# Introduction to Swift and iOS Application programming

## whoami?

**Artemiy Sobolev**,
Lecturer at Moscow Institute of Physics and Technology
Senior Software Developer at Parallels, Inc.

*will be assisted by*
**Anastasia Soboleva**,
Software Developer at Parallels, Inc.

## You will carry out

- New programming language - Swift
- How to create simple applications on iOS
- Projects-homeworks

# Administrative issues

- 16 lessons
- 45 mins for lecture
- 45 mins of practice
- we have a break for almost whole January
- we have different classrooms

# Prerequisites

- Knowledge of at least 1 Objective Oriented Programming
- Mac computer that supports 10.11.5 to install Xcode 8
- You **don't** need to have iPhone

**Go install Xcode:**
Finder ->
MacBook Pro - Artemiy ->
connect as ->
login: student, password: student

## Administrative issues

- 16 lessons
- 45 mins for lecture
- 45 mins of practice
- we have a break for almost whole January
- we have different classrooms

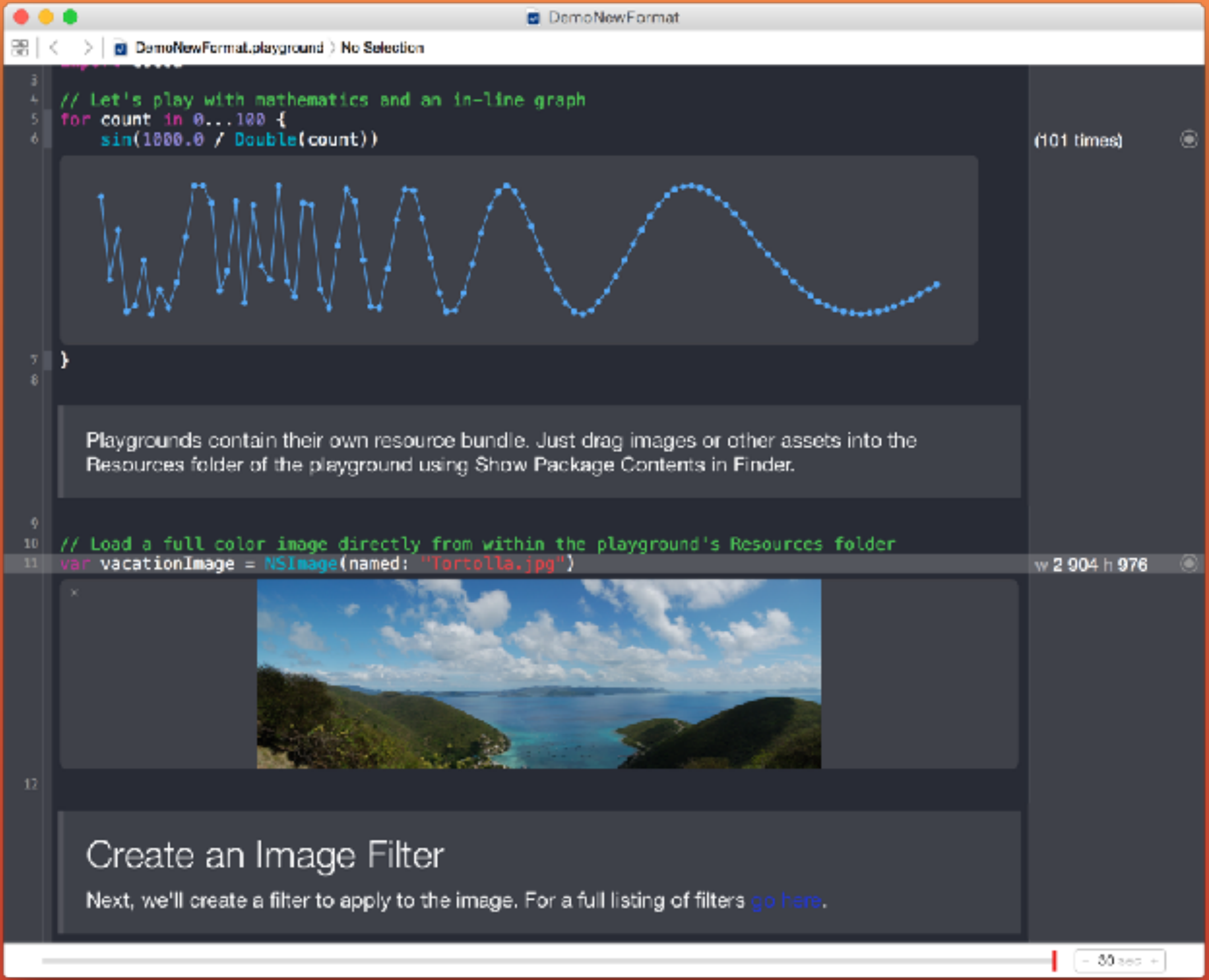
## How Mobile dev. is different?

- Edge cases
- New hardware
- Not 100% reliable internet connection
- Tiny phone, big data
- Unpredictable environment

## What is special about Swift

- Modern features
- Compiled with strict typing
- New programming language created by creators of LLVM and clang
- first Protocol oriented programming

# Playgrounds



```
// Let's play with mathematics and an in-line graph
for count in 0...100 {
    sin(1000.0 / Double(count))
}
```

(101 times)

Playgrounds contain their own resource bundle. Just drag images or other assets into the Resources folder of the playground using Show Package Contents in Finder.

```
// Load a full color image directly from within the playground's Resources folder
var vacationImage = NSImage(named: "Tortolla.jpg")
```

w 2 904 h 976

## Create an Image Filter

Next, we'll create a filter to apply to the image. For a full listing of filters go here.

# Consts and vars

```
// constant means you can not change it
let π = 3.141592

//  compiler would not be happy to see
//  π = 34

// variable means it can be changed later on
var votes = 10

// compiler is happy to see
votes += 1
```

# Functions

```
func simpleFunction() {
}

//  With return type
func simpleFunctionReturningInteger() -> Int {
    return 0
}

//  Local variables will be consts
func simpleFunction(string: String, integer: Int) -> Int {
    return 1
}
//  simpleFunction(string:integer:)
```

# Functions

```
//  the name is used to read the code like english language:
//  moveForward(position:by:)
func moveForward(position: Int, by amount: Int) -> Int {
    return position + amount
}

//  we can have a function overloading in Swift:
func print(string: String, reversed: Bool = false) {
    let stringToPrint = reversed ?
String(string.characters.reversed()) : string
    print(stringToPrint)
}

// you can call this function like
print(string: “Hello World")
```

## Value type vs. Reference type

```swift
var numberOfCars = 10
func changeNumberOfCars(numberOfCars: Int) {
    numberOfCars = 11
}
print(numberOfCars)
```

What do you expect to be printed?

## Value type vs. Reference type

- **value type semantics** - copy the whole peace of memory almost every time we pass it
- **reference type semantics** - copy only the pointer to a memory instead of value

# Value type vs. Reference type

| | Value | Reference |
|---|---|---|
| Copy, allocate | fast | expensive |
| Copies | for each use | shared |
| Preferred size | small | big |
| Expected lifetime | short | long |

All standard types are value types in swift

# Value type vs. Reference type

```
var votes = 10

var votesCopy = votes
votesCopy += 1
func print(votes: Int) {
    print(votes)
}
```

What do you expect to be printed?

# Value type vs. Reference type

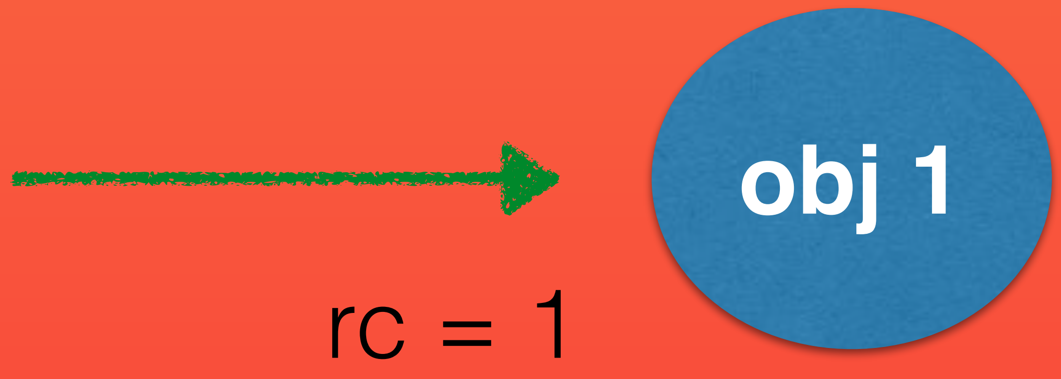
All classes have reference type semantics in Swift:

```
class Parking {
    var numberOfCars: Int = 0
}

func parkCar(on parking: Parking) {
    parking.numberOfCars += 1
}

let mektoryParking = Parking()
//  it means that if mutate class inside the function,
original value changes
parkCar(on: mektoryParking)
print(mektoryParking.numberOfCars)
```

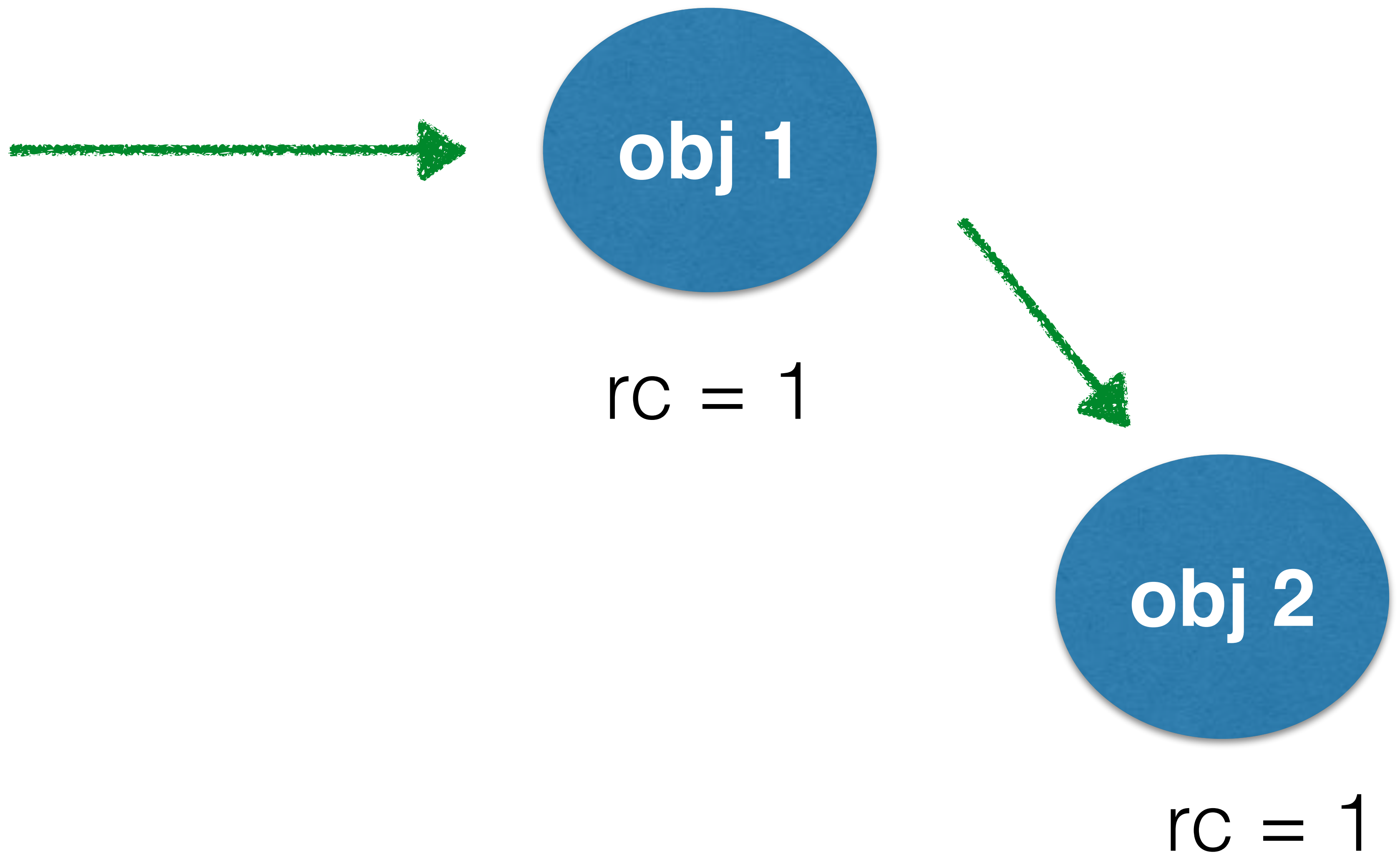
Question

# Automatic reference counting

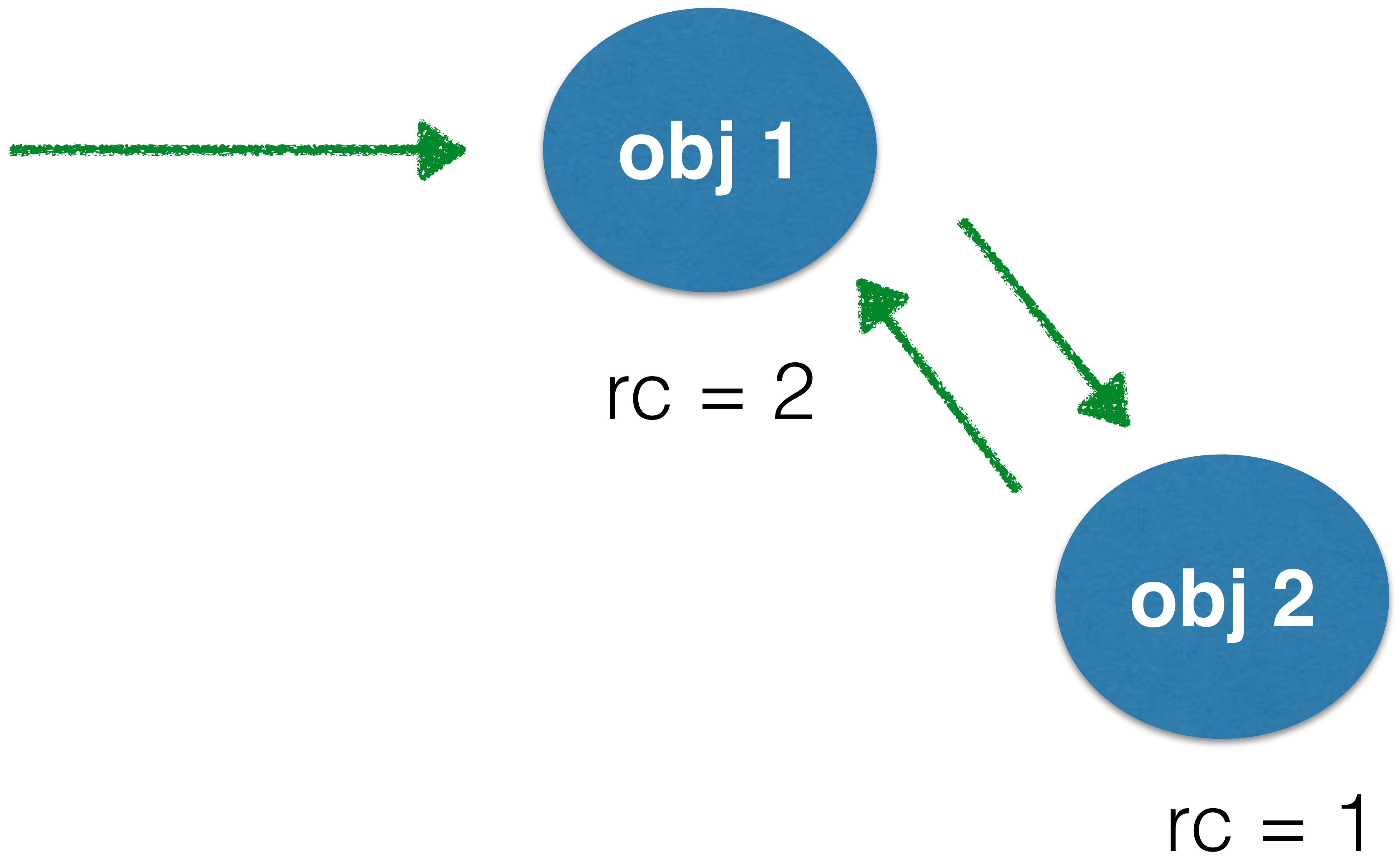
- How long will object stay in the heap
- Reference count for each object
- Objects live in heap
- There is no gb collector
- removed right when rc==0



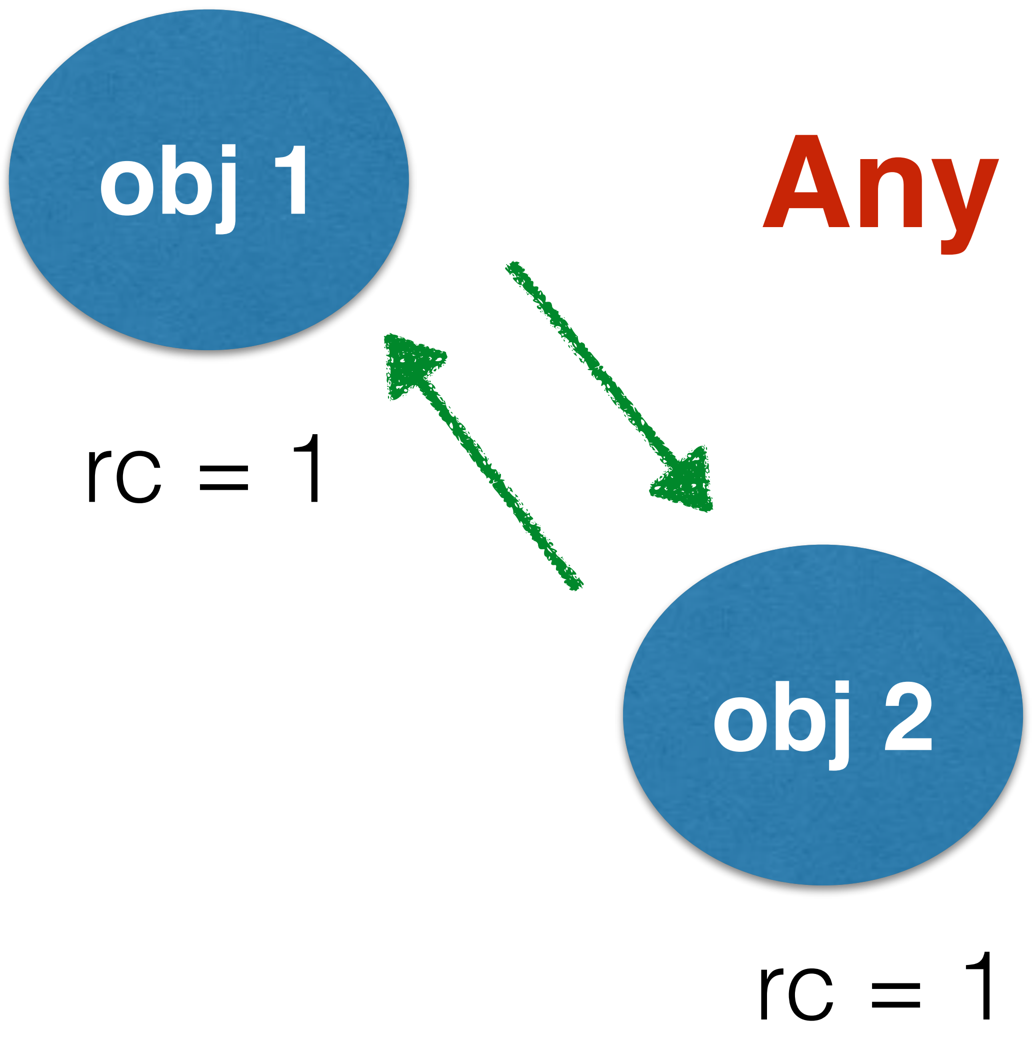
# ARC by example



# ARC by example

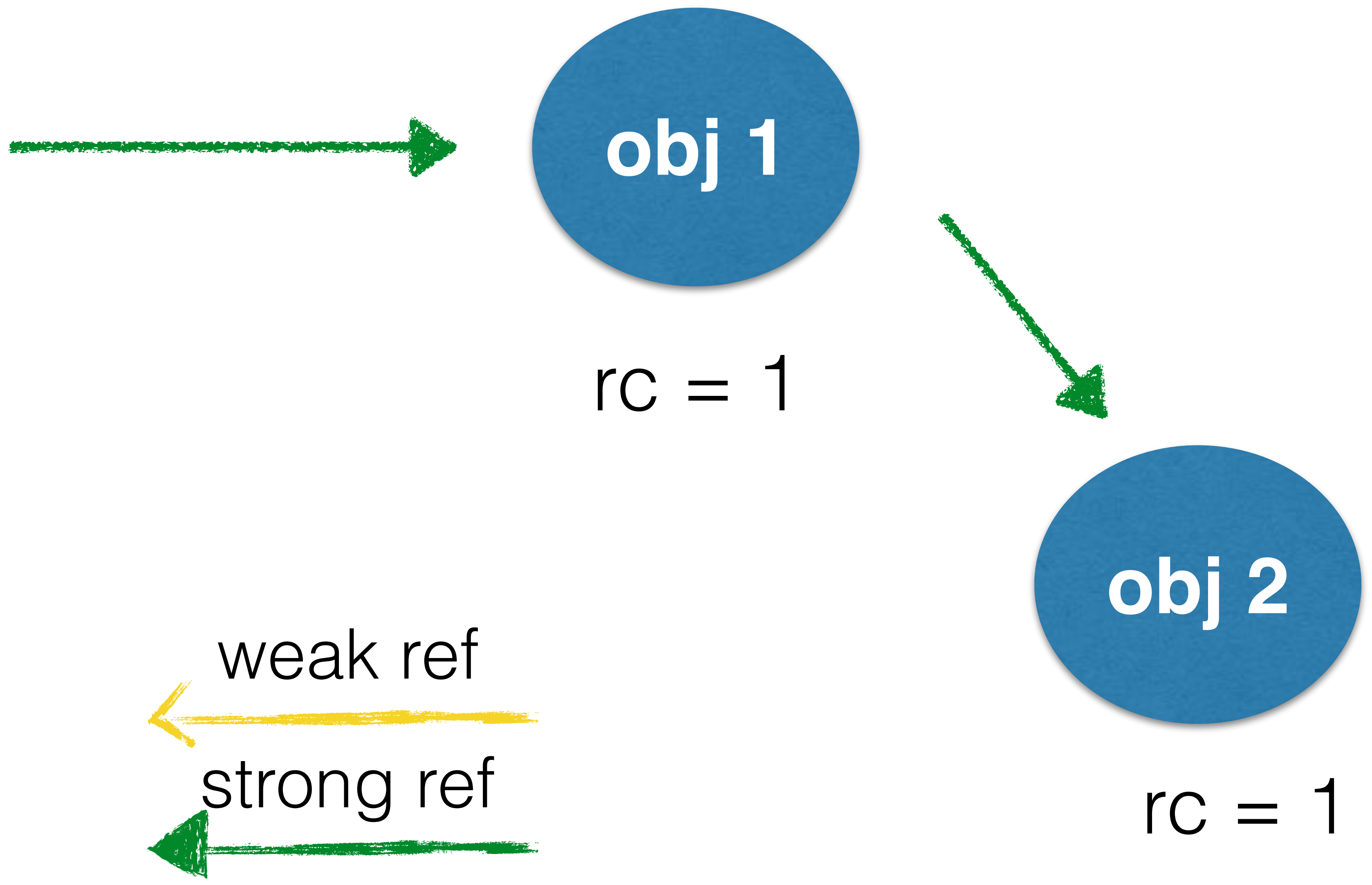


# ARC by example



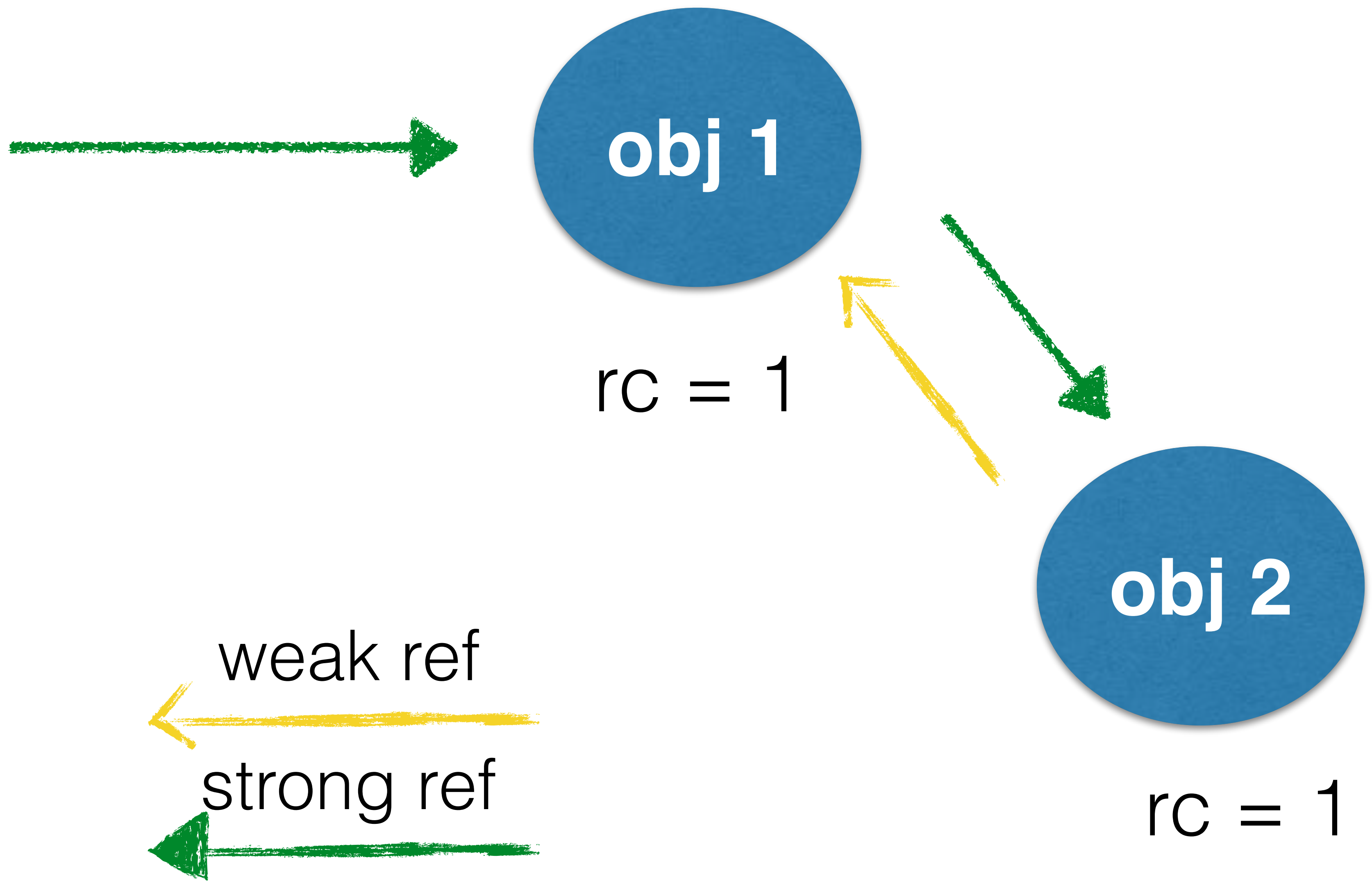
**Any problem?**

# ARC by example



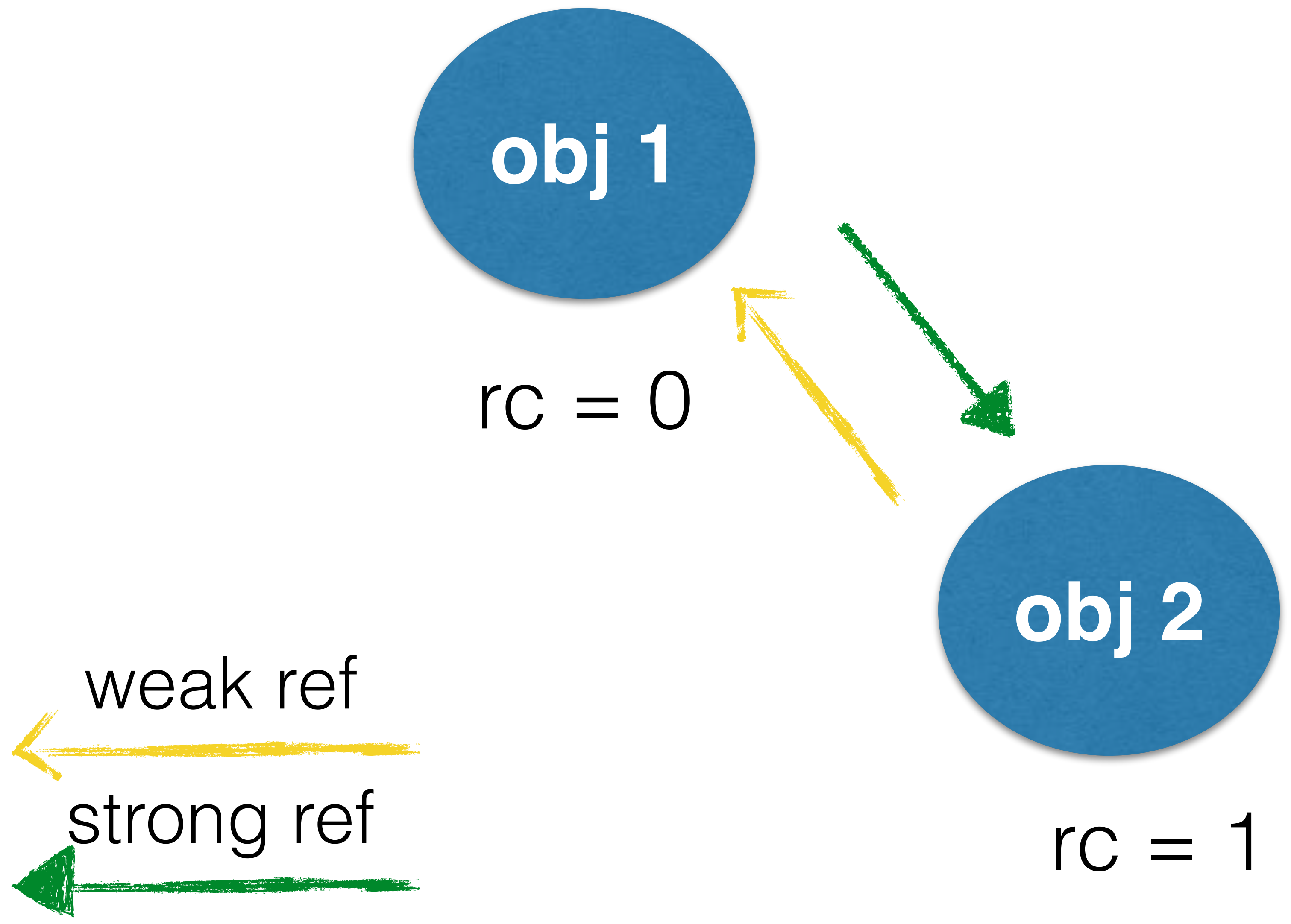
weak references don't increase rc, strong references do

# ARC by example



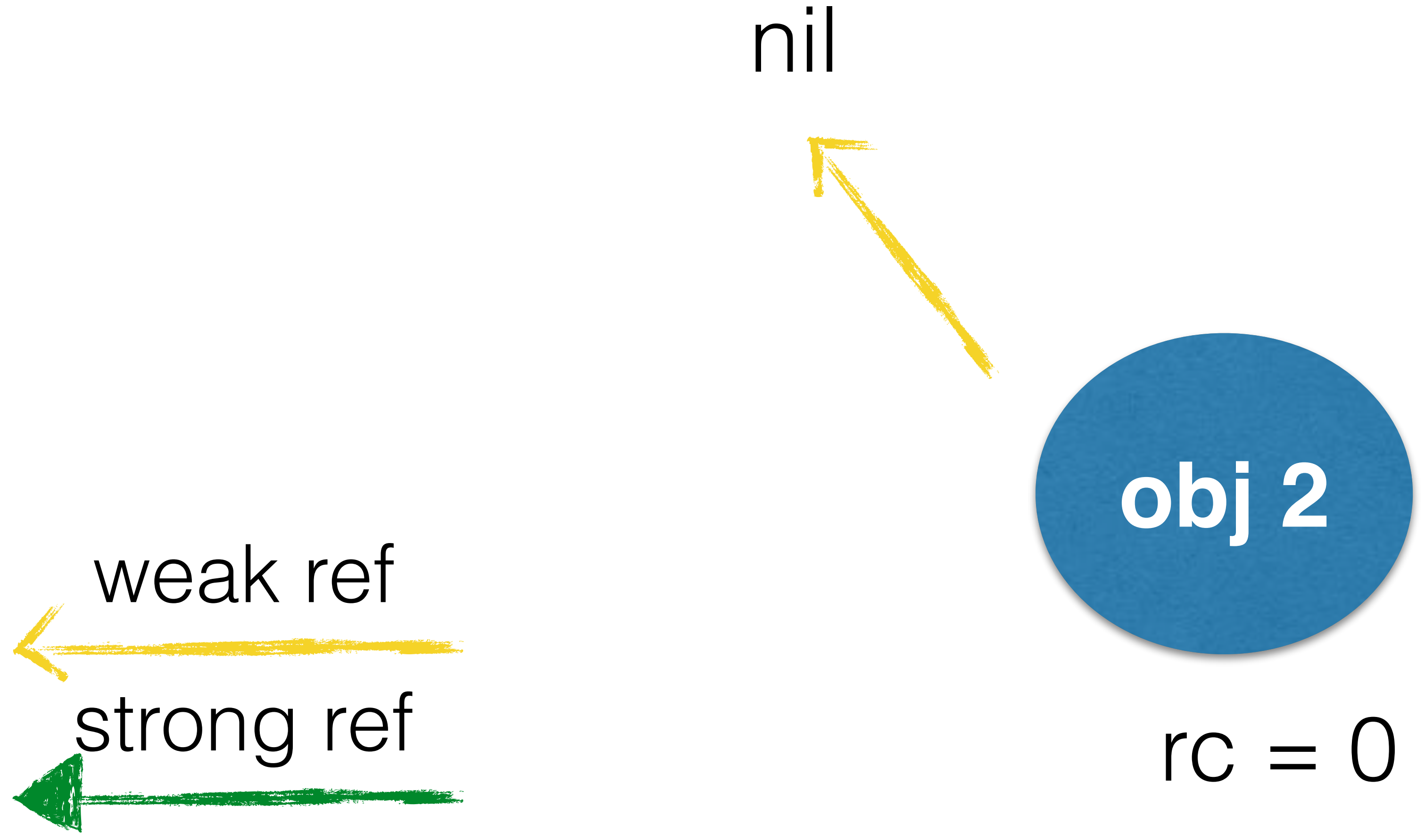
weak references don't increase rc, strong references do

# ARC by example



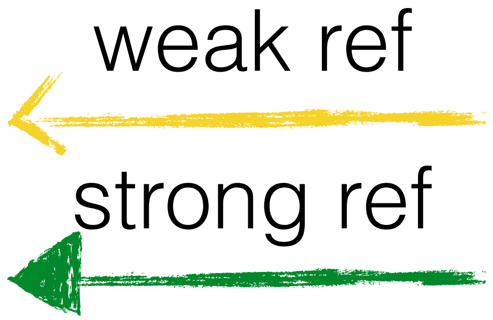
weak references don't increase rc, strong references do

# ARC by example



weak references don't increase rc, strong references do

# ARC by example



weak references don't increase rc, strong references do

# Value of nothing

- nil - nothing
- no function execution if calling from nil
- is called optionals in swift

# Optionals

```swift
var originalParking = Parking()
weak var parking = originalParking
parking = nil

//  Optional String, that can be nil
var myName: String?
myName = "Artemiy"
```

# Optionals

```
//  Optional chaining
var weakParking: Parking? = Parking()

//  We can do it implicitelly:
let numberOfCars = weakParking?.numberOfCars
//  It means that if parking was deallocated, the
numberOfCars will be nil

//  We can do it explicitelly:
let explicitNumberOfCars = weakParking!.numberOfCars
//  By doing this we mean weakParking should exist, crash
otherwise
```

## Task for you

Task for you - add simple class Car with

ability to check if it is parked on parking,

ability to park/unpark car

# Car task

```
class Car {
    weak var parking: Parking?
}

func park(car: Car, at parking: Parking) {
    car.parking = parking
    parking.numberOfCars += 1
}

func unpark(car: Car, from parking: Parking) {
    car.parking = nil
    parking.numberOfCars -= 1
}

func isParked(car: Car) -> Bool {
    return car.parking != nil
}
```

# Car task

```
//  If parking is no longer available(time passed),
isParked will automatically return nil:

let myCitroën = Car()
var europark: Parking? = Parking()
park(car: myCitroën, at: europark!)
isParked(car: myCitroën)                              true
europark = nil
isParked(car: myCitroën)                              false
```

# Optional chaining

```
let europark2 = Parking()
let myCitroën2 = Car()
var anotherCar: Car? = Car()
park(car: anotherCar!, at: europark2)
park(car: myCitroën2, at: europark2)



anotherCar?.parking?.numberOfCars
//  By saying this we are telling that we don't care if
anotherCar == nil or it is not parked

anotherCar?.parking!.numberOfCars
//  By saying we don't care that car can not exist, but it
should be parked
```

# Optional checks

```
//  Special way to get instance if it is not nil

if let car = anotherCar {
    print("car exists")
}

//  Special construction to execute rest of the code
if is not nil
guard let car2 = anotherCar else {
    fatalError("car2 dont exists")
}
```

# Methods and properties

```
class Parking {
    var numberOfCars: Int = 0
}

class Car {
    weak var parking: Parking?

    func park(at parking: Parking) {
        self.parking = parking
        parking.numberOfCars += 1
    }

    func uppark() {
        parking = nil
        parking?.numberOfCars += 1
    }

    var isParked: Bool {
        return parking != nil
    }
}
```

**computed property** - is calculated each time depending on other functions, is called

**method** - functions, associated to a class.

you can override property's **setter** as well

# Methods and properties

Compare method-based use case

```
let myCitroën = Car()
var europark = Parking()
myCitroën.park(at: europark)
myCitroën.isParked
```

To function base use case

```
let myCitroën = Car()
var europark: Parking? = Parking()
park(car: myCitroën, at: europark!)
isParked(car: myCitroën)
```

# Methods and properties

```
class Pet {
    var name: String {
        willSet {
            print("pet name is about to change")
        } didSet {
            print("pet name did change")
        }
    }

    init(name: String) {
        self.name = name
    }
}
```

**willSet/didSet** - with this construction you can execute code that
special method called **init** is used to construct instance with properties