SCIENCES SORBONNE UNIVERSITÉ

Swift, memory management and consistency

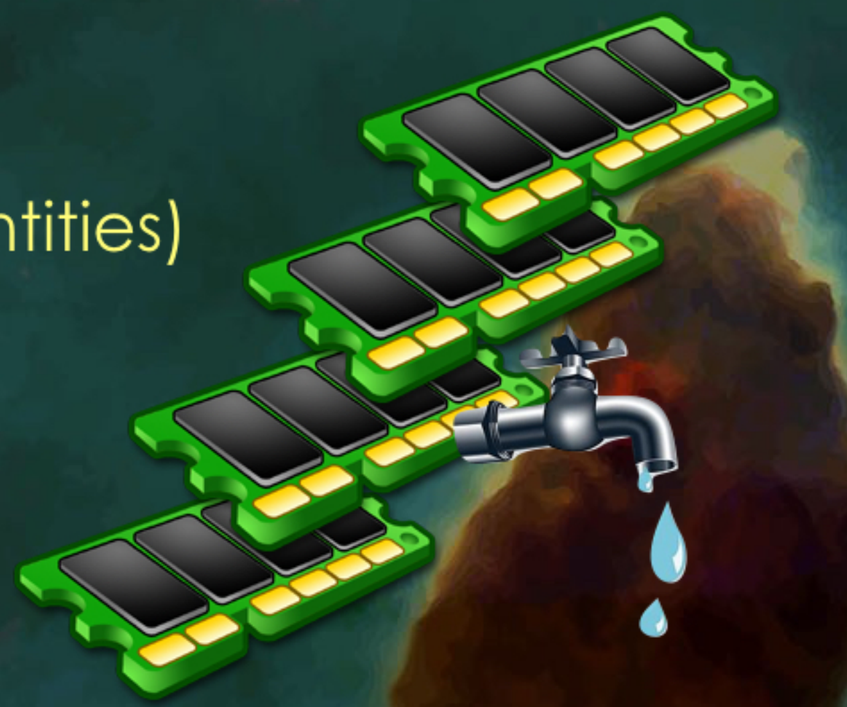
Fabrice.Kordon@lip6.fr

LIP6

As an introduction…

- **In Swift, memory is (almost) «for free»**

- **Safe memory management techniques**
  - Blocks (LIFO approach)
    - **E.g. function calls**
  - Classes (use of «classifiers» for entities)
    - **Mutable/immutable properties**
    - **Properties classification**
    - **Constructors/destructor**
  - Closures
    - **Management of the scope**



- **ARC always activated**
  - Impossible to «play manually» with memory

Object initialization (memory aspects)

Phase 1

- Invocation of an initializer
- Memory Allocation for he new instance
- Initializer confirms installation of properties
- Invocation of the super's initializer
  - **The memory allocated for he instance is valid**

Phase 2

- Finalizing initialization (self is available)
- Finalization of self via convenience init (when necessary)

Reminder about objet's liaisons

Strong

- Enclosed references must be «alive»
  - **Destroyed with the current object**
- Be aware of cycles that could hinder ARC's efficiency
- Default relation

Weak

- Enclosed reference may be «missing»
  - **Optional types**
- May «break» a cycle (and let ARC work)
- Keyword : weak

Unowned

- Like weak… but always have a value (like strong)
- Keyword: unowned

Reminder about objet's liaisons

How to deallocate?

Assign «nil» to a reference

Strong

- Enclosed refere
  - **Destroyed with th**
- Be aware of cycles that could hinder ARC's efficiency
- Default relation

Weak

- Enclosed reference may be «missing»
  - **Optional types**
- May «break» a cycle (and let ARC work)
- Keyword : weak

Unowned

- Like weak… but always have a value (like strong)
- Keyword: unowned

Reminder about objet's liaisons

Strong

- Enclosed refere
  - **Destroyed with th**
- Be aware of cycles that could hinder ARC's efficiency
- Default relati

Weak

- Enclosed refe
  - **Optional types**
- May «break» a cycl ork)
- Keyword : weak

Unowned

- Like weak… but alw e a value (like strong)
- Keyword: unowned

How to deallocate?

Assign «nil» to a reference

Is ARC efficient?

Object's liaisons must be well categorized…

Strong liaison, example


```
class Person {
    let name: String
    var atHome: Apartment?
    init(n: String) {
        name = n
    }
    deinit {
        print("\(name) is «deinitialized»")
    }
}

class Apartment {
    let number: Int
    var owner: Person?
    init(n: Int) {
        number = n
    }
    deinit {
        print("Apartment #\(number) is «deinitialized»")
    }
}
```

Strong liaison, example

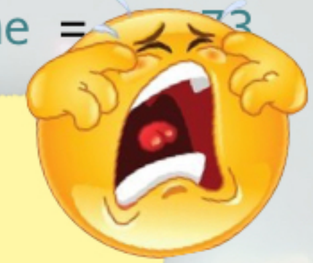

```
class Person {
    let name: String
    var atHome: Apartment?
    init(n: String) {
        name = n
    }
    deinit {
        print("\(name) is «deinitialized»")
    }
}

class Apartment {
    let number: Int
    var owner: Person?
    init(n: Int) {
        number = n
    }
    deinit {
        print("Apartment #\(number) is «deinitialized»")
    }
}
```

```
var him : Person? // value = nil
var apt73 : Apartment? // value =nil
him = Person(n: "Steve J")
apt73 = Apartment(n: 73)
apt73?.owner = him
him?.atHome = apt73

// Let's deallocate
apt73 = nil
him = nil
```

Strong liaison, example


```swift
class Person {
    let name: String
    var atHome: Apartment?
    init(n: String) {
        name = n
    }
    deinit {
        print("\(n
    }
}
```

```swift
var him : Person? // value = nil
var apt73 : Apartment? // value =nil
him = Person(n: "Steve J")
apt73 = Apartment(n: 73)
apt73?.owner = him
him?.atHome =
```



What is going on?

ARC does not catch this memory

```swift
class Apartment {
    let number: Int
    var owner: Person?
    init(n: Int) {
        number = n
    }
    deinit {
        print("Apartment #\(number) is «deinitialized»")
    }
}
```

Close look at the example

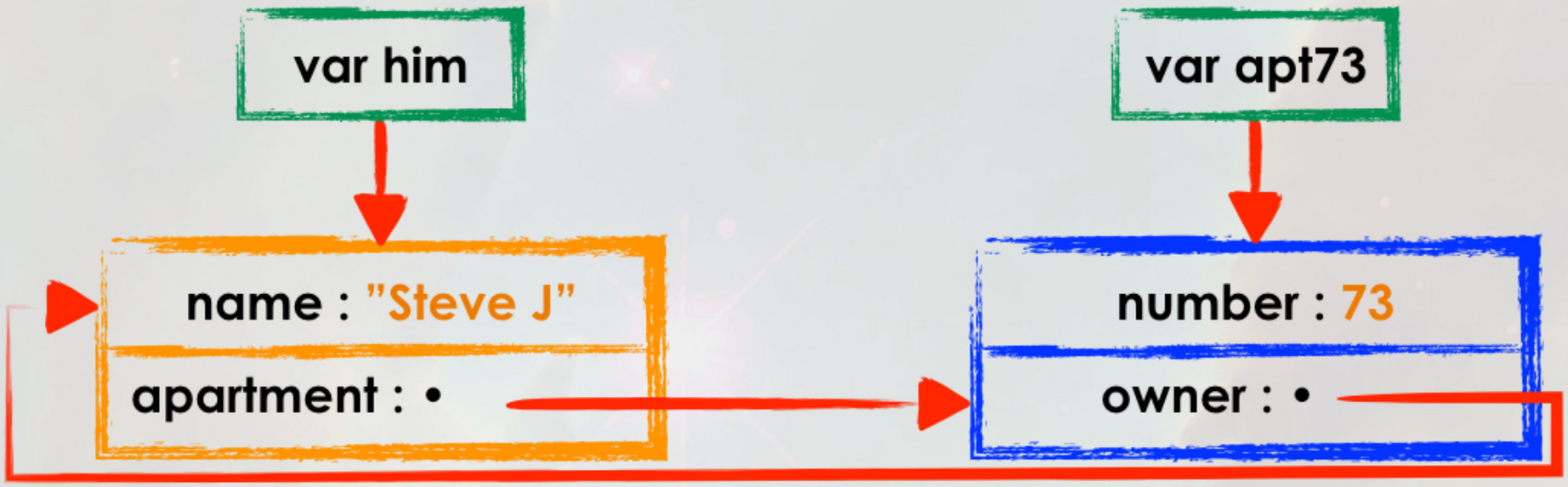

```swift
class Person {
    let name: String
    var atHome: Apartment?
    init(n: String) {
        name = n
    }
    deinit {
        print("\(name) is «deinitialized»")
    }
}
class Apartment {
    let number: Int
    var owner: Person?
    init(n: Int) {
        number = n
    }
    deinit {
        print("Apartment #\(number) is «deinitialized»")
    }
}
```

```swift
var him : Person? // value = nil
var apt73 : Apartment? // value =nil
him = Person(n: "Steve J")
apt73 = Apartment(n: 73)
apt73?.owner = him
him?.atHome = apt73

// Let's deallocate
apt73 = nil
him = nil
```

Close look at the example


```swift
class Person {
    let name: String
    var atHome: Apartment?
    init(n: String) {
        name = n
    }
    deinit {
        print("\(name) is «deinitialized»")
    }
}
class Apartment {
    let number: Int
    var owner: Person?
    init(n: Int) {
        number = n
    }
    deinit {
        print("Apartment #\(number) is «deinitialized»")
    }
}
```

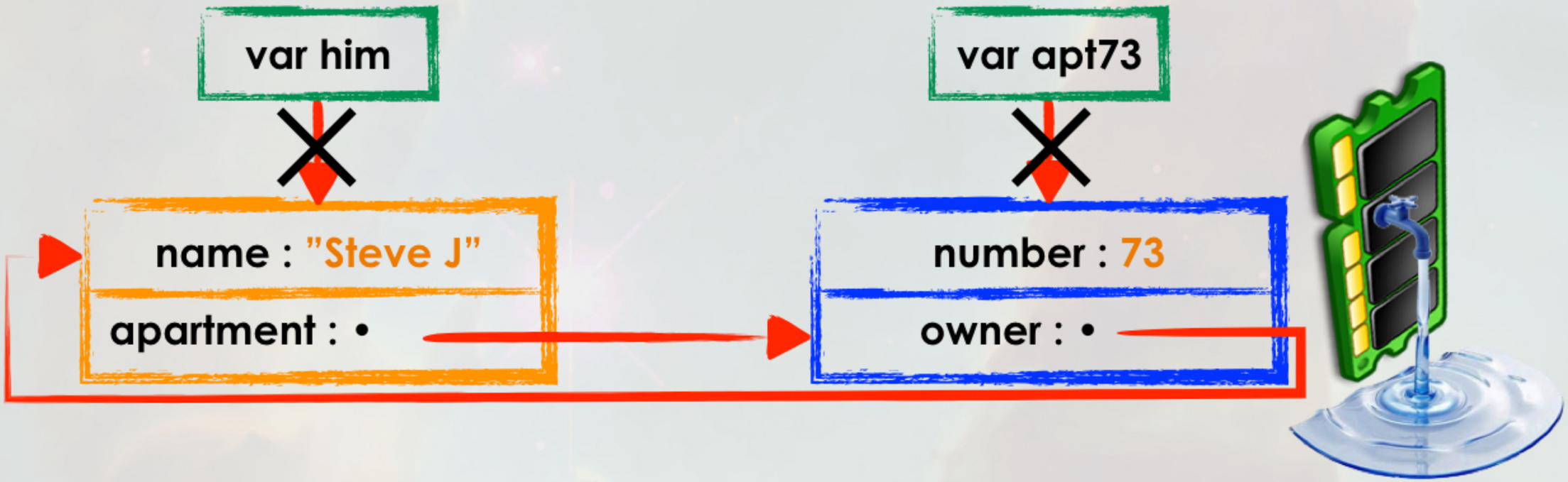
```swift
var him : Person? // value = nil
var apt73 : Apartment? // value =nil
him = Person(n: "Steve J")
apt73 = Apartment(n: 73)
apt73?.owner = him
him?.atHome = apt73

// Let's deallocate
apt73 = nil
him = nil
```



Close look at the example


```swift
class Person {
    let name: String
    var atHome: Apartment?
    init(n: String) {
        name = n
    }
    deinit {
        print("\(name) is «deinitialized»")
    }
}
class Apartment {
    let number: Int
    var owner: Person?
    init(n: Int) {
        number = n
    }
    deinit {
        print("Apartment #\(number) is «deinitialized»")
    }
}
```

```swift
var him : Person? // value = nil
var apt73 : Apartment? // value =nil
him = Person(n: "Steve J")
apt73 = Apartment(n: 73)
apt73?.owner = him
him?.atHome = apt73

// Let's deallocate
apt73 = nil
him = nil
```



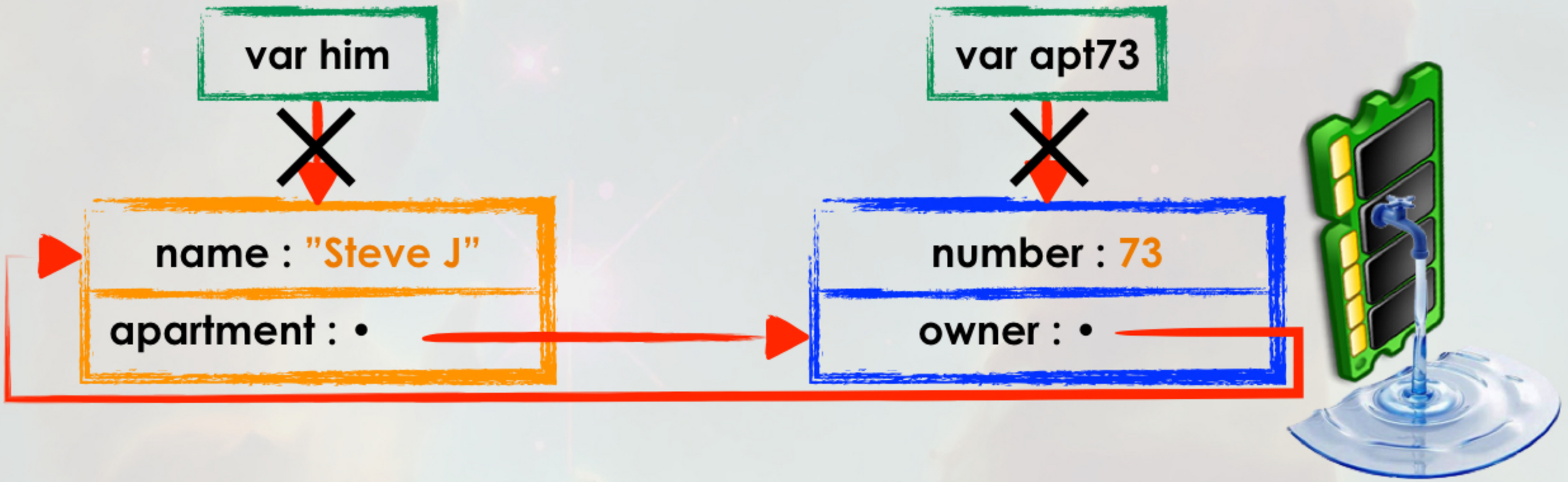
Close look at the example


```swift
class Person {
    let name: String
    var atHome: Apartment?
    init(n: String) {
        name = n
    }
    deinit {
        print("\(name) i
    }
}
class Apartment {
    let number: Int
    var owner: Person?
    init(n: Int) {
        number = n
    }
    deinit {
        print("Apartment #\(number) is «deinitialized»")
    }
}
```

```swift
var him : Person? // value = nil
var apt73 : Apartment? // value =nil
him = Person(n: "Steve J")
apt73 = Apartment(n: 73)
im
t73

ate
```

Cycle with strong liaisons...

ARC cannot deallocate



Breaking a cycle


```swift
class Person {
    let name: String
    var atHome: Apartment?
    init(n: String) {
        name = n
    }
    deinit {
        print("\(name) is «deinitialized»")
    }
}
class Apartment {
    let number: Int
    weak var owner: Person?
    init(n: Int) {
        number = n
    }
    deinit {
        print("Apartment #\(number) is «deinitialized»")
    }
}
```

```swift
var him : Person? // value = nil
var apt73 : Apartment? // value =nil
him = Person(n: "Steve J")
apt73 = Apartment(n: 73)
apt73?.owner = him
him?.atHome = apt73

// Let's deallocate
apt73 = nil
him = nil
```

Breaking a cycle

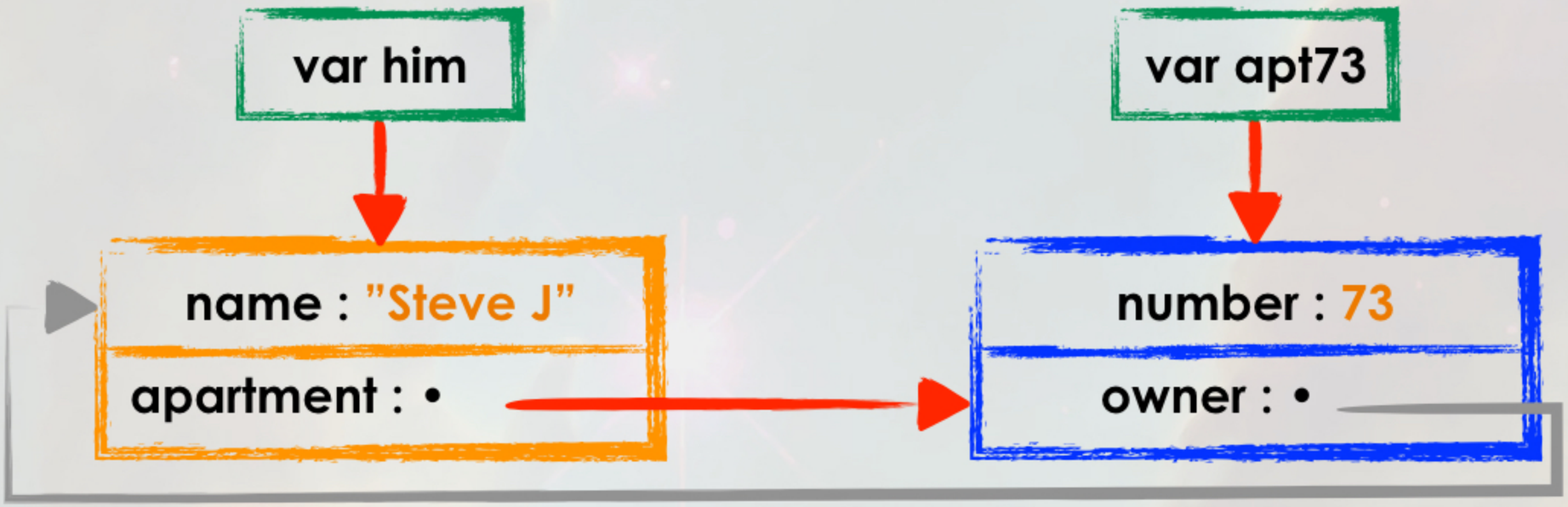

```swift
class Person {
    let name: String
    var atHome: Apartment?
    init(n: String) {
        name = n
    }
    deinit {
        print("\(name) is «deinitialized»")
    }
}
class Apartment {
    let number: Int
    weak var owner: Person?
    init(n: Int) {
        number = n
    }
    deinit {
        print("Apartment #\(number) is «deinitialized»")
    }
}
```

```swift
var him : Person? // value = nil
var apt73 : Apartment? // value =nil
him = Person(n: "Steve J")
apt73 = Apartment(n: 73)
apt73?.owner = him
him?.atHome = apt73

// Let's deallocate
apt73 = nil
him = nil
```

Breaking a cycle


```swift
class Person {
    let name: String
    var atHome: Apartment?
    init(n: String) {
        name = n
    }
    deinit {
        print("\(name) is «deinitialized»")
    }
}
class Apartment {
    let number: Int
    weak var owner: Person?
    init(n: Int) {
        number = n
    }
    deinit {
        print("Apartment #\(number) is «deinitialized»")
    }
}
```

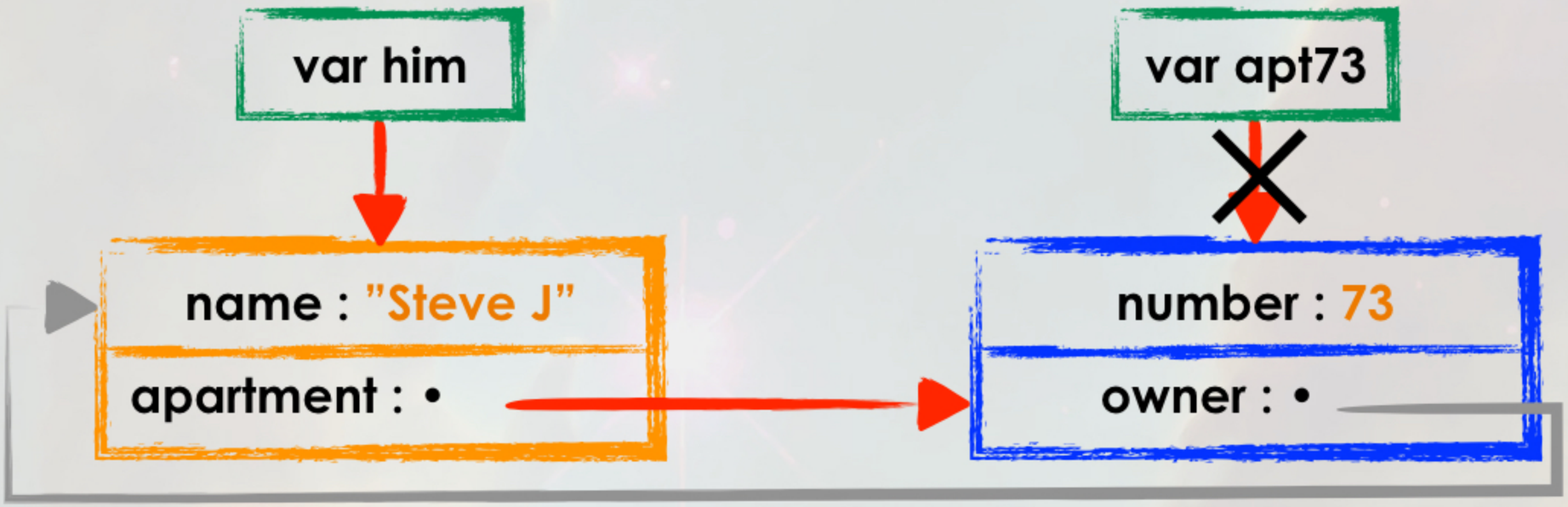
```swift
var him : Person? // value = nil
var apt73 : Apartment? // value =nil
him = Person(n: "Steve J")
apt73 = Apartment(n: 73)
apt73?.owner = him
him?.atHome = apt73

// Let's deallocate
apt73 = nil
him = nil
```



Breaking a cycle


```swift
class Person {
    let name: String
    var atHome: Apartment?
    init(n: String) {
        name = n
    }
    deinit {
        print("\(name) is «deinitialized»")
    }
}
class Apartment {
    let number: Int
    weak var owner: Person?
    init(n: Int) {
        number = n
    }
    deinit {
        print("Apartment #\(number) is «deinitialized»")
    }
}
```

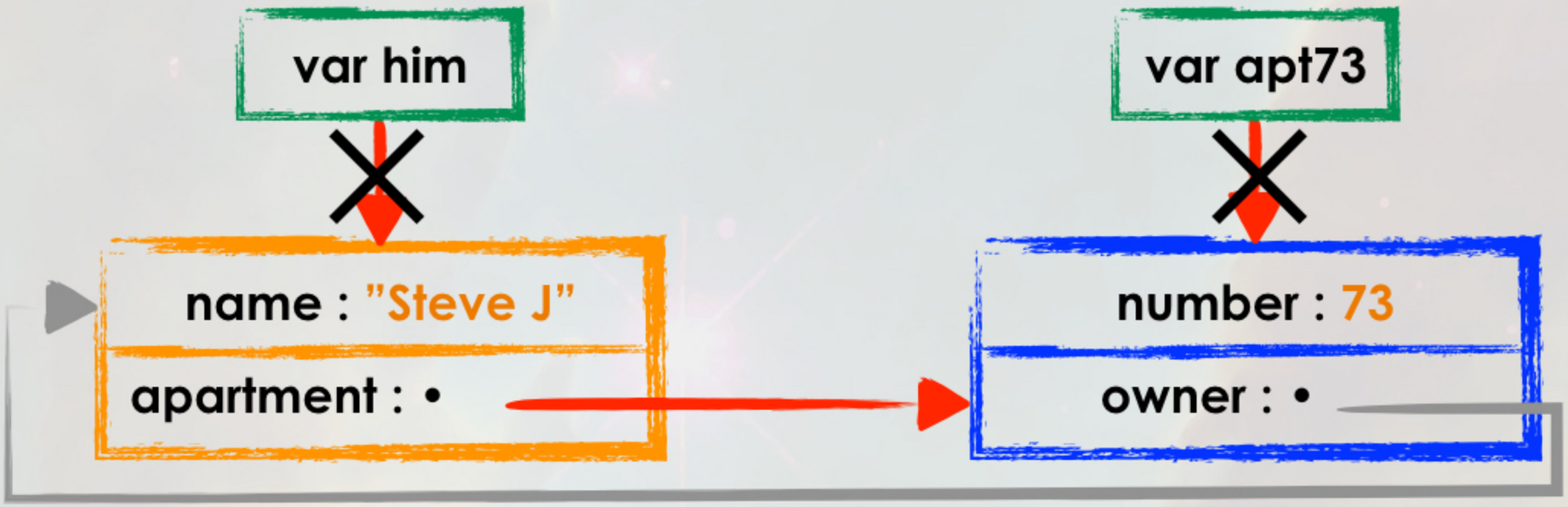
```swift
var him : Person? // value = nil
var apt73 : Apartment? // value =nil
him = Person(n: "Steve J")
apt73 = Apartment(n: 73)
apt73?.owner = him
him?.atHome = apt73

// Let's deallocate
apt73 = nil
him = nil
```



Breaking a cycle


```swift
class Person {
    let name: String
    var atHome: Apartment?
    init(n: String) {
        name = n
    }
    deinit {
        print("\(name) is «deinitialized»")
    }
}
class Apartment {
    let number: Int
    weak var owner: Person?
    init(n: Int) {
        number = n
    }
    deinit {
        print("Apartment #\(number) is «deinitialized»")
    }
}
```

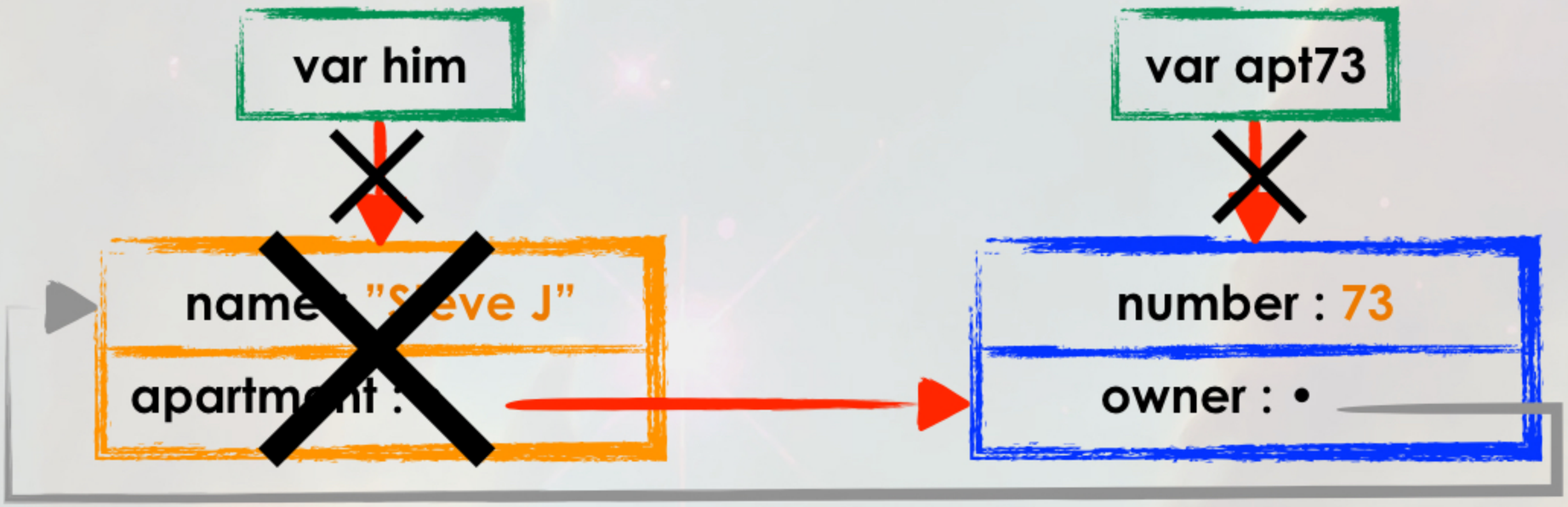
```swift
var him : Person? // value = nil
var apt73 : Apartment? // value =nil
him = Person(n: "Steve J")
apt73 = Apartment(n: 73)
apt73?.owner = him
him?.atHome = apt73

// Let's deallocate
apt73 = nil
him = nil
```



Breaking a cycle


```swift
class Person {
    let name: String
    var atHome: Apartment?
    init(n: String) {
        name = n
    }
    deinit {
        print("\(name) is «deinitialized»")
    }
}
class Apartment {
    let number: Int
    weak var owner: Person?
    init(n: Int) {
        number = n
    }
    deinit {
        print("Apartment #\(number) is «deinitialized»")
    }
}
```

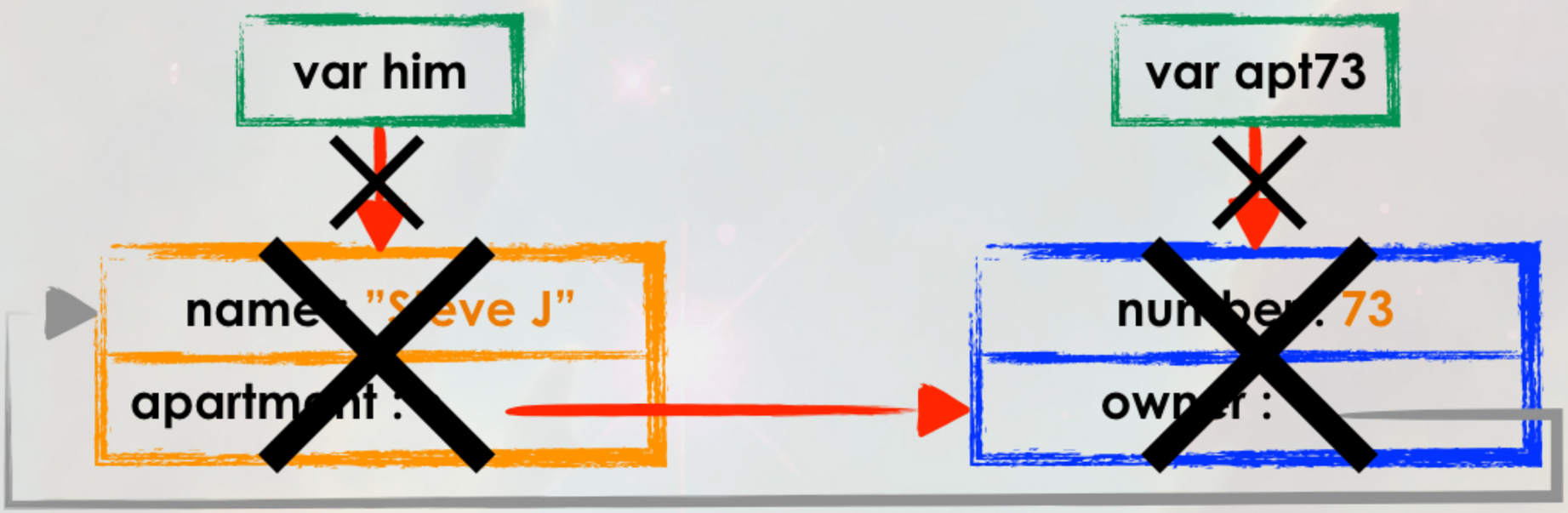
```swift
var him : Person? // value = nil
var apt73 : Apartment? // value =nil
him = Person(n: "Steve J")
apt73 = Apartment(n: 73)
apt73?.owner = him
him?.atHome = apt73

// Let's deallocate
apt73 = nil
him = nil
```



Breaking a cycle


```swift
class Person {
    let name: String
    var atHome: Apartment?
    init(n: String) {
        name = n
    }
    deinit {
        print("\(name) is «deinitialized»")
    }
}
class Apartment {
    let number: Int
    weak var owner: Person?
    init(n: Int) {
        number = n
    }
    deinit {
        print("Apartment #\(number) is «deinitialized»")
    }
}
```

```swift
var him : Person? // value = nil
var apt73 : Apartment? // value =nil
him = Person(n: "Steve J")
apt73 = Apartment(n: 73)
apt73?.owner = him
him?.atHome = apt73

// Let's deallocate
apt73 = nil
him = nil
```



Breaking a cycle


```swift
class Person {
    let name: String
    var atHome: Apartment?
    init(n: String) {
        name = n
    }
    deinit {
        print("\(name) is «deinitialized»")
    }
}
class Apartment {
    let number: Int
    weak var owner: Person?
    init(n: Int) {
        number = n
    }
    deinit {
        print("Apartment #\(number) is «deinitialized»")
    }
}
```

```swift
var him : Person? // value = nil
var apt73 : Apartment? // value =nil
him = Person(n: "Steve J")
apt73 = Apartment(n: 73)
apt73?.owner = him
him?.atHome = apt73

// Let's deallocate
apt73 = nil
him = nil
```


```
Steve J is «deinitialized»
Appartment #73 is «deinitialized»
```

Breaking a cycle


```
class Person {
    let name: String
    var atHome: Apartment?
    init(n: String) {
        name = n
    }
    deinit {
        print("\(nam
    }
}
class Apartment {
    let number: Int
    weak var owner:
    init(n: Int) {
        number = n
    }
    deinit {
        print("Apartment #\(number) is «deinitialized»")
    }
}
```

```
var him : Person? // value = nil
var apt73 : Apartment? // value =nil
him = Person(n: "Steve J")
apt73 = Apartment(n: 73)
apt73?.owner = him
him?.atHome = apt73
```

What if him deallocated first?

Memory of the «orange object» would have been caught first


```
Steve J is «deinitialized»
Appartment #73 is «deinitialized»
```

Weak versus unowned?

- **Unowned behave like weak**
  - Useful to break a cycle
- **Weak liaison**
  - May have no value
- **Unowned liaison**
  - Must always have a value
    - **After initialization**

deinit, more than a deallocator

At most one deinit per class

- Goal : appropriately «terminate» the object

Example

- Monopoly game
  - **Each player has a portfolio**
  - **when a player loose, its portfolio goes to the bank**

deinit, more than a deallocator

At most one deinit per class

- Goal : appropriately «terminate» the object

Example

- Monopoly game
  - **Each player has a portfolio**
  - **when a player loose, its portfolio goes to the bank**

Anytime in the game

$$portfolio(bank) + \sum_{p \in Players} portfoliot(p) = K$$

deinit, more than a deallocator


```
class Bank {

    func getMoney(val: Int) -> Int {...}
    func putMoney(val: Int) {...}

}

class Gamer {
    var theBank : Bank?
    var inPortfolio: Int // my money

    init(value: Int, myBank : Bank) {
        theBank = myBank
        inPortfolio = theBank!.getMoney(val : value)
    }
    func gangnerArgent(value: Int) {
        inPortfolio += theBank!.getMoney(val : value)
    }
    deinit {
        theBank?.putMoney(val : inPortfolio) // Return my money to the bank
    }
}
```

deinit, more than a deallocator

A way to preserve consistency

More powerful than the Objective-C dealloc

```
class Bank {

    func getMoney(val: Int) -> Int {...}
    func putMoney(val: Int) {...}
}

class Gamer {
    var theBank : Bank?
    var inPortfolio: Int // my money

    init(value: Int, myBank : Bank) {
        theBank = myBank
        inPortfolio = theBank!.getMoney(val : value)
    }
    func gangnerArgent(value: Int) {
        inPortfolio += theBank!.getMoney(val : value)
    }
    deinit {
        theBank?.putMoney(val : inPortfolio) // Return my money to the bank
    }
}
```

As a conclusion…

You might find retain/release/autorelease easy

- Because ARC is no «totally free»
  - **You still need to think a bit**

Obviously, ARC need some data to run

- Objective-C rely on conventions
  - **alloc/init/copy & counter's value**
  - **autorelease**
  - **etc.**
- Swift strongly relies on a classification of liaisons

By the way

- Remind that optional variables default is «nil»

As a conclusion…

You might find retain/release/autorelease easy

- Because ARC is no «totally free»
  - **You still need to think a bit**

Obviously, ARC need some data to run

- Objective-C rely on conventions
  - **alloc/init/copy & counter's value**
  - **autorelease**
  - **etc.**
- Swift strongly relies on a classifi s

By the way

- Remind that optional lt is «nil»