Lecture-6

- View Controller life cycle

  Which methods get called during the View Controller life cycle?
- Memory management

  How the memory management is handled in Swift?

  How we can influence this management?
- Retain Cycle

  What this cycle is about?

  How can we break this cycle?
- Demo(Closure)

  How the closures can create a retain cycle?

View Controller life cycle

- View Controller has a 'life cycle'

  Sequence of methods that get called at some certain time through out his life cycle
- Why is that important?

  In order to do some certain work you must override some methods in View Controller life cycle
- The start of the life cycle

  Creation of the ViewController.

  Most of the time it's created in storyboard.

  Very rarely created in code

View Controller life cycle

- Preparation

  This happens if VC is being segued to. Otherwise, it goes with outlet setting after creation
- Outlets get set
- Appearing and disappearing

  This can happen multiple times
- Geometry changes

  This happens whenever our device is changing it's orientation
- Low memory situations

  Almost never happens. If it did, it means your app is consuming a lot memory and you might want to free up some

View Controller life cycle

- After outlets setting, viewDidLoad is called

  This is a good place to put all of your set-up code.

```
override func viewDidLoad() {
    super.viewDidLoad()
    // do some set up of your MVC
}
```

One thing you may want to do here is updating your UI, because you are 100% sure that your outlets are set.

DO NOT do here anything related to your geometry. Because you are not sure at this point, what screen size will be established here.

View Controller life cycle

- Right before your view appears on screen, `viewWillAppear` is called

```
override func viewWillAppear(_ animated: Bool) {
    super.viewWillAppear(true)
    // put something here
}
```

This method gets called a lot over time. So DO NOT put here something that should be on viewDidLoad. Otherwise you will be doing unnecessary things

Do something here if things you display is changing, while your screen is OFF. You can also start something expensive in this method, like starting a new thread

View Controller life cycle

- After your view is on screen, `viewDidAppear` is called

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(true)
  // put something here
}
```

Here is not so much to do, but it's may be the good place to start your animation though. Because you know that you're fully on screen, to be interacted with something.

View Controller life cycle

- You also get notified when you will disappear

```
override func viewWillDisAppear(_ animated: Bool) {
    viewWillDisAppear(true)
  // do some clean up code here
  // do not do something time-consuming
}
```

- 'did' version of disappearing

```
override func viewDidDisappear(_ animated: Bool) {
    viewDidDisappear(true)
  // usually you undo the things in viewWillAppear
}
```

View Controller life cycle

- Geometry changes?

  Most of the time this is handled by AutoLayout

  But you can get involved with these methods

```
override func viewWillLayoutSubviews()
override func viewDidLayoutSubviews()
```

This methods get called whenever your bounds change, and your view's subviews must be layed-out

AutoLayout happens in between this two methods.

This can happen repeatedly. Not only because of due to the rotation. Even if your bounds don't change, this methods can be called.

View Controller life cycle

- Autorotation

  User can change the orientation switching it to portrait or landscape.

  If you want to participate in autorotation...

```
func viewWillTransition(
        to size: CGSize,
        with coordinator: UIViewControllerTransitionCoordinator
        )
```

Size is the new size for the container's view.

The transition coordinator object managing the size change. You can use this object to animate your changes.

View Controller life cycle

- In low-memory cases, you get notified also ...

```
override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
  // the things that can be recreated should be released here
  // just set your big objects in the heap to be nil
  // this is rarely happens, because iPhones have a lot of memory this days
}
```

View Controller life cycle

- `awakeFromNib`

  This method is sent to all of the objects that come out of the storyboard.

  Happens before viewDidLoad, before outlets are set, even before preparation

View Controller life cycle

- Overview
  - Creation(usually from the storyboard)
  - awakeFromNib
  - segue preparation
  - outlets are set
  - viewDidLoad
    - appearing and disappearing
    - geometry changes
    - didReceiveMemoryWarning

Demo

Memory Management

- Automatic Reference Counting

  Reference types are stored in the heap.

  ARC count the pointers to them and if the counter goes to 0, it automatically removes from the heap.

  You don't bother about this, because it happens automatically.

  It is not a garbage collector

- Influencing ARC

  strong

  weak

  unowned

Memory Management

- strong

  is normal reference counting

  Makes the instance be in the heap, until no-one points to it

- weak

  if no-one else is interested in this, set me to nil

  works only with Optionals

  Great example is outlet(strongly held by the view hierarchy, so outlet can be weak)

- unowned

  do not reference count this

  Rarely ever used

  Usually used to break a memory cycles

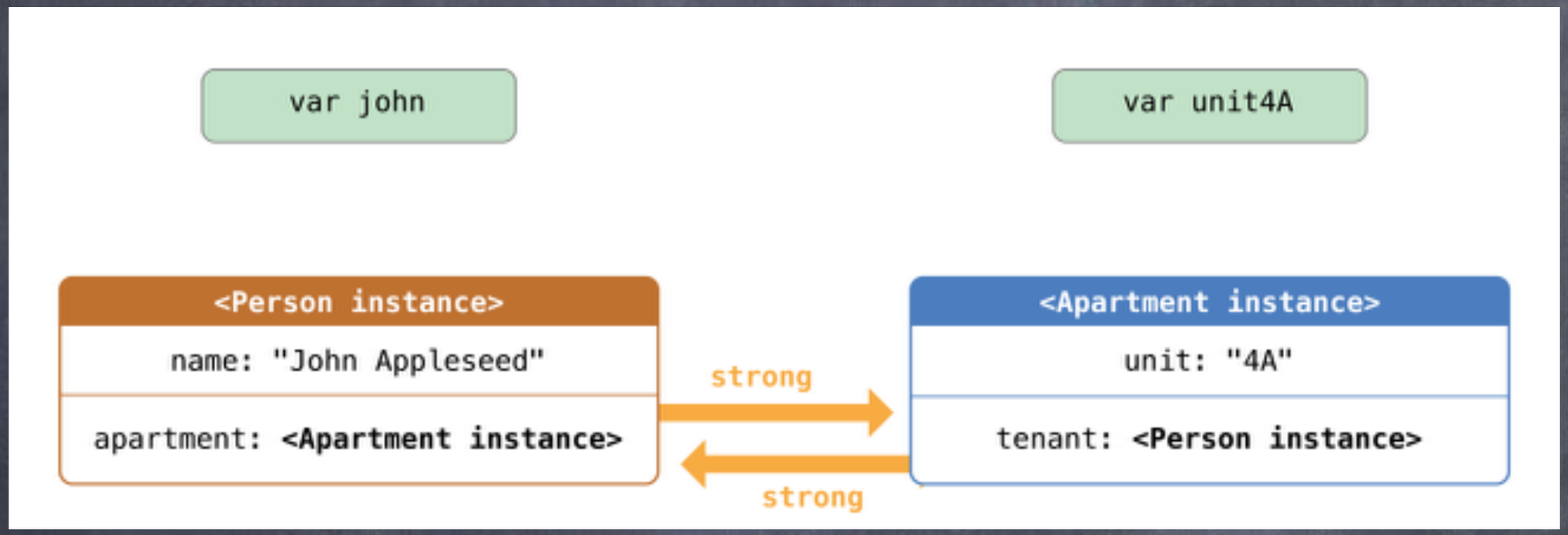
Retain Cycle

- Two strong pointers point to each other
- Neither of them leave the heap
- They will never be deallocated
- Not so good, because you will be having some objects you don't need
- Usually happens inside of the closures

Retain Cycle(Example)

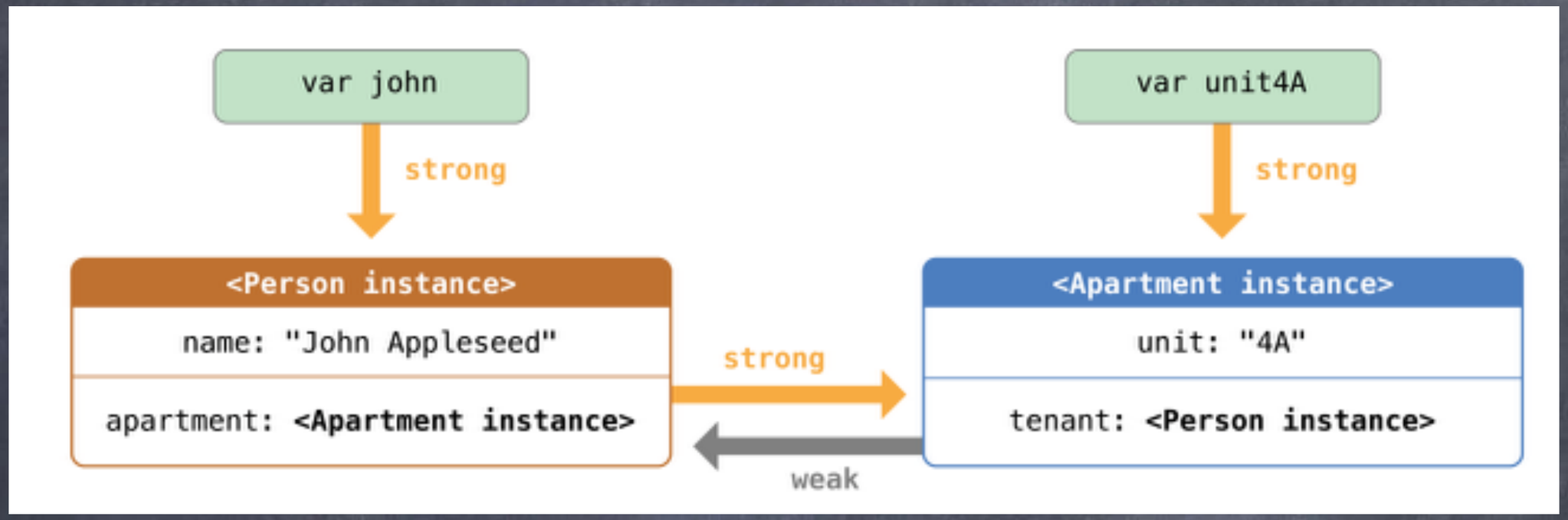
```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") }
}
class Apartment {
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}

var john: Person?
var unit4A: Apartment?

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

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

Retain Cycle(Example)



john and unit4A have strong pointers to each other, hence they will never leave the heap, even if you make one of them nil

Retain Cycle(Solution)



Just set one of the objects to be weak. Therefore, if the other will be deallocated, the object with weak pointer becomes nil

Retain Cycle(Solution)

```
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") }
}

var john: Person?
var unit4A: Apartment?

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

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

Demo