iOS alapú szoftverfejlesztés

Dr. Blázovics László

Blazovics.Laszlo@aut.bme.hu

2021. Szeptember 14.

BME AUT

Automatizálási és
Alkalmazott
Informatikai Tanszék

Labor – QB115

- Laborbeosztás a Github-on
  - L1 - 12:15-13:45
    Blázovics László
  - L2 14:15-15:45
    Gavrillás Kristóf
- Aki akar, hozhat saját gépet -> Lehetőleg oda üljön, ahol nincs gép
- Segédletek szintén a Github-on


Streameld a napfényes Kaliforniából.
Nézd meg az angol nyelvű Apple-eseményt ma 19:00 órától
az amerikai honlapunkon.
További információ ›

Memóriakezelés

Memóriakezelés iOS-en

```
let hero = GameCharacter()
```

- Egy GameCharacter objektum létrejött a memóriában
- A **hero** nevű változó (konstans referencia) hivatkozik a létrehozott *GameCharacter* példányra
- Ha túl sok objektumot hozunk létre, elfogy a memória

Memóriakezelés iOS-en II.

- **Memóriakezelés**: az olyan objektumok által lefoglalt memória felszabadítása, amikre többé már nincs szükség
- Hogyan?
  - Régi iOS verziók (iOS 5): a programozónak manuálisan kellett hívogatni a memória felszabadítását végző metódusokat
  - Most: automatikus referencia számlálás (**A**utomatic **R**eference **C**ounting: **ARC**)
  - *Objective-C-vel is működik*

A kezdetekben

- A programozó feladata az objektumok életciklusának felügyelete, speciális referenciakezelő üzenetekkel
- Bizonyos utasításokkal az objektum tulajdonosává válunk: **alloc**, **retain**, **new**…
- Ha nincs szükségünk az objektumra, le kell mondani róla: **release**
- Egy objektum automatikusan törlődik, ha már egy tulajdonosa sincs
- Sok hibalehetőség:
  - Ha elfelejtünk lemondani egy objektumról: memory leak
  - Ha elfelejtjük lefoglalni a „tulajdonjogot” egy objektumra: bármikor törlődhet a tudtunk nélkül
- Kezdő programozók számára sok idő volt elsajátítani

ARC - Automatic Reference Counting

- A **fordító** elemzi a kódot és automatikusan elhelyezi benne a memória felszabadítását végző műveleteket
  - Egy objektumra akkor nincs többé szükség, ha nincs rá több referencia
    - Referencia: változó, konstans, property…
- **Nem Garbage Collection!**
  - Fordítási időben generálódik a memóriakezelő kód
  - Programozó számára láthatatlan
  - Nincs overhead futási időben (ellentétben a Garbage Collectionnal)

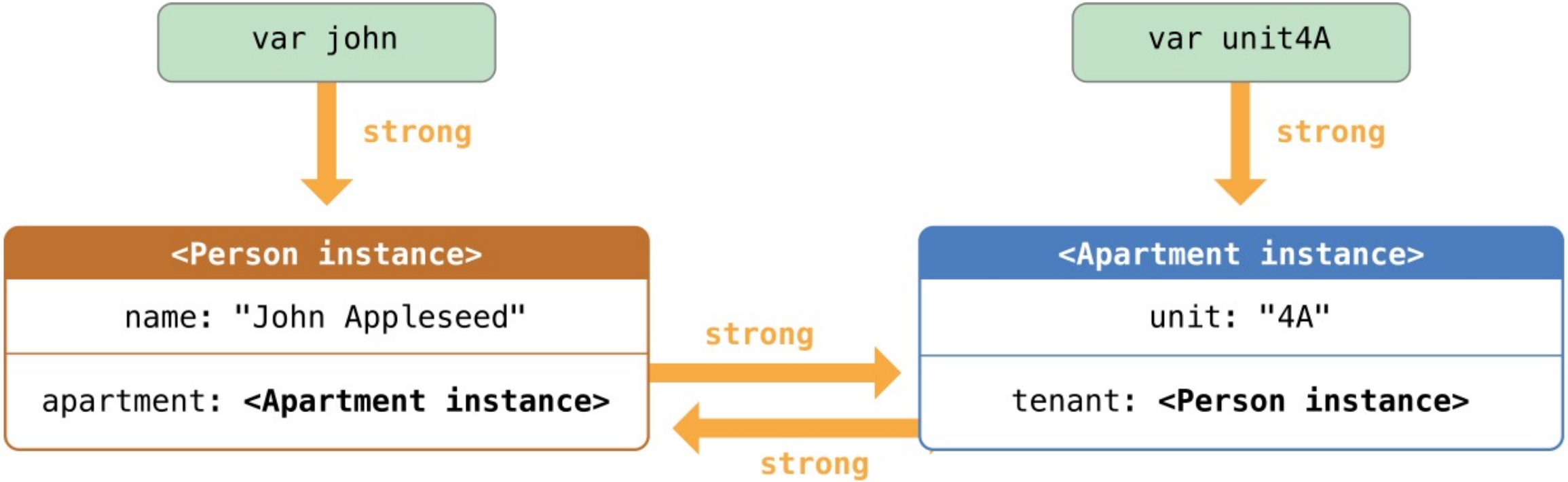
Objektumok élete

- Mikor törlődnek az objektumok?
- Amikor már egyetlen referencia sincs rájuk!

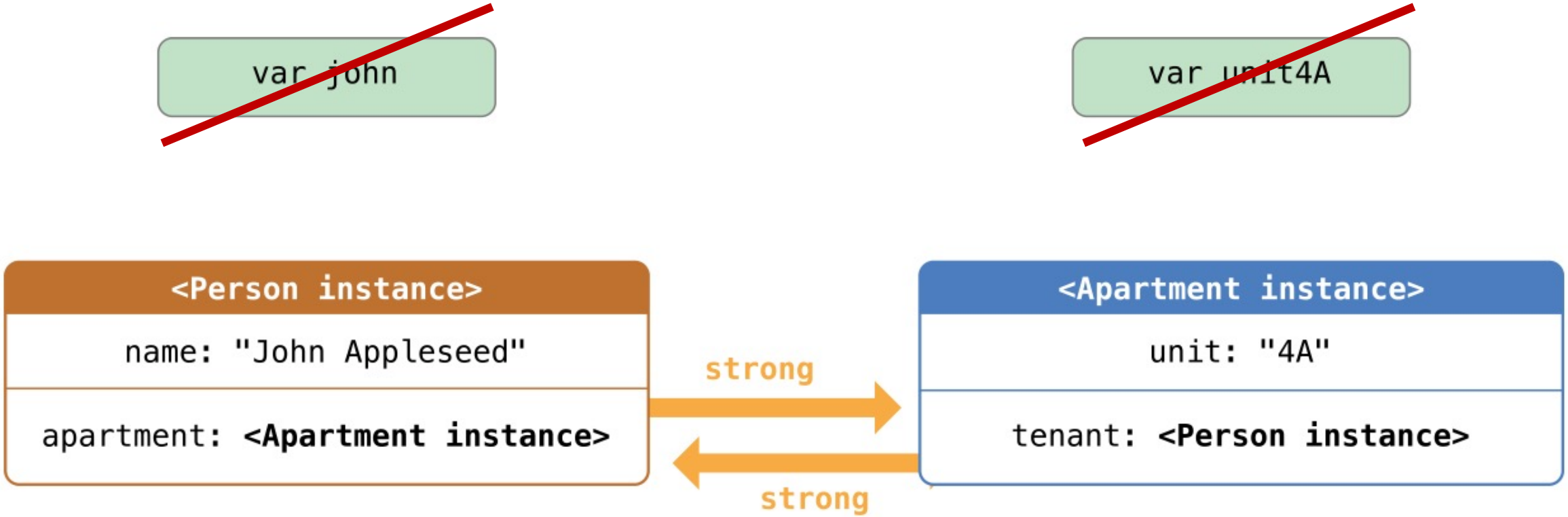
```
// GameCharater objektum létrehozása
var firstHero: GameCharacter? = GameCharater()
// A hero használata...
firstHero = nil
// az objektum törlődik, mert nincs rá több referencia


func printHero() {
// Egy új GameCharacter jön létre minden alkalommal, mikor a függvényt meghívják
    let secondHero = GameCharacter()
    print(secondHero)
}
// Amikor a függvény visszatér, törlődnek a lokális változók (itt GameCharacter
referencia), így az objektum törlődik a memóriából
```

Körkörös hivatkozás



Körkörös hivatkozás II



- Körkörös referencia (reference cycle): az objektumok körkörösen egymásra hivatkoznak, és így "életben tartják" egymást.

Gyenge referencia

- A körkörös referenciák megakadályozzák, hogy felszabaduljanak a szükségtelen objektumok, ezért finomhangolni kell
- **Gyenge (weak) referencia**: az ARC nem veszi figyelembe, mikor egy objektumra mutató referenciákat számolja

```
weak var date: Date? = Date()
```

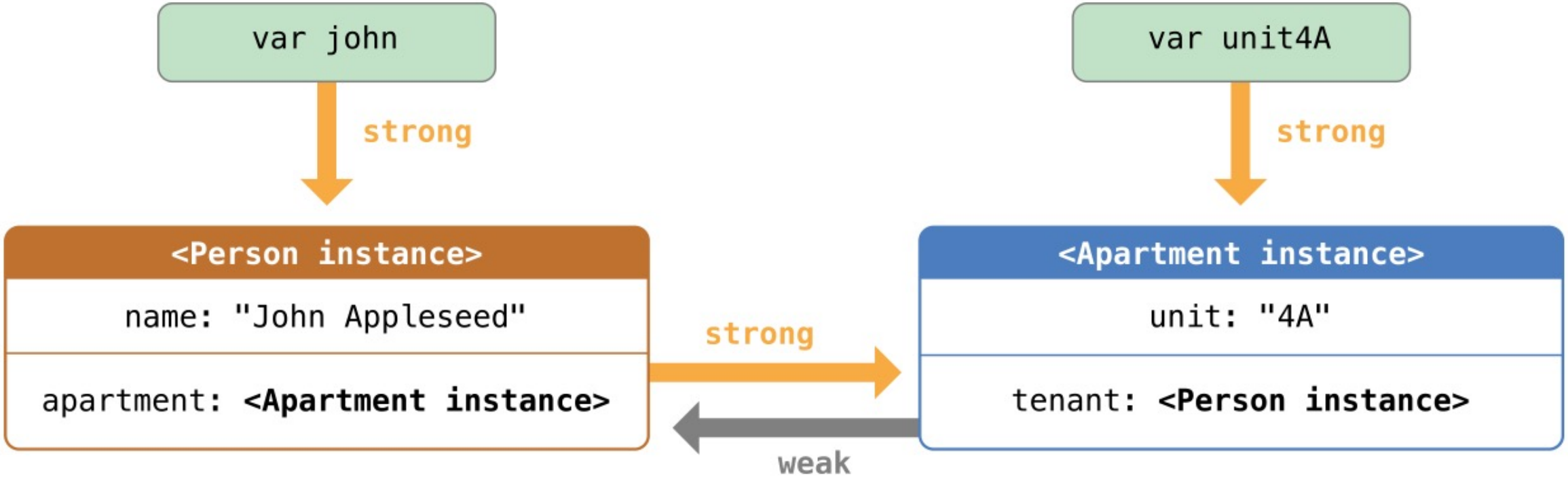
  > **A weak referenciák mindig optional-ök**, mert az általuk mutatott objektum törlődhet a memóriából (és ilyenkor nil lesz az értékük)

- **Erős (strong) referencia**: az alapértelmezett referenciatípus, melyet az ARC figyelembe vesz az objektumra mutató referenciák számolásánál

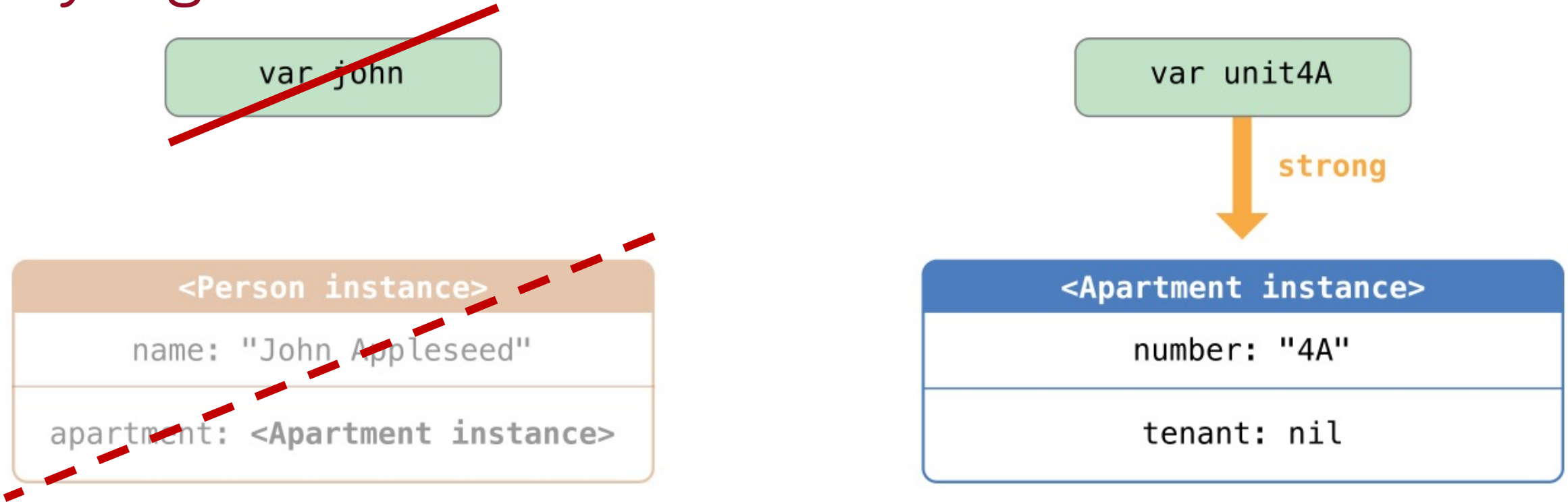
```
var date: Date = Date()
```

- **Általános szabály**: egy objektum akkor törlődik a memóriából, ha nincsen már több erős referencia rá

Gyenge referencia II.



Gyenge referencia III.

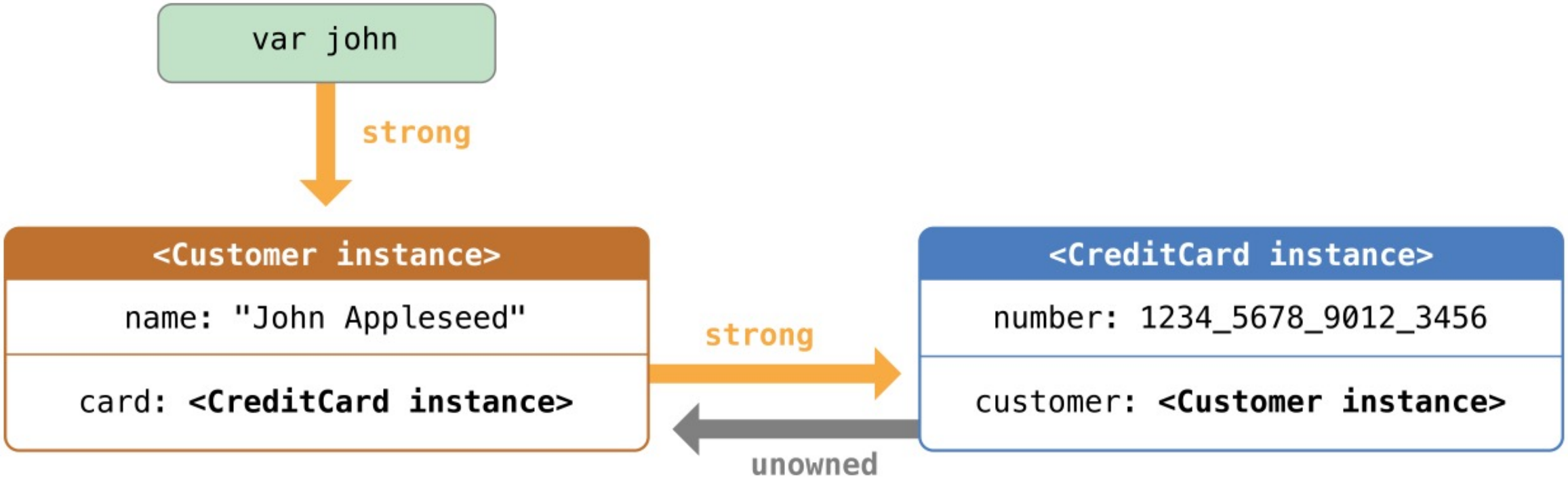


- Nem marad több erős referencia a *Person*-ra, így törlődik a memóriából

Unowned referencia

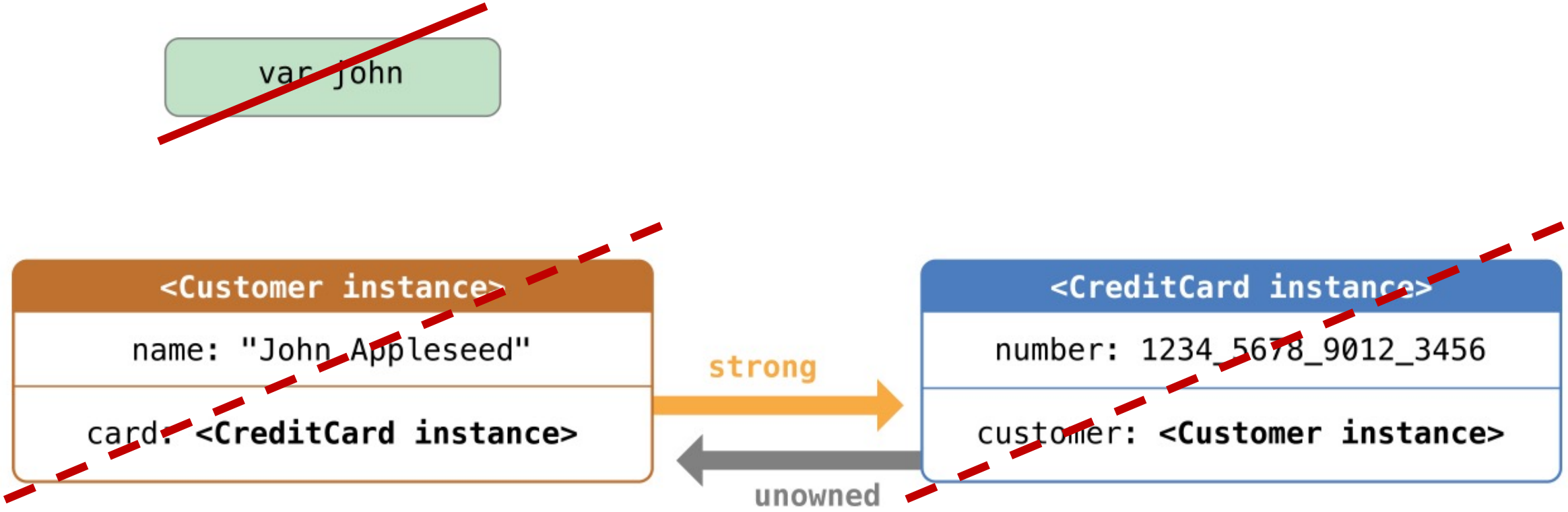
- A weak mellett létezik még egy "gyenge" referencia típus: **unowned**
  - Hasonlóan nem számít bele a referencia számlálásba mint a weak referencia
  - Ellentétben a weak referenciákkal, az **unowned referenciák nem optional értékek**
  - Az *unowned* referenciának mindig kell, hogy értéke legyen, különben crash-el a kód
- Olyan esetekben használjuk, mikor a hivatkozó objektum csak a hivatkozottal együtt létezhet
  - Tipikusan az init-ben létrejövő objektumok esetében

Unowned referencia II.



- A *Customer* birtokolja a *CreditCard*-ot (ezért strong)
- A *CreditCard* nem létezhet *Customer* nélkül, de nem is birtokolja a *Customer*-t

Unowned referencia III.



- Nem marad több erős referencia a *Customer*-re, így törlődik a memóriából

strong, weak, unowned?

- **strong**: szülő birtokolja a gyereket és ezért erősen hivatkozik rá, „amíg van szülő, a gyerek is életben marad”

```
var child: Child
```

- A gyerek nem birtokolja szülőt
  - **weak**: ha a gyerek létezhet szülő nélkül is: „ha nincs szülő, a gyerek tovább él”

```
weak var parent: Parent?
```

  - **unowned**: ha nem lehet olyan eset, hogy a gyerek tovább létezhet a szülő nélkül

```
unowned var parent: Parent
```

Memóriakezelés összefoglalás

- Az ARC teljesen automatikusan végzi a dolgát, **de**
  - a weak, strong és unowned referencia típusokat azért a **programozónak kell jól megválasztani**
- Szülő-gyerek / tartalmazás viszonyoknál a szülőre weak vagy unowned referenciát használjunk
  - **weak**: ha van értelme nil értéknek
  - **unowned**: ha csak addig létezhet az objektum, amíg a szülő

Függvény típusok és Closure-ök {}

Függvény típusok

- A függvények Swiftben **referencia** típusok
- Típusukat a **bemenő paramétereik** és a **visszatérési értékük** határozza meg
- Pl.: (Int, Int) -> Int típusú függvények:

```
func add(a: Int, b: Int) -> Int {
  return a + b
}
func multiply(a: Int, b: Int) -> Int {
  return a * b
}
```

Függvény típusok használata

- Ugyanúgy használhatóak, mint bármilyen más típus
- Lehetnek változók vagy konstansok:

```
func add(a: Int, b: Int) -> Int {
  return a + b
}

func multiply(a: Int, b: Int) -> Int {
  return a * b
}

var calculation: (Int, Int) -> Int = add

calculation = multiply
print("Result: \(calculation(2, 3))")
let anotherCalculation = add
```

Függvény típusok használata II.

- Lehetnek más függvények paraméterei:

```
func printResult(_ calculation: (Int, Int) -> Int, _ a: Int, _ b: Int) {
  print("Result: \(calculation(a, b))")
}

printResult(add, 3, 5)
```

- Egymásba ágyazhatók, ezáltal belső függvényeket hozva létre

Ismétlés: címkék, visszatérési típus

- Minden paraméterhez rendelhető egy címke, ebben az esetben azt kiírva kell hívni

```
func square(of number: Int) -> Int {
  return number * number
}
square(of: 10)
```

- Ha nem szeretnénk argumentum nevet, akkor ezt az _ jellel kell jelölni, ekkor a paraméter neve elhagyható híváskor
- A visszatérés nélküli függvények esetében a -> Void elhagyható

Closure

- Egy kódblokkba zárt kifejezés
- Más nyelvekben *lambda* kifejezés, Objective-C-ben pedig *blokk*
- A Closure egy olyan kódblokk, melyet referencia típusként használhatunk: értékül adhatjuk változóknak, átadhatjuk függvényeknek, stb.
  - A függvények is Closure-ök
  - Anonim Closure: nincs külön azonosítója

Closure II.

- Legtöbbször függvény paraméterként adjuk át őket
  - > Eseménykezelés
  - > Egy algoritmus definiálása és átadása
  - > Animáció
- Példa:

```
let alert = UIAlertController(title: "Alert!", message: "New Login?!",
                                                    preferredStyle: .alert)
let yesAction = UIAlertAction(title: "Yes!", style: .default, handler: {
action in
  print("Do something with login.")
})
alert.addAction(yesAction)
```

Closure szintaxis

- Legbővebb szintaxis:

```
{ (<paraméterek>) -> <visszatérési típus>
in
  <kifejezések>
}
```

- Példa

```
{ (a: String, b: String) -> Bool in
  return a < b
}
```

- Az in jelzi a paraméterek végét, és hogy következik a kifejezés
- A Closure paraméterei lehetnek inout típusúak, továbbá változó hosszúságúak

Closure "egyszerűsítése"

- Példa
  - Egy tömb képes visszaadni adott logika szerint rendezett tömböt a `sorted(by:)` metódusával, ami paraméteréül (mely egy Closure) a rendezés logikáját várja.

```
let names = ["Attila", "Laszlo", "Geza", "Istvan"]
func backward(_ s1: String, _ s2: String) -> Bool {
  return s1 > s2
}
var reversedNames = names.sorted(by: backward)
```

Closure "egyszerűsítése" II.

- Legbővebb szintaxis:

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
  return s1 > s2
})
```

- A fordító a típusokat (paraméterek, visszatérési érték) ki tudja találni, így elhagyhatók (*Type Inference*):

```
reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 })
```

Closure "egyszerűsítése" III.

- Egyszerű, egysoros kifejezésnél automatikus visszatérés (return elhagyható):

```
reversedNames = names.sorted(by: { s1, s2 in s1 > s2 })
```

- A Swift automatikusan biztosít argumentum neveket $ jellel és a paraméter sorszámával (0-val kezdve), így a paraméter lista is elhagyható az in kifejezéssel együtt

```
reversedNames = names.sorted(by: { $0 > $1 })
```

- Ahol a bemeneti paramétereknek és a visszatérési értéknek megfeleltehető egy operátor, még ezekre sincs szükség:

```
reversedNames = names.sorted(by: >)
```

Trailing Closure syntax

- Ha egy függvény/metódus utolsó paramétereként adunk át egy Closure-t, írhatjuk a zárójeleken kívül is
  - Default syntax

```
names.sorted(by: { $0 > $1 })
```

  - **Trailing Closure** syntax

```
names.sorted() { $0 > $1 }
```

- Egy paraméter esetén a zárójelek elhagyhatók

```
names.sorted { $0 > $1 }
```

Változók hivatkozása Closure-ből

- A Closure-ből hivatkozhatók a Closure környezetében látható változók
  - Ez a **capture** ("begyűjtés")
  - Nem kell paraméterként átadni a **Closure-ön kívül** definiált változókat
  - A változók akkor is tovább használhatók, ha az eredeti kontextus már nem létezik

```
var animals = ["fish", "cat", "chicken"]
var printAnimalsTask = { print(animals) }
```

- A Closure-ökben behivatkozott változók egészen addig léteznek, amíg a Closure létezik

Closure és self

- Amikor self property-kre vagy metódusokra hivatkozunk egy Closure-ből, mindig ki kell írni a **`self`**-et
  - > Kihangsúlyozza, hogy a Closure a self értékét is "begyűjti", és a self egészen addig nem törlődhet, amíg a Closure létezik

```
UIView.animate(withDuration: 3.0, delay: 0, animations: {
  self.horse.center = touchPoint
}, completion: nil)
```

Closure memóriakezelés

- A Closure-ök is referencia típusok: ugyanúgy ARC kezeli őket

```
// Erős hivatkozás egy Closure-re
var someClosure: (() -> Void)?
```

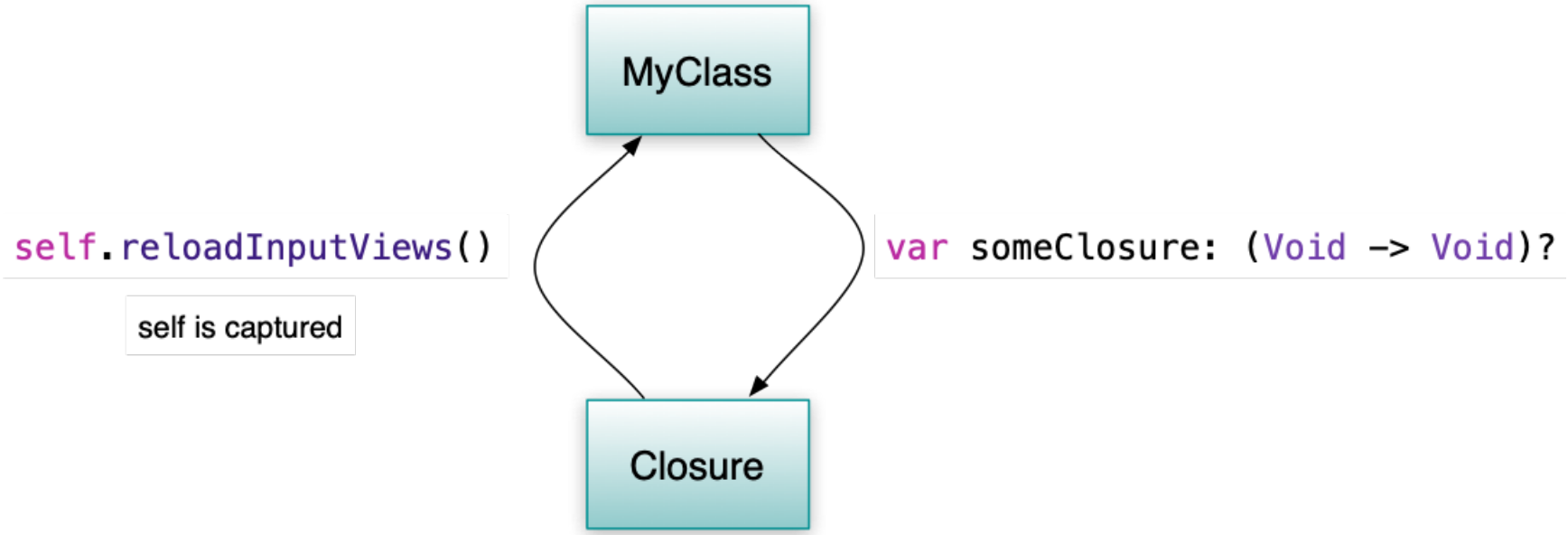
- Alapból egy Closure **minden behivatkozott külső változóra erősen hivatkozik**
  - self hivatkozása esetén arra is létrejön egy erős referencia!

```
// A Closure automatikusan létrehoz egy erős referenciát a view által
hivatkozott objektumra (a nézet egészen addig létezni fog, amíg a Closure)
let view = UIView()
someClosure = {
    print("\(view.frame)")
}
```

Closure-ök és körkörös hivatkozás

- Probléma: Closure-ökben könnyen **keletkezhetnek körkörös** hivatkozások a behivatkozott értékeken keresztül

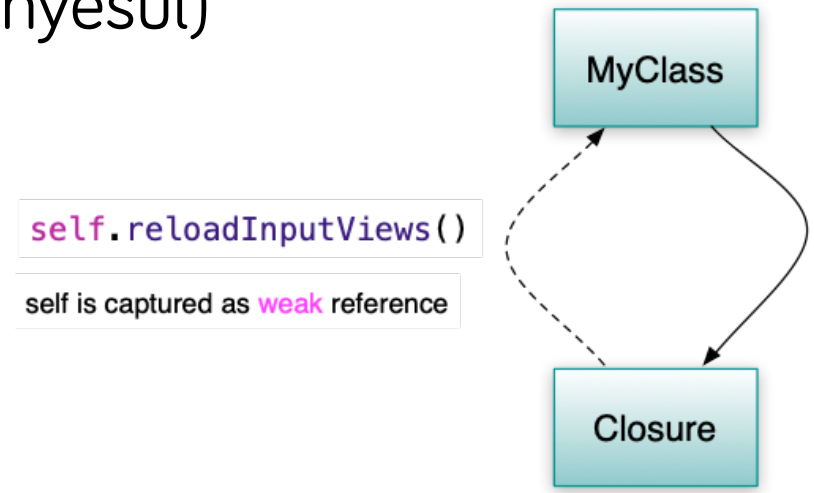
```
someClosure = {
  self.reloadInputViews()
}
```



Closure „begyűjtési lista” (Capture list)

- **Capture list**:
  - *Referencia típusok* esetén meghatározza, hogy a Closure-ben behivatkozott objektumokra milyen típusú hivatkozás jöjjön létre (**unowned**, **weak**)
  - Érték típusok esetén másolatot készít az aktuális értékről (annak későbbi külső változtatása a Closure-ön belül nem érvényesül)
- **[ ]** között, a Closure paraméterei előtt:

```
someClosure = { [weak self] in
  self?.reloadInputViews()
}
```



Escaping Closure

- Olyan Closure, melyet függvény paraméterként adunk át, de csak azt követően hívódik meg, hogy a függvény visszatért
- Kötelező kiírni: `@escaping`

```
var completionHandlers: [() -> Void] = []

func addHandler(completionHandler: @escaping () -> Void) {
  completionHandlers.append(completionHandler)
}
```

Funkcionális programozás

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Funkcionális programozás

- A programozási feladatot egy függvény kiértékelésének tekinti
- Fő eleme az érték és a függvény
- Azt specifikálja, hogy *mit* kell kiszámítani és nem azt, hogy *hogyan*
- A program gyakorlatilag függvény hívásokból és kiértékelésekből áll, **nincsenek állapotok és mellékhatások**

Funkcionális programozás

- A Swift nem funkcionális programozási nyelv, de tartalmaz bizonyos funkcionális programozási megoldásokat (lásd. majd SwiftUI)
- Funkcionális metódusok gyűjtemény típusokon
  - Új gyűjteménnyel térnek vissza, az eredetit nem változtatják
  - Láncolhatók
  - `filter(_:)`: szűrés
  - `map(_:)`: elemek transzformálása
  - `flatMap(_:)`: kollekciók "kilapítása"
  - `compactMap(_:)`: opcionális típust tároló kollekció esetén opcionális értékek kiszűrése
  - `reduce(_:)`: egy érték képzése az elemekből
  - `zip(_:_:)`: két gyűjtemény összefésülése

filter

- A Closure-ben megadott kód szerint szűri a gyűjteményt
- Imperatív:

```
var even = [Int]()

for num in [1, 2, 3, 4, 5] {
    if num % 2 == 0 {
        even.append(num)
    }
}
```

- Funkcionális:

```
let even = [1, 2, 3, 4, 5].filter { $0 % 2 == 0}
```

map

- A Closure-ben definiált kódot (transzformációt) végrehajtja minden elemen
- Imperatív:

```
var doubles = [Int]()

for num in [1, 2, 3, 4, 5] {
  doubles.append(num * 2)
}
```

- Funkcionális:

```
let doubles = [1, 2, 3, 4, 5].map { $0 * 2}
```

flatmap

- Kollekciók „kilapítása”: egyszintű kollekciók létrehozása

```
let lotrParties = [["Frodo", "Aragorn"], ["Sauron", "Saruman"]]

let lotrAll = lotrParties.flatMap { $0 }

print(lotrAll)

//["Frodo", "Aragorn", "Sauron", "Saruman"]
```

compactMap

- Opcionális típust tároló kollekció esetén opcionális értékek kiszűrése
- Kollekció elemeinek olyan átalakítása, ami nem biztos, hogy mindig sikerrel jár (pl.: `String`-ből `Int`)

```
let numbers: [String?] = ["1", "2", nil, "apple"]

let numbersWithoutNil: [String] = numbers.compactMap { $0 }
print(numbersWithoutNil)
//["1", "2", "apple"]

let integers: [Int] = numbersWithoutNil.compactMap { Int($0) }
print(integers)
// [1, 2]
```

reduce

- Egyetlen értéket készít a gyűjteményből a Closure-ben megadott kód alapján
- Imperatív

```
var sum = 0
for num in [1, 2, 3, 4, 5] {
  sum += num
}
```

- Funkcionális

```
let sum = [1, 2, 3, 4, 5].reduce(0) { $0 + $1 }
```

- Tovább egyszerűsítve

```
let sum = [1, 2, 3, 4, 5].reduce(0, +)
```

zip

- Két gyűjtemény összefésülése

```
let numbers = [1, 2, 3]
let topics = ["Introduction", "Swift", "MVC"]
let lectures = zip(numbers, topics)

for (number, topic) in lectures {
    print("\(number).: \(topic)")
}

//1.: Introduction
//2.: Swift
//3.: MVC
```

Protokoll orientált programozás

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Ismétlés: Protocol

- Swiftben csak egy ősosztály lehet
  - Csak `class`-ok között működik
- **Protocol**: metódus és property deklarációk listája (más nyelvekben "interfész")
  - Implementációt nem tartalmaz
- `class`-ok, `struct`-ok és `enum`-ok is megvalósíthatják (adoptálhatják)
  - **Tetszőleges számú Protocolt valósíthatnak meg**

Default Protocol implementáció

- A `protocol`-t megvalósítók megkapják az implementációt
  - > De felül is írhatják az alapértelmezett viselkedés
- **Extension**-ök segítségével

Milyen típusú objektum tudja adaptálni

```
protocol LoadingViewController: AnyObject {
    var activityIndicator: UIActivityIndicatorView! { get set }
    func showLoading()
    func hideLoading()
}
extension LoadingViewController {
    func showLoading() {
        activityIndicator.startAnimating()
    }
    func hideLoading() {
        activityIndicator.stopAnimating()
    }
}
```

Default Protocol implementáció II.

- Default implementáció szűkítése: `where`

```
extension LoadingViewController where Self: UIViewController {
  func showLoading() {
      activityIndicator.startAnimating()
  }
  func hideLoading() {
      activityIndicator.stopAnimating()
  }
  func addActivityIndicator() {
      activityIndicator = UIActivityIndicatorView(style: .gray)
      view.addSubview(activityIndicator)
  }
}
```

Protokollorientált programozás

- A közös funkcionalitás leszármazás helyett (default/specializált) Protocol implementációkon keresztül
- "Composition over inheritance”
- Nem OOP helyett, hanem mellett

Példa

```
// Protokoll definiálása
protocol ReusableView: class {}

// Default implementáció UIView-kra korlátozva
extension ReusableView where Self: UIView {
    static var reuseIdentifier: String {
        return String(describing: self)
    }
}
```

Példa II.

```
// Használat: hozzáadunk egy új metódust a meglévő UITableView class-hoz, ami
kihasználja a protokollunkat
extension UITableView {
    func dequeueReusableCell<T: UITableViewCell>(forIndexPath indexPath:
IndexPath) -> T where T: ReusableView {
        guard let cell = dequeueReusableCell(withIdentifier: T.reuseIdentifier,
for: indexPath as IndexPath) as? T else {
            fatalError("Could not dequeue cell")
        }
        return cell
    }
}
```

Példa III.

```
// Használat: létrehozunkg egy olyan cellát, ami adaptálja a protokolunkat
class FriendTableViewCell: UITableViewCell, ReusableView {}

// Használat: A tableView delegate metódusában már a saját megoldásunkat
használjuk
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
    let cell: FriendTableViewCell = tableView.dequeueReusableCell(forIndexPath:
indexPath)
    ...
    return cell
}
```

Objective-C

Objective-C

- Teljes egészében C nyelvre épül, a C kibővítésének tekinthető
  - 100% C kompatibilis: bármely standard C kód lefordítható Objective-C fordítóval
- C-hez képest újdonság
  - Objektumorientált programozás
  - Dinamikus típuskezelés
  - Reflection
  - Elsőrendű függvények, blokkok
- Dinamikus nyelv: amit lehet futási időben végez, fordítási idő helyett (rugalmasság)

Foundation

- A *Foundation* framework tartalmazza a legalapabb Objective-C-s típusokat és függvényeket
- Ezt a frameworköt kivétel nélkül használja minden iOS alkalmazás
- Objective-C-ben van írva, mai napig aktívan fejleszti az Apple
  - Olyan régi kódbázis, hogy manuális memóriamenedzsmentet használ, nem ARC-t
  - Objective-C-ben nincsenek névterek, ezért szokás minden framework-öt 2-3 betűs prefix-el ellátni, ez Foundation esetén az **NS**
    - NextStep (valójában)
    - NotSwift (viccelnek vele)

Adattípusok: NSString

- Objective-C-ben minden string **NSString**
- Hasonlít a Swift Stringre, de van pár fontos különbség
  - > Az NSString egy osztály, nem struct: tehát referencia típus
  - > *UTF-16*-ban tárolja a stringeket
  - > Alapesetben *immutable*, de nem konstans
    - String módosításnál nem az eredeti string változik, hanem új referencia kerül beállításra
- Swift String és Objective-C NSString között egyszerűen tudunk *bridge*-elni
  - > Sok, Swiftben használt `String` metódus valójában az `NSString` metódusa

```
let nsAlma = "alma" as NSString
let alma = nsAlma as String
```

Adattípusok: NSString II.

- `NSString` létrehozása:

```
NSString *hello = @"Hello, world!";
```

- `NSString`-eket soha ne hasonlítsunk össze == használatával, ugyanis az Objective-C-ben csak referenciát néz
  - Használjuk az **isEqualToString** metódust!
- Létezik egy *mutable* párja is: `NSMutableString`
  - `NSString` kiegészítése

Adattípusok: számok

- `NSInteger`: primitív típus: tehát value type
  - Processzor architektúrától függően 32 vagy 64 biten tárolja a számokat
  - Objective-C-ben mindig használjuk ezt a C-s int helyett
- `NSNumber`: referencia típus rengeteg segédmetódussal
  - `NSDecimalNumber`: `NSNumber` leszárazott, használata kicsit kényelmetlen, de fontos, ha egészen pontos számokkal kell dolgoznunk

Adattípusok: tömbök

- Swifttel ellentétben többféle típust is tárolhatunk a tömbben
- Az **`NSString`**-nél látottakhoz hasonlóan itt is van immutable **`NSArray`** és mutable **`NSMutableArray`**

```
NSArray *villains = @[@"Weeping Angels", @"Cybermen", @"Daleks", @"Vashta
Nerada"];

for (NSString *villain in villains) {
    NSLog(@"Can the Doctor defeat the %@? Yes he can!", villain);
}
```

- Minden olyan NSArray metódus, ami indexet használ túlindexelés esetén azonnal crash-el

Adattípusok: NSDictionary

- **NSDictionary** és **NSMutableDictionary**

```
NSDictionary *ships = @{
    @"Orion": @"Raumpatrouille",
    @"Enterprise": @"Star Trek",
    @"Executor": @"Star Wars"
};
```

- Mivel Objective-C-ben nincsenek tuple-ök, ezért ha több adattal kell visszatérni valahonnan NSDictionary-t tudunk használni (vagy új típust készíteni)

Öröklés

- Minden osztálynak kötelezően egy ősosztálya van (a gyökér osztály az `NSObject`)
- Minden metódus felüldefiniálható (minden metódus virtuális)
- Többszörös "öröklés" csak protokollokkal (interfész)

```
// protokoll deklarációja és neve
@protocol MyPrinterProtocol

// protokoll metódusai
(void)printThisData:(NSData *)data;

@end
```

Az `id` típus

- Bármilyen típusú objektumra mutathat, a referenciák alaptípusa
  - Ha nincs értéke, akkor **`nil`**-re állítjuk
  - Alapból mutató típus, nem kell *-ot külön kiírni hozzá
  - Swiftben **`Any`**-re képződik
- Objective-C

```
id myRobot = nil;
```

- Swift

```
var myRobot: Any? = nil
```

- Visszatérési értéknek használjunk inkább **`instancetype`**-ot, ez biztosít pár compiler checket

Swift vs. Objective-C

- Swift: metódushívás

```
myRobot.say("We came with peace")
myRobot.attack(target: president, with: laserCannon)
myRobot.selfDestruct()
robotFactory.createRobot().say("We came with peace")
```

- Objective-C: „üzenetküldés”

```
[myRobot say:@"We came with peace"];
[myRobot attackTarget: president withWeapon: laserCannon];
[myRobot selfDestruct];
[[robotFactory createRobot] say:@"We came with peace"];
```

Objective-C osztályok használata Swiftből *Interoperability*

- Minden Swift projekthez beállítható egy **Objective-C Bridging Header**
- Bármely, ebben a fájlban beimportált Objective-C fájl elérhető a Swift forrásfájlokból
- A *targethez* kapcsolódó **Build Settingsben** állítható be
- ProjectName-Bridging-Header.h

Key-Value Observing (KVO)

Key-Value Observing

- Olyan observing mechanizmus, mely lehetővé teszi objektumok számára, hogy értesüljenek más objektumok valamely property-jének változásáról
  - Kifejezetten hasznos a Model és Controller rétegek közti kommunikációkor (A macOS Controller rétege nagyban a KVO-ra épít.)
- Előnyei:
  - Legfőbb előnye, hogy nem kell saját sémát implementálni, hogy értesítést küldjünk mindig, amikor egy property változik
  - Framework szinten támogatott, könnyen adoptálható, alapvetően nem jár (sok) plusz kóddal

KVO implementálása

- Az obszervált objektumnak KVO kompatibilisnek kell lennie
  - Az osztálynak az **NSObject**-ből kell leszármaznia
  - Swiftben: **@objc dynamic** annotáció a propertyre, **@objc** a class-ra, mivel a KVO az Objective-C runtime-on alapszik
- Egy obszerver instance menedzseli az egyes property-k változását
  - A obszerválást az **observe(_:options:changeHandler:)** metódussal indíthatjuk el
  - Ha szükségünk van arra, hogy hogyan változott meg a property, akkor az options paramétert kell megfelelően beállítani.
- Hogy értesüljünk a változásról, össze kell kötnünk a obszervert az obszerválttal
  - Ekkor az obszervált objektumok értesítik az obszerverüket, a property változásról

KVO példa

- Az alábbi osztály **myDate** property-je obszerválható

```
@objc class MyObjectToObserve: NSObject {
    @objc dynamic var myDate = NSDate(timeIntervalSince1970: 0) // 1970
    func updateDate() {
        myDate = myDate.addingTimeInterval(Double(2 << 30))
        //Első hívás után: 2038-01-19
    }
}
```

KVO példa II.

- Obszerver példa osztály:

```
class MyObserver: NSObject {
    var objectToObserve: MyObjectToObserve
    var observation: NSKeyValueObservation?

    init(object: MyObjectToObserve) {
        objectToObserve = object
        super.init()

        //A \. a keypath eléréhez kell
        observation = objectToObserve.observe(\.myDate, options: [.old, .new]) {
           object, change in
              print("myDate changed from: \(change.oldValue!), to: \(change.newValue!)")
        }
    }
}
```

https://docs.swift.org/swift-book/ReferenceManual/Expressions.html#grammar_key-path-expression

KVO példa III.

- Obszerver összekötése az obszerválttal

```
let observed = MyObjectToObserve()
let observer = MyObserver(object: observed)
```

- Az `updateDate` metódus megváltoztatja a `myDate` property értékét, amely automatikusan triggereli az obszerver change handlerét

```
observed.updateDate()
//myDate changed from: 1970-01-01 00:00:00 +0000, to: 2038-01-19 03:14:08 +0000
```