
THObserversAndBinders

© 2012 James Montgomerie

jamie@montgomerie.net, <http://www.blog.montgomerie.net/>

jamie@th.ingsmadeoutofotherthin.gs, <http://th.ingsmadeoutofotherthin.gs/>

What it is

- Easy, lightweight, object-based key-value observing (KVO).
- Very lightweight object-based key-value binding (KVB).
- For iOS and Mac OS X, with ARC.
- Feels comfortable.
- Here are some examples.

Why it is

To me, Cocoa KVO has three problems (well, besides the conceptual arguments about whether KVO's a good idea in the first place):

- It makes your code messy. `-(void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object change:(NSDictionary *)change context:(void *)context` methods with huge flowing if statements in them. Need I say more?
- Lifetime management is hard to think about and therefore fragile.
- Encapsulating the above, it just doesn't 'feel comfortable'

Now, it seems like every Cocoa programmer out there has their own KVO and KVB solution, and I've tried a few of them. There are too many to enumerate here. Many of them are quite nice, but I couldn't find any that passed my personal 'feels comfortable' test with flying colours (and, though I know it's irrational, KVO just seems like it should be cleaner than a messily prefixed category on `NSObject`, `objc_setAssociatedObject()` or method swizzling).

How it works

- Observers are represented by simple, lightweight `THObserver` objects that are constructed with an object to observe, a keypath to observe on the object, and a block or target-action pair to call when the observed value changes.
- Optionally, you can also pass arbitrary Cocoa KVO options.

-
- The block or action can, again optionally, be passed the old and new value, or a whole Cocoa KVO change dictionary.
 - To keep code clean, there's also an option to use "value action" target-action callbacks that don't get passed the observed object and keypath like regular actions, but instead just get passed the new, or old and new, values.
 - The observation's lifetime is entirely managed by the `THObserver` object. Keep it around, the observation is alive. Release it, and the observations stop. You can also stop them manually by calling `-stopObserving`.
 - The observed object and the target are weakly referenced by the `THObserver`.
 - Nothing is going to blow up if the target object is released before the observer
 - You *must*, however, still ensure that the `THObserver` object is deallocated, or that `-stopObserving` has been called on it, before the observed object is deallocated (i.e. *before* its `dealloc` method, if you have implemented one, is called). If you don't ensure this, the weak reference held by the observer gets zeroed out and it has no chance to stop KVO-observing it. This makes Cocoa KVO very upset.
 - ★ If your `THObservers` are strongly held in instance variables of a parent object that also strongly holds the *observed* object in an ivar, and you're expecting them both be implicitly released when the parent object deallocates, remember to explicitly call `-stopObserving` on the observers (or perhaps just set them to `nil`) in the parent object's `-dealloc`. This will ensure that they're guaranteed to stop observing before the observed object is released.
 - ★ If you're observing `self` with a `THObserver` strongly held in an instance variable you can also call `-stopObserving` inside `dealloc`, as when observing ivar objects. Just relying on ARC to release the observer won't work here either because by the time an object's ARC-managed ivars are released the object that contained them is already 'gone', so it's too late to remove the observation.

I like this API. It's one simple call to set up a block that fires when a property changes. Want to observe a whole bunch of things? Just set up a bunch of `THObservers`, store them in an array, then when it comes time to stop observing, release the array (maybe calling `-stopObserving` on the observers in the array first if there might be a reference to them lying around elsewhere, like in an autorelease pool).

Results

- Code is no longer messy. Observer functionality is easy to set up and tear down, and the observation itself is neatly encapsulated in clean blocks or action methods.

-
- Observation lifetime management is really easy. The observation is just an object, and managing the lifetime of objects is intuitive.
 - It feels nice and looks clean in use. No messy prefixed methods etc.
 - Okay, I'll admit there is a little bit of monkeying with analysis of selectors and casting of blocks in `THObserver`'s implementation, but it's nicely encapsulated, and the code is reasonably straightforward.

It seemed like it would be pretty easy to write a straightforward binding mechanism with a similar API on top of `THObserver`, so I did. The `THBinder` object represents a binding, and is easy to construct and manage (see the binding examples). You can optionally supply an `NSValueTransformer` or a block to run the value through. Lifetime is managed similarly to `THObserver` - it'll stop binding when it's released, and there's also a `-stopBinding` method.

How to use it

I've packaged this as a static library, you should be able to use it as detailed in this blog post. It's only a couple of files though, so I won't tell anyone if you just copy them into your project instead.

Examples

Block-based observation:

Simple observation block:

```
1
2 THObserver *observer = [THObserver observerForObject:object keyPath:@"
    propertyToObserve" block:^(
3     NSLog(@"propertyToObserve changed, is now %@", object.
        propertyToObserve);
4 }];
```

Observation block with the old and new value passed in:

```
1
2 THObserver *observer = [THObserver observerForObject:object keyPath:@"
    propertyToObserve" oldAndNewBlock:^(id oldValue, id newValue) {
3     NSLog(@"propertyToObserve changed, was %@, is now %@", oldValue,
        newValue);
4 }];
```

Observation block with custom observation options and a Cocoa change dictionary:

```
1
2 THObserver *observer = [THObserver observerForObject:object
```

```

3         keyPath:@"
4             propertyToObserve"
5         options:
6             NSKeyValueObservingOptionInitial
7             |
8             NSKeyValueObservingOptionNew
9
10        changeBlock:^(NSDictionary *
11            change) {
12            NSLog(@"propertyToObserve
13                is %@", change[
14                    NSKeyValueChangeNewKey
15                ]);
16        }];

```

Target-action Based Observation

Any of the calls below could be made with or without an 'options' argument.

Simple target-action:

```

1
2 THObserver *observer = [THObserver observerForObject:object
3     keyPath:@"
4         propertyToObserve"
5     target:self
6     action:@selector(
7         targetActionCallback)
8 ];

```

Target-action, gets passed observed object

```

1
2 THObserver *observer = [THObserver observerForObject:object
3     keyPath:@"
4         propertyToObserve"
5     target:self
6     action:@selector(
7         targetActionCallbackForObject
8         :)]];

```

Target-action, gets passed observed object and keypath

```

1
2 THObserver *observer = [THObserver observerForObject:object
3     keyPath:@"
4         propertyToObserve"
5     target:self

```

```
5                                     action:@selector(  
                                     targetActionCallbackForObject  
                                     :keyPath:));
```

Target-action, gets passed observed object, keypath, old and new values

```
1  
2 THObserver *observer = [THObserver observerForObject:object  
3                           keyPath:@"  
                             propertyToObserve"  
4                           target:self  
5                           action:@selector(  
                             targetActionCallbackForObject  
                             :keyPath:oldValue:  
                             newValue:));
```

Target-action with options and change dictionary

```
1  
2 THObserver *observer = [THObserver observerForObject:object  
3                           keyPath:@"  
                             propertyToObserve"  
4                           options:  
                             NSKeyValueObservingOptionInitial  
                             |  
                             NSKeyValueObservingOptionNew  
5                           target:self  
6                           action:@selector(  
                             targetActionCallbackForObject  
                             :keyPath:oldValue:  
                             change:));
```

“Value action” target-action callback, gets passed the new value only: This supplies only the new value - useful in keeping code clean if you don’t need the object and keypath passed in.

```
1  
2 THObserver *observer = [THObserver observerForObject:object  
3                           keyPath:@"  
                             propertyToObserve"  
4                           options:  
                             NSKeyValueObservingOptionInitial  
                             |  
                             NSKeyValueObservingOptionNew  
5                           target:self  
6                           valueAction:@selector(  
                             targetActionCallbackForNewValue  
                             :));
```

“Value action” target-action callback, gets passed the old and new values only: This supplies only the old and new values. Again, useful in keeping code clean (see above).

```
1
2 THObserver *observer = [THObserver observerForObject:object
3                           keyPath:@"
4                               propertyToObserve"
5                           options:
6                               NSKeyValueObservingOptionInitial
                               |
                               NSKeyValueObservingOptionNew
                               target:self
                               valueAction:@selector(
                                   targetActionCallbackForOldValue
                                   :newValue:)];
```

Binding

Simple binding:

```
1
2 THBinder *binder = [THBinder binderFromObject:fromObject keyPath:@"
3   fromKey"
4                               toObject:toObject keyPath:@"toKey"
5                               ];
```

Binding with a Transformer Block:

```
1
2 THBinder *binder = [THBinder binderFromObject:fromObject keyPath:@"
3   fromKey"
4                               toObject:toObject keyPath:@"toKey"
5   transformationBlock:^(id(id value) {
6       return @([value integerValue] + 5);
7   }]);
```

Binding with NSValueTransformer:

```
1
2 THBinder *binder = [THBinder binderFromObject:fromObject keyPath:@"
3   fromKey"
4                               toObject:toObject keyPath:@"toKey"
5   valueTransformer:[MyAddFiveTransformer
6                     alloc] init];
```

This stuff seems to be making a lot of retain cycles and leaks...

I suspect you're doing something like this:

```
1
2 _observerIvar = [THObserver observerForObject:_objectIvar keyPath:@"
   propertyToObserve" block:^(
3     NSLog(@"propertyToObserve changed, is now %@", _objectIvar.
       propertyToObserve);
4 }];
```

This will create a retain cycle. The reference of `_objectIvar` inside the block will cause the block - and hence the observer - to strongly retain `self`. The observer is in turn retained by `self` when you assign it to `_observerIvar`, creating the cycle (`self` retains `_observerIvar`, which retains the block, which retains `self`).

You can instead do something like this:

```
1 MyObject *blockObject = _objectIvar;
2 _observerIvar = [THObserver observerForObject:blockObject keyPath:@"
   propertyToObserve" block:^(
3     NSLog(@"propertyToObserve changed, is now %@", blockObject.
       propertyToObserve);
4 }];
```

or:

```
1 __weak MySelf *weakSelf = self;
2 _observerIvar = [THObserver observerForObject:self.objectProperty
   keyPath:@"propertyToObserve" block:^(
3     NSLog(@"propertyToObserve changed, is now %@", weakSelf.
       objectProperty.propertyToObserve);
4 }];
```

And remember to ensure that the observer is not observing by the time that the object in `_objectIvar` is released (e.g. by calling `[_observerIvar stopObserving]` in your `dealloc`).

(Thanks to Peter Steinberger for pointing out that this could use elucidation.)