공식문서 읽기 : Automatic Reference Counting

Automatic Reference Counting - The Swift Programming Language (Swift 5.5)

Swift uses Automatic Reference Counting (ARC) to track and manage your app's memory usage. In most cases, this means that memory management "just works" in Swift, and you don't need to think about memory management yourself. ARC automatically frees up the memory used by class

https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html

Swift에서 Automatic Reference Counting(ARC)은 앱의 메모리 사용을 추적하고 관리한다.

즉, class 객체에 의해 사용되는 메모리들을 더이상 객체가 사용하지 않을 때 자동으로 수거한다.

따라서 C언어처럼 직접 메모리를 할당하고 수거하지 않아도 되지만, 몇몇 케이스는 직접 신경을 써야 한다.

Reference Counting은 class의 인스턴스에만 적용된다. 단어 그대로 reference counting은 참조타입에만 해당하므로 구조체나 열거형과 같은 값타입은 해당되지 않는다.

How ARC Works

Swift 공식 문서에 따르면 ARC는 다음과 같이 작동한다.

클래스의 객체를 생성할 때마다 ARC는 해당 객체에 대한 정보를 저장하기 위해 메모리를 할당한다. 이 메모리는 해당 객체의 type 정보와 객체의 값을 갖는다.

객체가 더 이상 쓸모가 없어지면 즉, 참조를 당하지 않는다면 ARC는 해당 메모리를 해제해 다른 곳에 사용될 수 있도록 한다. class 객체가 사용되지 않는다면 메모리의 일부를 차지하지 않는다는 것을 보장하므로 메모리의 관리에 있어 효율적이다.

하지만

ARC가 사용되고 있는 객체를 메모리에서 할당 해제하면 더 이상 해당 객체의 프로퍼티나 메서드로의 접근은 불가능해지며, 접근을 시도할 경우 crush가 발생한다.

따라서 위와 같은 일이 발생하지 않도록 ARC는 class 객체를 참조하는 속성, 상수 및 변수들의 수를 추적하고 관리한다. 그리고 하나라도 객체를 참조하고 있다면 ARC는 해당 객체는 사용중이라고 판단해 메모리에서 그 객체를 유지한다.

위와 같이 ARC가 계속해서 추적하고 관리하게 하기 위해서는 클래스 객체의 속성, 상수 및 변수들은 해당 객체와 strong reference의 관계가 되어야 한다. 이러한 참조는 강한 참조(strong reference)라고 하는데 그 이유는 강한 참조가 남아있는한 해당 인스턴스에 대한 메모리에서의 할당 해제는 일어나지 않기 때문이다.

ARC in Action

아래의 코드를 통해 ARC가 어떻게 작동하는지 확인해볼 수 있다.

Person이라는 클래스가 있고, 이 클래스는 name이라는 저장속성을 가지고 있다.

initializer는 name에 값을 설정하고 print를 하며, deinitializer도 print를 한다.

```
class Person {
    let name: String
    init(name: String) {
        self.name = name
        print("\(name) is being initialized")
    }
    deinit {
        print("\(name) is being deinitialized")
    }
}
```

Person class의 객체가 생상되면 init이 호출되고, 할당 해제되면 deinit이 호출된다.

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

Person type을 가질수도 있고, 그렇지 않을 수도 있는 세개의 변수를 정의한다.

```
reference1 = Person(name: "Jiwon")
// ARC 참조 카운트 +1
// prints "Jiwon is being initialized"
```

그리고 reference1에는 Person 객체를 생성해 할당한다. 이 때 initializer에 의해 문장이 출력된다.

이 말은 즉, Person 객체가 reference1 변수에 할당되어 있고 강한 참조가 하나라도 있기 때문에 ARC는 메모리에서 Person을 할당 해제하지 않는다.

만약 여기에서 같은 Person 객체를 남은 두 개의 변수에 할당한다면 어떻게 될까?

```
reference2 = reference1
// ARC 참조 카운트 +1
reference3 = reference1
// ARC 참조 카운트 +1
```

그럼 이제 두 개의 강한 참조가 추가로 생긴다. 따라서 총 세 개의 강한 참조가 Person 객체에 존재하게 된다.

여기에서 두 개의 변수에 nil을 할당해 강한 참조를 끊어주게 되어도, 하나의 강한 참조가 존재하기 때문에 ARC는 Person 객체를 메모리에서 할당 해제하지 않는다.

```
reference1 = nil
// ARC 참조 카운트 -1
reference2 = nil
// ARC 참조 카운트 -1
```

ARC가 Person 객체를 메모리에서 할당 해제하는 시점은?

```
reference3 = nil
// ARC 참조 카운트 -1
// prints "Jiwon is being deinitialized"
```

바로 reference3에 nil을 할당하는 시점 즉, 더 이상 Person 객체를 사용하지 않을 경우 참조 카운트가 0이 되면 deinit이 발생한다.

Strong Reference Cycle Between Class and Instance (순환참조)

위의 예시에서 Person 객체가 생성될 때마다 참조되는 횟수를 카운트 할 수 있었고, 더 이상 사용하지 않을 때 메모리에서 할당 해제될 때도 이를 추적하고 관리할 수 있었다.

그러나 코드를 잘못 작성하게 되면 클래스 객체의 참조 카운트가 절대로 0이 될 수 없는 상황이 생길 수 있다. 이 문제는 클래스 객체가 서로 강한 참조를 유지해 각 객체가 다른 객체를 참조하는 즉, 활성 상태로 유지되는 경우에 발생할 수 있는데, 이를 순환 참조(strong reference cycle)라고 한다.

```
class Person {
    let name: String
    init(name: String) {
        self.name = name
    }
    var apartment: Apartment?
```

```
    deinit {
        print("\(name) is being deinitialized")
    }
}

class Apartment {
    let unit: String
    init(unit: String) {
        self.unit = unit
    }
    var tenant: Person?
    deinit {
        print("Apartment \(unit) is being deinitialized")
    }
}
```

Person, Apartment 클래스가 있다.

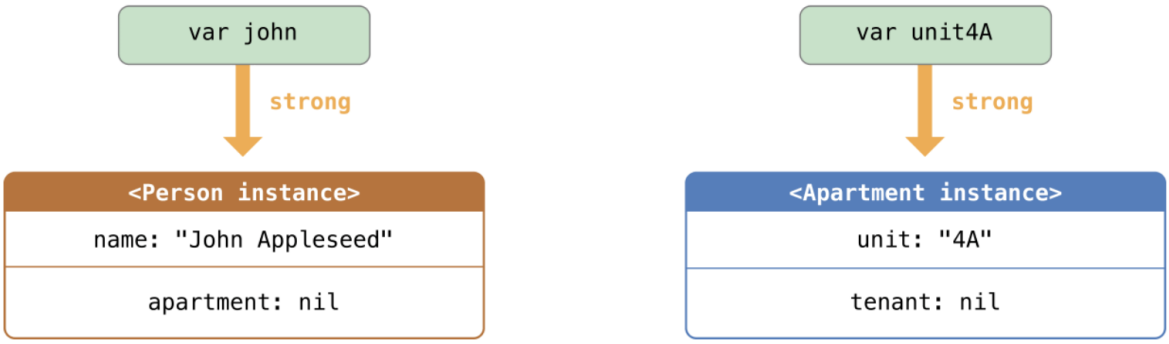
Person 클래스는 name, apartment를 가지고 있는데, 아파트가 없는 사람이 있을 수 있기 때문에 apartment는 optional type이다.

Apartment 클래스는 unit, tenant를 가지고 있는데, 비어있는 아파트가 있을 수 있기 때문에 tenant는 optional type이다.

```
var john: Person?
var unit4A: Apartment?

john = Person(name: "john Appleseed")
unit4A = Apartment(unit: "4A")
```

Person 인스턴스 변수인 jiwon, Apartment 인스턴스 변수인 unit4A를 생성하고 그 값을 할당한다.

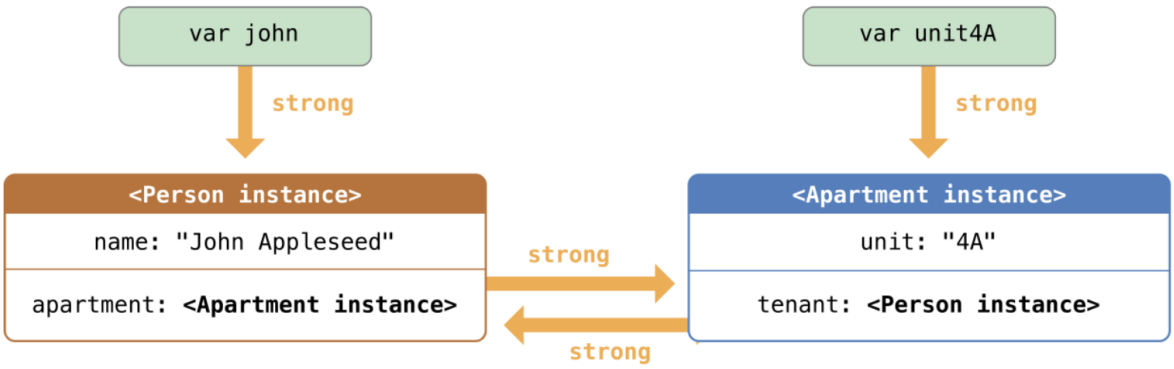


이때 john과 unit4A 변수는 Person 인스턴스와 Apartment 인스턴스에 위의 그림과 같이 강한 참조를 가지게 된다.

```
john?.apartment = unit4A
unit4A?.tenant = john
```

그리고 john은 unit4A를 소유하게 되고, unit4A의 세입자로는 john이 온다.

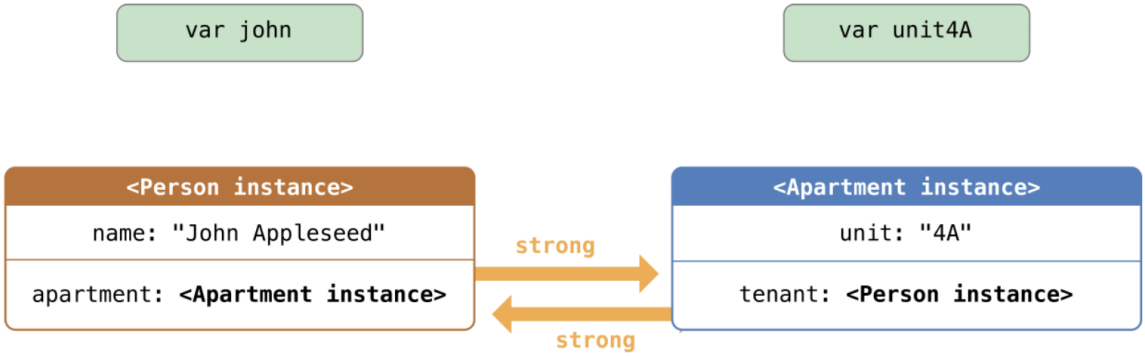
위의 코드처럼 두 인스턴스를 연결하게 되면 인스턴스 간에 순환 참조(strong reference cycle)가 생성된다.



위 그림처럼 Person의 인스턴스와 Apartment의 인스턴스 사이에도 강한 참조 결합이 생성되는 것을 확인할 수 있다. 즉, Person의 인스턴스는 strong reference로 Apartment 인스턴스를 가지고, Apartment 인스턴스도 strong reference로 Person 인스턴스를 가지게 된다.

```
john = nil
unit4A = nil
```

따라서 john과 unit4A 변수가 가지는 strong reference에 nil을 할당함으로써 해제시켜도 참조 카운트의 값은 0으로 떨어지지 않게 되어 ARC에 의해 메모리에서 할당 해제되지 않는다. 할당 해제가 발생하지 않았기 때문에 deinit도 호출되지 않는다.



위의 그림과 같이 Person 인스턴스와 Apartment 인스턴스 사이에는 여전히 강한 참조 결합이 남아있으며 이들 인스턴스에 접근할 수 있는 방법이 없어 두 인스턴스 사이의 강한 참조 결합은 깨지지 못해 메모리 누수(memory leak)가 발생하는 것이다.

Resolving Strong Reference Cycles Between Class Instances (순환 참조 해결)

순환 참조가 해결되지 않고 쌓이게 되면 메모리 상에는 자리만 차지하는 쓸모없는 인스턴스들이 많아지게 되고, 이는 나중에 메모리 할당이 필요한 인스턴스들이 자리가 없어 할당되지 못해 메모리가 부족하게 되는 현상을 야기할 수 있다.

Swift에서는 순환 참조를 해결하는 방법으로 두 가지를 제안한다.

1. weak reference
2. unowned reference

위 두 가지 참조 방법은 강한 참조 결합을 하지 않고 서로를 참조할 수 있는 방법이다.

Weak Reference / Unowned Reference 차이점

- weak reference: Optional Type

- unowned reference: Non Optional Type

Weak Reference / Unowned Reference 선택 기준

- weak reference: 다른 인스턴스의 수명이 짧을 때
- unowned reference: 다른 인스턴스의 수명이 같거나 긴 경우

Weak Reference (약한 참조)

weak reference는 참조하는 인스턴스를 약하게 참조한다.

따라서 weak reference가 참조하는 동안 해당 인스턴스는 메모리로부터 할당 해제가 가능하다.

ARC는 인스턴스가 할당 해제되면 weak reference를 자동으로 nil로 설정하게 되고, weak reference의 경우 runtime에 nil로 변경할 수 있어야 하므로 var(변수)로 선언되어야 한다.

```
class Person {
    let name: String
    init(name: String) {
        self.name = name
    }
    var apartment: Apartment?
    deinit {
        print("\(name) is being deinitialized")
    }
}

class Apartment {
    let unit: String
    init(unit: String) {
        self.unit = unit
    }
    weak var tenant: Person?
    deinit {
        print("Apartment \(unit) is being deinitialized")
    }
}
```

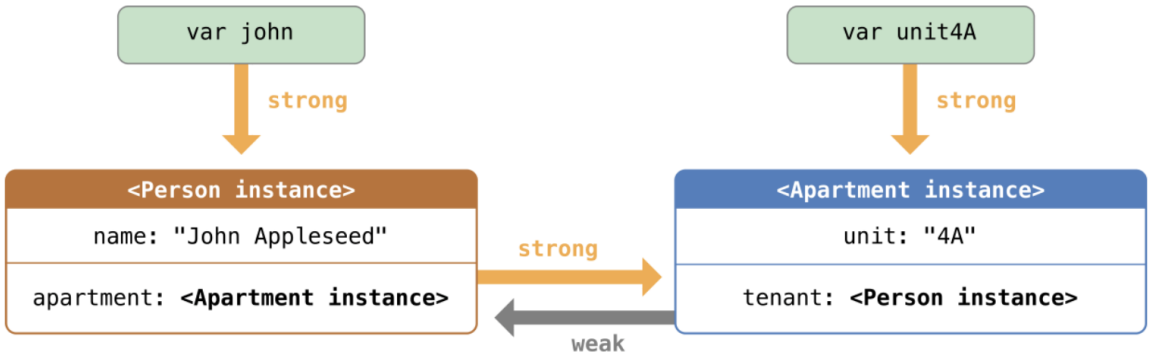
앞서 보았던 예시와 달라진 점은 tenant에 weak 키워드가 붙었다는 점이다.

```
var john: Person?
var unit4A: Apartment

john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")

john!.apartment = unit4A
unit4A!.tenant = john
```

이 코드는 예전과 동일하게 Person 인스턴스 변수인 john과 Apartment 인스턴스 변수인 unit4A를 생성해 값을 할당하고, unit4A는 john의 apartment로, john을 unit4A의 세입자로 설정했다.

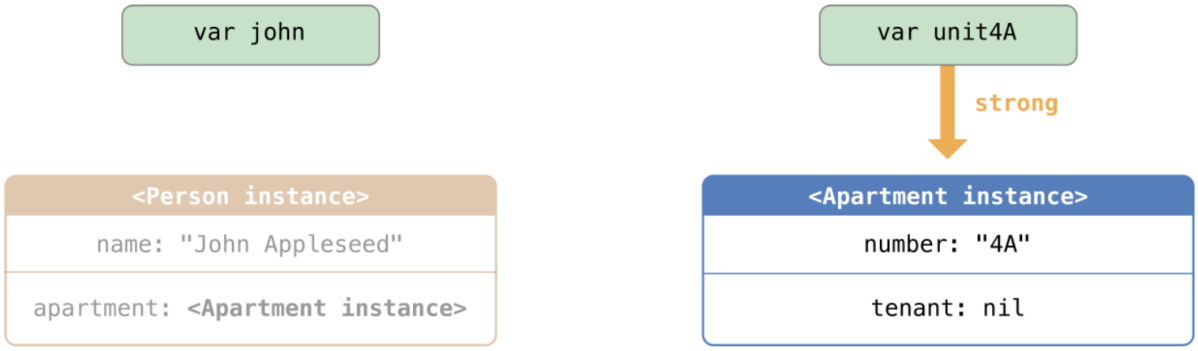


이 때 이전과 다른점은 Person 인스턴스는 여전히 Apartment 인스턴스에게 강한 참조 결합을 하는 반면, Apartment 인스턴스는 Person 인스턴스에게 약한 참조를 하게 된다.

이때 john이 하고 있는 강한 참조 결합을 끊어주면 Person 인스턴스에 대한 강한 참조는 더 이상 존재하지 않는다.

```
john = nil
// "John Appleseed is being deinitialized"
```

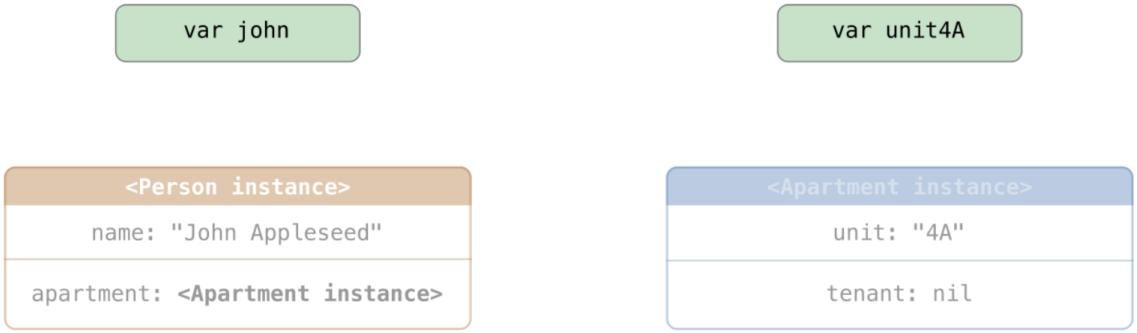
john에 nil을 할당하므로 Person 인스턴스가 메모리에서 할당 해제되고, deinitializer에 의해 구문이 출력된다.



그림처럼 Person 인스턴스는 더이상 강한 참조가 없기 때문에 할당 해제되고, Person 인스턴스를 참조했던 tenant ㅓ 프로퍼티는 ARC에 의해 nil로 설정된다.

```
unit4A = nil
// "Apartment 4A is being deinitialized"
```

unit4A에 nil을 할당해 강한 참조 결합을 끊으면 더 이상 Apartment 인스턴스에 남은 강한 참조는 없기 때문에 메모리에서 할당 해제된다.



weak reference를 사용해 strong reference cycle 문제를 해결할 수 있다.

Unowned Reference (비소유 참조)

unowned reference는 weak reference와 같이 참조하는 인스턴스를 강하게 참조하지 않는다.

그러나 weak reference와는 달리 다른 인스턴스의 수명이 같거나 긴 경우 사용한다.

unowned reference의 경우에는 항상 값이 있다고 생각한다.

따라서 선언을 할 때 optional type이 아니고, ARC는 unowned reference를 nil로 설정하지 않는다. 즉, unowned reference는 절대로 할당 해제가 되지 않을 인스턴스를 참조하는 경우에만 사용해야 하며, 만약 unowned reference로 선언된 할당이 해제된 인스턴스를 참조하려고 한다면 런타임 에러가 발생하게 될 것이다.

```
class Customer {
    let name: String
    var card: CreditCard?
    init(name: String) {
        self.name = name
    }
    deinit {
        print("\(name) is being deinitialized")
    }
}

class CreditCard {
    let number: UInt64
    unowned let customer: Customer
    init(number: UInt64, customer: Customer) {
        self.number = number
        self.customer = customer
    }
    deinit {
        print("Card #\(number) is being deinitialized")
    }
}
```

이번 예시는 Apartment와 Person 예제와는 다른 관계를 가지고 있는 Customer와 CreditCard 예시이다. 고객은 credit card를 가지고 있거나 그렇지 않을 수 있지만, credit care는 항상 고객과 연결이 되어 있어야만 한다. (반드시 소유주가 존재)

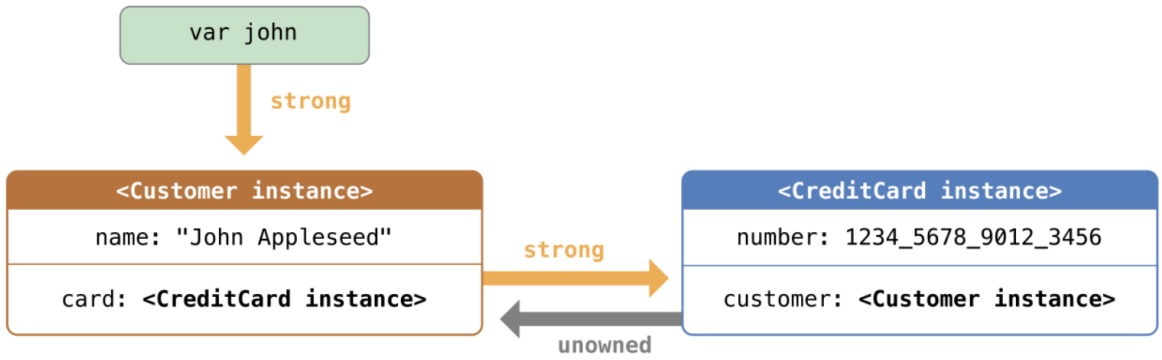
그래서 Customer 클래스의 card는 optional type으로 선언되어 있다.

반면 CreditCard 클래스의 customer 프로퍼티는 optional type이 아닌 unowned reference 타입으로 선언되어있다. 게다가 CreditCard 인스턴스는 number, customer의 정보를 init할 때 넘겨주어야만 생성되고, 이는 CreditCard 인스턴스가 생성될 때 항상

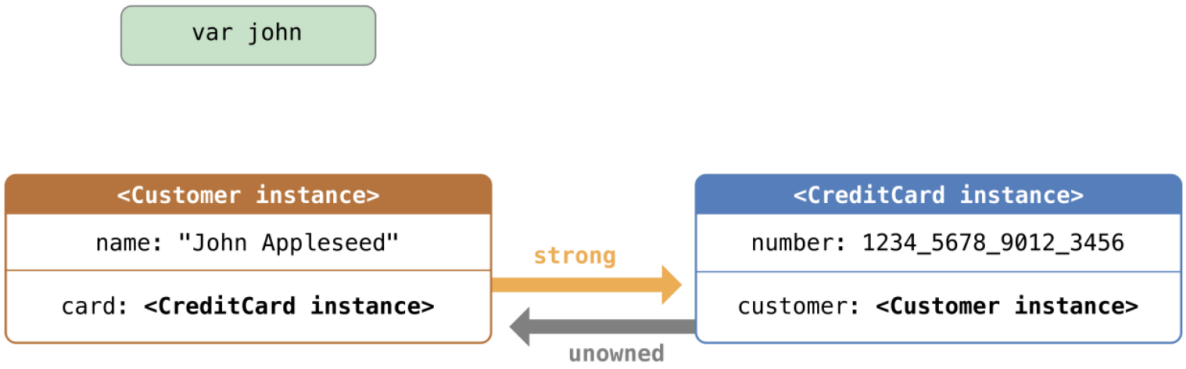
customer 인스턴스를 참조하게 되고, 순환참조를 피하기 위해 unowned reference 참조를 사용한다.

```
var john: Customer?
john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john!)
```

이제 john 변수에 Customer 인스턴스를 만들어 초기화한다. 그리고 john의 card로 CreditCard 인스턴스를 생성하며 초기화시켜 할당한다.



이 관계는 위와 같이 표현된다. Customer 인스턴스는 CreditCard 인스턴스를 강하게 참조하고, CreditCard 인스턴스는 Customer 인스턴스를 unowned reference하게된다.



여기에서 john 변수의 Customer 인스턴스로의 강한 참조를 끊어주게 되면 더 이상 Customer 인스턴스는 강한 참조를 가지고 있지 않게 된다.

```
john = nil
// prints "John Appleseed is being deinitialized"
// prints "Card #1234567890123456 is being deinitialized"
```

Customer 인스턴스는 메모리에서 할당 해제되고, 이후 CreditCard 인스턴스도 강한 참조가 없기 때문에 메모리에서 할당 해제된다. 따라서 deinitializer에 의해 구문이 출력되는 것을 확인할 수 있다.

unowned reference는 다른 인스턴스의 수명과 같거나 수명이 더 긴 경우 사용한다고 했는데, Customer의 card 프로퍼티의 경우 다른 인스턴스(Customer)의 수명과 같았기 때문에 unowned reference로 선언이 되어있다.

Unowned Optional References

기존 unowned reference와 weak reference의 차이 중 하나는 optional이었다.

weak는 nil이 될 수 있고, unowned는 nil이 될 수 없다.

그러나 Swift5 이상부터는 unowned reference의 경우에도 optional type이 될 수 있다.

즉, weak reference와 unowned reference는 이제 같은 맥락에서 사용이 가능해졌다.

차이점은?

weak reference의 경우에는 이전과 동일하지만, unowned optional reference는 이제 항상 값이 있는 유효한 객체를 참조하거나, nil로 설정이 되어있는 경우멘 사용하면 될 것이다.

예시를 통해 확인해보자.

```
class Department {
    var name: String
    var courses: [Course]
    init(name: String) {
        self.name = name
        self.courses = []
    }
}

class Course {
    var name: String
    unowned var department: Department
    unowned var nextCourse: Course?
    init(name: String, in department: Department) {
        self.name = name
        self.department = department
```

```
        self.nextCourse = nil
    }
}
```

위 예시는 학과(Department)와 강좌(Course) 사이의 관계이다. 학과는 각각의 학과에서 제시하는 코스(강좌)와 강한 참조를 유지한다. 강좌의 경우에는 두 개의 unowned reference를 가지는데, 하나는 학과이며, 다른 하나는 다음 학생이 다음으로 수강해야할 nextCourse로서 어떤 객체도 소유하지 않는다.

모든 강좌는 학과와 연결이 되어야 하므로 department 프로퍼티는 non-optional 타입이지만 어떠한 강좌를 들은 다음으로 수강해야할 강좌가 없을수도 있기 때문에 optional type을 가지게 된다.

```
let department = Department(name: "Computer Science Engineering")

let intro = Course(name: "Welcome to C language", in: department)
let intermediate = Course(name: "Welcome to C++ language", in: department)
let advanced = Course(name: "Welcome to Swift", in: department)

intro.nextCourse = intermediate
intermediate.nextCourse = advanced
department.courses = [intro, intermediate, advanced]
```

Department와 Course를 사용한 예시를 살펴보면,

department라는 상수는 Department 인스턴스로, 컴퓨터공학과라는 이름으로 생성되고 초기화 되어 할당된다.

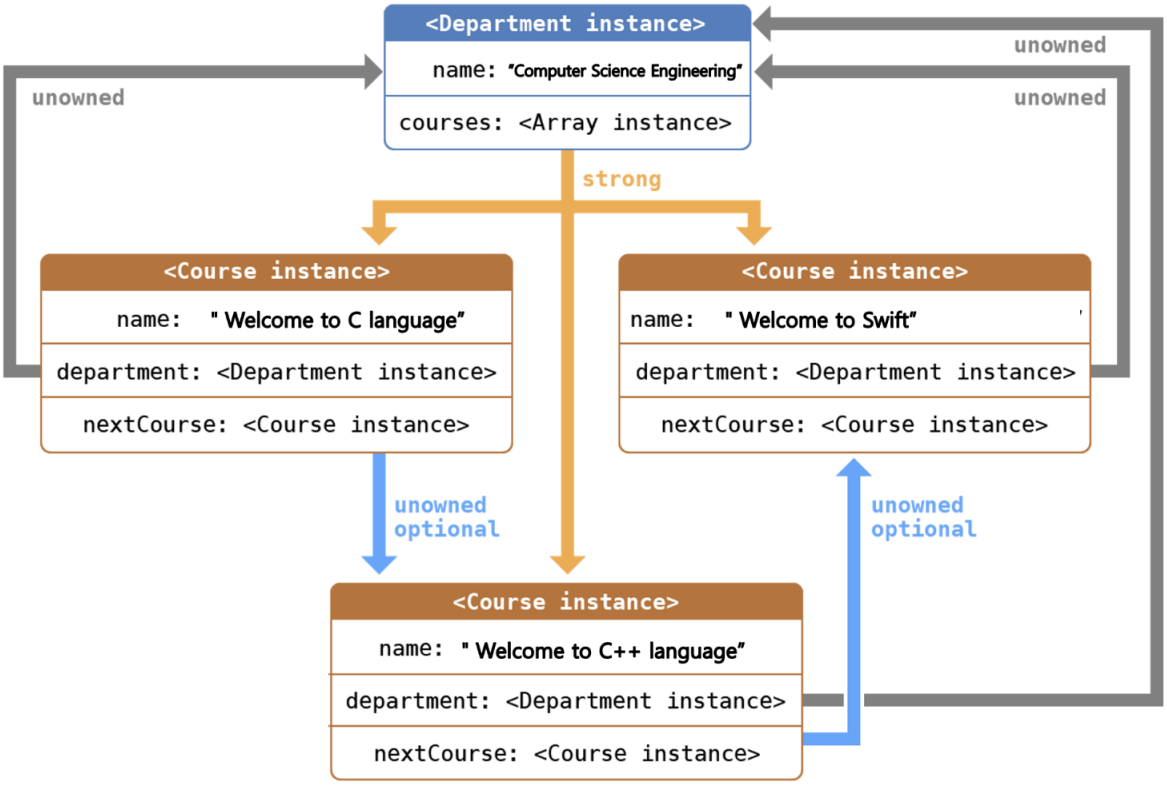
intro라는 상수는 Course 인스턴스로 C언어, 학과는 department(컴공)로 초기화되며 할당된다.

intermediate라는 상수는 Course 인스턴스로 C++, 학과는 department(컴공)로 초기화되며 할당된다.

advanced라는 상수는 Course 인스턴스로 Swift, 학과는 department(컴공)로 초기화되며 할당된다.

이 때 intro(C언어)의 다음 강의로 intermediate(C++), intermediate(C++)의 다음 강의로 advanced(Swift)를 설정한다.

마지막으로 department(컴공과)의 강좌에는 intro(C), intermediate(C++), advanced(Swift)가 있다고 설정한다.



해당 코드들은 위의 그림과 같은 관계를 갖는다. intro와 intermediate의 경우 다음 강의가 있다고 설정했기 때문에 설정해준 course와 unowned optional reference를 가지게 된다.

unowned optional reference는 nil이 될 수 있다는 점을 제외하고는 ARC의 측면에서는 unowned reference와 동일하게 동작한다.

따라서 unowned reference와 마찬가지로 unowned optional reference는 nextCourse가 항상 메모리에서 할당 해제된 것을 참조하지 않도록 관리 해야하며 위의 예시에서는 department(컴공)에서 course(강좌)를 삭제하는 경우, 삭제되는 강좌를 참조하는 관계를 모두 제거해야 한다.

Unowned References and Implicitly Unwrapped Optional Properties

weak reference와 unowned reference를 통해 대부분의 strong refernce cycle을 해결할 수 있다.

1. Person, Apartment 예시에서는 두 가지 프로퍼티 모두 옵셔널 타입으로 nil이 될 수 있는 경우에 strong reference cycle이 발생할 수 있고, 이를 weak reference를 사용해 해결했다.
2. Custom, CreditCard 예시에서는 하나의 프로퍼티는 optional 타입으로 nil이 될 수 있고, 다른 하나는 Non-optional 타입으로 nil이 될 수 없는 경우에 strong reference cycle을 unowned reference를 통해 해결했다.
3. 두 개의 프로퍼티가 항상 값을 가지고, 한 번 initialize된 이후에는 nil이 되지 않는 경우가 있다. 이 시나리오에서는 하나의 클래스에서 unowned 프로퍼티를 가지고, unowned 프로퍼티를 가지지 않는 다른 클래스에서 암시적으로 벗겨진(Implicitly unwrapped) optional 프로퍼티를 통해 해결할 수 있다.

마지막 3번의 경우 예시를 통해 알아보자.

```
class Country {
    let name: String
    var capitalCity: City!
    init(name: String, capitalName: String) {
        self.name = name
        self.capitalCity = City(name: capitalName, country: self)
    }
}

class City {
    let name: String
    unowned let country: Country
    init(name: String, country: Country) {
        self.name = name
        self.country = country
    }
}
```

두 클래스의 상호 의존성을 살펴보면 City의 init은 Country 인스턴스를 country 프로퍼티에 저장한다.

City의 init은 Country의 init 내부에서 호출이 되는데, 이 때 Country의 init은 새로운 Country 인스턴스가 완전히 초기화되기 전에 City init으로 self를 매개변수로 넘겨주지 못하게 된다.

따라서 이런 요구사항을 만족시키기 위해서는 Country 클래스 안의 capitalCity 프로퍼티를 implicitly unwrapped optional 타입으로 선언해야 한다. 이는 capitalCity 프로퍼티는 default 값으로 nil을 가지게 되지만 다른 옵셔널들과는 달리 값에 접근할 때 unwrapping이 필요 없다.

여기서 capitalCity 프로퍼티가 기본값으로 nil을 가지므로 새로운 Country 인스턴스는 name 프로퍼티가 설정되자마자 완전히 초기화될 수 있고, 이것은 name 프로퍼티가 설정되자마자 Country init은 reference를 시작하고 암시적 self 프로퍼티를 전달할 수 있게 된다. 따라서 Country init은 self를 City init의 매개변수로 보낼 수 있다.

즉, strong reference cycle 없이 Country와 City 인스턴스를 하나의 문장으로 생성할 수 있고 capitalCity 프로퍼티는 옵셔널 값을 unwrapping하지 않고 바로 접근해 사용이 가능하다.

```
var country = Country(name: "Canada", capitalName: "Ottawa")
print("\(country.name)'s capital city is called \(country.capitalCity.name)")
// prints "Canada's capital city is called Ottawa"
```

위의 코드를 보면 country 변수에 Country를 생성하며 초기화해주는데 이 과정 안에는 City도 생성되면서 초기화가 진행된다. 그리고 country의 capitalCity에 unwrapping 없이 접근하는 것을 확인할 수 있다.

위의 예시에서 implicitly unwrapping optional을 사용하는 것은 two-phase class initializer의 요구사항을 만족시켜주기 위해 사용했으며, capitalCity 프로퍼티는 한 번 initialize가 끝나면 strong reference cycle을 만들지 않고 Non-optional 값처럼 접근하고 사용할 수 있게 된다.

RESULT

class의 객체끼리에서 strong reference cycle을 해결할 수 있는 방법은 네 가지가 있다.

1. weak reference
2. unowned reference
3. unowned optional reference
4. unimplicitly unwrapped optional

여기에서 weak, unowned의 차이에 대해 설명하자면 Swift5 이전에는 optional/non-optional로 구분되었으나 unowned optional reference가 생기면서 unowned도 nil 값을 가질 수 있게 되었다.

그렇다면 weak, unowned의 차이점은

1. unowned는 let 키워드로 선언할 수 있다.
   weak의 경우 runtime동안 ARC에 의해 nil 값으로 변경되기 때문에 변수로만 선언이 가능했다면, unowned는 optional이 가능한 타입과 그렇지 않은 타입으로 선언이 가능하다. optional이 불가능한 타입의 경우에는 let으로 선언이 가능하며, optional이 가능한 타입은 weak 타입과 마찬가지로 변수로만 선언이 가능하다.
2. unowned와 weak는 사용되는 시점이 다르다.
   unowned의 경우에는 다른 인스턴스의 생명주기가 같거나 더 긴 경우에 사용된다.
   weak의 경우에는 다른 인스턴스의 생명주기가 짧은 경우에 사용된다. 그렇다면 unowned optional이 나오기 전까지는 인스턴스의 생명주기가 짧은 경우 weak를 사용했는데, 이제는 unowned optional을 사용하는 것이 유리하다. 그 이유는 weak의 경우 ARC가 계속해서 인스턴스를 추적하다가 객체가 사라질 때 nil로 값을 변경하게 되는데 이 과정은 오버헤드라고 할 수 있기 때문에 unowned optional을 사용하면 이러한 오버헤드를 줄일 수 있다.