

Swift

Kelényi Imre

imre.kelenyi@aut.bme.hu



Department of
Automation and
Applied Informatics



AutSoft
BME

Swift dióhéjban



- iOS/OS X fejlesztés új nyelve
- Script nyelv szerű, modern lehetőségekkel, kompakt szintaxissal
- Natív kódra fordul
 - > Mint C, C++ vagy Objective-C
- Sok más nyelvből merít: Python, Haskell, Ruby, C#
 - > *Objective-C without the C*
- Könnyen összekapcsolható Objective-C-ben írt kóddal (interoperability)
- Elsőre egyszerűnek tűnik, de sok funkciója kifejezetten komplex és ezek helyes elsajátítása nem kevés időt igényel

A Swift mindent megváltoztat?

Egy app leprogramozásához kapcsolódó feladatok “mennyisége”

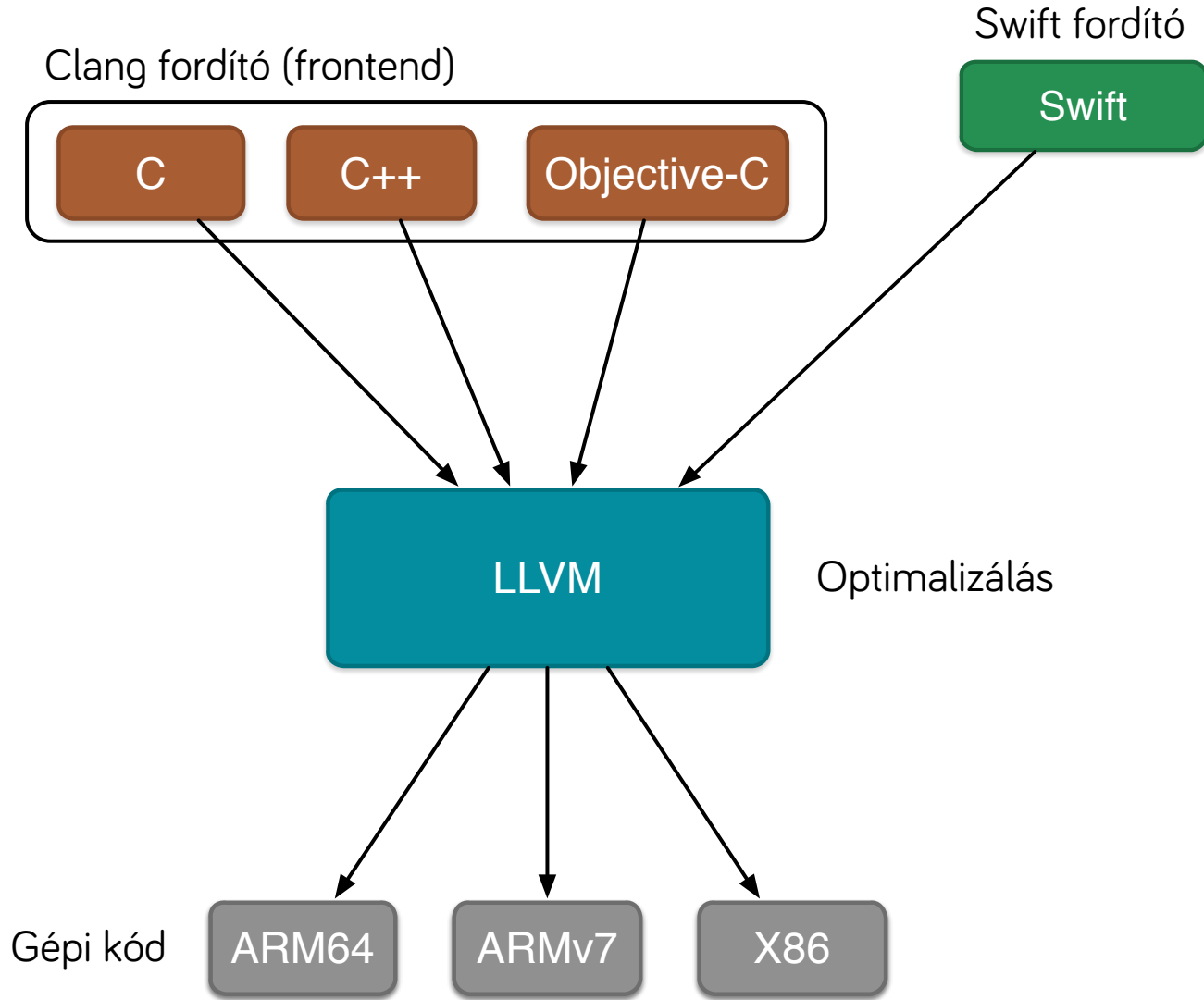
Ennyi lett más a
Swift miatt...

Csak a programozási nyelvet érintő
feladatok

Frameworkök (UIKit, Core Data, stb...) és
library-k használata

Továbbra is a különböző
frameworkök megtanulása
és használata fogja
kitenni a fejlesztés
legjelentősebb részét...

Swift fordító?



Miért kell a Swift?

- Objective-C-hez képest sokkal tömörebb kód
 - > Gördülékenyebb kódolás
- Sokkal szigorúbb szabályok: hibák már fordítási időben kibuknak
 - > Statikus típusos
 - > Optionals
 - > Generics
 - > ...
- Modern, új programozók és veterán ObjC-sek számára is vonzó
- Teljesen az Apple kontrollja alatt

Kompatibilitás

- OS verziók
 - > iOS 7.0+
 - > OS X 10.9 (Mavericks)+
- Fejlesztéshez Xcode 6.0+
- Az összes standard Apple API elérhető Swiftből

Evolúció



- A Swift az első béták (nyár) óta rengeteget változott / fejlődött
 - > Teljesítmény
 - > Szintaxis és nyelvi elemek
- 2014. Szeptember 9. - Swift 1.0
 - > A nyelv folyamatosan fejlődni fog
 - > Xcode 6.1: “failable initializers” és még sok változás
- Az App Store fogadja a Swiftben írt appokat
- Apple: 4000+ iOS 8 API-t már Swiftben írtak

Swift alapozás

Szintaxis

- Bármilyen UNICODE karakter használható az azonosítókban
 - > Maradjunk csak az angol ABC betűinél...
- Pontosvesszők opcionálisak
- Zárójelek, a legtöbb esetben, opcionálisak
- Kód blokkoknál mindig kötelezők a kapcsos zárójelek (egy utasítás esetén is) :

```
if i > 10 {  
    println("That's more than nothing")  
}
```

- Általános tanács: használjuk ki a Swift nyújtotta lehetőségeket és törekedjünk a legtömörebb kódra (kivéve ha a megértés vagy helyes működés megkívánja a “bővebb” kódot)

Változók és konstansok

- Konstans: egyszer adhatunk értéket
 - > **let** kulcsszóval deklarálva
let PI = 3.14159265
PI = 3 // ERROR: PI értéke nem változhat
- Változó: megváltozhat az értéke
 - > **var** kulcsszóval deklarálva
var i = 12
i += 1 // OK i egy változó

A Swift erősen típusos

- Sehol sincs automatikusa (implicit) típus konverzió (pl. Int és Double között vagy Bool-ra valamilyen számértékről)
- Aritmetikai operátorok csak azonos típusú attribútumokkal működnek

```
let doubleNum = 7.13
```

```
let intNum = 3
```

```
let result = doubleNum + intNum – ERROR
```

```
let result = doubleNum + Double(intNum)
```

- Minden konverziót explicit jelölni kell, “cast”-olással, pl. Double(intNum)

Type Inference

- A Swift **statikusan típusos**: minden változónak/konstansnak deklarálástól kezdve konkrét típusa van, ami később nem változhat
- **Type inference**: a Swift megpróbálja kikövetkeztetni a változó típusát a kezdeti értékből

| | | |
|------------------------------------|---|---------|
| > var str = "Hello, playground" | ← | String |
| > var d = 3.14 | ← | Double |
| > let green = UIColor.greenColor() | ← | UIColor |

- Explicit is megadhatók a típusok:

```
var color: UIColor?
```

Alaptípusok

- Integer : **Int**
 - > Ahol csak lehet, használjunk sima Int-tet! Csak ott használjunk specifikusabb típusokat, ahol kifejezetten igény van rá
 - > Előjel nélküli integer: UInt, explicit méretű integer: Int8, Int64, UInt8, ...
- Lebegőpontos számok: **Double** vagy **Float**
 - > Double 64-bites, Float 32-bites
 - > Az alapértelmezett lebegőpontos típus a Double

```
var d = 3.14
```
- Boolean: **Bool**
 - > Két lehetséges érték: **true** vagy **false**

String

```
var str: String = "Hello"
```

- String összefűzés

```
str = "Hello " + "HWSW"
```

- Append to strings

```
str += ". How are you?"
```

- Sztring interpoláció

```
studentsStr =  
    "The number of students is \$(studentCount)"
```

- Karakterszám: `countElements(str)`
- A String érték típus (**value type**): mindig egy másolat készül és adódik át

Gyűjtemények (generikus tárolók)

- Array

```
var animals: [String] =  
    ["elephant", "tiger", "dolphin"]
```

- > Egy tömbön belül csak **ugyanazon típus** elemeit tárolhatjuk (kiskapu: AnyObject vagy Any tömb)

- Dictionary

```
var applicants: [String: Int] =  
    ["Martin" : 23, "Laura" : 33]
```

- > Kulcs-érték párokat tárol
- > Mind a kulcs, mind az értékek típusa rögzített
- > A kulcsnak meg kell valósítani a **Hashable** protokollt

Vezérlési szerkezetek

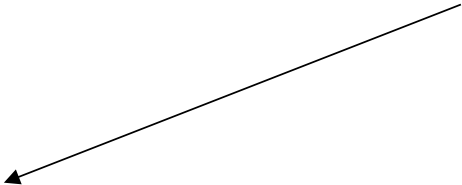
A kapcsos zárójelek
{ } használata
mindenhol kötelező!

- **for** and **for-in** loops
- **while** and **do-while**

```
do {  
    i++  
} while i < 10
```

- **if else**
- **switch**

```
switch type {  
    case .Customer:  
        println("Hello our dear customer!")  
        ...  
    default:  
        println("Are you an alien?")  
}
```



switch

- automatikus break a következő case ág előtt
- **fallthrough** és **break** utasítások
- minden lehetséges értéket le kell fedni case ágakkal (vagy felvenni **default:** ágat)

Optional

- Adattípus melynek nem biztos, hogy van értéke (képes kifejezni az érték hiányát)
 - > `var points: Int? = nil`
 - Vagy egy `Int`-et vagy `nil`-t (= “nincs érték”) tartalmaz
- A `?` operátorral deklaráljuk
 - > Bármilyen Swift típus optional-lá tehető egy “?” a típus neve után írásával
 - `Int?`, `String?`, `UIColor?` ...
- Egyik legalapvetőbb Swift mechanizmus
 - > Ha egy property-nek nincs mindig értéke
 - > Ha egy metódus nem mindig tér vissza konkrét értékkel
 - > ...

Opcionális értékek használata

```
var points: Int? = nil
```

- Mielőtt felhasználhatnánk egy optional típusú értéket, először ellenőrizni kell, hogy van-e értéke:

```
if points != nil { }
```

- > Ha egy üres (nil) értékű optional-t próbálunk használni, **az alkalmazás crash-el**

- Az optional típusú értékeket ki kell csomagolni (**unwrap**)

- > **Force unwrapping** operátor: !

```
println("The value of points: \(points!)" )
```

- > Ellenőrzés és kicsomagolás **optional binding**-gal:

```
if let pointsValue = points { }
```

- > Optional chaining

Optional Chaining (? operátor)

- Biztonságos és kényelmes módja egy esetleg nil értékű optional egy metódusának meghívására vagy érték átadására
- A ? operátorral “kicsomagolva” az optionalt, majd erre meghívva egy metódust vagy lekérve egy property-t: ha az optional nil, a művelet nem hatódik végre (nincs crash)

```
var str: String? // str egy optional
```

```
let capitalizedStr = str?.capitalizedString
```

Ha str nil, az utasítás nem hajtódik végre

Egy “optional chain” visszatérési értéke mindig egy optional lesz (itt String? mert capitalizedString String-et ad vissza)

- Az ellenőrzés + force unwrapping (! operátor) alternatívája

Optional Chaining: Példák

- More than one method/property call can be chained; the entire chain fails gracefully

```
let hashValue = str?.toInt()?.hashCode
```



`toInt()` is called only if the *previous link in the chain* (accessing `str`) succeeds and returns not nil

- Works with calling methods, subscripts and setting property values

```
var labels = [String: UILabel]()  
labels["redLabel"]?.text = "This is red"
```



The `text` property is set only if `labels["redLabel"]` returns a value, otherwise the assignment is cancelled

Implicitly Unwrapped Optional

- Ha egy olyan propertyre van szükségünk, mely definiáláskor még üres, de első használata előtt biztos értéket kap
- **Implicitly unwrapped optional:** egy olyan opcionális érték, mely használatkor automatikusan “kicsomagolódik” (nem kell !)
 - > A fordító nem panaszkodik, hogy nincs kezdeti értéke
 - > Miután később értéket adtunk neki, úgy használható mint egy standard (nem opcionális) érték
- A **!** jelet kell hozzáadni egy típushoz (? helyett)
 - > `var someText : String!`
- FONTOS: nil értékű “implicitly unwrapped optional” elérésekor továbbra is elszáll az app!

Függvények

- Függvényeket a **func** kulcsszóval deklarálunk

```
func countConsonants(str: String) -> Int { }
```

- Külső paraméter nevek

```
func greetStudents(students: [String],  
    withGreeting greeting: String) { }
```

```
greetStudents(students, withGreeting: "Szia ")
```

- Paraméterekhez megadható alapértelmezett érték

```
func greetStudents(students: [String],  
    withGreeting greeting: String = "Hi ")
```

Osztályok és struktúrák

Class és Struct

```
class SomeClass { }
```

```
struct SomeStructure { }
```

- Újrafelhasználható, általános célú típusok **property-ekkel** és **metódusokkal**
- **class** és **struct** közötti különbséges
 - > **struct: value types** (mindig másolat adódik át), **class: reference type** (referencia adódik át)
 - > csak az osztályok öröklődhetnek (de protokollokat struct is megvalósíthat)
 - > csak az osztályoknak lehet “**deinicializálója**” (destruktor)
 - > futásidejű típusellenőrzés csak osztáloknál (as, is)

Osztályok deklarációja

```
class Book {
```

```
    var title: String
```

```
    var pageCount: Int?
```

Property-k **var** vagy **let**
kulcsszóval deklarációk

```
    init(title: String, pageCount: Int? = nil) {
```

```
        self.title = title
```

```
        self.author = author
```

```
        self.pageCount = pageCount
```

```
    }
```

Inicializálók (~konstruktor)
felelősek az osztály
példányainak kiindulási
állapotának beállításáért.
Mindig **init** kulcsszóval.

```
    func print() {
```

```
        println("\t(author): \t(title) - publish in \t(publishYear)")
```

```
    }
```

```
}
```

Metódusok **func**-kal
deklaráció

Osztályok példányosítása

- Osztálynév + ()

```
var myColor = UIColor()
```

- A paramétereknek meg kell felelniük az osztály egyik inicializálójának

```
var rating = Rating(user: "Fanboy", stars: 5)
```

```
init(user: String, stars: Int)
```

A double-headed arrow points from the 'Rating' part of the first code line to the 'init' part of the second code line, indicating that the first line is an instantiation of the class defined in the second line.

Tárolt és kiszámított property-k

- **Stored property:** az értéke az osztály memóriaterületén tárolódik (mint változóknál)

```
var title: String
```

- **Computed property:** egy metódus pár, mely meghívódik ha a property értékét lekérdezzük vagy megváltoztatjuk

```
var titleWithAuthor: String {  
    get {  
        return "\ (author): \ (title)"  
    }  
}
```

Metódusok

- Az osztály törzsén belül **func**-kal deklarált függvények
 - > **self**-el hivatkozhatunk az objektumra, melyre a metódust meghívták

```
func addRatingWithStars(stars: Int, user: String){  
    self.ratings.append(  
        Rating(user: user, stars: stars))  
}
```

- Metódushívásnál alapesetben **az első paraméter neve nem kerül kiírásra**

```
myBook.addRatingWithStars(5, user: "Murakami")
```

Inicializálók – init

- Példányosításkor az osztály minden property-jét kötelező inicializálni
 - > Vagy a property deklarációjával együtt a kezdeti értékének megadásával

```
var stars: Int = 0
```

- > Vagy inicializálás során, az **init** metódus(ok)ban:

```
init(stars: Int) {  
    self.stars = stars  
}
```

- Egy osztálynak tetszőleges számú inicializálója lehet (a paramétereiknek legalább névben el kell térniük)
- Példányosításkor alapesetben az inicializáló minden paraméterének nevét ki kell írni

```
var rating = Rating(stars: 5)
```

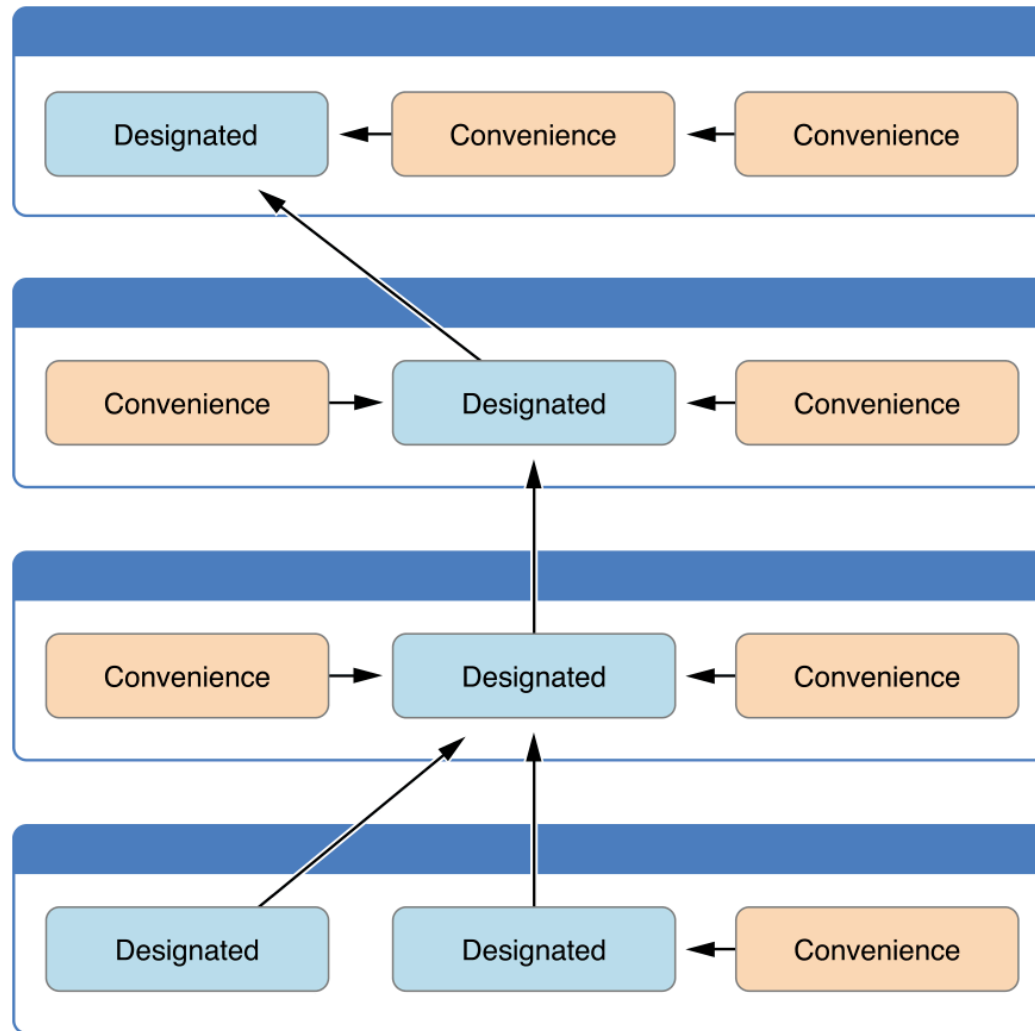
Kétfázisú inicializáció

- Swift-ben az inicializáló metódusok tartalma 2 fázisra osztható, melyet a fordító is szigorúan betartat
 - > **1. fázis:** az osztály minden property-jét inicializálni kell
 - > **2. fázis:**
 - Öröklés esetén az inicializálónak meg kell hívnia az őssztály egy inicializálóját (designated initializer – lásd következő dia)
 - Az őssztályból öröklött property-k vagy metódusok használata
- Az 1. fázis lezárultáig nem használhatók az őssztály metódusai
- Az őssztály inicializálóját kötelezően az első fázis végén kell meghívni

Designated vs. convenience init

- **init()**: designated initializer
 - > “standard” inicializáló, minden property-t inicializálni kell és meghívni az őssztály egyik designated inicializálóját
- **convenience init()**: convenience initializer
 - > Az osztály egy másik convenience vagy designated inicializálóját kell, hogy meghívja (őssztály inicializálót nem hívhat!)
 - > A convenience inicializálók láncának végül egy designated inicializáló hívásban kell végetérni

Inicializáló lánc



Láthatóság

- 3 láthatósági osztály
 - > **internal**: látható a teljes modulban (alkalmazás vagy framework), ez az **alapértelmezett láthatóság**
 - > **public**: látható külső modulokban
 - > **private**: csak abban a .swift fájlban látható, ahol deklarálták
- A láthatósági osztályok a globális scope-ban definiált típusokra (pl. class, struct) is vonatkoznak
 - > Pl. egy frameworkben külön public-ra kell állítani azokat az osztályokat, amiket más modul számára is elérhetővé szeretnénk tenni

Paraméter nevek kiírása

- **Függvények:** alapesetben egyik paraméternevet se kell kiírni

```
greetStudents(students, "Szia ")
```

- **Metódusok:** az első paraméter kivételével minden paraméternév kiírandó

```
myBook.addRatingWithStars(5,  
    user: "Murakami")
```

- **Inicializálók:** minden paraméter nevét ki kell írni

```
var myBook = Book(title: "The Road",  
    author: "Cormac McCarthy")
```

Öröklés

- Csak egyszer öröklés megengedett (egy őosztály)

```
class RectangleView: UIView
```

- Alapból minden metódus + property öröklődik (private kulcsszó használható a file szintű láthatósághoz)
- Őosztály metódusai és property-jei felüldefiniálhatók (kivéve **final**)

```
override fun drawRect(rect: CGRect) {  
    super.drawRect(rect)  
    // rajzoljunk valamit  
}
```

a metódus őosztálybeli implementációját a **super**-rel érjük el

Closure-ök

Függvények, Metódusok és Closure-ök

- (Globális) függvények
- Metódusok
 - > Egy olyan függvény, mely mindig egy adott típushoz (class, struct, enum) tartozik
- Closure
 - > Egy anoním függvény / lambda expression
 - > Automatikusan “begyűjtik” a behivatozott változókat

Minden függvény/metódus/closure referencia típus és **elsőrendű eleme a nyelvnek**: paraméterként átadható, változóhoz rendelhető, visszatérési érték lehet, stb.

Closure

- A closure egy kód blokk, melyet referencia típusként használhatunk (értékül adhatjuk változóknak, átadhatjuk függvényeknek, stb.)
 - > Egy függvények is closure-ök
 - > Anonim closure: nincs külön azonosítója
- Legtöbbször függvény paraméterként adjuk át őket
 - > Eseménykezelés
 - > Egy algoritmus definiálása és átadása
 - > Animáció

```
let sortedStrings = animals.sorted( { (a, b: String) -> Bool in  
    return a < b  
} )
```

Closure szintaxis

- Legbővebb szintaxis

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

- Tömörítsünk!
 - > Paramétertípusok és visszatérési érték elhagyható, ha kikövetkezhető a closure felhasználási területéből
 - > Paraméternevek helyett: **\$0, \$1,...**
 - > Egy utasításból álló closure-nél a **return** kulcsszó is elhagyható

Záró closure szintaxis

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

> Standard szintaxis

```
sorted(animals, { $0 < $1 })
```

> Záró closure szintaxis

```
sorted(animals) { $0 < $1 }
```


Változók hivatkozás closure-ből

- A closure-ből hivatkozhatók a closure környezetében látható változók
 - > A closure alapból minden felhasznált változóra erősen hivatkozik (**strong reference**)
 - > A hivatkozások egész addig megmaradnak, amíg a closure létezik

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

- A closure-ökben behivatkozott változók könnyen hivatkozási köröket (reference cycle) okozhatnak
 - > Megadható, hogy egyes változókat gyengén hivatkozzunk (**weak reference**)

Egyéb témák

Framework

- Swift framework
 - > Egy külön modul, melyben osztályokat, függvényeket definiálhatunk
 - Külön névteret kap, pl. **CoreLocation.CLLocationManager**
 - > Bináris formában terjeszthető (nem lophatják el a kódját)
 - > Továbbra is csak egy alkalmazás sandboxából elérhető
 - Két különálló alkalmazásnál mindkét apphoz csatolni kell
 - App Extension esetén az extension és a konténer app közötti kód megosztásnál hasznos

Enum

```
enum PartnerType {  
    case Customer  
    case Employee  
    case Owner  
}
```


```
var type: PartnerType = PartnerType.Customer
```

- Az Enum értékek mögött tetszőleges “nyers/mögöttes” típus állhat (**raw value**), nem csak Int
- Enum értékekhez hozzárendelhetők társértékek (**associated value**)

```
case Employee(jobgrade: Int)
```

- Enumok kiterjeszthetők metódusokkal és property-kkel, sőt még protokollokat is megvalósíthatnak! (mint az osztályok)

Swift és Objective-C egy projektben

- Alapvetően a két nyelvben írt osztályok vegyíthetők egy projekten belül is
 - > Az ObjC API-k átkonvertálódnak Swift-re
 - `(void)insertString:(NSString *)aString atIndex:(NSUInteger)loc;`

`func insertString(aString: String, atIndex loc: Int)`
 - > A Swift osztályok interfésze átkonvertálódik ObjC-re, **kivéve a Swift specifikus funkciókat** (pl. generics, tuples)
- 3rd party ObjC libraryk gond nélkül használhatók
- Cocoapods működik Swift projektekben

Teljesítmény

- A Swift a legtöbb numerikus műveletekben gyorsabb az Objective-C-nél
- 100000 véletlen pozitív egész szám (UInt32) rendezése:

| $T = 10$ $N = 100,000$ Release | Std lib sort | Quick sort $O(n \log n)$ | Heap sort $O(n \log n)$ | Insertion sort $O(n^2)$ | Selection sort $O(n^2)$ |
|--------------------------------------|--------------|-----------------------------|----------------------------|----------------------------|----------------------------|
| Objective-C -03 | 0.151701 s | 0.121619 s | 0.251310 s | 175.421688 s | 349.182743 s |
| Swift -O | 0.013933 s | 0.015712 s | 0.036932 s | 27.532488 s | 18.969978 s |
| Difference | 10.9x | 7.7x | 6.8x | 6.4x | 18.4x |

<http://www.jessesquires.com/apples-to-apples-part-two/>

Playground és REPL ~ Swift homokozó

- Playground dokumentum
 - > Interaktív Swift környezet
 - > A kód automatikusan fordul és minden sorhoz/parancshoz kiértékeli/kiírja az eredményét
 - > Tanuláshoz, kódrészletek kipróbálásához, új osztályok prototípusának megírásához
- REPL (Read-Eval-Print-Loop)
 - > Parancssorból vagy debuggolás közben elérhető interaktív Swift környezet
 - > Debuggerre (LLDB) épül
 - > Egyszerűen injektálható a kód



```
1> var i = 4
i: Int = 4
2> println("i is \(i)")
i is 4
```

Tanuláshoz

- Apple ingyenes ebookok
- raywenderlich.com
 - > Ingyenes webes Swift tutorialok:
<http://www.raywenderlich.com/tutorials>
 - > Swift by Tutorial könyv (\$54)
- objc issue #16 (Swift):
<http://www.objc.io/issue-16/>
- <http://nshipster.com/>