```
{
kind = 1;
new = Michael;
old = George;
}
```

The age of the child was changed.

```
{  
    kind = 1;  
    new = 20;  
    old = 15;  
}
```

The age of the child was changed.

```
{  
    kind = 1;  
    new = 45;  
    old = 35;  
}
```

As you notice, we receive two notifications regarding changes to the *age* property. But that seems confusing, because even though we know the object that each notification belongs to, programmatically we can do nothing to determine the object that sent the notification. So, how do we face that, and how can we programmatically be a 100% sure about the object that the changed property belongs to?

The answer to the above question is one: We will make use of the *context* argument of the *addObserver:forKeyPath:options:context:* method. I have already mentioned before that the purpose of the context is to uniquely identify a change on a property, so it's the best tool we have at our disposal. As I have said, the context is a pointer, and it actually points to itself, so the way that is declared is the following:

```
static void *someContext = &someContext;
```

Note that the context value for each observed property must be a global variable, because it has to be accessible from both the *addObserver...* and the *observeValueForKeyPath...* methods. Let's get back to our example, and let's add a

couple of context values. In the *ViewController.m* file, go at the very top of it, just right before the *@interface ViewController ()* line. There, add the next two commands:

```
static void *child1Context = &child1Context;
static void *child2Context = &child2Context;
```

Alternatively, instead of declaring these two as global variables, you could use them as properties of your class, but I consider the first way to be much simpler.

Now, we just have to modify the code in the *viewWillAppear:* method by setting the proper context value to the *addObserver:forKeyPath:options:context:* calls. Here it is in one piece:

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

    [self.child1 addObserver:self forKeyPath:@"name" options:N
    [self.child1 addObserver:self forKeyPath:@"age" options:NS

    [self.child1 setValue:@"Michael" forKey:@"name"];
    [self.child1 setValue:[NSNumber numberWithInt:20] forKey:@"age"]

    [self.child2 addObserver:self forKeyPath:@"age" options:NS

    [self.child2 setValue:[NSNumber numberWithInt:45] forKey:@"age"]
}
```

Finally, we have to modify the *observeValueForKeyPath:ofObject:change:context:* appropriately, so we can programmatically determine the object that sends the notification.

```

- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)ofObject
inContext:(id)context {
    if (context == child1Context) {
        if ([keyPath isEqualToString:@"name"]) {
            NSLog(@"The name of the FIRST child was changed.");
            NSLog(@"%@", change);
        }

        if ([keyPath isEqualToString:@"age"]) {
            NSLog(@"The age of the FIRST child was changed.");
            NSLog(@"%@", change);
        }
    }
    else if (context == child2Context){
        if ([keyPath isEqualToString:@"age"]) {
            NSLog(@"The age of the SECOND child was changed.");
            NSLog(@"%@", change);
        }
    }
}

```

I said before that the drawback of using this method is that you have to write a bunch of *if* statements, and now you can see this fact. Anyway, you can clearly understand how easy it is to identify a property that was changed based on the context and the key path. Running the app once again, we get the next output:

```

The name of the FIRST child was changed.
{
    kind = 1;
    new = Michael;
    old = George;
}

The age of the FIRST child was changed.
{
    kind = 1;
    new = 20;
    old = 15;
}

```

The age of the SECOND child was changed.

```
{  
    kind = 1;  
    new = 45;  
    old = 35;  
}
```

Much better now! We modified and messages in such way, so they are more explanatory and of course, we managed to programmatically specify each changed property.

Lastly and before we reach at the end of this chapter, it's also quite important at some point to remove the observers you add. There is not a recipe on where you should do that. For instance, in many cases it would be useful to do that in the *observeValueForKeyPath:ofObject:change:context:*, after you have handled a received notification. In other cases, you should do so upon the dismissal of a view controller. Generally, it's up to your application's structure the decision you will make about that. In this example, we will do it in the *viewWillDissappear:* method. Here it is:

```
-(void)viewWillDisappear:(BOOL)animated{  
    [super viewWillDisappear:animated];  
  
    [self.child1 removeObserver:self forKeyPath:@"name"];  
    [self.child1 removeObserver:self forKeyPath:@"age"];  
    [self.child2 removeObserver:self forKeyPath:@"age"];  
}
```

So, now you know the most important steps you should make when you want to apply Key-Value Observing logic to your application. As you found out, the rules that should be followed are simple, and all the hard work of monitoring for changes

and sending notifications is done by iOS itself. However, there are more details we should discuss about, but the knowledge you acquired up to this point is the most significant.

Automatic and Manual Notifications

By default, the system sends a notification every time a property gets changed when you observe using KVO. This is suitable in most cases, however there are times that we don't want to get a notification once a change has happened, but after a bunch of changes have taken place in multiple properties or at a later time. Thankfully, iOS SDK provides us with some quite convenient methods that gives us control over the notifications, so we can manually send them whenever it's actually needed. Before we get into more details, let me just say that using the method you'll see right next is not mandatory. On the contrary, you may implement it if and when it is really necessary.

Getting into the point now, in order to control the notifications that are sent upon property changes, you must implement the *automaticallyNotifiesObserverForKey:* class method. The parameter it accepts is a string representation of the key of the property for which you need to control the notification, and it returns a boolean value. In case that you don't want a notification to be sent after the observed property's value has been changed, then the method must return NO. In any other case, you should let iOS decide about the notifications (you'll see how in the example that follows).

In practice, let's suppose that we don't want a notification to be posted when the *name* property of the *Children* class get changed. With that in mind, here's the implementation of that method in the *Children* class (*Children.m* file):

```
+(BOOL)automaticallyNotifiesObserversForKey:(NSString *)key{
```



```
    if ([key isEqualToString:@"name"]) {  
        return NO;  
    }  
    else{  
        return [super automaticallyNotifiesObserversForKey:key  
    }  
}
```

I believe that the method is quite straightforward. In the *else* clause, we call the same method using the *super* class in order to let iOS handle all the keys that we haven't explicitly added here, and the value that we get back is the one that we return at the end.

If you run the app at this point, you'll find out that no message regarding the name changing is appeared on the debugger. Of course, this is what we desire, so we've managed to achieve our goal. But, have we really done it?

Well, as you understand by returning NO in the above method for the specific key, we've only managed to stop the relevant notifications from being sent. And of course, this doesn't mean manual notifications, it means no notifications at all! In order to send a notification when we decide so, we must use two other methods. These are the *willChangeValueForKey:* and the *didChangeValueForKey:*. When using them, the *willChangeValueForKey:* must be called first, then the new value must be assigned to the property, while the *didChangeValueForKey:* should be called at the end.

Back in the action again, go to the *ViewController.m* file and then to the *viewWillAppear:* method. Next, replace this line:

```
[self.child1 setValue:@"Michael" forKey:@"name"];
```

with these three:

```
[self.child1 willChangeValueForKey:@"name"];  
self.child1.name = @"Michael";  
[self.child1 didChangeValueForKey:@"name"];
```

If you run the app now, the message regarding the name change is appeared on the debugger, and that means that we've sent the notification manually with success!

```
The name of the FIRST child was changed.  
{  
    kind = 1;  
    new = Michael;  
    old = George;  
}
```

Actually, the notification is sent after the *didChangeValueForKey:* method is invoked, therefore if you don't want to send the notification right after the value assignment, simply get this line:

```
[self.child1 didChangeValueForKey:@"name"];
```

and place it right to the point that is appropriate for the notification to be sent. For example, if we add the above call at the end of the *viewWillAppear:* method, then the name change messages will be the last appearing on the debugger when you run the app:

```
The age of the FIRST child was changed.  
{
```

```
kind = 1;
new = 20;
old = 15;
}
```

The age of the SECOND child was changed.

```
{
    kind = 1;
    new = 45;
    old = 35;
}
```

The name of the FIRST child was changed.

```
{
    kind = 1;
    new = Michael;
    old = George;
}
```

That's what exactly we have been expecting from the app! As you can see, we've managed to control the notification sending point, and all that with a little effort! Note that between the *willChangeValueForKey:* and the *didChangeValueForKey:* methods, you can have more than one property values assigned.

The *willChangeValueForKey:* and the *didChangeValueForKey:* methods are not mandatory to be used as shown above. They can be implemented in the *Children* class (or in your own class) as well. In order to see such an example, we will override the *setter* method of the *name* property. Before doing so, just comment our or remove these lines in the *viewWillAppear:* method so we won't face any problem later on:

```
[self.child1 willChangeValueForKey:@"name"];
self.child1.name = @"Michael";
[self.child1 didChangeValueForKey:@"name"];
```

Now, open the *Children.m* file, and add the following implementation.

```
-(void)setName:(NSString *)name{
    [self willChangeValueForKey:@"name"];
    _name = name;
    [self didChangeValueForKey:@"name"];
}
```

Back to the *ViewController.m* now, add the next one as the last command to the *viewWillAppear:* method:

```
-(void)viewWillAppear:(BOOL)animated{
    ...

    self.child1.name = @"Michael";
}
```

Finally, run the app once again. What are you watching on the debugger? The messages about the *name* property of course after the rest of the notification messages!

```
The age of the FIRST child was changed.
{
    kind = 1;
    new = 20;
    old = 15;
}
```

```
The age of the SECOND child was changed.
{
    kind = 1;
    new = 45;
    old = 35;
}
```

The name of the FIRST child was changed.

```
{  
    kind = 1;  
    new = Michael;  
    old = George;  
}
```

Awesome, but most importantly, by using the *willChangeValueForKey:*, the *didChangeValueForKey:* or by overriding the setter method of the *name* property, we managed to set the new value directly to it as we would normally do, and to still have the KVO working! Pretty useful, right?

Working With Arrays

Arrays is a special case in Key-Value Coding and Key-Value Observing, as there are extra actions that should be performed before you successfully manage to observe for changes made on them. Actually, there are a lot of details regarding the arrays, but here we'll see just the basics and most important things you need to know so you can work with them.

The first thing that you should know is that arrays are not KVC compliant, therefore dealing with them is not as simple as with single properties. There are certain methods regarding the arrays that should be implemented in order to make them KVC compliant, and ultimately to be able to observe any changes on them. So, let's talk about them.

For the sake of the tutorial we will discuss about mutable arrays. However, immutable arrays are handled the same way, except for the fact that fewer methods are needed to be implemented. So, let's suppose that we have a mutable array named *myArray*. The methods that should be implemented are similar to those we use for inserting, removing, and counting objects usually to an array, but with some

differences on their naming (you will get the meaning of that pretty soon). A minimum number of methods is required to be written, so let's see them:

- *countOfMyArray*
- *objectInMyArrayAtIndex:*
- *insertObject:inMyArrayAtIndex:*
- *removeObjectFromMyArrayAtIndex:*

All of those methods look familiar, don't they? The new here is that in their names we append the name of our array. Notice that even though the array is named *myArray*, when it is used in the methods the first letter becomes capital (*MyArray*). It's obvious that in a real implementation, the *MyArray* name is replaced by the name of the actual array. Beyond that, there are a few more methods that could be implemented in order to increase performance and flexibility, but here we will stay on them. Of course, when talking about immutable arrays, the last two methods should not be implemented.

There are both good and bad news when making an array KVC compliant. The good news is that Xcode suggests you each method's name once you start typing, so you don't have necessarily to remember them. The bad news however is that you should implement all those methods for every single array you have in your class and you want to observe. Don't worry though, as I am going to show you a technique at the next section that could get you out of trouble very easily.

Now, it's time to go to practice, and to see how everything we discussed so far is actually applied in code. For our purposes, we will add a new mutable array to the *Children* class that is supposed to contain the names of the siblings of a child. Go to the *Children.h* file, and add the next declaration:

```

@interface Children : NSObject

...

@property (nonatomic, strong) NSMutableArray *siblings;

@end

```

Next, open the *Children.m* file, and in the *init* method initialize the array:

```

- (instancetype)init
{
    self = [super init];
    if (self) {
        ...

        self.siblings = [[NSMutableArray alloc] init];
    }
    return self;
}

```

Now we are ready to focus on the methods that should be implemented. Go back to the *Children.h* file, and add the following method declarations. I would recommend to type them instead of copy-paste them, in order for you to see how Xcode suggests them to you.

```

@interface Children : NSObject

...

-(NSUInteger)countOfSiblings;

-(id)objectInSiblingsAtIndex:(NSUInteger)index;

...

```

```
-(void)insertObject:(NSString *)object inSiblingsAtIndex:(NSUInteger)
index;

@end
```

Notice how the *siblings* array name is used in the above declarations, and that the first letter is always capital. There is one comment that I would like to do here, and that is that for the *insertObject:inSiblingsAtIndex:* method, Xcode does not set the type of the objects to the parameters automatically, but it let's you type them according to the kind of data you are going to have in your array. In other words, if you start typing it Xcode will suggest you this:

```
-(void)insertObject:(<object-type> *)object inSiblingsAtIndex:(
```

...and you manually have to go and set the proper types of the parameters. As I said, in our case we are going to add the names of the siblings, so we will set the *NSString* type. Alternatively, in the first parameter you can specify the *id* keyword (without the asterisk), which is suitable for any type of data you may have.

Let's implement them now. Open the *Children.m* file, and add the next code segment. As you will see, what it is written is pretty simple and straightforward.

```
-(NSUInteger)countOfSiblings{
    return self.siblings.count;
}

-(id)objectInSiblingsAtIndex:(NSUInteger)index{
    return [self.siblings objectAtIndex:index];
}
```



```

-(void)insertObject:(NSString *)object inSiblingsAtIndex:(NSUInteger)
    [self.siblings insertObject:object atIndex:index];
}

-(void)removeObjectFromSiblingsAtIndex:(NSUInteger)index{
    [self.siblings removeObjectAtIndex:index];
}

```

That's it! Now the *siblings* array is KVC compliant, so we can track down any changes that happen to it as we normally do with every other property. Of course, we will add and remove data using only the above methods.

Open the *ViewController.m* file, and go to the *viewWillAppear:* method. The first step in KVO is to always observe for the desired property, so add the next line at the end of the method (we will observe only for changes to the *siblings* array of the *child1* object):

```

-(void)viewWillAppear:(BOOL)animated{
    ...

    [self.child1 addObserver:self forKeyPath:@"siblings" options:
}

```

Now, let's add some objects to the array, and then remove one of them:

```

-(void)viewWillAppear:(BOOL)animated{
    ...

    [self.child1 insertObject:@"Alex" inSiblingsAtIndex:0];
    [self.child1 insertObject:@"Bob" inSiblingsAtIndex:1];
    [self.child1 insertObject:@"Mary" inSiblingsAtIndex:2];
}

```

```
        [self.child1 removeObjectFromSiblingsAtIndex:1];  
    }  
}
```

Perfect! Not only we implemented the methods needed to make our array KVC compliant, we also used them to add and remove objects. There's one step remaining, and that is to handle the received notification to the *observeValueForKeyPath:object:change:context:* method. Here's the additional code needed to display on the console the changes that arrive with each notification:

```
-(void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object  
change:(NSDictionary *)change context:(void *)context {  
    ...  
    else{  
        if ([keyPath isEqualToString:@"siblings"]) {  
            NSLog(@"%@", change);  
        }  
    }  
}
```

Time to test it... Here are the results displayed on the console:

```
{  
    indexes = "<NSIndexSet: 0x8b92910>[number of indexes: 1 (in  
kind = 2;  
new = (Alex  
);  
}  
  
{  
    indexes = "<NSIndexSet: 0x8e29ac0>[number of indexes: 1 (in  
kind = 2;  
new = (Bob  
);  
}
```

```

    );
}

{
    indexes = "<NSMutableIndexSet: 0x8b92910>[number of indexes: 1 (in
    kind = 2;
    new = (
        Mary
    );
}

{
    indexes = "<NSMutableIndexSet: 0x8e29ac0>[number of indexes: 1 (in
    kind = 3;
    old = (
        Bob
    );
}

```

The results are in accordance to our actions. We performed three insertions and one deletion, and that's what exactly the above messages describe. In the first three cases, there is no previous value to display, so only the new one is shown. In the last case, it's shown only the old value, while there's not a new one as we just remove an object.

A General Approach on Arrays

Applying KVC and KVO to arrays is not that difficult ultimately according to what we have seen to the previous section, however it requires some extra coding, and if you have several arrays that you need to observe, then the amount of code writing gets increased. Obviously, it would be great if we could minimize that effort, and at the same time to be able to KVO on arrays. Well, actually there is a simple but nice technique. What I will show you in this section is a method that I've been using in my projects repeatedly for a long time, and after some research on the web I found out that similar techniques are recommended by others too. So, let's go for it.

The big idea on the approach that I will present here, is to create and use an extra class that contains just a mutable array and it implements the necessary methods for making it KVC compliant. Once it's ready, objects of this class can be used to any other classes instead of arrays. The benefits from all that are multiple:

- The necessary methods we implemented in the previous section don't have to be written more than once.
- You can use objects of that class everywhere instead of normal mutable arrays.
- Most importantly, it is reusable, meaning that you implement it once and you use it to many projects!

In other words, think of it as a mechanism that provides you an advanced version of the mutable arrays. Let's work on it.

Begin by adding a new class to the project. Go to the **File > New > File...** menu, and in the guide that appears select the **Objective-C class** option as the template. To the next step, make sure that the **Subclass of** field contains the **NSObject** value, and set the **KVCMutableArray** as the name of the class to the **Class** field. Get finished with the guide and wait until the new files are added to the project.

Go to the *KVCMutableArray.h* file, and add the next array declaration:

```
@interface KVCMutableArray : NSObject

@property (nonatomic, strong) NSMutableArray *array;

@end
```

Now, let's create an *init* method, where we will initialize the array. To do so, open the *KVCMutableArray.m* file and add the next code segment:

```
- (instancetype)init
{
    self = [super init];
    if (self) {
        self.array = [[NSMutableArray alloc] init];
    }
    return self;
}
```

Great! This array is going to be the *container* for all of our data, so its role is very important. Now we have it declared and initialized, we can move to the methods that will make it KVC compliant. Firstly, go to the *KVCMutableArray.h* file and do the proper declarations:

```
@interface KVCMutableArray : NSObject

...

-(NSUInteger)countOfArray;

-(id)objectInArrayAtIndex:(NSUInteger)index;

-(void)insertObject:(id)object inArrayAtIndex:(NSUInteger)index;

-(void)removeObjectFromArrayAtIndex:(NSUInteger)index;

-(void)replaceObjectInArrayAtIndex:(NSUInteger)index withObject:

@end
```

Notice that I added a new method, the *replaceObjectInArrayAtIndex:withObject:*, so

our class is even more powerful. Besides that, our approach is more general here, so in the *insertObject:inArrayAtIndex:* method we specified the *id* instead of a specific data type or class. Once again, notice how the name of the array is used to the methods.

Now, open the *KVCMutableArray.m* file, and implement them:

```
-(NSUInteger)countOfArray{
    return self.array.count;
}

-(id)objectInArrayAtIndex:(NSUInteger)index{
    return [self.array objectAtIndex:index];
}

-(void)insertObject:(id)object inArrayAtIndex:(NSUInteger)index{
    [self.array insertObject:object atIndex:index];
}

-(void)removeObjectFromArrayAtIndex:(NSUInteger)index{
    [self.array removeObjectAtIndex:index];
}

-(void)replaceObjectInArrayAtIndex:(NSUInteger)index withObject:(id)object{
    [self.array replaceObjectAtIndex:index withObject:object];
}
```

All implementations are very simple, so our work here is over. The new class is ready!

Let's assume now that in the *Children* class we want to add and observe a new array, which will contain the names of the cousins of a child. It's obvious that instead of a normal mutable array, we will use an object of the *KVCMutableArray* class. Without

wasting more time, let's open the *Children.h* file, and let's declare a new property:

```
@interface Children : NSObject

...

@property (nonatomic, strong) NSMutableArray *cousins;

@end
```

Xcode will become grumpy, so go at the top of the file and import this:

```
#import "NSMutableArray.h"
```

We are fine here, so open the *Children.m* file and initialize the new object:

```
- (instancetype)init
{
    self = [super init];
    if (self) {
        ...

        self.cousins = [[NSMutableArray alloc] init];
    }
    return self;
}
```

And finally, the big step. It's time to try it, so open the *ViewController.m* file and at the end of the *viewWillAppear:* method add the next lines:

```
-(void)viewWillAppear:(BOOL)animated{
```

```

...

[self.child1 addObserver:self forKeyPath:@"cousins.array"
                    [self.child1.cousins insertObject:@"Antony" inArrayAtIndex:1
                    [self.child1.cousins insertObject:@"Julia" inArrayAtIndex:1
                    [self.child1.cousins replaceObjectInArrayAtIndex:0 withObject:]
}

```

Firstly, notice the keypath value we use. Remember that what we want to observe is the array of the *KVCMutableArray* class, and not the object itself, so we use the dot syntax to specify that. Secondly, to insert new objects we call the custom KVC compliant method we implemented, and lastly we make use of the new method to replace the first object.

The final step:

```

-(void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)
...
else{
    ...

    if ([keyPath isEqualToString:@"cousins.array"]) {
        NSLog(@"%@", change);
    }
}
}
}

```

Ready to test it? If we run it, here are the results we get on the console:

```

{
    indexes = "<NSIndexSet: 0x8ba2e60>[number of indexes: 1 (in
    kind = 2;

```



```

        new = (
            Antony
        );
    }

    {
        indexes = "<NSSet: 0x8ba2e60>[number of indexes: 1 (in
        kind = 2;
        new = (
            Julia
        );
    }

    {
        indexes = "<NSSet: 0x8ba2e60>[number of indexes: 1 (in
        kind = 4;
        new = (
            Ben
        );
        old = (
            Antony
        );
    }

```

Excellent! It is what exactly we have been wanted to see. Both new objects are inserted to the array, and the replacement successfully takes place at the end.

So, it seems that the new class we implemented here is a new great tool, which you can wrap up and get with you when you are done with the tutorial. You see how really easy it is to observe for changes on arrays, and all that with almost no effort at all!

Summary

Both Key-Value Coding and Key-Value Observing are mechanisms that allow you to create more powerful, more flexible and more efficient applications. These

mechanisms are not adopted by a great number of developers, and that's a pity, because even though they look weird at the beginning, at the end it's proved to be very easy to be handled. In this tutorial I made an effort to provide you with a detailed introduction to both the KVC and KVO concepts. What you have learnt here is good enough to let you work with them in your projects, but there is more advanced information you could look up if you want so. It would be impossible to cover everything into one tutorial, but the most you need is here. So, having this post as a guide, I hope you'll manage to use the demonstrated techniques to your projects too, and of course, you can add the last class we created in the previous section to your programming toolbox. Happy Key-Value programming!

For your reference, here is the [complete Xcode project](#) for your download.

IOS PROGRAMMING

KVC

KVO

OBJECTIVE C

SHARE:



@gabtheodor

Gabriel Theodoropoulos

Gabriel has been a software developer for about two decades. He has long experience in developing software solutions for various platforms in many programming languages. Since middle-2010 he has been developing almost exclusively for iOS. Tutorials consist of the best way to share knowledge with people all over the world. Follow Gabriel at Google+ and Twitter.

TUTORIAL

Design Patterns in Swift #2: Observer and Memento

TUTORIAL

The Absolute Guide to Networking in Swift with Alamofire

Intro to Alamofire

IOS

Building an RSS Reader Using UISplitViewController and UIPopoverViewController

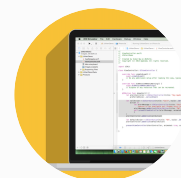


PREVIOUS POST

Understanding XML and JSON Parsing in iOS Programming

NEXT POST

Introduction to UIAlertController, Swift Closures and Enumeration



AppCoda is one of the leading iOS programming communities. Our aim is to teach everyone how to build apps with high quality and easy-to-read tutorials. Learn by doing is the heart of our learning materials.

MEET APPCODA

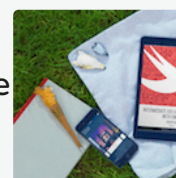
About
Our Team
Write for Us
Advertise

OUR BOOKS



Beginning iOS 12 Programming with Swift

Written for beginners without any programming experience. Supports Xcode 10, Swift 4 and iOS 12.



Intermediate iOS 12 Programming with Swift

Written for developers with some iOS programming experience. The book uses a problem-solution approach to discuss the APIs and frameworks of iOS SDK.

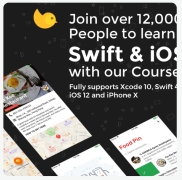
OUR PRODUCTS



RSS App Template

Save you thousands of dollars. Simply plug your own RSS feeds and turn the Xcode template into a RSS reader or a Blog reader app.

OUR
COURSE



Beginning iOS 12 Programming with Swift

Learn how to code in Swift and build a real world app from scratch. Now supports Xcode 10, Swift 4 and iOS 12.

Copyright © AppCoda. 2019 • All rights reserved.

[Terms of Service](#) | [Privacy Policy](#) | [RSS Feed](#) | [Contact Us](#)



TWITTER



FACEBOOK



GITHUB