# SWIFT ARC

Bill Kim(김정훈) | ibillkim@gmail.com

# 목차

# Memory Management

언어마다 메모리를 관리하는 방법은 다를 수 있습니다.
메모리 관리 방법은 참조 방식을 통하여 분류할 수 있는데 참조 계산 시점에 따라서 크게 두가지 방법으로 나눠서 생각할 수 있습니다.

* GC(Garbage Collection)

참조 계산 시점 : Run time
장점 : 인스턴스가 해제될 확률이 비교적 높은, 특별히 규칙을 신경쓰지 않아도 됨
단점 : Run tiem에 참조를 계속 추적하므로 추가 리소스 사용 및 성능 저하가 발생할 수 있음, 개발자가 메모리 해제 시점을 예측하기 어려움

# Memory Management

* RC(Reference Counting)

참조 계산 시점 : Complile time
장점 : Complie시 메모리 해제 시점이 결정되므로 개발자가 대략적으로 해제 시점을 예측할 수 있음, Run time에 메모리 해제를 위해 추가 리소스가 발생하지 않음
단점 : 기본적인 규칙과 동작 방식을 알아야 함, 개발자 실수에 의한 순환 참조 등 Instance가 메모리에서 영원히 해제되지 않을 수 있음(메모리 릭)

# MRC vs ARC

애플은 Objective C에서부터 RC에 대한 메모리 관리 방식에 따라서 MRC(Manual Reference Counting), ARC(Automatic Reference Counting) 등의 방식을 사용하고 있습니다.

MRC(Manual Reference Counting)

할당(retain)과 해제(release)를 명시적으로 호출하기 때문에 MRR(Manual Retain-Release) 이라는 이름으로도 불렸습니다.

alloc, new, copy, mutableCopy, retain 등으로 RC 증가
release 로 RC 감소

최종 Reference Count가 0이 되면 매모리 해제

2011년부터 Objective-C의 메모리 관리 방식은 MRC에서 ARC로 대체되었고, 2014년 발표된 Swift도 ARC를 사용한 메모리 관리 방식을 채택했습니다.

# MRC vs ARC

## ARC(Automatic Reference Counting)

컴파일 타임(compile time) 에 기존 MRC 때 개발자가 직접 코드를 작성해야 되던 부분을 자동으로 구문을 분석해서 적절하게 레퍼런스 감소 코드 삽입해 주어, 실행 중에 별도의 메모리 관리가 이루어 지지 않습니다.

메모리 해제에 대해서는 개발자가 신경쓰지 않아도되나 메모리 참조 순환(Reference Cycles)에 대해서는 주의할 필요가 있다.

# How ARC Works

ARC가 동작하는 방식은 아래와 같습니다.

- 클래스의 인스턴스가 만들어질 때 ARC는 메모리를 할당해준다. 할당된 메모리가 가지는 hold 정보는 인스턴스의 타입 정보, 인스턴스와 연관된 stored 프로퍼티의 값들이다.

- 클래스의 인스턴스가 더 이상 필요하지 않을 때 ARC는 할당된 메모리를 해제하여 다른 목적으로 쓰일 수 있도록 해준다. (쓸데없는 메모리 차지를 방지)

- ARC는 아직 사용중인 인스턴스를 해제하지 않도록 주의를 기울인다. 만약 사용중인 인스턴스가 해제 deallocate될 경우 인스턴스의 프로퍼티나 메서드에는 더 이상 접근할 수 없게 되기 때문에 Crash의 위험이 있기 때문이다.

- 따라서 ARC는 클래스 인스턴스를 참조하고 있는 프로퍼티, 상수, 변수들을 추적하고, 만약 "strong" 참조가 하나라도 있다면 해제하지 않는다.

# How ARC Works

참조 카운트와 관련 참조 방식에 따른 규칙을 적용하기 위해서는
Swift에서는 아래의 지시어를 사용할 수 있습니다.

| | strong | weak | unowned |
|---|---|---|---|
| Reference Counting | O | X | X |
| Variable(`var`) | O | O | O |
| Constant(`let`) | O | X | O |
| Optional | O | O | X |
| Non-Optional | O | X | O |
| Memory Release | 명시적으로 `nil` 할당 | auto deinit<br>`nil` 할당 | auto deinit<br>메모리 주소를 계속 갖고 있음 |
| Expected Problem | Strong Reference Cycle<br>Memory Leak | 인스턴스 해제 후 접근하면 `nil` 반환 | 인스턴스 해제 후 접근하면 오류<br>Dangling Pointer |

# How ARC Works

```swift
class Person {
    let name: String

    init(name: String)
    {
        self.name = name
        print("\(name) init")
    }

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

var reference1: Person?
var reference2: Person?
var reference3: Person?

reference1 = Person(name: "찜토끼") // print "찜토끼 init"
// 변수에 클래스의 인스턴스가 할당될 때는 strong 참조
// reference1과 Person의 인스턴스(찜토끼 인스턴스) 사이에 strong 참조가 생김
// reference1 RC = 1

reference2 = reference1 // reference1 RC = 2
reference3 = reference1 // reference1 RC = 3, 찜토끼 인스턴스에 대한 strong 참조는 3개가 됨

reference1 = nil // reference1 RC = 2
reference2 = nil // reference1 RC = 1, 찜토끼 인스턴스에 대한 strong 참조는 1개가 됨

reference3 = nil // reference1 RC = 0, print "찜토끼 deinit"
// 찜토끼 인스턴스는 어떤 strong 참조도 걸려있지 않음
```

# Strong Reference

강한 참조(Strong Reference)

- Strong은 어떠한 설정도 없을 때의 기본값
- Strong은해당 레퍼런스에 대해 강한 참조(strong reference)를 유지 하겠다는 뜻
- 해당 키워드는 레퍼런스 카운트를 증가

# Strong Reference

```
class Sample1 {
    var ref2: Sample2?

    init() { print("Sample1 init") }
    deinit { print("Sample1 deinit") }
}

class Sample2 {
    var ref1: Sample1?

    init() { print("Sample2 init") }
    deinit { print("Sample2 deinit") }
}

var sample1: Sample1? = Sample1()  // Sample1 RC 1
var sample2: Sample2? = Sample2()  // Sample2 RC 1

sample1?.ref2 = sample2            // Sample2 RC 2
sample2?.ref1 = sample1            // Sample1 RC 2
sample1 = nil                      // Sample1 RC 1
sample2 = nil                      // Sample2 RC 1

// Print Sample1 init
// Print Sample2 init
```

# Strong Reference

앞서 예제에서는 강한 순환 참조(Strong Reference Cycle)로 인하여 인스턴스가 정상적으로 해제되지 않는 현상이 발생하였습니다.

강한 순환 참조(Strong Reference Cycle)란 서로 다른 인스턴스가 서로를 강하게 참조하고 있어서 참조 횟수를 0으로 만들지 못하고 영원히 메모리에서 해제되지 않는 현상을 말합니다.

그러한 강한 순환 참조를 피하기 위하여 약한 참조 및 미소유 참조 방식을 사용하여 메모리릭을 방지할 수 있습니다.

# Weak Reference

약한 참조(Weak Reference)

약한 참조는 instance를 참조할 때 RC를 증가시키지 않으면서 인스턴스를 참조하도록 합니다.

만약 참조하고 있던 인스턴스가 메모리에서 해제되면 자동으로 nil이 할당되어 참조를 해제하고 메모리에서 반환합니다.

프로퍼티 선언 이후 nil을 할당해야 하기 때문에 약한 참조로 선언된 프로퍼티는 항상 옵셔널 타입의 변수여야 합니다.

# Weak Reference

아까 강한 참조 방식과 달리 Sample2의 ref1 함수를 weak 참조 방식으로 선언하면 아래와 같이 메모리가 해제되는 것을 볼 수 있습니다.

```
class Sample1 {
    var ref2: Sample2?

    init() { print("Sample1 init") }
    deinit { print("Sample1 deinit") }
}

class Sample2 {
    weak var ref1: Sample1?

    init() { print("Sample2 init") }
    deinit { print("Sample2 deinit") }
}
```

# Weak Reference

```
var sample1: Sample1? = Sample1()  // Sample1 RC 1
var sample2: Sample2? = Sample2()  // Sample2 RC 1

sample1?.ref2 = sample2            // Sample2 RC 2
sample2?.ref1 = sample1            // Sample1 RC 2
sample1 = nil                      // Sample1 RC 1
sample2 = nil                      // Sample2 RC 1

// Print Sample1 init
// Print Sample2 init
// Print Sample1 deinit
// Print Sample2 deinit
```

# Unowned Reference

## 미소유 참조(Unowned Reference)

미소유 참조도 약한 참조와 마찬가지로 RC를 증가시키지 않으면서 인스턴스를 참조합니다.

하지만 미소유 참조는 인스턴스를 참조하는 도중에 해당 인스턴스가 메모리에서 사라질 일이 없다고 가정하기 때문에 약한 참조와 달리 암묵적으로 옵셔널을 해제(!)하여 선언합니다.

미소유 참조는 참조하고 있던 인스턴스가 먼저 메모리에서 해제될 때 nil을 할당할 수 없어 오류를 발생시키므로 위험한 방법입니다.

일반적으로 약한 참조만 사용해도 문제가 없기 때문에, 특별한 경우가 아니면 미소유 참조는 사용하지 않는 것이 좋을 것 같습니다.

# Unowned Reference

```
class Sample2 {
    weak var ref1: Sample1?
    unowned var unownedRef: Sample1!

    init() { print("Sample2 init") }
    deinit { print("Sample2 deinit") }
}
```

# Unowned Reference

아래의 소스 코드는 약한 참조와 미소유 참조를 통하여 강한 순환 참조(메모리 릭)를 피하는 방법에 대한 예제 소스입니다.

```
class Person {
    var car: Car?

    init() { print("Person init") }
    deinit { print("Person deinit") }
}

class Car {
    var owner: Person!

    init() { print("Car init") }
    deinit { print("Car deinit") }
}

var bill: Person? = Person()
var genesis: Car? = Car()

bill?.car = genesis
genesis?.owner = bill

bill = nil
genesis = nil

// Person init
// Car init
```

# Unowned Reference

약한 참조 방식을 사용하여 강한 순환 참조(메모리 릭) 방지

```
class Person {
    weak var car: Car?

    init() { print("Person init") }
    deinit { print("Person deinit") }
}

class Car {
    var owner: Person!

    init() { print("Car init") }
    deinit { print("Car deinit") }
}

var bill: Person? = Person()
var genesis: Car? = Car()

bill?.car = genesis
genesis?.owner = bill

bill = nil
genesis = nil

// Person init
// Car init
// Car deinit
// Person deinit
```

# Unowned Reference

**미소유 참조** 방식을 사용하여 강한 순환 참조(메모리 릭) 방지

```
class Person {
    var car: Car?

    init() { print("Person init") }
    deinit { print("Person deinit") }
}

class Car {
    unowned var owner: Person!

    init() { print("Car init") }
    deinit { print("Car deinit") }
}

var bill: Person? = Person()
var genesis: Car? = Car()

bill?.car = genesis
genesis?.owner = bill

bill = nil
genesis = nil

// Person init
// Car init
// Person deinit
// Car deinit
```

# Strong Reference Between Class and Closure

강한 참조 순환 문제는 클래스 인스턴스와 그 인스턴스의 프로퍼티에 할당된 클로저 사이에서도 발생할 수 있습니다.

클로저도 클래스와 마찬가지로 참조 타입이기 때문에 클로저를 프로퍼티에 할당할 때는 참조를 할당하게 되고, 그래서 strong reference cycle이 생길 수 있는 것입니다.

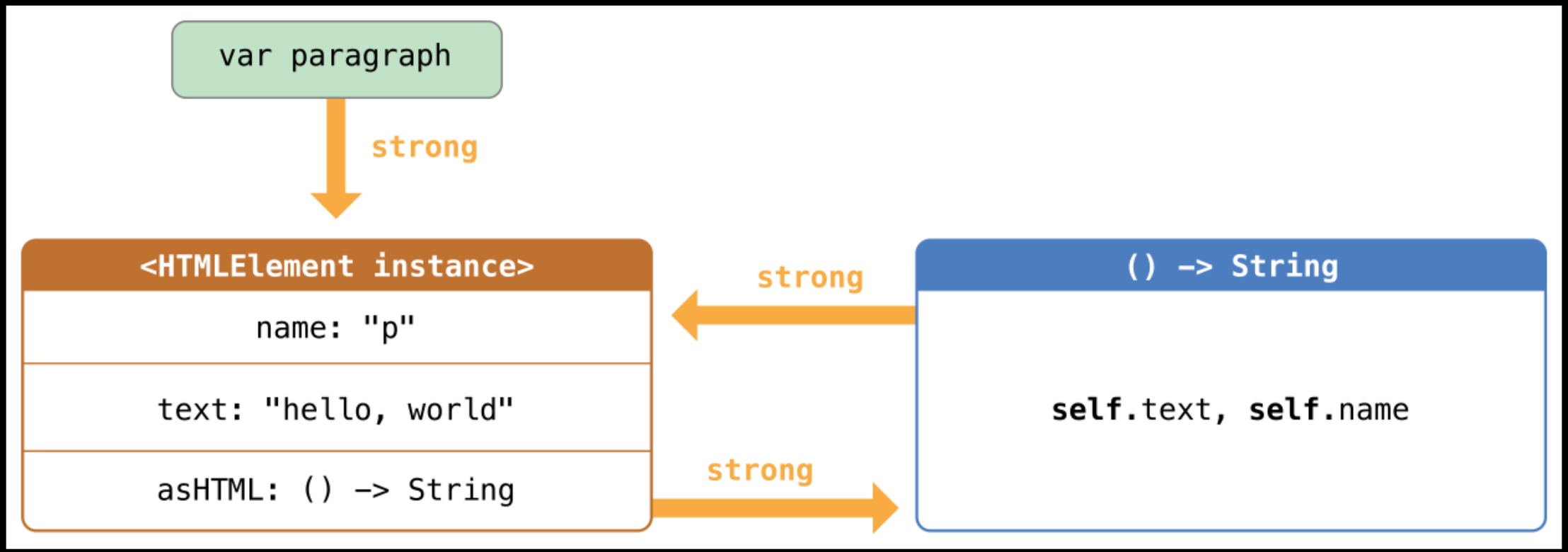
# Strong Reference Between Class and Closure

```swift
class HTMLElement {
    let name: String
    let text: String?
    lazy var asHTML: () -> String = {
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }
    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }
    deinit {
        print("\(name) is being deinitialized")
    }
}


var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
print(paragraph!.asHTML()) // Prints "<p>hello, world</p>"

paragraph = nil // Not Prints "p is being deinitialized"
```

# Strong Reference Between Class and Closure



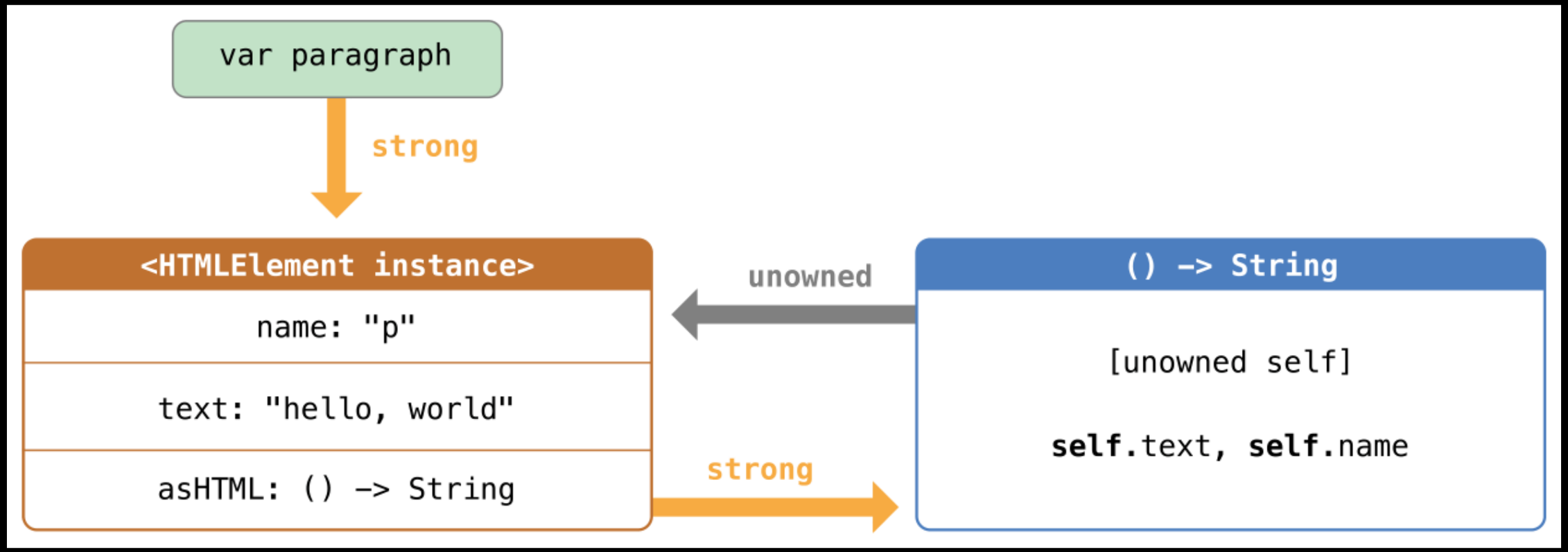
# Strong Reference Between Class and Closure

```
class HTMLElement {
    let name: String
    let text: String?
    lazy var asHTML: () -> String = {
        [unowned self] in
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }
    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }
    deinit {
        print("\(name) is being deinitialized")
    }
}


var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
print(paragraph!.asHTML()) // Prints "<p>hello, world</p>"

paragraph = nil // Prints "p is being deinitialized"
```

# Strong Reference Between Class and Closure



# References


[1] [iOS Swift] RC, ARC 와 MRC 란? 그리고 Strong, Weak, Unowned 는? : https://medium.com/@jang.wangsu/ios-swift-rc-arc-와-mrc-란-그리고-strong-weak-unowned-는-간단하게-적어봤습니다-988a293c04ac

[2] 자동 참조 카운트 (Automatic Reference Counting) : https://jusung.gitbook.io/the-swift-language-guide/language-guide/23-automatic-reference-counting

[3] [Swift]Automatic Reference Counting 정리 : http://minsone.github.io/mac/ios/swift-automatic-reference-counting-summary

[4] Swift ARC : https://hcn1519.github.io/articles/2018-07/swift_automatic_reference_counting

[5] [Swift] 메모리 관리와 ARC : https://velog.io/@cskim/ARCAutomatic-Reference-Counting

# References


[6] Swift ARC에 대해서(1) : https://hyerios.tistory.com/32

[7] Swift - ARC와 메모리 관리 : https://www.slideshare.net/LeeDaheen/swift-arc-152213706

[8] [swift] ARC(Automatic Reference Counting) : https://zetal.tistory.com/entry/swift-Automatic-Reference-Counting

[9] [Swift] 메모리 관리 ARC : http://jhyejun.com/blog/memory-management-arc

[10] Automatic Reference Counting (ARC) : https://wlaxhrl.tistory.com/22

Thank you!