Swift 4

Ciclo de vida de las instancias

Inicialización

Inicialización

- Es el proceso que prepara una instancia para su uso
- Hasta que no se completa, no queda fijado el estado de la instancia
- Se realiza en métodos especiales llamados inicializadores o mediante la asignación de un valor predeterminado
- Los inicializadores no devuelven valor
- Las clases y estructuras deben definir un valor para todas sus propiedades almacenadas en la inicialización

Inicializadores

```
init() {
    // perform some initialization here
}
```

Inicializadores

```
struct Fahrenheit {
    var temperature: Double
    init() {
        temperature = 32.0
    }
}

var f = Fahrenheit()

print("The default temperature is \(f.temperature)° Fahrenheit")
// Prints "The default temperature is 32.0° Fahrenheit"
```

Asignación de valores predeterminados

```
struct Fahrenheit {
    var temperature = 32.0
}
```

Inicializador con parámetros

```
struct Celsius {
    var temperatureInCelsius: Double
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
}

let boilingPointOfWater = Celsius(fromFahrenheit: 212.0)
// boilingPointOfWater.temperatureInCelsius is 100.0

let freezingPointOfWater = Celsius(fromKelvin: 273.15)
// freezingPointOfWater.temperatureInCelsius is 0.0
```

Etiquetas de argumentos

- Como todos los inicializadores se llaman `init`, Swift crea etiquetas de argumentos para cada parámetro que pongamos en el inicializador para distinguirlos
- Podemos evitarlo poniendo _ como nombre de argumento
- Al llamar al inicializador siempre hay que poner los nombres de los argumentos que estén definidos

Etiquetas de argumentos

```
struct Color {
    let red, green, blue: Double
    init(red: Double, green: Double, blue: Double) {
        self.red   = red
        self.green = green
        self.blue  = blue
    }
    init(white: Double) {
        red   = white
        green = white
        blue  = white
    }
}
```

Etiquetas de argumentos

```
let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)
let halfGray = Color(white: 0.5)

let veryGreen = Color(0.0, 1.0, 0.0)
// this reports a compile-time error - argument labels are required
```

Parámetros sin etiquetas de argumentos

```
struct Celsius {
    var temperatureInCelsius: Double
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
    init(_ celsius: Double) {
        temperatureInCelsius = celsius
    }
}

let bodyTemperature = Celsius(37.0)
// bodyTemperature.temperatureInCelsius is 37.0
```

Características de los inicializadores

- Las propiedades opcionales se inicializan automáticamente a `nil`
- Durante la inicialización podemos modificar las propiedades constantes de la instancia, sólo se fijan cuando termina el `init()`
- Las subclases no pueden modificar las constantes heredadas

Inicialización de opcionales

```
class SurveyQuestion {
    var text: String
    var response: String?
    init(text: String) {
        self.text = text
    }
    func ask() {
        print(text)
    }
}

let cheeseQuestion = SurveyQuestion(text: "Do you like cheese?")
cheeseQuestion.ask()
// Prints "Do you like cheese?"

cheeseQuestion.response = "Yes, I do like cheese."
```

Inicialización de constantes

```
class SurveyQuestion {
    let text: String // modificable durante la inicialización
    var response: String?
    init(text: String) {
        self.text = text
    }
    func ask() {
        print(text)
    }
}

let beetsQuestion = SurveyQuestion(text: "How about beets?")
beetsQuestion.ask()
// Prints "How about beets?"

beetsQuestion.response = "I also like beets. (But not with cheese.)"
```

Inicializador por defecto

- Swift proporciona este inicializador si no hay ninguno definido y todas las propiedades tienen un valor predeterminado
- Este inicializador no recibe parámetros

Inicializador por defecto

```
class ShoppingListItem {
    var name: String?
    var quantity = 1
    var purchased = false
}

var item = ShoppingListItem()
```

Inicializador miembro a miembro para estructuras

- Se genera si no hay definidos inicializadores propios
- No importa que las propiedades de la estructura no tengan un valor predeterminado

Inicializador miembro a miembro para estructuras

```
struct Size {
    var width = 0.0, height = 0.0
}

let twoByTwo = Size(width: 2.0, height: 2.0)
```

Delegación de inicializadores

- Permite que unos inicializadores llamen a otros para no duplicar código
- En los tipos por valor (estructuras y enumeraciones) no hay herencia, así que sólo se puede delegar en los inicializadores definidos en el mismo tipo
- En los tipos por referencia (clases) el mecanismo de herencia permite además llamar a inicializadores de las clases padre

Delegación en tipos por valor

```
struct Size {
    var width = 0.0, height = 0.0
}

struct Point {
    var x = 0.0, y = 0.0
}
```

Delegación en tipos por valor

```
struct Rect {
    var origin = Point()
    var size = Size()
    init() {}
    init(origin: Point, size: Size) {
        self.origin = origin
        self.size = size
    }
    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)
    }
}
```

Delegación en tipos por valor

```
let basicRect = Rect()

let originRect = Rect(origin: Point(x: 2.0, y: 2.0),
                      size: Size(width: 5.0, height: 5.0))

let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
                      size: Size(width: 3.0, height: 3.0))
```

Inicializadores y herencia

- Todas las propiedades almacenadas de una clase, incluidas las heredadas de su superclase deben recibir un valor en la inicialización

- Swift define dos tipos de inicializadores para clases, designados y de conveniencia

Inicializadores de clases

Tipo	Características
Designados	Inicializan todas las propiedades añadidas en la clase y llaman al inicializador adecuado de la superclase para que el proceso continue por la cadena de herencia
De conveniencia	Son para casos especiales o por comodidad, por ejemplo llamando al designado con parámetros por defecto y se prefijan con la palabra clave `convenience`

Cadena de inicialización

1. Los inicializadores designados deben llamar a un inicializador designado de su superclase inmediata

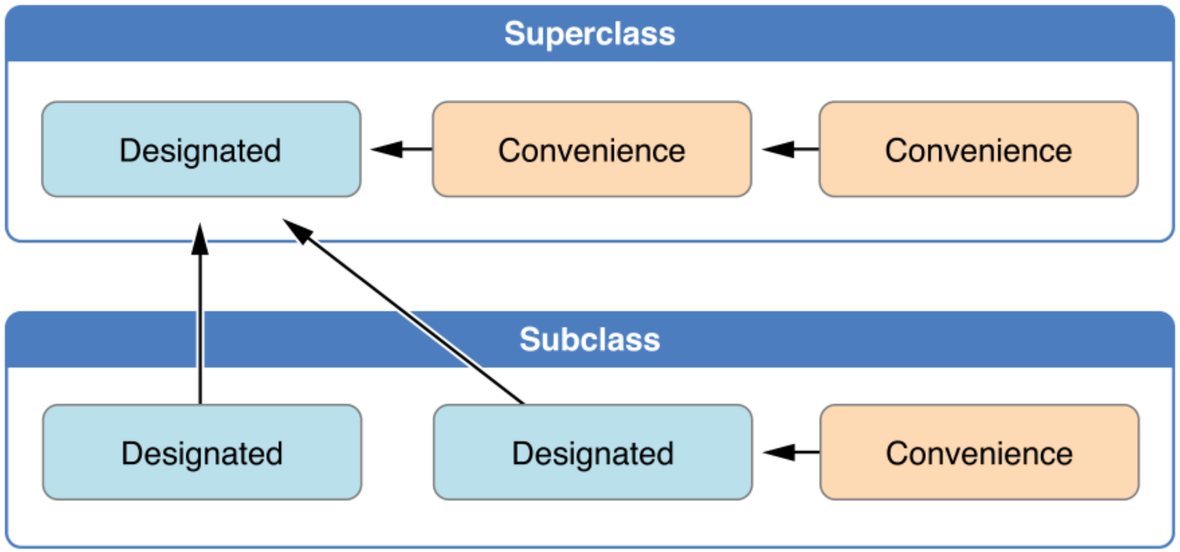
2. Los inicializadores de conveniencia deben llamar a otro inicializador en la misma clase

3. Los inicializadores de conveniencia deben terminar llamando a un inicializador designado

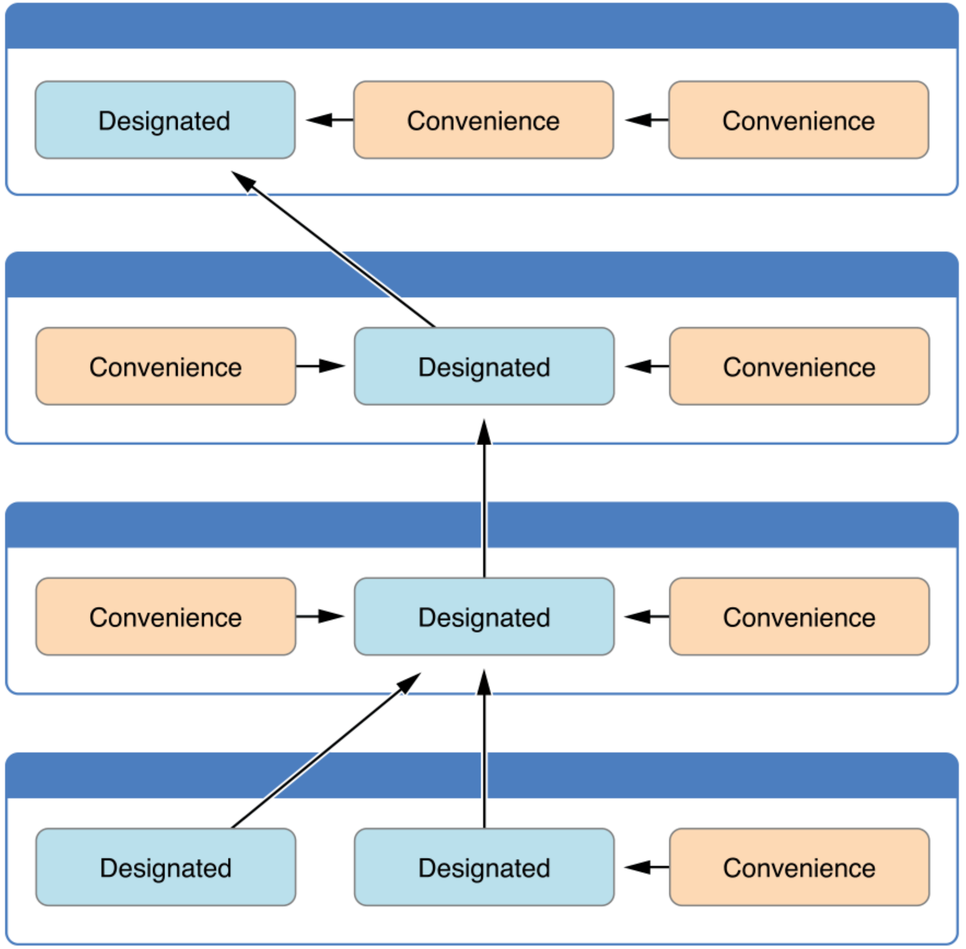
Cadena de inicialización

- Los inicializadores designados delegan hacia arriba
- Los inicializadores de conveniencia delegan al mismo nivel

Cadena de inicialización



Cadena de inicialización



Inicialización en dos fases

- Las propiedades almacenadas de una clase reciben un valor inicial por parte de la clase que las introduce
- Cada clase tiene oportunidad de modificar sus propiedades almacenadas antes de que la instancia esté lista para usar

Comprobaciones de seguridad

1. Un inicializador designado debe garantizar que todas las propiedades introducidas por su clase están inicializadas antes de delegar en el inicializador de su superclase

2. Un inicializador designado debe delegar en el inicializador de la superclase antes de modificar propiedades heredadas. Si no lo hace los valores serán sobrescritos por la superclase

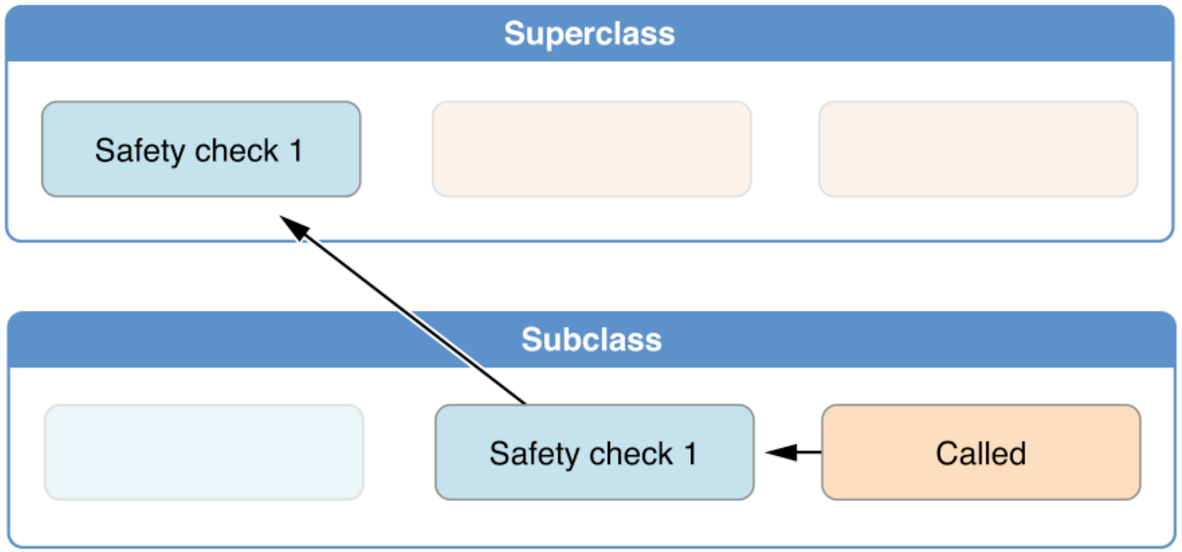
3. Un inicializador de conveniencia debe delegar en otro antes de modificar cualquier valor. Si no lo hace los valores serán sobrescritos por el inicializador designado de la clase

4. Un inicializador no puede llamar a métodos de instancia, leer valores de propiedades o utilizar self hasta que se haya completado la primera fase de inicialización

Fase 1

- Se llama a un inicializador designado o de conveniencia
- Se reserva memoria para la nueva instancia
- Un inicializador designado confirma que todas las propiedades introducidas por la clase tienen valor. Se inicializa la memoria de esas propiedades
- El inicializador designado delega en el inicializador de la superclase
- El proceso continua por la cadena de herencia hasta la clase base

Fase 1

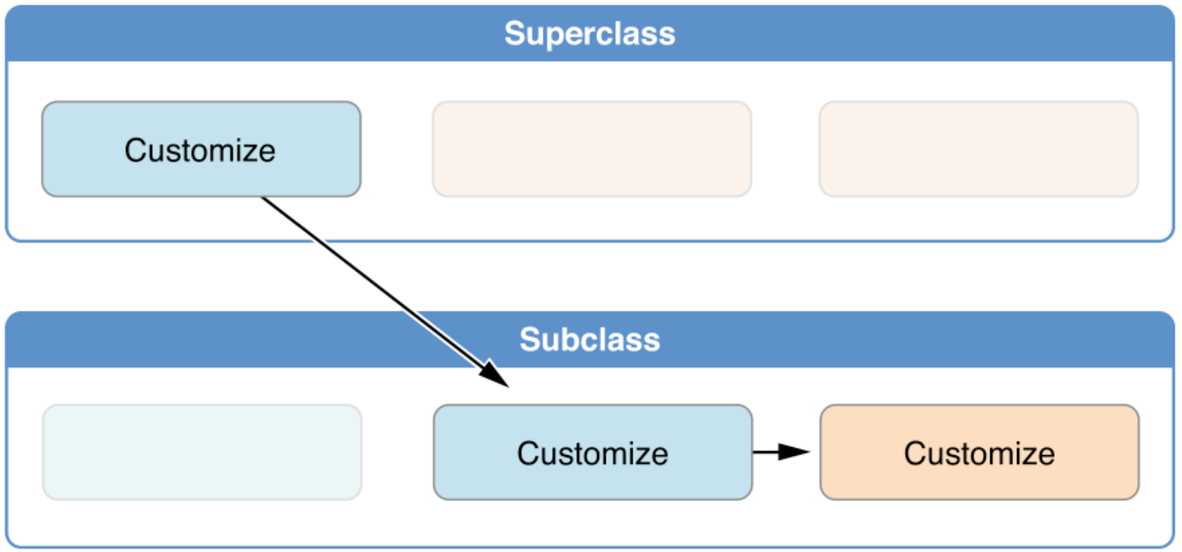


Fase 2

- Comenzando por la cima de la cadena de herencia, los inicializadores pueden personalizar más las instancias. Pueden acceder a self, cambiar propiedades y llamar a métodos

- Los inicializadores de conveniencia tienen la oportunidad de modificar la instancia y acceder a self

Fase 2



Herencia de inicializadores

- Las subclases, por defecto, no heredan los inicializadores
- Las subclases pueden reemplazar inicializadores de las superclases
- Si es designado se puede reemplazar y llamar a la versión de la superclase
- Si es de conveniencia, el reemplazo debe llamar a un inicializador designado de su propia clase
- No es necesarios añadir `override`

Herencia de inicializadores

```
class Vehicle {
    var numberOfWheels = 0
    var description: String {
        return "\(numberOfWheels) wheel(s)"
    }
}

let vehicle = Vehicle()
print("Vehicle: \(vehicle.description)")
// Vehicle: 0 wheel(s)
```

Herencia de inicializadores

```
class Bicycle: Vehicle {
    override init() {
        super.init()
        numberOfWheels = 2
    }
}

let bicycle = Bicycle()
print("Bicycle: \(bicycle.description)")
// Bicycle: 2 wheel(s)
```

Herencia automática de inicializadores

- Se supone que se proporcionan valores por defecto para todas las propiedades nuevas que introduce la subclase

Herencia automática de inicializadores

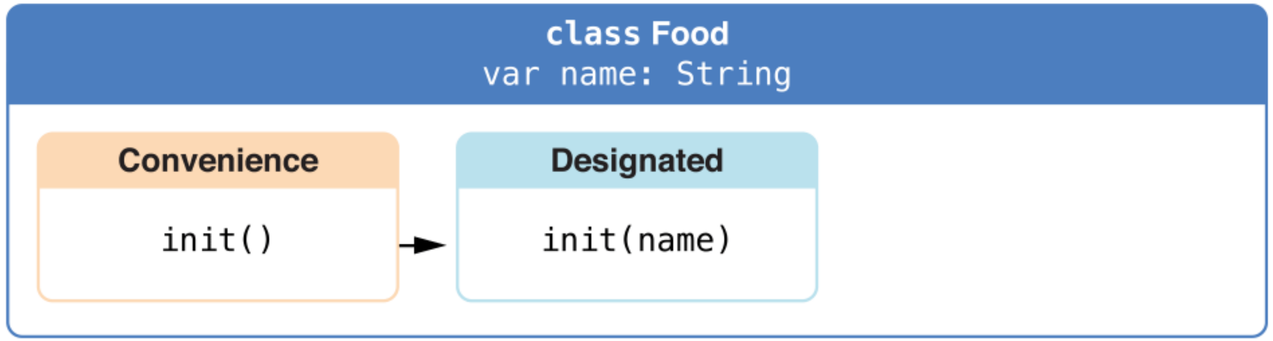
1. Si la subclase no define ningún inicializador designado, hereda automáticamente los designados de la superclase

2. Si la subclase proporciona implementación de todos los designados de la superclase, heredará automáticamente todos los inicializadores de conveniencia de la superclase

Ejemplo de inicializadores

```
class Food {
    var name: String
    init(name: String) {
        self.name = name
    }
    convenience init() {
        self.init(name: "[Unnamed]")
    }
}
```

Ejemplo de inicializadores



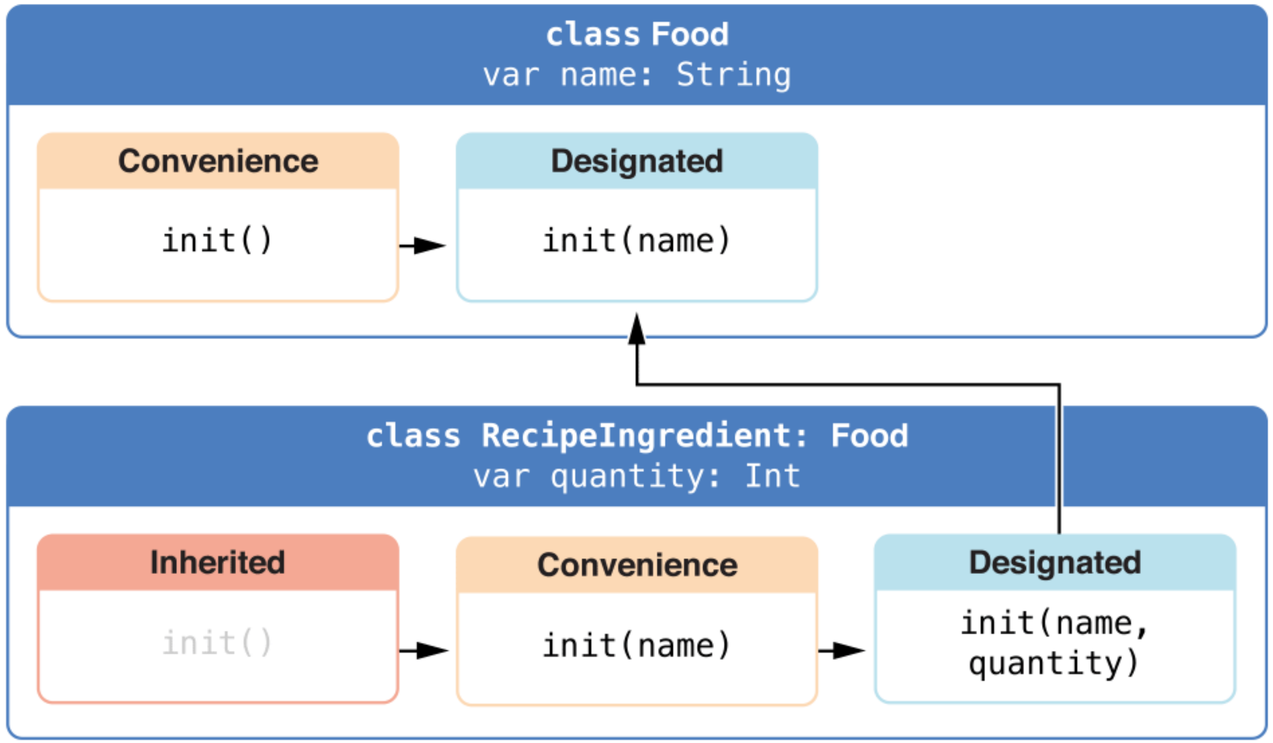
Ejemplo de inicializadores

```
let namedMeat = Food(name: "Bacon")
// namedMeat's name is "Bacon"

let mysteryMeat = Food()
// mysteryMeat's name is "[Unnamed]"
```

Ejemplo de inicializadores

```
class RecipeIngredient: Food {
    var quantity: Int
    init(name: String, quantity: Int) {
        self.quantity = quantity
        super.init(name: name)
    }
    override convenience init(name: String) {
        self.init(name: name, quantity: 1)
    }
}
```

Ejemplo de inicializadores



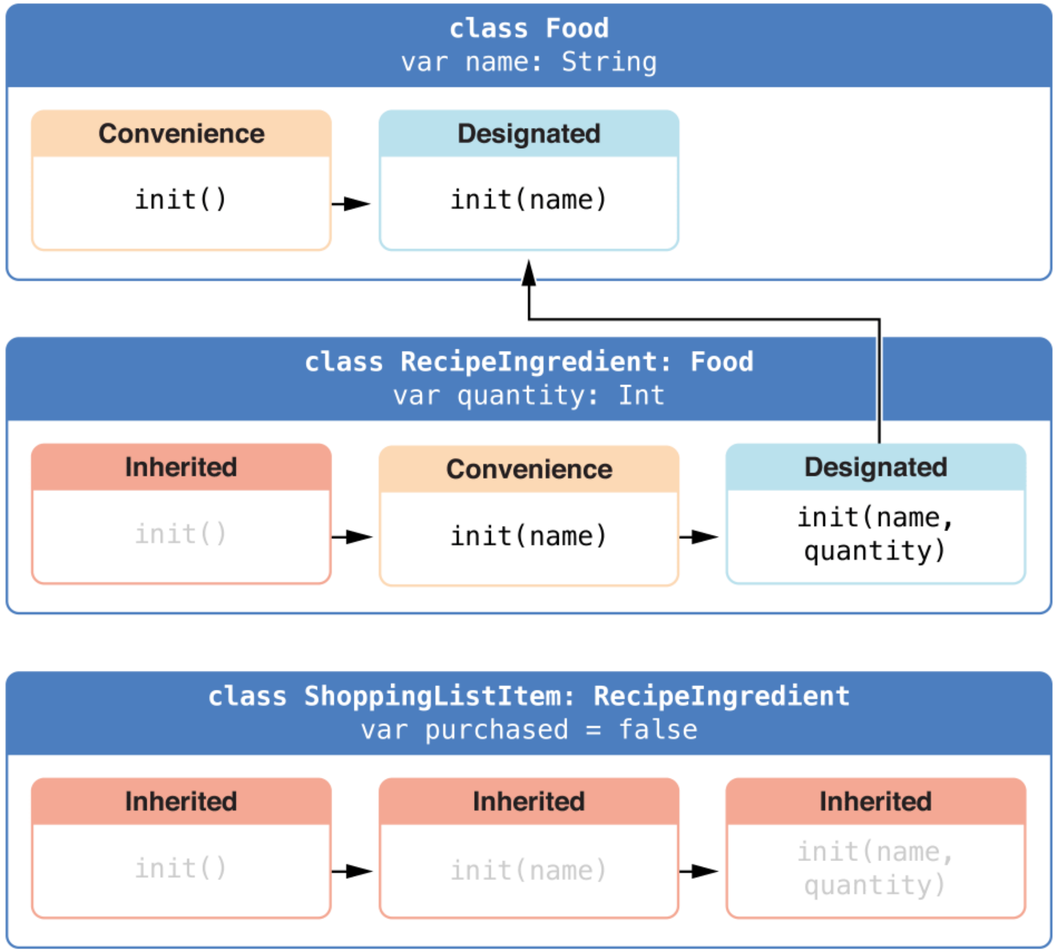
Ejemplo de inicializadores

```
let oneMysteryItem = RecipeIngredient()
let oneBacon = RecipeIngredient(name: "Bacon")
let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)
```

Ejemplo de inicializadores

```
class ShoppingListItem: RecipeIngredient {
    var purchased = false
    var description: String {
        var output = "\(quantity) x \(name)"
        output += purchased ? " ✔" : " ✘"
        return output
    }
}
```

Ejemplo de inicializadores



Ejemplo de inicializadores

```
var breakfastList = [
    ShoppingListItem(),
    ShoppingListItem(name: "Bacon"),
    ShoppingListItem(name: "Eggs", quantity: 6),
]
breakfastList[0].name = "Orange juice"
breakfastList[0].purchased = true
for item in breakfastList {
    print(item.description)
}
// 1 x Orange juice ✔
// 1 x Bacon ✘
// 6 x Eggs ✘
```

Desinicialización

Desinicialización

- Es el proceso que se ejecuta cuando se liberan los recursos de una instancia mediante ARC
- Sólo está disponible para clases
- Se realiza mediante el método especial `deinit()`
- Normalmente sólo lo implementaremos cuando tengamos que liberar recursos externos (ficheros, conexiones de red...)

Desinicialización

```
deinit {
    // perform the deinitialization
}
```

Ejemplo de desinicialización

```
class Bank {
    static var coinsInBank = 10_000
    static func distribute(coins numberOfCoinsRequested: Int) -> Int {
        let numberOfCoinsToVend = min(numberOfCoinsRequested, coinsInBank)
        coinsInBank -= numberOfCoinsToVend
        return numberOfCoinsToVend
    }
    static func receive(coins: Int) {
        coinsInBank += coins
    }
}
```

Ejemplo de desinicialización

```
class Player {
    var coinsInPurse: Int
    init(coins: Int) {
        coinsInPurse = Bank.distribute(coins: coins)
    }
    func win(coins: Int) {
        coinsInPurse += Bank.distribute(coins: coins)
    }
    deinit {
        Bank.receive(coins: coinsInPurse)
    }
}
```

Ejemplo de desinicialización

```
var playerOne: Player? = Player(coins: 100)

print("A new player has joined the game with \(playerOne!.coinsInPurse) coins")
// Prints "A new player has joined the game with 100 coins"

print("There are now \(Bank.coinsInBank) coins left in the bank")
// Prints "There are now 9900 coins left in the bank"
```

Ejemplo de desinicialización

```
playerOne!.win(coins: 2_000)

print("PlayerOne won 2000 coins & now has \(playerOne!.coinsInPurse) coins")
// Prints "PlayerOne won 2000 coins & now has 2100 coins"

print("The bank now only has \(Bank.coinsInBank) coins left")
// Prints "The bank now only has 7900 coins left"
```

Ejemplo de desinicialización

```
playerOne = nil

print("PlayerOne has left the game")
// Prints "PlayerOne has left the game"

print("The bank now has \(Bank.coinsInBank) coins")
// Prints "The bank now has 10000 coins"
```

ARC

ARC

- Automatic Reference Counting (recuento automático de referencias)
- Cuando se generan instancias, ARC reserva automáticamente la memoria
- Si una instancia ya no se necesita, la memoria se libera automáticamente
- Para saber si una instancia no está en uso, se lleva un recuento del número de referencias que la apuntan y cuando llega a 0 se libera
- Mientras haya por lo menos una “referencia fuerte” (una referencia almacenada en una propiedad, constante o variable) la instancia seguirá existiendo

Ejemplo de ARC

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

Ejemplo de ARC

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

reference1 = Person(name: "John Appleseed")
// Prints "John Appleseed is being initialized"

reference2 = reference1
reference3 = reference1

reference1 = nil
reference2 = nil

reference3 = nil
// Prints "John Appleseed is being deinitialized"
```

Referencias cíclicas

- Puede ocurrir que el recuento de referencias de una instancia nunca llegue a 0
- Si dos instancias mantienen referencias mutuas, se produce un ciclo que las mantiene a ambas “vivas” aunque ya no sean necesarias
- Para evitarlo, se usan las referencias `weak` y `unowned`

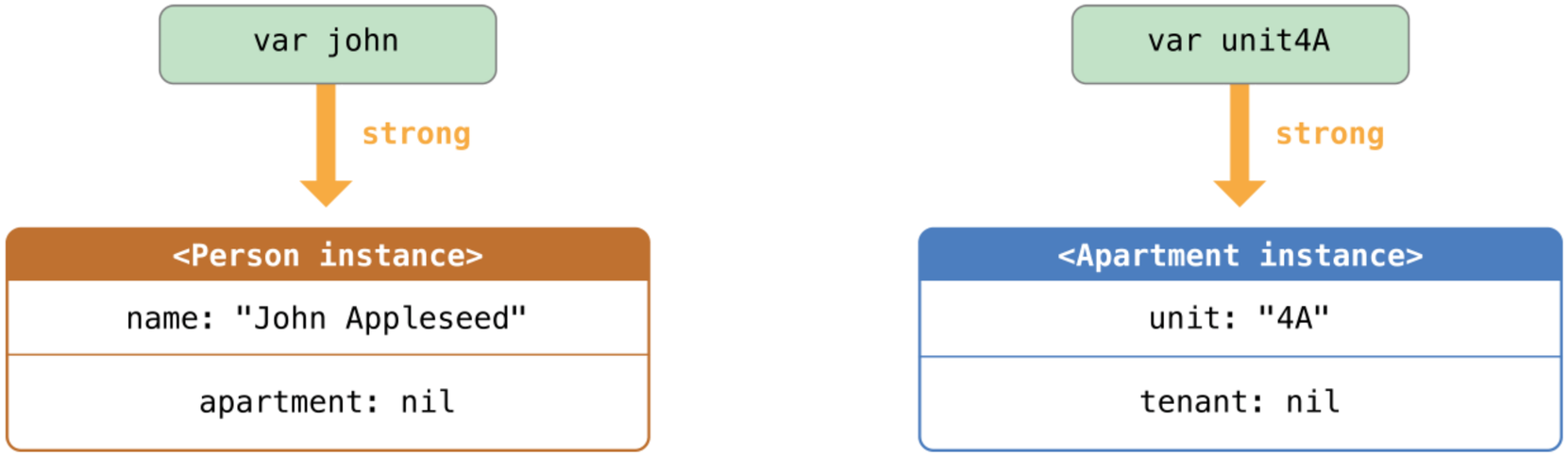
Referencias cíclicas

```
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") }
}
```

Referencias cíclicas

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

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

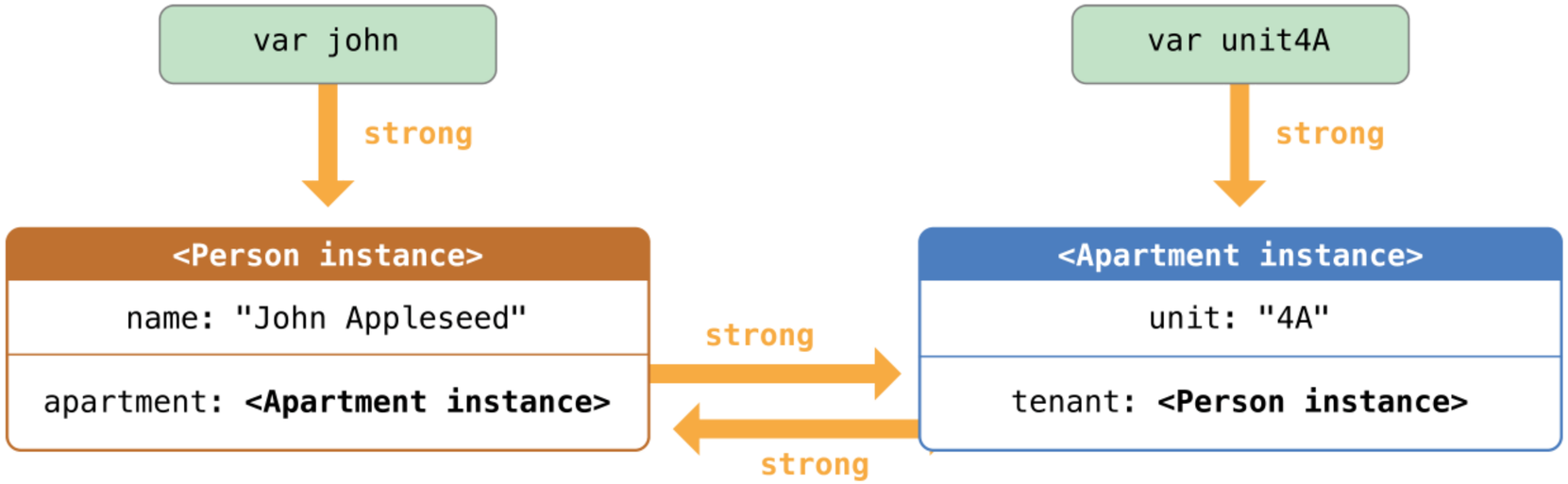
Referencias cíclicas



Referencias cíclicas

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

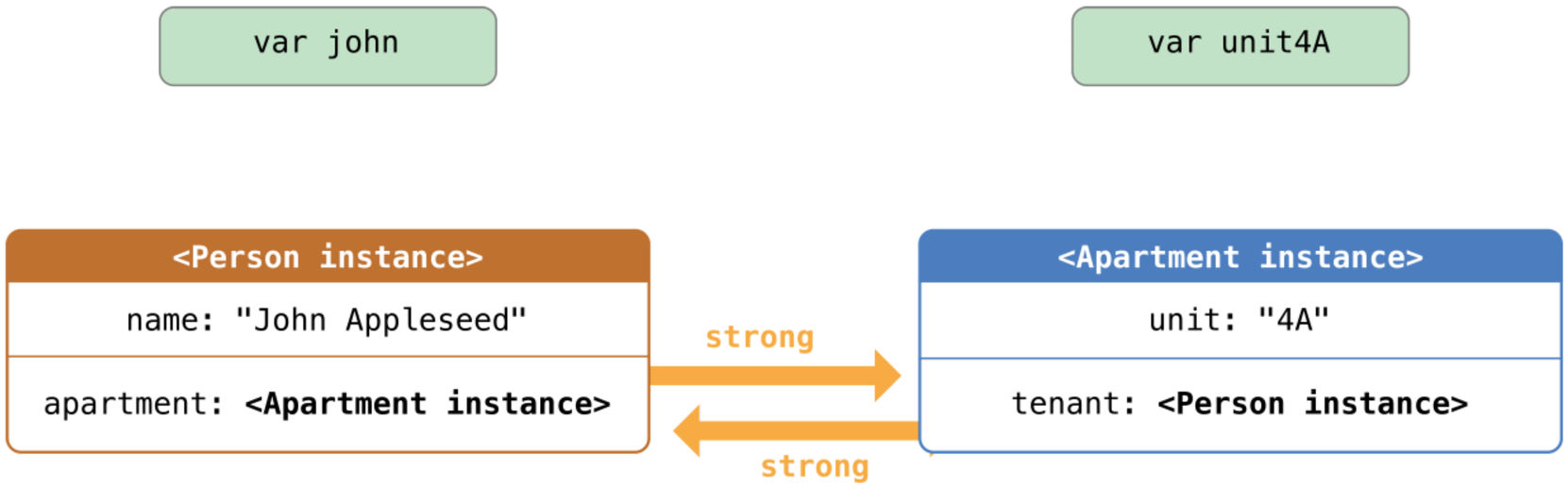
Referencias cíclicas



Referencias cíclicas

```
john = nil
unit4A = nil
// No se muestran los mensajes de los deinit()
```

Referencias cíclicas



weak y unowned

- Ambas crean “referencias débiles”
- `weak` supone que puede no haber una instancia asociada y se define como un opcional
- `unowned` se utiliza cuando siempre va a haber una instancia asociada (si no la hay, tendremos un error de tiempo de ejecución)

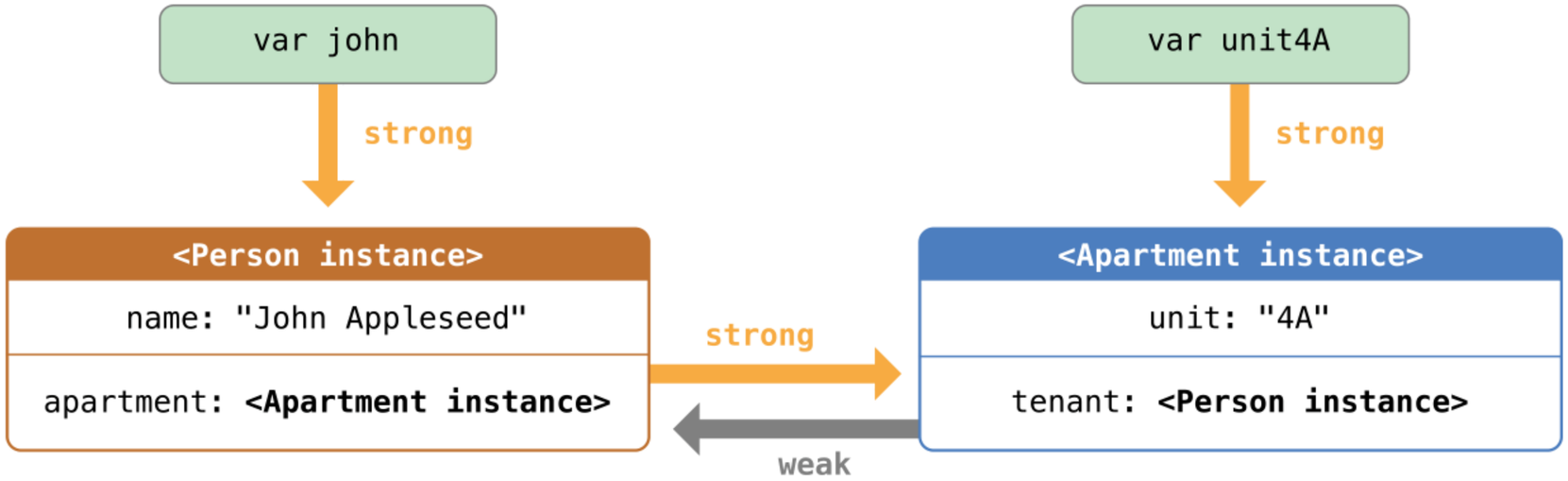
Referencias débiles con weak

```
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") }
}
```

Referencias débiles con weak

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

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

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

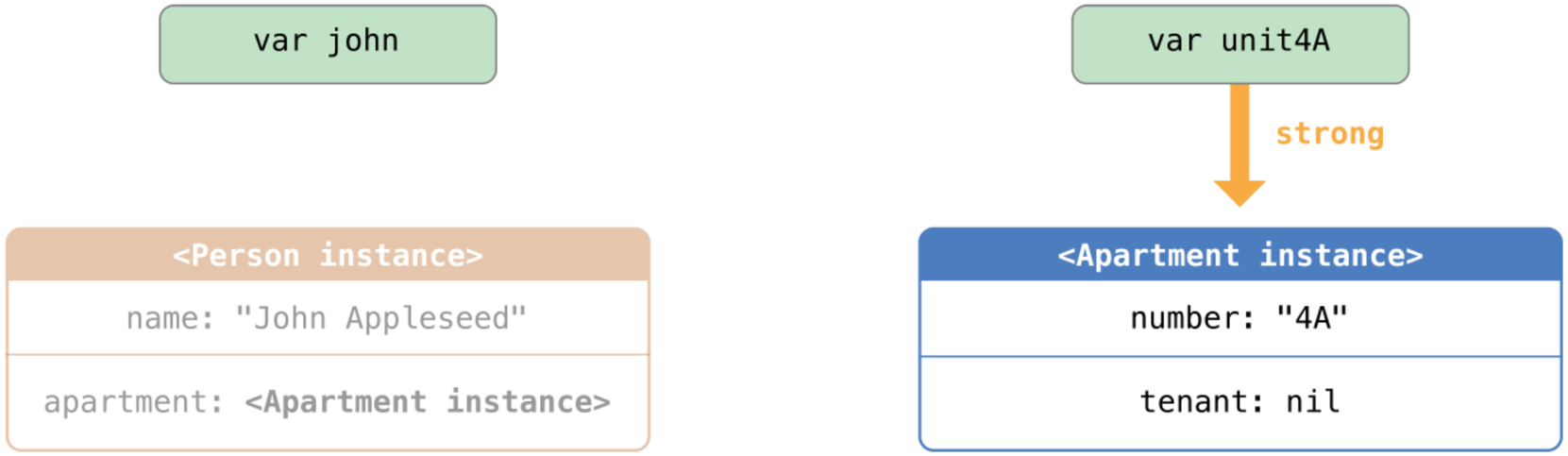
Referencias débiles con weak



Referencias débiles con weak

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

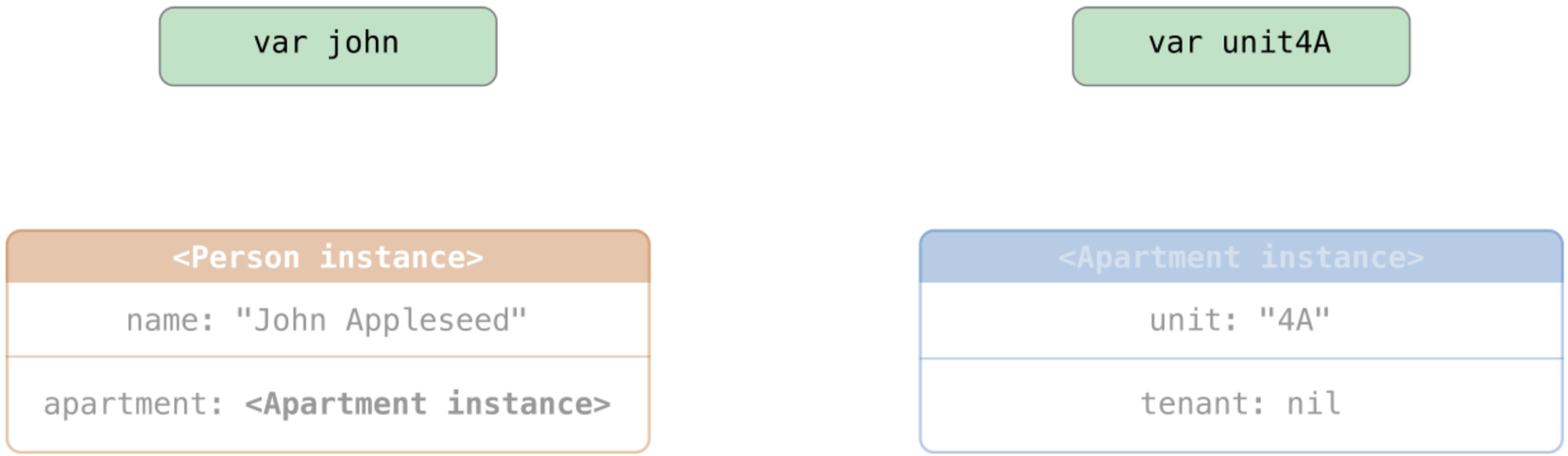
Referencias débiles con weak



Referencias débiles con weak

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

Referencias débiles con weak



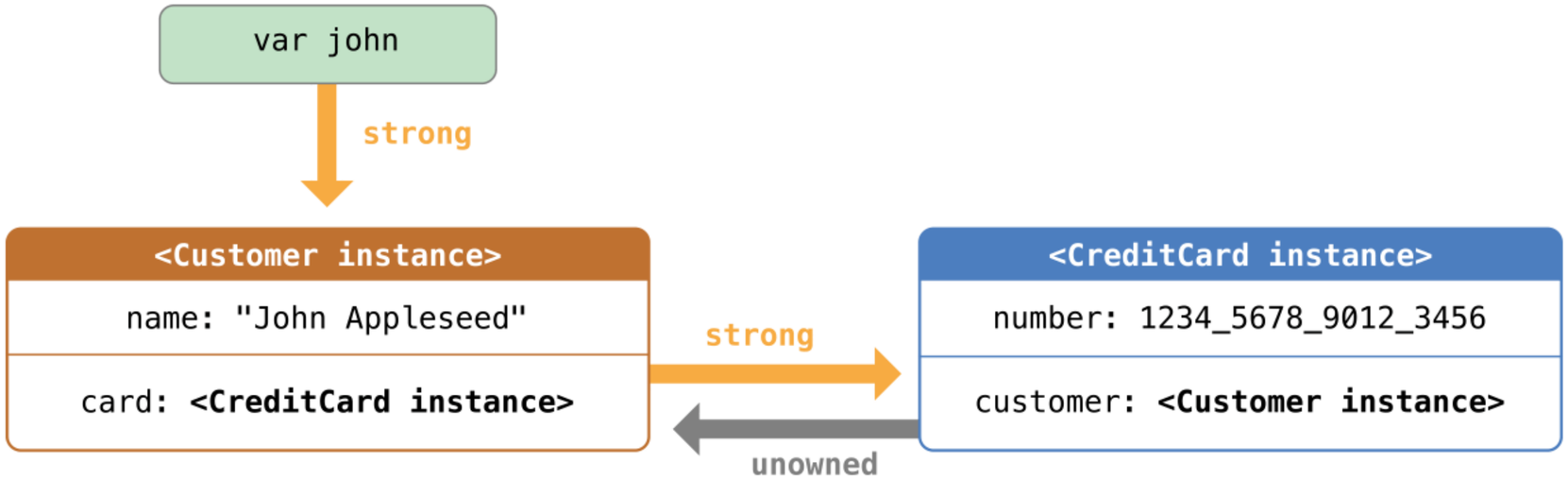
Referencias débiles con unowned

```
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") }
}
```

Referencias débiles con unowned

```
var john: Customer?

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

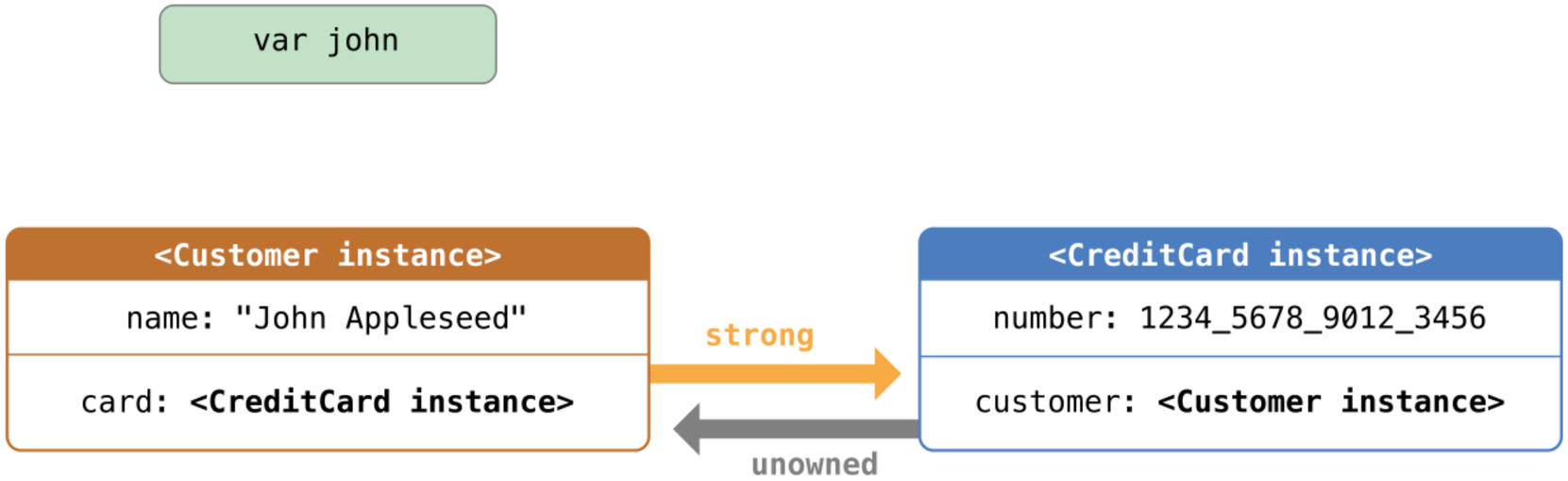
Referencias débiles con unowned



Referencias débiles con unowned

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

Referencias débiles con unowned



Posibles situaciones

- Ambas pueden ser `nil`: se resuelve con `weak`
- Una de ellas puede ser `nil` pero la otra no: se resuelve con `unowned`
- Si ninguna de las dos puede ser `nil`, usaremos `unowned` y un opcional implícito definido con `!`

Referencias débiles con unowned y opcional implícito

```
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
    }
}
```

Referencias débiles con unowned y opcional implícito

```
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"
```

Referencias cíclicas con clausuras

- Es el mismo problema de dos instancias manteniendo “vivas” una a la otra
- En este caso se da entre una clase y una clausura asignada a una propiedad de la clase

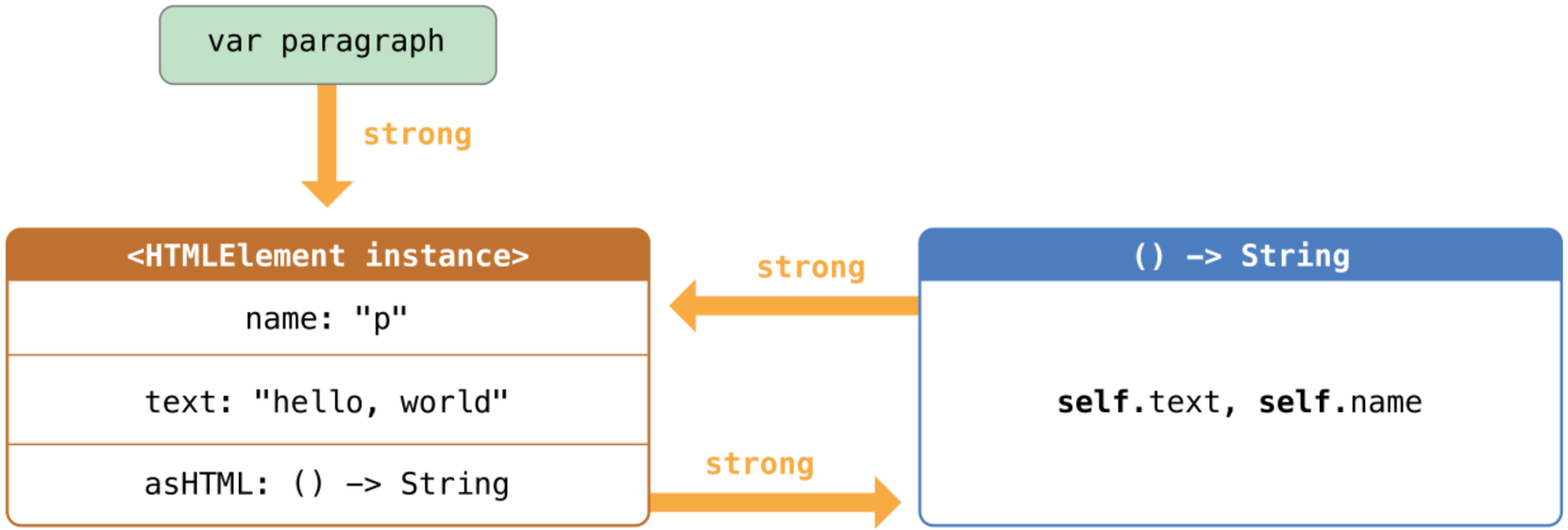
Referencias cíclicas con clausuras

```
class HTMLElement {

    let name: String
    let text: String?

    lazy var asHTML: () -> String = {
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

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

}
```

Referencias cíclicas con clausuras

```
let heading = HTMLElement(name: "h1")
let defaultText = "some default text"

heading.asHTML = {
    return "<\(heading.name)>\(heading.text ?? defaultText)</\(heading.name)>"
}

print(heading.asHTML())
// Prints "<h1>some default text</h1>"

var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
print(paragraph!.asHTML())
// Prints "<p>hello, world</p>"
```

Referencias cíclicas con clausuras



Referencias cíclicas con clausuras

```
paragraph = nil
// no muestra el deinit()
```

Listas de captura

- La lista de captura define las reglas a utilizar al capturar uno o más tipos por referencia dentro del cuerpo de la clausura

Listas de captura

- Si la referencia capturada puede ser `nil`: se marca como `weak`
- Si nunca va a ser `nil` se marca como `unowned`

Listas de captura

```
lazy var someClosure: (Int, String) -> String = {
    [unowned self, weak delegate = self.delegate!] (index: Int,
                                                    stringToProcess: String) -> String in
    // closure body goes here
}
```

```
lazy var someClosure: () -> String = {
    [unowned self, weak delegate = self.delegate!] in
    // closure body goes here
}
```

Listas de captura

- Swift exige que escribamos `self` al acceder a miembros de instancia desde una clausura, para ayudar a recordar que el posible capturar `self` por accidente

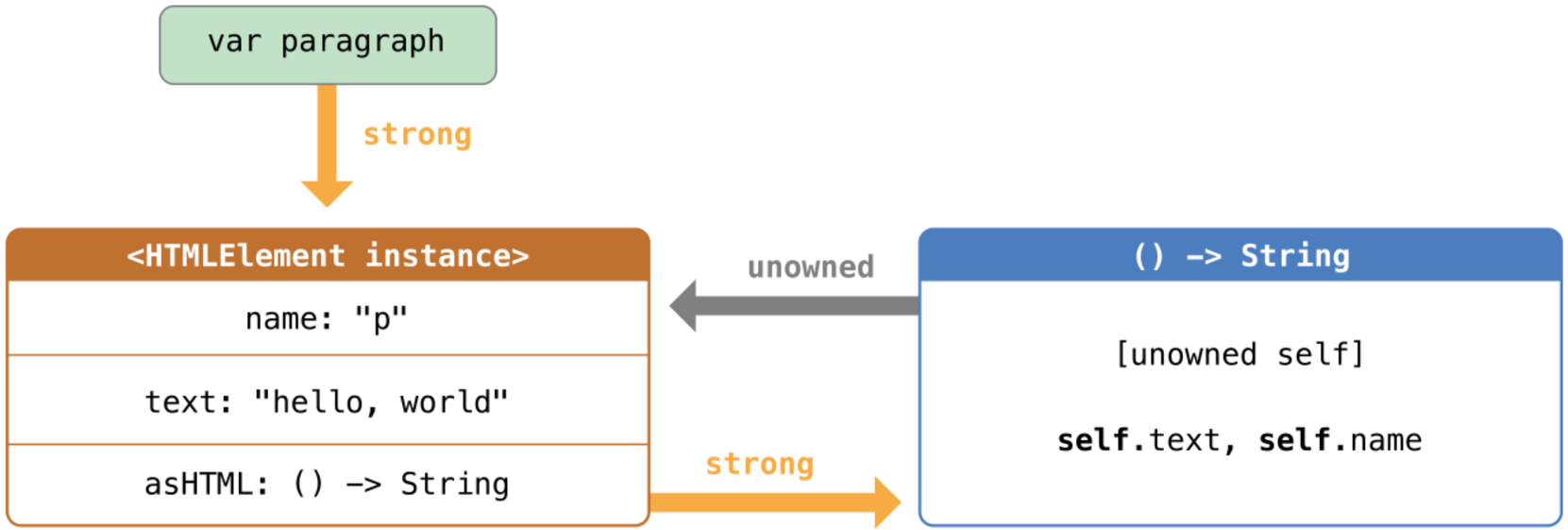
Listas de captura

```
class HTMLElement {

    let name: String
    let text: String?

    lazy var asHTML: () -> String = {
        [unowned self] in
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

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

}
```

Listas de captura

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello, world")
print(paragraph!.asHTML())
// Prints "<p>hello, world</p>"
```

Listas de captura



Listas de captura

```
paragraph = nil
// Prints "p is being deinitialized"
```