

# 错误处理



扫码试看/订阅

《Swift核心技术与实战》视频课程

# 错误表示

- 在 swift 中如果我们要定义一个表示错误的类型非常简单，只要遵循 Error 协议就可以了，我们通常用枚举或结构体来表示错误类型，枚举可能用的多些，因为它能更直观的表达当前错误类型的每种错误细节。

```
enum VendingMachineError: Error {  
    case invalidSelection  
    case insufficientFunds(coinsNeeded: Int)  
    case outOfStock  
}
```

# 如何抛出错误

- 函数、方法和初始化器都可以抛出错误。需要在参数列表后面，返回值前面加 throws 关键字。

```
func canThrowErrors() throws -> String
```

```
func cannotThrowErrors() -> String
```

# 如何抛出错误

```
struct Item {  
    var price: Int  
    var count: Int  
}
```

```
class VendingMachine {  
    var inventory = [  
        "Candy Bar": Item(price: 12, count: 7),  
        "Chips": Item(price: 10, count: 4),  
        "Pretzels": Item(price: 7, count: 11)  
    ]  
    var coinsDeposited = 0  
  
    func vend(itemName: String) throws {  
        guard let item = inventory[itemName] else {  
            throw VendingMachineError.invalidSelection  
        }  
  
        guard item.count > 0 else {  
            throw VendingMachineError.outOfStock  
        }  
  
        guard item.price <= coinsDeposited else {  
            throw VendingMachineError.insufficientFunds(coinsNeeded: item.price - coinsDeposited)  
        }  
  
        coinsDeposited -= item.price  
  
        var newItem = item  
        newItem.count -= 1  
        inventory[itemName] = newItem  
  
        print("Dispensing \(itemName)")  
    }  
}
```

# 如何抛出错误

```
let favoriteSnacks = [
    "Alice": "Chips",
    "Bob": "Licorice",
    "Eve": "Pretzels",
]
func buyFavoriteSnack(person: String, vendingMachine: VendingMachine) throws {
    let snackName = favoriteSnacks[person] ?? "Candy Bar"
    try vendingMachine.vend(itemNamed: snackName)
}
```

# 使用 Do-Catch 做错误处理

- 在 Swift 中我们使用 do-catch 块对错误进行捕获，当我们在调用一个 throws 声明的函数或方法时，我们必须把调用语句放在 do 语句块中，同时 do 语句块后面紧接着使用 catch 语句块。

```
do {  
    try expression  
    statements  
} catch pattern 1 {  
    statements  
} catch pattern 2 where condition {  
    statements  
} catch {  
    statements  
}
```

# 使用 Do-Catch 做错误处理

```
var vendingMachine = VendingMachine()
vendingMachine.coinsDeposited = 8
do {
    try buyFavoriteSnack(person: "Alice", vendingMachine: vendingMachine)
    print("Success! Yum.")
} catch VendingMachineError.invalidSelection {
    print("Invalid Selection.")
} catch VendingMachineError.outOfStock {
    print("Out of Stock.")
} catch VendingMachineError.insufficientFunds(let coinsNeeded) {
    print("Insufficient funds. Please insert an additional \(coinsNeeded) coins.")
} catch {
    print("Unexpected error: \(error).")
}
```



# try?

- try?会将错误转换为可选值，当调用try? + 函数或方法语句时候，如果函数或方法抛出错误，程序不会发崩溃，而返回一个nil，如果没有抛出错误则返回可选值。

```
func someThrowingFunction() throws -> Int {  
    // ...  
}  
  
let x = try? someThrowingFunction()  
  
let y: Int?  
do {  
    y = try someThrowingFunction()  
} catch {  
    y = nil  
}
```

# try!

- 如果你确信一个函数或者方法不会抛出错误，可以使用 try! 来中断错误的传播。但是如果错误真的发生了，你会得到一个运行时错误。

```
let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")
```

# 指定退出的清理动作

- defer 关键字：defer block 里的代码会在函数 return 之前执行，无论函数是从哪个分支 return 的，还是有 throw，还是自然而然走到最后一行。

```
func processFile(filename: String) throws {
    if exists(filename) {
        let file = open(filename)
        defer {
            close(file)
        }
        while let line = try file.readline() {
            // Work with the file.
        }
        // close(file) is called here, at the end of the scope.
    }
}
```

# 权限控制

# 模块和源文件

- 模块指的是独立的代码分发单元， 框架或应用程序会作为一个独立的模块来构建和发布。在 Swift 中,一个模块可以使用 `import` 关键字导入另外一个模块。
- 源文件就是 Swift 中的源代码文件, 它通常属于一个模块, 即一个应用程序或者框架。尽管我们一般会将不同的类型分别定义在不同的源文件中, 但是同一个源文件也可以包含多个类型、函数之类的定义。

# 访问级别

- open: 公开权限, 最高的权限, 可以被其他模块访问, 继承及复写。只能用于类和类的成员。
- public: 公有访问权限, 类或者类的公有属性或者公有方法可以从文件或者模块的任何地方进行访问。那么什么样才能成为一个模块呢? 一个App就是一个模块, 一个第三方API, 第三等方框架等都是一个完整的模块, 这些模块如果要对外留有访问的属性或者方法, 就应该使用 public 的访问权限。public 的权限在 Swift 3.0 后无法在其他模块被复写方法/属性或被继承。
- internal: 顾名思义, internal 是内部的意思, 即有着 internal 访问权限的属性和方法说明在模块内部可以访问, 超出模块内部就不可被访问了。在 Swift 中默认就是 internal 的访问权限。
- fileprivate: 文件私有访问权限, 被 fileprivate 修饰的类或者类的属性或方法可以在同一个物理文件中访问。如果超出该物理文件, 那么有着 fileprivate 访问权限的类, 属性和方法就不能被访问。
- private: 私有访问权限, 被 private 修饰的类或者类的属性或方法可以在同一个物理文件中的同一个类型(包含 extension)访问。如果超出该物理文件或不属于同一类型, 那么有着 private 访问权限的属性和方法就不能被访问。

# 潜规则1

- 如果一个类的访问级别是 `fileprivate` 或 `private` 那么该类的所有成员都是 `fileprivate` 或 `private`（此时成员无法修改访问级别），如果一个类的访问级别是 `open`、`internal` 或者 `public` 那么它的所有成员都是 `internal`，类成员的访问级别不能高于类的访问级别(注意：嵌套类型的访问级别也符合此条规则)。

## 潜规则2

- 常量、变量、属性、下标脚本访问级别低于其所声明的类型级别，并且如果不是默认访问级别（internal）要明确声明访问级别（例如一个常量是一个 private 类型的类类型，那么此常量必须声明为 private 或 fileprivate）。



## 潜规则3

- 在不违反1、2两条潜规则的情况下，setter 的访问级别可以低于 getter 的访问级别(例如一个属性访问级别是 internal，那么可以添加 private(set) 修饰将 setter 权限设置为 private，在当前模块中只有此源文件可以访问，对外部是只读的)。

## 潜规则4

- 必要构造方法（required 修饰）的访问级别必须和类访问级别相同，结构体的默认逐一构造函数的访问级别不高于其成员的访问级别（例如一个成员是 private 那么这个构造函数就是 private，但是可以通过自定义来声明一个 public 的构造函数），其他方法（包括其他构造方法和普通方法）的访问级别遵循潜规则1。

# 不透明类型

# why

```
protocol Shape {
    func draw() -> String
}

struct Triangle: Shape {
    var size: Int
    func draw() -> String {
        var result = [String]()
        for length in 1...size {
            result.append(String(repeating: "*", count: length))
        }
        return result.joined(separator: "\n")
    }
}
```

```
struct FlippedShape<T: Shape>: Shape {
    var shape: T
    func draw() -> String {
        let lines = shape.draw().split(separator: "\n")
        return lines.reversed().joined(separator: "\n")
    }
}

struct JoinedShape<T: Shape, U: Shape>: Shape {
    var top: T
    var bottom: U
    func draw() -> String {
        return top.draw() + "\n" + bottom.draw()
    }
}

struct Square: Shape {
    var size: Int
    func draw() -> String {
        let line = String(repeating: "*", count: size)
        let result = Array<String>(repeating: line, count: size)
        return result.joined(separator: "\n")
    }
}
```

# why

- 代码是可以编译通过的，但是 makeTrapezoid 的返回类型又臭又长，被暴露了出去。

```
func makeTrapezoid() -> JoinedShape<Triangle, JoinedShape<Square, FlippedShape<Triangle>>> {  
    let top = Triangle(size: 2)  
    let middle = Square(size: 2)  
    let bottom = FlippedShape(shape: top)  
    let trapezoid = JoinedShape(  
        top: top,  
        bottom: JoinedShape(top: middle, bottom: bottom)  
    )  
    return trapezoid  
}  
let trapezoid = makeTrapezoid()  
print(trapezoid.draw())
```

# why

- 不能将其Container用作函数的返回类型，因为该协议具有关联类型。也不能将它用作返回类型的泛型约束，因为函数体外没有足够的信息来推断泛型类型需要什么。

```
protocol Container {  
    associatedtype Item  
    var count: Int { get }  
    subscript(i: Int) -> Item { get }  
}  
extension Array: Container { }
```

```
// Error: Protocol with associated types can't be used as a return type.
```

```
func makeProtocolContainer<T>(item: T) -> Container {  
    return [item]  
}
```

❗ Protocol 'Container' can only be used as a generic constraint because it has Self or associated type requirements

```
// Error: Not enough information to infer C.
```

```
func makeProtocolContainer<T, C: Container>(item: T) -> C {  
    return [item]  
}
```

❗ Cannot convert return expression of type '[T]' to return type 'C'

# 解决问题

```
func makeTrapezoid() -> some Shape {  
    let top = Triangle(size: 2)  
    let middle = Square(size: 2)  
    let bottom = FlippedShape(shape: top)  
    let trapezoid = JoinedShape(  
        top: top,  
        bottom: JoinedShape(top: middle, bottom: bottom)  
    )  
    return trapezoid  
}  
let trapezoid = makeTrapezoid()  
print(trapezoid.draw())
```

# 解决问题

```
protocol Container {  
    associatedtype Item  
    var count: Int { get }  
    subscript(i: Int) -> Item { get }  
}  
extension Array: Container { }  
  
func makeProtocolContainer<T>(item: T) -> some Container {  
    return [item]  
}
```



# 返回不透明类型 vs 返回协议类型

- 返回 opaque 类型看起来非常类似于使用协议类型作为函数的返回类型，但这两种返回类型的不同之处在于它们是否保留了类型标识。opaque 类型是指一种特定类型，尽管函数的调用者不能看到是哪种类型；协议类型可以指代符合协议的任何类型。一般来说，协议类型为存储的值的基础类型提供了更大的灵活性，而不透明类型可以对这些基础类型做出更强有力的保证。

```
127
128 func invalidFlip<T: Shape>(_ shape: T) -> some Shape {
129     if shape is Square {
130         return shape
131     }
132     return FlippedShape(shape: shape)
133 }
134
```

```
error: Other.playground:128:6: error: function declares an opaque return type, but the r
func invalidFlip<T: Shape>(_ shape: T) -> some Shape {
  ^
```

```
Other.playground:130:16: note: return statement has underlying type 'T'
    return shape
    ^
```

```
Other.playground:132:12: note: return statement has underlying type 'FlippedShape<T>'
    return FlippedShape(shape: shape)
    ^
```

```
func invalidFlip<T: Shape>(_ shape: T) -> Shape {
    if shape is Square {
        return shape
    }
    return FlippedShape(shape: shape)
}
```

# 自动引用计数

# ARC

- Swift 使用自动引用计数(ARC)来跟踪并管理应用使用的内存。大部分情况下,这意味着在 Swift 语言中,内存管理“仍然工作”,不需要自己去考虑内存管理的事情。当实例不再被使用时,ARC会自动释放这些类的实例所占用的内存。
- 引用计数只应用在类的实例。结构体(Structure)和枚举类型是值类型,并非引用类型,不是以引用的方式来存储和传递的。

# ARC 如何工作

- 每次创建一个类的实例，ARC 就会分配一个内存块，用来存储这个实例的相关信息。这个内存块保存着实例的类型，以及这个实例相关的属性的值。
- 当实例不再被使用时，ARC 释放这个实例使用的内存，使这块内存可作它用。这保证了类实例不再被使用时，它们不会占用内存空间。
- 但是，如果 ARC 释放了仍在使用的实例，那么你就不能再访问这个实例的属性或者调用它的方法。如果你仍然试图访问这个实例，应用极有可能会崩溃。
- 为了保证不会发生上述的情况，ARC 跟踪与类的实例相关的属性、常量以及变量的数量。只要有一个有效的引用，ARC 都不会释放这个实例。
- 为了让这变成现实，只要你将一个类的实例赋值给一个属性或者常量或者变量，这个属性、常量或者变量就是这个实例的强引用(strong reference)。之所以称之为“强”引用，是因为它强持有这个实例，并且只要这个强引用还存在，就不能销毁实例。

# 循环引用

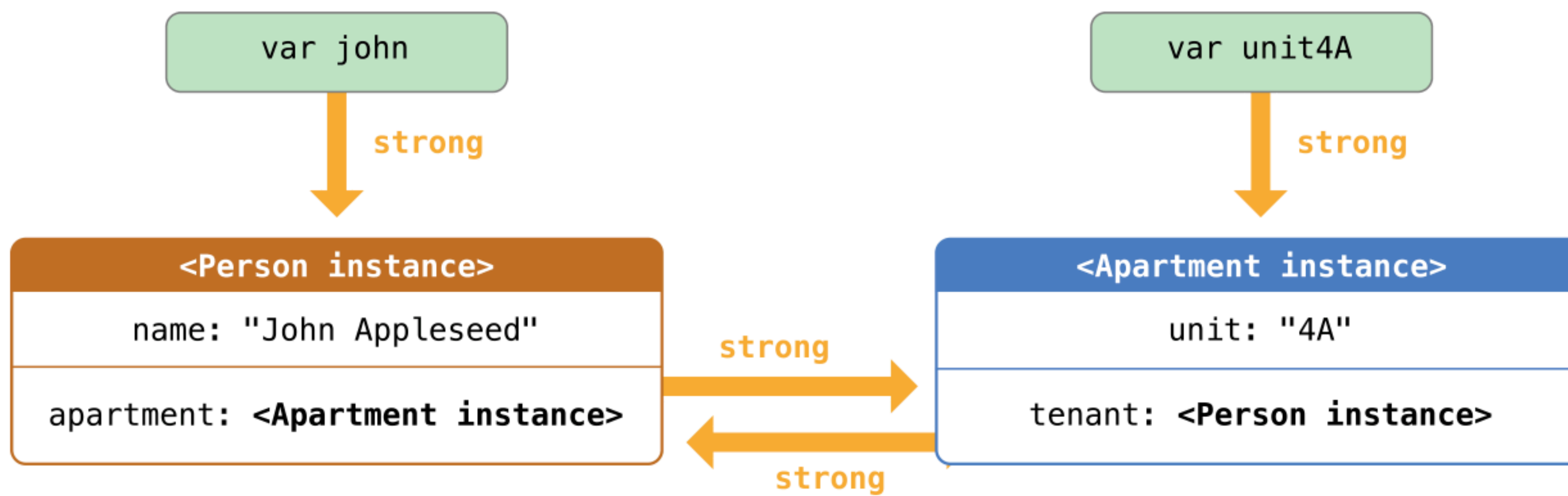
- 在两个类实例彼此保持对方的强引用，使得每个实例都使对方保持有效时会发生这种情况。我们称之为强引用环。
- 通过用弱引用或者无主引用来取代强引用，我们可以解决强引用环问题。

```
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") }  
}
```

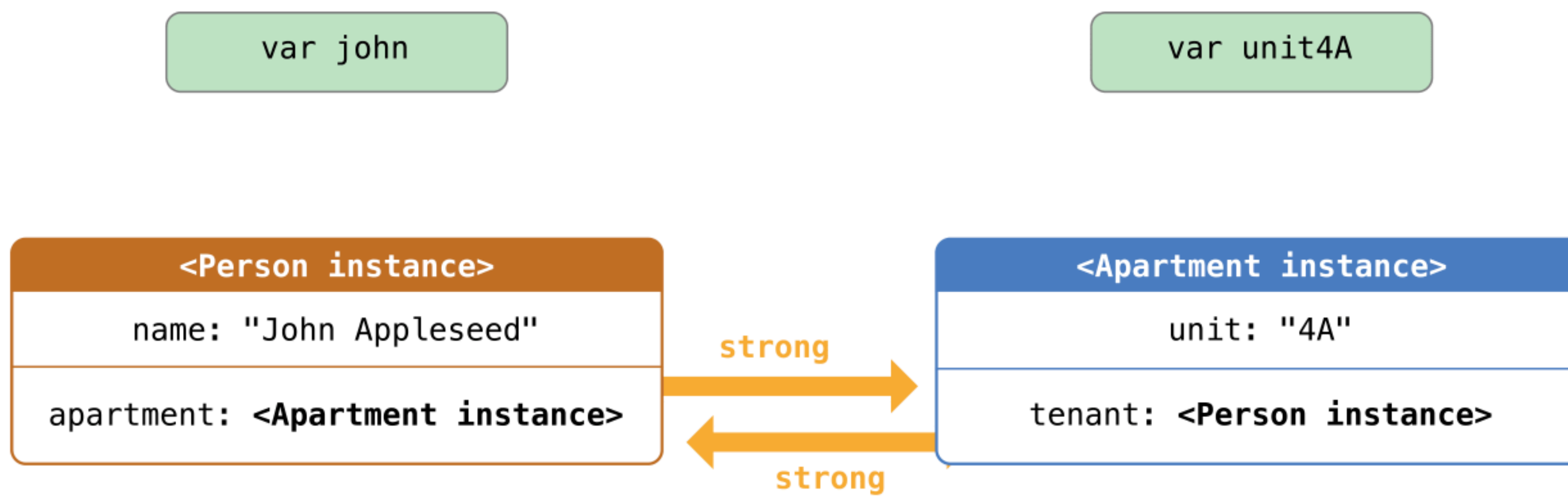
# 循环引用



# 循环引用



# 循环引用





# 解决循环引用

- 弱引用和无主引用允许引用环中的一个实例引用另外一个实例，但不是强引用。因此实例可以互相引用但是不会产生强引用环。
- 对于生命周期中引用会变为 nil 的实例，使用弱引用；对于初始化时赋值之后引用再也不会赋值为 nil 的实例，使用无主引用。

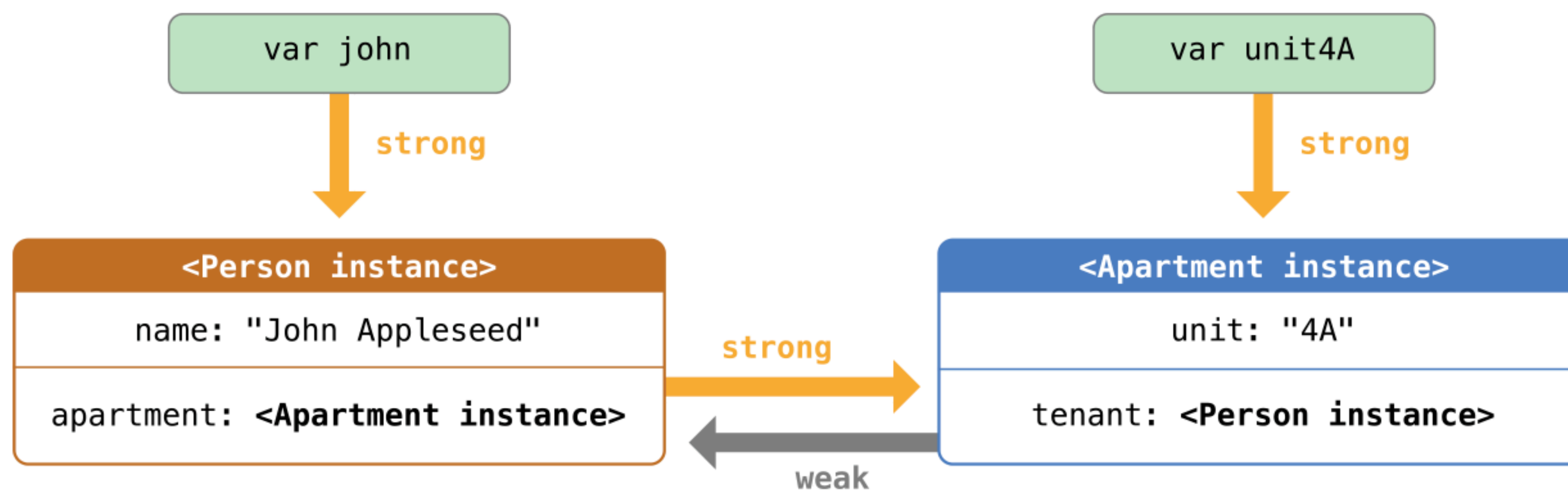
# 弱引用

- 弱引用不会增加实例的引用计数，因此不会阻止ARC销毁被引用的实例。这种特性使得引用不会变成强引用环。声明属性或者变量的时候，关键字weak表明引用为弱引用。
- 弱引用只能声明为变量类型，因为运行时它的值可能改变。弱引用绝对不能声明为常量。
- 因为弱引用可以没有值，所以声明弱引用的时候必须是可选类型的。在Swift语言中，推荐用可选类型来作为可能没有值的引用的类型。

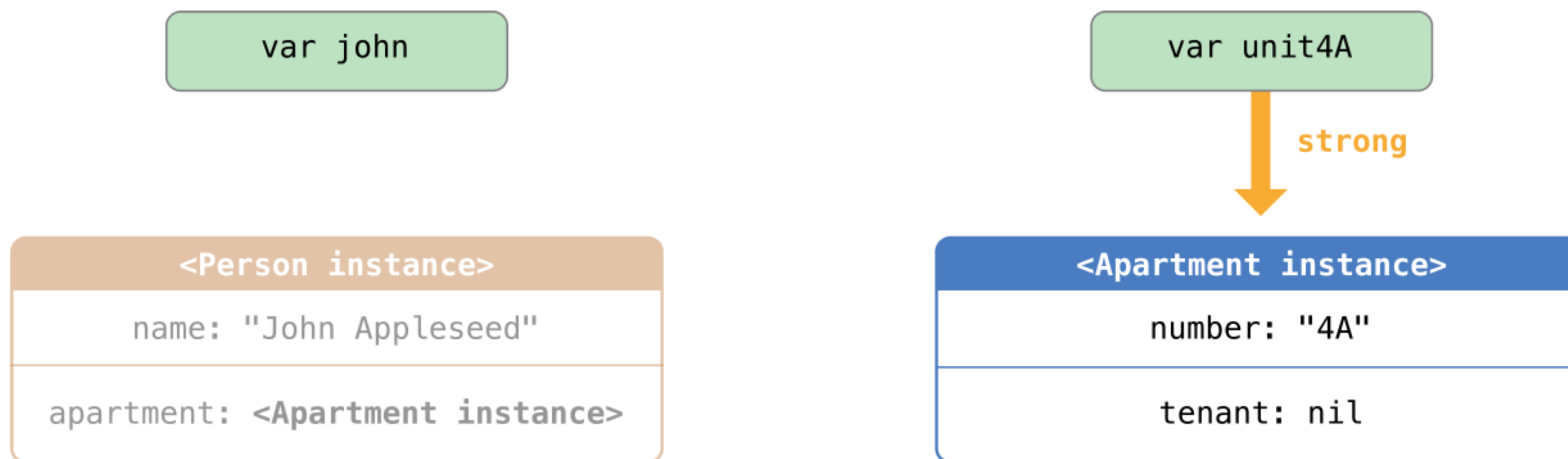
# 弱引用

```
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") }  
}
```

# 弱引用



# 弱引用



# 弱引用

var john

<Person instance>

name: "John Appleseed"

apartment: <Apartment instance>

var unit4A

<Apartment instance>

unit: "4A"

tenant: nil

# 无主引用

- 和弱引用相似，无主引用也不强持有实例。但是和弱引用不同的是，无主引用默认始终有值。因此，无主引用只能定义为非可选类型（non-optional type）。在属性、变量前添加 `unowned` 关键字，可以声明一个无主引用。
- 因为是非可选类型，因此当使用无主引用的时候，不需要展开，可以直接访问。不过非可选类型变量不能赋值为 `nil`，因此当实例被销毁的时候，ARC 无法将引用赋值为 `nil`。
- 当实例被销毁后，试图访问该实例的无主引用会触发运行时错误。使用无主引用时请确保引用始终指向一个未销毁的实例。

# 无主引用

```
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  
    }  
}
```



# 闭包引用循环

- 将一个闭包赋值给类实例的某个属性，并且这个闭包使用了实例，这样也会产生强引用环。这个闭包可能访问了实例的某个属性，例如 `self.someProperty`，或者调用了实例的某个方法，例如 `self.someMethod`。这两种情况都导致了闭包使用 `self`，从而产生了循环引用。

# 闭包引用循环解决

- 定义占有列表 - 占有列表中的每个元素都是由 weak 或者 unowned 关键字和实例的引用(如 self 或 someInstance)组成。每一对都在中括号中，通过逗号分开。
- 当闭包和占有的实例总是互相引用时并且总是同时销毁时，将闭包内的占有定义为无主引用。
- 相反的，当占有引用有时可能会是nil时，将闭包内的占有定义为弱引用。

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

# 内存安全性

# 内存安全

- 默认情况下，Swift 会克服代码层面上的一些不安全的行为，如：确保一个变量被初始化完后才能被访问、确保变量在销毁后不会被访问等等安全操作。
- Swift 也会确保在多路访问内存中同一区域时不会冲突（独占访问该区域）。通常情况下，我们完全无需考虑内存访问冲突的问题，因为 Swift 是自动管理内存的。然而，在码代码的时候，了解那些地方可能发生内存访问冲突是非常重要的。通常情况下，如果你的代码有内存访问冲突，那么 Xcode 会提示编译错误或者运行时错误。

# 内存安全

- 访问可以分两种：
  - 即时访问：即在访问开始至结束前都不可能有其他代码来访问同一区域。
  - 长期访问：即在访问开始至结束前可能有其他代码来访问同一区域。长期访问可能和其他即时访问或者长期访问重叠。

# inout 参数访问冲突

```
172
173 var stepSize = 1
174
175 func increment(_ number: inout Int) {
176     number += stepSize
177 }
178
179 increment(&stepSize)
180
```

13  
Simultaneous accesses to 0x10e71aef0, but modification requires exclusive access.  
Previous access (a modification) started at (0x10e71b34d).  
Current access (a read) started at:

0	libswiftCore.dylib	0x00000001058cac30	swift_beginAccess + 568
3	Other	0x0000000105541570	main + 0
4	CoreFoundation	0x00007fff23b0ca30	__CFRunLoopServiceMachPort + 197
5	CoreFoundation	0x00007fff23b07190	__CFRunLoopRun + 1671
6	CoreFoundation	0x00007fff23b06cb0	CFRunLoopRunSpecific + 438
7	GraphicsServices	0x00007fff38346b6f	GSEventRunModal + 65
8	UIKitCore	0x00007fff4757877b	UIApplicationMain + 1621
9	Other	0x0000000105541570	main + 205
10	libdyld.dylib	0x00007fff516ecd28	start + 1

Fatal access conflict detected.

# inout 参数访问冲突

```
func increment(_ number: inout Int){  
    number += stepSize  
}
```



# inout 参数访问冲突解决

```
var stepSize = 1

func increment(_ number: inout Int) {
    number += stepSize
}

// Make an explicit copy.
var copyOfStepSize = stepSize
increment(&copyOfStepSize)

// Update the original.
stepSize = copyOfStepSize
```

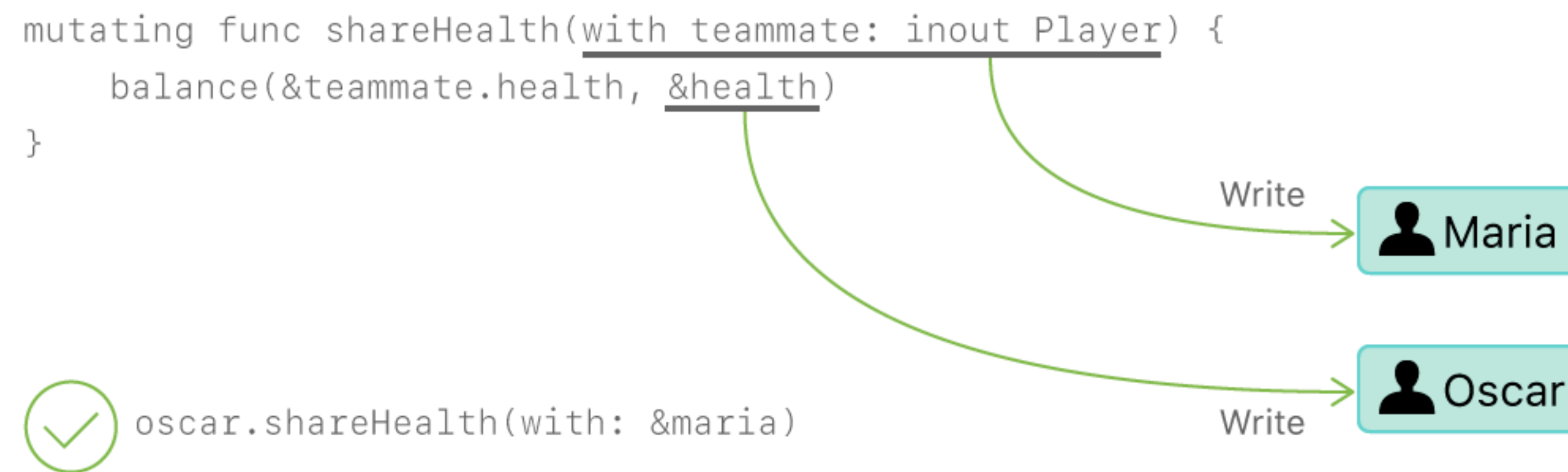


# self 访问冲突

```
struct Player {  
    var name: String  
    var health: Int  
    var energy: Int  
  
    static let maxHealth = 10  
    mutating func restoreHealth() {  
        health = Player.maxHealth  
    }  
}
```

```
extension Player {  
    mutating func shareHealth(with teammate: inout Player) {  
        balance(&teammate.health, &health)  
    }  
}  
  
var oscar = Player(name: "Oscar", health: 10, energy: 10)  
var maria = Player(name: "Maria", health: 5, energy: 10)  
oscar.shareHealth(with: &maria) // OK
```

# self 访问冲突



# self 访问冲突


```
oscar.shareHealth(with: &oscar)
```

```
mutating func shareHealth(with teammate: inout Player) {  
    balance(&teammate.health, &health)  
}
```



```
oscar.shareHealth(with: &oscar)
```

 Maria

 Oscar

Write

Write



扫码试看/订阅

《Swift核心技术与实战》视频课程