클래스와 구조체

Classes & Structures

"*Classes* and *structures* are general-purpose, flexible constructs that become the building blocks of your program's code.
You define properties and methods to add functionality to your classes and structures by using exactly the same syntax as for constants, variables, and functions."

클래스와 구조체는 당신의 프로그램코드에 범용적이고 유연한 구조로 만들어진 블록입니다.
상수, 변수 및 함수와 동일한 문법을 사용하여 클래스 및 구조체에 프로퍼티와 메서드를 정의합니다

Classes & Structures

- 프로그램 코드 블록의 기본 구조이다.
- 프로퍼티와 메서드를 추가 할수 있다. (두 구조의 문법 같음)
- 단일 파일에 정의 되며(필수는 아니다) 다른 코드에서 자동으로 사용 할수 있습니다.(접근 제한자에 따라 접근성은 차이가 있다. internal 기본 접근 제한자)
- 인스턴스 생성 및 초기 상태를 설정하기 위해 initializer(초기화 메소드)를 사용하고, 사용자의 니즈에 맞춰 Custom한 initializer를 만들어 사용할 수 있다.
- 둘 구조 모두 메모리에 적재되면 instance라고 불린다.
- 기본 블록에 추가하여 확장된 블록을 추가 할 수 있다. (Extensions)
- 프로토콜을 채택하여 사용할 수 있다.

기본 구조

```
class SomeClass {
    // class definition goes here
}
```

```
struct SomeStructure {
    // structure definition goes here
}
```

기본 구조

```
class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}
```

class 는 struct instance를 가질 수 있으나

struct class instance를 가질 수 없다.

```
struct Resolution {
    var width = 0
    var height = 0
}
```

인스턴스

```
let someVideoMode: VideoMode = VideoMode()
```

```
let someResolution: Resolution = Resolution()
```

Properties 접근

```
let someVideoMode: VideoMode = VideoMode()
print("VideoMode is \(someVideoMode.resolution.width)”)
```

```
let someResolution: Resolution = Resolution()
print("Resolution is \(someResolution.width)”)
```

- 닷(.) 문법을 통해 접근

Initialization(초기화)

초기화 : 갖고 있는 모든 property에 초기값을 지정해야 함

Initialization is the process of preparing an instance of a class, structure, or enumeration for use.

“초기화는 클래스, 구조체, 열거형의 인스턴스를 만들기 위한 준비 과정으로 사용됩니다. “

초기화의 결과물 : instance
직전에 해야할 일 : initialization

초기화

- 인스턴스에 설정된 속성의 초기값을 설정과 초기화하는데 목적이 있다.
- 클래스 및 구조체는 인스턴스로 만들어 질때 모든 프로퍼티는 적절한 초기값으로 모두 초기화 해야 한다.
- 구조체는 자동으로 Memberwise Initializers가 만들어 진다.

base Initializers

```
struct Subject {
    var name:String
}

class Student {

    var subjects:[Subject] = []

    func addSubject(name:String) {

        let subject = Subject(name: name)
        subjects.append(subject)
    }
}

var wingMan:Person = Person()
 = var wingMan:Person = Person.init()
```

Memberwise Initializers

Initializers

Memberwise Initializers

```
struct Subject {
    var name:String

    init(name: String) {
        //memberwise Initializer
        self.name = name
    }

}

let subject = Subject(name: name)
    = let subject = Subject.init(name: name)
```

개발자가 초기화 메서드를 정의하지 않으면 구조체는 자동으로 모든 프로퍼티를 대응하는 초기화 메서드를 제공한다.

Memberwise Initializers

Custom Initializers

```
class Student {
    var subjects:[Subject] = []

    func addSubject(name:String) {
        var sub1:Subject = Subject(gender:true)
        sub1.name = "joo"
        sub1.age = 30
        subjects.append(sub1)
    }
}

struct Subject {
    var name:String?
    var age:Int?
    var gender:Bool

    init(gender:Bool) {
        self.gender = gender
    }
}
```

상속은 class에만 있는 특성

상속과 Initializers

- 부모 클래스로부터 상속받은 모든 저장 속성은 초기화할 때 초기 값을 할당받아야 함.
- Swift는 클래스 타입에 모든 저장 속성에 초기 값을 받도록 도와주는 두가지 이니셜라이저를 정의함. 이를 지정 이니셜라이저(designated initializers)와 편의 이니셜라이저(convenience initializers)라고 함.

Designated initializers

```
init(parameters) {

   statements

}
```

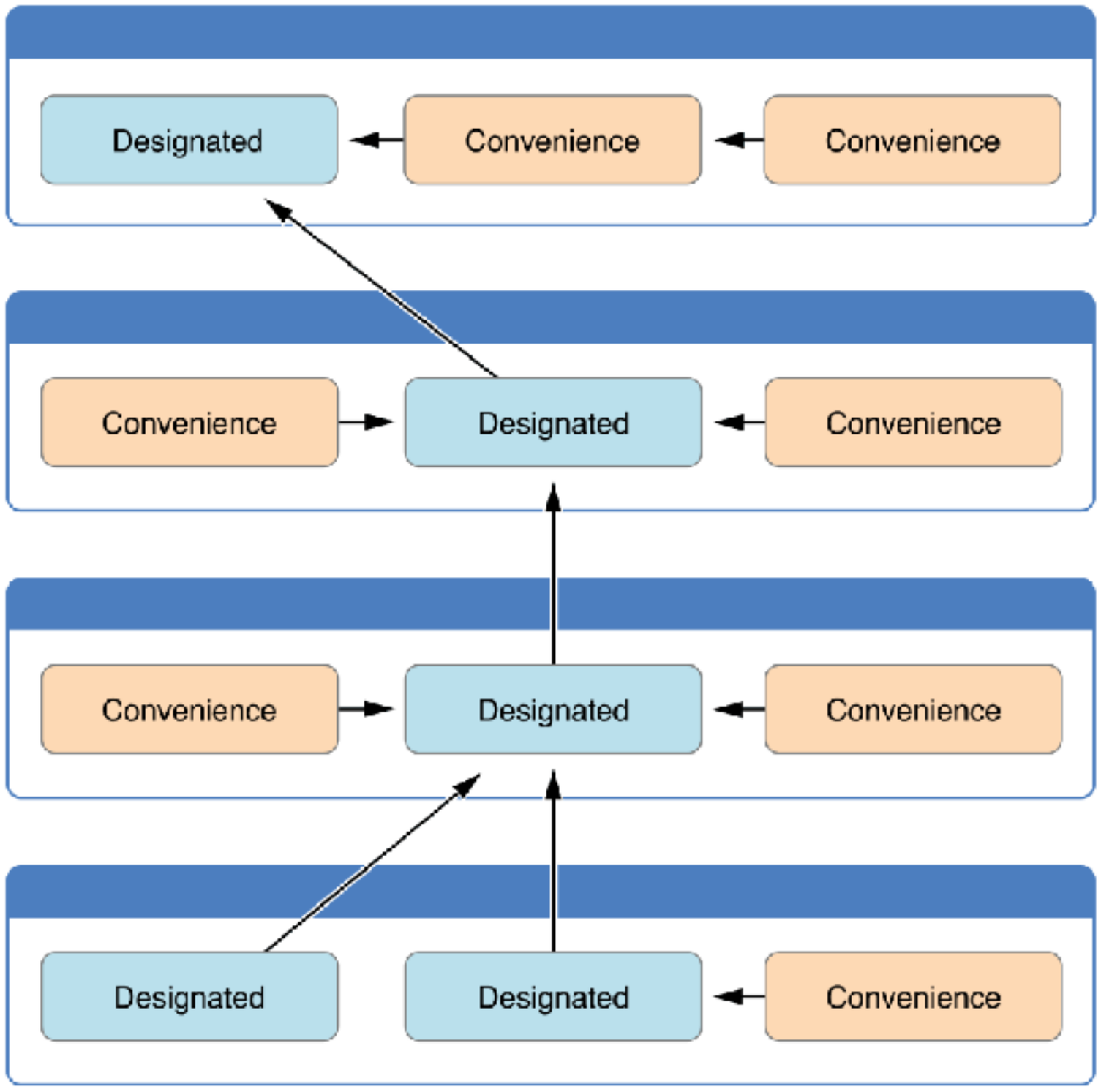
- 모든 프로퍼티가 초기화 되어야 한다.
- 상속을 받았다면 부모 클래스의 Designated initializers를 호출 해야 한다.

Convenience initializers

```
convenience init(parameters) {
    statements
}
```

- 다른 convenience initializer을 호출할수 있다.
- 하지만 궁극적으론 designated initializer을 호출해야만 한다.

Designated & Convenience 모식도



Required Initializers

상속을 받는 subclass가 꼭 구현을 해야 한다.

```
class SomeClass {
    required init() {
        // initializer implementation goes here
    }
}
```

- 해당 initializer는 필수적으로 구현해야만 한다.
- 상속받은 모든 클래스는 필수로 구현해야 한다.
- required initializer를 구현할땐 override 수식어를 사용할 필요 없다.

Setting a Default Property Value with a Closure or Function

```
class SomeClass {

    let someProperty: SomeType = {
        // 해당 클로져 안에 프로퍼티의 기본값을 지정한다.
        // someValue는 반드시 SomeType과 타입이 같아야 한다.
        return someValue
    }()

}
```

- 클래스의 init시 해당 프로퍼티의 값이 할당되며, 값대신 클로져나 전역 함수를 사용할수 있다.
- 클로져 사용시 마지막에 **()**를 붙여 클로져를 바로 실행 시킨다.

Classes VS Structures

- Class는 참조 타입이며, Structure는 값 타입이다.
- Class는 상속을 통해 부모클래스의 특성을 상속받을수 있다.
- Class는 Type Casting을 사용할수 있다.(Structure 불가)
- Structure의 프로퍼티는 instance가 var를 통해서 만들어야 수정 가능하다.
- Class는 Reference Counting을 통해 인스턴스의 해제를 계산합니다.
- Calss는 deinitializer를 사용할수 있습니다.

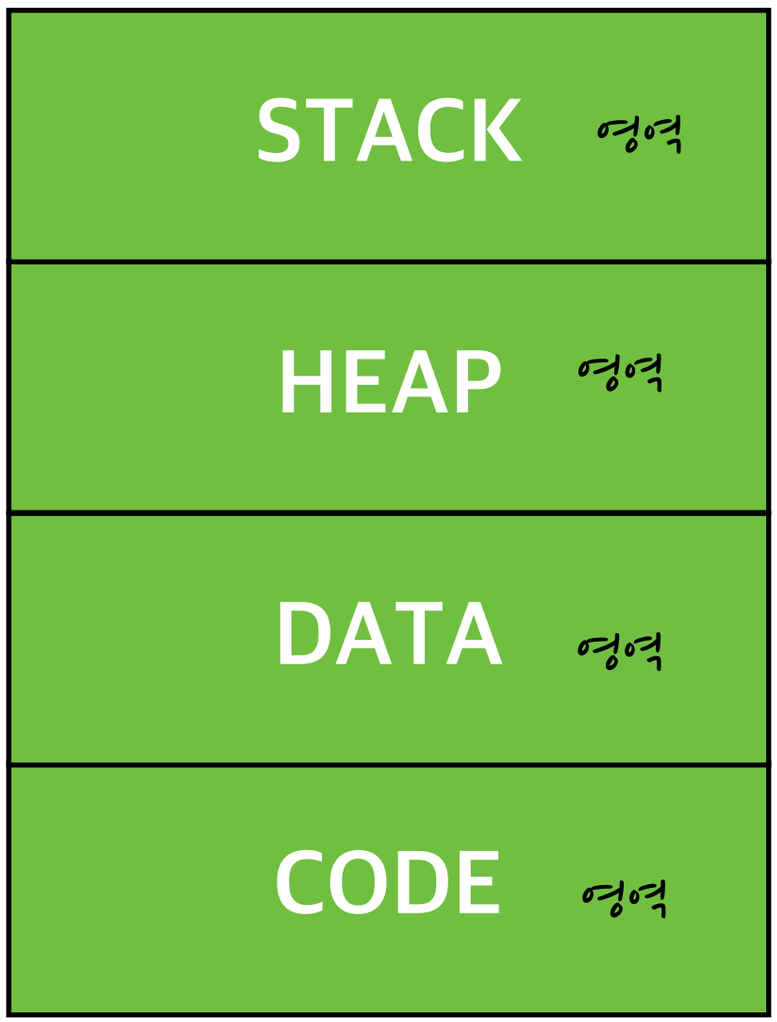
Classes VS Structures

- Class는 참조 타입이며, Structure는 값 타입이다.
- Class는 상속을 통해 부모클래스의 특성을 상속받을수 있다.
- Class는 Type Casting을 사용할수 있다.(Structure 불가)
- Structure의 프로퍼티는 instance가 var를 통해서 만들어야 수정 가능하다.
- Class는 Reference Counting을 통해 인스턴스의 해제를 계산합니다.
- Calss는 deinitializer를 사용할수 있습니다.

값타입 vs 참조타입

Memory구조

논리적 메모리 구조를 형상화한 그림



← 지역 변수, 매개변수 등

← 동적 할당을 위한 영역

← 전역 변수, 정적변수가 저장

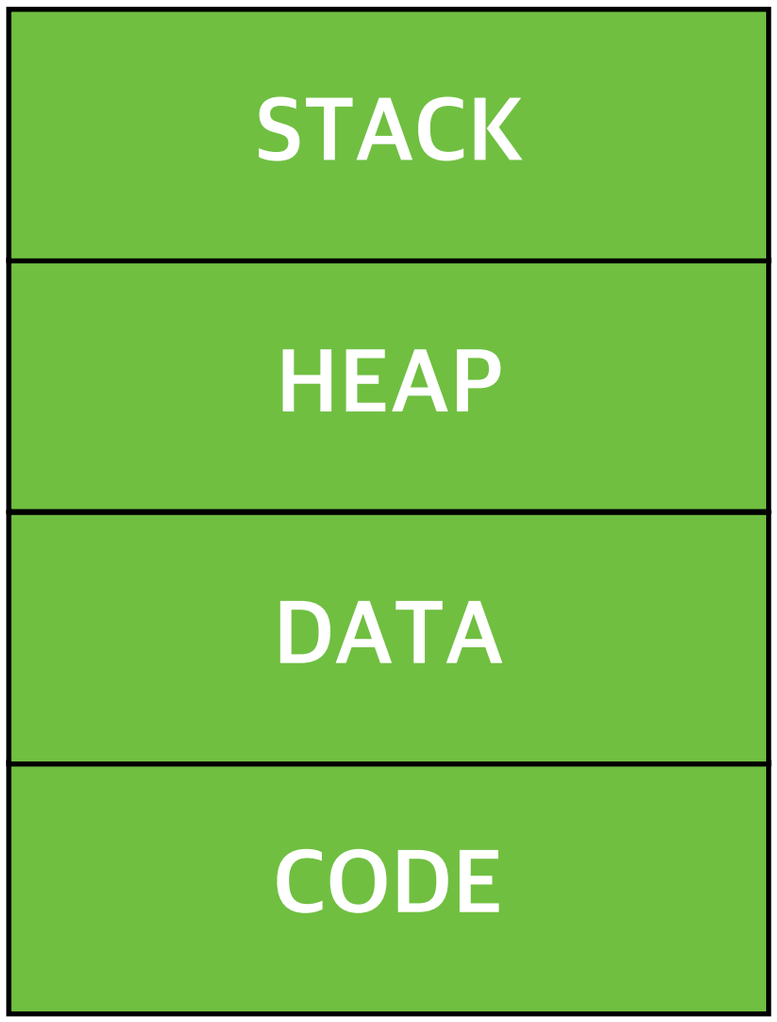
← 프로그램 code 저장

Memory구조 파악하기

다음 코드가 메모리에 어떻게 들어 갈까요?

```
var num:Int = 4
var num2:Int = 5
```

Memory구조 파악하기



num = 4, num2 = 5

프로그램 code 저장

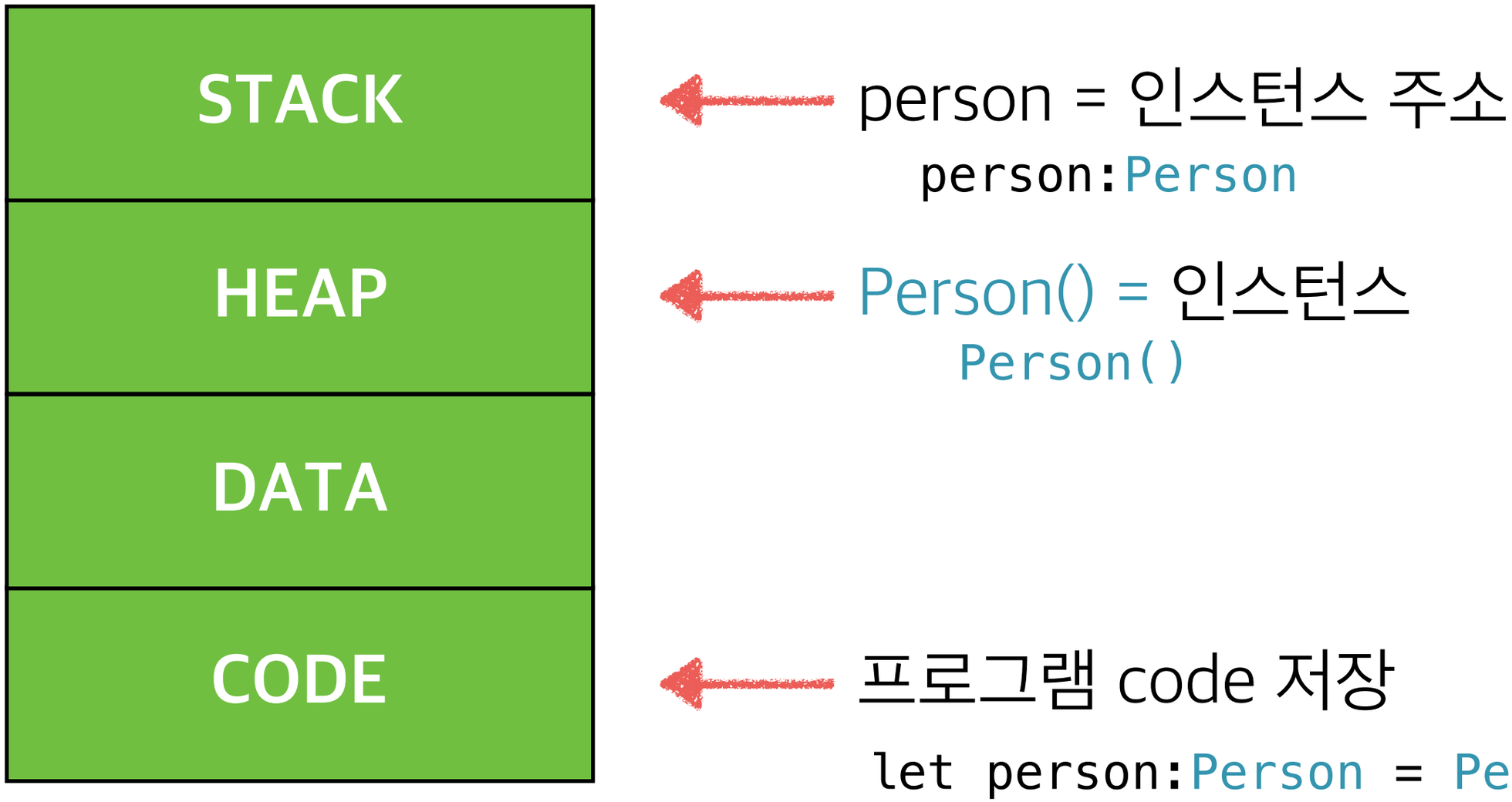
```
var num:Int = 4
var num2:Int = 5
```

Memory구조 파악하기

다음 코드가 메모리에 어떻게 들어 갈까요?

```
let person:Person = Person()
```

Memory구조 파악하기



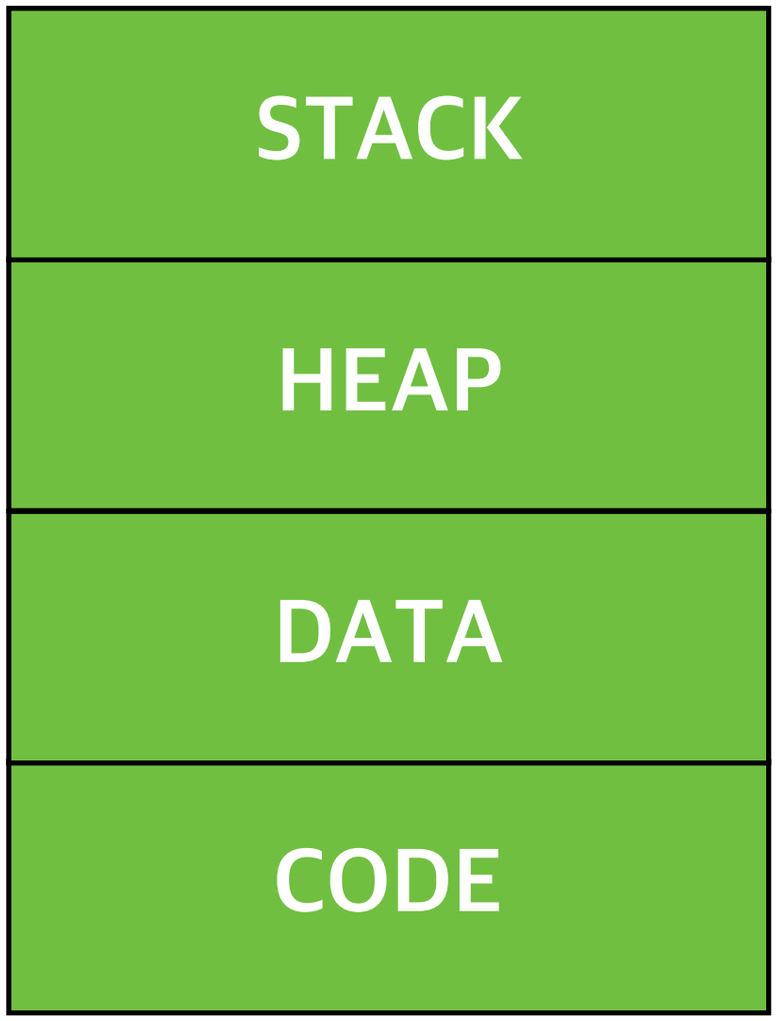
Memory구조 파악하기

다음 코드가 메모리에 어떻게 들어 갈까요?

```
static let number:Int = 5
```

정적

Memory구조 파악하기



number = 5

프로그램 code 저장

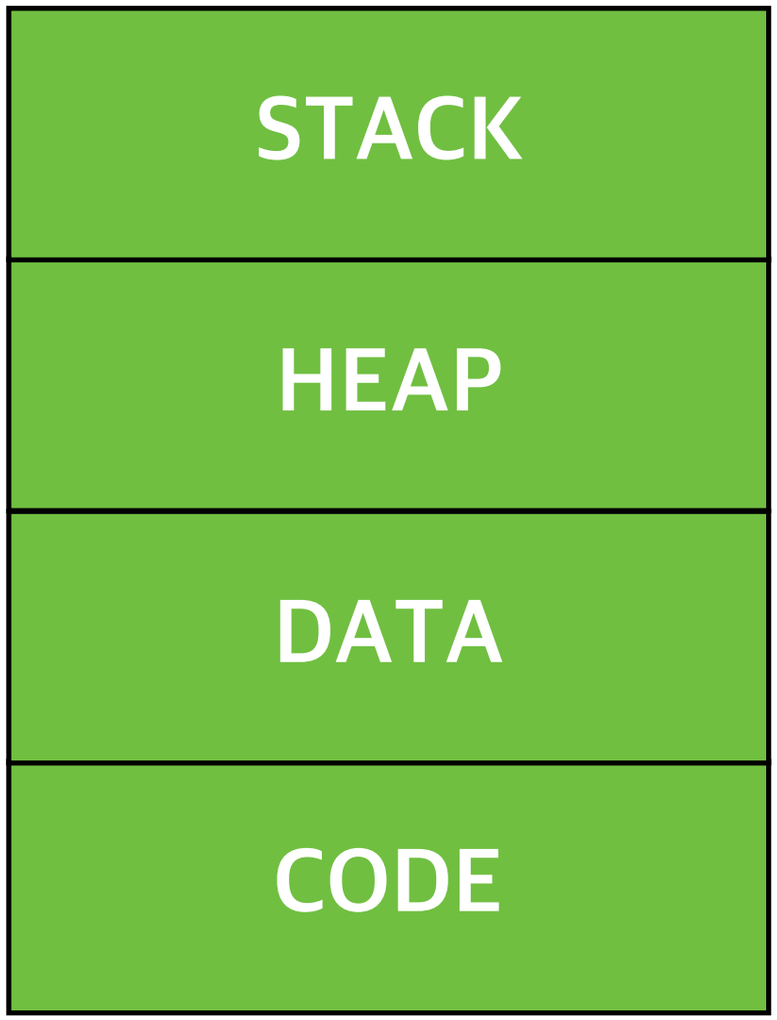
```
static let number:Int = 5
```

Memory구조 파악하기

다음 코드가 메모리에 어떻게 들어 갈까요?

```
func sumTwoNumber(num1:Int, num2:Int) -> Int
{
    return num1 + num2
}
```

Memory구조 파악하기



num1 = 3, num2 = 4

(num1:3, num2:4)

프로그램 code 저장

```
func sumTwoNumber(num1:Int, num2:Int)
-> Int {
      return num1 + num2
}
```

Fast campus

두 코드의 차이점

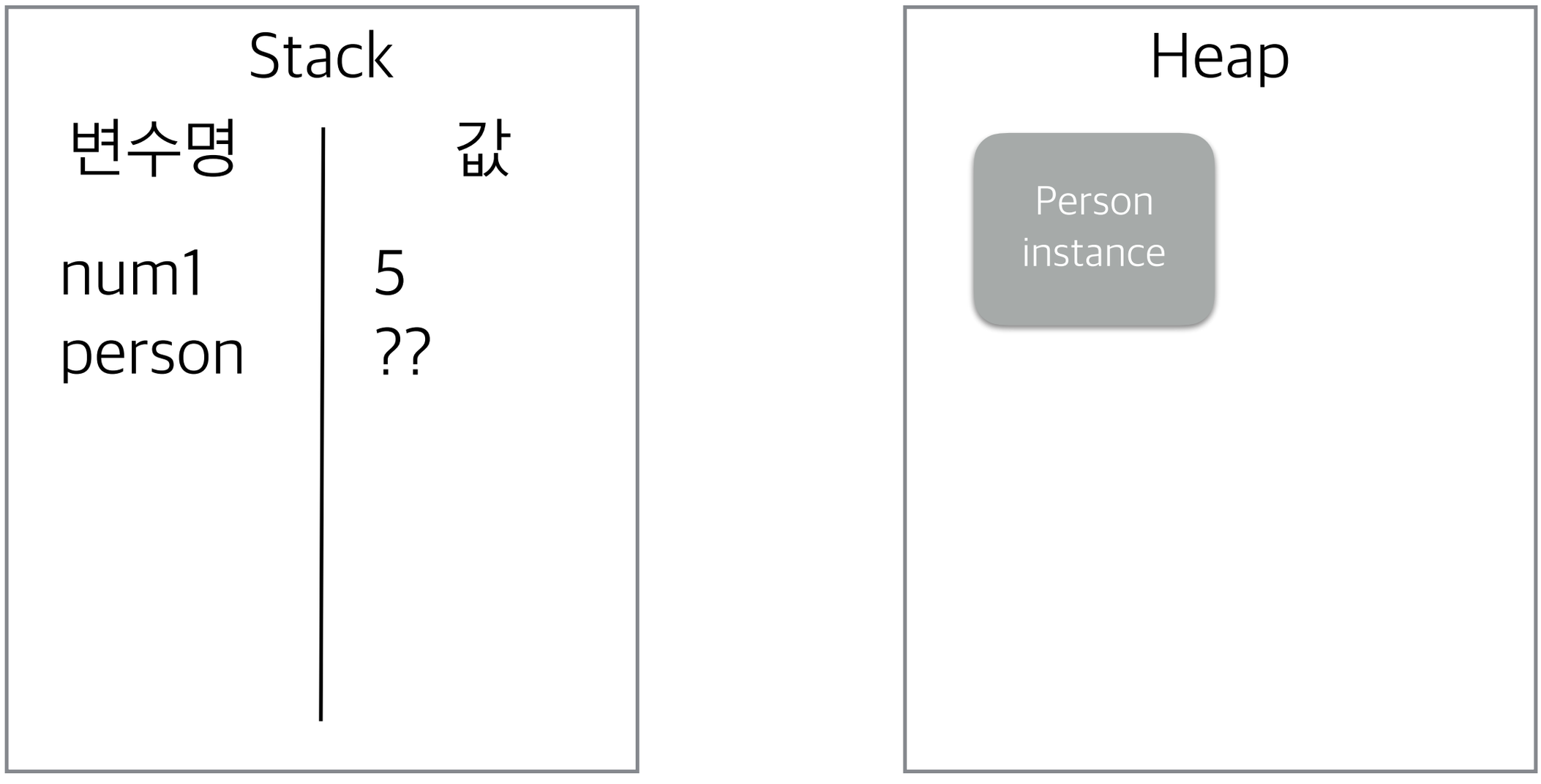
```
let num2:Int = 5
```

Struct VS Class

```
let person:Person = Person()
```

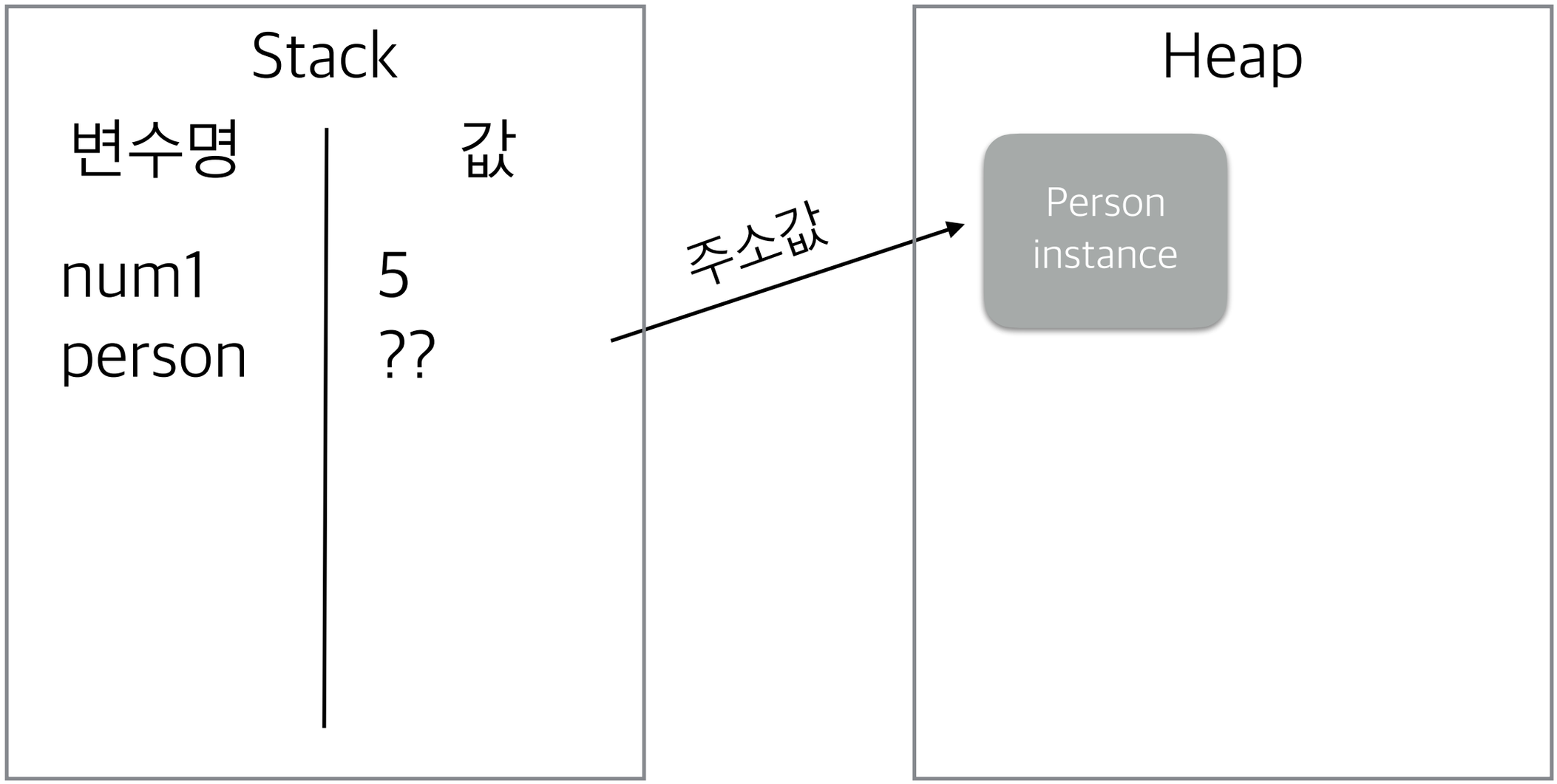
두 코드의 차이점

```
let num1:Int = 5
let person:Person = Person()
```



두 코드의 차이점

```
let num1:Int = 5
let person:Person = Person()
```



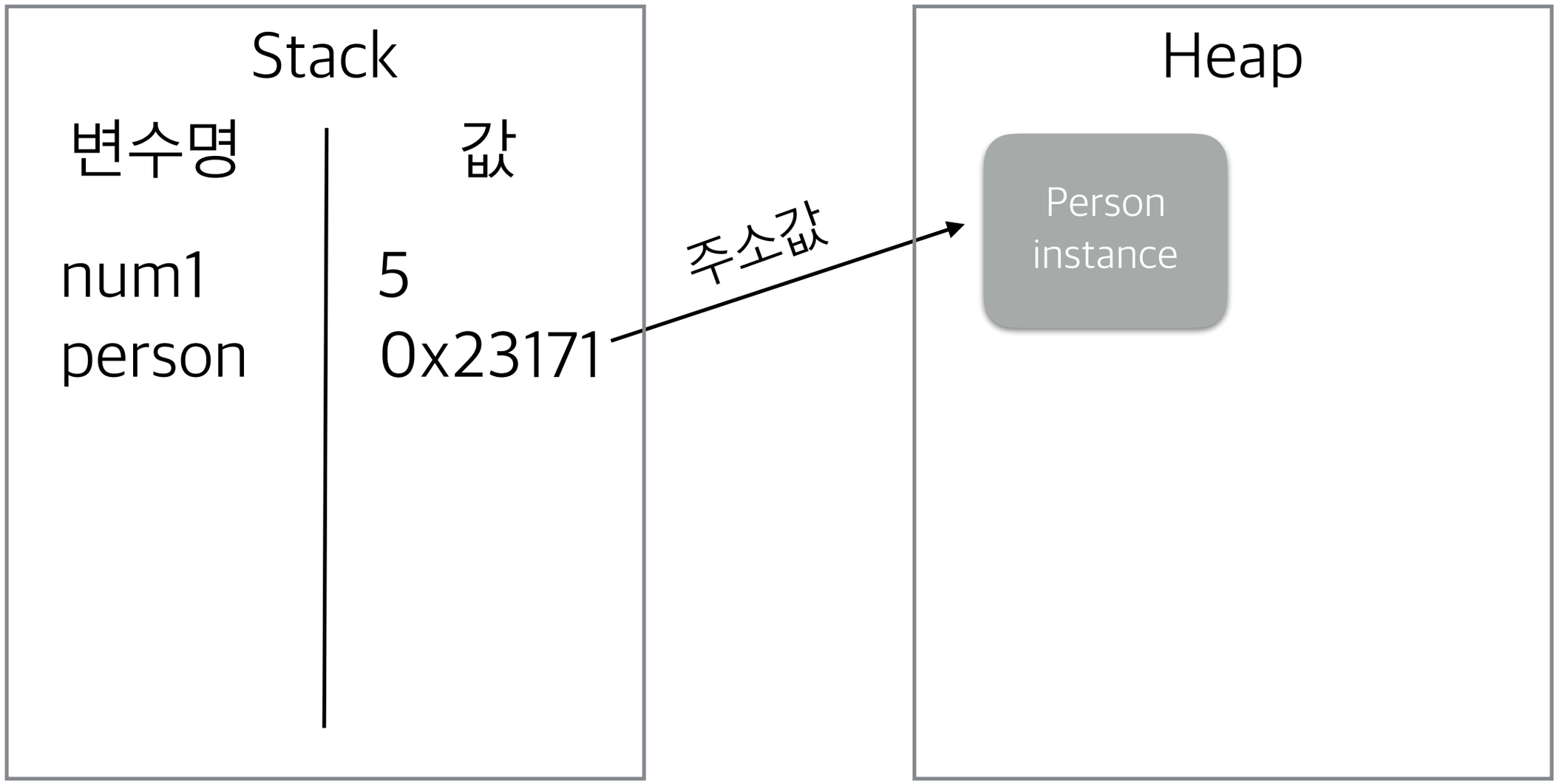
Pointer

- 포인터(pointer)는 프로그래밍 언어에서 다른 변수, 혹은 그 변수의 메모리 공간주소를 가리키는 변수를 말한다.

-Wikipedia-

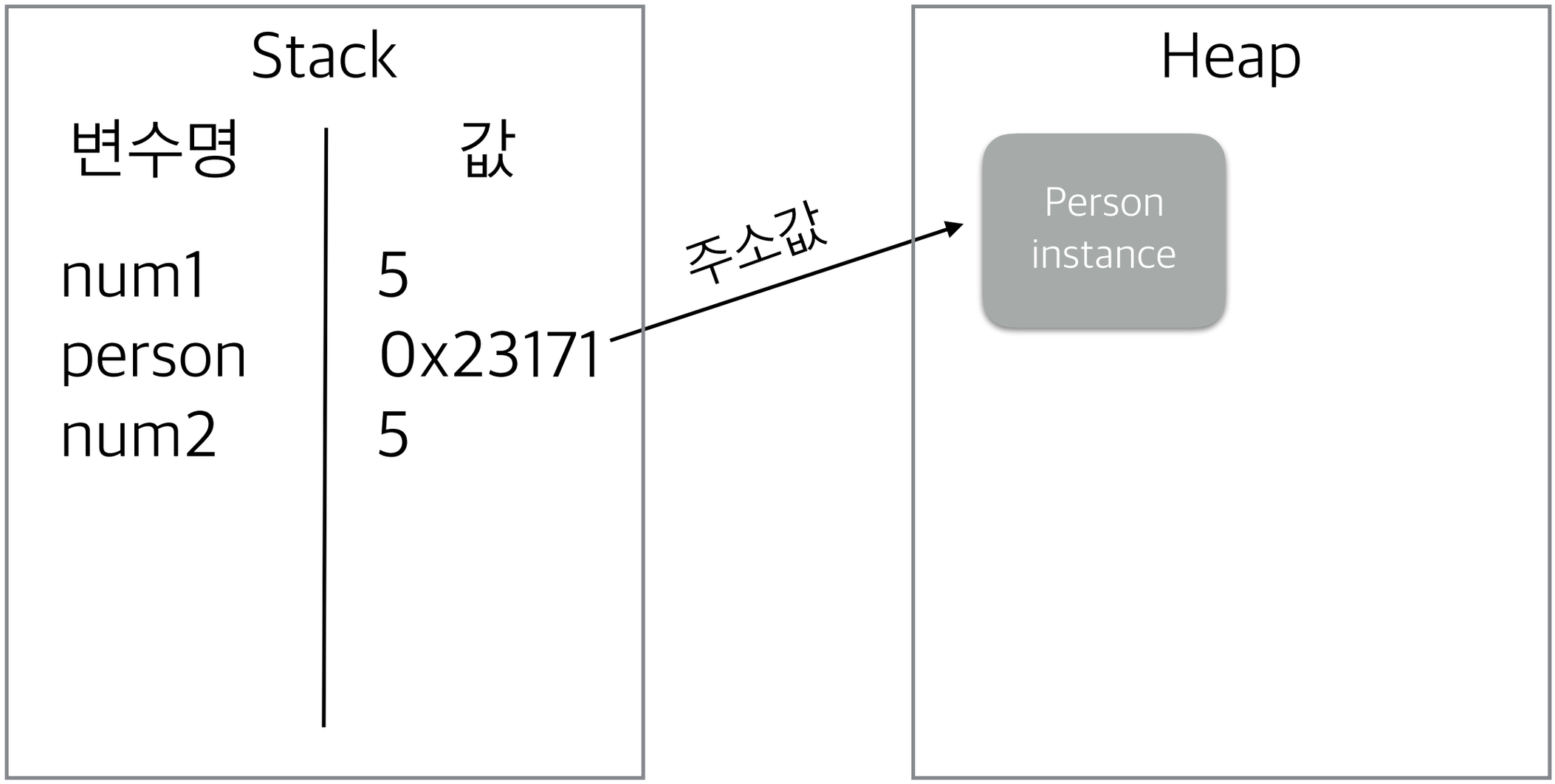
Pointer

```
let num1:Int = 5
let person:Person = Person()
```



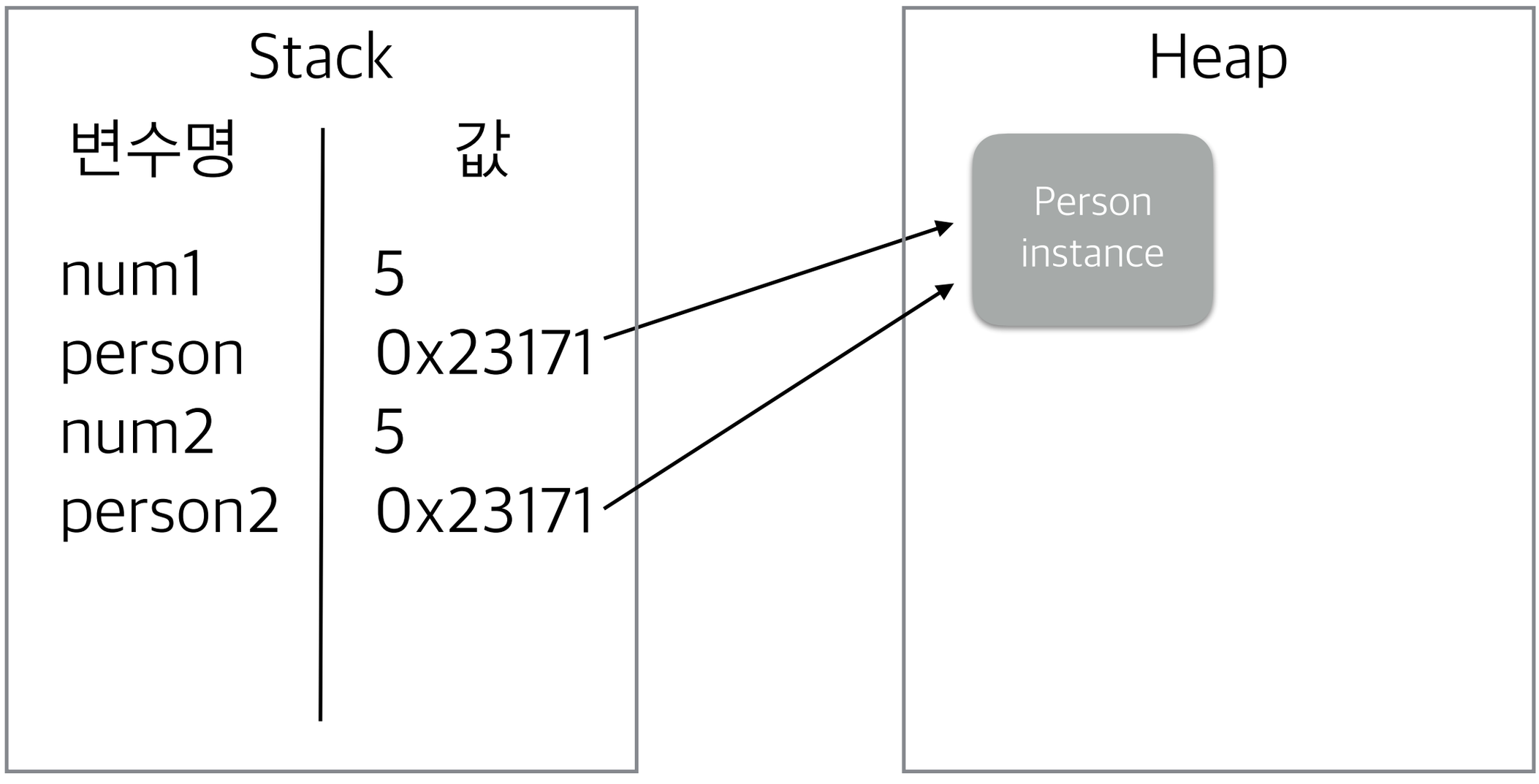
Pointer

```
let num1:Int = 5
let person:Person = Person()
let num2 = num1
```



Pointer

```
let person:Person = Person()
let num2 = num1
let person2:Person = person
```



결과값은?

```
var name: String = "joo"
var reName: String  = name

reName = "wing"
//name의 값은??
```

Struct VS Class

```
let person: Person = Person()
let person2: Person = person
person.name = “joo”

person2.name = “wing”
//person 인스턴스의 name 프로퍼티의 값은??
```

Classes VS Structures

- Class는 참조 타입이며, Structure는 값 타입이다.
- Class는 상속을 통해 부모클래스의 특성을 상속받을수 있다.
- Class는 Type Casting을 사용할수 있다.(Structure 불가)
- Structure의 프로퍼티는 instance가 var를 통해서 만들어야 수정 가능하다.
- Class는 Reference Counting을 통해 인스턴스의 해제를 계산합니다.
- Calss는 deinitializer를 사용할수 있습니다.

Struct - Value Type 프로퍼티 수정

- 기본적으로 구조체와 열거형의 값타입 프로퍼티는 인스턴스 메소드 내에서 수정이 불가능 하다.

- 그러나 특정 메소드에서 수정을 해야 할경우에는 mutating 키워드를 통해 변경 가능하다.

```
struct Point {
    var x = 0.0, y = 0.0
    mutating func moveBy(x deltaX: Double, y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}
```

Class - Deinit

```
class Student {

    init() {
        //인스턴스 생성시 필요한 내용 구현
    }

    deinit {
        //종료직전 필요한 내용 구현
    }

}
```

example: Notification에 사용
- 중앙에 모든 인스턴스의 행동을 관리하는 아이가 있을 때, ARC로 사라질 때, notification, single tone에서 init이나 deinit을 통해 알릴 수 있다.

Class - 상속

class에서만 가능함
기존에 class를 확장하는 개념

- Subclassing
- 기존에 구현되어있는 클래스를 확장, 변형
- 부모 클래스(super class, parent class)와 자식 클래스(sub class, child class)로 관계를 표현
- 상속 할 수록 더 확장되는 구조
  - 즉, 자식이 기능이 더 많다.

super class: 추상적, 포괄적
sub class : 구체적

예제: 동물
움직이는 생물체
포유류
- 새끼를 낳음
- 움직임

UniversityStudent

Student

Person

name
age
eat()

grade
study()

major
goMT()

클래스의 상속

- Swift에선 단 하나의 클래스만 상속 받을수 있다.
  단일상속
- Struct, enum은 상속 받을수 없다. class만 상속 받을 수 있음

- 클래스 이름뒤에 (: 부모클래스) 를 추가 한다.

- 프로토콜과 문법이 같다.

protocol은 여러 개를 받을 수 있음
protocol은 rule이다. 법이기 때문에 반드시 꼭 만들어야 한다. 지켜야 한다.

자식class : 부모class, 프로토콜1, 프로토콜2, ….

재정의

재정의

- 영어로 Override

재정의

- 영어로 override(오버라이드)
- 부모 클래스에게서 물려받은 성질을 그대로 사용하지 않고
  자식 클래스에게 맞는 형태 또는 행위로 변경하여
  사용할 수 있는 기능

재정의

- Person 클래스의 eat 메서드는 집밥을 먹도록 하고,
  Person 클래스를 상속받은 Student 클래스의 eat 메서드는 급식을 먹게 하고,
  Student 클래스를 상속받은 UniversityStudent 클래스의 eat 메서드는 학식을 먹게 만들어 봅시다

다형성

- **재정의(Override)**와 **중복정의(Overload)**는 OOP의 **다형성**의 또다른 모습

- Objective-C는 **중복정의(Overload)**를 **지원하지 않습**니다

- Swift에서는 **중복정의(Overload)**를 **지원합니다.**

Classes VS Structures

- 어떤걸 선택해서 써야할까요?
- 기본 SDK에 클래스와 구조체의 예제를 찾아봅시다.

애플 가이드라인

type에 가까울 때 : struct
object에 가까울 때 : class

- 연관된 간단한 값의 집합을 캡슐화하는 것만이 목적일 때 struct
  데이터를 표현할 때
- 캡슐화된 값이 참조되는 것보다 복사되는 것이 합당할 때

- 구조체에 저장된 프로퍼티가 값타입이며 참조되는 것보다는 복사되는 것이 합당할때

- 다른 타입으로부터 상속받거나 자신이 상속될 필요가 없을 때