Table of Contents

Credits

Why Use Observable

Observable is a code package that lets you observe changes made to object properties. Like Apple's KVO solution you refer to properties with key path strings, your 'observing' object registers for observation with an 'observed' object, and when the observed value gets changed, you get told.

Unlike Apple's KVO, Observable:

- Is based on blocks. You get notified of changes by providing a block to call.
- Delays reporting of changes, and coalesces notifications of multiple changes that happen in quick succession.
- Does not tell your block **what** sort of change happened, or what the previous value was. This is a side effect of the previous point.
- Is clear on what happens when a key path entry is nil, or when a value in the middle of a key path changes while the path is being observed.
- Is forgiving of observers that don't clean up their observations, or that clean up their observations more times than they make them.
- Implements observable collection classes that work as you'd expect.

Observable is also designed with debugging in mind, providing several debug methods letting you see what properties are currently being observed for a particular object, and lets you set break-on-change breakpoints that act similar to watchpoints.

Plus, Observable has a subclass called LazyLoader, which makes it easy for you to implement lazily loaded properties, synthetic properties, and computed properties in Objective-C.

Observable Reference

A Note on Nomenclature

This documentation refers to Observable as the SDK package, although the core class is EBNObservable, and all of its classes are prefixed EBN. This is because this code is plucked straight from the eBay for iPhone app, and the team has coding standards. Observable and EBNObservable are more or less interchangeable terms.

Observable Objects

Observable was initially written so that only subclasses of the EBNObservable class could be observed. That has since changed, and now any object can be observed; the EBNObservable class is still in the package for compatibility with older code.

Generally speaking, an object that can be observed is an NSObject subclass that has properties. It is possible to observe readonly properties of objects, although doing so won't generate automatic observer notifications, unless the object privately has a readwrite version of the property or implements a private custom setter.

Collection classes such as NSSet, NSDictionary, and NSArray and their mutable subclasses are observable objects, in that they have properties such as a `count` property and those properties can be observed, but much more often you want to observe an item in a collection class, which doesn't work for the Foundation collection classes. Using a dictionary key as a key path of an NSDictionary object doesn't work because the dictionary key is not a @property. But, see the section on Observable Collections, a set of collection subclasses that do support this.

Just like with Apple's KVO, for automatic observation of property changes to work properly the implementor of a class needs to always use property setter semantics when setting properties. You can use "object.property = x" notation or [object setProperty:x] notation. Using object->_property = x will change the property value without notifying observers, which is usually bad form, but be sure to follow the rules about using that form in init, dealloc, and within the setter method itself.

Internally, when Observable begins observing a property for changes it creates a runtime subclass of the observed object's class, adds a custom setter method to that subclass that calls through to the original setter and also handles notifying observers, and changes the class of the observed object to be the runtime-created subclass. This is very similar to what Apple's KVO does to implement automatic change notification.

Key Paths

A key path is a string of property names separated by periods. Key paths are always relative paths, rooted from the observed object. For example, for a singleton `UserCache` object that has a `currentUser` property (of type `User`) which in turn had an `Address` property, you might have a key path rooted at the singleton that looked like this:

```
@interface UserCache : NSObject
@property User currentUser;
@end

@interface User : NSObject
```

```
@property StreetAddress address;
@end

@interface StreetAddress : NSObject
@property NSString *zipCode;
@end

NSString *keyPath = @"currentUser.Address.zipCode";
```

This is all very much like Apple's KVO. Each component of the key path must be an actual property of the preceding object. Because of this design, only the last property in a key path can be of non-object type.

There are a few special key path elements. The `*` element means, "every property of this object". It can only be the final element in a path, as a hedge against insanity. There's no technical reason we couldn't make a key path of `"*.*.*.*.*"` work, but it could generate thousands of individual observations when used for a complicated object graph and since Observable doesn't tell what it was that changed, it quickly degenerates into uselessness. Also, when we get to talking about observing collection objects, there's some special element notations for sets and arrays. We'll get to those later.

The Act of Observing

When you start observing a key path, Observable watches for changes to each property in the key path. When a property in the middle of an observed key path changes value (that is, that property's setter method gets called) Observable will stop observing the old value's object and start observing the new value. Importantly, this mechanism works if the old or new value for a property in the middle of a key path is nil. An important pattern to understand is that you can set up an observation in your object's init method, observing a property of some object and also requesting the data for that property be fetched asynchronously (network or file load, asynchronous image operation, whatever). When the fetch completes the property value goes from nil to non-nil, and your observer gets called. If the object changes again later (or a value in the observed key path changes), you'll get notified again. In this way your code that responds to the initial fetch completion is the same code that responds to changes.

In the `User` example above, if the key path "currentUser.Address.zipCode" was being observed and the currentUser's `Address` object was replaced with a new `StreetAddress` object, the observation will update automatically to observe the new `StreetAddress` object–and also the new `zipCode` property of the new `StreetAddress`.

In code:

```
StreetAddress *homeAddress, *workAddress;

// homeAddress is being observed
currentUser.address = homeAddress;

// now workAddress is being observed, and homeAddress isn't.
currentUser.address = workAddress;

// Change the zipCode
currentUser.address.zipCode = @"97210";
```

Now for a feature that makes Observable work significantly different than Apple's KVO. If you're observing "currentUser.address.zipCode", all 3 of those assignment statements above would cause your observation callback to be called. That is, you get notified about changes to any element in the key path, not just the last one.

But, multiple changes that happen during the processing of the same event get coalesced, so in the code above, you'd still only get called once. More on this later.

Beginning Observation

There are 3 basic ways to set up observation: The `ObserveProperty()` macro, EBNObservable's `tell:` methods, and EBNObservation's `observe:` method. The `ObserveProperty()` macro is the easiest way to set up observation. The macro can only be used to observe a single key path, and the observer object must be `self`– you can't use the macro to set up observation while making some other object the observer. But, the macro does some very useful syntax checking at compile time. The macro looks like this:

```
ObserveProperty(observedObject, keyPath, { blockContents });
```

Note that keyPath is not in quotes, and that blockContents isn't actually a block. The ObserveProperty macro tries to validate each property in the key path by ensuring that the compiler thinks observedObject.keyPath is a valid construction. It also tries to prevent you from using self or observedObject inside the block, as this can cause retain loops or other object lifetime issues.

The property validation is a useful defense against future changes that may be made to the object(s) you're observing. Code that uses the macro won't compile if the names of properties in the key path change, or the type of a property in the middle of the key path changes and the new type doesn't have the property that the path specifies.

There is also a `ObservePropertyNoPropCheck()` variant that doesn't do validation on the key path. This variant is useful for cases where the key path contains entries that aren't actually properties. Is there someone your team that insists on using `array.count` instead of `[array count]`? This variant is for them.

Next, the tell: methods. These are methods defined by EBNObservable, so when you use them, the syntax declares that you're asking an Observable object to tell you when things happen.

```
- (EBNObservation *) tell:(id) observer when:(NSString *) propertyName changes:(ObservationBlock) callBlock;
- (EBNObservation *) tell:(id) observer whenAny:(NSArray *) propertyList changes:(ObservationBlock) callBlock;
```

The first method observes a single key path, the second takes an array of paths. Both methods take an ObservationBlock as a parameter, which is defined as:

```
typedef void (^ObservationBlock)(id observingObj, id observedObj);
```

One of the conveniences of the ObserveProperty macro is that you don't need to declare your block's parameters; the macro does that for you. But, a more concise syntax means less flexibility.

On that note, the most flexible way to set up observation is to create an EBNObservation yourself. EBNObservation objects are returned by the macro and also by the tell: methods, but if you set it up yourself you can easily use the same block to observe multiple key paths, set up a custom debugging description, execute the observation block yourself, and more.

Ending Observation

When you are done observing you can call one of these macros:

```
StopObservingPath(observedObject, keyPath);
StopObserving(observedObject);
```

These macros behave similarly to the `ObserveProperty()` macro in that they try to validate the key path. Both of these macros assume '`self`' is the observing object. If that's not the case, see below. `StopObserving()` removes all observations on observedObject where `'self'` is the observer, while `StopObservingPath()` only remove ones that match the given key path.

You can also use one of these methods to end observation:

```
- (void) stopTellingAboutChanges:(id) observer;
- (void) stopAllCallsTo:(ObservationBlock) block;
- (void) stopTelling:(id) observer aboutChangesTo:(NSString *) propertyName;
- (void) stopTelling:(id) observer aboutChangesToArray:(NSArray *) propertyList;
```

The first method tells the receiver to deregister all tells to the given observer object, and the second deregisters all tells to the given block. If you want to call the second method, be sure to save the block pointer when you register.

The last two methods deregister all tells to your object that were registered for a particular property or set of properties. If for some reason you registered for the same property more than once (for example, both you and your superclass registered for the same thing), all matching registrations will be removed.

It is safe to call these methods when you're not actually observing the given object.

Observation Blocks

Your code gets informed of a change to a observed key path by providing an Observation block. Observation blocks take this form:

```
typedef void (^ObservationBlock)(id observingObj, id observedObj);
```

When Observable calls your block, it will pass back the observing and observed objects. Inside your block, you should refer to the observing and observed objects using these block parameters. If you use e.g. `self` and `modelObject` instead, you will probably run into retain loop issues. You can create a `__weak self`, but you need a `__weak modelObject` as well and, … just use the parameters. They already handle retain loop issues for you.

Unlike Apple's KVO, the block arguments do **not** tell you what the old or new value of the property is, or even which property changed. You can, of course, access the new value of the property directly. You can also set it, or set other properties, without infinite-looping.

When Observation Blocks Get Called

Another difference between this class and Apple's KVO is that notification of changes is delayed until the end of the event in which the change occurred. This is done so that we can coalesce multiple changes into a single observationBlock call when possible, following this rule:

A single ObservationBlock will get called at most once per event loop iteration.

This means that if a property changes value multiple times in a single event, the block gets called once. If the same block is used in registering for many properties (such as the array variant), the block gets called once, even if all the properties were modified during the event.

If an ObservationBlock sets another property when it runs (in the same or a different object), and that property is being observed as well, that property's observationBlock will be called only if it hasn't been called yet during this event loop iteration.

Your ObservationBlock will always be called on the main thread, even if the property is set in a background thread.

If your block is observing a key path with more than one element, your block will get called when any element in the key path changes value. If, for example, you are observing the key path `currentUser.address.zipCode` but currentUser is initially nil, your observation block will get called when the currentUser gets set to a non-nil User, even if that user doesn't have an address. In this case, the zipCode has changed from nil to nil (or more correctly, from 'not reachable' to 'not reachable', because its containing object doesn't exist) but you get called anyway. Often you may not care about these changes, but sometimes you do.

Immediate Mode Observations

Generally, having your observer blocks run at the end of the main thread's event loop is a good thing. It avoids issues where a value you are observing gets its value changed from a thread you weren't expecting. It avoids problems stemming from a value being changed thousands of times in quick succession, which in turn enables some really useful sorts of observations where you observe many properties with one observer block, and are called just once if all of the observed properties change at once.

However, sometimes you really need to know what the previous value of a property was, or need to know immediately when a property changes value, or need to run code on the same thread where the change happened. Observable does have a way to create immediate observation blocks. They look like this:

```
// Step 1
EBNObservation *observation = [[EBNObservation alloc] initForObserved:modelObject
                observer:self immedBlock:
        ^(ObservableTests *blockSelf, ModelObjectA *observed, id previousValue)
        {
                int prev = [previousValue intValue];
                // Do something with previous value here
        }];

// Step 2
[observation observe:@"intProperty"];
```

This code first creates an observation object, giving it a target object to observe and a observer block as part of its initialization. Then in the second step we tell the observation object to begin observing on the key path @"intProperty".

The observation block is a bit different in this case, as it takes an extra parameter. The previousValue will contain what value the property had before the change occurred. Since you can just get the current value from "modelObject.intProperty" there isn't a parameter for that.

And again, remember that immediate mode observations can get called from any thread. They can get called from some thread when that thread is holding a lock that you don't know about. They can get called from multiple threads at once. If the value being observed changes thousands of times in quick succession, the observation block will get called thousands of times. If two immediate mode observations change each other's observed property from inside their observer blocks, you'll infinitely recurse. Regular observation blocks protect against all of these things, so use them whenever you can.

What Observable Does While Observing

…and why it makes your choice of key paths more important than ever.

If you create an observation on a key path such as `"address.zipCode"` of the currentUser object and while the observation is active, someone sets the `address` property of the currentUser, Observable will stop observing the `zipCode` property of the previous `Address` object and start observing the `zipCode` of the new `Address` object. Additionally, if at the time you create the observation the `address` property is nil, Observable will remember to set up observation on the `address` object (and the address's `zipCode` property) when they receive a non-nil value.

But what if you want to have your observation 'follow' the old address if someone sets a new address for the currentUser, instead of switching over to the new address value? In that case, you should observe the `zipCode` key path of the `currentUser.address` object.

In code:

```
// This observation follows the new address if the address object is replaced
ObserveProperty(currentUser, address.zipCode, ...);

// This observation follows the old address if the address object is replaced
ObserveProperty(currentUser.address, zipCode, ...);
```

The ability to observe a path that may be nil at the time of observation can be very powerful, as you can in many cases set up an observation on a model object path when your object gets initialized, and then fire off a request to the model layer to fetch the object you're observing. When the model layer completes the fetch (and attaches the fetched object to a model object hierarchy) you get informed automatically.

But, careful selection of what object should be the 'root' of the observing key path and what should be parts of the key path property list is very important. The way to think about constructing a proper key path is in terms of update behavior at each point in the path.

Cleanup and Object Lifetime

You should deregister observers when the observing object deallocs, although it's not a requirement. Unlike Apple's KVO, failing to deregister won't cause an assert. Also, deregistering multiple times won't cause an assert. Deregistering an observation you never made won't cause an assert. Registering the same observation multiple times won't cause an assert.

An observation does not hold the observed or observing object strongly. A EBNObservable instance can be dealloc'ed while its properties are being observed, and an observer object can safely be dealloc'ed while it has observations in place. The only exception to this is that an EBNObservable subclass is held strongly from the time one of its observed properties is set until the observer block for that property is executed. This is to ensure the observers get called before the observed object goes away.

ObservationBlocks themselves have a lifetime that is dependent on both the lifetime of the observed and observing objects. Observable checks that both objects still exist before calling observation blocks. If an observing object is deallocated while it has blocks registered, those blocks may not be deallocated immediately–although they'll never be called again.

Observing the Observers

oblig latin phrase: Qui custodiet ipsos custodes?

You can query an Observable object at any time to see which of its properties are being observed, and how many observers there are on any property.

```
// Returns all properties currently being observed, as an array of strings.
- (NSArray *) allObservedProperties;

// Returns how many observers there are for the given property
- (NSUInteger) numberOfObservers:(NSString *) propertyName;
```

You can also implement the following method to get notified when your properties are being observed. You will only get called when the number of observers for a particular property goes from 0 to 1 or 1 to 0. Additional observers for an already-observed property do not cause this method to be called.

```
- (void) property:(NSString *) propName observationStateIs:(BOOL) isBeingObserved;
```

An enterprising engineer could use this to dynamically decide when certain parts of an object needed to be loaded in from backing store, or requested from the network, based entirely on whether anyone wants to know the value.

Dealloc Protocol

If you're observing properties of some other object, you can implement the ObservedObjectDeallocProtocol. This protocol has one method, which tells you if an object whose properties you were observing has been deallocated.

For example, suppose you have:

```
@class SubObject;

@interface BigClass
@property (strong) SubObject *subObjectThing;
@end
```

If you observe a property of subObjectThing, and someone replaces subObjectThing with a new value (that is, calls `BigClass.subObjectThing = newObjectThing;` you'll get notified and can update your observation to observe the new subObject. HOWEVER: this only works if the old subObject is deallocated after getting replaced (that is, if nobody else owns it).

Manually Triggering Observers

If you have a case where you really need to modify a property value in an Observable object, but can't call the setter method, you can use one of these methods:

```
- (void) manuallyTriggerObserversForProperty:(NSString *) propertyName previousValue:(id) prevValue;
- (void) manuallyTriggerObserversForProperty:(NSString *) propertyName previousValue:(id) prevValue newValue:(id) newValue;
```

The concept here is similar to Apple's `willChangeValueForKey:` and `didChangeValueForKey:` methods, except you only have to make one call. To ensure everything works correctly:

1. Save the previous value : `NSString *oldZip = address->_zipCode;`
2. Change the value before calling : `address->_zipCode = @"97210";`
3. Then call the manual trigger method, passing in the previous value : `[Address manuallyTriggerObserversForProperty:@"zipCode" previousValue:oldZip];`

In most sane cases you will not have to deal with calling `manuallyTriggerObserversForProperty:`. But, because Observable will dynamically keep a key path up to date, meaning that as elements in the middle of a key path change value we update what object properties are being observed, it is important that you add manual trigger calls in cases where you can't call the setter method. Much of what these methods do is keep key paths up to date.

An important caveat is that you do not have to do manual triggering for a property inside of that property's custom setter method.

Debugging

You can see who all is observing a given object by calling `debugShowAllObservers` on an observed object. In the debugger, call `po [modelObject debugShowAllObservers]` to see who's observing the modelObject's properties.

For example, while stopped in the lldb debugger:

```
        (lldb) po [currentUser debugShowAllObservers]

        <User: 0x109311660>
            address notifies:
               0x109311960: for <AddressLabelCell: 0x10fe0f100> declared at AddressLabelCell.m:879
```

This shows that the `address` property of the `currentUser` object is being observed by an `AddressLabelCell` object.

The other neat debugging trick is that you can use Observable to set breakpoints that act very similar to debugger watchpoints, without the extreme slowness.

Again, in the debugger:

```
        (lldb) po [CurrentUser debugBreakOnChange:@"address.zipCode"]
        Will break in debugger when address.zipCode changes.

...(continue execution)...

        debugBreakOnChange breakpoint on keyPath: address.zipCode
                debugString: Set from debugger.
                prevValue: nil
                newValue: 97210
        (lldb)
```

What this does is create an observation whose observer block will break into the debugger when it gets called. This is very similar to setting a breakpoint on the setter method for a property, with a breakpoint condition that checks the self pointer against a specific value. However, `debugBreakOnChange:` will actually follow changes to the key path, meaning that if the currentUser's address object is changed, the breakpoint will get hit when the zipCode property of the new address object is modified.

Minutiae

- Setting a property to the same value it currently has will not cause observers to fire. For properties of object type, uses pointer comparison and not `isEqual:` to determine equality.
- Observable is designed to work with ARC exclusively, and it requires iOS 6.
- Observable tries to keep asserts to a minimum, but it has several asserts for programmer errors and places that may need improvement. One such place is that we only currently work with a few types of struct properties. Every struct property type needs to be special cased.
- Observable can interoperate with Apple's KVO. Both systems can observe the same property of the same object without interfering with each other.

Observable Collections Reference

Observable has subclasses for `NSMutableArray`, `NSMutableSet` and `NSMutableDictionary` that can be observed and their contents can be observed just like any Observable subclass.

Specifically:

```
        EBNObservableDictionary *wordDictionary = [[EBNObservableDictionary alloc] init];
        [wordDictionary tell:self when:@"steampunk" changes:
                ^(WordOfTheDay *blockSelf, EBNObservableDictionary *observed)]
                {
                        // This gets called when the key "steampunk" is added, changed, or removed

                        // Get the object of the key normally
                        WordDefinition *def = [observed objectForKey:@"steampunk"];
                });
```

If wordDictionary were a property of some other object, you could also observe the path `@"otherObject.wordDictionary.steampunk"`. Similarly, you can make a key path that goes through the object in the dictionary, such as `@"otherObject.wordDictionary.steampunk.pronunciationGuide"`.

Common Features

All 3 collection classes have `count` as a declared property. This means that you can observe the count of members of a collection, just as with any other Observable object.

As with all other observable objects you can observe the key '*' and be informed whenever the collection is mutated. The collection objects treat '*' in a special way, and don't place observations on the actual properties of the collection objects (such as `count` and `description`). But the semantics are the same– your observer gets called in response to any mutation. The '*' can only be the last element in the key path.

All 3 classes conform to NSSecureCoding, NSCopying, and NSMutableCopying, although they do not copy or encode observations. That is, a copied collection or a encoded/decoded collection will not carry over any of the observations that were active in the source collection object.

EBNObservableDictionary

Dictionary keys must be NSStrings in order for their contents to be observed.

You can observe a key in a dictionary when that key does not yet have any value. Observation blocks on a key will be executed when that key's value is changed, where 'changed' means: exactly one of old value and new value are nil, or `![newValue isEqual: oldValue]`. Calling `removeObjectForKey:` on an observed key will call observers if and only if the value for that key was previously non-nil.

EBNObservableSet

Creating observable sets requires that we have some way to specify a member of a set in a text string. EBNObservableSet therefore has a special method, `keyForObject:` that returns a NSString that can be used in a key path to refer to that object. This method works by taking the hash of the object and turning it into a string starting with '&'. The object does not need to be in the set at the time its key is created, but beware since mutating the object can modify its hash value, meaning that were you to generate and observe a key, mutate the object, and then add it to the set, your observer block will not execute as the object added to the set no longer matches the key being observed.

EBNObservableSet will call observers when observed objects are actually added or removed from the set. Calling `addObject:` on an object that is already in the set (or where the set already has an object that passes the isEqual: test) will not execute observers; neither will calling `removeObject:` on an object not in the set.

There is also an `objectForKey:` method, letting you find the object in the set that matches a generated key.

Really, the most common use case is to observe '*' on the set and get notified whenever the set is mutated.

EBNObservableArray

Observable arrays have two different ways to refer to their elements in a key path: Object-Following and Index-Following.

Object-Following Observations

First is the "object-following' style, which looks like ""array.4". In this style, the key for the element in the array you want to observe is indicated with the current index of that element. For this observation style, the observation follows that object in the array. If elements before the observed element are added or removed, the observation updates so that it will continue to observe the initial array element. This means that the observed element's current position in the array may no longer match its key path. With this style of observation, the observer block is only executed when the observed object is removed from the array. And, of course, the object you wish to observe must be in the array before you begin observation.

Attempting to observe an array with an object-following observation where the index to observe is beyond the end of the array is an error.

Object-following observations will remove themselves after they call the observer block. Note that this happens even if the array element isn't the endpoint of the key path. Since the observer block is only called when the observed object leaves the array, and the object in the array that was being observed was only referenced by index at the time at the time the observation was set up, there is no way to re-connect the path in this case, even if the same object were re-added to the array.

Index-Following Observations

The other element reference style precedes the element index with a '#', and the observation thus created will follow the index, not the object. If elements are added or removed at previous positions in the array, the observation will update itself so that it is always observing the element at the originally given index. If the size of the array shrinks so that the array no longer has an element at the given index, it is treated as if the value has become nil. For this observation style, observer blocks are executed whenever the object at the observed index changes–this includes the case where `replaceObjectAtIndex:` is called for the observed index.

It is allowed to create index-following observations on indexes that are beyond the current end of the array. In fact, it's fairly common practice to initialize an empty array and immediately observe the first 7–12 indexes, before anything is placed in the array.

Minutiae

The collection classes work by declaring themselves to be subclasses of their NSMutable* classes, while also using composition to actually implement their collection behavior with an instance variable of the same NSMutable* type.

The reason for the odd subclass-plus-private-implementation thing is because the NS collection classes are class clusters, where the top-level object just defines the protocol and hidden subclasses actually implement the behavior. Therefore, our collection objects need to subclass Apple's in order to interoperate as collections, but they can't call super to implement their behaviors as the superclass doesn't implement them.

Copying an observable collection object does not copy the observations. However, mutableCopy will return an observable collection object (an EBNObservable), not an NSMutable.

EBNObservable collections are NSCoded using an archiver proxy object that deals with strange issues with how class clusters work with NSCoder.

LazyLoader Reference

LazyLoader is a class that leverages Observable to introduce a few new capabilities appropriate for model objects. LazyLoader allows for the creation of several types of *synthetic properties*, which in this context means a property whose value is synthesized from other property values.

Using LazyLoader used to require that you declare your object to be a subclass of LazyLoader, but this is no longer the case–LazyLoader is now a category on NSObject.

LazyLoader synthetic properties are **properties**, just like normal Objective-C properties. You can declare them strong, weak, assign, copy, atomic or not, readonly or readwrite. You can declare custom setters and getters or not.

If these concepts sound somewhat similar to Apple's new property attributes in the Swift language, as it happens, they are. I'd written the code for this before hearing about Swift, and was very happy to see these sort of features in the Swift language (remember, Apple had been working on Swift for years, much longer than I've been working on LazyLoader). Lots of other people have come up with similar ideas as well; you can find examples on the internet.

Lazily Loaded Properties

The most basic type of property supported is a *lazily loaded* property, which is a property that only computes its value when its value is requested. Lazily loaded properties compute their value in the getter method when the getter is called, cache the computed value in an ivar, and use an invalidation method to force a re-computation of their value. Although lazily loaded properties are usually somewhat CPU-intensive to compute, they need to perform their computation synchronously–you shouldn't have lazily loaded properties initiate network requests to fetch their value.

Lazily Loaded properties have a concept of a valid/invalid state. When the property is in the valid state, LazyLoader has cached the current value of the property in its instance variable, and will return this value when the getter is called without calling the getter method. When the property is in the invalid state, the value in the ivar does not reflect the correct value, and LazyLoader will cause the proper value to be computed the next time the property's value is requested. LazyLoader tracks valid/invalid states separately from the property itself; a pointer property can have a value of `NULL` while in the valid state.

All lazily loaded properties need to be declared in the object's code, generally in the init method. You can use either the `syntheticProperty:` method or the `SyntheticProperty()` macro. Just like Observable, the major difference is that the macro performs compile-time checks to ensure the property is actually a property of the class. Here's the code, for an example property named `fullName`:

```
@property (strong) NSString *fullName;

...

// Method
[self syntheticProperty:@"fullName"];

// Macro
SyntheticProperty(fullName);
```

These calls will replace the getter method for the given property with a wrapper method that checks to see if the property is in a valid state, calls through to the original getter method if the property state isn't valid, and returns a cached value if the property state is valid.

If the program state changes such that a Lazily Loaded property's value is no longer correct, you can reflect this by invalidating the value. Invalidating marks the property invalid, meaning the value will get recomputed later. Again, there's a method and a macro:

```
// Method
[self invalidatePropertyValue:@"fullName"];

// Macro
InvalidatePropertyValue(fullName);
```

There's also a convenience method to invalidate all the synthetic properties of an object at once:

```
[self invalidateAllSyntheticProperties];
```

So far this code makes for a slight convenience, but isn't terribly useful. Hand-rolling lazily loaded properties isn't that difficult. However, this does make it easy to test your synthetic properties with and without lazy loading, simply by commenting out the `SyntheticProperty` call for that property. Without the `SyntheticProperty` call, your synthetic property will compute its value every time its getter is called.

Synthetic Properties

A Synthetic property has a list of key paths that the property depends on; that is, computing the value for the computed property uses data from the properties in the key paths. Synthetic properties are like lazily loaded properties in that they don't compute their value until someone asks for it, but synthetic properties manage their own invalidation–they observe their list of key paths and mark themselves invalidated when any of the values they depend on change.

To declare a property to be a synthetic property, use code like the following, again in both method and macro form:

```
// Method
[self syntheticProperty:@"fullName" dependsOnPaths:@[@"firstName", @lastName]];

// Macro
SyntheticProperty(fullName, firstName, lastName);
```

The `SyntheticProperty()` macro takes a variable number of arguments; the first argument is the property to be made synthetic, and the rest of the arguments are key paths (rooted at `self`) that the synthetic property uses to compute its value. If the value of the synthetic property can be completely determined by its dependent properties, there's no need to manually invalidate it–it'll automatically invalidate when any of its dependencies change.

You still can manually invalidate a synthetic property; use this if the property value is dependent on both other properties and some non-property value (in this example case, that could be a non-property setting that determines whether fullName should be formatted "First Last" or "Last, First").

Internally, LazyLoader uses Observable to watch the dependent paths; this means that it performs path updating as values in the path change, and that it's okay to declare a dependent path before the start point of the path has been assigned a value. Again, LazyLoader currently requires that all dependent paths be rooted at `self`, although the only reason for this is simplicity.

Collection Properties and Mutability: a Case Study

If you have a mutable collection in your class and you want it exposed publicly so that others can access it but definitely do not want others to mutate it, you can create a private mutable collection and a public immutable collection, and write a custom getter that copies the internal collection to the external one.

```
// In the .h file
@property NSDictionary *aDictionary;

// In the .m file
@property (strong) NSMutableDictionary *aMutableDictionary;
...
- (NSDictionary) aDictionary
{
        return [aMutableDictionary copy];
}
```

However, the public property isn't observable. Well, you can observe it, but your observation won't get called when you want it to. It also creates a new copy of the dictionary every time the property is accessed. Copy-on-access in this fashion is generally not a performance nor a memory usage problem–but still. Using LazyLoader we can easily improve on this:

```
// In the .m file: change our mutable property's type to EBNObservableDictionary
@property (strong) EBNObservableDictionary *aMutableDictionary;

// And, somewhere in the init method add this line
SyntheticProperty(aDictionary, aMutableDictionary.*);
```

And you're done. The aDictionary property will now be copied at most once per mutation performed on aMutableDictionary, and the public property is now properly observable–that is, observers of aDictionary will get called in response to aMutableDictionary being changed.

What that SyntheticProperty declaration does is make it so that every time anything in aMutableDictionary changes, invalidate the cached NSDictionary kept in the ivar and re-calculate it the next time aDictionary is requested. If aDictionary is being observed, that re-calculation will happen immediately.

Custom Loaders

Classes that manage a set of values that are stored in a dictionary or dictionary-like-object but expose a bunch of properties as the API for retrieving the dictionary values are really very common. LazyLoader makes the creation of these sorts of classes easier by including support for custom loader methods.

When you declare a synthetic you can specify a selector to be a loader for that property. The loader method gets called when LazyLoader would otherwise call your getter method, but the loader method takes the name of the property that needs to be loaded:

```
- (void) loader:(NSString *) propertyName
```

In this way, you can declare a single loader method for all your classes' synthetic properties, and that method's job is to set the ivar backing the given property to the correct value, usually by calling `setValue:forKey:`. This is much easier than having to write custom getters for each property or doing dynamic message resolution trickery.

Hybrid Properties

Synthetic properties compute their value in their getter method, and therefore usually don't have or need a setter method. With LazyLoader however, you can have properties that usually compute their value, but can also have their value overridden via the setter. No extra setup is required for this, just make sure your property is declared `readwrite`. Calling the setter will set the value as normal, and will mark the property as being valid. The property will keep the set value until the next time it is invalidated.

It is still unwise to call a property's setter method from within its getter, as it is likely to lead to infinite recursion.

Observing Synthetic Properties

A lazily loaded property that is being observed, and isn't the last element in the observation's key path, will effectively stop being lazily loaded because Observable needs to know the new value of the property immediately when it changes, in order to keep the observation on the key path up to date. Observable may need to set up observations on properties of the newly set object (and properties on that object, and so on) which forces the lazy loaded property to compute its value immediately.

This doesn't mean that a property can't be both synthetic and observed, just that observing sometimes needs to force immediate evaluation of the lazily loaded property.

Chaining Synthetic Properties

Take the example above where we have a `fullName` property that depends on the `firstName` and `lastName` properties. What if `fullName` were itself made into a dependency of another property?

```
@property (strong) NSString *firstName;
@property (strong) NSString *lastName;
@property (strong) NSString *fullName;
@property (strong) NSString *zipCode;
@property (strong) NSString *city;
@property (strong) NSString *state;
@property (strong) NSString *street;
@property (strong) NSString *address;

...

SyntheticProperty(fullName, firstName, lastName);
SyntheticProperty(address, fullName, zipCode, city, state, street);
```

This works just fine. Changing the `firstName` property will cause the `fullName` property to be invalidated, as well as the `address` property. Manually invalidating the `fullName` property will also invalidate the `address` property.

Debugging LazyLoader Properties

In the debugger, you can query which properties of a LazyLoader object are valid and invalid like this:

```
(lldb) po [self debug_validProperties]
{(
        fullName,
)}

(lldb) po [self debug_invalidProperties]
{(
        address,
)}
```

You can also force all synthetic properties to be computed immediately (and therefore switch to the valid state) using `po [self debug_forceAllPropertiesValid]`. This makes all the instance variables that back your classes' synthetic properties cache their currently valid value, meaning you can inspect them in the debugger and see the proper values for them.

Future Work

When working on this code base, be aware that there's a fair number of unit tests implemented. Use them. This codebase is just the sort of thing that gains a lot from unit test support.

There's lots of room to improve on this code. Here's some good possibilities, in no particular order:

- Observable uses @synchronize in places where it needs thread-safety. This is very safe, as @synchronize handles unlocking in the case of exceptions being thrown, but it is quite slow compared to other available synchronization methods. As always, profile your code first, and only undertake this optimization if you find that @synchronize is actually slowing you down.
- LazyLoader is designed to make it easy to enable and disable its functionality–in most cases, just commenting out one line of code will return a property to its non-synthetic state. This is the main reason I didn't go with a model where you could declare a block that would compute the property's value–the block would work nothing like the getter method and testing a property with and without lazy loading would become very difficult. However, this means that the point in the code where you declare a property synthetic is often far away from where its value is computed. There's a couple of ways to remedy this; I'm interested in what people come up with.
- The demonstration app is currently pretty weak; I didn't even mention it in this documentation. Writing something that loads some data from a web service call, puts the data into a model object, and has views that observe the model object's properties would be ideal.
- The LazyLoader documentation concerning loader methods discusses classes that expose a property-based API but store their values in a dictionary, but the SDK doesn't currently include one of these classes.
- The current design is such that declaring an observation at the class level (probably in `+initialize`?) so that every object of that type would have that observation active could be a thing. It could have a performance advantage relative to setting up observations in `-init` if you were creating a large number of objects. In other words, write a new class method that says, "Every object of this class that gets created should have this here observation on it, right when it gets initialized."
- Better debugging: for the observations that operate on multiple key paths, having console logging that shows what change(s) caused the observer block to be scheduled would be useful.
- More unit tests, as ever.

Credits

- Chall Fry - engineering chall@challfry.com
- Ben Yarger - QE, QA, unit test cases
- Mark Yuan - Open Sourcing help