# El Patrón Observador en Swift

Ariel Elkin

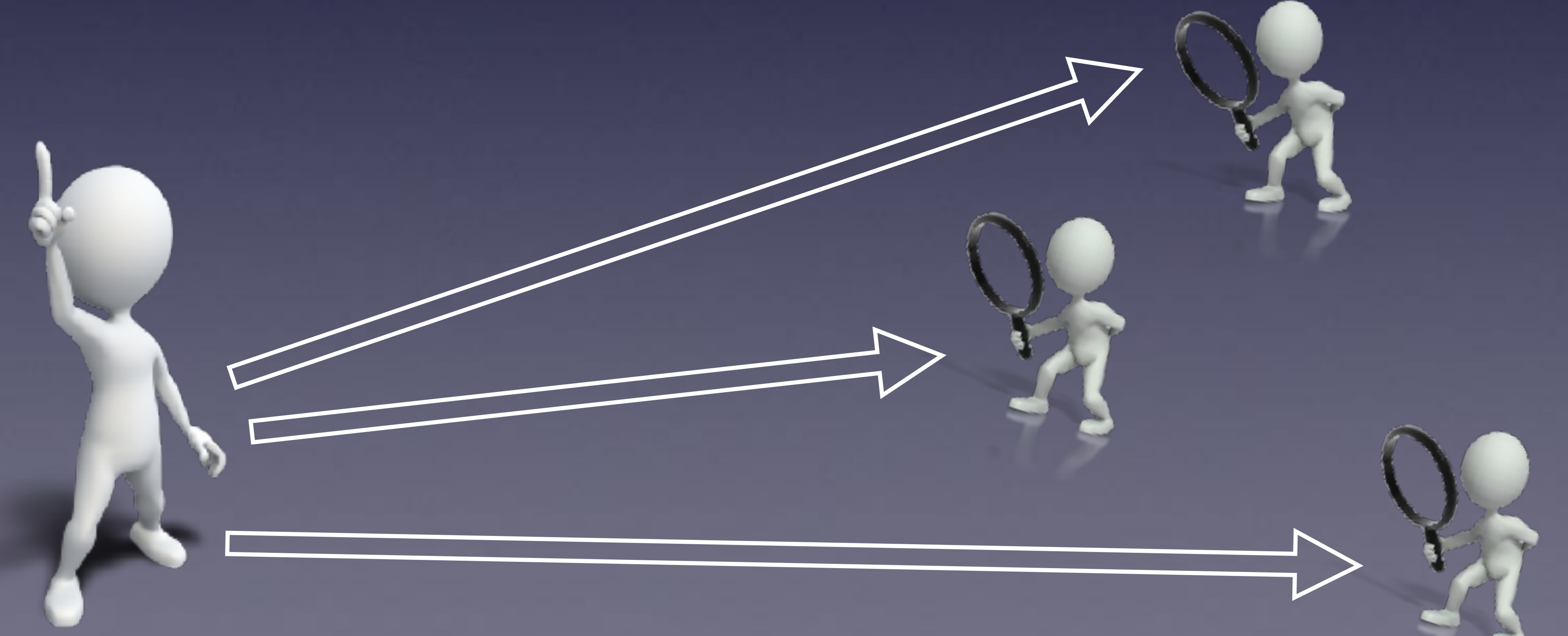
# Agenda

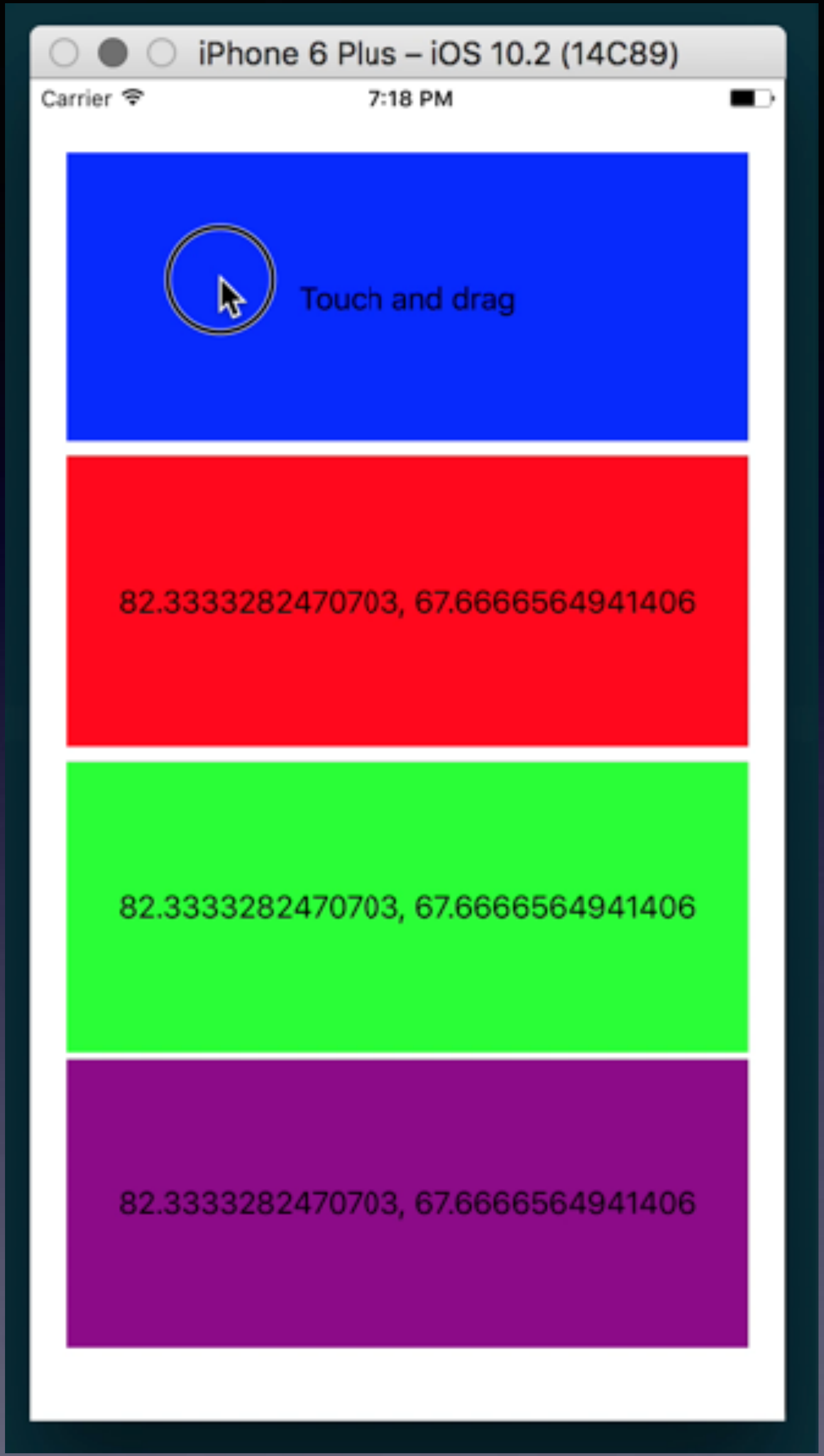
1. Teoría
2. Implementaciones
   a. `NSNotificationCenter`
   b. `Swift puro`

# Teoría

# Teoría

- “Observador es un **patrón de diseño** que define una dependencia del tipo uno-a-muchos entre objetos, de manera que cuando uno de los objetos cambia su estado, notifica este cambio a todos los dependientes.”





El Patrón Observador es una de las maneras de definir la relación entre Modelo, Vista y Controlador (MVC)

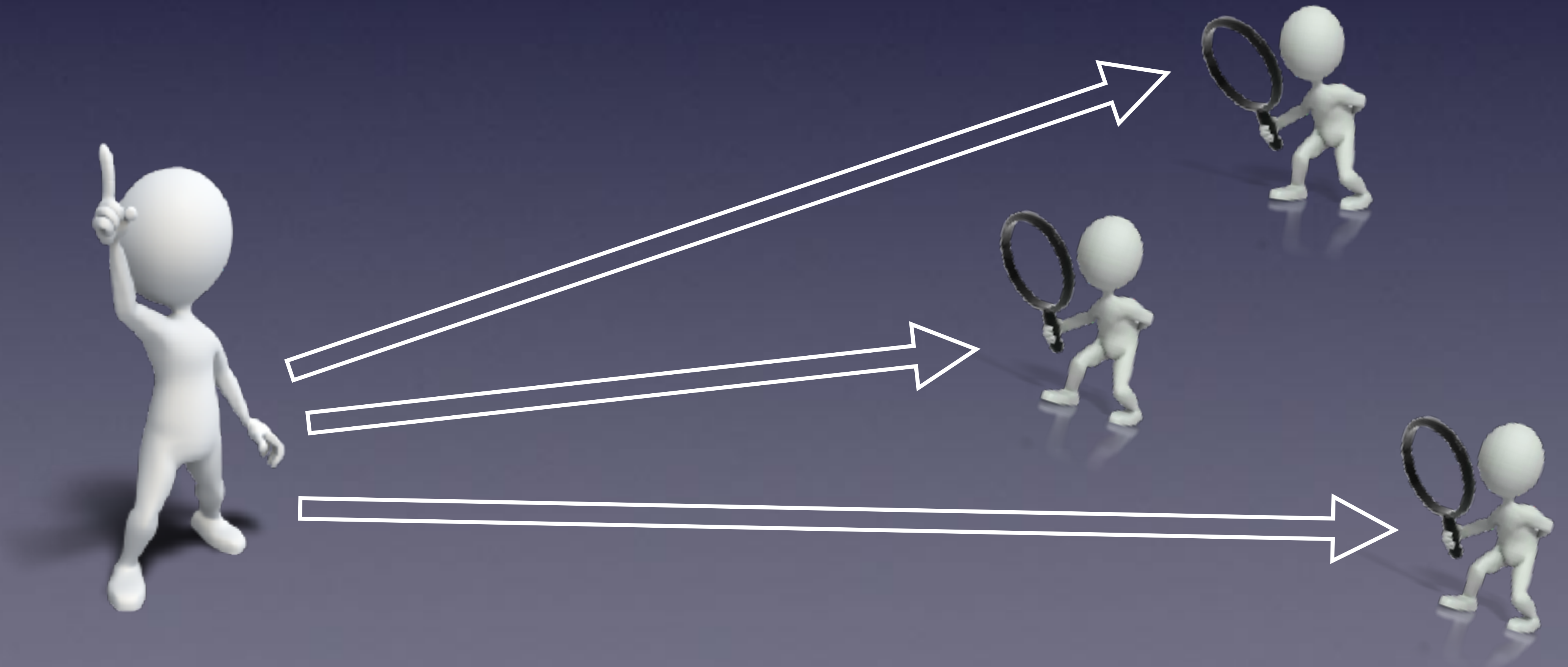
# Teoría

**Sujeto**

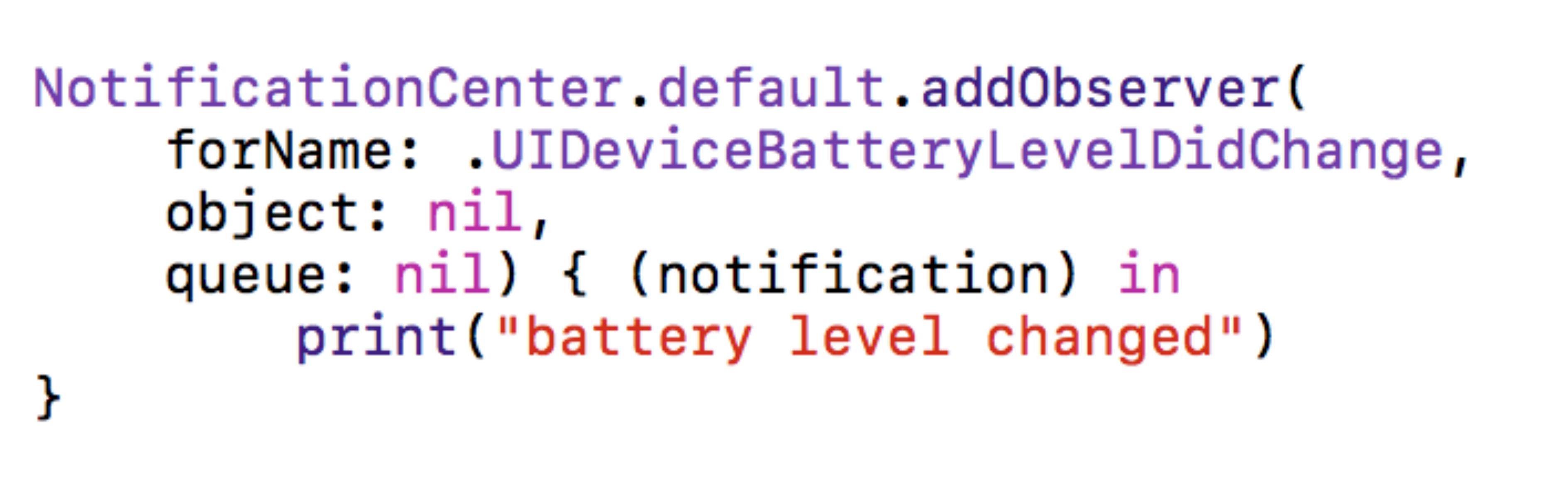
```
agregar(Observador)
desatar(Observador)
notificar()
```

**Observador**

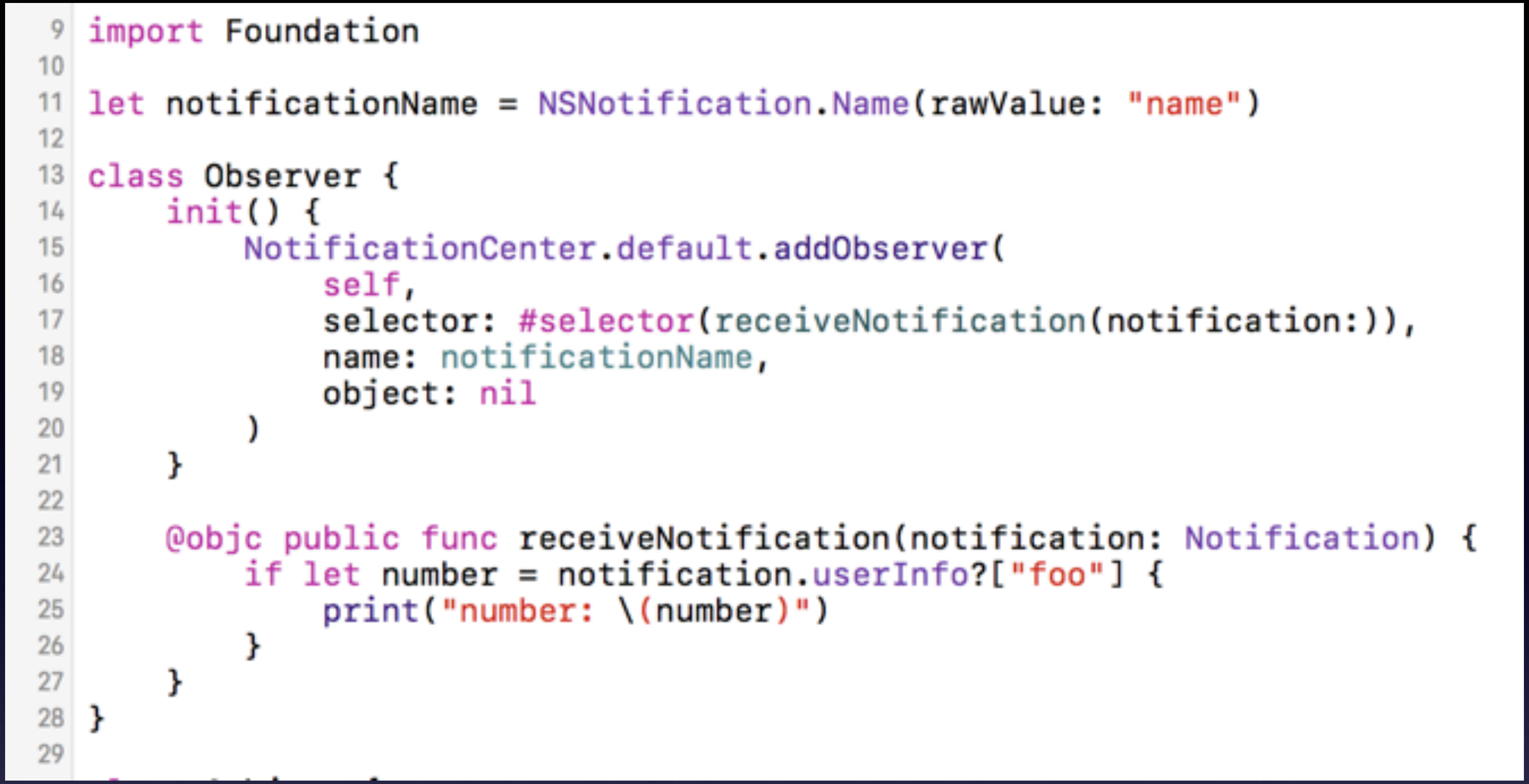
```
actualizar()
```



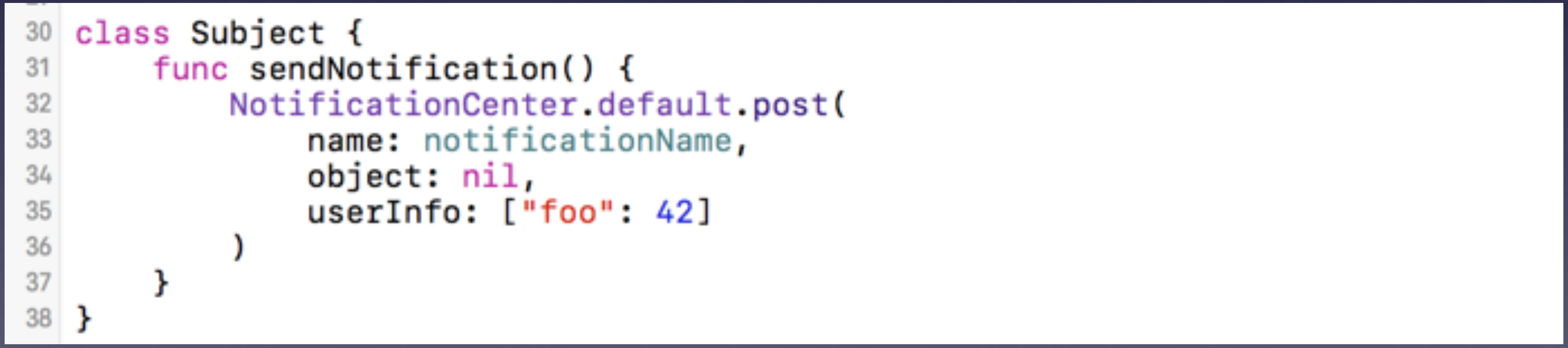
# Implementaciones: `NotificationCenter`



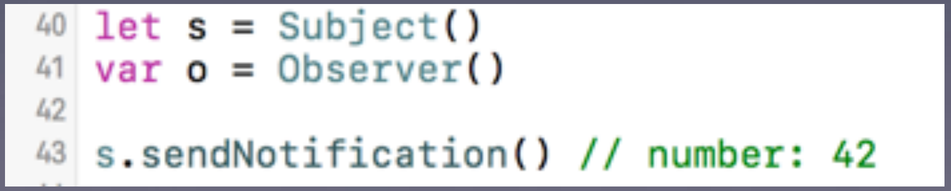
```
NotificationCenter.default.addObserver(
    forName: .UIDeviceBatteryLevelDidChange,
    object: nil,
    queue: nil) { (notification) in
        print("battery level changed")
}
```



```swift
import Foundation

let notificationName = NSNotification.Name(rawValue: "name")

class Observer {
    init() {
        NotificationCenter.default.addObserver(
            self,
            selector: #selector(receiveNotification(notification:)),
            name: notificationName,
            object: nil
        )
    }

    @objc public func receiveNotification(notification: Notification) {
        if let number = notification.userInfo?["foo"] {
            print("number: \(number)")
        }
    }
}

```



```swift
class Subject {
    func sendNotification() {
        NotificationCenter.default.post(
            name: notificationName,
            object: nil,
            userInfo: ["foo": 42]
        )
    }
}
```

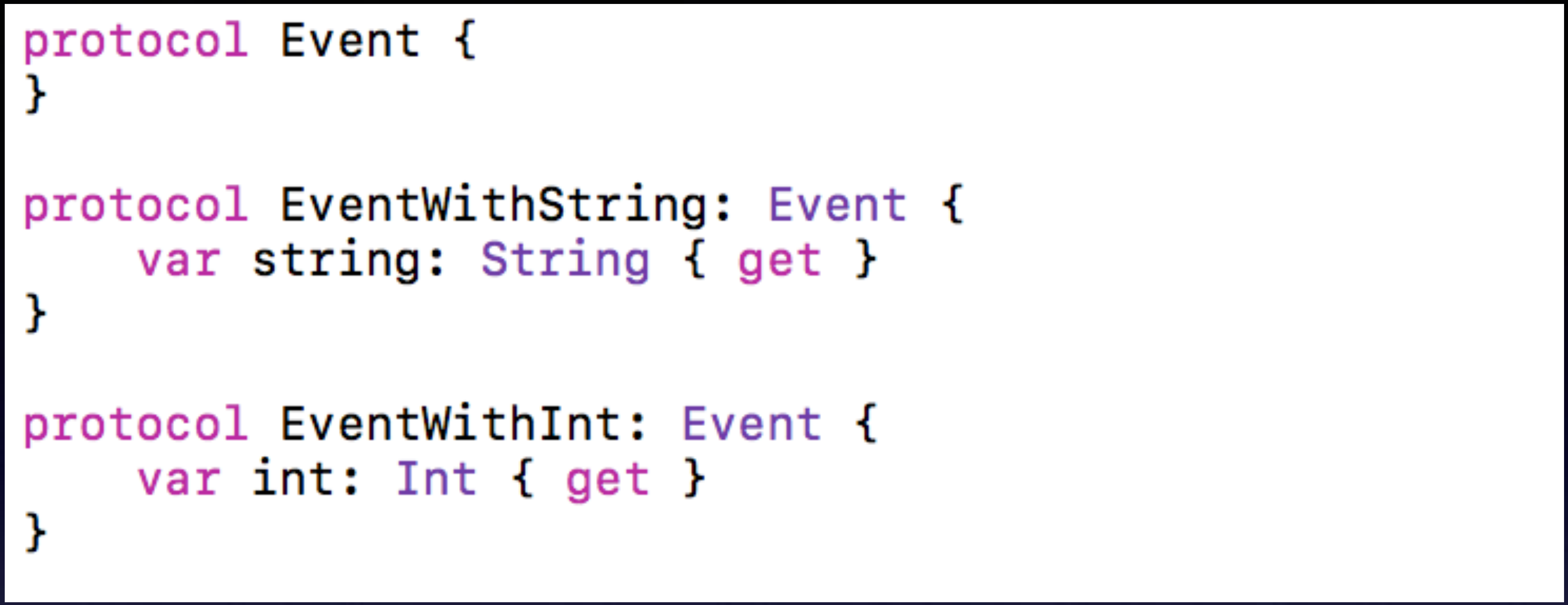


```swift
let s = Subject()
var o = Observer()

s.sendNotification() // number: 42
```

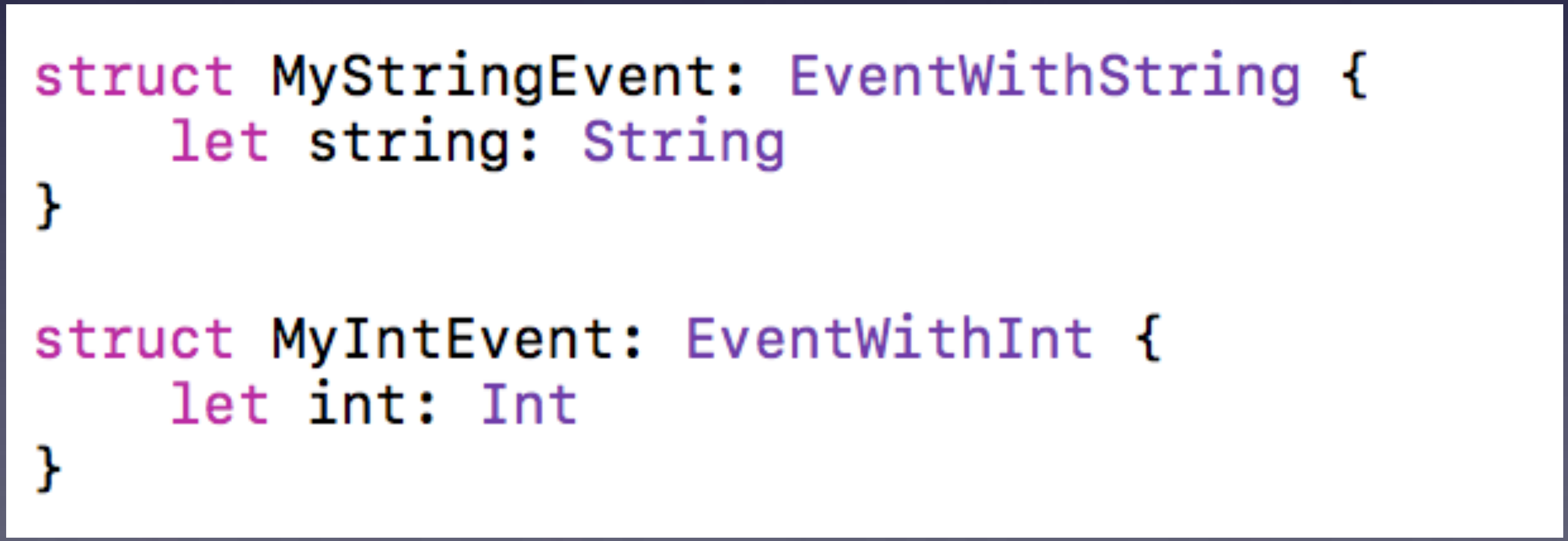
# Desventajas de NotificationCenter

- Mantenimiento de una lista de `NSNotification.Name`
- Imposibilidad de definir tipos de notificaciones
- No sabemos quienes son los observadores

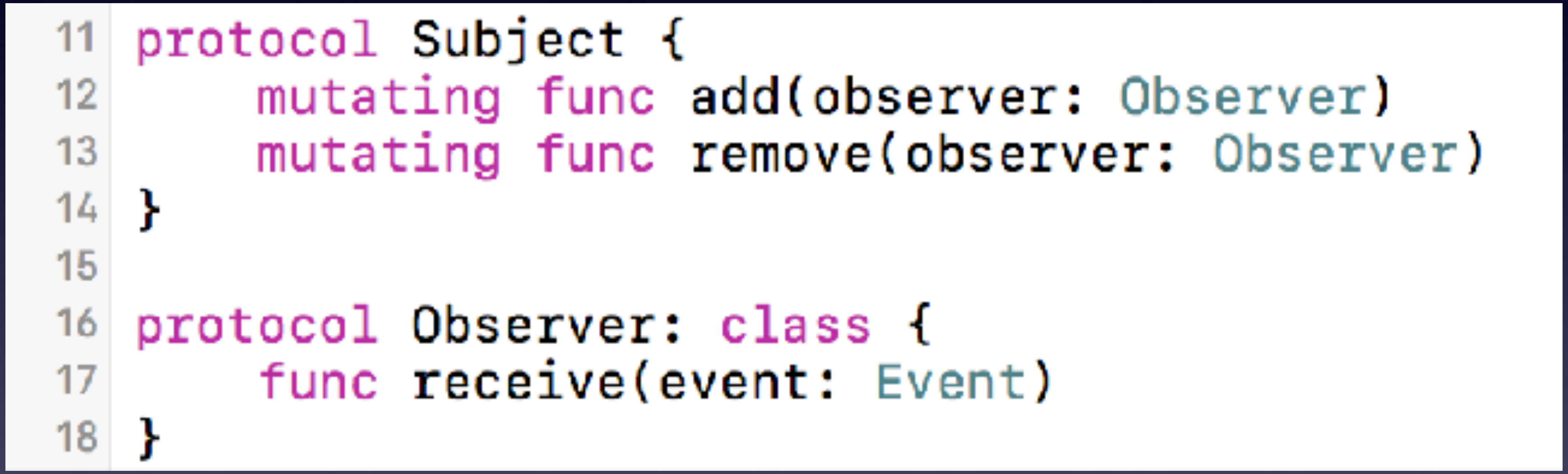
# Implementaciones:
# `Swift puro`



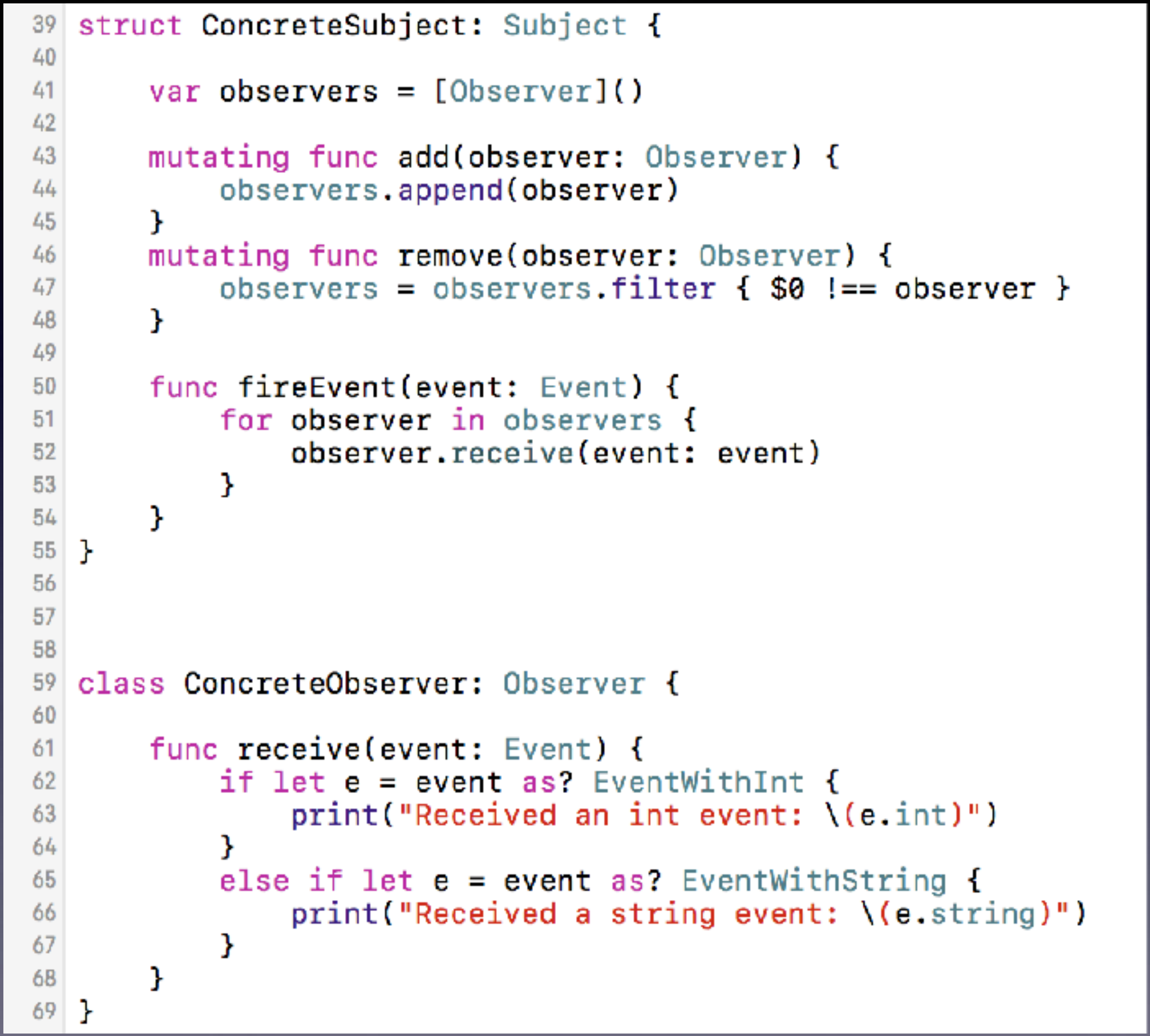
```swift
protocol Event {
}

protocol EventWithString: Event {
    var string: String { get }
}

protocol EventWithInt: Event {
    var int: Int { get }
}
```



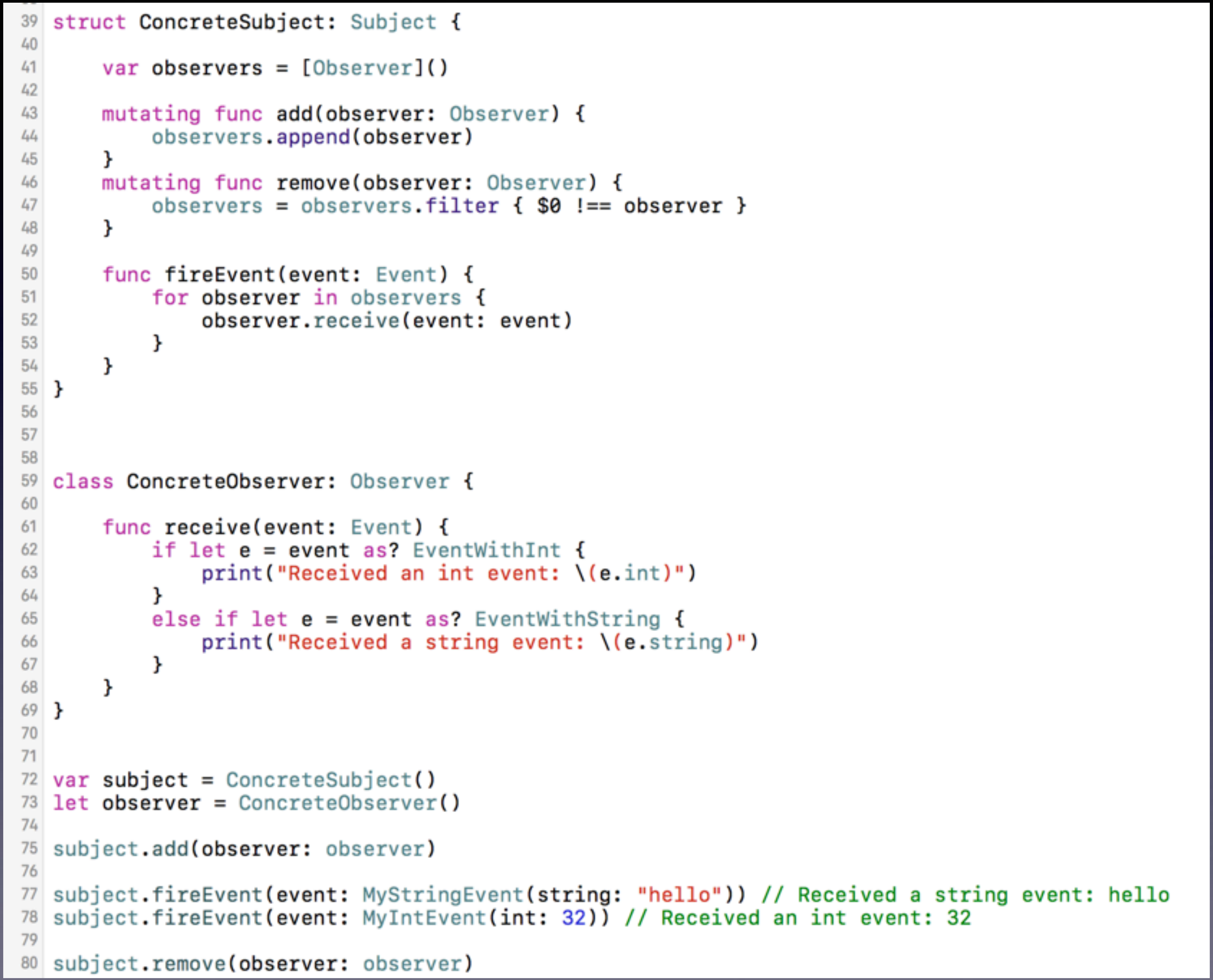
```swift
struct MyStringEvent: EventWithString {
    let string: String
}

struct MyIntEvent: EventWithInt {
    let int: Int
}
```



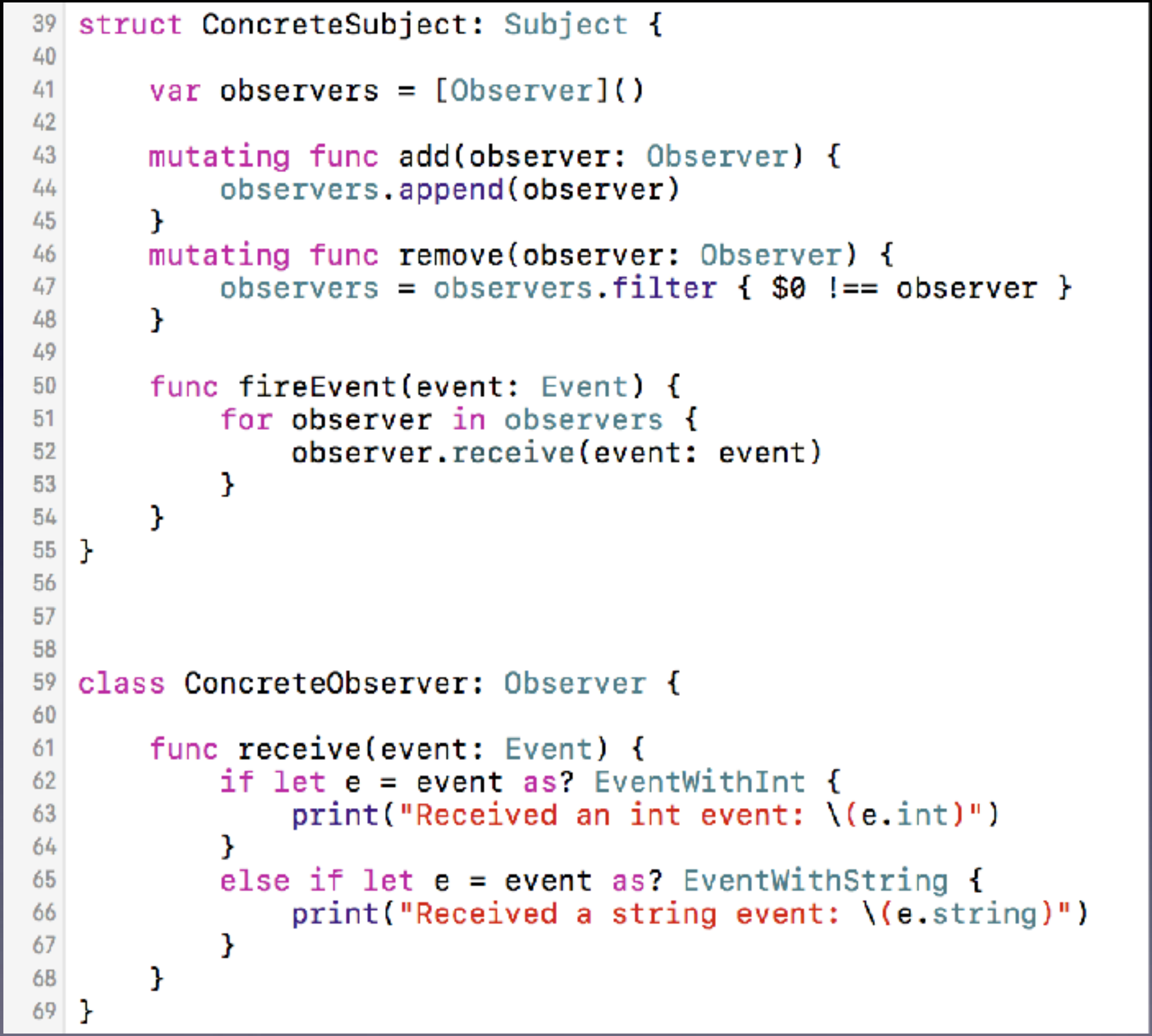
```
protocol Subject {
    mutating func add(observer: Observer)
    mutating func remove(observer: Observer)
}

protocol Observer: class {
    func receive(event: Event)
}
```



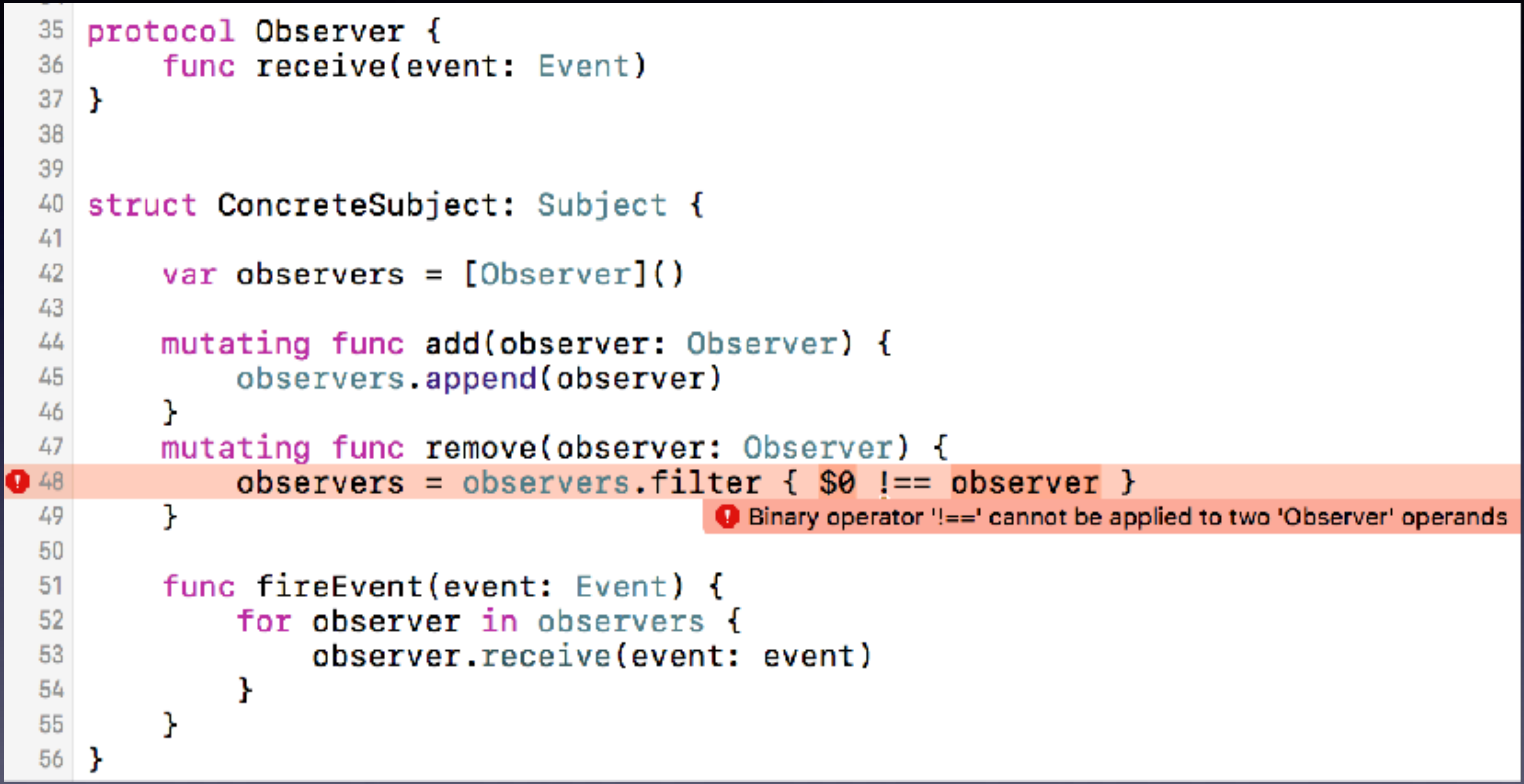
```swift
struct ConcreteSubject: Subject {

    var observers = [Observer]()

    mutating func add(observer: Observer) {
        observers.append(observer)
    }
    mutating func remove(observer: Observer) {
        observers = observers.filter { $0 !== observer }
    }

    func fireEvent(event: Event) {
        for observer in observers {
            observer.receive(event: event)
        }
    }
}



class ConcreteObserver: Observer {

    func receive(event: Event) {
        if let e = event as? EventWithInt {
            print("Received an int event: \(e.int)")
        }
        else if let e = event as? EventWithString {
            print("Received a string event: \(e.string)")
        }
    }
}
```



```swift
struct ConcreteSubject: Subject {

    var observers = [Observer]()

    mutating func add(observer: Observer) {
        observers.append(observer)
    }
    mutating func remove(observer: Observer) {
        observers = observers.filter { $0 !== observer }
    }

    func fireEvent(event: Event) {
        for observer in observers {
            observer.receive(event: event)
        }
    }
}



class ConcreteObserver: Observer {

    func receive(event: Event) {
        if let e = event as? EventWithInt {
            print("Received an int event: \(e.int)")
        }
        else if let e = event as? EventWithString {
            print("Received a string event: \(e.string)")
        }
    }
}


var subject = ConcreteSubject()
let observer = ConcreteObserver()

subject.add(observer: observer)

subject.fireEvent(event: MyStringEvent(string: "hello")) // Received a string event: hello
subject.fireEvent(event: MyIntEvent(int: 32)) // Received an int event: 32

subject.remove(observer: observer)
```

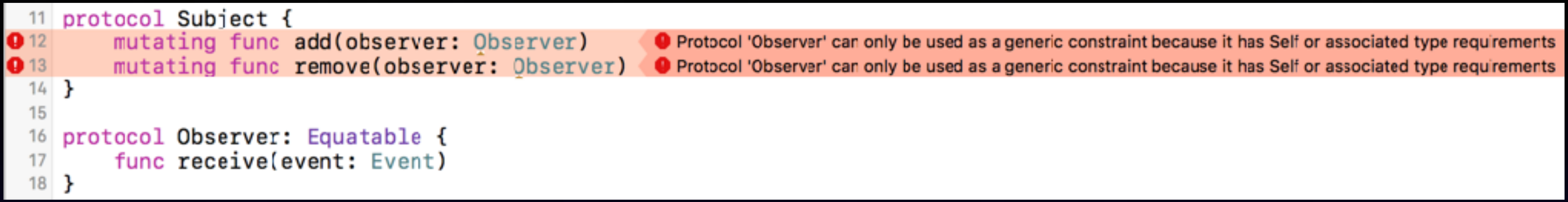


```swift
struct ConcreteSubject: Subject {

    var observers = [Observer]()

    mutating func add(observer: Observer) {
        observers.append(observer)
    }
    mutating func remove(observer: Observer) {
        observers = observers.filter { $0 !== observer }
    }

    func fireEvent(event: Event) {
        for observer in observers {
            observer.receive(event: event)
        }
    }
}



class ConcreteObserver: Observer {

    func receive(event: Event) {
        if let e = event as? EventWithInt {
            print("Received an int event: \(e.int)")
        }
        else if let e = event as? EventWithString {
            print("Received a string event: \(e.string)")
        }
    }
}
```



```swift
protocol Observer {
    func receive(event: Event)
}


struct ConcreteSubject: Subject {

    var observers = [Observer]()

    mutating func add(observer: Observer) {
        observers.append(observer)
    }
    mutating func remove(observer: Observer) {
        observers = observers.filter { $0 !== observer }
    }

    func fireEvent(event: Event) {
        for observer in observers {
            observer.receive(event: event)
        }
    }
}
```

Binary operator '!==' cannot be applied to two 'Observer' operands

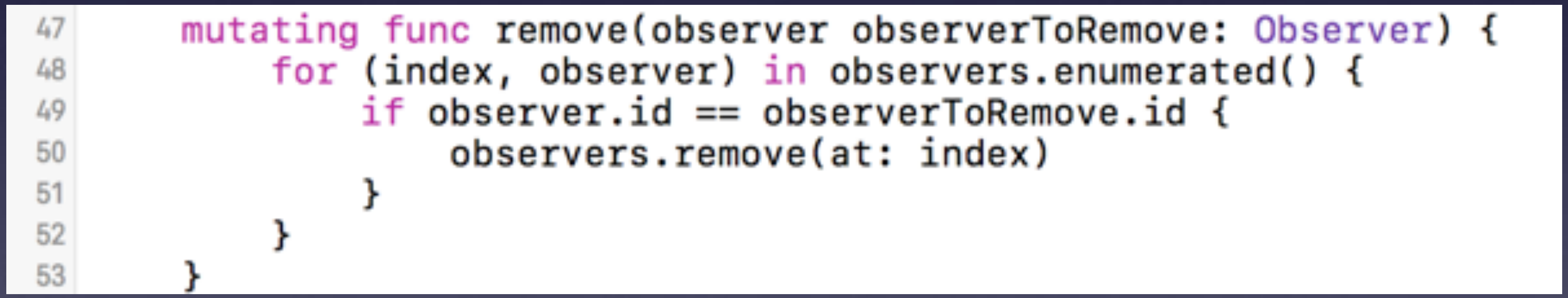


```
protocol Subject {
    mutating func add(observer: Observer)
    mutating func remove(observer: Observer)
}

protocol Observer: Equatable {
    func receive(event: Event)
}
```

Protocol 'Observer' can only be used as a generic constraint because it has Self or associated type requirements

Protocol 'Observer' can only be used as a generic constraint because it has Self or associated type requirements

Agregar una variable **`id`** a la interfaz de los observadores:



```
    mutating func remove(observer observerToRemove: Observer) {
        for (index, observer) in observers.enumerated() {
            if observer.id == observerToRemove.id {
                observers.remove(at: index)
            }
        }
    }
```

O bien esperar que lleguen los “existentials"

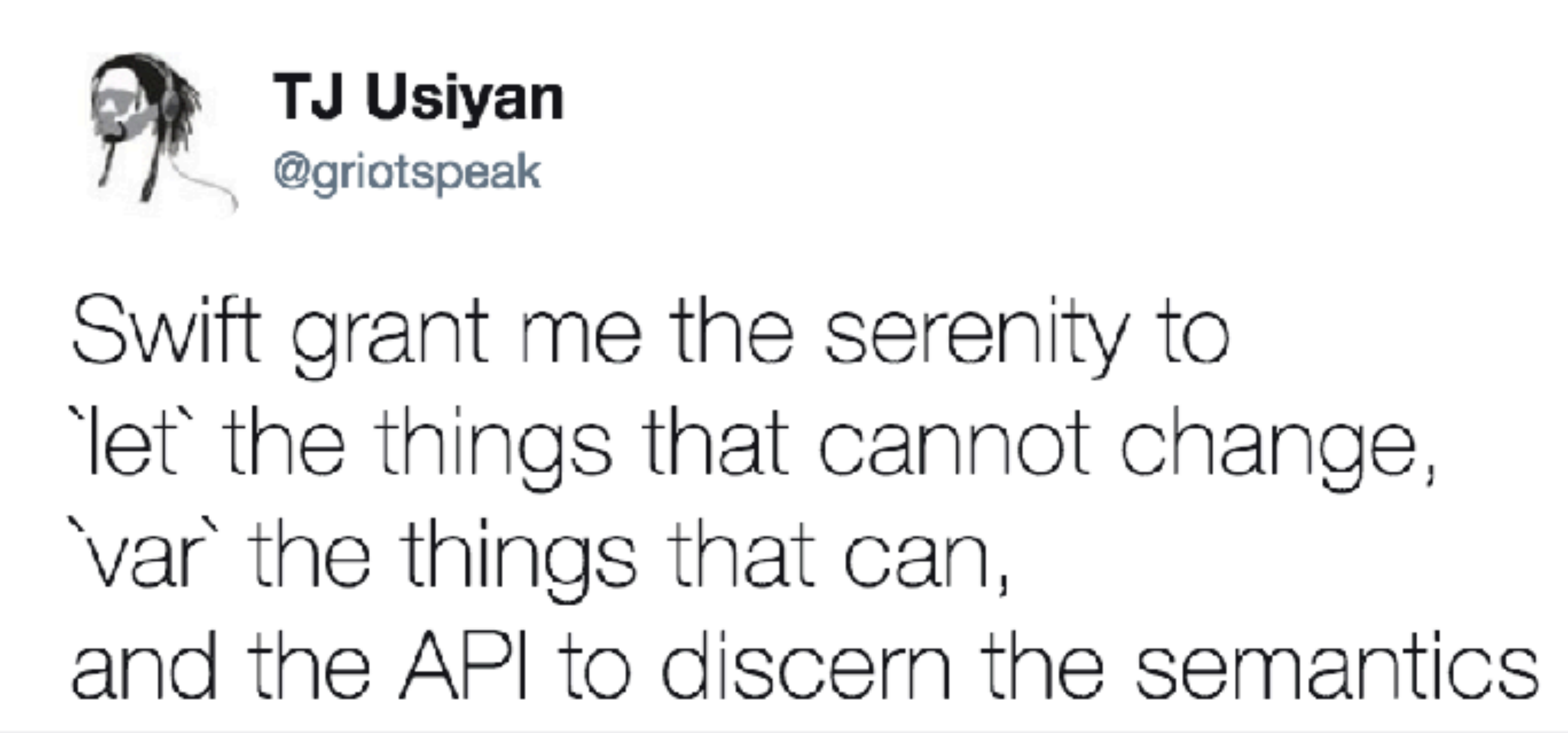
# Ventajas

- Tipado fuerte
- Control de la definición de Eventos
- Control del despacho de Eventos
- Sabemos quienes son los observadores

# Desventajas

- Los observadores tienen que ser clases o bien tener atributos públicos identificadores

- Manejo manual de las referencias fuertes a los Observadores



**TJ Usiyan**
@griotspeak

Swift grant me the serenity to
`let` the things that cannot change,
`var` the things that can,
and the API to discern the semantics

# Gracias!

# Preguntas?

ariel@arivibes.com
@AriVocals
arielelkin.github.io

# Fuentes

- *Design Patterns*. The "Gang of Four"

- *Advanced Swift*. Chris Eidhof, Ole Begemann and Airspeed Velocity

- *Pro Design Patterns in Swift*. Adam Freeman

- *NSNotification & NSNotification Center.* Matt Thompson