



Swift Quiz

Vincent Pradeilles ([@v_pradeilles](#)) – [PhotoRoom](#)



Question #01

Closure, Capture List and Retain Cycle


```
class ViewController: UIViewController {  
  
    let service = Service()  
    let formatter = Formatter()  
    let label = UILabel()  
  
    var cancellables = Set<AnyCancellable>()  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        service.call().sink { [formatter, label] data in  
            let formatted = formatter.format(data: data)  
            label.text = "Retrieved data: \(formatted)"  
        }.store(in: &cancellables)  
    }  
}
```

```
class ViewController: UIViewController {  
  
    let service = Service()  
    let formatter = Formatter()  
    let label = UILabel()  
  
    var cancellables = Set<AnyCancellable>()  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        service.call().sink { [formatter, label] data in  
            let formatted = formatter.format(data: data)  
            label.text = "Retrieved data: \(formatted)"  
        }.store(in: &cancellables)  
    }  
}
```

// Will instances of `ViewController` leak?



```
class ViewController: UIViewController {  
  
    let service = Service()  
    let formatter = Formatter()  
    let label = UILabel()  
  
    var cancellables = Set<AnyCancellable>()  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        service.call().sink { [formatter, label] data in  
            let formatted = formatter.format(data: data)  
            label.text = "Retrieved data: \(formatted)"  
        }.store(in: &cancellables)  
    }  
}
```

// Will instances of `ViewController` leak?





```
class ViewController: UIViewController {  
  
    let service = Service()  
    let formatter = Formatter()  
    let label = UILabel()  
  
    var cancellables = Set<AnyCancellable>()  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        service.call().sink { [formatter, label] data in  
            let formatted = formatter.format(data: data)  
            label.text = "Retrieved data: \(formatted)"  
        }.store(in: &cancellables)  
    }  
}
```

// Will instances of `ViewController` leak?

Yes!  

```
class ViewController: UIViewController {  
  
    let service = Service()  
    let formatter = Formatter()  
    let label = UILabel()  
  
    var cancellables = Set<AnyCancellable>()  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        service.call().sink { [formatter, label] data in  
            let formatted = formatter.format(data: data)  
            label.text = "Retrieved data: \(formatted)"  
        }.store(in: &cancellables)  
    }  
}
```

// Will instances of `ViewController` leak?

No!  

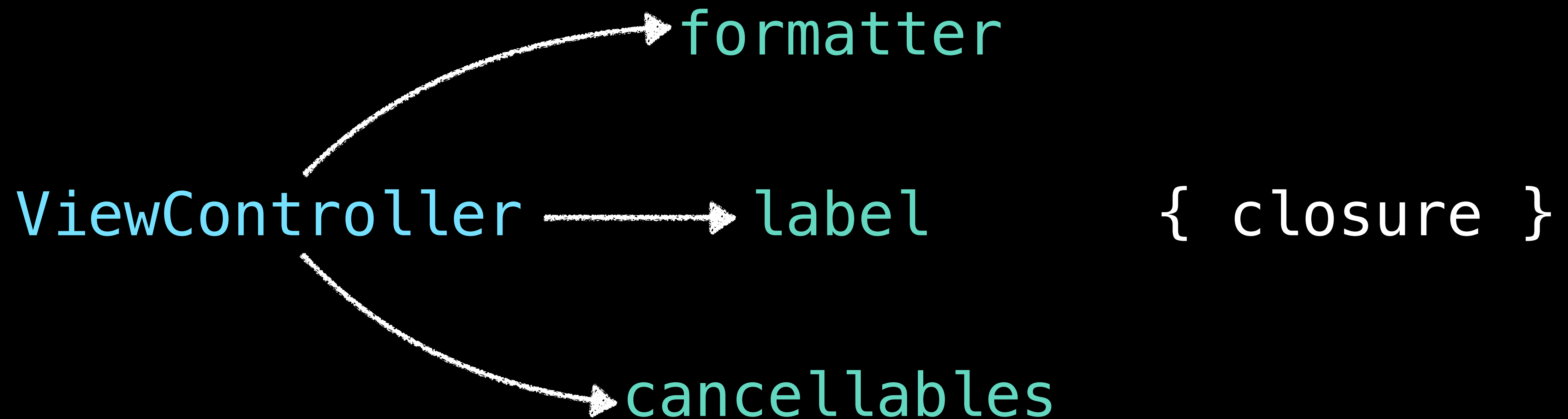
// Will instances of `ViewController` leak?

formatter

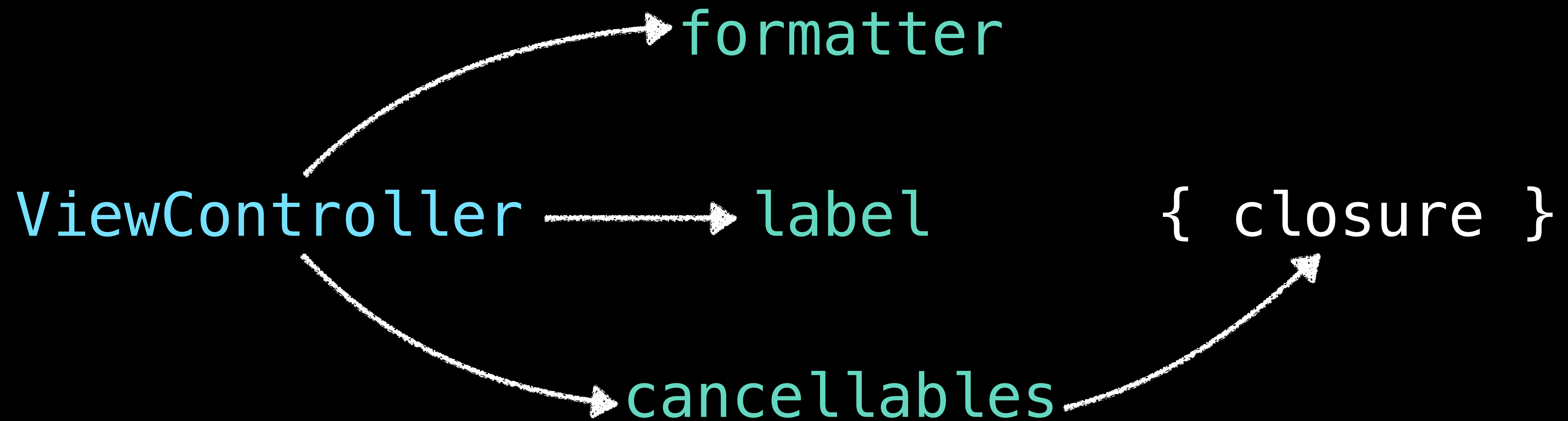
ViewController label { closure }

cancellables

// Will instances of `ViewController` leak?



// Will instances of `ViewController` leak?



// Will instances of `ViewController` leak?



// Will instances of `ViewController` leak?



No cycle here 

```
class ViewController: UIViewController {  
  
    let service = Service()  
    let formatter = Formatter()  
    let label = UILabel()  
  
    var cancellables = Set<AnyCancellable>()  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        service.call().sink { [formatter, label] data in  
            let formatted = formatter.format(data: data)  
            label.text = "Retrieved data: \(formatted)"  
        }.store(in: &cancellables)  
    }  
}
```

// Will instances of `ViewController` leak? No ❌

```
class ViewController: UIViewController {  
  
    let service = Service()  
    let formatter = Formatter()  
    let label = UILabel()  
  
    var cancellables = Set<AnyCancellable>()  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        service.call().sink { [formatter, label] data in  
            let formatted = formatter.format(data: data)  
            label.text = "Retrieved data: \(formatted)"  
        }.store(in: &cancellables)  
    }  
}
```

// Will instances of `ViewController` leak? No ❌

- Capturing properties rather than self inside a closure won't create a retain cycle...

```
class ViewController: UIViewController {  
  
    let service = Service()  
    let formatter = Formatter()  
    let label = UILabel()  
  
    var cancellables = Set<AnyCancellable>()  
  
    override func viewDidLoad() {  
        super.viewDidLoad()  
  
        service.call().sink { [formatter, label] data in  
            let formatted = formatter.format(data: data)  
            label.text = "Retrieved data: \(formatted)"  
        }.store(in: &cancellables)  
    }  
}
```

// Will instances of `ViewController` leak? No ❌

- Capturing properties rather than self inside a closure won't create a retain cycle...
- ...however, if the value of the property changes, the closure will still be using the one it had when the closure was created!

Question #02

Constant Property and Mutating Method


```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
    }  
}
```

```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
    }  
}
```

```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
    }  
}
```

```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
    }  
}
```



```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
    }  
}
```

Yes! 🙋🙋

```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
    }  
}
```

No! 🙅‍♀️ 🙅‍♂️


```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
        self.age += 1 // X  
    }  
}
```

```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
    }  
}
```

```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
    }  
}
```

```
Person(age: 20)
```

```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
    }  
}
```

```
var person = Person(age: 20)
```

```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
    }  
}
```

```
var person = Person(age: 20)
```

```
person.incrementAge()
```

```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
    }  
}
```

```
var person = Person(age: 20)
```

```
person =
```

```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
    }  
}
```

```
var person = Person(age: 20)
```

```
person = Person(age: )
```

```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
    }  
}
```

```
var person = Person(age: 20)
```

```
person = Person(age: person.age + 1)
```



```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
        person = Person(age: person.age + 1)  
    }  
}
```

```
var person = Person(age: 20)
```

```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
        person = Person(age: self.age + 1)  
    }  
}
```

```
var person = Person(age: 20)
```

```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
        self = Person(age: self.age + 1)  
    }  
}
```

```
var person = Person(age: 20)
```

```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
        self = Person(age: self.age + 1)  
    }  
}
```

```
var person = Person(age: 20)
```

```
person.incrementAge()
```

```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
        self = Person(age: self.age + 1)  
    }  
}
```

```
var person = Person(age: 20)
```

```
person.incrementAge() 🎉
```

```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
        self = Person(age: self.age + 1)  
    }  
}
```

```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
        self = Person(age: self.age + 1)  
    }  
}
```

- This technique is known as “re-assigning self”

```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
        self = Person(age: self.age + 1)  
    }  
}
```

- This technique is known as “re-assigning self”

objc ↑↓


```
struct Person {  
    let age: Int  
  
    mutating func incrementAge() {  
        // Can we increment `age` here?  
        self = Person(age: self.age + 1)  
    }  
}
```

objc ↑↓

- This technique is known as “re-assigning self”
- It can result in some tricky code, as it allows assigning a new value without having a “=” operator at the call site

Question #03

Empty Enum


```
enum Empty { }
```

```
func magic<T>(_ empty: Empty) -> T { }
```

```
// Do you think this code builds?
```

```
enum Empty { }
```

```
func magic<T>(_ empty: Empty) -> T { }
```

```
// Do you think this code builds?
```

```
enum Empty { }
```

```
func magic<T>(_ empty: Empty) -> T { }
```

```
// Do you think this code builds?
```

```
enum Empty { }
```

```
func magic<T>(_ empty: Empty) -> T { }
```

```
// Do you think this code builds?
```

```
enum Empty { }
```

```
func magic<T>(_ empty: Empty) -> T { }
```

```
// Do you think this code builds?
```




```
enum Empty { }
```

```
func magic<T>(_ empty: Empty) -> T { }
```

// Do you think this code builds?

Yes! 🙋🙋

```
enum Empty { }
```

```
func magic<T>(_ empty: Empty) -> T { }
```

```
// Do you think this code builds?
```

No! 🙅‍♀️ 🙅‍♂️

```
enum Empty { }
```

```
func magic<T>(_ empty: Empty) -> T { }
```

```
// Do you think this code builds? 
```



```
struct Person {  
    let age: Int  
}
```

```
([] as [Person]).allSatisfy { $0.age > 18 }
```

```
struct Person {  
    let age: Int  
}
```

```
// this evaluates to `true`  
([], as [Person]).allSatisfy { $0.age > 18 }
```



```
enum Empty { }
```

```
func magic<T>(_ empty: Empty) -> T { }
```

```
// Do you think this code builds? 
```

```
enum Empty { }
```

```
enum Never { }
```

```
enum Never { }
```

```
func handle<Value>(result: Result<Value, Never>) {  
    switch result {  
    case .success(let value):  
        print(value)  
//    case .failure(let error):  
//        no need to implement an  
//        impossible code path  
    }  
}
```


`fatalError()`

Summary

Unconditionally prints a given message and stops execution.

Declaration

```
func fatalError(_ message: @autoclosure () -> String = String(),  
file: StaticString = #file, line: UInt = #line) -> Never
```

Parameters

message	The string to print. The default is an empty string.
file	The file name to print with message. The default is the file where <code>fatalError(_:file:line:)</code> is called.
line	The line number to print along with message. The default is the line number where <code>fatalError(_:file:line:)</code> is called.

[Open in Developer Documentation](#)

fatalError()

Summary

Unconditionally prints a given message and stops execution.

Declaration

```
func fatalError(_ message: @autoclosure () -> String = String(),  
file: StaticString = #file, line: UInt = #line) -> Never
```

Parameters

message	The string to print. The default is an empty string.
file	The file name to print with message. The default is the file where <code>fatalError(_:file:line:)</code> is called.
line	The line number to print along with message. The default is the line number where <code>fatalError(_:file:line:)</code> is called.

[Open in Developer Documentation](#)

fatalError()

```
enum Never { }
```

```
enum Never { }
```

- Never is a type that lets us represent impossible codepaths

```
enum Never { }
```

- Never is a type that lets us represent impossible codepaths
- The Swift compiler is smart enough to understand it and adapt the errors and warnings it will emit in consequence

```
enum Never { }
```

- Never is a type that lets us represent impossible codepaths
- The Swift compiler is smart enough to understand it and adapt the errors and warnings it will emit in consequence
- Never can also be helpful when prototyping and trying to make new code build successfully 🙌

Thank You! 🤗

Twitter



YouTube

