Asynchronous Code

Asynchronous Code

`URLSession`

`NSTimer`

`MKLocalSearch`

`NSOperation`

Asynchronous Code

```swift
func fetchData(_ completionHandler: @escaping (Data) -> Void)
```

Asynchronous Code

```
func fetchData(_ completionHandler: @escaping (Data) -> Void)
```

Asynchronous Code

```
fetchData(completionHandler: (Data) -> Void)
```

Asynchronous Code

```
fetchData { data in
    // ...
}
```

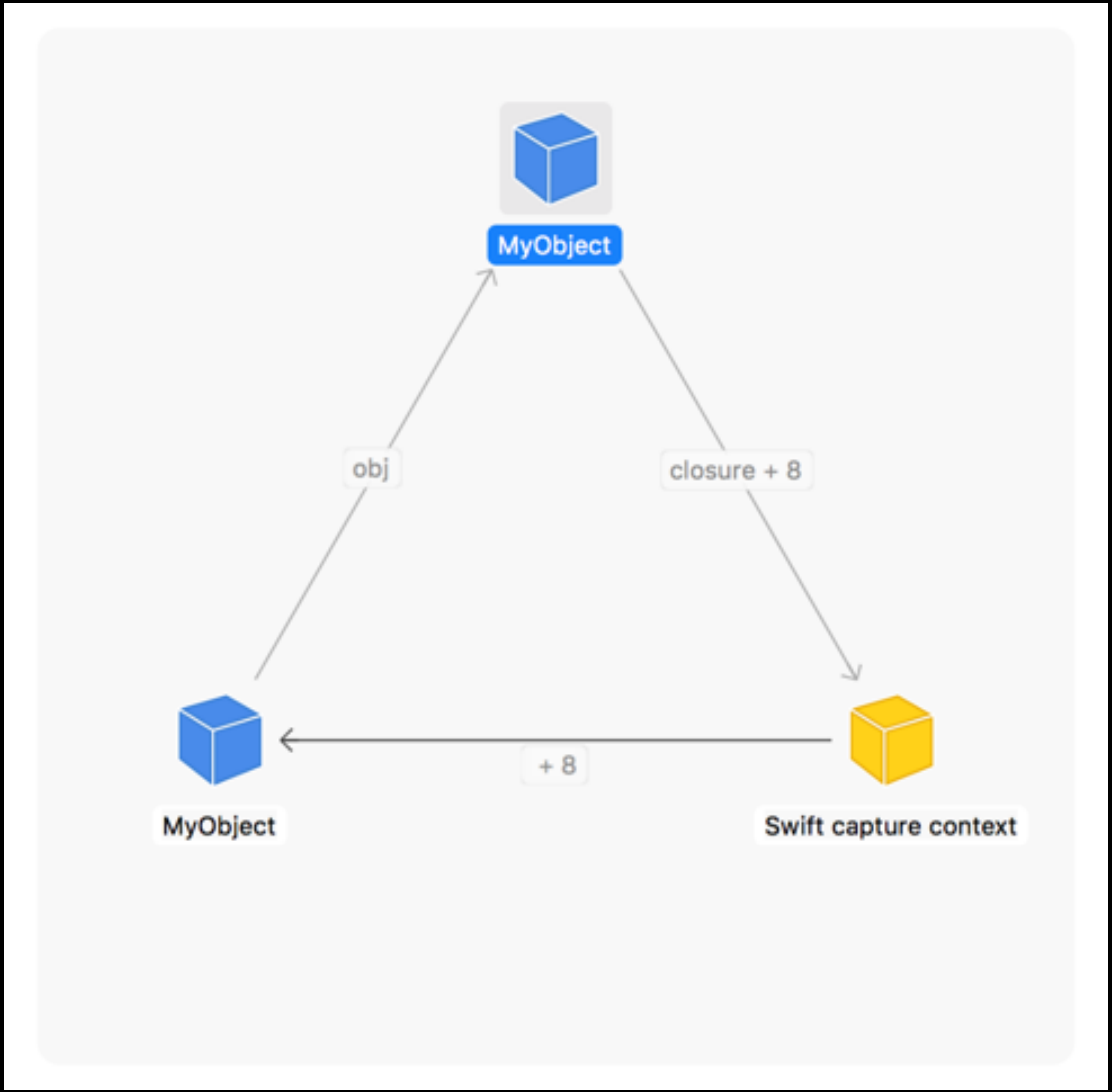
Asynchronous Code

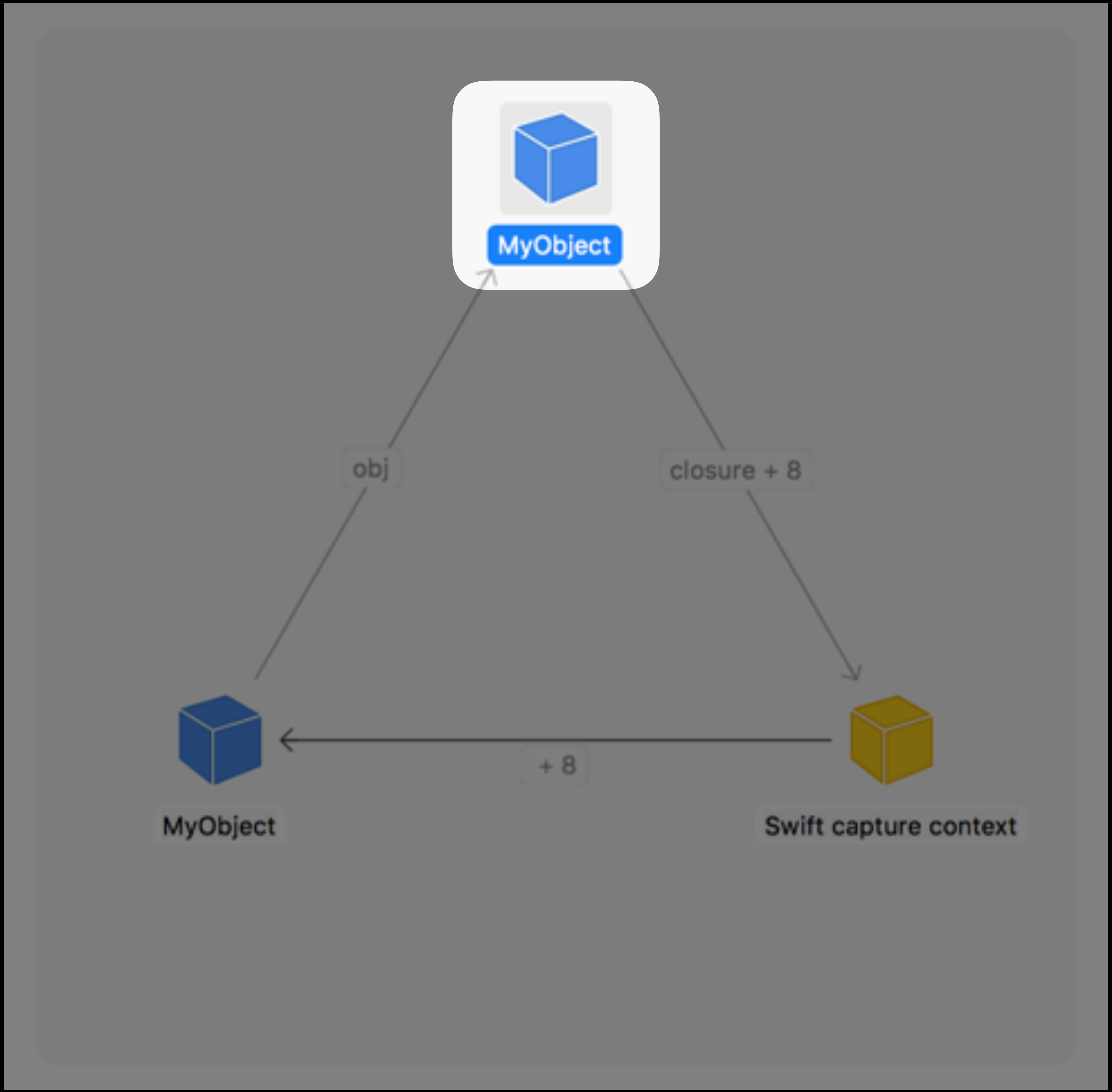
```
fetchData { data in
    self.doSomething(with: data)
}
```

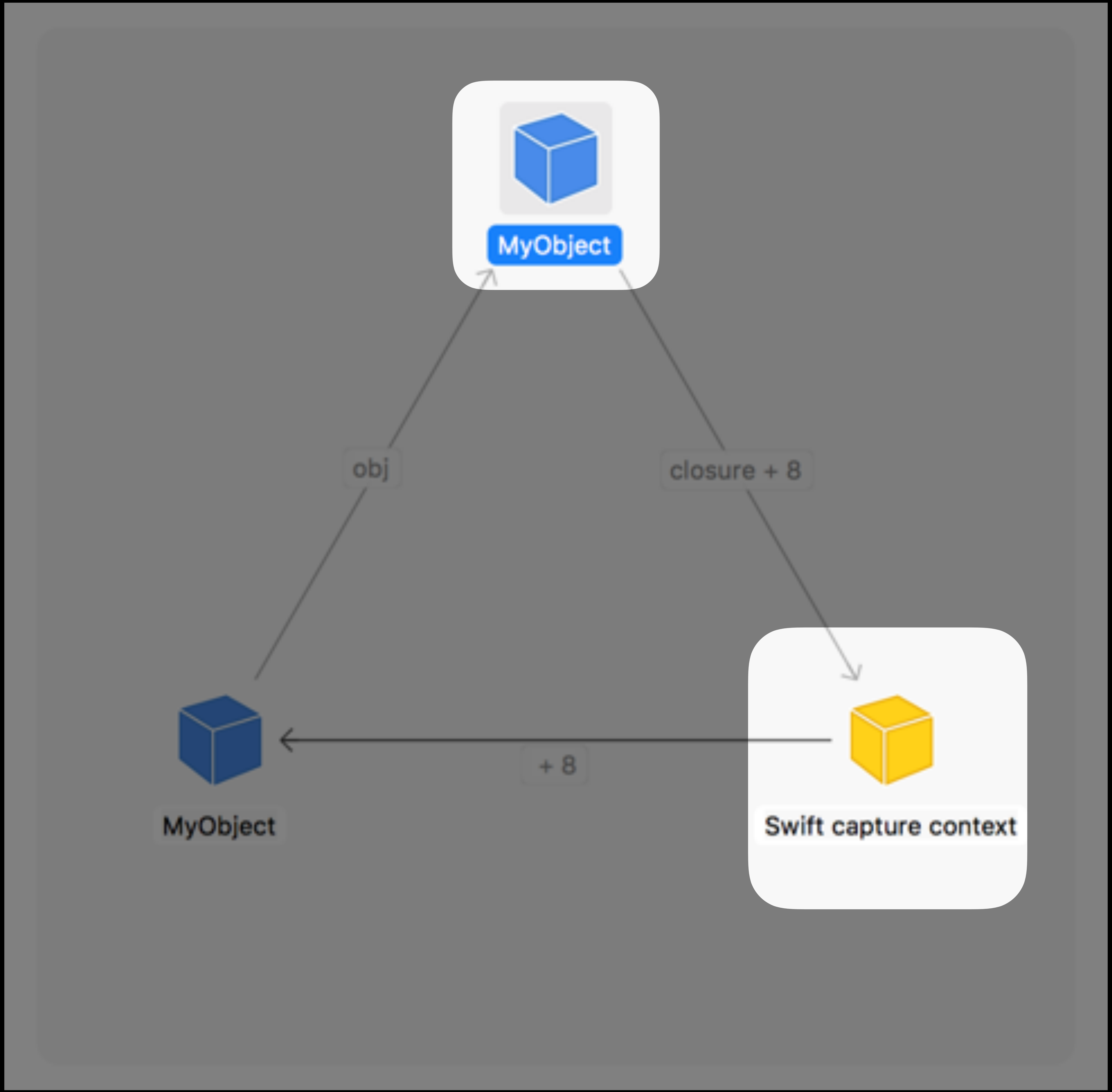
Asynchronous Code

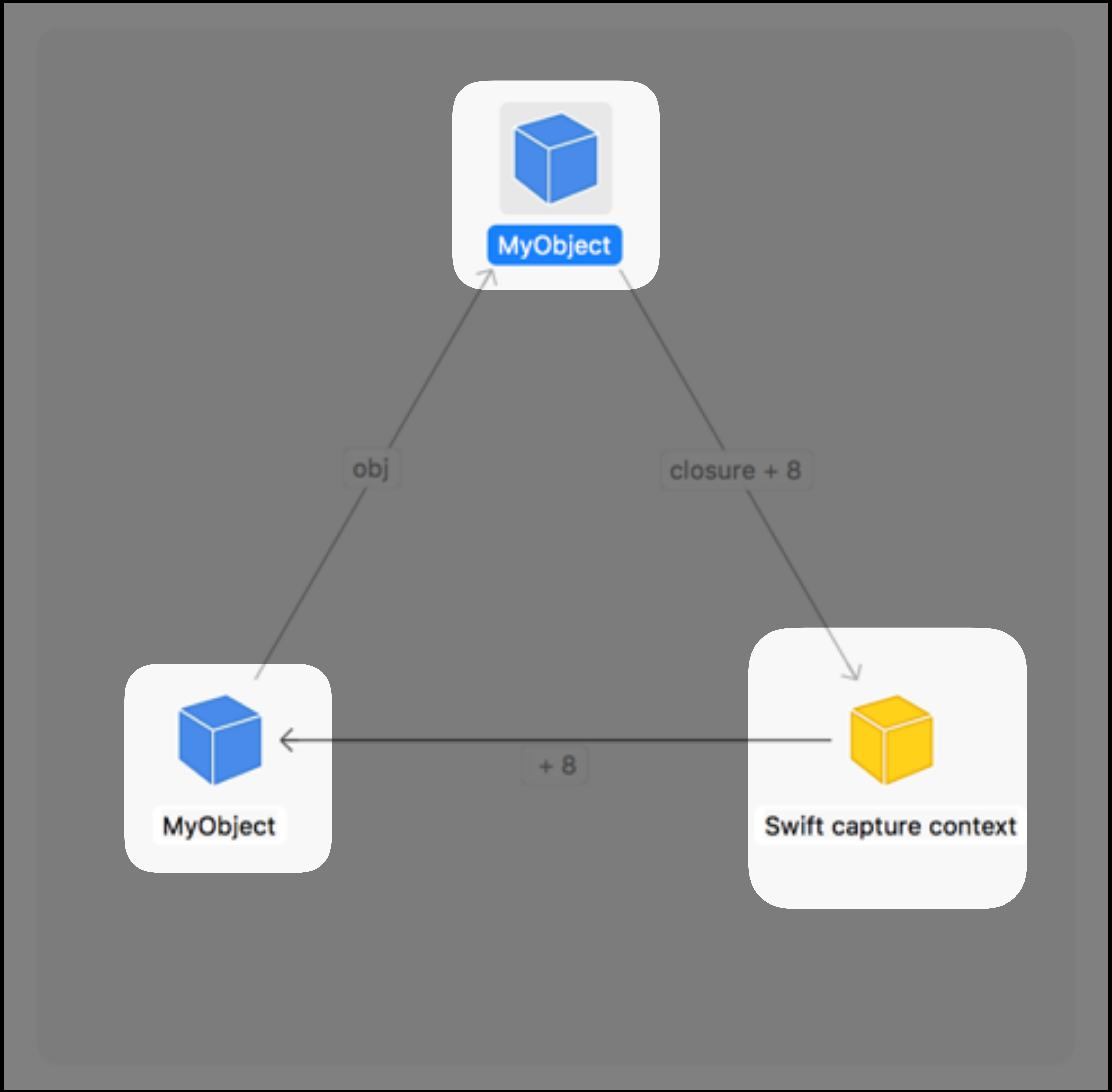
```
fetchData { data in
    self.doSomething(with: data)
}
```

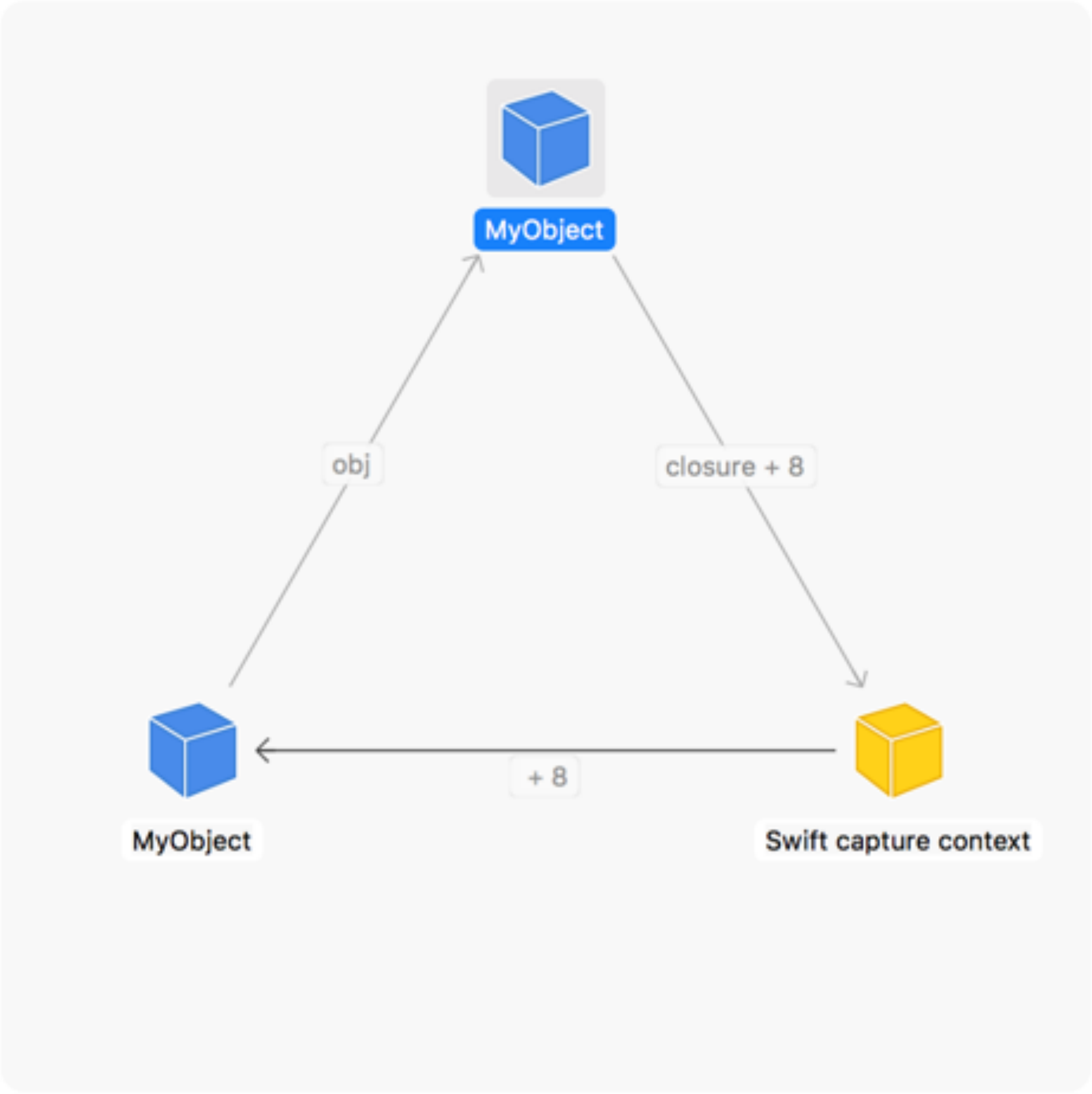
That’s how you get
a retain cycle!


MyObject
obj
closure + 8
MyObject
+ 8
Swift capture context


MyObject
obj
closure + 8
+ 8
MyObject
Swift capture context


MyObject
obj
closure + 8
MyObject
+ 8
Swift capture context


MyObject
obj
closure + 8
MyObject
+ 8
Swift capture context


MyObject
obj
closure + 8
MyObject
+ 8
Swift capture context

Asynchronous Code

```
fetchData { data in
    self.doSomething(with: data)
}
```

Asynchronous Code

```
fetchData { [weak self] data in
    self.doSomething(with: data)
}
```

Asynchronous Code

```
fetchData { [weak self] data in
    self.doSomething(with: data)
}
```

⛔

Asynchronous Code

```
fetchData { [weak self] data in
    self?.doSomething(with: data)
}
```

Asynchronous Code

```
fetchData { [weak self] data in
    self?.doSomething(with: data)
}
```

Asynchronous Code

```
fetchData { [weak self] data in
    guard let self = self else { return }

    self.doSomething(with: data)
}
```

Asynchronous Code

```
fetchData { [weak self] data in
    guard let self = self else { return }

    self.doSomething(with: data)
}
```

Asynchronous Code

```
fetchData { [weak self] data in
    guard let self = self else { return }

    self.doSomething(with: data)
}
```

So much boilerplate 😔

Look how they massacred my boy

We can do better 💪

Where do we start? 🤔

completionHandler

`completionHandler`

is a function

What do we know about functions in Swift?

“Functions are first-class citizens in Swift”

What does it mean?

Three things!

Functions can be **stored in variables**

```swift
var increment: (Int) -> Int = { $0 + 1 }
```

Functions can be **passed as arguments**

```swift
[1, 2, 3].map { $0 * $0 }
```

Functions can **return functions**

```swift
func buildIncrementor() -> (Int) -> Int {
    return { $0 + 1 }
}
```

So a function can take a function as its argument…

…and can also return a function…

…so we can build functions that “enhance” other functions!

Let’s take a look at how this might work 👀

```
protocol Weakifiable: class { }

extension NSObject: Weakifiable { }
```

```swift
extension Weakifiable {




}
```

```
extension Weakifiable {
    func weakify(_ code: @escaping (Self) -> Void) -> () -> Void {





    }
}
```

```swift
extension Weakifiable {
    func weakify(_ code: @escaping (Self) -> Void) -> () -> Void {





    }
}
```

```
extension Weakifiable {
    func weakify(_ code: @escaping (Self) -> Void) -> () -> Void {





    }
}
```

```
extension Weakifiable {
    func weakify(_ code: @escaping (Self) -> Void) -> () -> Void {





    }
}
```

```swift
extension Weakifiable {
    func weakify(_ code: @escaping (Self) -> Void) -> () -> Void {
        return {



        }
    }
}
```

```
extension Weakifiable {
    func weakify(_ code: @escaping (Self) -> Void) -> () -> Void {
        return {



        }
    }
}
```

```swift
extension Weakifiable {
    func weakify(_ code: @escaping (Self) -> Void) -> () -> Void {
        return { [weak self] in



        }
    }
}
```

```
extension Weakifiable {
    func weakify(_ code: @escaping (Self) -> Void) -> () -> Void {
        return { [weak self] in
            guard let self = self else { return }


        }
    }
}
```

```swift
extension Weakifiable {
    func weakify(_ code: @escaping (Self) -> Void) -> () -> Void {
        return { [weak self] in
            guard let self = self else { return }

            code(self)
        }
    }
}
```

```swift
extension Weakifiable {
    func weakify(_ code: @escaping (Self) -> Void) -> () -> Void {
        return { [weak self] in
            guard let self = self else { return }

            code(self)
        }
    }
}
```

```swift
extension Weakifiable {
    func weakify(_ code: @escaping (Self) -> Void) -> () -> Void {
        return { [weak self] in
            guard let self = self else { return }

            code(self)
        }
    }

    func weakify<T>(_ code: @escaping (T, Self) -> Void) -> (T) -> Void {
        return { [weak self] arg in
            guard let self = self else { return }

            code(arg, self)
        }
    }
}
```

```
extension Weakifiable {
    func weakify(_ code: @escaping (Self) -> Void) -> () -> Void {
        return { [weak self] in
            guard let self = self else { return }

            code(self)
        }
    }

    func weakify<T>(_ code: @escaping (T, Self) -> Void) -> (T) -> Void {
        return { [weak self] arg in
            guard let self = self else { return }

            code(arg, self)
        }
    }
}
```

```
extension Weakifiable {
    func weakify(_ code: @escaping (Self) -> Void) -> () -> Void {
        return { [weak self] in
            guard let self = self else { return }

            code(self)
        }
    }

    func weakify<T>(_ code: @escaping (T, Self) -> Void) -> (T) -> Void {
        return { [weak self] arg in
            guard let self = self else { return }

            code(arg, self)
        }
    }
}
```

```swift
extension Weakifiable {
    func weakify(_ code: @escaping (Self) -> Void) -> () -> Void {
        return { [weak self] in
            guard let self = self else { return }

            code(self)
        }
    }

    func weakify<T>(_ code: @escaping (T, Self) -> Void) -> (T) -> Void {
        return { [weak self] arg in
            guard let self = self else { return }

            code(arg, self)
        }
    }
}
```

Let’s use this 🤩

Asynchronous Code

```
fetchData { [weak self] data in
    guard let self = self else { return }

    self.doSomething(with: data)
}
```

Asynchronous Code

```
fetchData( weakify { data, strongSelf in
    strongSelf.doSomething(with: data)
})
```

Asynchronous Code

```
fetchData( weakify { data, strongSelf in
    strongSelf.doSomething(with: data)
})
```

Asynchronous Code

```
fetchData( weakify { data, strongSelf in
    strongSelf.doSomething(with: data)
})
```

No more boilerplate 😍

Let’s reflect on what we’ve just achieved 🧐

`weakify` is a function that enhances a piece of code

We could call `weakify` a “pseudo-keyword”

What other “pseudo-keywords” could we implement? 🤔

Debouncing

Debouncing

Definition: waiting for a given **timespan** to elapse before performing an action.

Any new call during that timeframe **resets** the chronometer.

Some use cases:

- When users inputs text in a search field, we want to wait until they’ve paused their typing before we fire a network request.
- When users scroll a view, we want to wait until they’ve stopped scrolling to fire an analytics event.

```swift
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {







}
```

```swift
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {







}
```

```
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {






}
```

```
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {







}
```

```
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {







}
```

```swift
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {







}
```

```swift
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {
    var workItem: DispatchWorkItem?







}
```

```
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {
    var workItem: DispatchWorkItem?

    return {



    }
}
```

```
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {
    var workItem: DispatchWorkItem?

    return {
        workItem?.cancel()


    }
}
```

```swift
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {
    var workItem: DispatchWorkItem?

    return {
        workItem?.cancel()
        workItem = DispatchWorkItem(block: action)

    }
}
```

```
func debounced(delay: TimeInterval = 0.3,
               queue: DispatchQueue = .main,
               action: @escaping (() -> Void))
               -> () -> Void {
    var workItem: DispatchWorkItem?

    return {
        workItem?.cancel()
        workItem = DispatchWorkItem(block: action)
        queue.asyncAfter(deadline: .now() + delay, execute: workItem!)
    }
}
```

```swift
func debounced<T>(delay: TimeInterval = 0.3,
                  queue: DispatchQueue = .main,
                  action: @escaping ((T) -> Void))
                  -> (T) -> Void {
    var workItem: DispatchWorkItem?

    return { arg in
        workItem?.cancel()
        workItem = DispatchWorkItem(block: { action(arg) })
        queue.asyncAfter(deadline: .now() + delay, execute: workItem!)
    }
}
```

```
     debounced<T>(delay: TimeInterval = 0.3,
                  queue: DispatchQueue = .main,
                  action: @escaping ((T) -> Void))
                  -> (T) -> Void {
    var workItem: DispatchWorkItem?

    return { arg in
        workItem?.cancel()
        workItem = DispatchWorkItem(block: { action(arg) })
        queue.asyncAfter(deadline: .now() + delay, execute: workItem!)
    }
}
```

```swift
func debounced<T>(delay: TimeInterval = 0.3,
                  queue: DispatchQueue = .main,
                  action: @escaping ((T) -> Void))
                  -> (T) -> Void {
    var workItem: DispatchWorkItem?

    return { arg in
        workItem?.cancel()
        workItem = DispatchWorkItem(block: { action(arg) })
        queue.asyncAfter(deadline: .now() + delay, execute: workItem!)
    }
}
```

```swift
class ViewController: UIViewController, UIScrollViewDelegate {









}
```

```swift
class ViewController: UIViewController, UIScrollViewDelegate {
    // ...





}
```

```swift
class ViewController: UIViewController, UIScrollViewDelegate {

    // ...

    let didScrollHandler = { (scrollView: UIScrollView) in
        print(scrollView.contentOffset)
    }



}
```

```swift
class ViewController: UIViewController, UIScrollViewDelegate {

    // ...

    let didScrollHandler = debounced { (scrollView: UIScrollView) in
        print(scrollView.contentOffset)
    }



}
```

```
class ViewController: UIViewController, UIScrollViewDelegate {

    // ...

    let didScrollHandler = debounced { (scrollView: UIScrollView) in
        print(scrollView.contentOffset)
    }

    func scrollViewDidScroll(_ scrollView: UIScrollView) {

    }
}
```

```swift
class ViewController: UIViewController, UIScrollViewDelegate {

    // ...

    let didScrollHandler = debounced { (scrollView: UIScrollView) in
        print(scrollView.contentOffset)
    }

    func scrollViewDidScroll(_ scrollView: UIScrollView) {
        self.didScrollHandler(scrollView)
    }
}
```

But wait, isn’t there a built-in way in Swift to support these “pseudo-keywords”?

A little something called “Property Wrappers”?

```
struct ContentView : View {
    @State var model = Themes.listModel

    var body: some View {
        List(model.items, action: model.selectItem) { item in
            Image(item.image)
            VStack(alignment: .leading) {
                Text(item.title)
                Text(item.subtitle).color(.gray)
            }
        }
    }
}
```

Property Wrappers

```
struct Handler {
    @Debounced(delay: 1.0) var action: () -> Void

    func handle() {
        action()
    }
}
```

Property Wrappers

```
@Debounced(delay: 1.0) var action: () -> Void
```

```
struct Debounced {

}
```

```swift
struct Debounced {
    let delay: TimeInterval
    let queue: DispatchQueue
    var action: () -> Void = { }

}
```

```swift
struct Debounced {
    let delay: TimeInterval
    let queue: DispatchQueue
    var action: () -> Void = { }

    init(delay: TimeInterval, queue: DispatchQueue = .main) {
        self.delay = delay
        self.queue = queue
    }














}
```

```swift
@propertyWrapper
struct Debounced {
    let delay: TimeInterval
    let queue: DispatchQueue
    var action: () -> Void = { }

    init(delay: TimeInterval, queue: DispatchQueue = .main) {
        self.delay = delay
        self.queue = queue
    }

}
```

```
@propertyWrapper
struct Debounced {
    let delay: TimeInterval
    let queue: DispatchQueue
    var action: () -> Void = { }

    init(delay: TimeInterval, queue: DispatchQueue = .main) {
        self.delay = delay
        self.queue = queue
    }

    var wrappedValue: () -> Void {










    }
}
```

```swift
@propertyWrapper
struct Debounced {
    let delay: TimeInterval
    let queue: DispatchQueue
    var action: () -> Void = { }

    init(delay: TimeInterval, queue: DispatchQueue = .main) {
        self.delay = delay
        self.queue = queue
    }

    var wrappedValue: () -> Void {
        get { return action }









    }
}
```

```swift
@propertyWrapper
struct Debounced {
    let delay: TimeInterval
    let queue: DispatchQueue
    var action: () -> Void = { }

    init(delay: TimeInterval, queue: DispatchQueue = .main) {
        self.delay = delay
        self.queue = queue
    }

    var wrappedValue: () -> Void {
        get { return action }
        set {
            var workItem: DispatchWorkItem?






        }
    }
}
```

```
@propertyWrapper
struct Debounced {
    let delay: TimeInterval
    let queue: DispatchQueue
    var action: () -> Void = { }

    init(delay: TimeInterval, queue: DispatchQueue = .main) {
        self.delay = delay
        self.queue = queue
    }

    var wrappedValue: () -> Void {
        get { return action }
        set {
            var workItem: DispatchWorkItem?

            self.action = { [queue, delay] in



            }
        }
    }
}
```

```swift
@propertyWrapper
struct Debounced {
    let delay: TimeInterval
    let queue: DispatchQueue
    var action: () -> Void = { }

    init(delay: TimeInterval, queue: DispatchQueue = .main) {
        self.delay = delay
        self.queue = queue
    }

    var wrappedValue: () -> Void {
        get { return action }
        set {
            var workItem: DispatchWorkItem?

            self.action = { [queue, delay] in
                workItem?.cancel()
                workItem = DispatchWorkItem(block: newValue)
                queue.asyncAfter(deadline: .now() + delay, execute: workItem!)
            }
        }
    }
}
```

Property Wrappers

```swift
struct Handler {
    @Debounced(delay: 1.0) var action: () -> Void

    func handle() {
        action()
    }
}
```

Conclusion

Conclusion

Functional programming is a powerful tool 💪

There's definitely several use cases waiting to be discovered in each of your projects and codebases 🚀

Nice things should be appreciated, but never abused 😱

Property Wrappers 😍

Implementing pseudo-keywords through functional programing

Vincent Pradeilles (@v_pradeilles) – Worldline 🇫🇷