

---

# ARC

---

강사 주영민

# 메모리 관리 방식

ARC : Auto Reference count

- 메모리 관리 방법

- 명시적 해제 : 모든것을 개발자가 관리함 할당과 해제를 해야 함  
해제를 하지 않을 경우 memory leak 발생
- 가비지컬렉터 : 가비지 컬렉터가 수시로 확인해서 안쓰는 객체를 해제 시킴 (시스템 부하) Garbage collector가 돌아 다니며 함.  
좋은점 : memory에 대해서 신경 쓸 것이 없음  
단점 : 컨트롤을 할 수 없음
- 레퍼런스 카운팅 : 오너십 정책에 의해 객체의 해제 정의  
숫자를 셈  
참조 카운트를 세고, 참조 카운트가 0이 되는 순간 사라짐  
카운팅을 개발자가 해야 함  
오너십 정책? 소유권과 사용권을 주는 것, 사용권을 주는 것

# Reference counting

---

```
NSString *str1 = [[NSString alloc] init];  
NSString *str2 = [str1 retain];  
NSString *str3 = str2;
```

alloc : instance 생성  
init : 초기화

refcount = 1  
refcount = 2

```
[str1 release];  
[str2 release];
```

reference count를 해제함

refcount = 1

reference count를 해제함

refcount = 0

# Reference counting

---

```
- (id)init{
    retainCount = 1;
}

- (id)retain{
    retainCount += 1;
    return self;
}

- (void)init{
    retainCount -= 1;
    if(retainCount == 0)
        [self dealloc];
}
```

# 무엇이 문제 일까요?

---

```
NSString *str1 = [[NSString alloc] init];  
NSString *str2 = [[NSString alloc] init];  
NSString *str3 = [[NSString alloc] init];  
str2 = [[NSString alloc] init];  
  
[str1 release];  
[str2 release];  
[str3 release];
```

# Ownership Policy

---

- 인스턴스 객체의 오너만이 해당 인스턴스의 해제에 대해서 책임을 진다.
- 오너십을 가진 객체만 reference count가 증가 된다.

- 2011년 WWDC에서...

# Memory Management is Harder Than It Looks...

- Instruments
  - Allocations, Leaks, Zombies
- Xcode Static Analyzer
- Heap
- ObjectAlloc
- vmmap
- MallocScribble
- debugger watchpoints
- ...and lots more





- 애플은 ARC 도입 이유

---

- 앱의 비정상 종료 원인 중 많은 부분이 메모리 문제. 메모리 관리 는 애플의 앱 승인 거부(Rejection)의 대다수 원인 중 하나.
- 많은 개발자들이 수동적인 (retain/release) 메모리 관리로 힘들어함.
- retain/release 로 코드 복잡도가 증가.

# Welcome to ARC!

- Automatic **Object** memory management
  - Compiler synthesizes retain/release calls
  - Compiler obeys and enforces library conventions
  - Full interoperability with retain/release code
- New runtime features:
  - Zeroing weak pointers
  - Advanced performance optimizations
  - Compatibility with Snow Leopard and iOS4

# Welcome to ARC

---

- ARC는 Automatic Reference Counting의 약자로 기존에 수동 (MRC라고 함)으로 개발자가 직접 retain/release를 통해 reference counting을 관리해야 하는 부분을 자동으로 해준다.

# What ARC Is Not...

- No new runtime memory model
- No automation for malloc/free, CF, etc.
- No garbage collector
  - No heap scans
  - No whole app pauses
  - No non-deterministic releases



# ARC 규칙

---

- retain, release, retainCount, autorelease, dealloc을 직접 호출할 수 없다.
- 구조체내의 객체 포인터를 사용할 수 없다.
- id나 void \* type을 직접 형변환 시킬 수 없다.
- NSAutoreleasePool 객체를 사용할 수 없다.

# Rule #1/4: No Access to Memory Methods

- Memory mangement is part of the language
  - Cannot call retain/release/autorelease...
  - Cannot implement these methods

```
while ([x retainCount] != 0)  
    [x release];
```

Broken code, Anti-pattern

- Solution
  - The compiler takes care of it
  - NSObject performance optimizations
  - Better patterns for singletons

# Rule #2/4: No Object Pointers in C Structs

- Compiler must know when references come and go
  - Pointers must be zero initialized
  - Release when reference goes away
- Solution: Just use objects
  - Better tools support
  - Best practice for Objective-C

```
struct Pair {  
    NSString *Name; // retained!  
    int Value;  
};
```

```
Pair *P = malloc(...);  
...  
...  
free(P); // Must drop references!
```



## Rule #3/4: No Casual Casting id ↔ void\*

- Compiler must know whether void\* is retained
- New CF conversion APIs
- Three keywords to disambiguate casts

```
CFStringRef W = (__bridge CFStringRef)A;  
NSString *X   = (__bridge NSString *)B;  
CFStringRef Y = (__bridge_retain CFStringRef)C;  
NSString *Z   = (__bridge_transfer NSString *)D;
```



# Rule #4/4: No NSAutoreleasePool

- Compiler must reason about autoreleased pointers
- NSAutoreleasePool not a real object—cannot be retained

```
int main(int argc, char *argv[]) {  
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];  
    int retVal = UIApplicationMain(argc, argv, nil, nil);  
    [pool release];  
    return retVal;  
}
```

```
int main(int argc, char *argv[]) {  
    @autoreleasepool {  
        return UIApplicationMain(argc, argv, nil, nil);  
    }  
}
```

Works in all modes!

# 새로운 지시어

---

compile time에 ARC를 위한 code가 생성됨

- Strong      자동으로 소유권이 감, ref count 줄어드는
- weak      참조권이 감

# strong 객체 선언

---

```
@property(strong) Person *p1;
```

```
@property(strong) Person *p2;
```

---

강한 참조 객체 선언

```
var p1:Person
```

```
var p2:Person
```

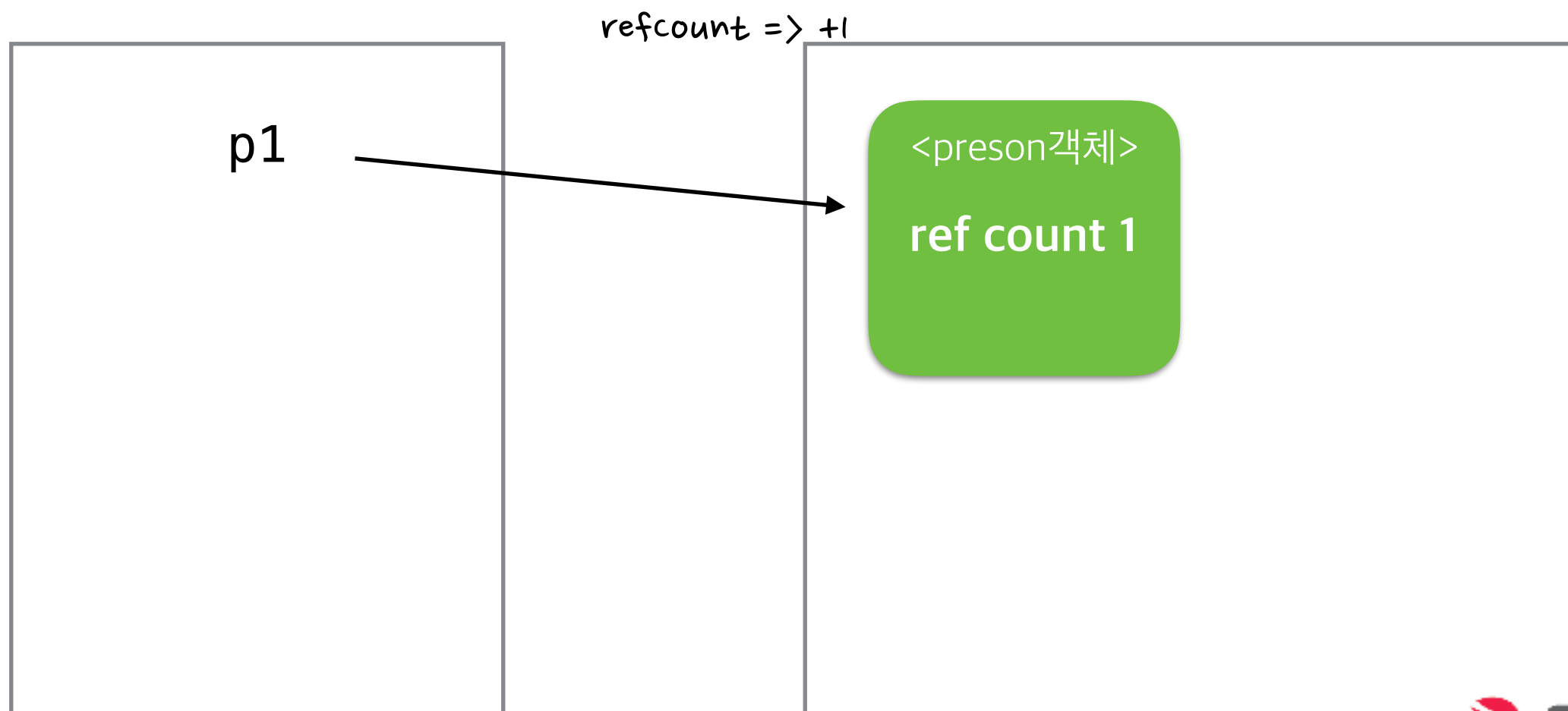
# 할당

---

```
p1 = [[Person alloc] init];
```

```
p1 = Person()
```

객체 할당



# 할당

```
p1 = [[Person alloc] init];
```

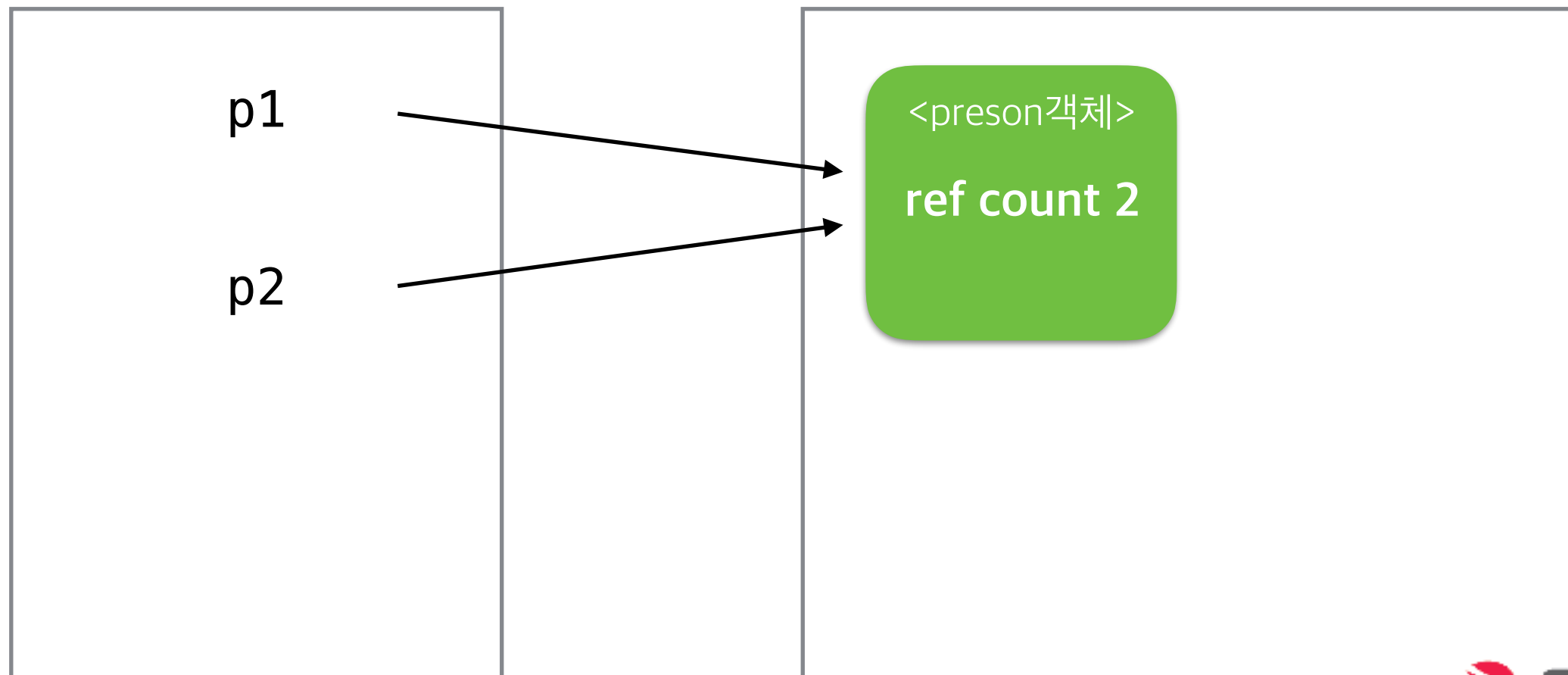
```
p1 = Person()
```

객체 할당

```
p2 = p1
```

RefCount -> +2

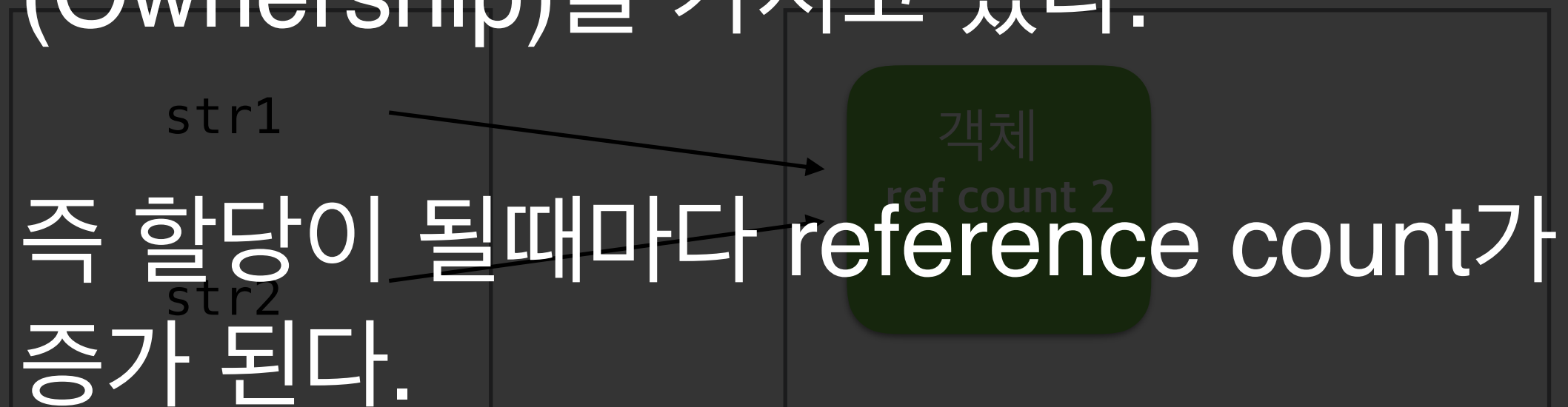
참조가 끊기거나, 생명주기에 따라 사라짐



# 할당

```
str1 = [[NSString alloc] init]
```

두 변수는 **strong** 지시어로 만들었기 때문에  
객체에 대한 참조 포인트와 소유권  
(Ownership)을 가지고 있다.



# weak 객체 선언

---

```
@property(strong) Person *p1;
```

```
@property(weak) Person *p2;
```

---

강한 참조 객체 선언!

```
var p1:Person
```

```
weak var p2:Person
```

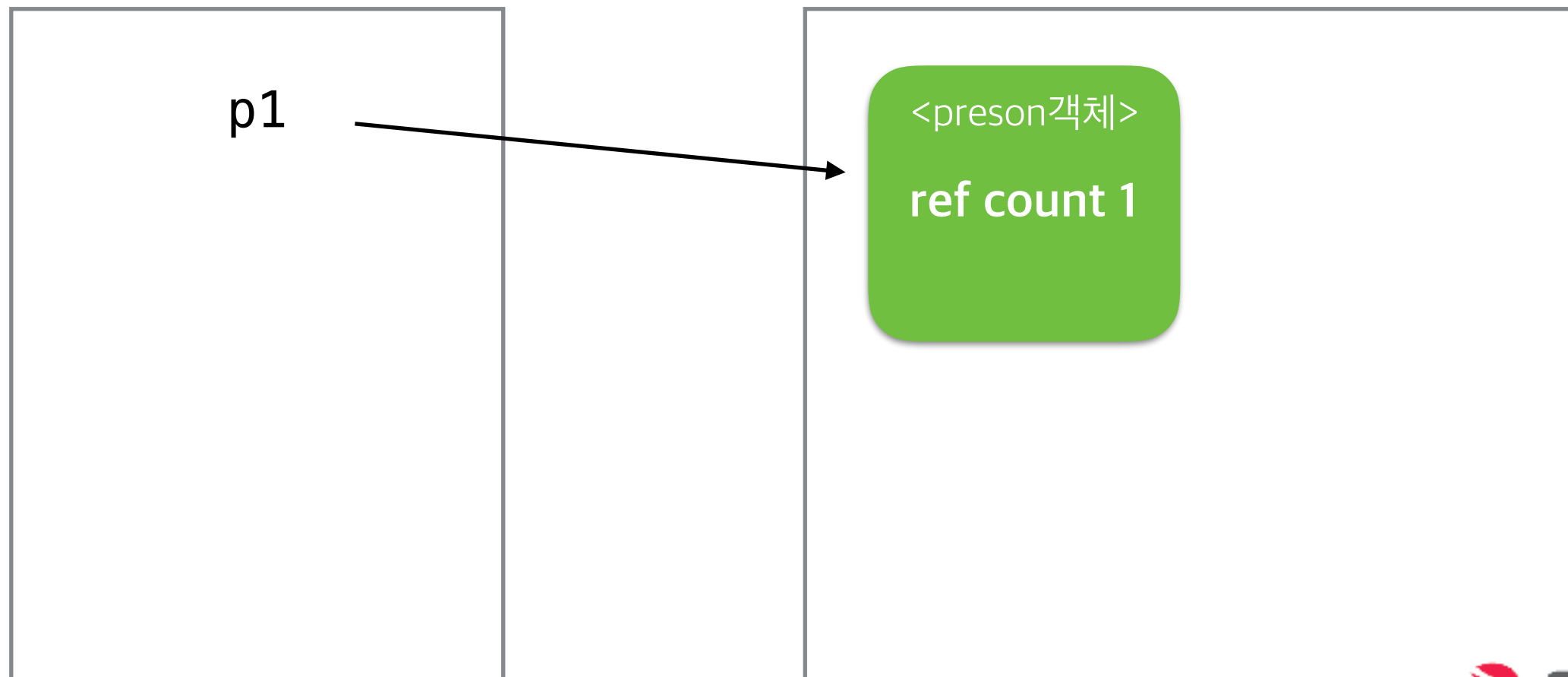
# 할당

---

```
p1 = [[Person alloc] init];
```

```
p1 = Person()
```

객체 할당





# 할당

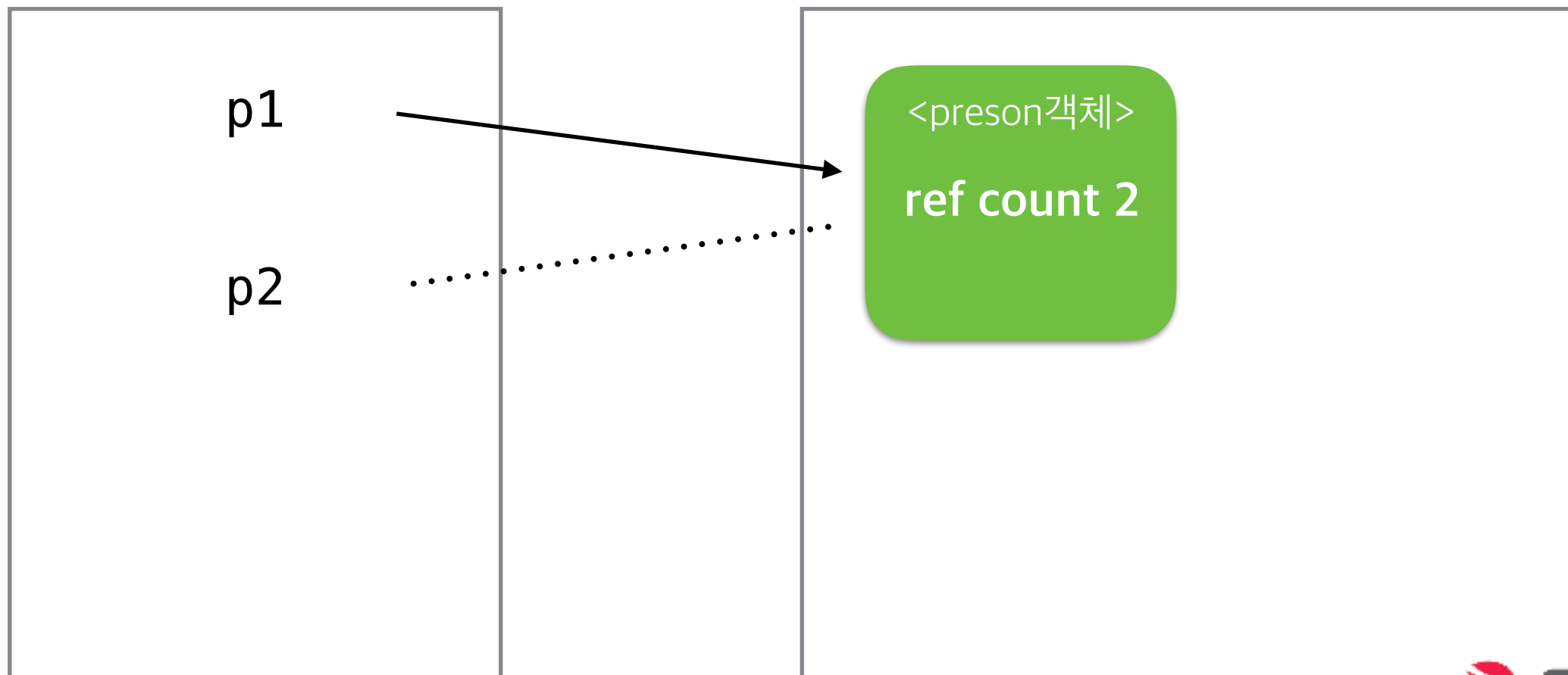
weak로 만들어진 것은 nil이 될 가능성이 있기 때문에 optional이다.

```
p1 = [[Person alloc] init];
```

```
p1 = Person()
```

객체 할당

```
p2 = p1
```



p1은 strong 지시어로 만들었기 때문에 객체에 대한 참조 포인트와 소유권(Ownership)을 가지고 있지만 p2는 약한 참조로 소유권은 없이 참조를 할 수 있는 권한만 있다.

즉 p2가 참조해도 reference count는 증가하지 않는다.



# weak - 할당

---

p2 = Person()

---

객체 할당???

만약 약한 참조로 만든 p2에 객체를 만들고 할당을 한다면?

# weak - 할당

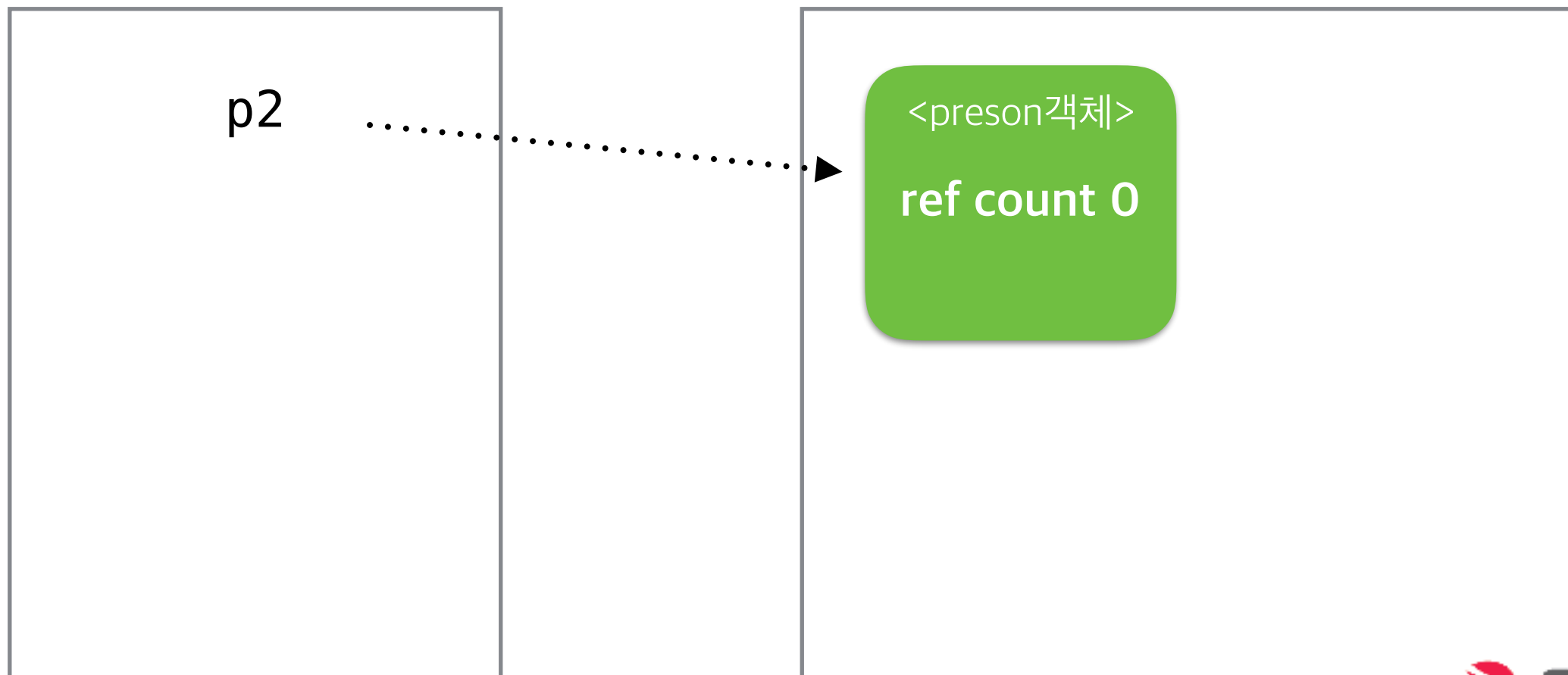
참조조건 : reference

```
p2 = Person()
```

UI에 대해서는 weak으로 가는 것이 좋다  
view에서 ref-count가 1을 갖고 있다. 화면 전환 등으로 사라지게 되면  
사라져야 하기 때문에 weak으로 가는 것이 좋다

## 객체 할당

Multi-thread환경에서는 메모리 참조가 중요하다  
closure 환경  
예) Network, capture에 관한 메모리 관리



# weak - 할당

---

```
str2 = [[NSString alloc] init];
```

p2는 소유권이 없어 reference count를 증가시킬수 없고, reference count가 0인 객체는 자동으로 해제되기 때문에 ...p2는 곧 바로 nil값을 가지게 된다.

# weak - 할당

---

```
p2 = Person()
```

---

객체 할당

```
p2 = nil
```

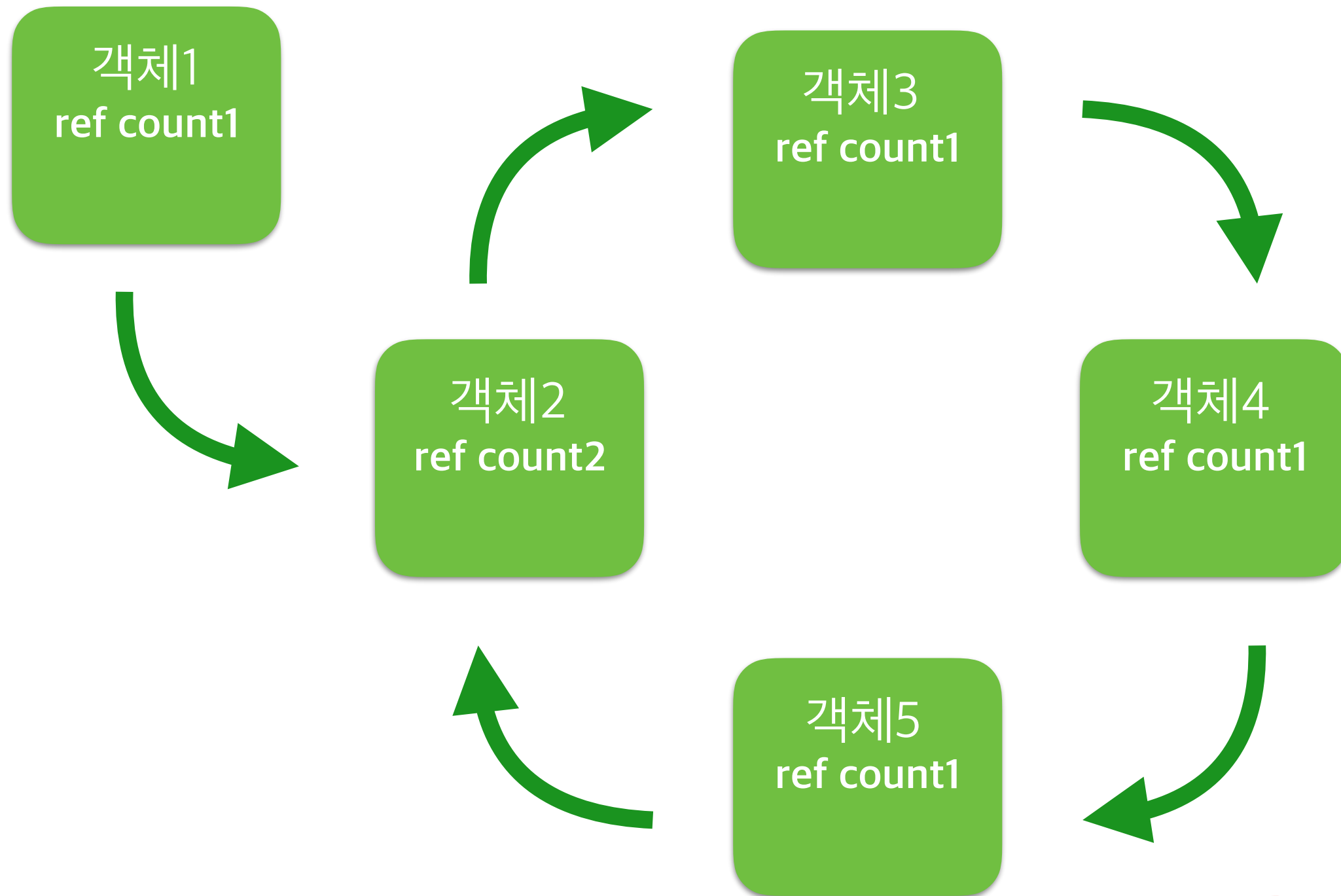


소멸

“반푼이 weak!! 왜 사용하는 것인가??”

# 순환 참조

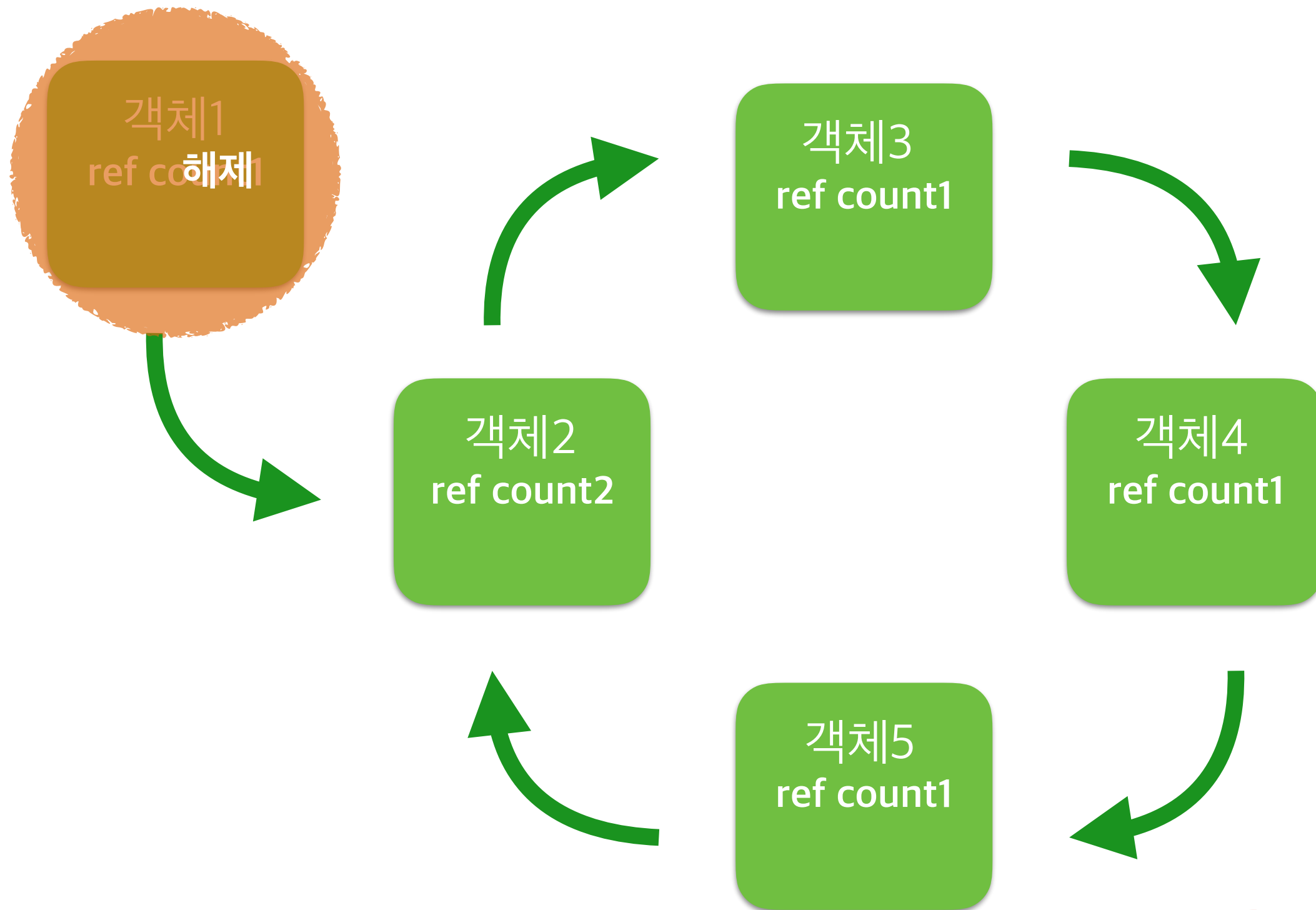
---





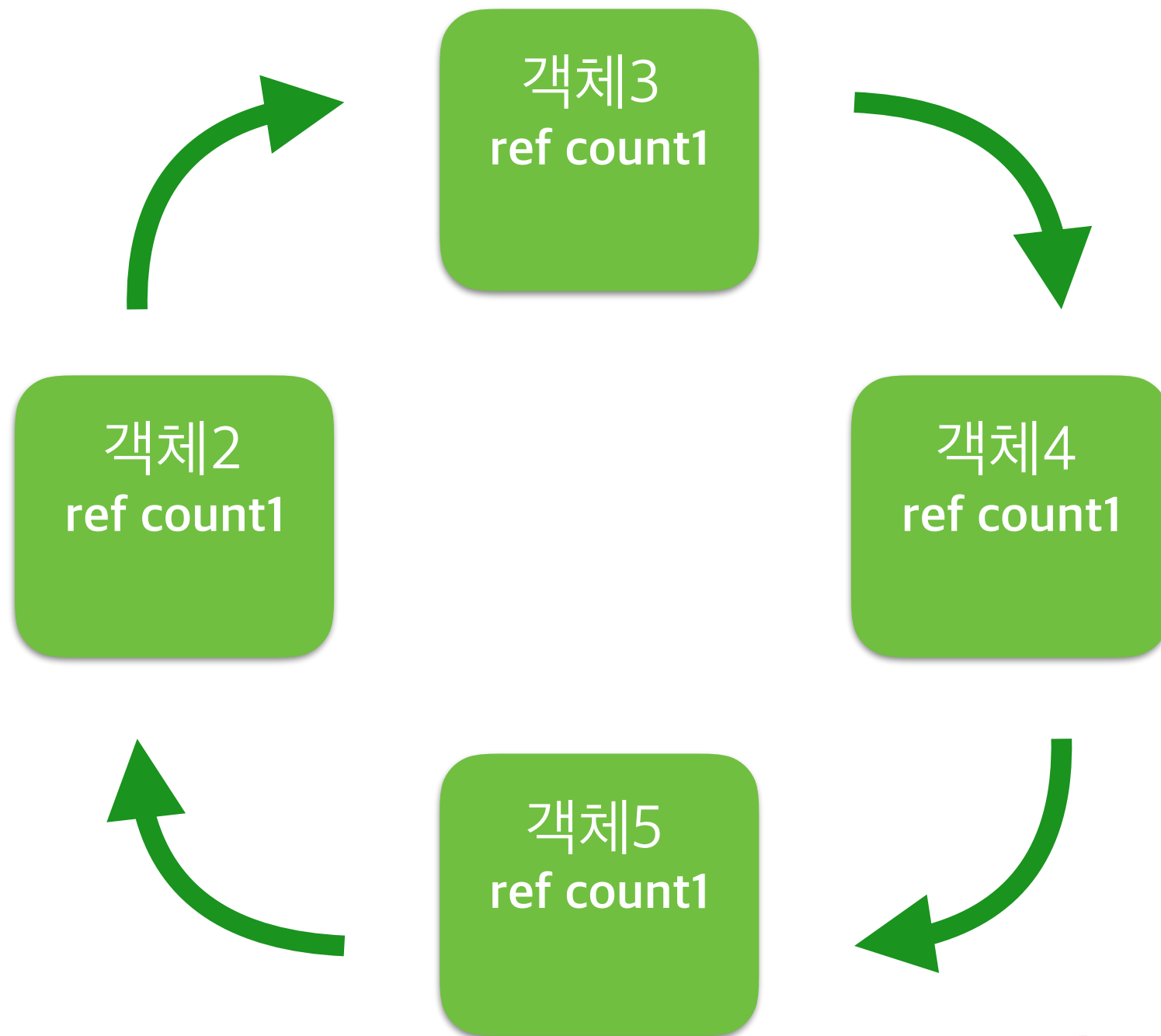
# 순환 참조

---



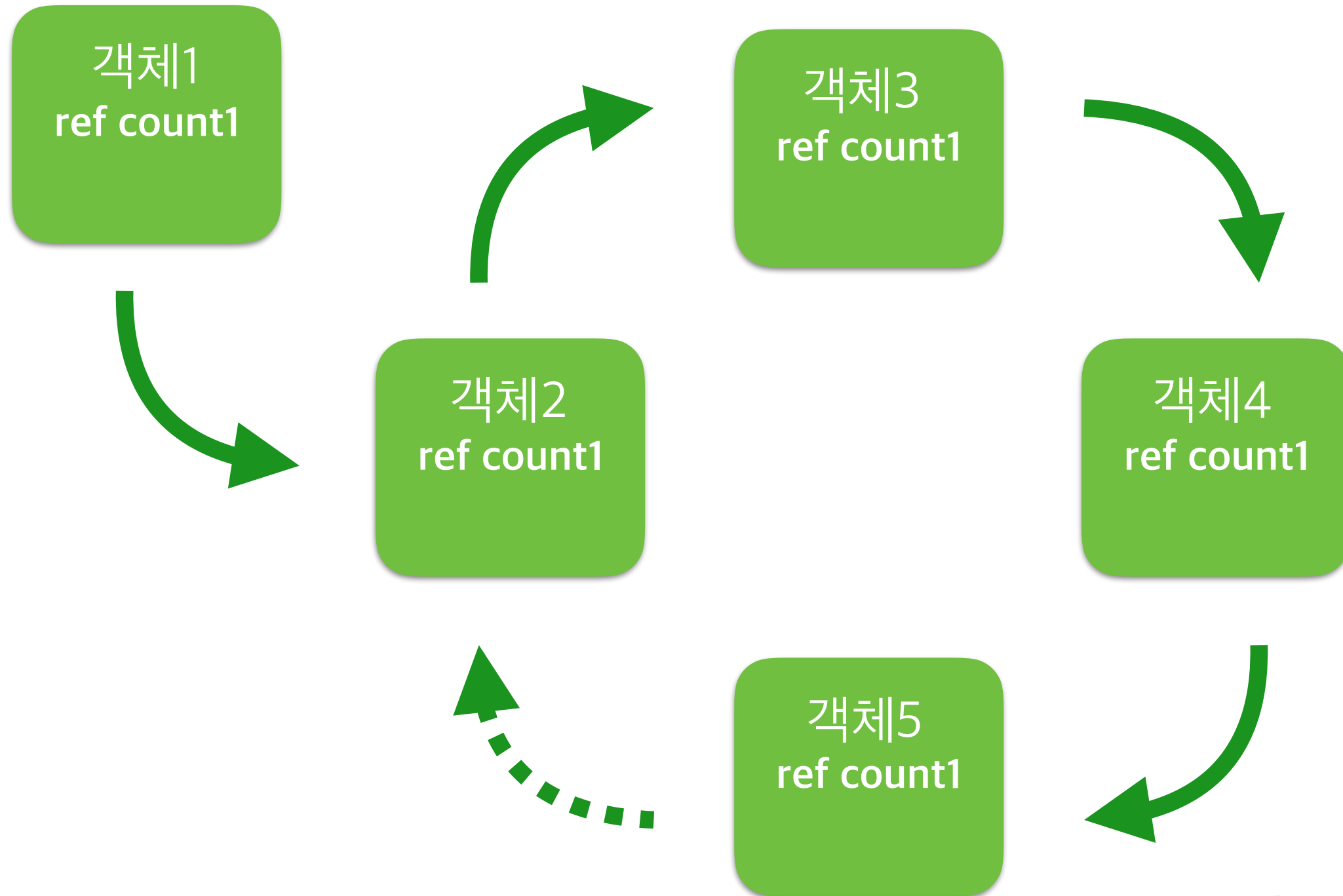
# 순환 참조

---



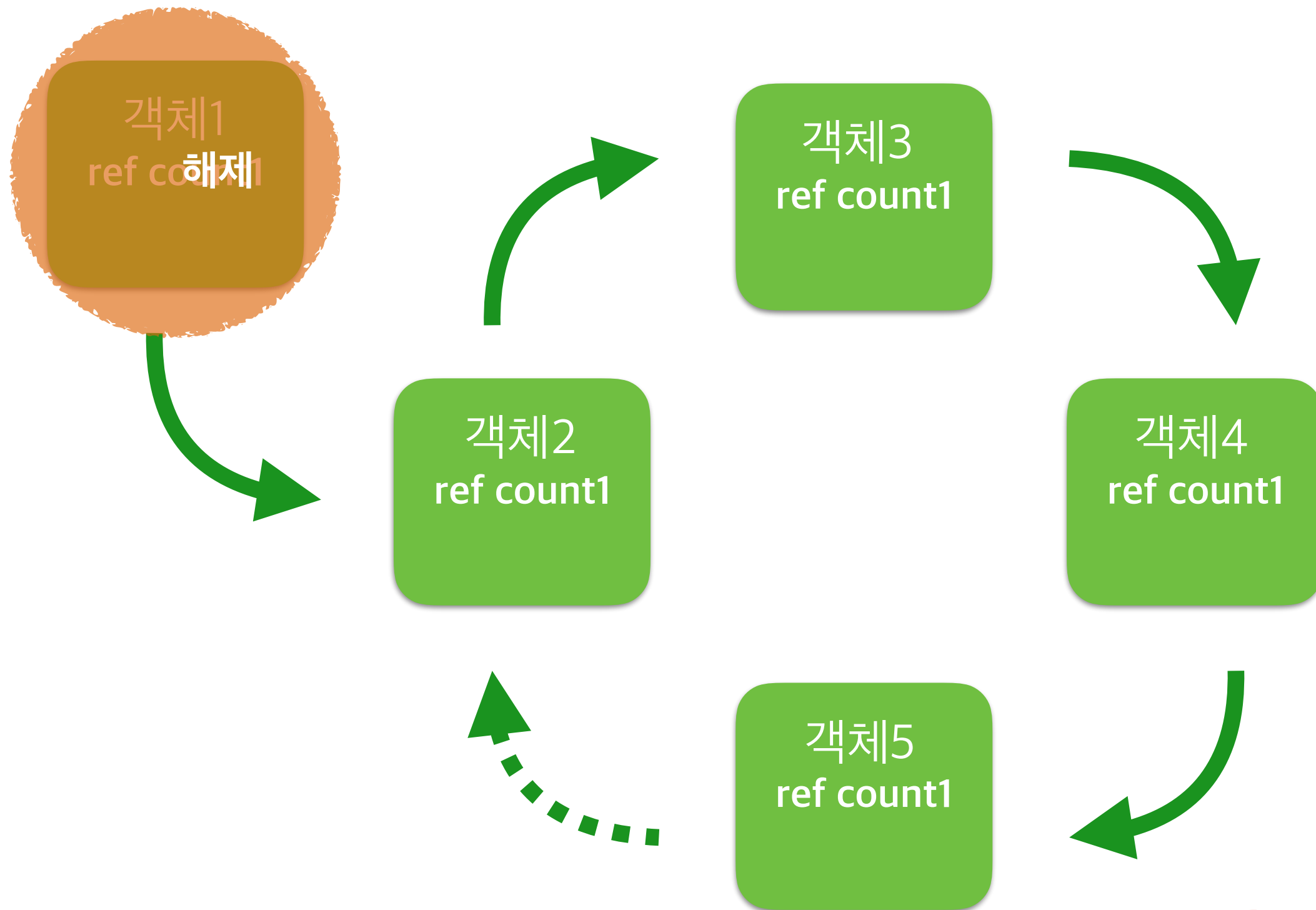
# 순환 참조

순환 참조의 해제를 위해서 "weak"을 사용함



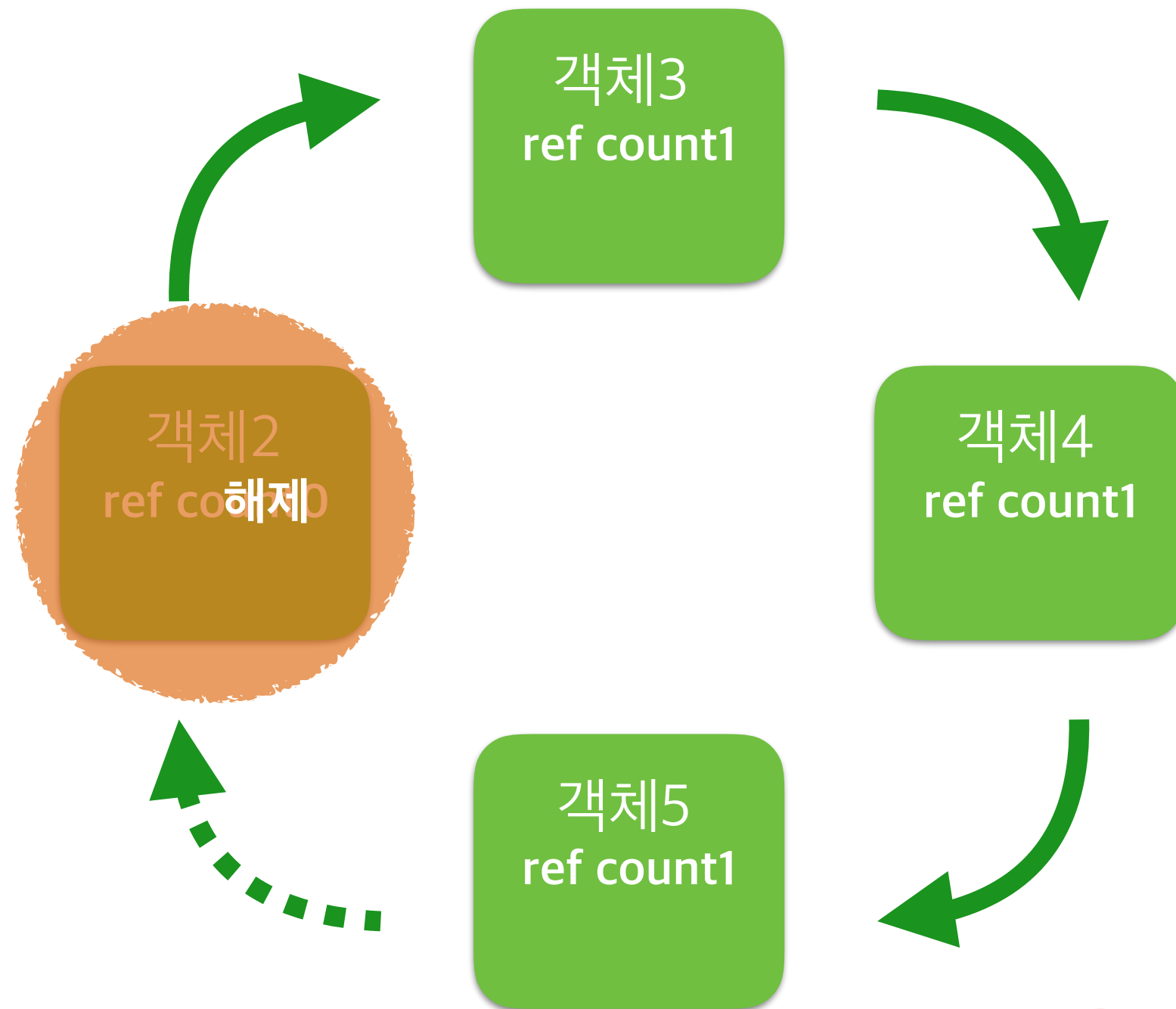
# 순환 참조

---



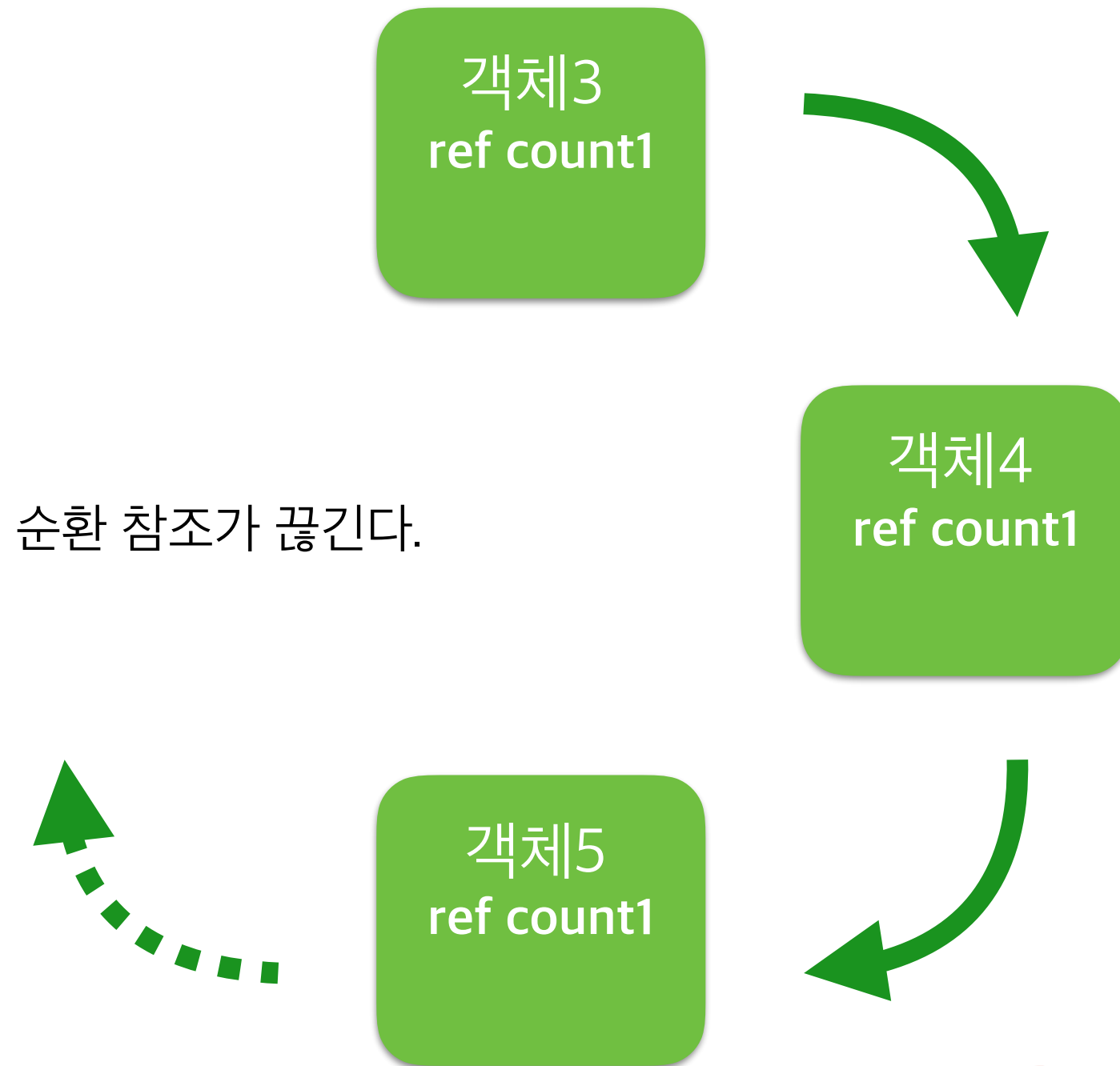
# 순환 참조

---



# 순환 참조

---



# weak pointer 사용 이유

---

delegate -> weak

- 순환 참조를 막기위해
- Autorelease pool을 대신해서 자동 해제가 필요한 경우
- view의 strong 참조 때문에

# Unowned vs Weak

---

- Unowned : 소유권이 없는 참조임을 나타내는 지시어
- Optional 차이
  1. Unowned : 절대 nil이 아니다. ex) closure 안에 self에서 unowned
  2. Weak : nil 일수도 있다



---

# 코드 확장

---

---

# Subscript

---

# Subscript

---

- 클래스, 구조체, 열거형의 collection, list, sequence의 멤버에 접근 가능한 단축문법인 Subscript를 정의 할수 있다.
- Subscript는 별도의 setter/getter없이 index를 통해서 데이터를 설정하거나 값을 가져오는 기능을 할 수 있다.
- Array[index] / Dictionary["Key"] 등의 표현이 Subscript이다.

# 문법

---

```
subscript(index: Type) -> Type {  
    get {  
        // return an appropriate subscript value here  
    }  
    set(newValue) {  
        // perform a suitable setting action here  
    }  
}
```

.....

```
subscript(index: Type) -> Type {  
    // return an appropriate subscript value here  
}
```

\*연산 프로퍼티와 문법이 같음

# 예제 - Array

---

```
class Friends {  
    private var friendNames:[String] = []  
  
    subscript(index:Int) -> String  
    {  
        get {  
            return friendNames[index]  
        }  
        set {  
            friendNames[index] = newValue  
        }  
    }  
}
```

```
let fList = Friends()  
fList[0] = "joo"
```

# 예제 - struct

---

```
struct TimesTable {  
    let multiplier: Int  
    subscript(index: Int) -> Int {  
        return multiplier * index  
    }  
}
```

```
let threeTimesTable = TimesTable(multiplier: 3)  
print("six times three is \(threeTimesTable[6])")
```

# 예제 - 다중 parameter

---

```
struct Matrix {  
    let rows: Int, columns: Int  
    var grid: [Double]  
    init(rows: Int, columns: Int) {  
        self.rows = rows  
        self.columns = columns  
        grid = Array(repeating: 0.0, count: rows * columns)  
    }  
  
    subscript(row: Int, column: Int) -> Double {  
        get {  
            return grid[(row * columns) + column]  
        }  
        set {  
            grid[(row * columns) + column] = newValue  
        }  
    }  
}
```

```
var metrix = Matrix(rows: 2, columns: 2)  
metrix[0,0] = 1  
metrix[0,1] = 2.5
```

---

# Extension

---

저장 프로퍼티 사용할 수 없음

프로토타입 가능

기능 확장

기능별로 나눌 때



# Extensions

---

- Extensions 기능은 기존 클래스, 구조, 열거 형 또는 프로토콜 유형에 새로운 기능을 추가합니다
- Extensions으로 할수 있는것은...
  1. Add computed instance properties and computed type properties
  2. Define instance methods and type methods
  3. Provide new initializers
  4. Define subscripts
  5. Define and use new nested types
  6. Make an existing type conform to a protocol

# 문법

---

```
extension SomeType {  
    // new functionality to add to SomeType goes here  
}
```

```
extension SomeType: SomeProtocol, AnotherProtocol {  
    // implementation of protocol requirements goes here  
}
```

# 유형 : Compute Properties

---

```
extension Double {  
    var km: Double { return self * 1_000.0 }  
    var m: Double { return self }  
    var cm: Double { return self / 100.0 }  
    var mm: Double { return self / 1_000.0 }  
    var ft: Double { return self / 3.28084 }  
}
```

```
let oneInch = 25.4.mm  
print("One inch is \(oneInch) meters")  
// Prints "One inch is 0.0254 meters"  
let threeFeet = 3.ft  
print("Three feet is \(threeFeet) meters")  
// Prints "Three feet is 0.914399970739201 meters"
```

# 유형 : init

---

```
extension Rect {  
    init(center: Point, size: Size) {  
        let originX = center.x - (size.width / 2)  
        let originY = center.y - (size.height / 2)  
        self.init(origin: Point(x: originX, y: originY), size:  
size)  
    }  
}
```

# 유형 : method

---

```
extension Int {  
    func repetitions(task: () -> Void) {  
        for _ in 0..  
            task()  
        }  
    }  
}
```

```
3.repetitions {  
    print("Hello!")  
}  
  
. // Hello!  
. // Hello!  
. // Hello!
```

# 유형 : mutating method

---

```
extension Int {  
    mutating func square() {  
        self = self * self  
    }  
}
```

```
var someInt = 3  
someInt.square()
```

# 유형 : Subscript

---

```
extension Int {  
    subscript(digitIndex: Int) -> Int {  
        var decimalBase = 1  
        for _ in 0..  
            digitIndex {  
            decimalBase *= 10  
        }  
        return (self / decimalBase) % 10  
    }  
}
```

```
746381295[0]  
// returns 5  
746381295[1]  
// returns 9  
746381295[2]  
// returns 2  
746381295[8]  
// returns 7
```

---

# Generic

---



# Generic

---

- 어떤 타입에도 유연한 코드를 구현하기 위해 사용되는 기능
- 코드의 중복을 줄이고, 깔끔하고 추상적인 표현이 가능하다.

# 왜 Generic을 사용하는가?

---

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```

두 Int를 받아 서로 바꿔주는 스왑함수가 있다.

우리는 Int 외에도 Double, String 등 다양한 타입의 데이터를 스왑하고 싶다면 어떻게 해야될까?

# Generic을 사용한 swap함수

---

```
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {  
    let temporaryA = a  
    a = b  
    b = temporaryA  
}
```

```
var someInt = 3  
var anotherInt = 107  
swapTwoValues(&someInt, &anotherInt)  
// someInt is now 107, and anotherInt is now 3
```

```
var someString = "hello"  
var anotherString = "world"  
swapTwoValues(&someString, &anotherString)  
// someString is now "world", and anotherString is now "hello"
```

# Framework확인

---

- Array / Dictionary 파일 확인하기

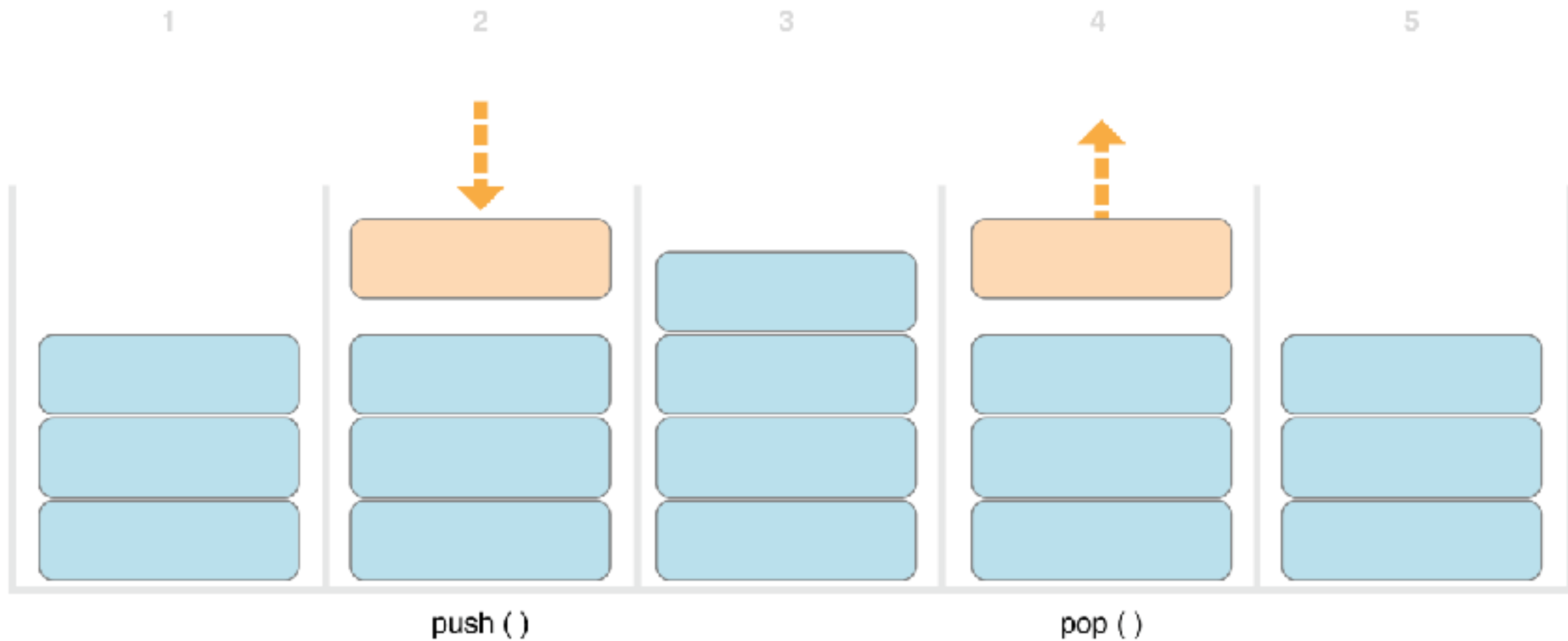
# Type Parameters

---

- 제넥릭에 사용된 “T”는 타입의 이름으로 사용되었다기 보다는 placeholder 역할로 사용되었다.
- 타입은 꺾쇠<> 로 감싸 표시한다.
- 타입의 이름은 보통 사용되는 속성에 맞게 네이밍 할수 있으나 아무런 연관이 없는 타입의 경우에는 T,U,V 같은 알파벳으로 사용된다.

# 실습 : Stack만들기

---



# Stack: Int

---

```
struct MyStack
{
    private var intStackTemp: [Int] = []

    mutating func push(_ data: Int)
    {
        intStackTemp.append(data)
    }

    mutating func pop() -> Int
    {
        return intStackTemp.removeLast()
    }
}
```

# Stack 다양한 타입

---

```
struct MyStackInt
{
    private var intStackTemp:[Int] = []

    mutating func push(_ data:Int)
    {
        intStackTemp.append(data)
    }

    mutating func pop() -> Int
    {
        return intStackTemp.removeLast()
    }
}
```

```
struct MyStackString
{
    private var strStackTemp:[String] = []

    mutating func push(_ data:String)
    {
        strStackTemp.append(data)
    }

    mutating func pop() -> String
    {
        return strStackTemp.removeLast()
    }
}
```



# Stack Using Generic

---

```
struct MyStack<T>
{
    private var stackTemp:[T] = []

    mutating func push(_ data:T)
    {
        stackTemp.append(data)
    }

    mutating func pop() -> T
    {
        return stackTemp.removeLast()
    }
}
```

```
var stack:MyStack<Int> = MyStack<Int>()
```