(Observable) Object Lifetimes

2022. 12. 28. Stat So

Contents

- **Object Lifetimes and ARC**
- Observable Object lifetimes

Object Lifetimes and ARC

- Object's lifetime begin at init() and ends at last use
- ARC deallocates an object after its lifetime ends
- ARC tracks object's lifetime with reference count
- Swift compiler inserts retain/release operations
- Swift runtime deallocates object with 0 reference count

```swift
class Traveler {
    var name: String
    var destination: String?
}

func test() {
    let traveler1 = Traveler(name: "Lily")
    let traveler2 = traveler1
    traveler2.destination = "Big Sur"
    print("Done traveling")
}
```

At compile time

```
class Traveler {
    var name: String
    var destination: String?

    init(name: String) {
        self.name = name
    }
}

func test() {
    let traveler1 = Traveler(name: "Lily")
    retain (traveler2)
    let traveler2 = traveler1
    release (traveler1)
    traveler2.destination = "Big Sur"
    release (traveler2)
    print("Done traveling")
}
```

At runtime

```
class Traveler {
    var name: String
    var destination: String?

    init(name: String) {
        self.name = name
    }
}

func test() {
    let traveler1 = Traveler(name: "Lily")
    retain
    let traveler2 = traveler1
    release
    traveler2.destination = "Big Sur"
    release
    print("Done traveling")
}
```

Object lifetimes in Swift are use-based

```
class Traveler {
    var name: String
    var destination: String?

    init(name: String) {
        self.name = name
    }
}

func test() {
    let traveler1 = Traveler(name: "Lily")
    let traveler2 = traveler1
    traveler2.destination = "Big Sur"
    -> OBJECT DEALLOCATED HERE
    print("Done traveling")
}
```

- Object lifetime
  - C++ : close brace 까지
  - Swift : last use까지 수명 보장 (minimum, in most cases)
- **ARC 최적화**에 따라서 observed object lifetime은 last use보다 더 길 수 있다.

Observed Object Lifetimes

- ARC 최적화에 따른 객체 수명 변화

- observed object lifetime은 이후에 상세 구현이 업데이트 되면 변경될 가능성이 있음
- observed object lifetime에 의존해서 개발을 하면 지금은 잘 될지 몰라도 나중에 고장날 수 있음

Weak and unowned references

- Do not participate in reference counting
- Break reference cycles

Reference Cycle

```
class Traveler {
    var name: String
    var account: Account?
    func printSummary() {
        if let account = account {
            print("\(name) has \(account.points) points")
        }
    }
}

class Account {
    var traveler: Traveler?
    var points: Int
}

func test() {
    let traveler = Traveler(name: "Lily")
    let account = Account(traveler: traveler, points: 1000)
    traveler.account = account
    traveler.printSummary()
}
```

Reference Cycle -> Weak

```
class Traveler {
    var name: String
    var account: Account?
    func printSummary() {
        if let account = account {
            print("\(name) has \(account.points) points")
        }
    }
}

class Account {
    weak var traveler: Traveler?
    var points: Int
}

func test() {
    let traveler = Traveler(name: "Lily")
    let account = Account(traveler: traveler, points: 1000)
    traveler.account = account
    traveler.printSummary()
}
```

Reference Cycle -> Weak

```
class Traveler {
    var name: String
    var account: Account?
}

class Account {
    weak var traveler: Traveler?
    var points: Int
    func printSummary() {
        print("\(traveler!.name) has \(points) points")
    }
}

func test() {
    let traveler = Traveler(name: "Lily")
    let account = Account(traveler: traveler, points: 1000)
    traveler.account = account
    account.printSummary()
}
```

Reference Cycle -> Weak + Optional Binding

```
class Traveler {
    var name: String
    var account: Account?
}

class Account {
    weak var traveler: Traveler?
    var points: Int
    func printSummary() {
        if let traveler = traveler {
            print("\(traveler.name) has \(points) points")
        }
    }
}

func test() {
    let traveler = Traveler(name: "Lily")
    let account = Account(traveler: traveler, points: 1000)
    traveler.account = account
    account.printSummary()
}
```

- 옵셔널 바인딩을 사용하면..
- 알 수 없는 이유로 lifetime 이 변경 되더라도 크래시가 나지 않기 때문에 문제가 생겼을 때 알아차리기 어렵다 (silent bug)

Weak and unowned references

Safe techniques

withExtendedLifetime()

Redesign to access via strong reference

Redesign to avoid weak/unowned reference

withExtendedLifetime()

```
func test() {
    let traveler = Traveler(name: "Lily")
    let account = Account(traveler: traveler, points: 1000)
    traveler.account = account
    withExtendedLifetime(traveler) {
        account.printSummary()
    }
}
```

Explicitly
extend lifetime

withExtendedLifetime()

```
func test() {
    let traveler = Traveler(name: "Lily")
    let account = Account(traveler: traveler, points: 1000)
    traveler.account = account
    withExtendedLifetime(traveler) {
        account.printSummary()
    }
}
```

Explicitly
extend lifetime

```
func test() {
    let traveler = Traveler(name: "Lily")
    let account = Account(traveler: traveler, points: 1000)
    traveler.account = account
    account.printSummary()
    withExtendedLifetime(traveler) {}
}
```

Explicitly
till end of scope

withExtendedLifetime()

```
func test() {
    let traveler = Traveler(name: "Lily")
    let account = Account(traveler: traveler, points: 1000)
    trav
    with

    }
}
```

```
func test() {
    let traveler = Traveler(name: "Lily")
    let account = Account(traveler: traveler, points: 1000)
    defer { withExtendedLifetime(traveler) {} }
    traveler.account = account
    account.printSummary()
```

Explicitly
till end of scope

```
func tes
    let
    let }
    trav
    account.printSummary()
    withExtendedLifetime(traveler) {}
}
```

Explicitly
till end of scope

withExtendedLifetime()

but, this technique is fragile

- lifetime 관련해서 잠재적 위험이 있는 한, weak을 사용하는 모든 코드에서 withExtendedLifetime() 써줘야 함
- 코드베이스 전반에 퍼질 수 있음 -> 유지보수 비용 증가

Redesign to access via strong reference

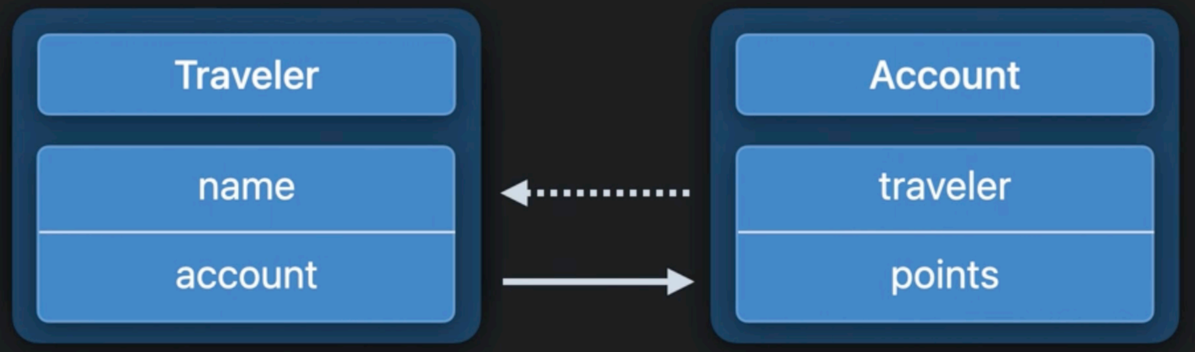
```
class Traveler {
    var name: String
    var account: Account?
    func printSummary() {
        if let account = account {
            print("\(name) has \(account.points) points")
        }
    }
}

class Account {
    private weak var traveler: Traveler?
    var points: Int
}

func test() {
    let traveler = Traveler(name: "Lily")
    let account = Account(traveler: traveler, points: 1000)
    traveler.account = account
    traveler.printSummary()
}
```

Access via strong reference

Limit visibility of weak reference

Redesign to avoid weak/unowned reference

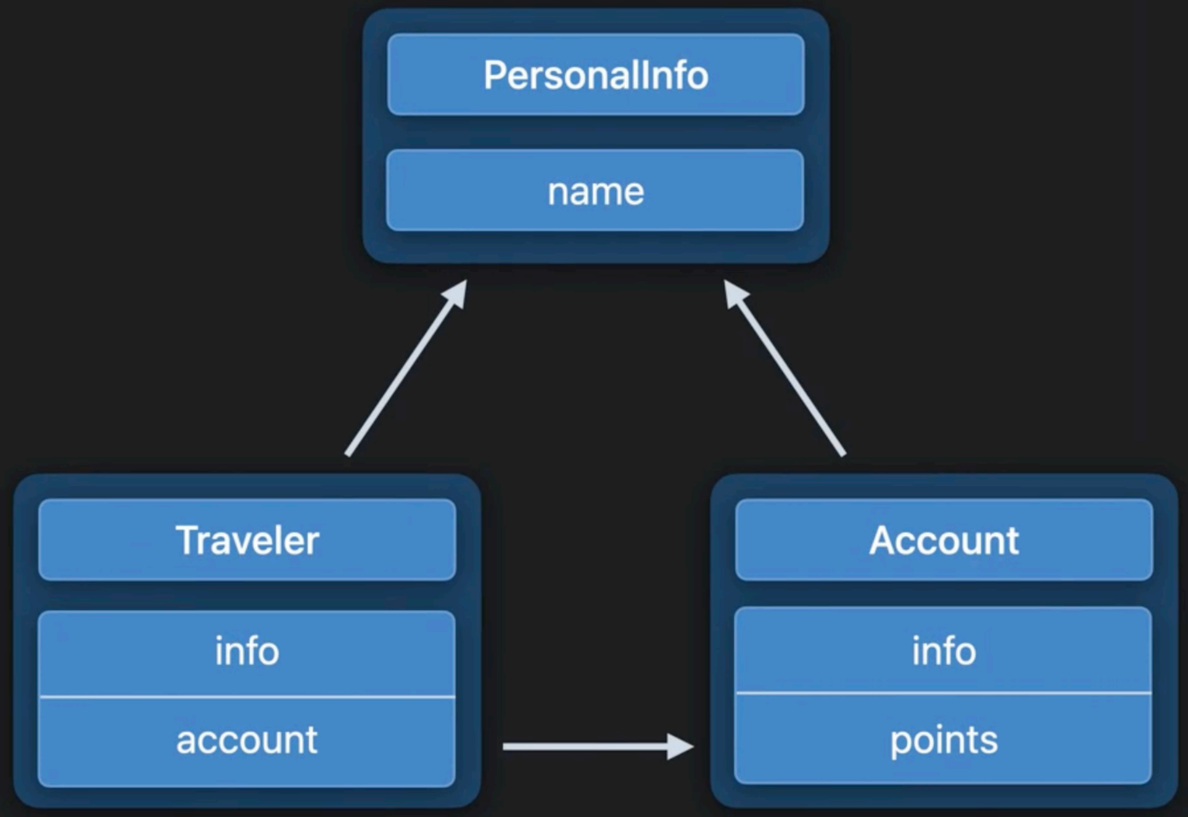
```swift
// Before redesign


class Traveler {
    var name: String
    var account: Account?
}

class Account {
    weak var traveler: Traveler?
    var points: Int
}
```



- 순환 참조를 깨기 위해서 weak을 쓸 수 밖에 없는 구조

Redesign to avoid weak/unowned reference

```swift
// After redesign

class PersonalInfo {
    var name: String
}

class Traveler {
    var info: PersonalInfo
    var account: Account?
}

class Account {
    var info: PersonalInfo
    var points: Int
}
```



- 순환 구조 -> 트리 구조로 변경하여 weak 제거

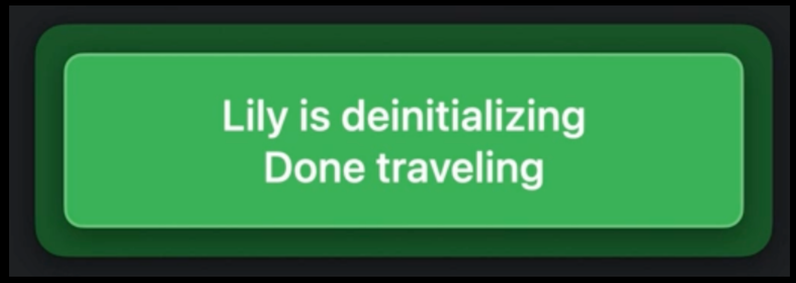
Observable object lifetimes

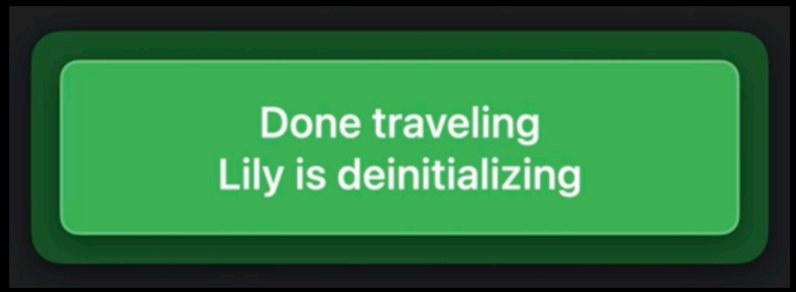
Deinitializer side-effects

- Deinit은 deallocation 전에 실행됨
  - deinit이 lifetime에 영향을 받는다
- Observable side-effects
  - Global side-effects
  - Side-effects on exposed class details

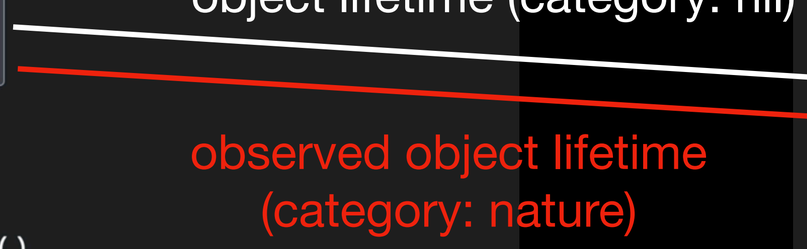
Deinitializer side effect

```
class Traveler {
    var name: String
    var destination: String?

    deinit {
        print("\(name) is deinitialzing")
    }
}

func test() {
    let traveler1 = Traveler(name: "Lily")
    let traveler2 = traveler1
    traveler2.destination = "Big Sur"
    print("Done traveling")
}
```





```swift
class Traveler {
    var name: String
    var id: UInt
    var destination: String?
    var travelMetrics: TravelMetrics
    // Update destination and record travelMetrics
    func updateDestination(_ destination: String) {
        self.destination = destination
        travelMetrics.destinations.append(self.destination)
    }
    // Publish computed metrics
    deinit {
        travelMetrics.publish()
    }
}

class TravelMetrics {
    let id: UInt
    var destinations = [String]()
    var category: String?
    // Finds the most interested travel category based on r
    func computeTravelInterest()
    // Publishes id, destinations.count and travel interest
    func publish()
}
```




```swift
func test() {
    let traveler = Traveler(name: "Lily", id: 1)
    let metrics = traveler.travelMetrics
    ...
    traveler.updateDestination("Big Sur")
    ...
    traveler.updateDestination("Catalina")
    metrics.computeTravelInterest()
}

verifyGlobalTravelMetrics()
```

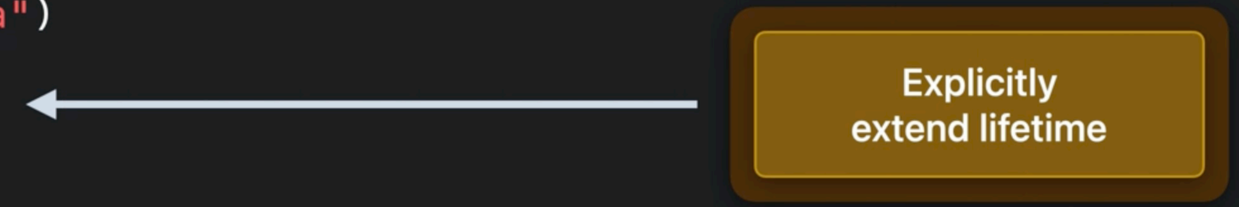
Deinitializer side-effects

Safe techniques

- withExtendedLifetime()
- Redesign to limit visibility of internal class detail
- Redesign to avoid deinitializer side-effects

withExtendedLifetime()

```
func test() {
    let traveler = Traveler(name: "Lily", id: 1)
    let metrics = traveler.travelMetrics
    ...
    traveler.updateDestination("Big Sur")
    ...
    traveler.updateDestination("Catalina")
    withExtendedLifetime(traveler) {
        metrics.computeTravelInterest()
    }
}
```



- 명시적으로 lifetime 연장하는 방법
- 코드 전반에 이런 코드가 들어가면 유지보수 비용이 증가

Redesign to limit visibility of internal class detail

```
class Traveler {
    ...
    private var travelMetrics: TravelMetrics
    deinit {
        travelMetrics.computeTravelInterest()
        travelMetrics.publish()
    }
}

func test() {
    let traveler = Traveler(name: "Lily", id: 1)
    ...
    traveler.updateDestination("Big Sur")
    ...
    traveler.updateDestination("Catalina")
}
```

- 재설계를 통해서 해결하는 방법
- 외부에서 접근하는 경로를 제한하여 사이드 이펙트 가능성을 차단함

Redesign to avoid deinitializer side-effect

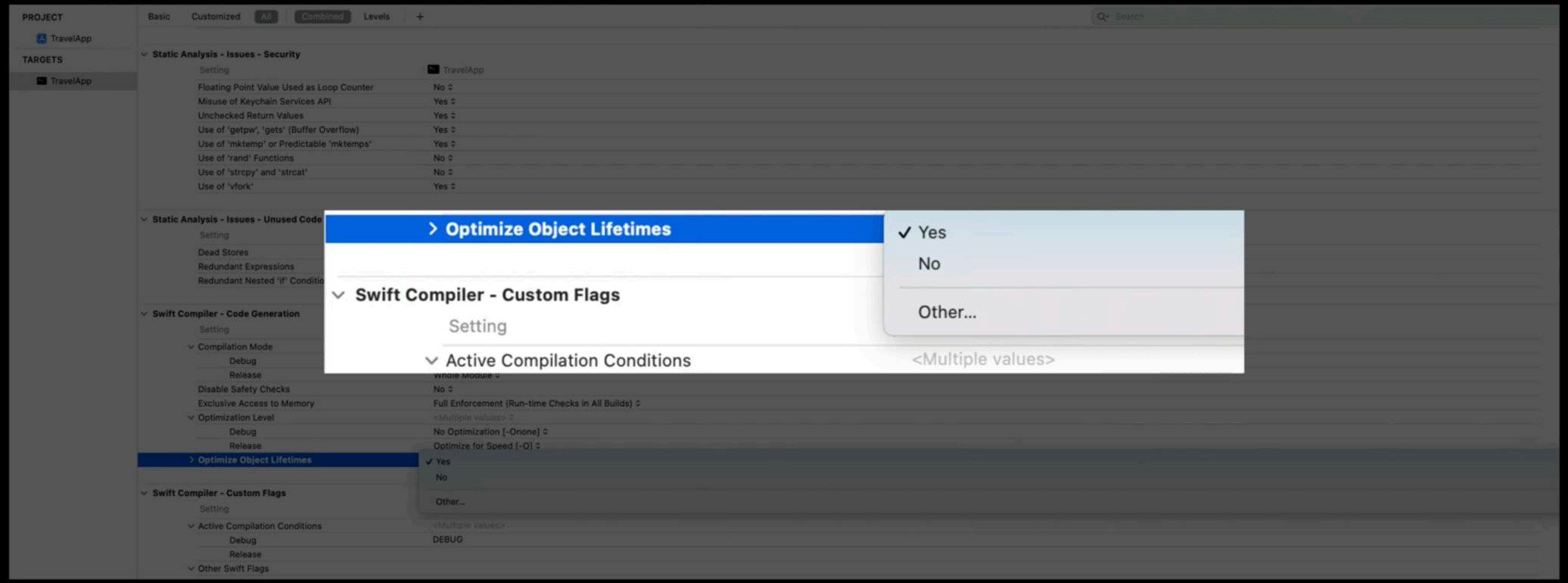
```
class Traveler {
    ...
    private var travelMetrics: TravelMetrics

    func publishAllMetrics() {
        travelMetrics.computeTravelInterest()
        travelMetrics.publish()
    }

    deinit {
        assert(travelMetrics.published)
    }
}

class TravelMetrics {
    ...
    var published: Bool
    ...
}

func test() {
    let traveler = Traveler(name: "Lily", id: 1)
    defer { traveler.publishAllMetrics() }
    ...
    traveler.updateDestination("Big Sur")
    ...
    traveler.updateDestination("Catalina")
}
```

Verify instead of mutating state

defer instead of deinitializer side-effects

Enable powerful lifetime shortening optimization

NEW

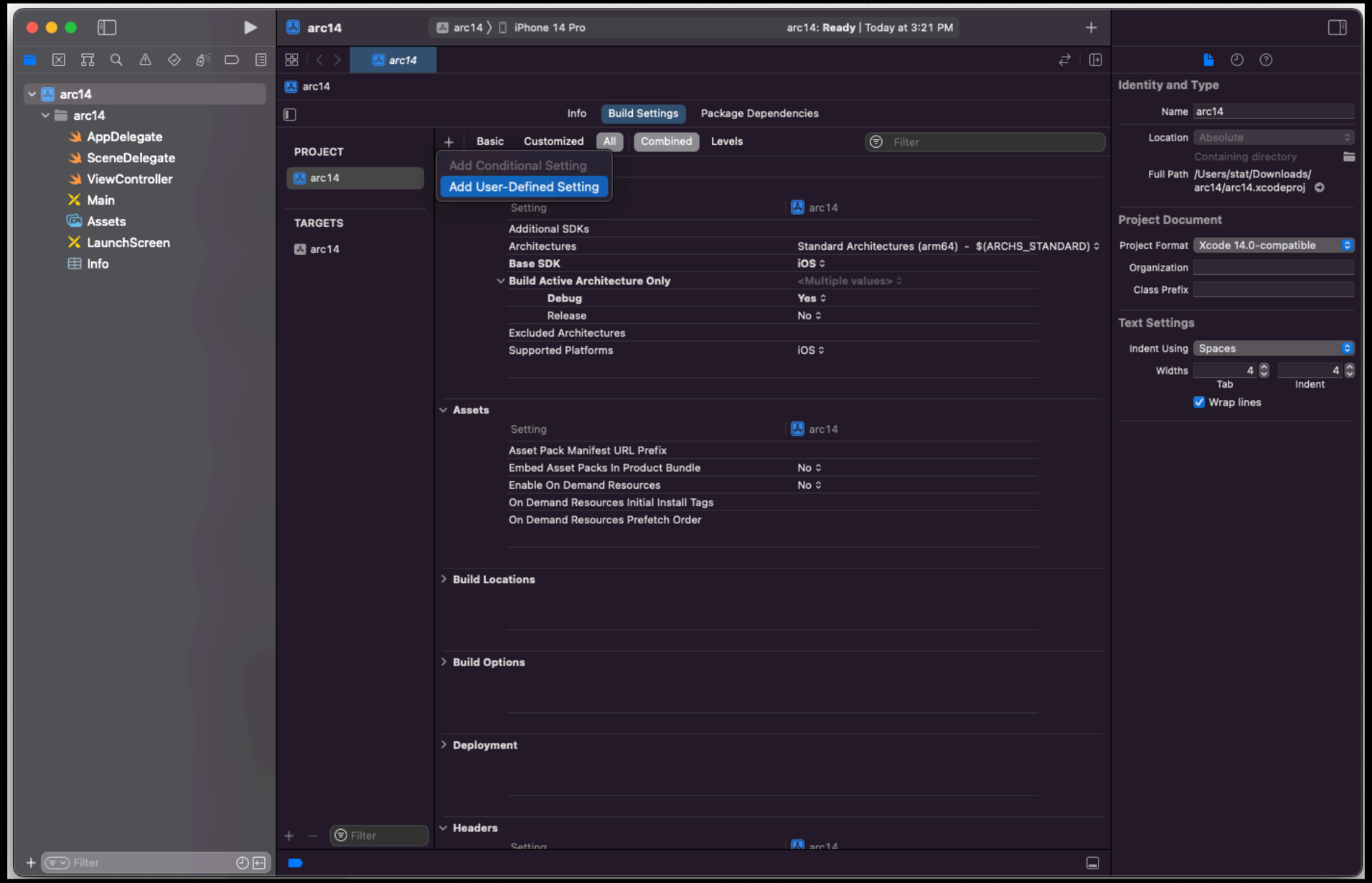


그런데 아무리 찾아봐도 없다.

Xcode 14 Release Notes

Update your apps to use new features, and test your apps against API changes.

- Xcode 13 provided a Swift build setting called “Optimize Object Lifetimes” that’s not available in Xcode 14. If your project already customized this build setting, it now becomes a user-defined setting. It has no effect and you can remove it. Xcode 14 now consistently optimizes object lifetimes. (91971848)


arc14
arc14 〉 iPhone 14 Pro
arc14: Ready | Today at 3:21 PM
arc14
AppDelegate
SceneDelegate
ViewController
Main
Assets
LaunchScreen
Info
Info
Build Settings
Package Dependencies
PROJECT
arc14
TARGETS
arc14
Basic
Customized
All
Combined
Levels
Filter
Add Conditional Setting
Add User-Defined Setting
Setting
Additional SDKs
Architectures
Standard Architectures (arm64) - $(ARCHS_STANDARD)
Base SDK
iOS
Build Active Architecture Only
<Multiple values>
Debug
Yes
Release
No
Excluded Architectures
Supported Platforms
iOS
Assets
Asset Pack Manifest URL Prefix
Embed Asset Packs In Product Bundle
No
Enable On Demand Resources
No
On Demand Resources Initial Install Tags
On Demand Resources Prefetch Order
Build Locations
Build Options
Deployment
Headers
Identity and Type
Name
arc14
Location
Absolute
Containing directory
Full Path
/Users/stat/Downloads/arc14/arc14.xcodeproj
Project Document
Project Format
Xcode 14.0-compatible
Organization
Class Prefix
Text Settings
Indent Using
Spaces
Widths
4
Tab
4
Indent
Wrap lines

Summary

- lifetime != observed lifetime
- observed lifetime에 의존하는 코드는 위험
- observed lifetime에 따라 결과가 달라지는 상황
  - weak / unowned
  - deinit side effect
- 명시적으로 lifetime을 확장하거나 재설계를 통해서 해결