Memory Management

Giftbot

Memory Management without ARC

Fast campus

1. GC (Garbage Collection)

- 정기적으로 Garbage Collector 가 동작하여 더이상 사용되지 않는 메모리를 반환하는 방식
- OS X 에서만 지원했었으나 버전 10.8 (Mountain Lion) 부터 deprecated

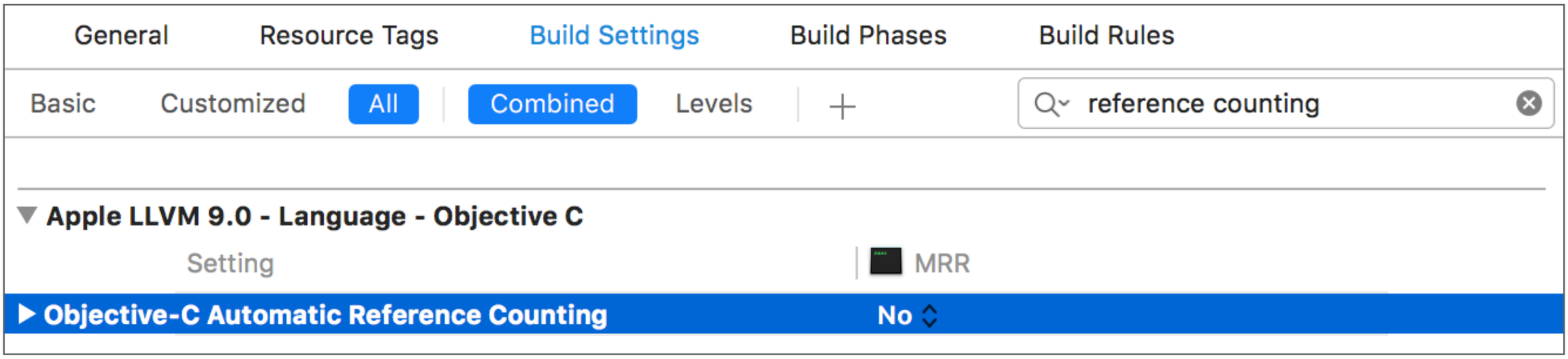
2. MRR (Manual Retain-Release) / MRC (Manual Referece Counting)

- RC(Reference Counting) 를 통해 메모리를 수동으로 관리하는 방식
- retain / release / autorelease 등의 메모리 관리 코드를 직접 호출
- 개발자가 명시적으로 RC 를 증가시키고 감소시키는 작업 수행

MRR

RC 에 대한 이해 필요 (Reference Counting / Retain Count / Reference Count)

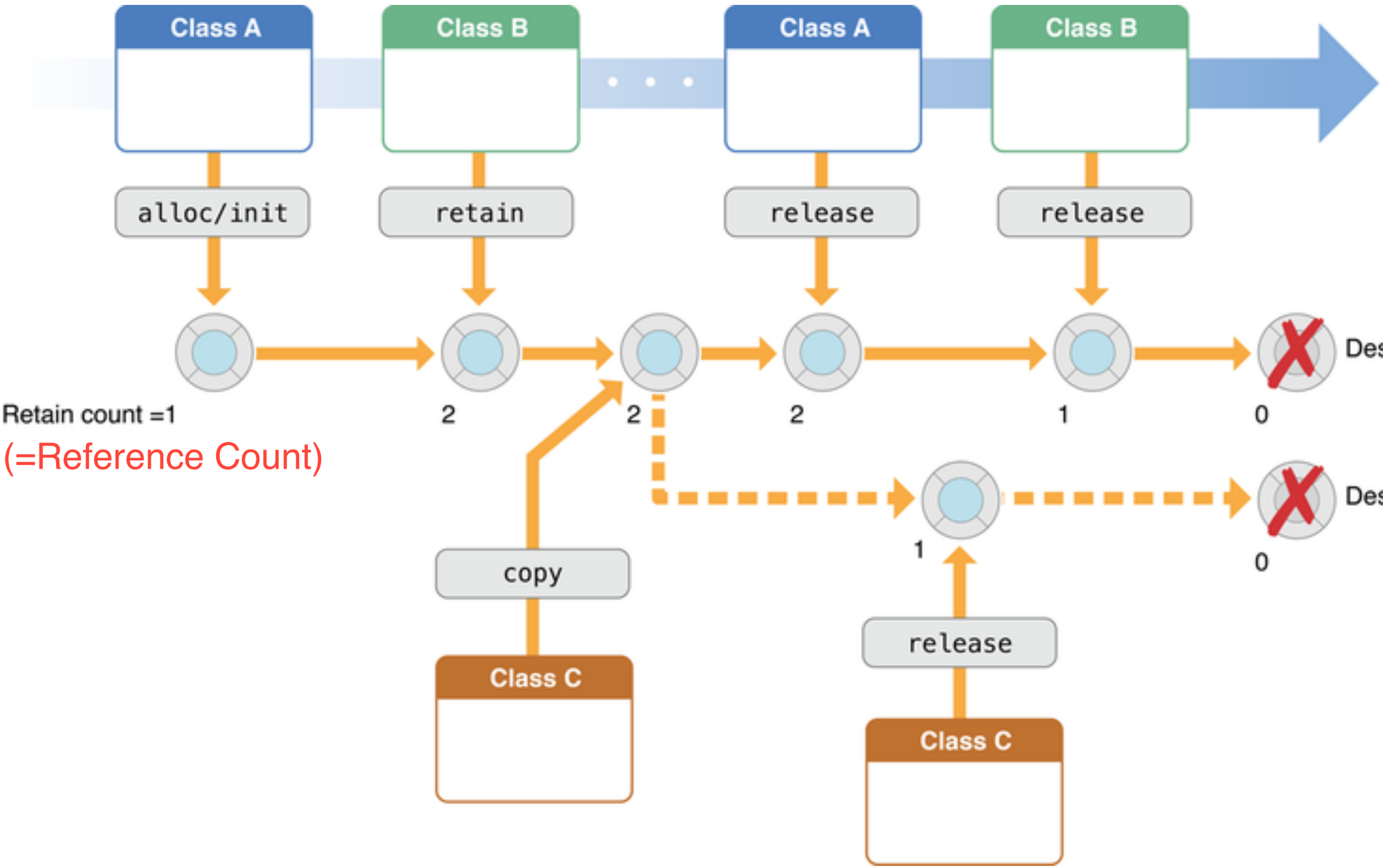
Objective-C 에서는 ARC 해제 가능



Reference Counting

```
int main(int argc, const char * argv[]) {

    Person *man = [[Person alloc] init];          ——— count : 1
    [man doSomething];

    [man retain];                                 ——— count : 2
    [man doSomething];

    // [man release];
    [man doSomething];                            ——— count : 2
    [man release];                                ——— count : 1
    [man release];                                ——— count : 0
    return 0;
}
```

Reference Counting (Obj-C)

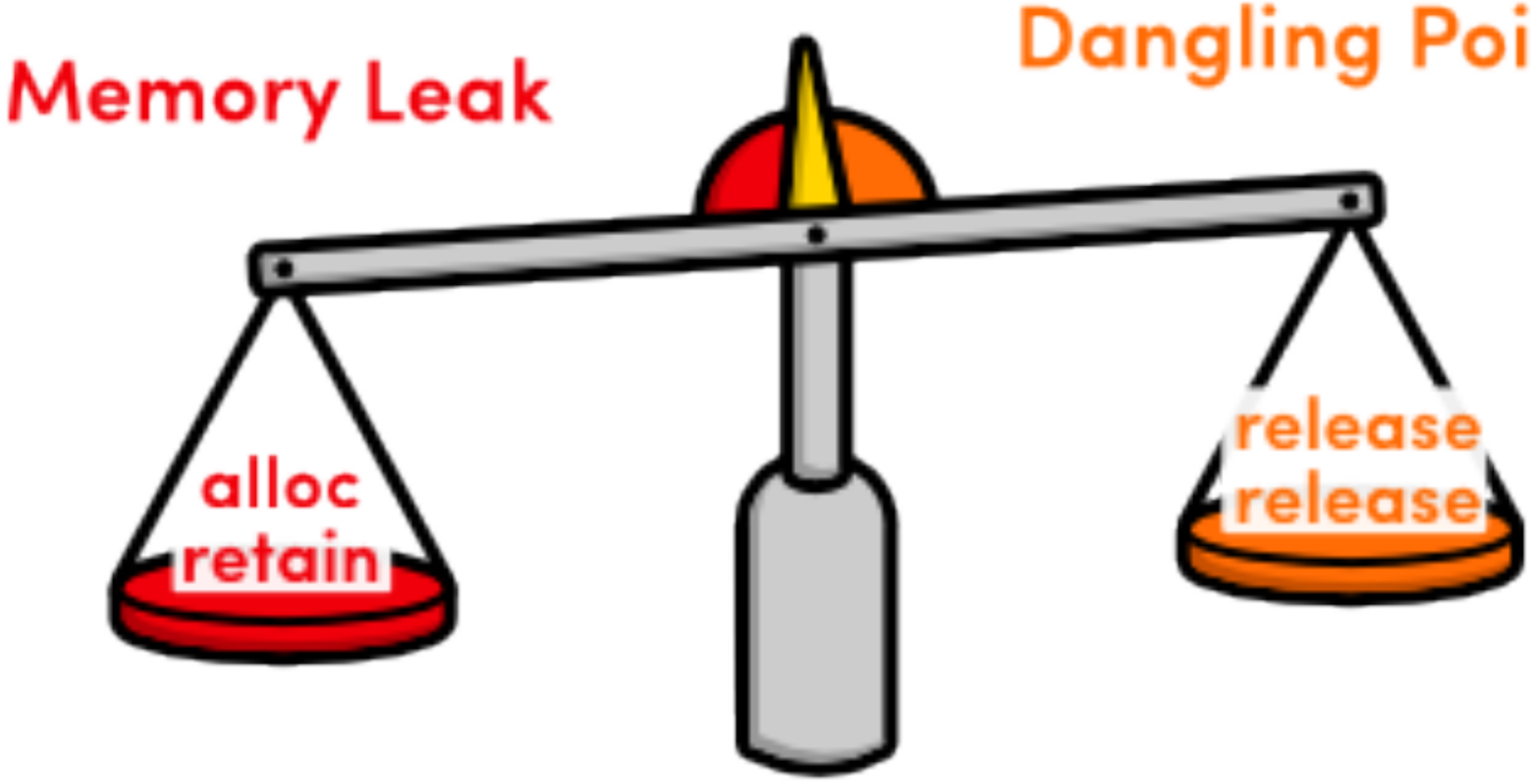


Leak vs Dangling Pointer

카운트 할당과 해제는 균형이 맞아야 함

- alloc , retain 이 많을 경우는 Memory Leak 발생
- release 가 많을 경우 Dangling Pointer (허상, 고아) 발생

메모리 낭비 or 부족 발생
데이터 접근 불가 현상 발생
(Dangling Pointer)



MemoryManagement Is Hard

Fast campus

- Lots of rules and conventions
- High hurdles for new developers
- Constant attention for existing developers
- Requires perfection

MemoryManagement Is Hard

Fast campus

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

Programming with Retain/Release

Fast campus

Xcode Static Analyzer

Fast campus

```
NSObject *objectID = 0;

for (NSUInteger i=0; i < count; ++i) {          Looping back to the head of the loop

    NSObject *object = [trackedElements objectAtIndex:i];

    if ([object isMemberOfClass:[NSString class]])
    {
        objectID = [[NSString alloc] initWithString:aString];
    }                                           Method returns an Objective-C object with a +1 retain count (owning reference)

    if (objectID != nil)
    {
        [objectID release];                     Object released
    }                                           Reference-counted object is used after it is released
}
```

Xcode Static Analyzer

```
13 int main(int argc, const char * argv[]) {
14     Person *man = [[Person alloc] init];        1. Method returns an Objective-C object with a +1 retain count
15     [man doSomething];
16
17     return 0;        2. Object leaked: object allocated and stored into 'man' is not referenced later in this execution path and has a retain count of +1
18 }
```

[Memory Leak]

```
13 int main(int argc, const char * argv[]) {
14     Person *man = [[Person alloc] init];        1. Method returns an Objective-C object with a +1 retain count
15     [man doSomething];
16     [man release];                              2. Object released
17
18     [man doSomething];                          3. Reference-counted object is used after it is released
19     [man doSomething];
20     [man doSomething];
21
22     return 0;
23 }
24
```

[Released Object]

ARC

2011년 애플에서 발표한 메모리 자동관리

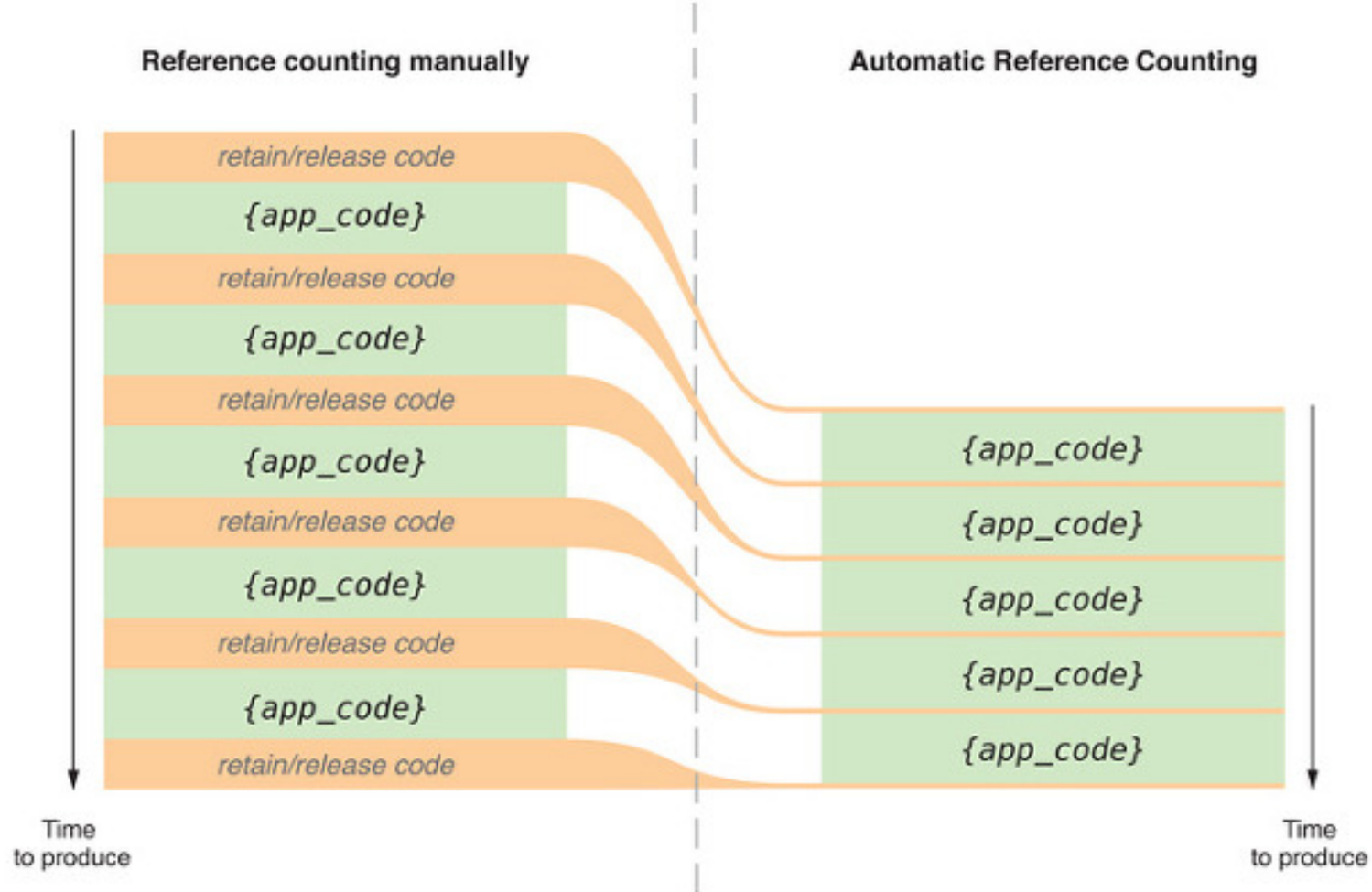
ARC (Automatic Reference Counting)

RC 자동 관리 방식 (WWDC 2011 발표)

컴파일러가 개발자를 대신하여 메모리 관리 코드를 적절한 위치에 자동으로 삽입

GC 처럼 런타임이 아닌 컴파일 단에서 처리 (Heap 에 대한 스캔 불필요 / 앱 일시 정지 현상 없음)

메모리 관리 이슈를 줄이고 개발자가 코딩 자체에 집중할 수 있도록 함



ARC (Automatic Reference Counting)

heap에 관련된 것만 컨트롤

struct랑 비교했을때

ARC 는 클래스의 인스턴스에만 적용 (Class - Reference 타입 , Struct / Enum - Value 타입)

활성화된 참조카운트가 하나라도 있을 경우 메모리에서 해제 되지 않음

참조 타입

- 강한 참조 (Strong) : 기본값. 참조될 때마다 참조 카운트 1 증가

- 약한 참조 (Weak), 미소유 참조 (Unowned) : 참조 카운트를 증가시키지 않음

강한 순환 참조 (Strong Reference Cycles) 에 대한 주의 필요

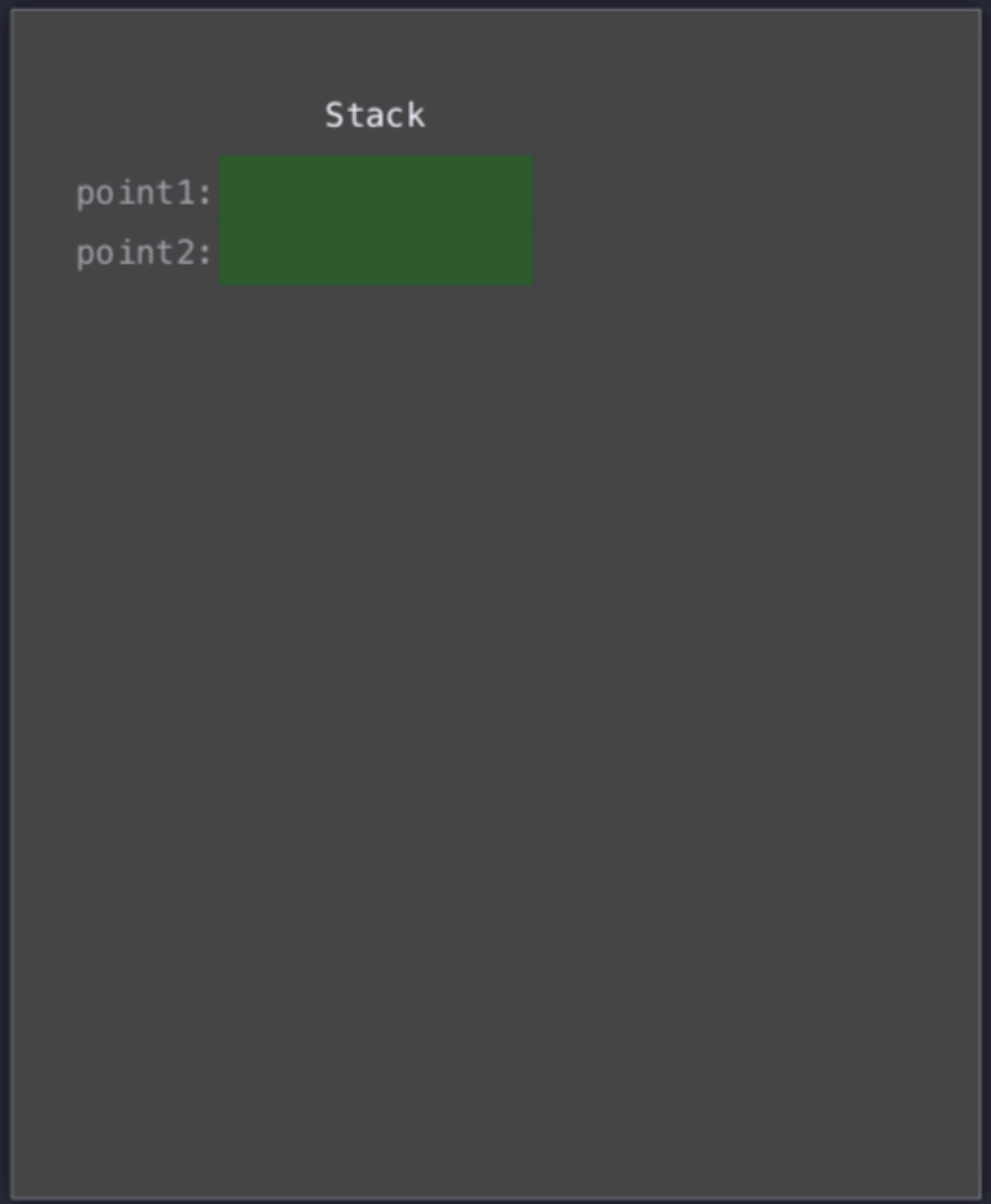
	Var	Let	Optional	Non-Optional
Strong	👍	👍	👍	👍
Weak	👍	⛔	👍	⛔
Unowned	👍	👍	⛔	👍

```
// Reference Counting
// Class

class Point {
    var x, y: Double
    func draw() { … }
}


let point1 = Point(x: 0, y: 0)
let point2 = point1
point2.x = 5
// use `point1`
// use `point2`
```

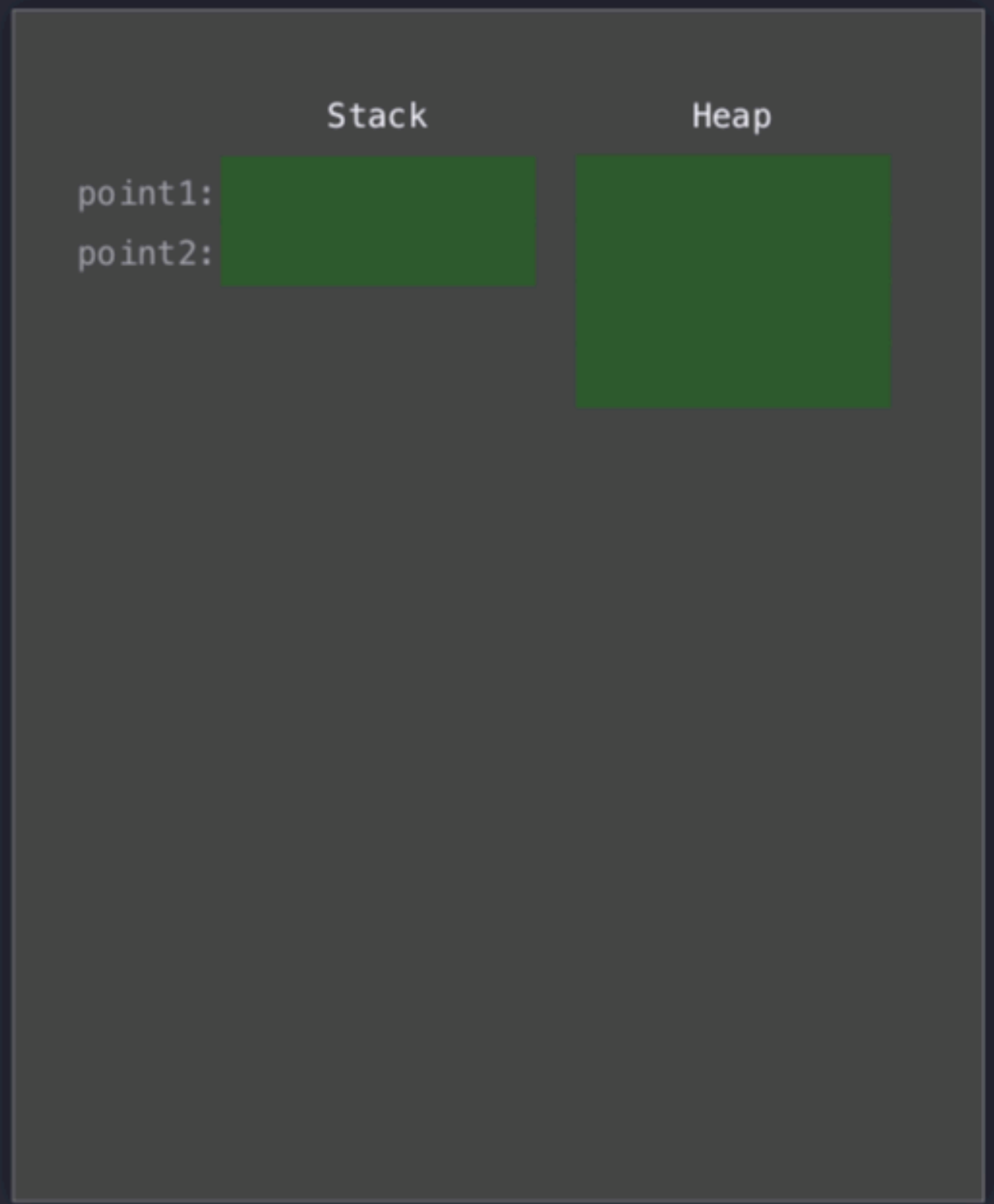
```
// Reference Counting
// Class

class Point {
    var x, y: Double
    func draw() { … }
}


let point1 = Point(x: 0, y: 0)
let point2 = point1
point2.x = 5
// use `point1`
// use `point2`
```

```
// Reference Counting
// Class (generated code)

class Point {
    var refCount: Int
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
retain(point2)
point2.x = 5
// use `point1`
release(point1)
// use `point2`
release(point2)
```

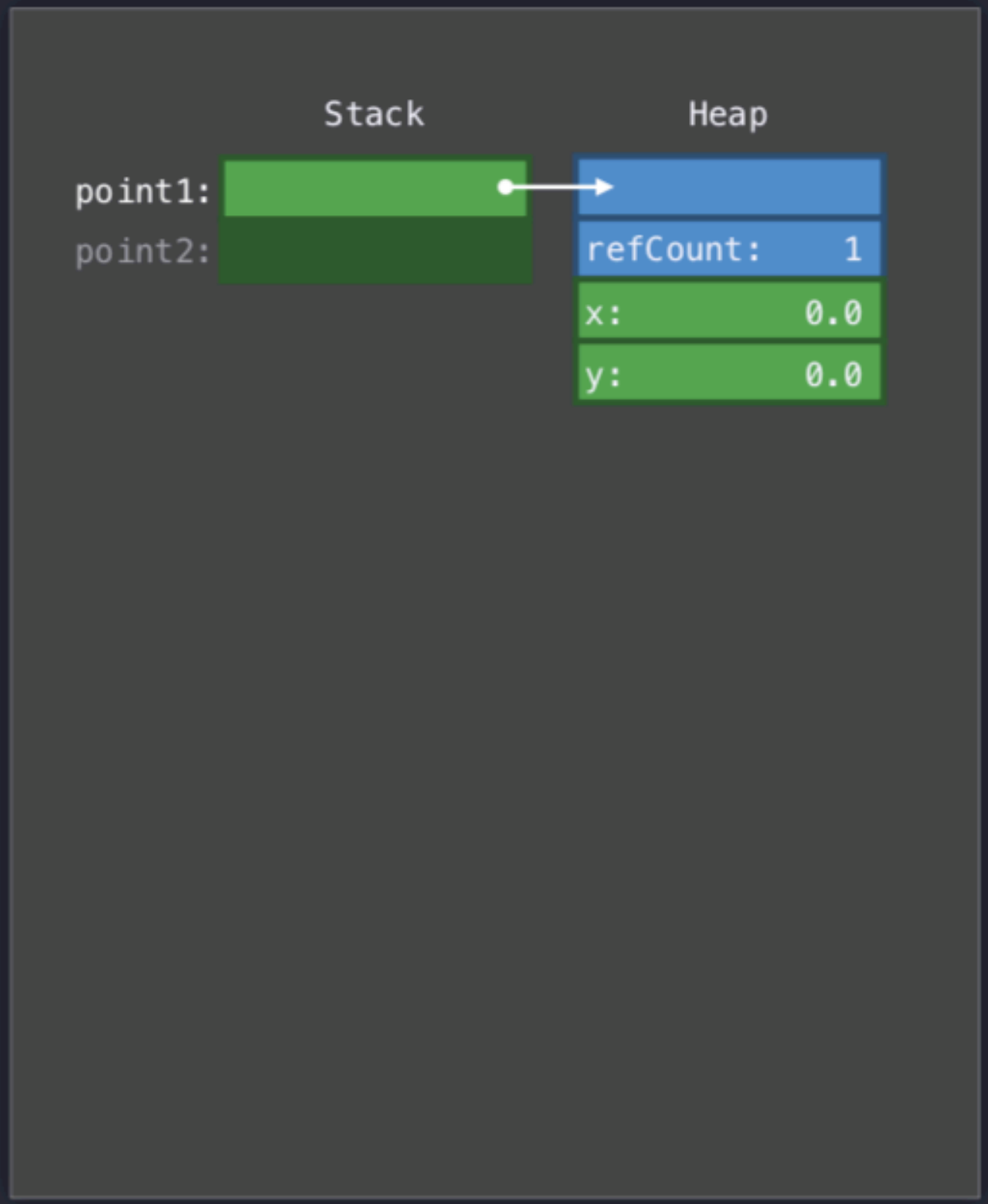
```
// Reference Counting
// Class (generated code)

class Point {
    var refCount: Int
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
retain(point2)
point2.x = 5
// use `point1`
release(point1)
// use `point2`
release(point2)
```



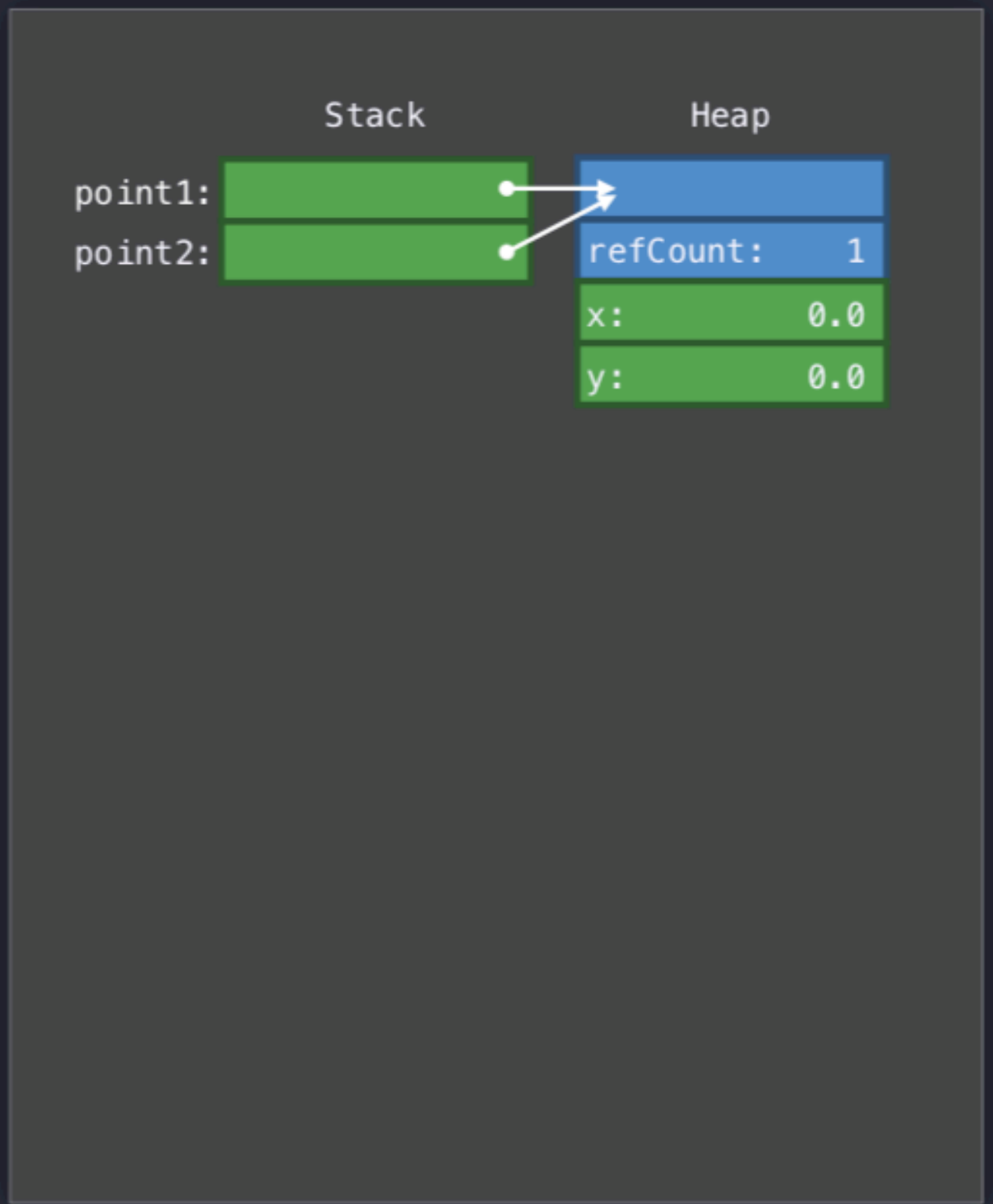
```
// Reference Counting
// Class (generated code)

class Point {
    var refCount: Int
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
retain(point2)
point2.x = 5
// use `point1`
release(point1)
// use `point2`
release(point2)
```



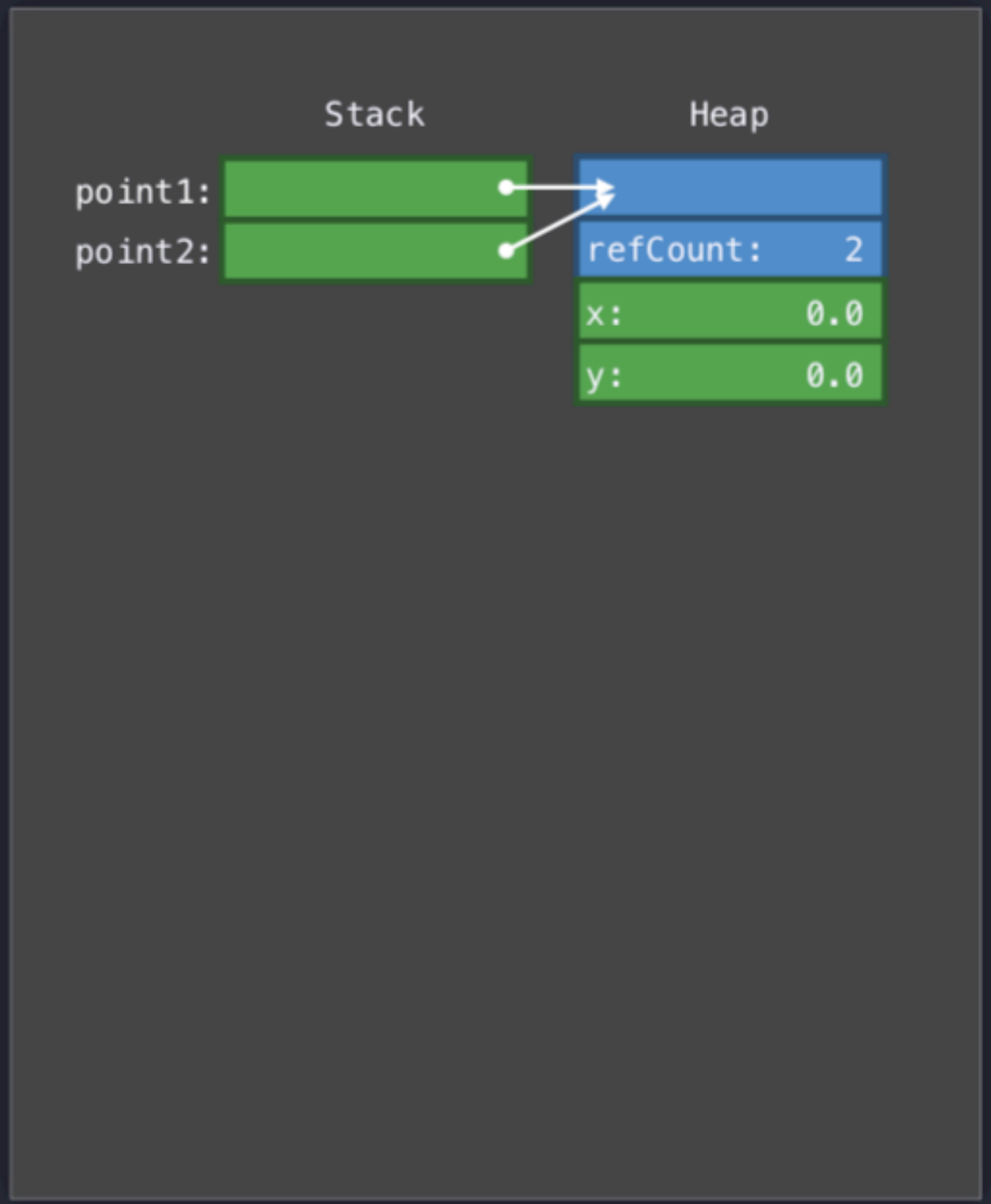
```
// Reference Counting
// Class (generated code)

class Point {
    var refCount: Int
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
retain(point2)
point2.x = 5
// use `point1`
release(point1)
// use `point2`
release(point2)
```



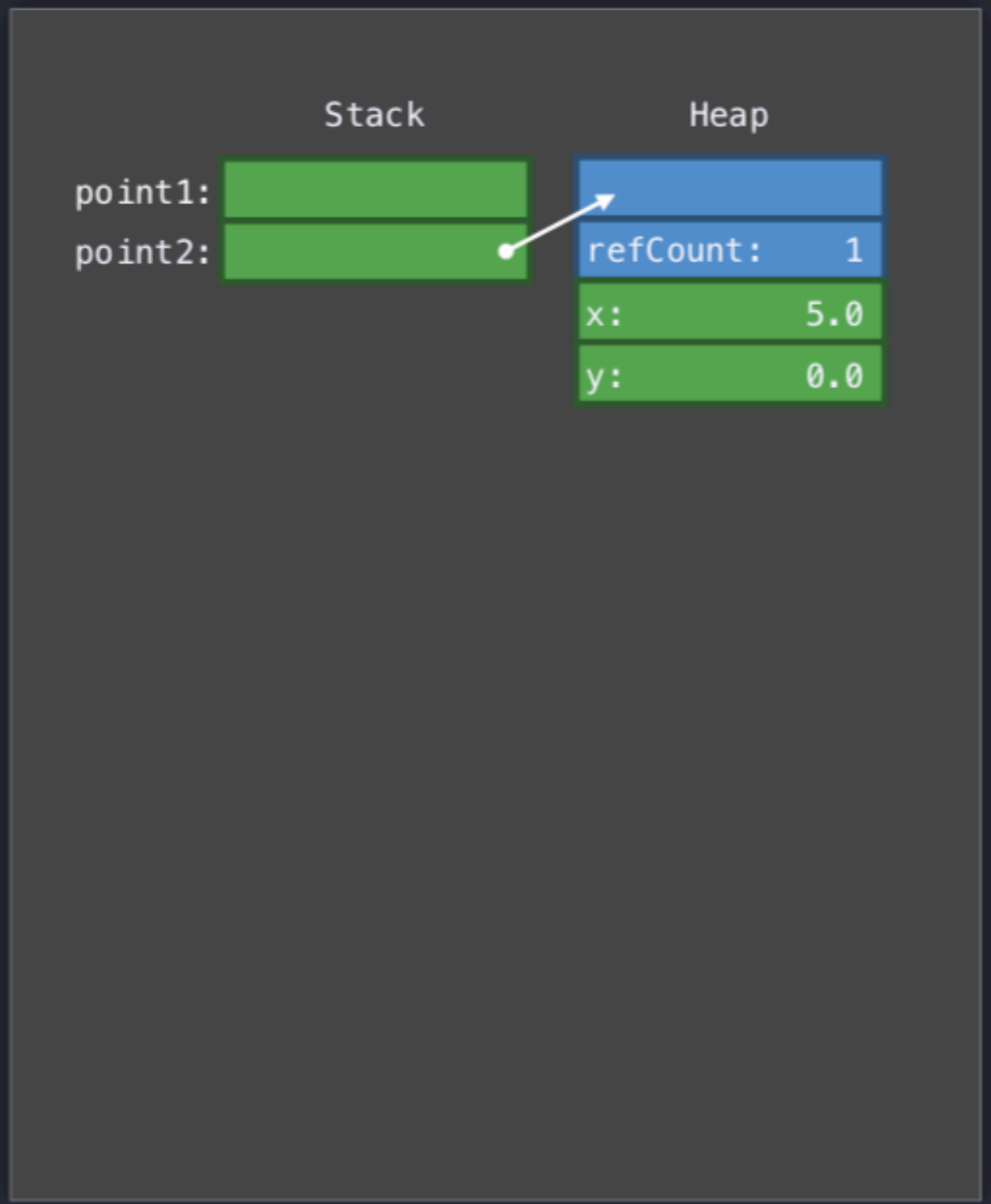
```
// Reference Counting
// Class (generated code)

class Point {
    var refCount: Int
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
retain(point2)
point2.x = 5
// use `point1`
release(point1)
// use `point2`
release(point2)
```



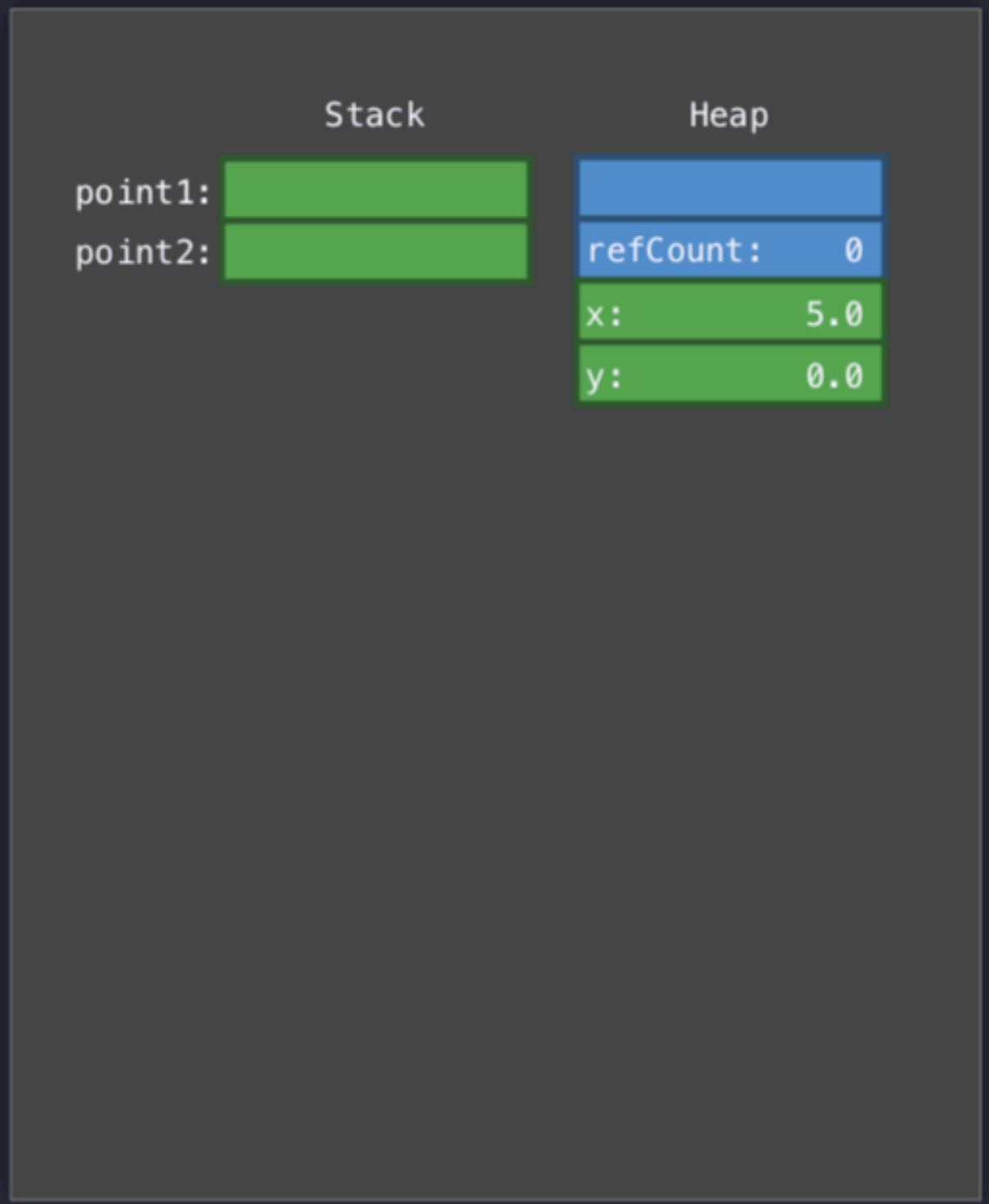
```
// Reference Counting
// Class (generated code)

class Point {
    var refCount: Int
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
retain(point2)
point2.x = 5
// use `point1`
release(point1)
// use `point2`
release(point2)
```



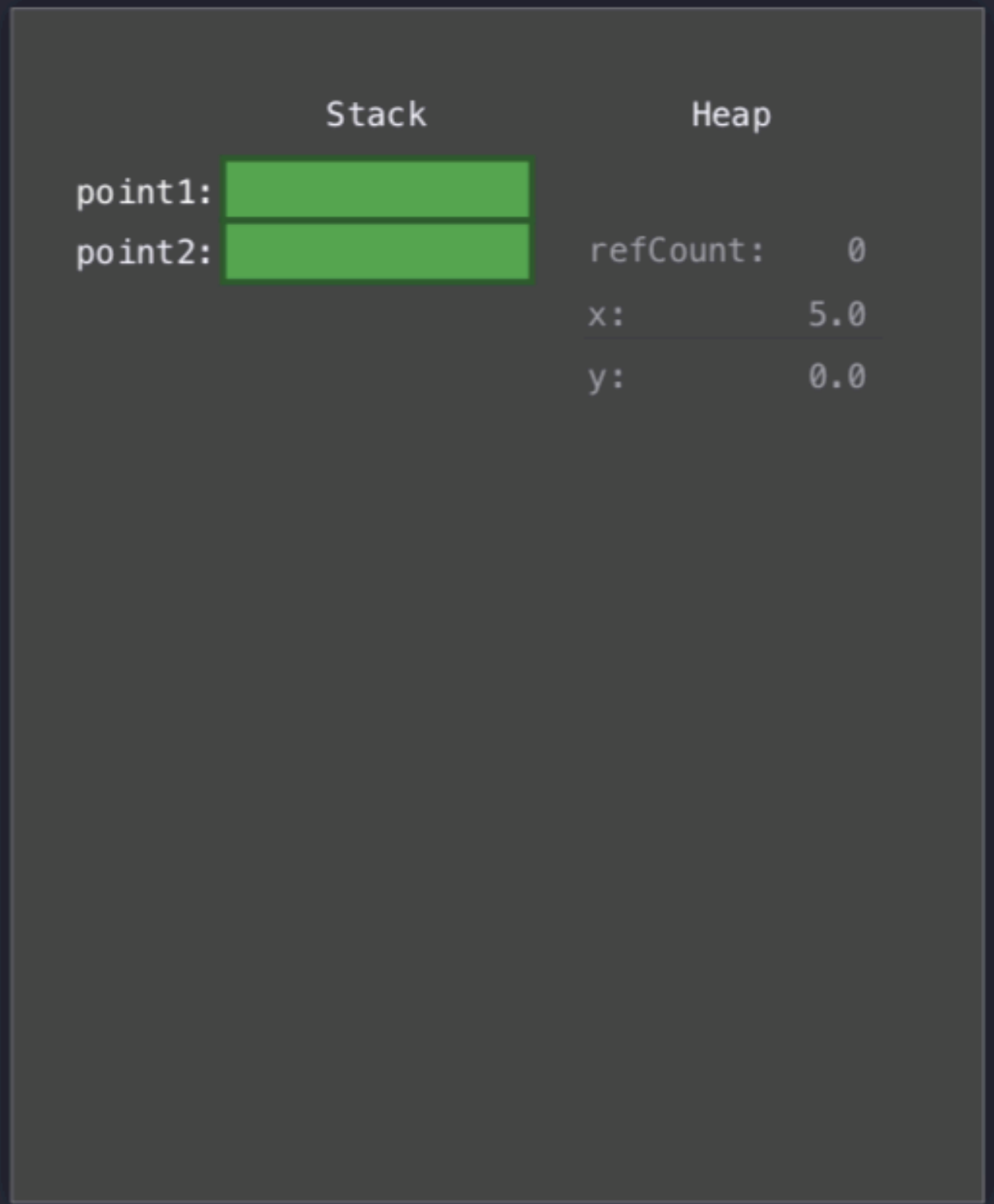
```
// Reference Counting
// Class (generated code)

class Point {
    var refCount: Int
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
retain(point2)
point2.x = 5
// use `point1`
release(point1)
// use `point2`
release(point2)
```



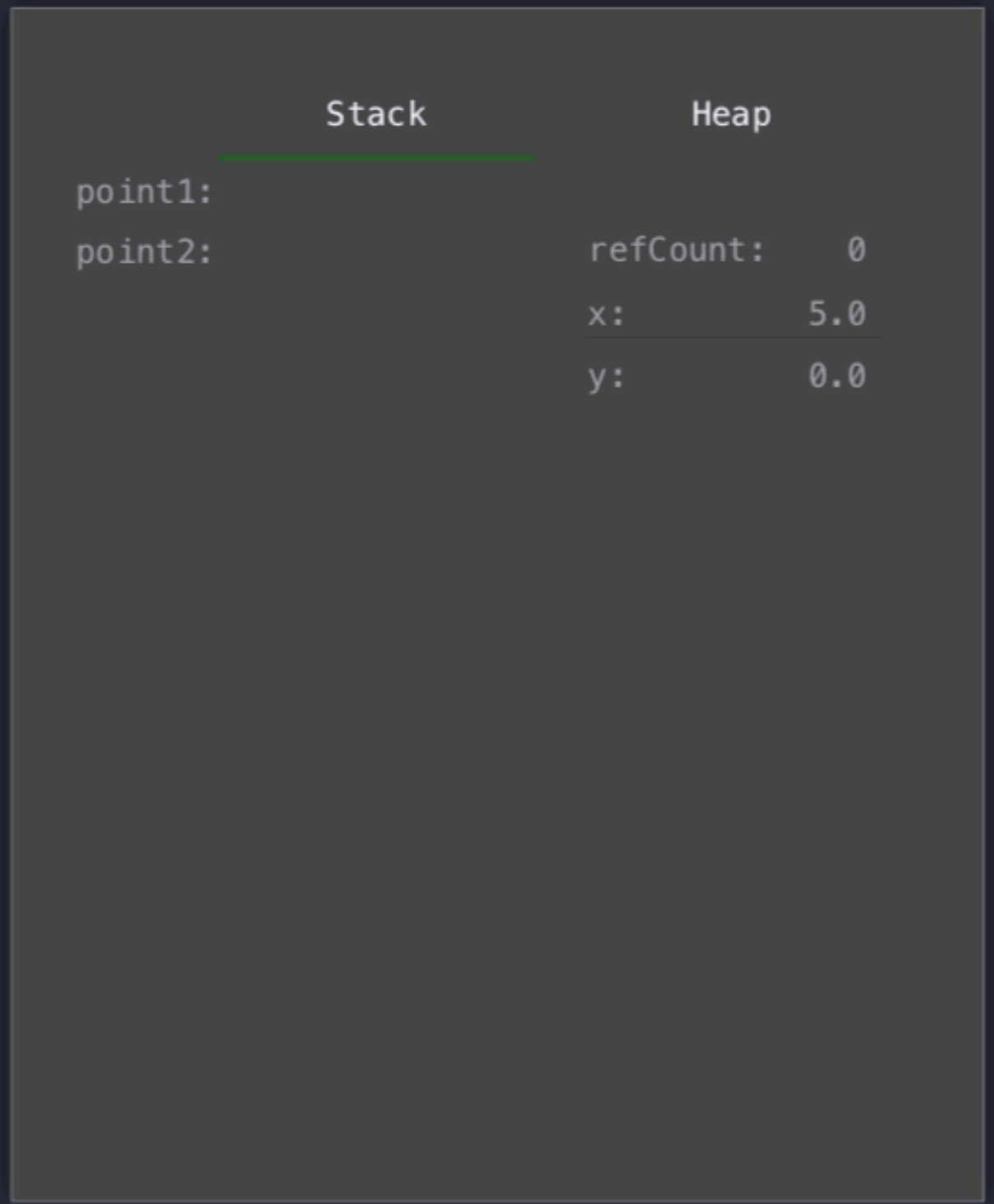
```
// Reference Counting
// Class (generated code)

class Point {
    var refCount: Int
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
retain(point2)
point2.x = 5
// use `point1`
release(point1)
// use `point2`
release(point2)
```



```
// Reference Counting
// Class (generated code)

class Point {
   var refCount: Int
   var x, y: Double
   func draw() { … }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
retain(point2)
point2.x = 5
// use `point1`
release(point1)
// use `point2`
release(point2)
```



```
// Reference Counting
// Class (generated code)

class Point {
   var refCount: Int
   var x, y: Double
   func draw() { … }
}

let point1 = Point(x: 0, y: 0)
let point2 = point1
retain(point2)
point2.x = 5
// use `point1`
release(point1)
// use `point2`
release(point2)
```



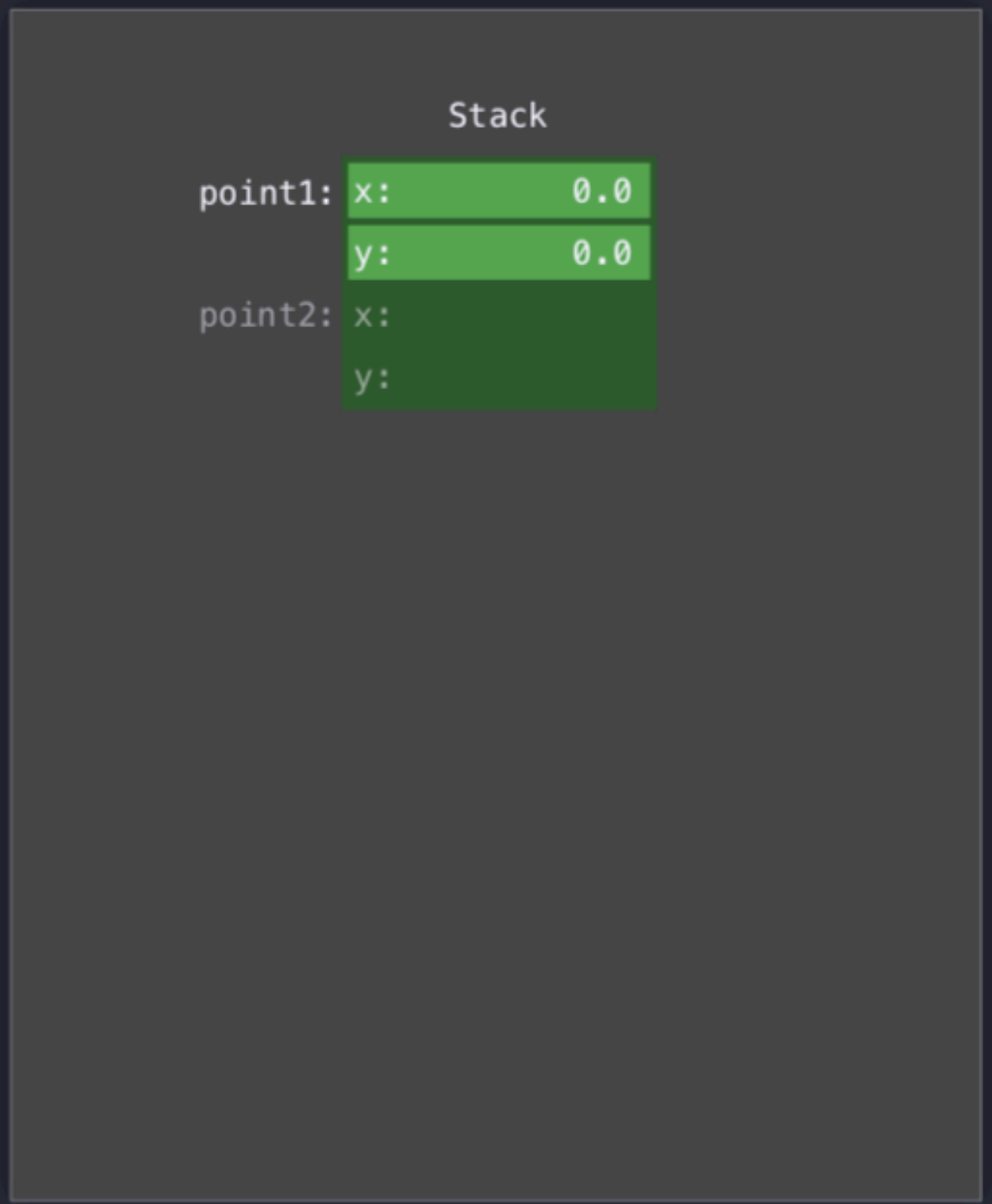
ARC in Struct ?

RC는 heap에 관련하여 작동하므로 구조체에는 역할을 하지 않음

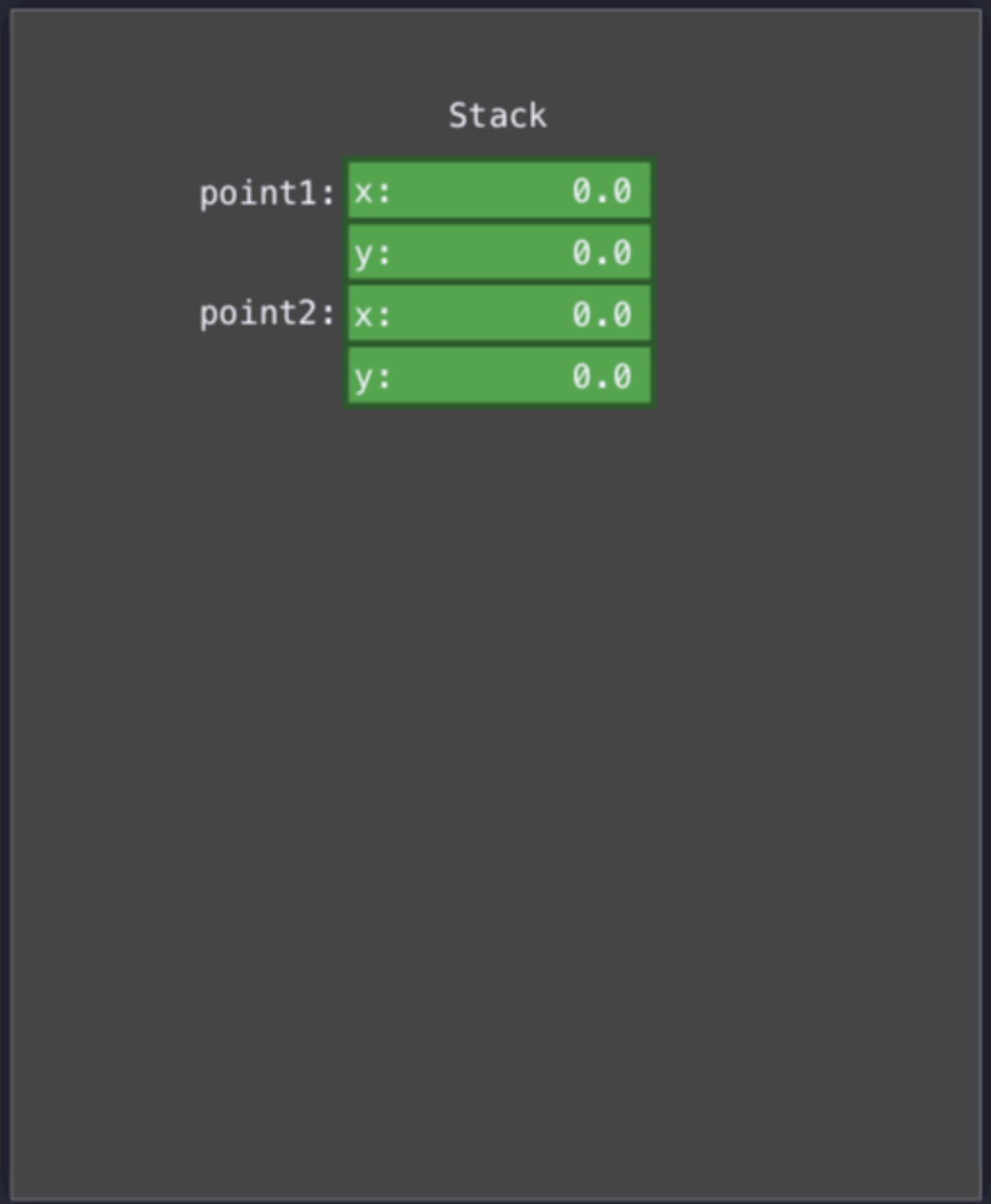
```
// Allocation
// Struct

struct Point {
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
var point2 = point1
point2.x = 5
// use `point1`
// use `point2`
```

```
// Allocation
// Struct

struct Point {
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
var point2 = point1
point2.x = 5
// use `point1`
// use `point2`
```

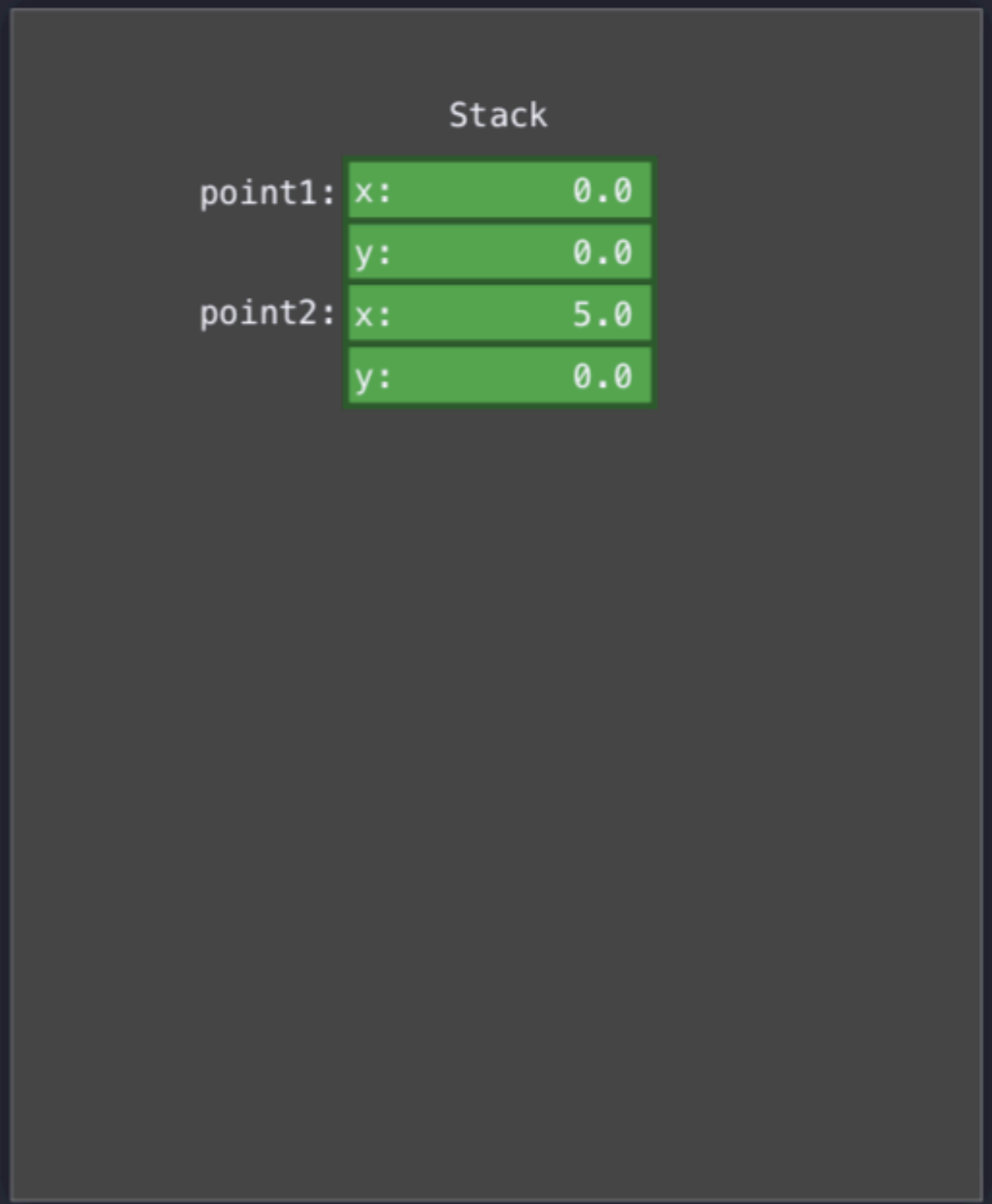
```swift
// Allocation
// Struct

struct Point {
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
var point2 = point1
point2.x = 5
// use `point1`
// use `point2`
```



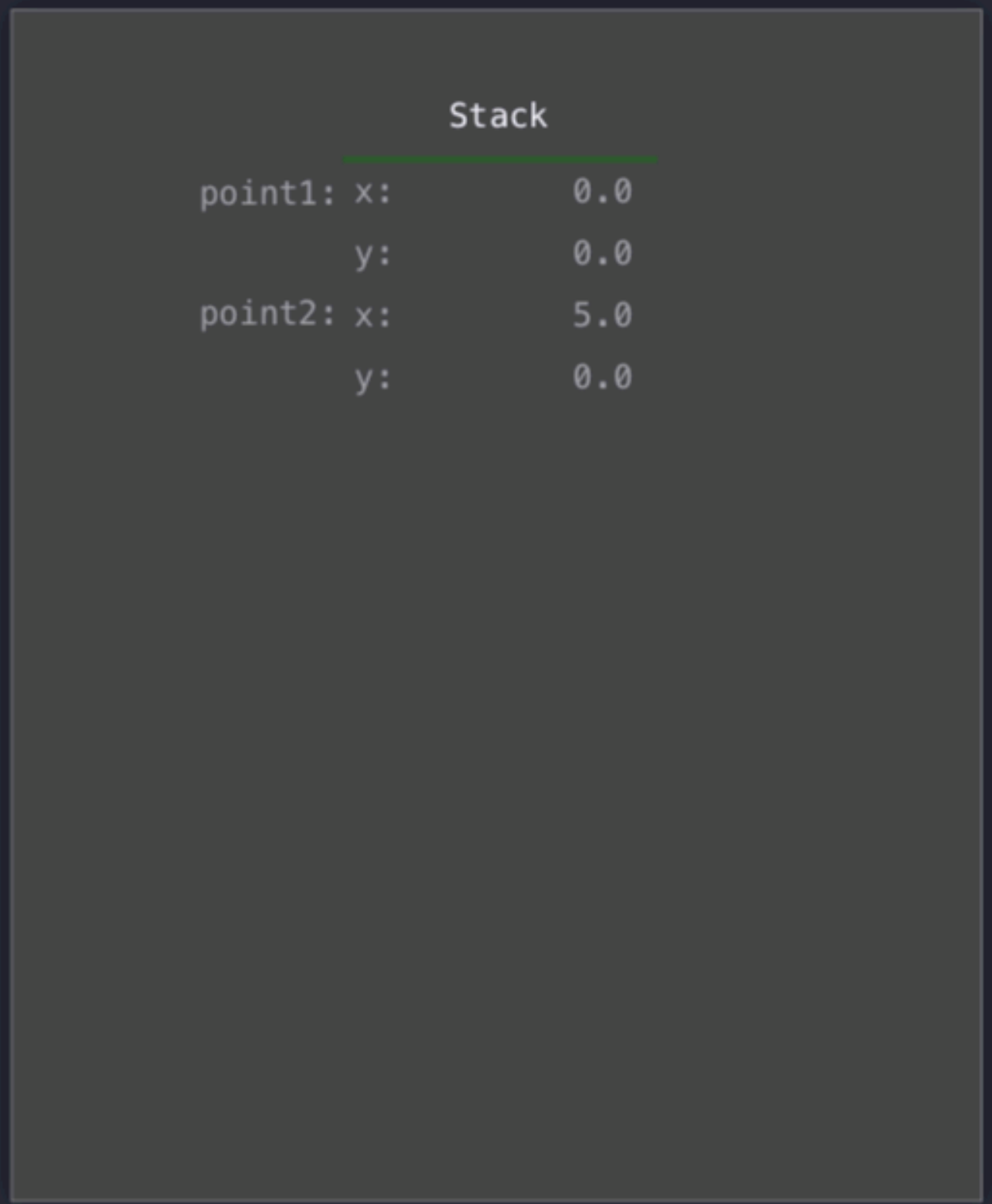
```
// Allocation
// Struct

struct Point {
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
var point2 = point1
point2.x = 5
// use `point1`
// use `point2`
```



```swift
// Allocation
// Struct

struct Point {
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
var point2 = point1
point2.x = 5
// use `point1`
// use `point2`
```



```swift
// Allocation
// Struct

struct Point {
    var x, y: Double
    func draw() { … }
}

let point1 = Point(x: 0, y: 0)
var point2 = point1
point2.x = 5
// use `point1`
// use `point2`
```



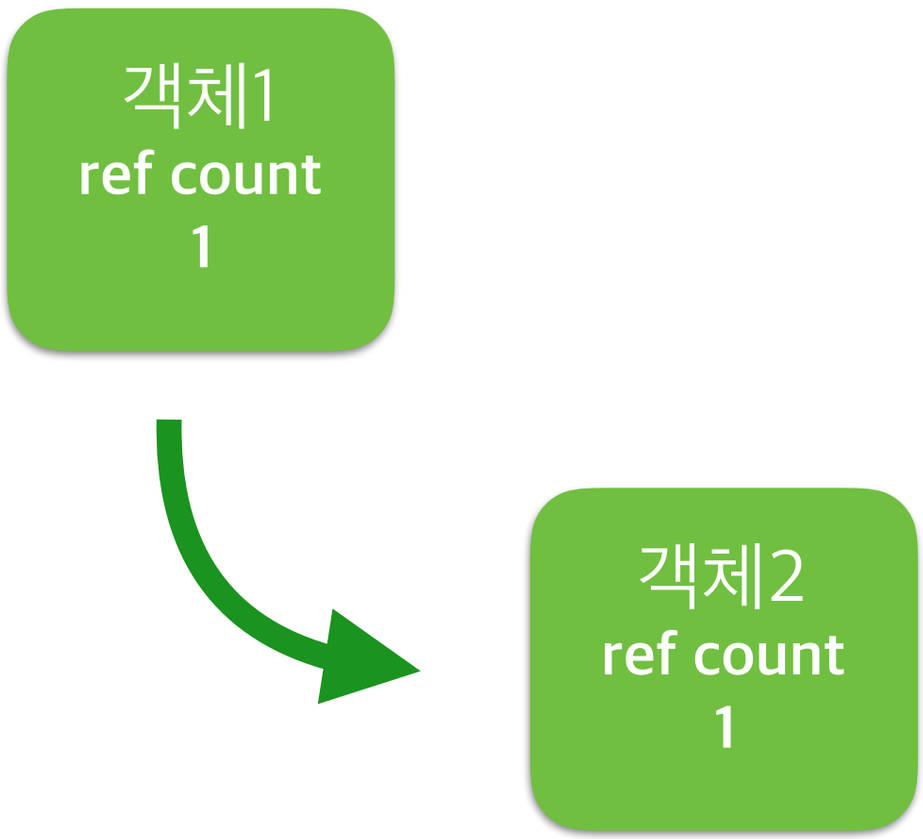
Strong Reference Cycle

Strong Reference Cycle

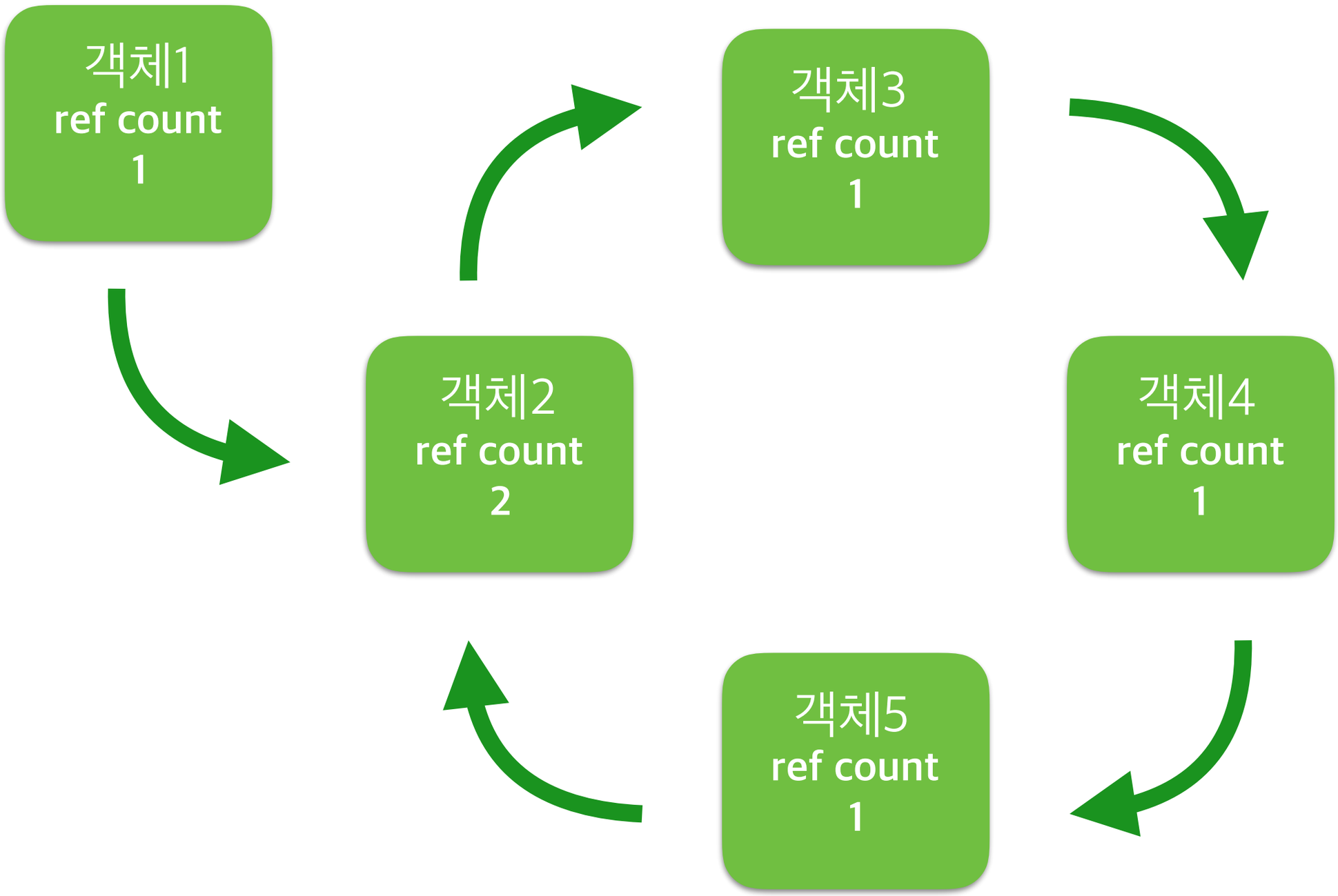
- 객체에 접근 가능한 모든 연결을 끊었음에도 순환 참조로 인해
  활성화된 참조 카운트가 남아 있어 메모리 누수가 발생하는 현상
- 앱의 실행이 느려지거나 오동작 또는 오류를 발생시키는 원인이 됨

레퍼런서 카운팅이 0이 되어야 메모리에서 사라지는데 0이 되지 않는 상황을 만들어서 메모리 누수를 발생시킴. 없어질 놈이 안없어지고 계속 쌓이기 때문
이를 방지하기 위해서 Weak을 사용.

Strong Reference Cycle

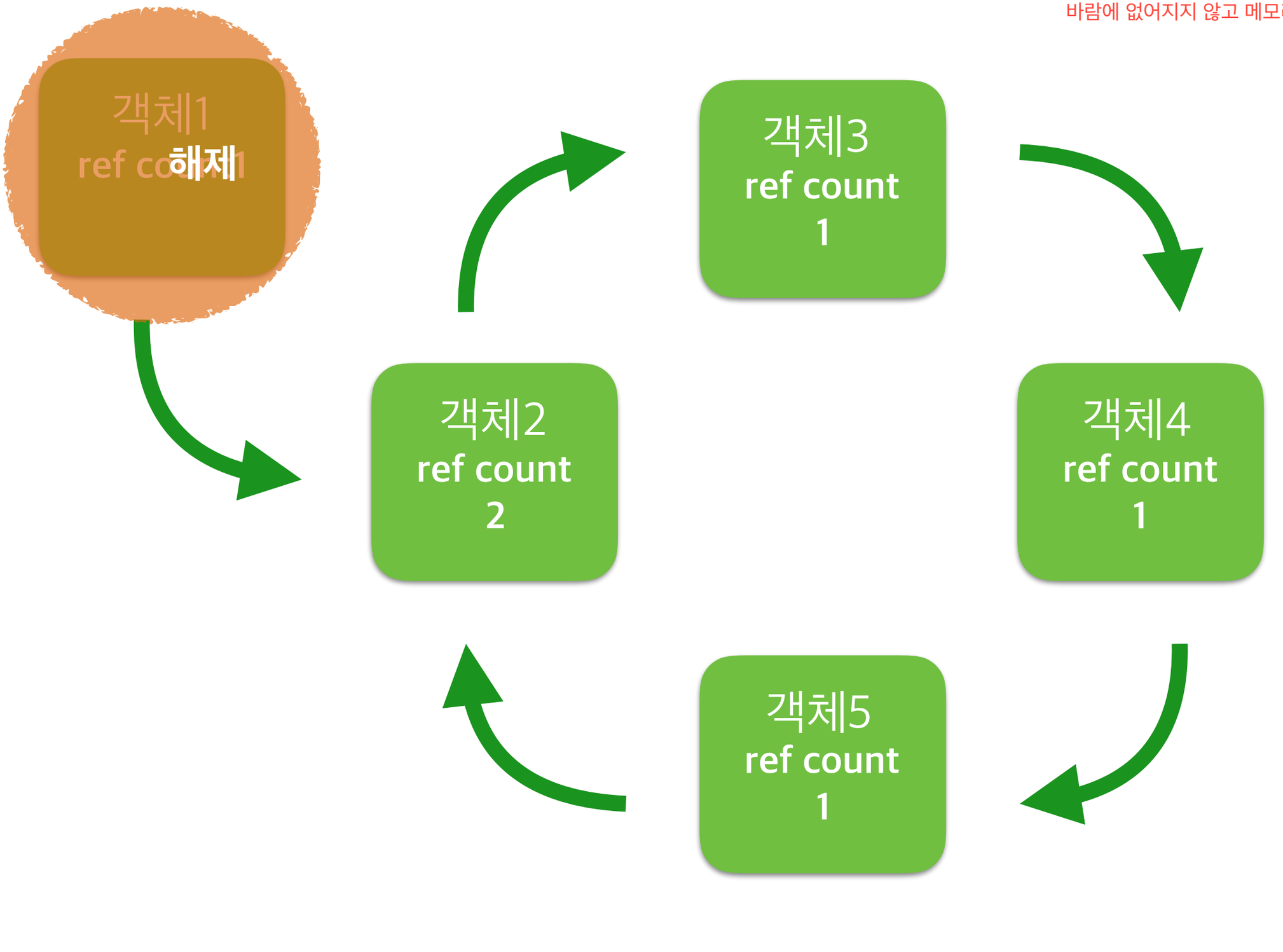


Strong Reference Cycle

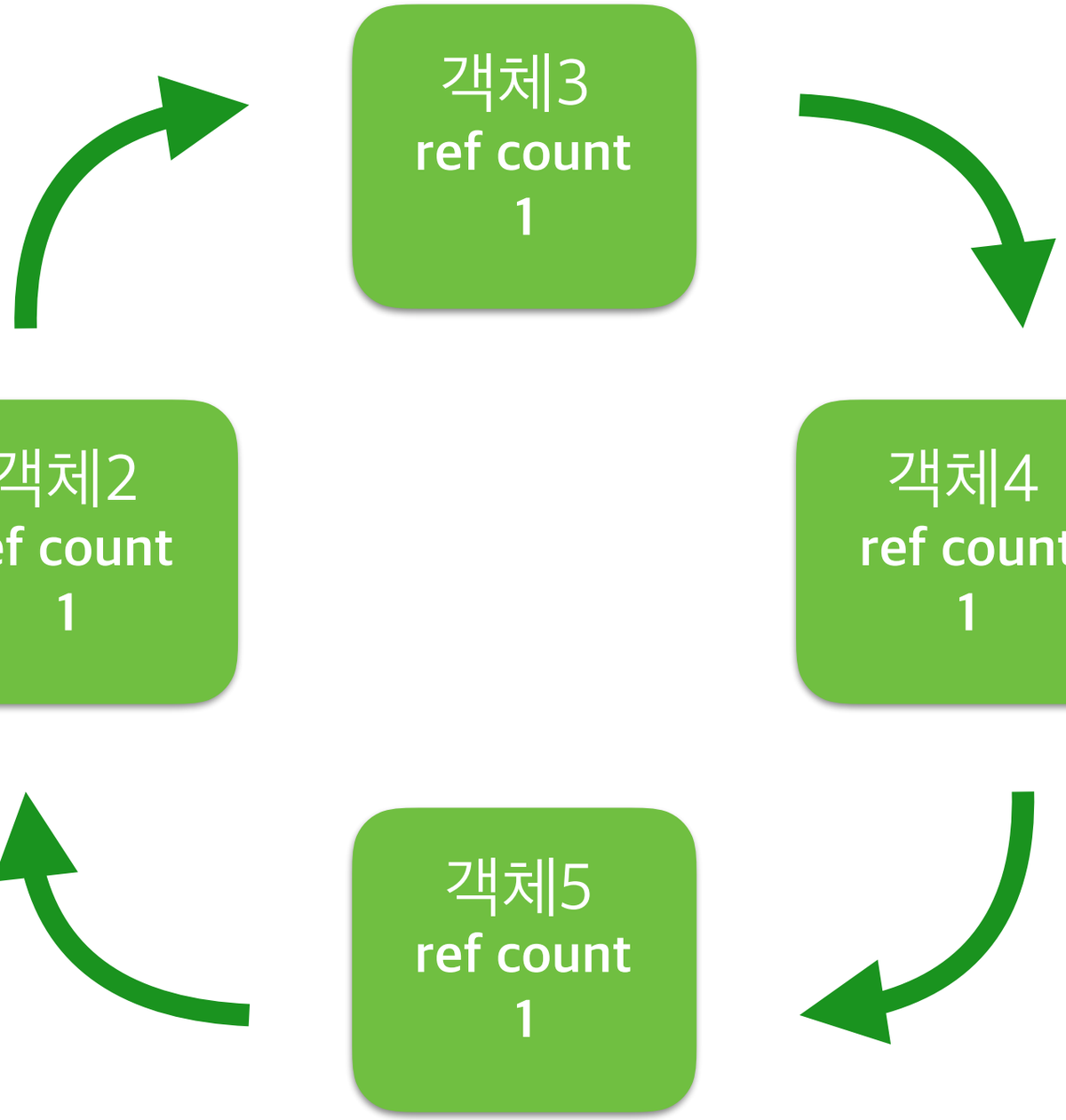


Strong Reference Cycle

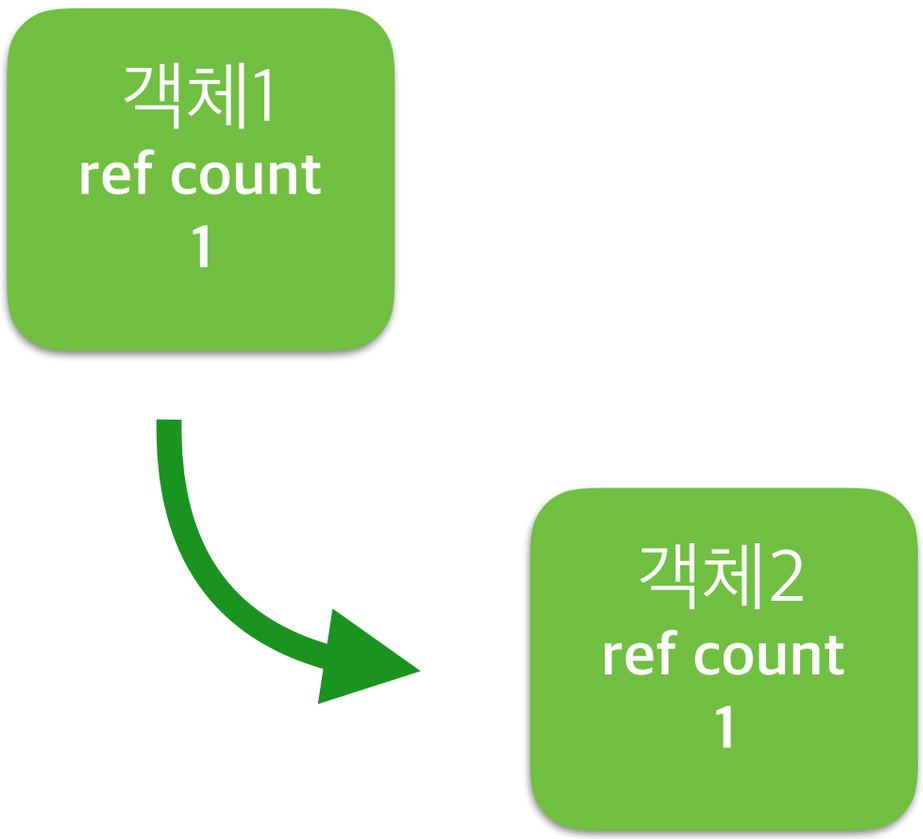
객체 1일 지우면 객체 2에 접근할 방법이 없기때문에 사실 개체2부터 없어져야되는데 객체 5가 1을 올려주는 바람에 없어지지 않고 메모리에서 계속 돌고 있는 상태



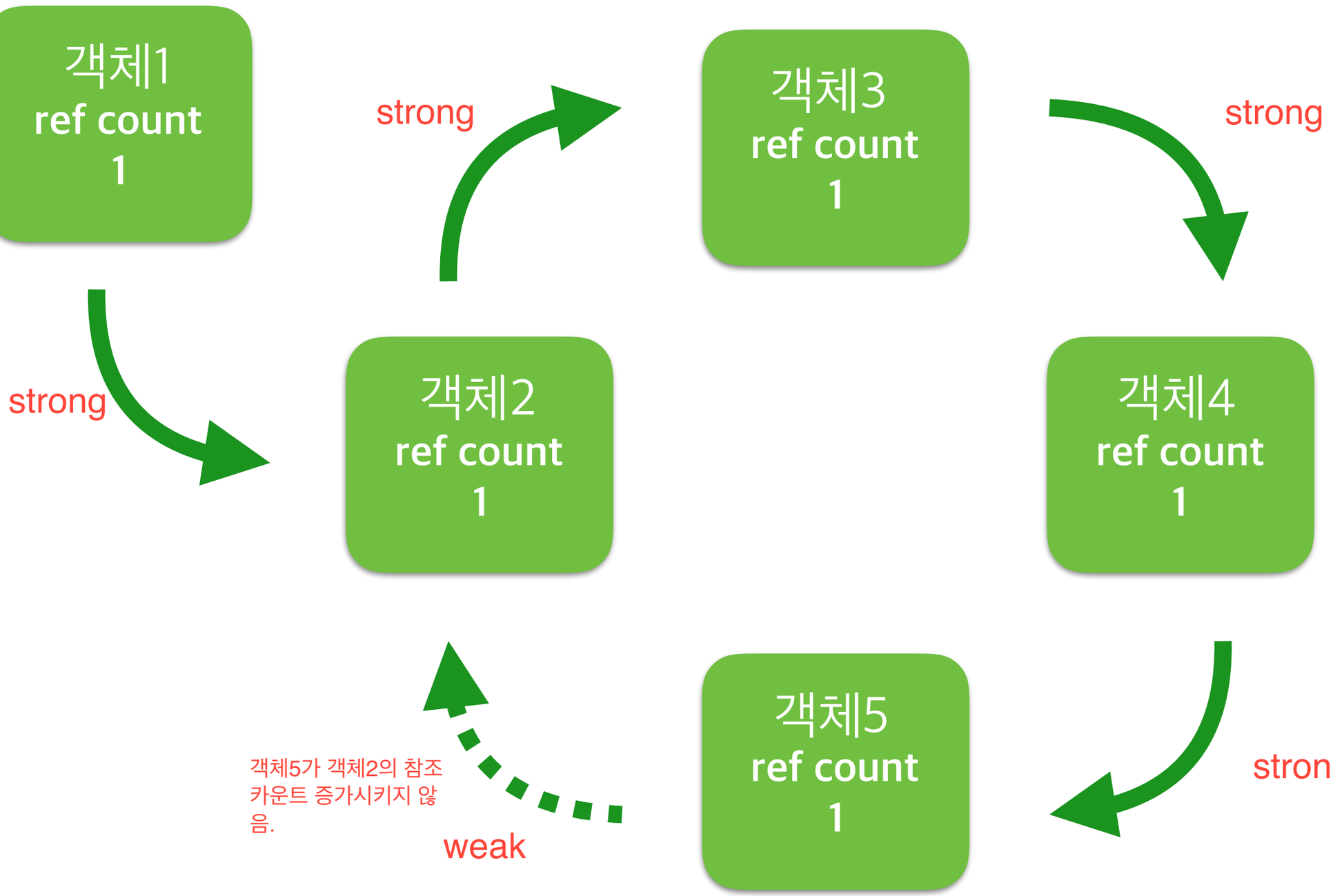
Strong Reference Cycle



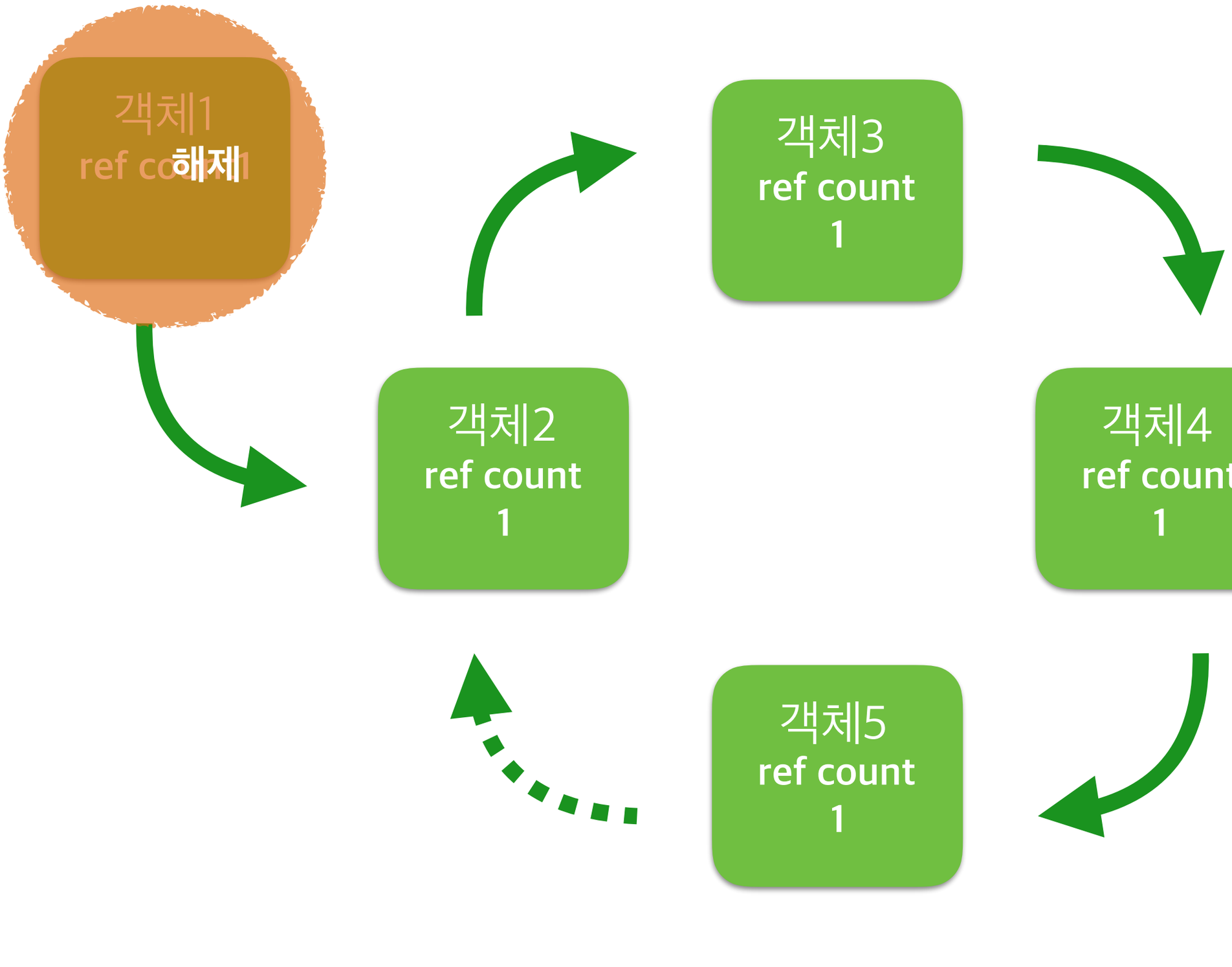
Weak Reference



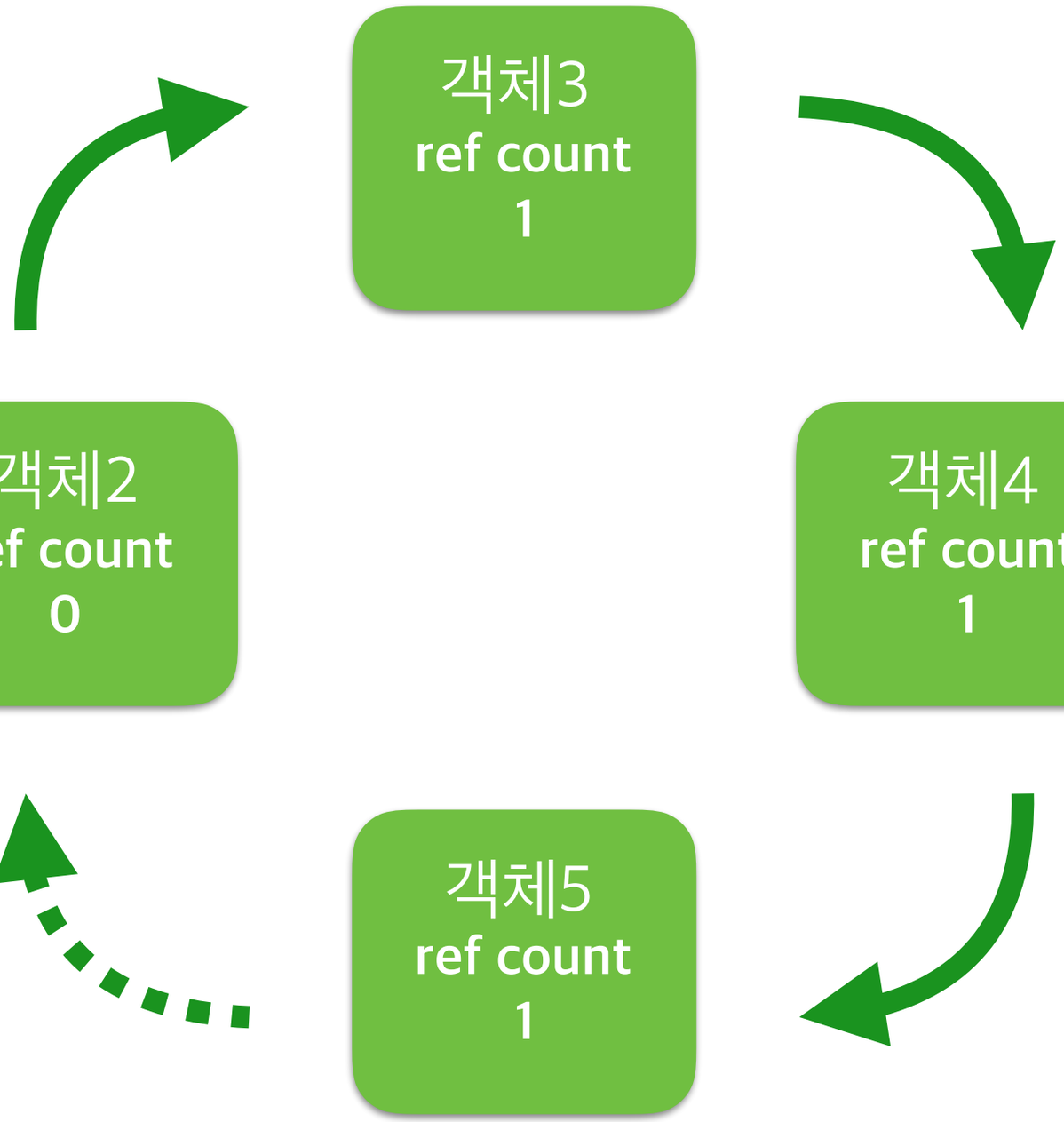
Weak Reference



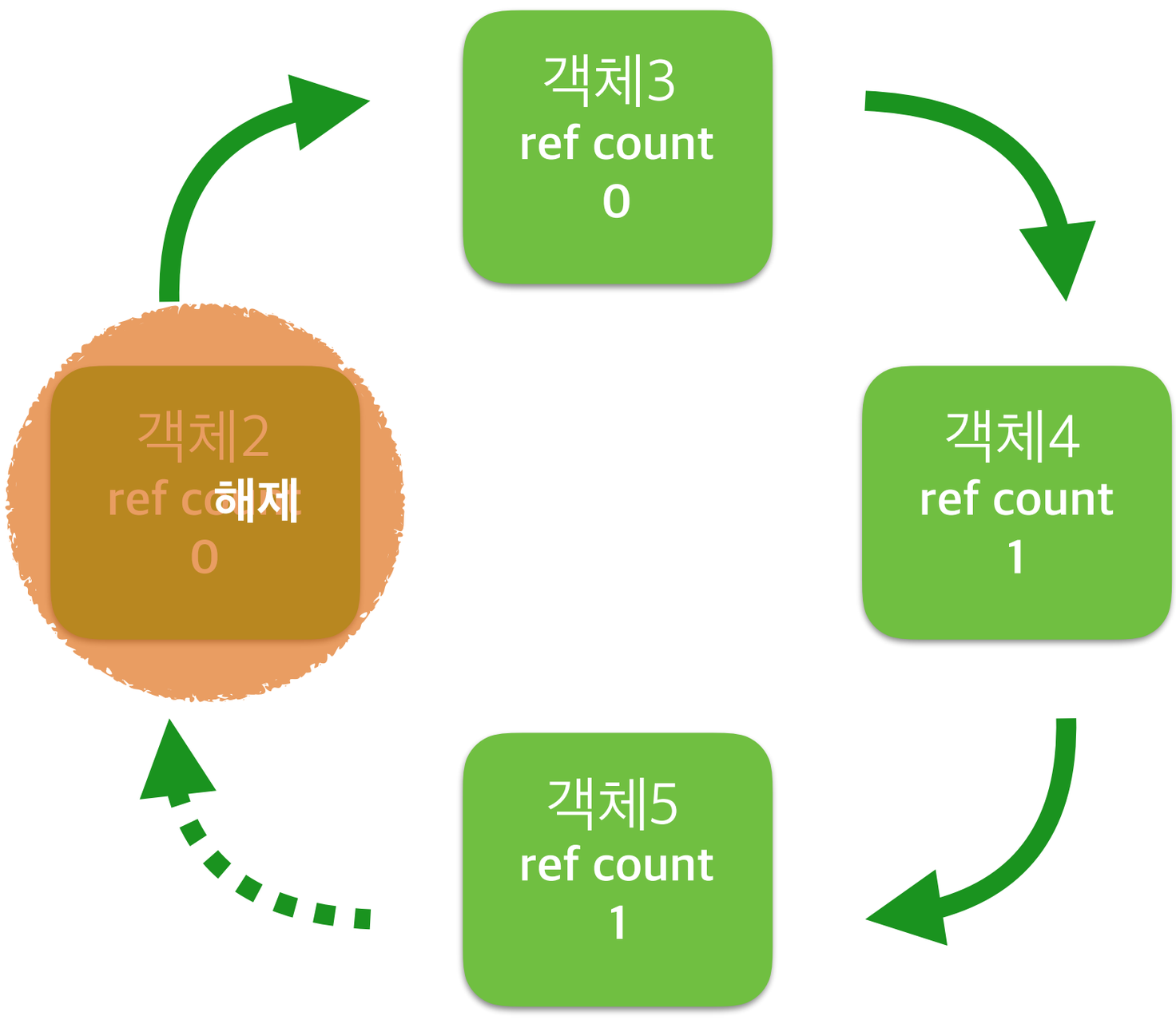
Weak Reference



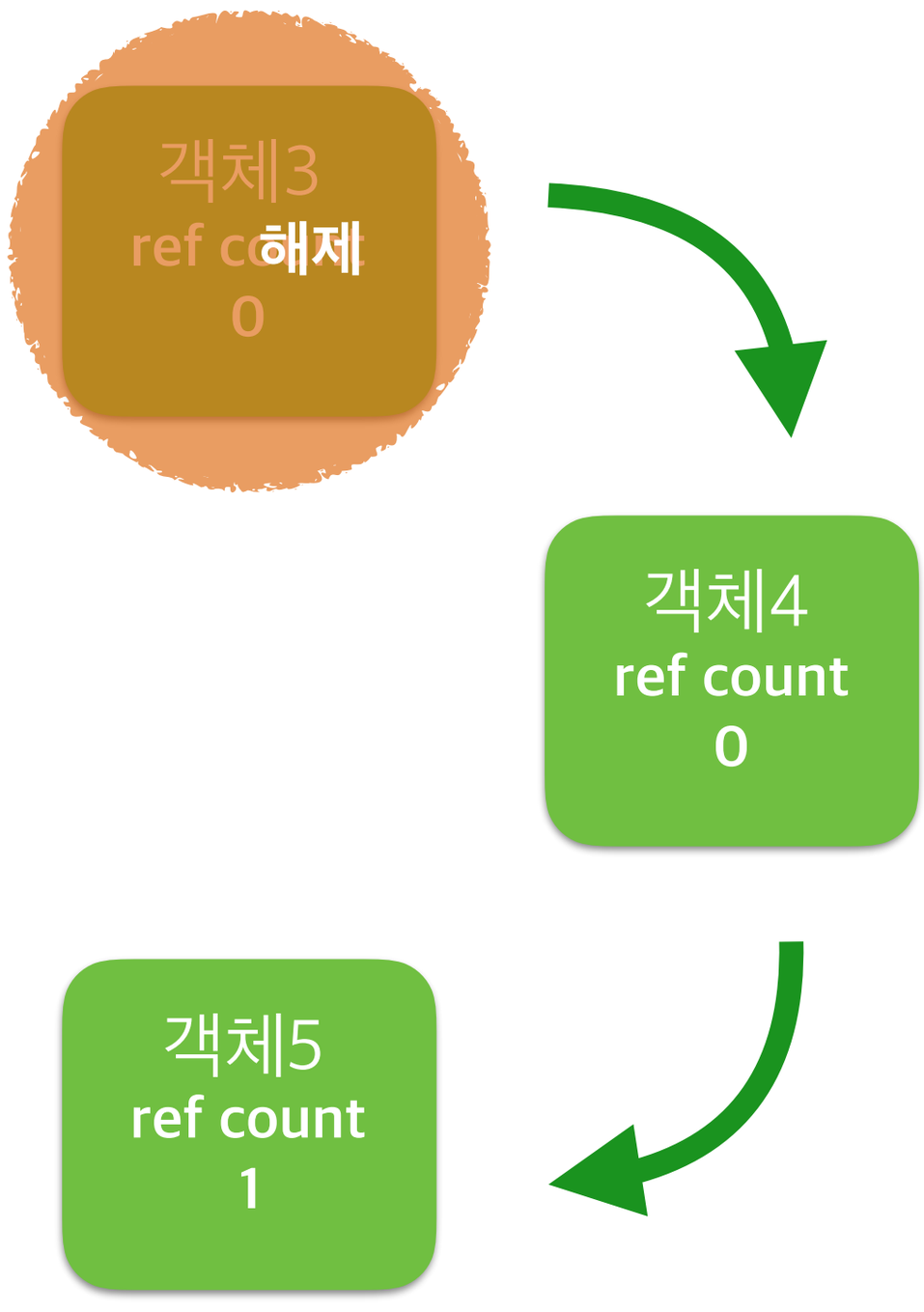
Weak Reference



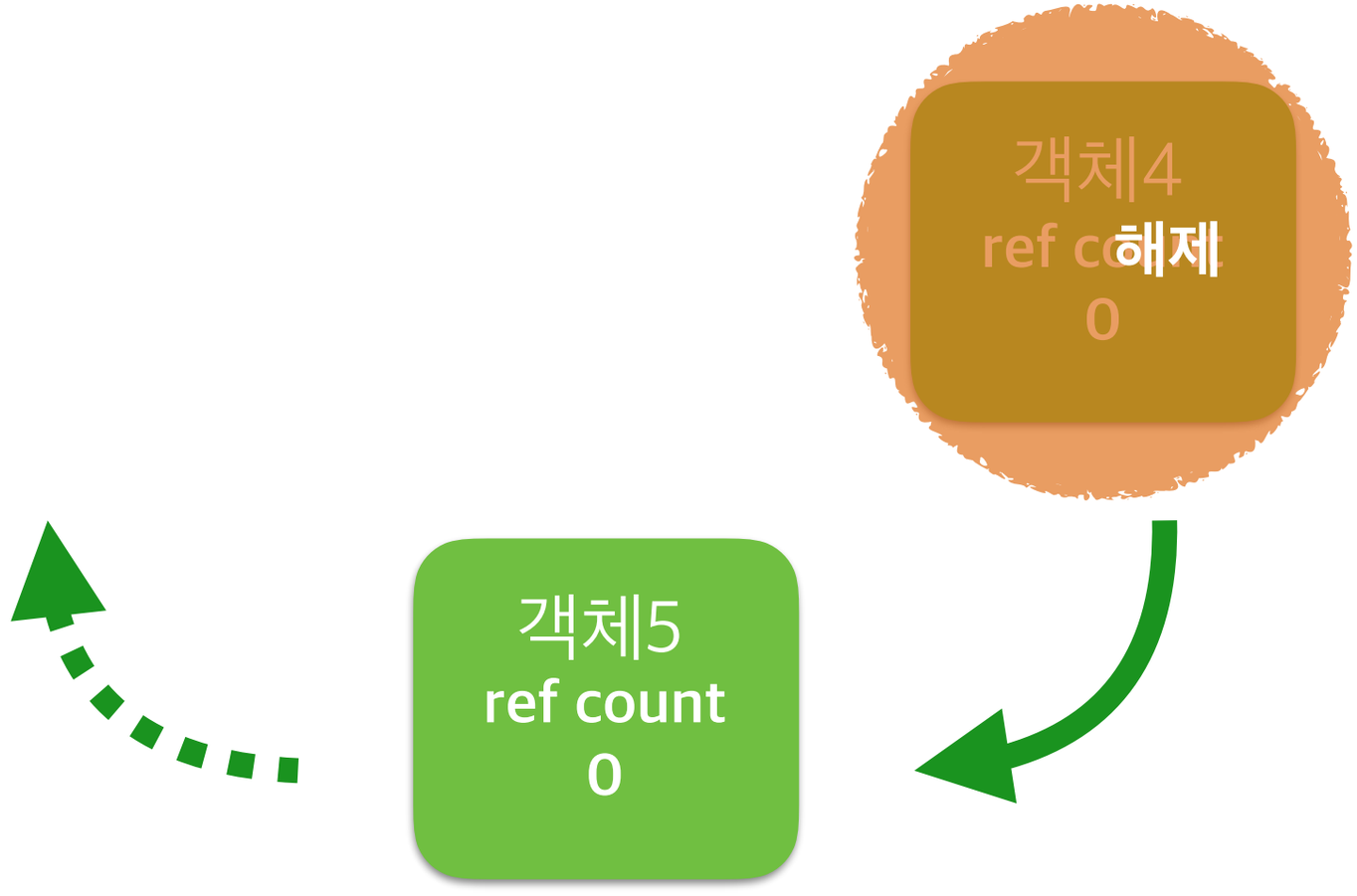
Strong Reference Cycle



Strong Reference Cycle



Strong Reference Cycle



Strong Reference Cycle