ITM103 iOS Application Development

2016 S1

Topic 2: Introduction to Swift - Part 2

Objectives

- By the end of the lesson, you will be able to:
  - Understand how to manage memory with ARC
  - Understand more advanced languages features such as Protocols, Delegates, Closures

Introduction to Swift Part 2

Optionals

Optionals

- All non-optionals declared must be initialised. The following is **not allowed**.

```
class MyPoint : NSObject
{
    var x: Int
    var y: Int

    override init()
    {


    }
}
```

❌

Since x and y are not optionals, Swift will run into an error if you do not initialize them with a value in your init method.

Optionals

- All non-optionals declared must be initialised inside **init**. The following is correct.

```
class MyPoint : NSObject
{
    var x: Int
    var y: Int

    override init()
    {
        x = 0
        y = 0
    }
}
```

✓

Optionals

- All non-optionals can also be declared this way

```
class MyPoint : NSObject
{
    var x: Int = 0
    var y: Int = 0

    override init()
    {


    }
}
```

✓

Optionals

- Optionals are fields that may be **nil** in value. They do **not** need to be initialised in **init**.

```
class MyRect
{
    var x: Int?
    var y: Int?
}
```

- It is similar to C#'s nullable types, except, in Swift:
  - Int, Doubles, etc can be optional / non-optional.
  - Class types can be optional / non-optional.

Optionals

- To use the values stored in optional vars, we must **unwrap** with ‘!’ it to retrieve its value.

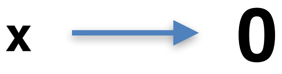
```
class MyRect
{
    var x: Int?
    var y: Int?

    func computeArea() -> Int
    {
        return x! * y!
    }
}
```

Optionals

- Visualize this:

```
var x: Int
x = 0
```

A standard variable contains a value. But you cannot have an empty variable.



Optionals

- Visualize this:

```
var x: Int
x = 0
```

A standard variable contains a value. But you cannot have an empty variable.

x → **0**

```
print(x)
```

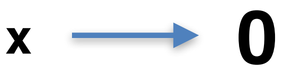
When we access the variable, the value is retrieved directly.

Optionals

- Visualize this:

```swift
var x: Int
x = 0
```

A standard variable contains a value. But you cannot have an empty variable.


x → 0

```swift
print(x)
```

When we access the variable, the value is retrieved directly.

0

In this case, print (x) takes the value 0 from the variable x and prints it to the screen.

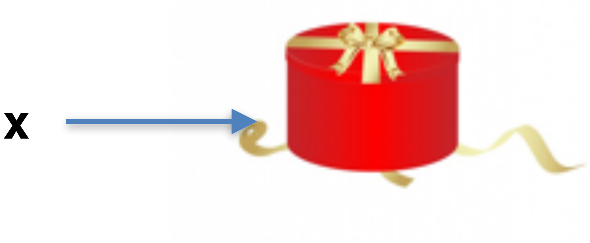
Optionals

- Visualize this:

```
var x: Int?
x = 0
```

```
var x: Int?
x = nil
```

an optional variable is like a gift. So you can either have a gift box,



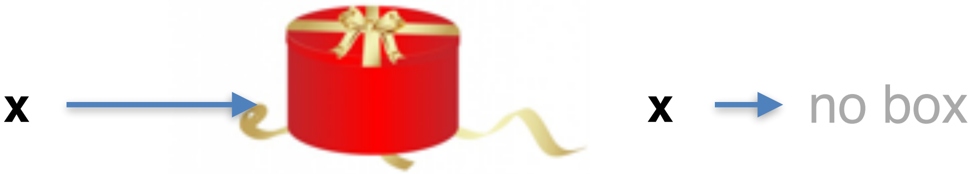
Optionals

- Visualize this:

```
var x: Int?
x = 0
```

```
var x: Int?
x = nil
```

an optional variable is like a gift. So you can either have a gift box, or there's no box.



x →

x → no box

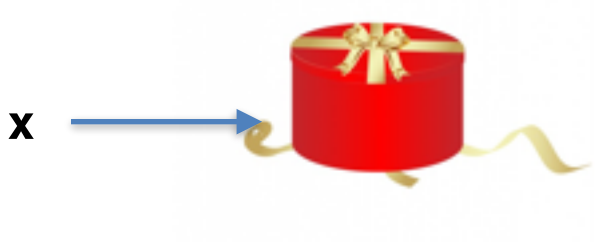
Optionals

- Visualize this:

```
var x: Int?
x = 0
```

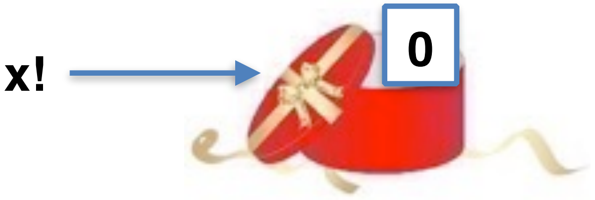
```
var x: Int?
x = nil
```

an optional variable is like a gift. So you can either have a gift box, or there's no box.



x → no box

x! unwraps the gift box to see what's inside.



Optionals

- Visualize this:

```
var x: Int?
x = 0
```

```
var x: Int?
x = nil
```

an optional variable is like a gift. So you can either have a gift box, or there's no box.

x →

x → no box

x! unwraps the gift box to see what's inside.

x! →

x! → can't unwrap

Optionals

- Visualize this:

```
var x: Int?
x = 0
```

```
var x: Int?
x = nil
```

an optional variable is like a gift. So you can either have a gift box, or there's no box.

x →

x → no box

x! unwraps the gift box to see what's inside.

x! →

x! → can't unwrap

If you do a:

```
  print(x)
```

0

Optionals

- Visualize this:

```
var x: Int?
x = 0
```

```
var x: Int?
x = nil
```

an optional variable is like a gift. So you can either have a gift box, or there's no box.

x →

x → no box

x! unwraps the gift box to see what's inside.

x! →

x! → can't unwrap

If you do a:

```
  print(x)
```

0

crashes

Optionals

- It is recommended to check for nil values before unwrapping. Otherwise, your program will crash.

```
class MyRect
{
    var x: Int?
    var y: Int?

    func computeArea() -> Int
    {
        if x == nil || y == nil
        {
            return 0
        }
        return x! * y!
    }
}
```

Safety checks for nil

Optionals

- Swift-style shorthand for checking for nil and unwrapping values simultaneously:

```
class MyRect
{
    var x: Int?
    var y: Int?

    func computeArea() -> Int
    {
        if let unwrappedX = x
        {
            if let unwrappedY = y
            {
                return unwrappedX * unwrappedY
            }
        }
        return 0
    }
}
```

If not nil, then unwrap the value to a constant, and enter the if block.

Optionals

- To destroy objects, you need to remove all references to it.

```
var s1 : MyRect? = nil
var s2 : MyRect? = nil

s1 = MyRect()
s2 = s1

s1 = nil
s2 = nil
```

- Managed with Automatic Reference Counting

More on ARC in the next section

Optionals

- **Implicitly Unwrapped Optionals**
  - Declare with type! instead of type?

```
class MyRect
{
    var x: Int!
    var y: Int!

    func computeArea() -> Int
    {
        if x == nil || y == nil
        {
            return 0
        }
        return x * y
    }
}
```

Implicitly unwrapped optional

Safety checks for nil

No need to unwrap using ! when accessing value.

It's like the box is always unwrapped and open!

Optionals

- **Implicitly Unwrapped Optionals**

In Swift 3.0, except for @IBOutlets

the use of standard optionals is **preferred**

the one with the ‘?’

Optionals

- More on Optionals

  https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html

Optional Chaining

- Allows you to use ‘.’ notation to chain without worrying about nils.

- Traditional way:

```
var s = Student("John", "123456D")
if s.teacher != nil && s.teacher!.school != nil
{
    print (“Teacher's school’s name = 
        \(s.teacher!.school!.schoolName)")
}
```

Optional Chaining

- Allows you to use ‘.’ notation to chain without worrying about nils.

- Swift way:

```
var s = Student("John", "123456D")
if let schoolName = s.teacher?.school?.name
{
    print (“Teacher's school’s name =
           \(schoolName)")
}
```

Optional Chaining

- More on Optional Chaining
  https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/OptionalChaining.html

Introduction to Swift Part 2

Memory Management

Memory Management

- Swift uses:
  - **Automatic Reference Counting (ARC)**
  - Free objects with 0 reference counts
  - Freeing of memory occurs immediately
  - Does not free objects in circular reference

- In contrast, C#, Java uses:
  - Garbage Collection - mark and sweep objects no accessible by main program at a non-deterministic time
  - Handles circular references

Memory Management

- What is Automatic Reference Counting (ARC)?
  - Object referenced: **increment** counter by 1
  - Object de-referenced: **decrement** counter by 1
  - When counter = 0: **free memory**

- Problem of circular references.

Memory Management

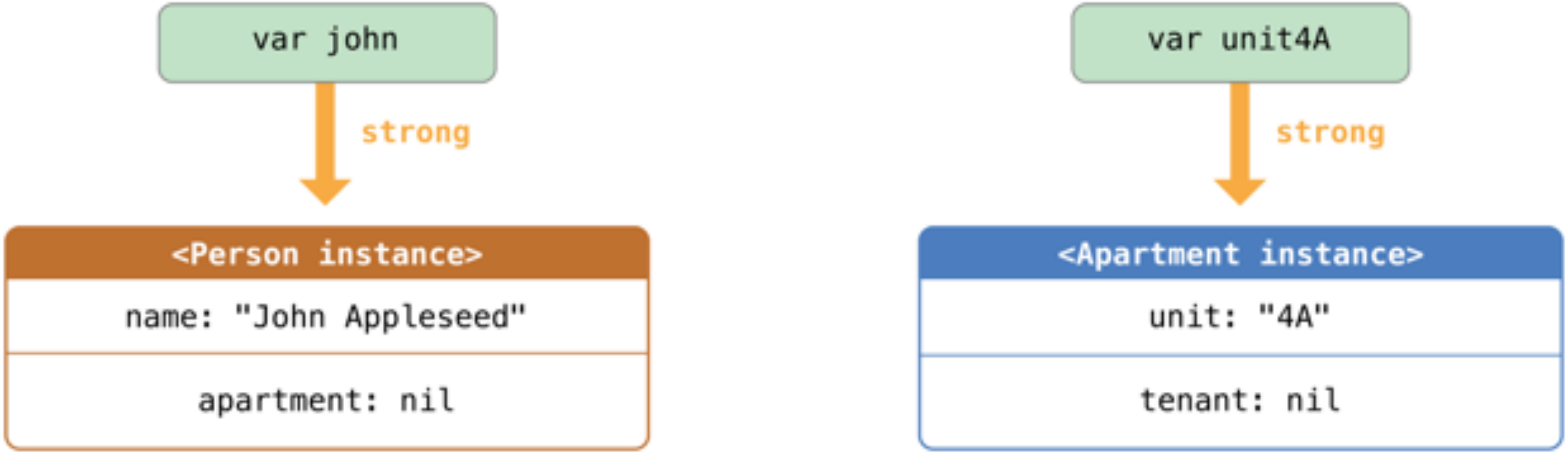
- As an example:

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


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

Memory Management

- Normal reference of objects:

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



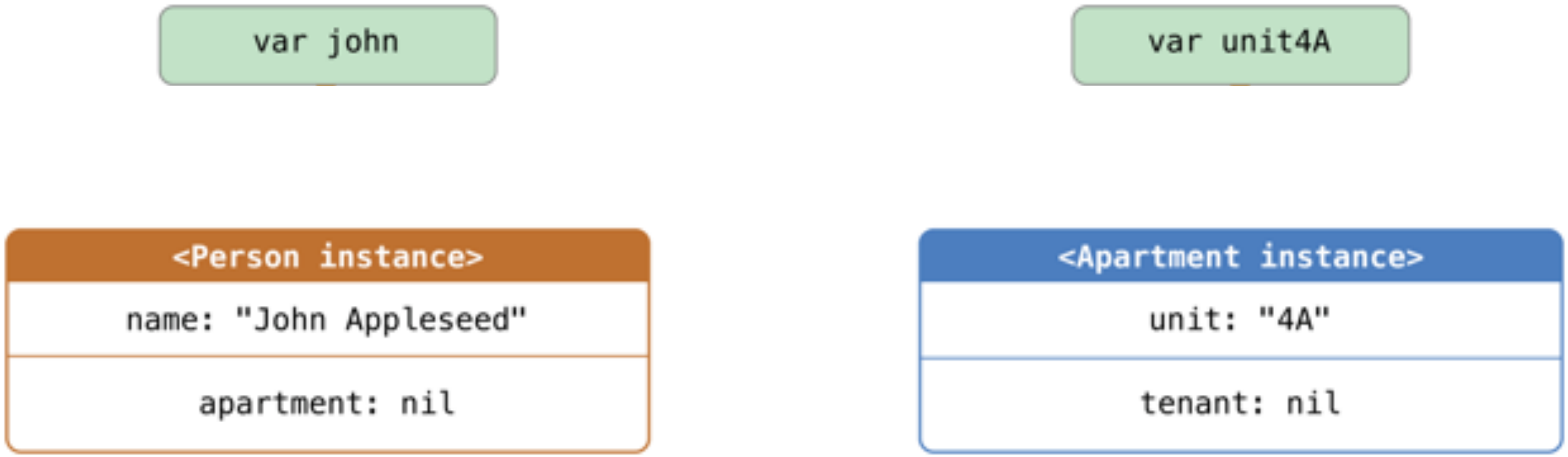
reference count = 1

reference count = 1

Memory Management

- Normal reference of objects:

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



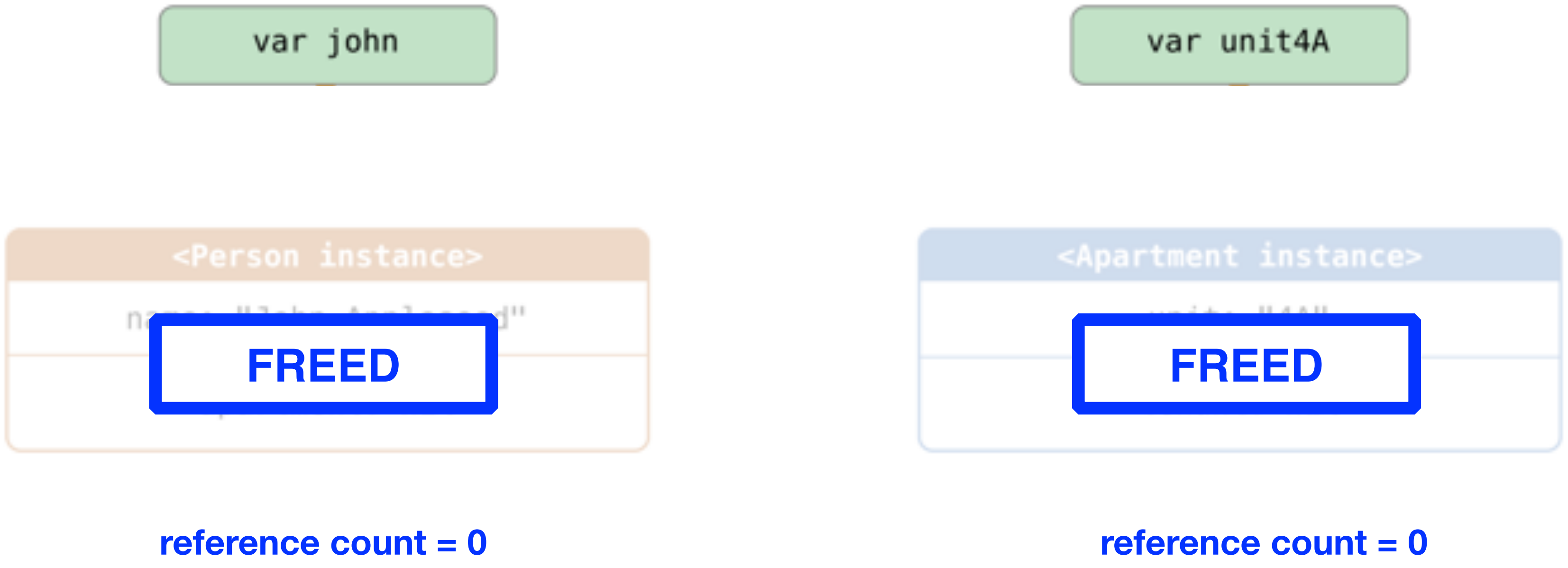
reference count = 0

reference count = 0

Memory Management

- Normal reference of objects:

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



var john

var unit4A

<Person instance>

FREED

<Apartment instance>

FREED

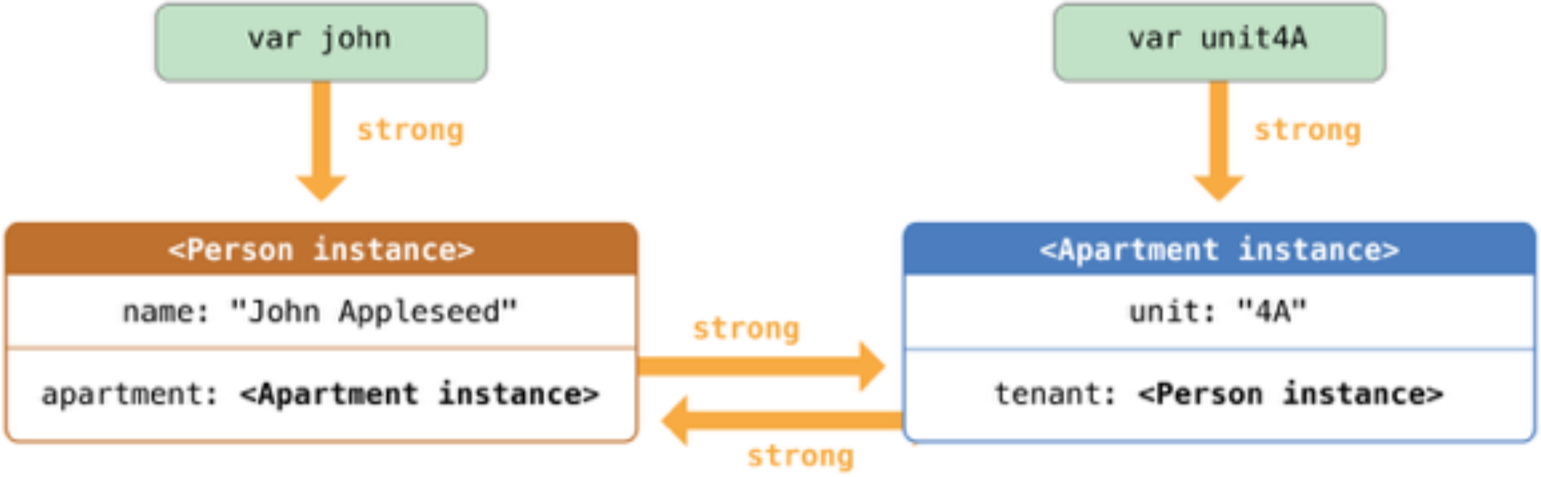
reference count = 0

reference count = 0

Memory Management

- Circular reference of objects:

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

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



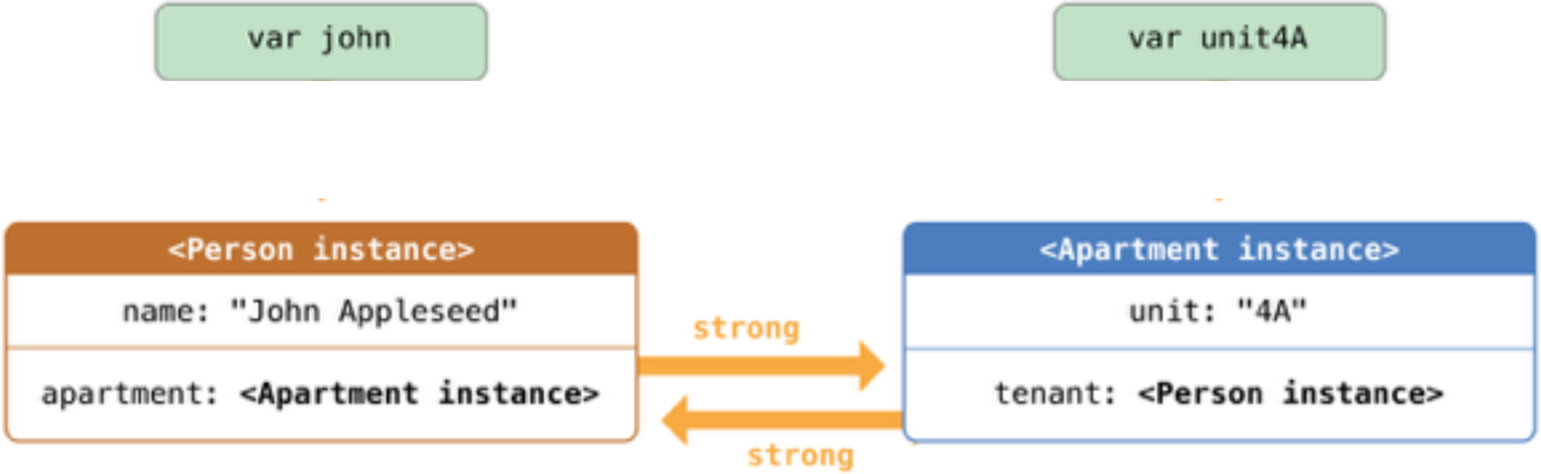
reference count = 2

reference count = 2

Memory Management

- Circular reference of objects:

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



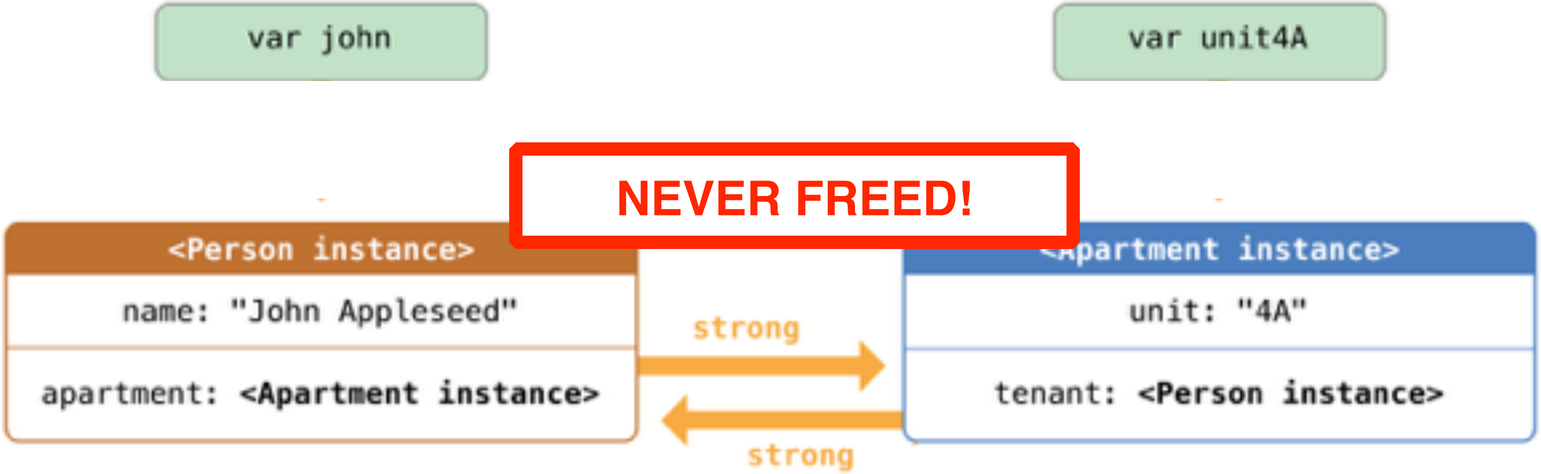
reference count = 1

reference count = 1

Memory Management

- Circular reference of objects:

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



NEVER FREED!

reference count = 1

reference count = 1

Memory Management

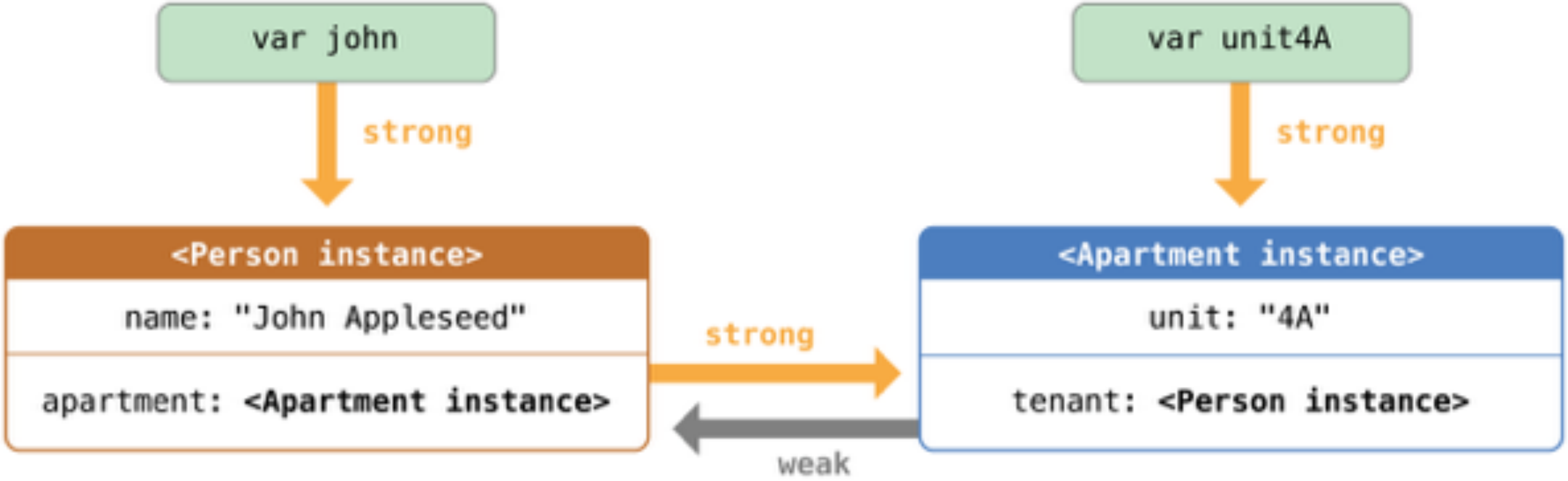
- To resolve the problem:

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

class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    weak var tenant: Person?
}
```

Add a ‘**weak**’ keyword here

Memory Management

- Strength of reference:
  - **Strong** reference: +1 reference count
  - **Weak** reference: no change in reference count



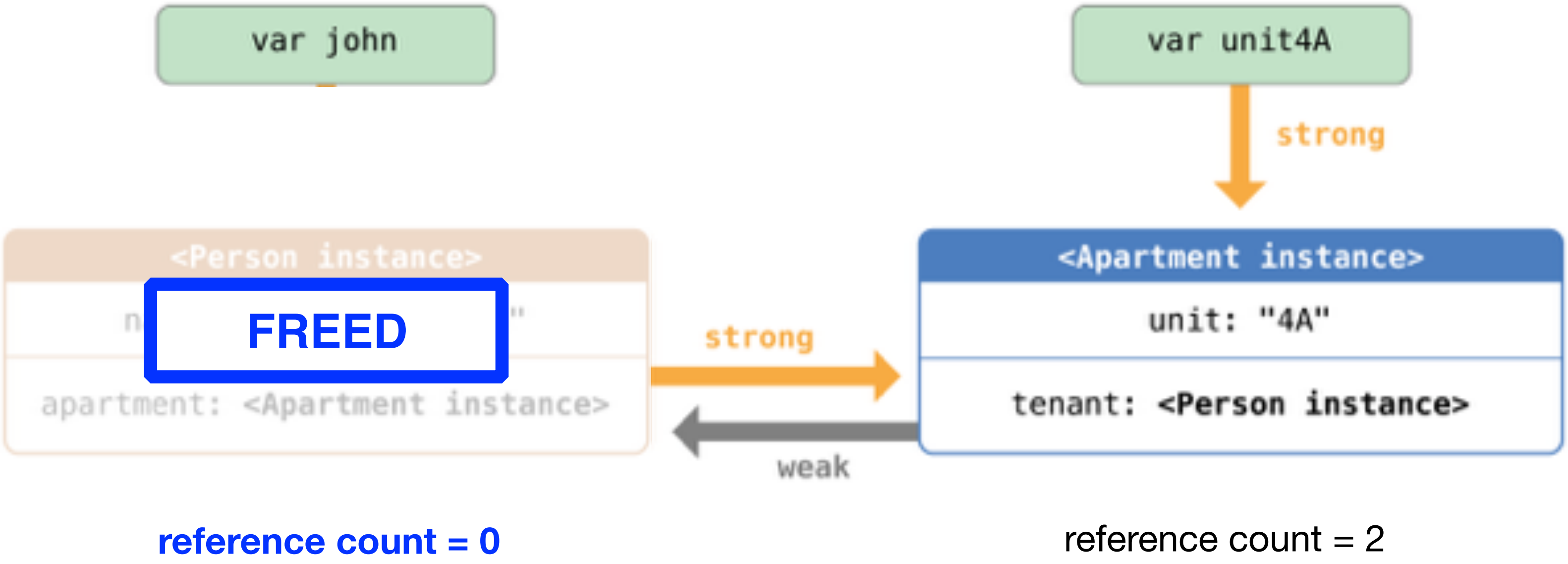
reference count = 1

reference count = 2

Memory Management

- Effect of using **strong**/**weak** references:

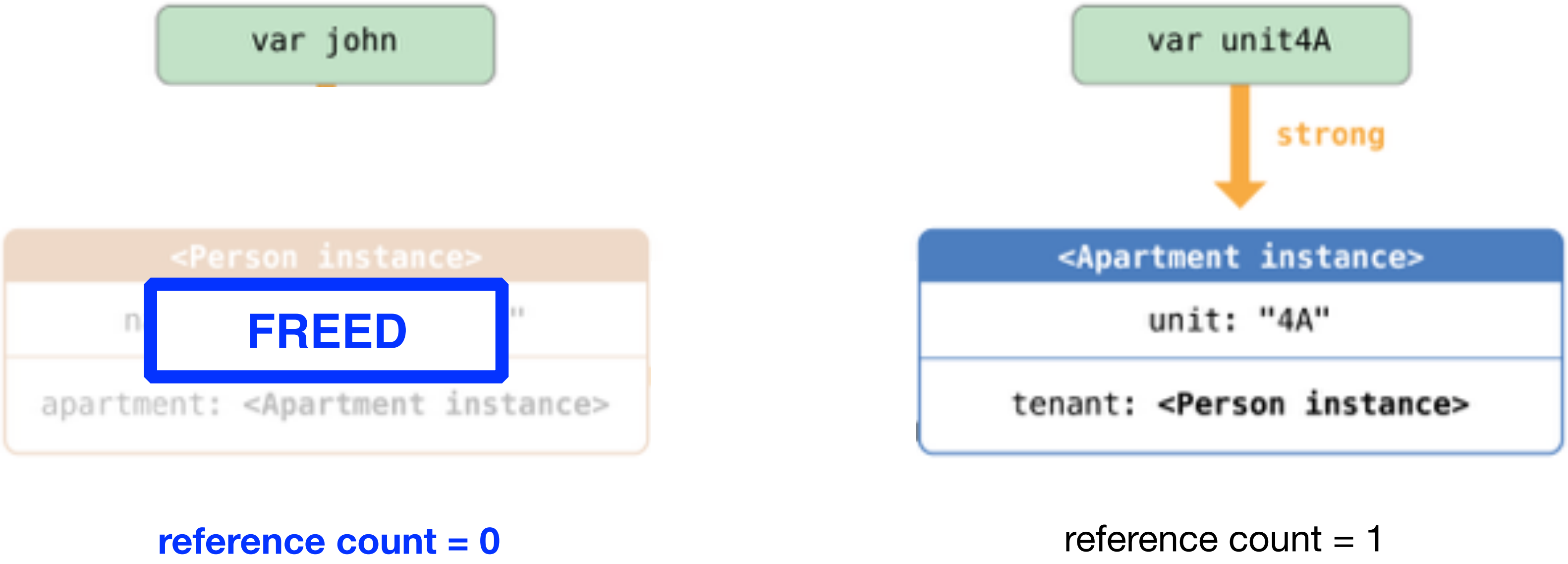
```
john = nil
```



Memory Management

- Effect of using **strong**/**weak** references:

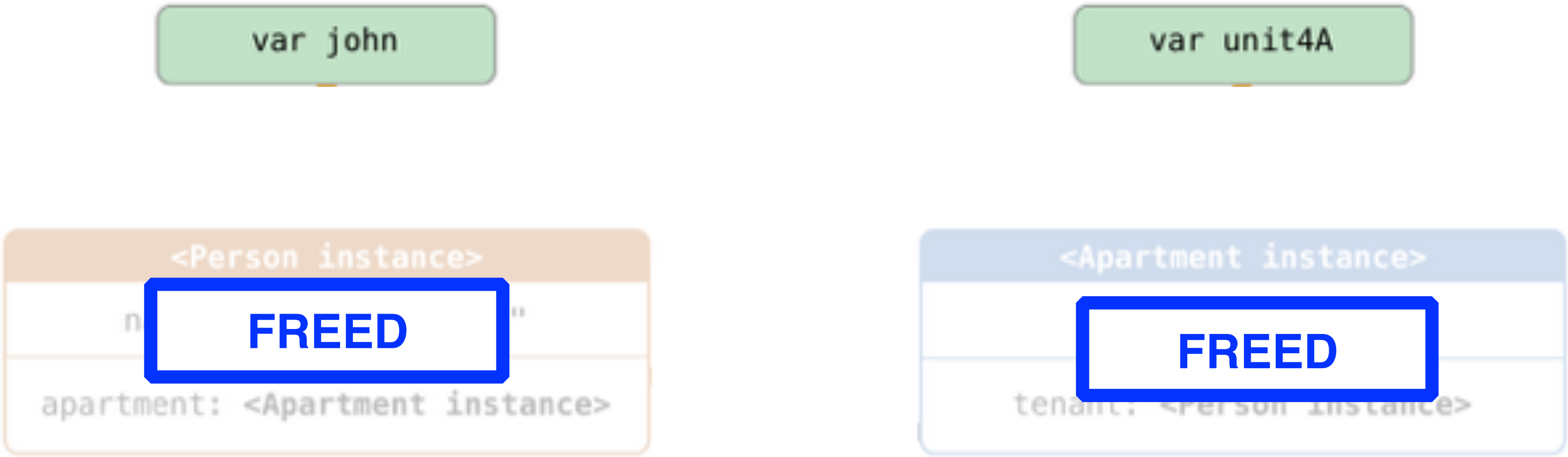
```
john = nil
```



Memory Management

- Effect of using **strong**/**weak** references:

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



var john

var unit4A

<Person instance>

FREED

apartment: <Apartment instance>

<Apartment instance>

FREED

tenant: <Person instance>

reference count = 0

reference count = 0

Memory Management

- Strength of reference - 3rd type:
  - **Strong** reference: +1 reference count
  - **Weak** reference: no change in reference count (optional type)
  - **Unowned** reference: no change in reference count (non-optional type)

Memory Management

- More on Automatic Reference Counting:
    - https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/AutomaticReferenceCounting.html

Introduction to Swift Part 2

Protocols and Delegates

Protocols and Delegates

- **Protocol**
  - A blueprint of functions / properties that suit a particular task; it attaches a blueprint of capabilities to a class.
  - **None** of the functions / properties are implemented when a protocol is defined.
  - One class can implement **multiple** protocols
  - Similar to C# / Java’s interface

Protocols and Delegates

- Example:

```
protocol CanSparkle
{
    var sparkleFrequency : Int { get set }

    func drawSparkles()
}
```

CanSparkle is a protocol.

This is a **gettable**/**settable** property

This is a **function**.

Protocols and Delegates

- Example:

```
protocol CanSparkle
{
    var sparkleFrequency : Int { get set }

    func drawSparkles()
}

class Star : Shape, CanSparkle
{
    var mySparkleFrequency : Int = 0
    var sparkleFrequency : Int {
        get { return mySparkleFrequency }
        set { mySparkleFrequency = newValue }
    }

    func drawSparkles() {
        // Draw the sparkles on screen
    }
}
```

Since Star implements the CanSparkle protocol, it must implement **all** its properties / functions.

sparkleFrequency

drawSparkles

Protocols and Delegates

- **Delegates**
  - A design pattern that allows a class / structure to hand off some of its responsibilities to another class type.
  - It can used to:
    - Respond to an action
    - Retrieve data from external source

Protocols and Delegates

- Example:

```
protocol BrushDelegate {
    func getColor() -> String
    func getBrushType() -> String
}

class Star : Shape, BrushDelegate {
    var brush : Brush

    override init() {
        brush = Brush()
        super.init()
        brush.delegate = self
    }

    func getColor() -> String {
        return "#FFFFFF"
    }
    func getBrushType() -> String {
        return "Solid"
    }
}
```

In this case, BrushDelegate is a set of responsibilities that the class (that has this protocol) must implement

The Brush object needs to retrieve more information at some point in time, and the Star object is responsible for providing that information.

Protocols and Delegates

- In Apple's UIKit, you will find this pattern occurring very often

- More on Protocols and Delegates

  https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Protocols.html

Introduction to Swift Part 2

Closures

Closures

- **Closures**
  - Self-contained blocks of functionality that can be passed around in code.

  - Similar to lambdas in C#.

Closures

- Example:
  - Swift provides a **sort** function for arrays.
  - The sort function expects a special type of parameter.
  - This parameter must be a function of the following declaration.

    ```
    func functionName(v1: Type, _ v2: Type) -> Bool
    ```

  - The sort function uses your function to determine whether v2 is should appear *after* v1, and your function should return true if so.

Closures

- Closure version
  - A closure is a function without a name:

```
{
    (p1: Type, p2: Type, ...) -> ReturnType in

    /* function body */
}
```

  - It can be declared *directly* in a parameter that expects a function.

Closures

- Closure version
  - Without creating the ascending and descending functions:

```
var list = [5,78,20,3,15]

list = list.sort(
{
    (s1: Int, s2: Int) -> Bool in
    return s1 > s2
}
)
print (list)
// outputs: [78, 20, 15, 5, 3]

list = list.sort(
{
    (s1: Int, s2: Int) -> Bool in
    return s2 > s1
}
)
print (list)
// outputs: [3, 5, 16, 20, 78]
```

Closures

- Closure version
  - Without creating the ascending and descending functions:

```
var list = [5,78,20,3,15]

list = list.sort(
{
    (s1: Int, s2: Int) -> Bool in
    return s1 > s2
}
)
print (list)
// outputs: [78, 20, 15, 5, 3]

list = list.sort(
{
    (s1: Int, s2: Int) -> Bool in
    return s2 > s1
}
)
print (list)
// outputs: [3, 5, 16, 20, 78]
```

These are the closures we create to pass into the sort function.

Closures

- Inferring of Types
  - Even shorter version:

```
var list = [5,78,20,3,15]

list = list.sort({(s1, s2) -> Bool in s1 > s2 })
print (list)
// outputs: [78, 20, 15, 5, 3]

list = list.sort({(s1, s2) -> Bool in s2 > s1 })
print (list)
// outputs: [3, 5, 16, 20, 78]
```

list is a array of integer, so the types inferred by Swift.

The line becomes even shorter.

Swift knows you are returning a Bool. And since s1 > s2 and s2 > s1 is a Bool, Swift automatically infers the return keyword.

Closures

- Declaring a Function Accepting Closure for Asynchronous Methods

```
func updateServerAsync(
    postData: [String: String], onComplete: (() -> Void)?)
{
    // Do some asynchronous work
    //
    if onComplete != nil
    {
        onComplete!()
    }
}
```

onComplete is a optional closure / function that accepts no parameter, and returns no value.

if **onComplete** is not null, then call that closure.

```
updateServerAsync (["name": "Tan", "email": "tan@gmail.com"],
    onComplete:
    { () -> Void in
        // Do some work to update UI
        //
    }
)
```

When the update is complete, execute this closure

Closures

- More on Closures:
  https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Closures.html

Introduction to Swift Part 2

Error Handling

Error Handling

- Methods can be declared to ‘throw’ an error

```
func CheckEnglish(text: String) throws
```

This function **may** throw an error

```
func PrintText()
```

This function will **never** throw an error

Error Handling

- Methods can then throw an error

```
enum EnglishError: Error {
    case SpellingError
    case GrammaticalError
}

func CheckEnglish(text: String) throws
{
    // ... some other code ...

    throw EnglishError.GrammaticalError
}
```

Error Handling

- Do-Try-Catch Pattern:

```
do
{
    try CheckEnglish(str)
    statements
}
catch EnglishError.GrammaticalError {
    statements
}
catch EnglishError.SpellingError {
    statements
}
```

- When calling a function that throws, you **must** ‘try’.

Error Handling

- You can **try** without **catch**ing:

```
func buyFavoriteSnack(
    vendingMachine: VendingMachine) throws
{
    ...
    try vendingMachine.vend(itemNamed: snackName)
}
```

Any error encountered will be thrown out.

buyFavoriteSnack vendingMachine.vend

Error Handling

- Or this way:

```
 func buyFavoriteSnack(
      vendingMachine: VendingMachine)
{
      ...
      try? vendingMachine.vend(itemNamed: snackName)
}
```

Any error encountered will be **suppressed and ignored**.

Error Handling

BEWARE

Unwrapping a nil **cannot be caught** using do-try-catch!

It will terminate your program!

(unlike languages like C# / Java)

Summary

- Optionals and Optional Chaining
- Memory Management (ARC)
- Protocols and Delegates
- Closures
- Error Handling