
ARC

강사 주영민

메모리 관리 방식

- 명시적 해제 : 모든것을 개발자가 관리함
- 가비지컬렉터 : 가비지 컬렉터가 수시로 확인해서 안쓰는 객체를 해제 시킴 (시스템 부하)
- 레퍼런스 카운팅 : 오너십 정책에 의해 객체의 해제 정의

Reference counting

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

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 management 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

- Strong
- weak

strong 객체 선언

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

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

강한 참조 객체 선언

```
var p1:Person
```

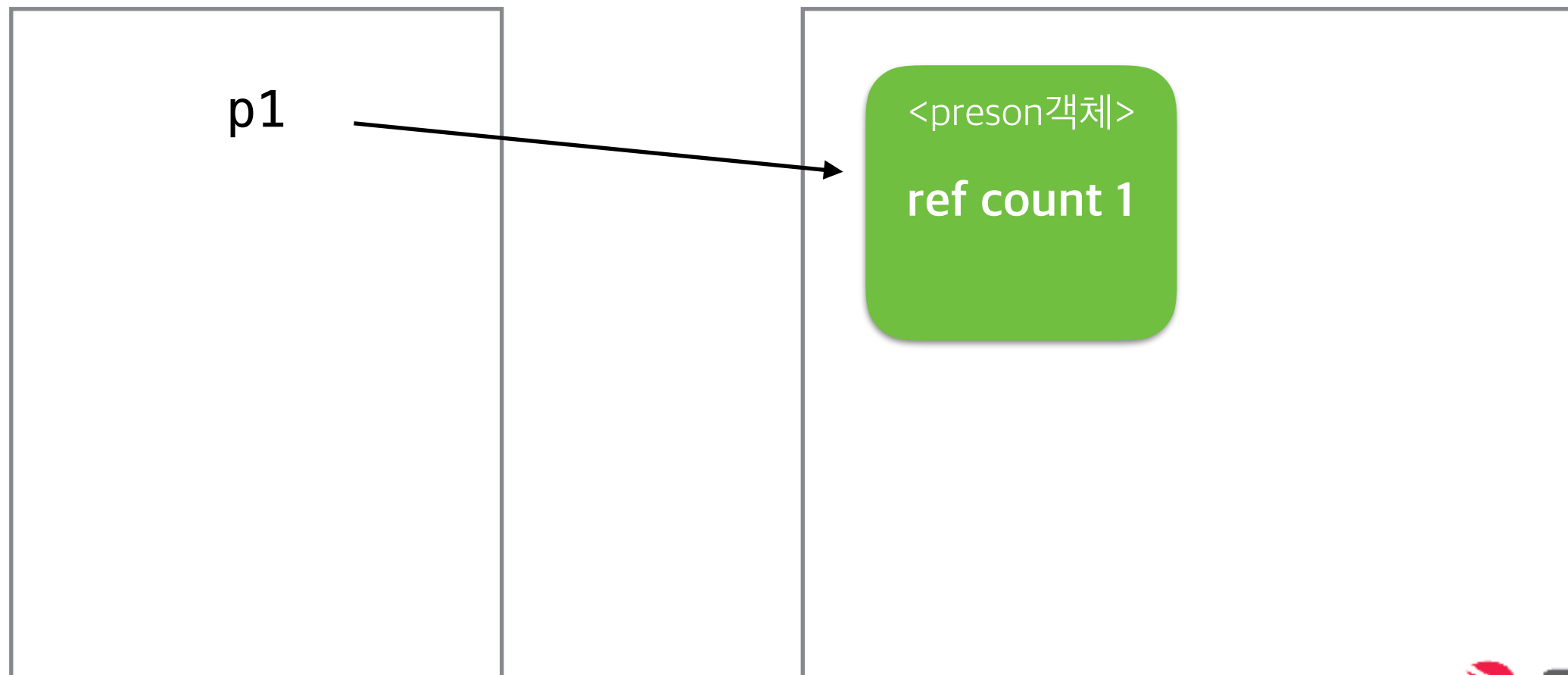
```
var p2:Person
```

할당

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

```
p1 = Person()
```

객체 할당



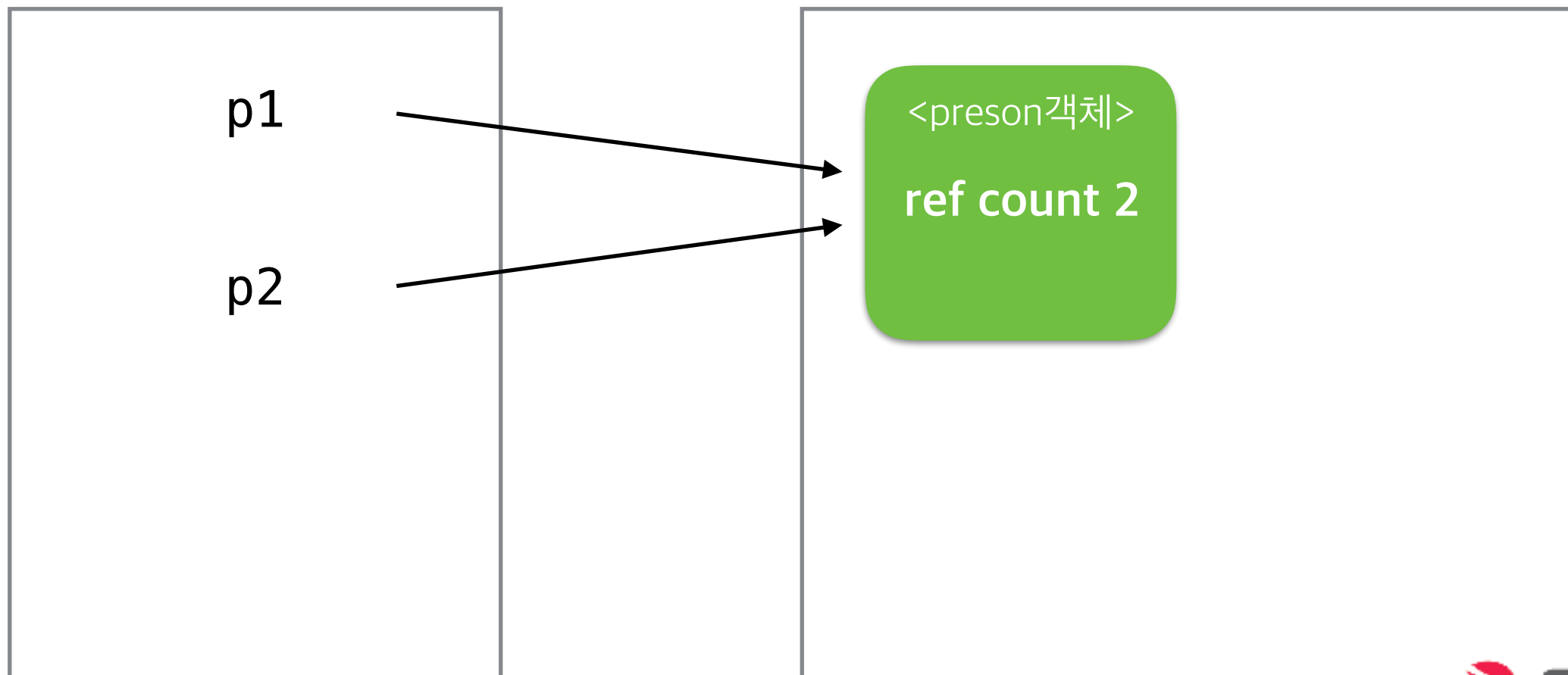
할당

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

```
p1 = Person()
```

```
p2 = p1
```

객체 할당



할당

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

두 변수는 **strong** 지시어로 만들었기 때문에

객체에 대한 참조 포인트와 소유권

(Ownership)을 가지고 있다.

즉 할당이 될 때마다 reference count가
증가 된다.



weak 객체 선언

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

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

강한 참조 객체 선언!

```
var p1:Person
```

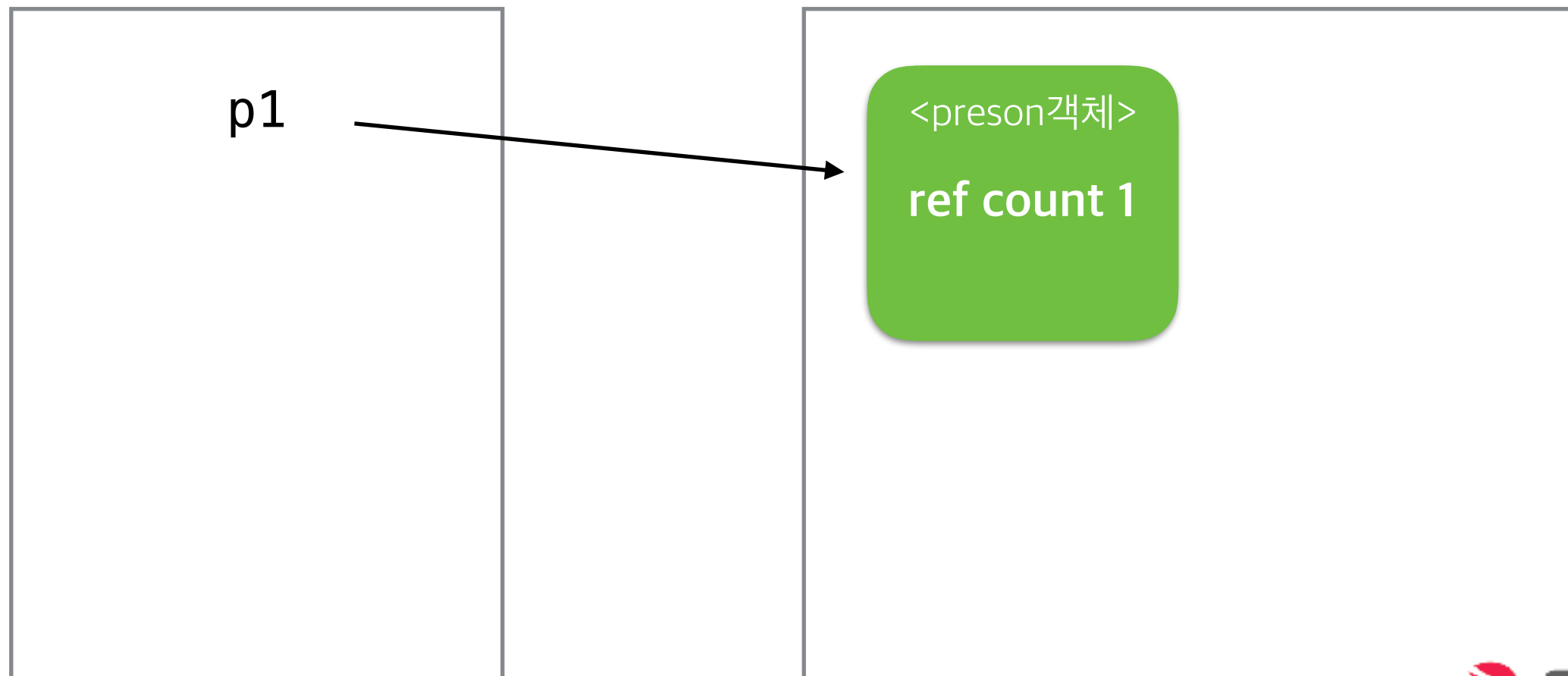
```
weak var p2:Person
```

할당

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

```
p1 = Person()
```

객체 할당



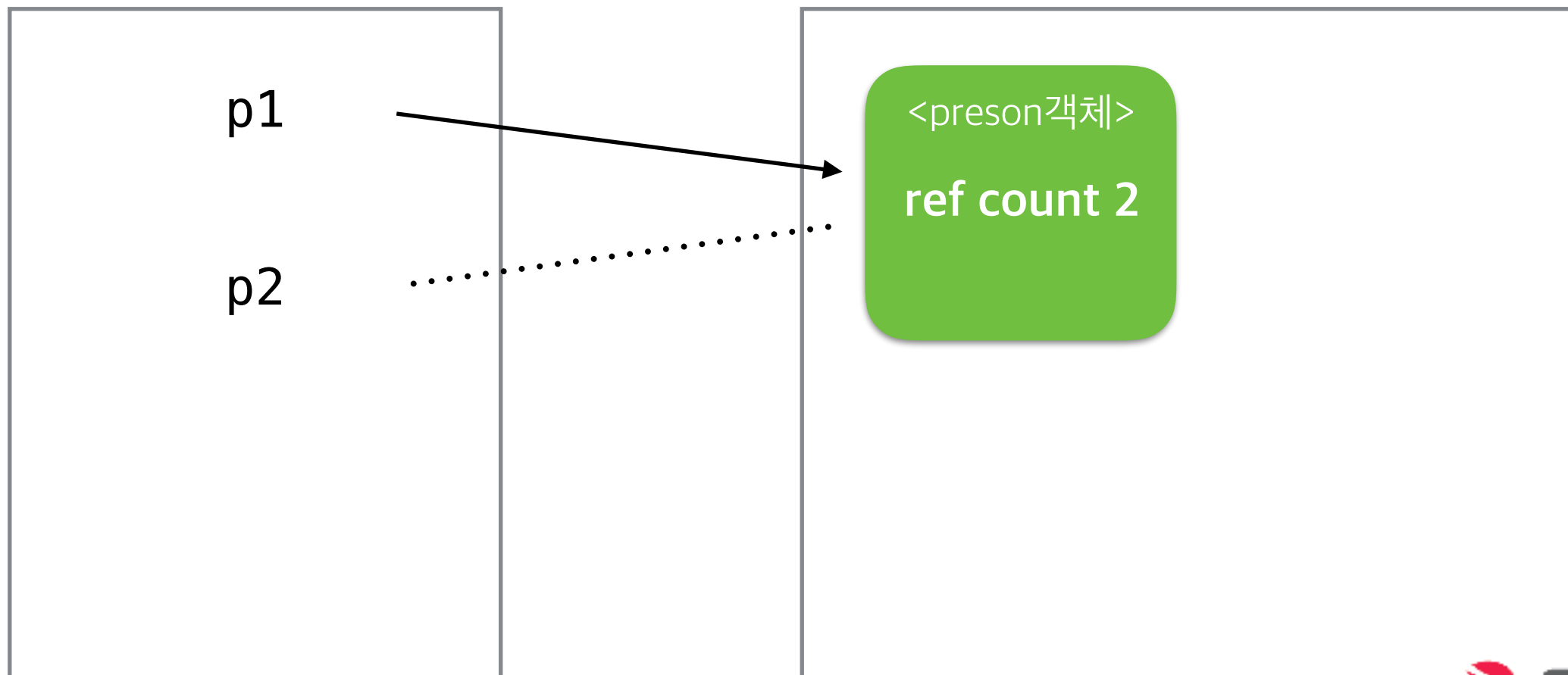
할당

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

```
p1 = Person()
```

```
p2 = p1
```

객체 할당



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()

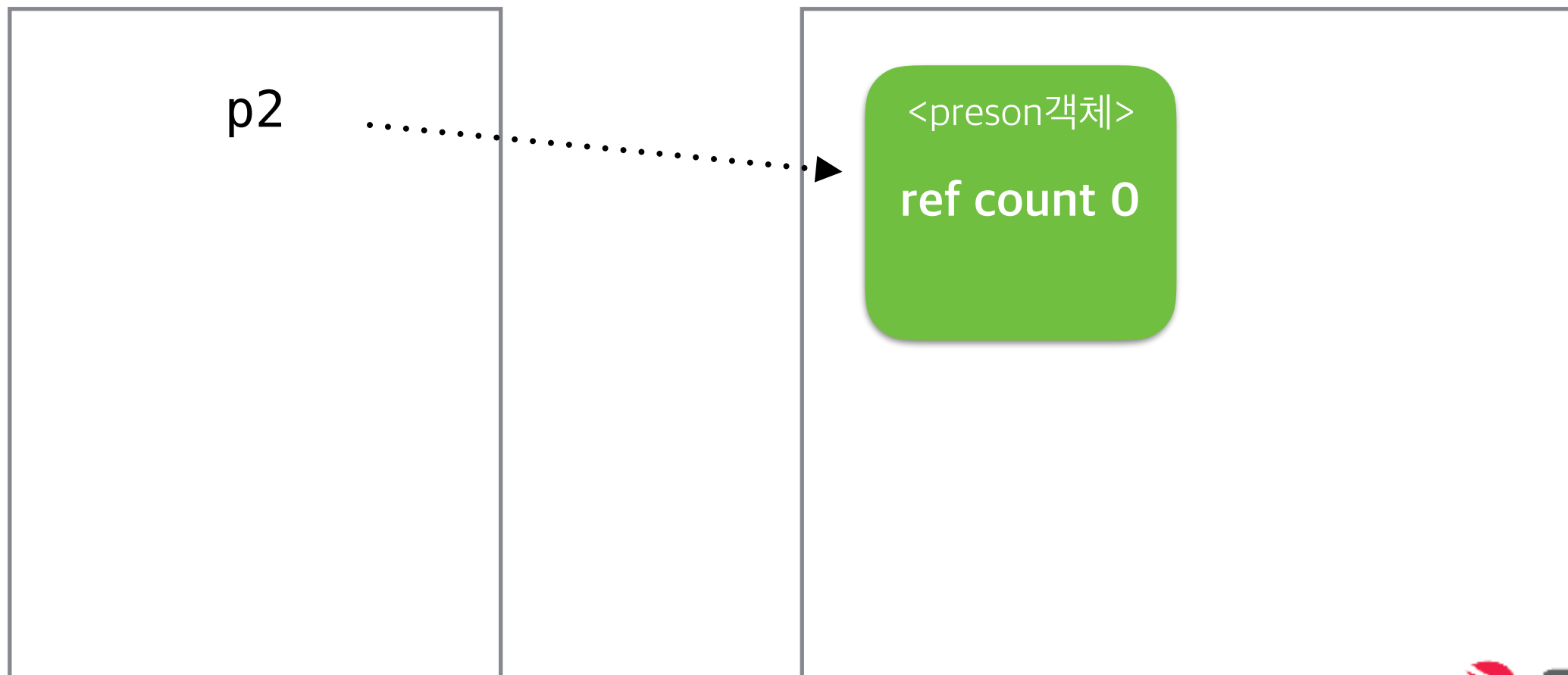
객체 할당???

만약 약한 참조로 만든 p2에 객체를 만들고 할당을 한다면?

weak - 할당

```
p2 = Person()
```

객체 할당



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

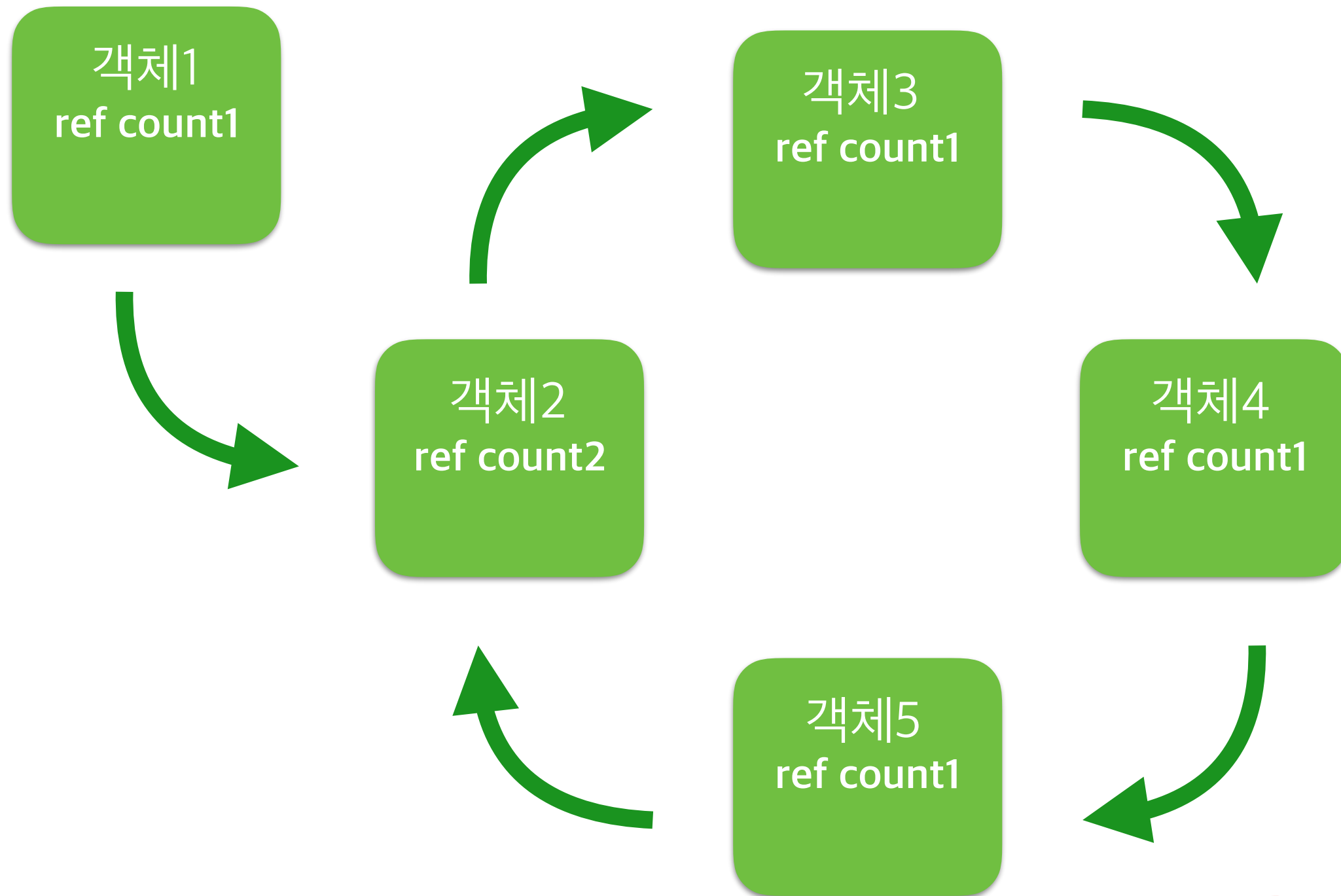
```
p2 = nil
```



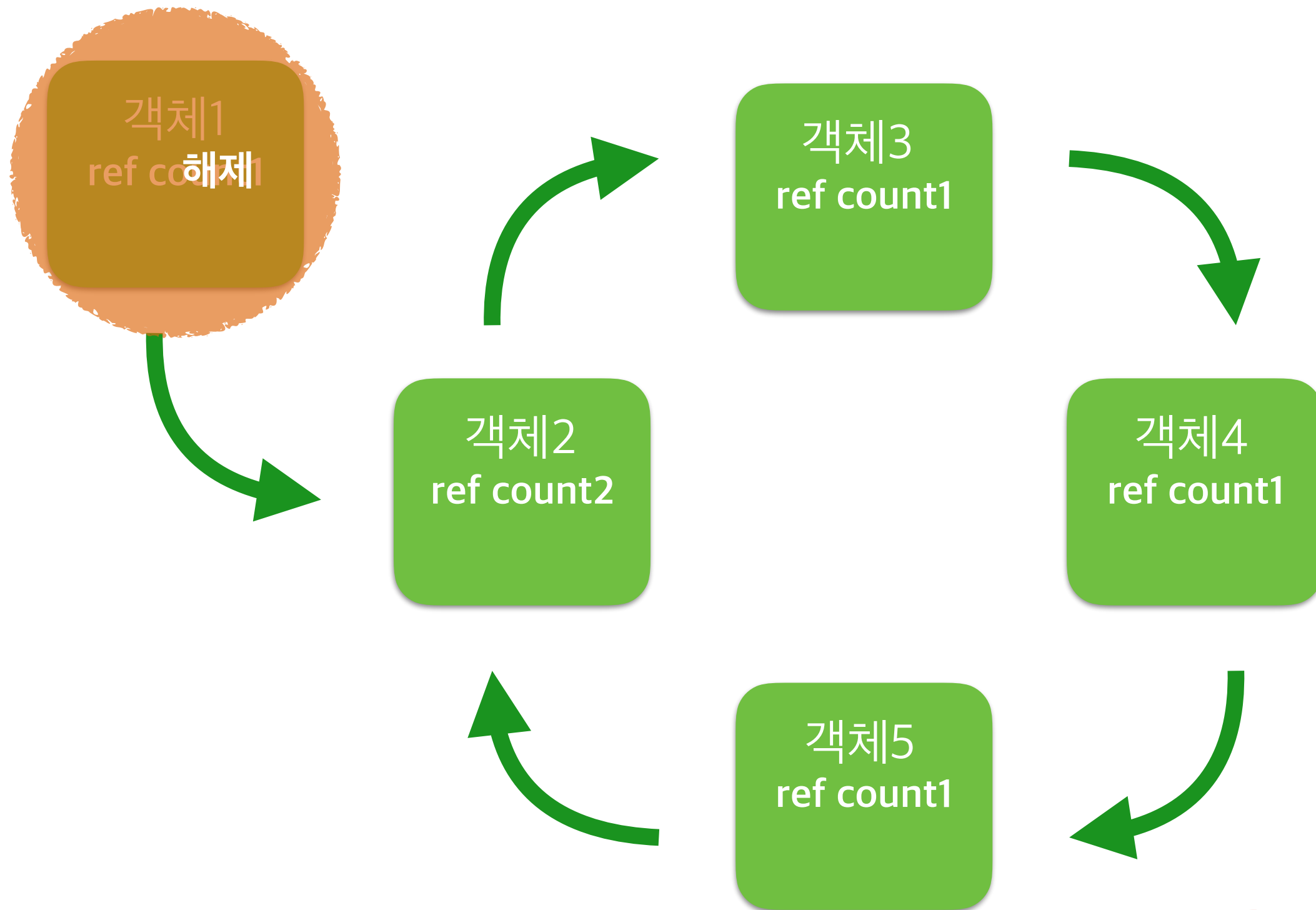
소멸

“반푼0이 weak!! 왜 사용하는 것인가??”

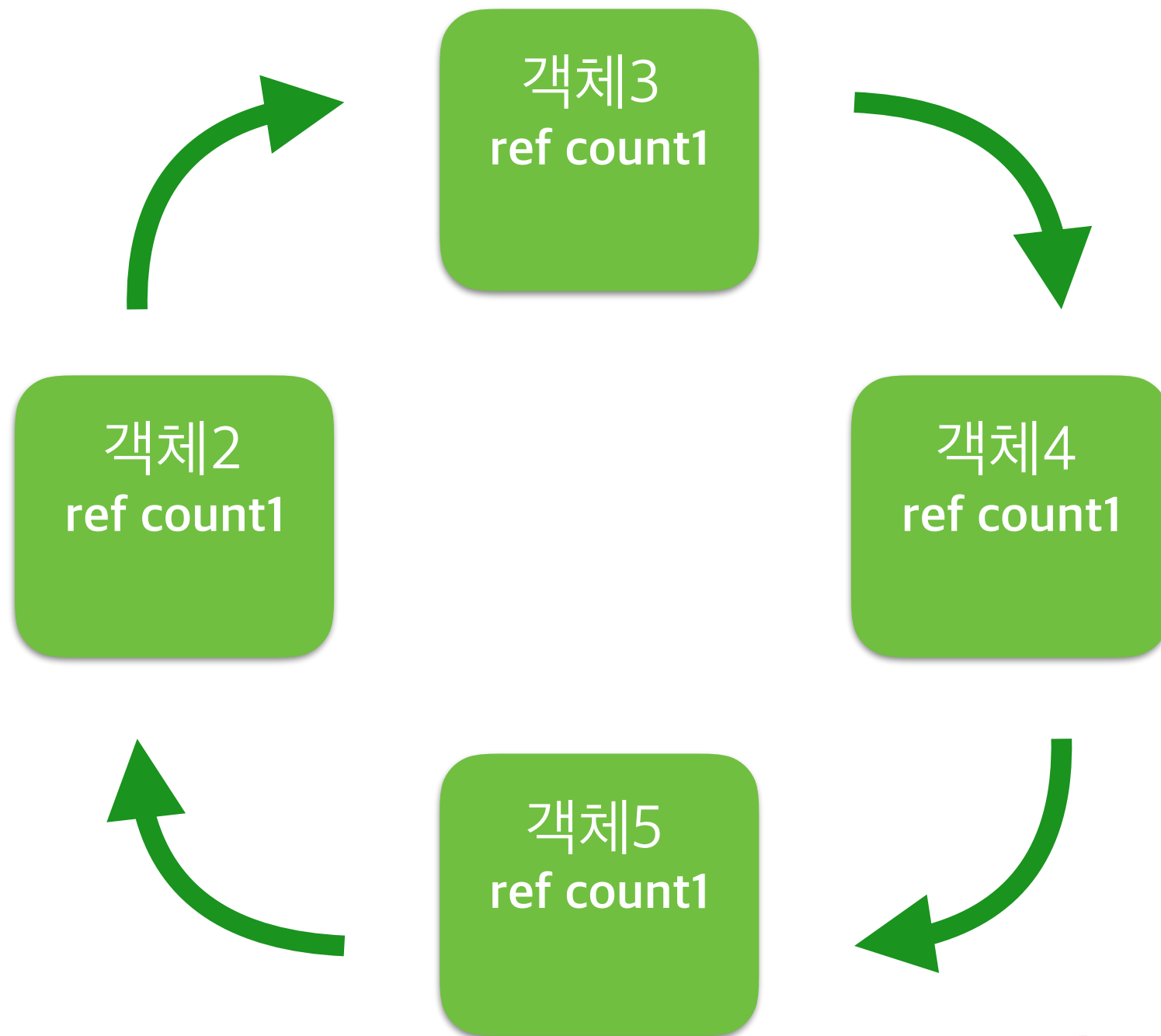
순환 참조



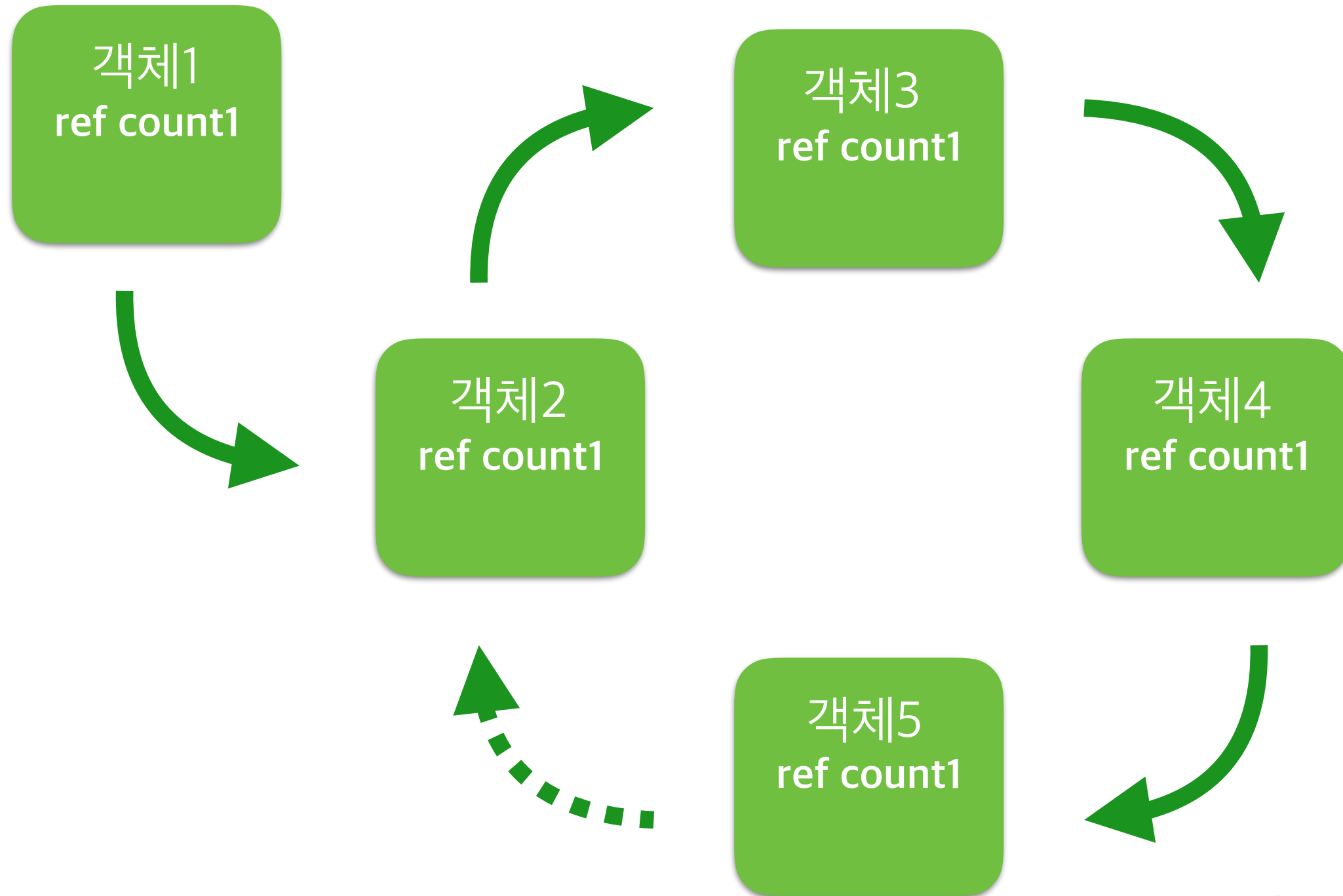
순환 참조



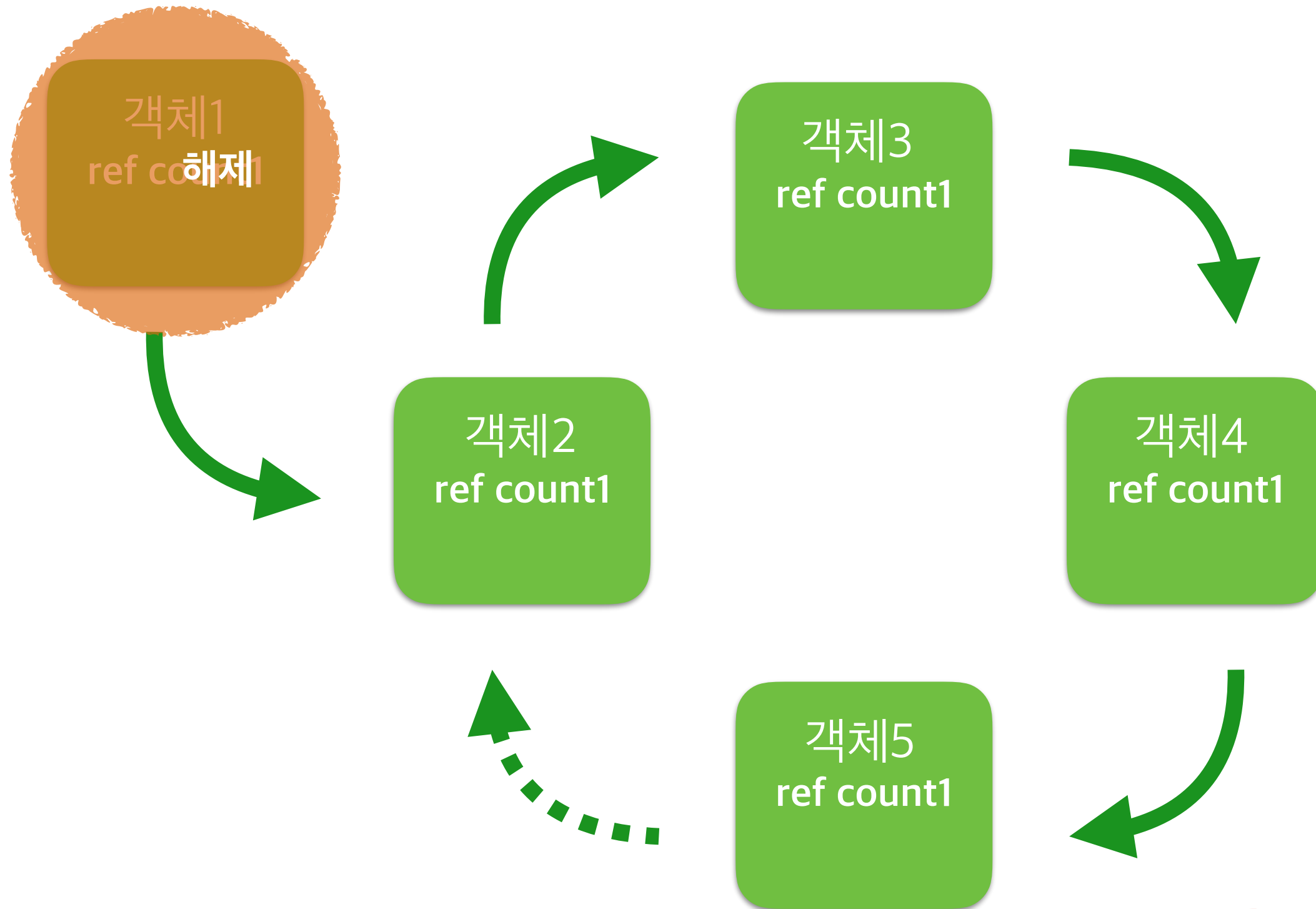
순환 참조



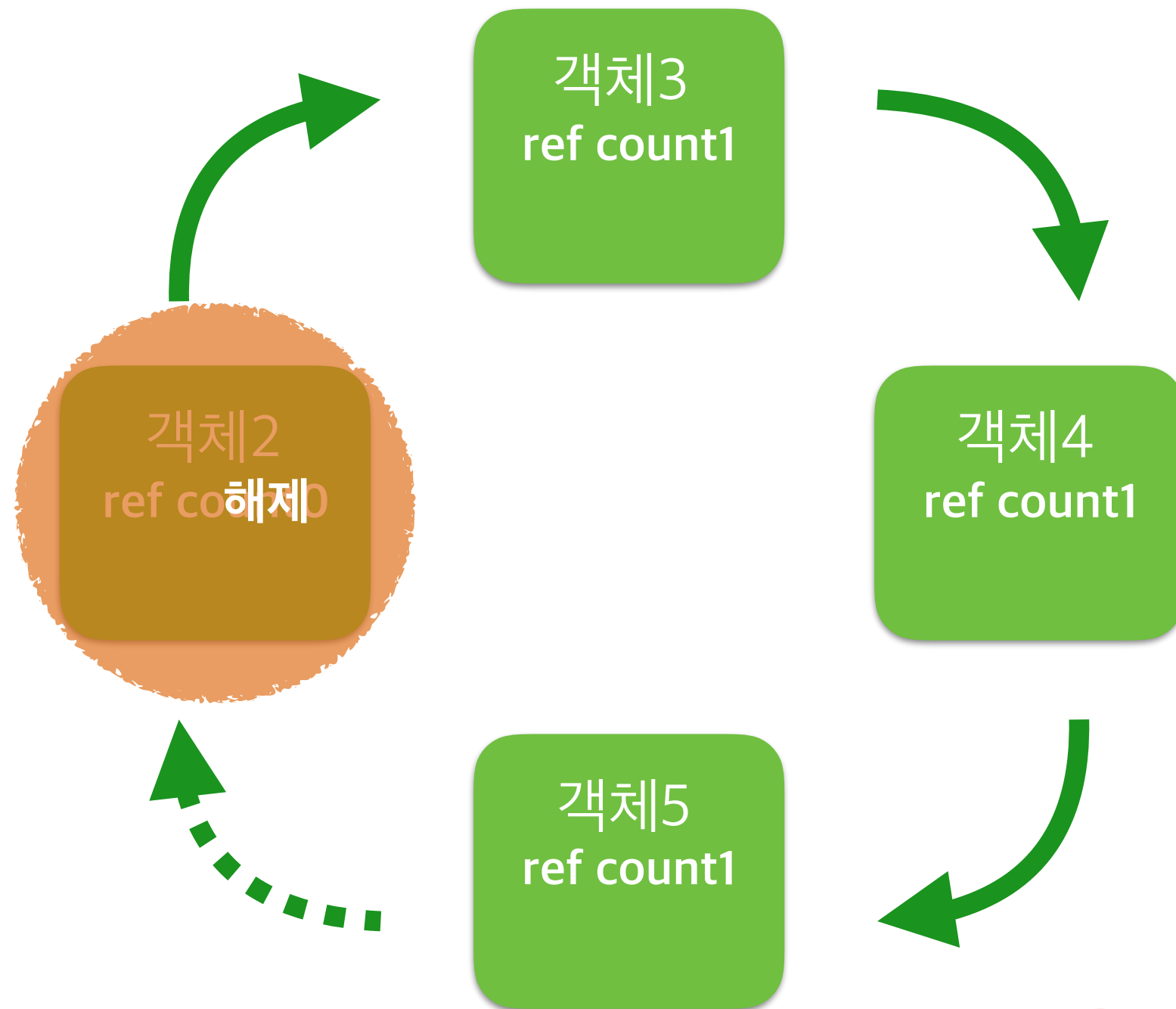
순환 참조



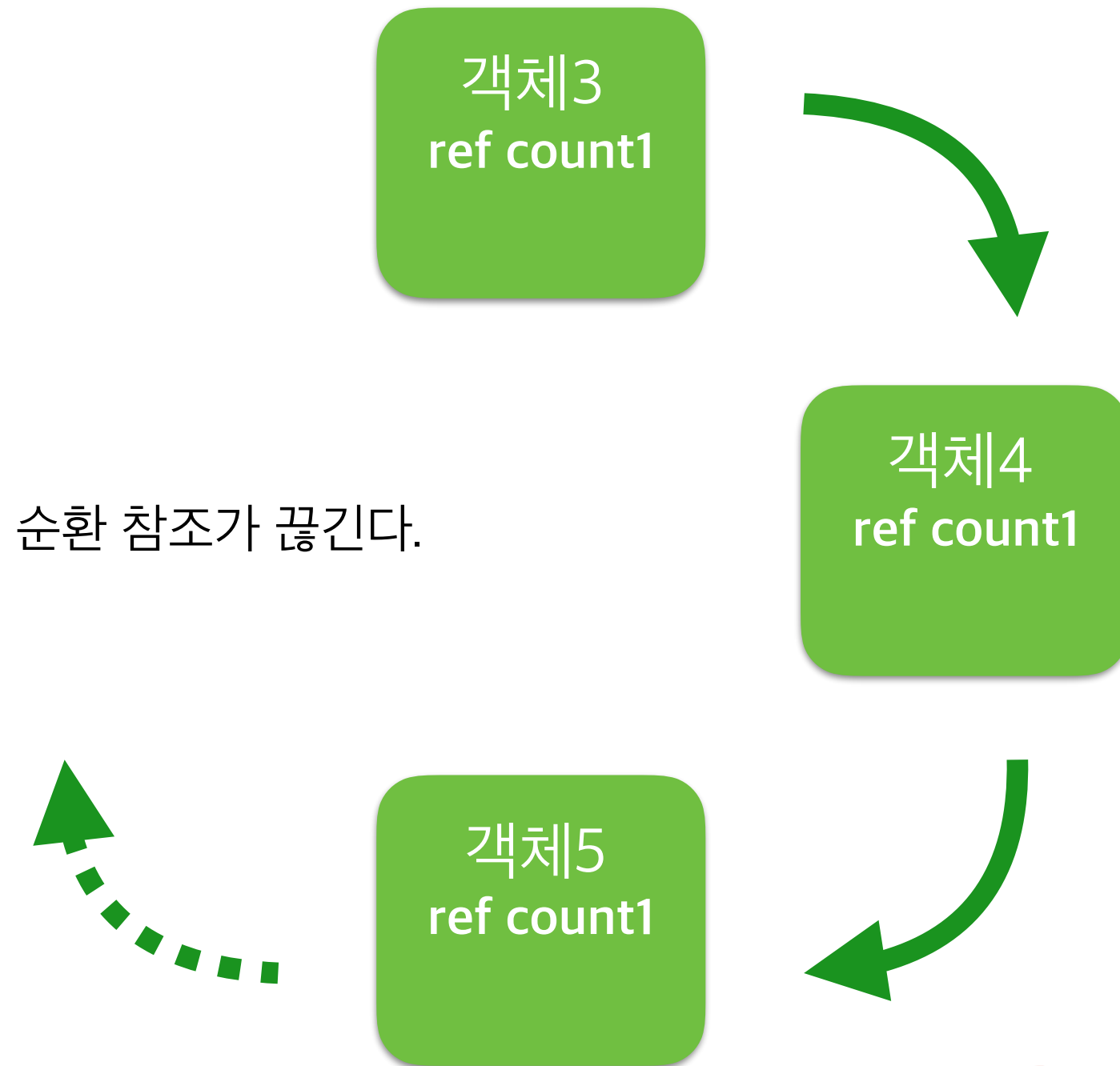
순환 참조



순환 참조



순환 참조



weak pointer 사용 이유

- 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.
 2. Weak : nil 일수도 있다