# iOS alapú szoftverfejlesztés - Labor 02

A laborsegédletet összeállította: Kelényi Imre - imre.kelenyi@aut.bme.hu

A labor témája:

* Swift alapismeretek
* Playgorundok használata
* Alaptípusok
* Optionals
* Osztályok, property-k és metódusok

### Cheat sheets (még nem frissültek Swift 2.0-ra)

* <https://github.com/iwasrobbed/Swift-CheatSheet>
* <http://fuckingfunctionsyntax.com/>
* <http://www.raywenderlich.com/73967/swift-cheat-sheet-and-quick-reference>

Egy lehetséges "style-guide" és kódolólási konvenció szabályzat (nem szentírás!):

<https://github.com/raywenderlich/swift-style-guide>

## Swift alapismeretek

Hozzunk létre egy új **.playground** fájlt! A playgroundban egy interaktív fejlesztőkörnyezet, melyben minden sor/kifejezés értéke automatikusan kiértékelődik és megjelenik a jobb szélső panelon.

Bár úgy tűnhet mintha egy interpretált nyelvvel dolgoznánk, valójában a minden módosítás után újrafordul a teljes playground.

A következőkben a playgroundban próbáljuk ki a Swift legfontosabb funkcióit.

### Osztályok definiálása és példányosítása

Hozzunk létre egy új, GameCharacter nevű osztályt, mely egy "játékos" adatait zárja egységbe:

class GameCharacter {

var name: String

var level: Int

}

Hibát kapunk, mert az osztálynak nincs minden property-je inicializálva. A Swift fordító kötelezően előírja, hogy az osztály példányosításakor minden property inicializálva legyen.

Hozzunk létre egy új inicializálót, mely bekéri a karakter nevét és szintjét:

init(name: String, level: Int) {

self.name = name

self.level = level

}

Fontos kiemelni, hogy a **self** általában elhagyható, azonban itt most mind az inicializáló paramétereinek, mind az osztály property-jeinek ugyanaz a neve, ezért muszáj kiírni, ha a property-kre szeretnénk hivatkozni!

Hozzunk létre egy új hőst a GameCharacter osztály példányosításával:

let hero1 = GameCharacter(name: "Harcos Huba", level: 1)

Figyeljük meg a következőket:

* Inicializálók hívásánál (vagyis példányosításkor) alapesetben minden paraméter nevét ki kell írni
* mivel hero1-et **let**-tel definiáltuk, ez egy konstans és nem változtatható az értéke (azonban a hivatkozott objektumnak ettől még módosíthatjuk a property-jait)
* Option+Click-el vizsgáljuk meg hero1 típusát, láthatjuk, hogy hála a *type inference*-nek, egy GameCharacter típusú változót kaptunk.

A Swift **statikusan típusos nyelv**: minden változónak van típusa a definiálásuktól kezdődően és ez a típus nem is változhat.

A változók típusát azonban a legtöbb esetben nem kötelező explicit megadni, mert a fordító kitalálja azt a változó/konstans kezdeti értékéből. Ezt a mechanizmust hívjuk **type inference**-nek.

### Opcionális értékek (optionals) és failable initializer

Módosítsuk az inicializálót oly módon, hogy ha a megadott szint nem esik értelmes határok közé, ne jöjjön létre az objektum (vagyis térjünk vissza nil-el). Ezt a mechanizmus (vagyis, hogy init? -el definiálunk egy inicializálót) **failable initializer**-nek nevezik:

init?(name: String, level: Int) {

self.name = name

self.level = level

**if level >= 100 || level < 0 {**

**return nil**

**}**

}

Nézzük meg, hogy mi történik, ha rossz értéket adunk meg a példányosításkor:

let hero1 = GameCharacter(name: "Harcos Huba", level: -2)

* Option+Klikk-el megnézve hero1 típusa már nem GameCharacter, hanem **GameCharacter?** vagyis egy "GameCharacter optional"
* nil-t kapunk vissza értékül

Hozzunk létre még egy hőst és próbáljuk meg módosítani egy property-jét:

let hero2 = GameCharacter(name: "Varázsló Vilmos", level: 3)

hero2.level = 30 // HIBA

A hiba oka, hogy opcionális típusokon közvetlenül nem hívhatjuk meg az általuk becsomagolt objektum műveleteit, csak ha előtte "kicsomagoljuk" őket!

Ezt többféleképpen is megtehetjük:

if hero2 != nil { // Mindig ellenőrizzük le az optional tartalmát a ! operátor használata előtt

hero2!.level = 30 // "! operátor: force unwrap"

}

hero2?.level = 30 // "optional chaining" ha hero2 nil, akkor nem hajtódik végre a művelet

if let unwrappedHero = hero2 { // "if let binding"

unwrappedHero.level = 30

}

Ha a force unwrap operátort (!) használjuk, akkor előtte minden esetben ellenőrizzük le, hogy a kicsomagolni készült option érték nem nil-e. Ha nil értékű optionalt próbálunk kicsomagolni, az alkalmazás el fog szállni!

Állítsuk vissza hero1-nél az inicializálónál használt "szintet" egy érvényes értékre és

let hero1 = GameCharacter(name: "Harcos Huba", level: 0)

### Generikus tárolók, metódusok és osztályhierarchiák

Hozzunk létre egy Party nevű osztályt, mely egy csapatnyi játékos-karaktert gyűjt egybe:

class Party {

private var members = [GameCharacter]()

func addMember(member: GameCharacter) {

members.append(member)

}

}

* A csapatba tartozó karaktereket a **members** property tárolja, melynek típusa [GameCharacter] vagyis egy tömb, mely GameCharacter példányokat tartalmaz. Ennek a propertynek **kezdeti értéket adunk**: egy üres tömböt.
* Az addMember metódus felvesz egy új karaktert a csapatba

A private láthatóság Swiftben azt jelenti, hogy csak az adott forráskódfájlban érhető el az így megjelölt elem. Ez playgroundok esetén nem sokat jelent, hisz minden kódot "egy forrásfájlba írunk". Ha az osztályaink külön fájlokban lennének definiálva (pl. GameCharacter.swift és Party.swift), akkor már nem tudnák elérni egymás privát mezőit.

let heroes = Party()

heroes.addMember(hero1!)

heroes.addMember(hero2!)

Figyeljük meg, hogy metódusok hívásánál (alapesetben) a legelső paraméter nevét nem kell explicit megadni! Ez egy konvenció, amit felül lehet definiálni, de alapvetően (Objective-C-s történelmi okok miatt) a legtöbb Apple API is úgy van megírva, hogy a legelső paraméter neve gyakorlatilag kitalálható a metódus nevéből, ezért azt "felesleges" ismét kiíratni.

Készítsünk egy metódust, mely végigiterál a csapaton és kiírja a nevüket:

func printMembers() {

for member in members {

print(member.name)

}

}

heroes.printMembers()

Hozzunk létre Party-ban egy új metódus, mely megmondja, hogy egy adott karakter tagja-e a csapatnak:

func isMember(member: GameCharacter) -> Bool {

return contains(members, member)

}

Fordítási hibát fogunk kapni. A probléma az, hogy a contains függvény működéséhez szükség van arra, hogy a tömb elemeiről el lehessen dönteni, hogy azonosak-e vagy sem. Azonban a GameCharacter osztály jelenleg erre nem képes. Swift-ben az Equatable protokollt (interfészt) kell megvalósítani azoknak az osztályoknak, melyek vállalják, hogy megvalósítják a == operátort (mellyel az érték szerinti azonosságot vizsgáljuk).

Swiftben a == az egyenlőség ("equal to") operátor, ezt használják Swift standard library függvényei (pl. contains(), find(), stb.). A == operátort egy megfelelő szintaktikájú globális függvény megvalósításával lehet definiálni (ezt írja elő az Equatable protokoll is), pl.: **func ==(left: GameCharacter, right: GameCharacter) -> Bool**

A === operátor az azonosság ("identity") operátor, mely akkor tér vissza igaz értékkel, ha az összehasonlított két érték ténylegesen ugyanaz az objektum.

Esetünkben lényegében referencia szerinti azonosságot szeretnénk vizsgálni (vagyis két játékos akkor azonos, ha tényleges ugyanarról az objektumról beszélünk). Ilyen esetekben a legegyszerűbb megoldás, ha leszármaztatjuk az ősosztályt NSObject-ből, mely alapból tartalmazza ezt a viselkedést:

class GameCharacter: NSObject {

Ekkor azonban az inicializálóban kapunk hibát, mert nem hívjuk meg az ősosztály (NSObject) konstruktorát. Javítsuk:

init?(name: String, level: Int) {

self.name = name

self.level = level

super.init()

**NSObject** a Foundation framework alaposztálya. Objective-C-ben minden osztálynak NSObject-ből kell származnia (legalább közvetett módon). Swift-ben ez már opcionális, de később látni fogjuk, hogy sok API-nál elvárás, hogy valamilyen NSObject leszármazott osztályú objektummal dolgozzon, ezért viszonylag ritkán fogunk olyan osztályokat látni, melyeknek nem szerepel az öröklési hierarchiájában.

Most már kipróbálhatjuk, hogy az isMember metódust:

if heroes.isMember(hero1!) {

print("\(hero1!.name) a csapatban van")

}

Vegyünk fel egy új propertyt GameCharacter-hez a karakterek életerejének tárolására és inicializáljuk 100-ra:

var hitPoints = 100

Ezen felül készítsünk egy **computed property**-t, mely gyorsan megadja, hogy a karakter még az élők sorát gazdagítja-e:

var isDead: Bool {

get {

return hitPoints <= 0

}

}

Érdemes megjegyezni, hogy ha csak gettert írunk egy computed property-hez, akkor még a **get** is elhagyható:

var isDead: Bool {

return hitPoints <= 0

}

Definiáljunk egy computed property-t, mely visszatér a karakter "támadóerejével". Ezt számítsuk a karakter szintje alapján:

var power: Int {

return level \* 10

}

### Öröklés és Cast-olás

Hozzunk létre egy Hero nevű osztályt, mely GameCharacter-ből származik. "Hero" egy olyan karakter, melynél lehet egy fegyver (de nem minden esetben van fegyvere). A fegyvert egy enum-mal jelképezzük:

class Hero: GameCharacter {

enum WeaponType {

case LaserCannon

case Spoon

}

var weapon: WeaponType?

}

Nyilván azok a hősök, melyeknél fegyver van, nagyobb támadóerővel rendelkeznek. Definiáljuk felül a GameCharacter ősosztály power property-jét:

override var power: Int {

get {

var extraPower = 0

if weapon != nil {

switch weapon! {

case WeaponType.LaserCannon:

extraPower = 100

case WeaponType.Spoon:

extraPower = 1

}

}

return super.power + extraPower

}

}

Swiftben metódusok és property-k (legyen akár stored akár computed property) egyaránt felüldefiniáhatók a leszármazott osztályokban (kivéve ha **final**-ként vannak megjelölve). Az override kulcsszó kiírás kötelező.

### Protokollok

Írjunk egy protokollt (más nyelvekben *interfész*), mely tartalmazza a "harcoláshoz" szükséges metódus és property sablonokat. Lényegében azt szeretnénk elérni, hogy minden olyan osztály, mely megvalósítja ezt a protokollt, részt vehessen egy csatában.

protocol Fightable {

var isDead: Bool { get }

var power: Int { get }

func takeDamageFromEnemy(enemy: Fightable)

}

Írjunk egy metódust, mely lejátszik egy ütközetet két "harcoló fél" között. A két fél addig támadja egymást felváltva, míg az egyik fél el nem esik (isDead property true-val tér vissza). A metódus térjen vissza a győztessel, vagy nil-lel, ha mindkét fél elesik az utolsó körben.

func fight(fighter1: Fightable, fighter2: Fightable) -> Fightable? {

while !fighter1.isDead && !fighter2.isDead {

fighter1.takeDamageFromEnemy(fighter2)

fighter2.takeDamageFromEnemy(fighter1)

}

if !fighter1.isDead {

return fighter1

}

else if !fighter2.isDead {

return fighter2

}

return nil

}

Valósítsuk meg a Fightable protokollt GameCharacter osztállyal!

class GameCharacter: NSObject, Fightable {

Hibát kapunk, mert még nem definiáltuk a protokollban felsorolt összes metódust vagy propertyt. Ténylegesen csak a takeDamageFromEnemy() metódus hiányzik, vegyük fel:

func takeDamageFromEnemy(enemy: Fightable) {

hitPoints -= enemy.power

}

### A Swift erősen típusos (strongly typed) nyelv

Módosítsuk a takeDamageFromEnemy metódust oly módon, hogy kisorsolunk egy véletlen % értéket (10% és 100% között) és ezzel megszorozzuk a támadó erejét.

func takeDamageFromEnemy(enemy: Fightable) {

let attackRating = Double(arc4random\_uniform(10) + 1) / 10

hitPoints -= Int(Double(enemy.power) \* attackRating)

}

Swiftben alapesetben csak azonos típusú objektumokon lehet műveleteket végezni, ezt hívjuk erős típusosságnak. Minden olyan esetben, mikor eltérő típusú objektumokat próbálunk kombinálni vagy rajtuk valamilyen operátort használni, explicit konvertálással (pl. Double(someVar)) azonos típusra kell hoznunk őket.

Mindez még az eltérő bitszámú vagy nemnegatív/előjeles egész számokra is igaz, tehát pl. UInt32 és Int típusok között is konvertálni kell!

## Önálló feladatok

Írjunk egy Monster nevű osztályt, mely megvalósítja a Fightable protokollt! A Monsterrel szemben támasztott elvárásaink a következőek:

* Monster rendelkezzen egy headCount: Int propertyvel, mely a szörny, még meglévő felyeinek száma.
* A szörny támadóereje (power) a fejeinek száma \* 20
* A szörny akkor hal meg ha nincs már egy feje sem
* Mikor a szörnyet megtámadják, power / 10 eséllyel leesik egy feje (tehát 1-es power esetén 10%, 10-es vagy nagyobb power esetén 100%)

Véletlen egész számok generálásához használjuk az arc4random\_uniform(max) függvényt, mely [0..max) intervallumban fog visszaadni véletlen egész számokat.

Játsszunk le néhány ütkezetet különböző Monster és Hero példányok között!