SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tin Tomašić

Studij: Informacijski i poslovni sustavi

Modul: Razvoj programskih sustava

Kolegij: Razvoj aplikacija za mobilne i pametne uređaje

NAPREDNI KONCEPTI PROGRAMSKOG JEZIKA SWIFT

Varaždin, studeni 2022.

Sadržaj

1. Uvod

Cilj je ovog priručnika objasniti napredne koncepte programskog jezika Swift s naglaskom na koncepte vezane uz razvoj aplikacija za Appleov mobilni uređaj iPhone, odnosno iOS. Priručnik se nadograđuje na prethodno obrađene osnovne koncepte programskog jezika Swift.

1.1. Preduvjeti

Kako bi Vam lakše bilo pratiti koncepte obrađene u ovom priručniku, potrebno je osnovno poznavanje Swift programskog jezika. Dodatno se preporuča osnovno predznanje objektno-orijentiranog pristupa te iskustvo u programskim jezicima sličnim Swiftu - Kotlin, Python, Java.

Zbog lakšeg razumijevanja koncepata koji se obrađuju u ovom priručniku, kreiran je javno dostupan GitHub repozitorij Advanced Swift Concepts koji prati teme u ovom priručniku. Repozitorij možete lokalno klonirati te tako testirati i detaljnije proučiti sve programske isječke dane u ovom priručniku.

Napomena: Iako je Swift otvorenog kôda i može se kompajlirati na većini operacijskih sustava, za lokalno testiranje pojedinih poglavlja, a posebice rad s hardverom iPhone uređaja, biti će Vam potrebno Apple računalo s XCode razvojnim okruženjem.

2. Napredno objektno-orijentirano programiranje

Objektno orijentirano programiranje temeljna je programska paradigma koju morate savladati ako ozbiljno namjeravate učiti Swift. To je zato što je objektno orijentirano-programiranje u srcu većine okvira s kojima ćete raditi. Razbijanje problema na objekte koji šalju poruke jedni drugima moglo bi se isprva činiti čudnim, ali to je dokazani pristup za pojednostavljenje složenih sustava.

Objekti se mogu koristiti za modeliranje gotovo bilo čega - koordinata na karti, dodira na ekranu, čak i fluktuirajućih kamatnih stopa na bankovnom računu. Stoga se prvo poglavlje u ovom priručniku odnosi upravo na napredan rad s klasama, svojstvima i metodama koje su temelj programskog jezika Swift i okvira koja koristi.

2.1. Klase

Klase su podatkovne konstrukcije opće namjene koje postaju građevni blokovi u arhitekturama modernih aplikacija. Svaka klasa agregira svojstva i metode za dodavanje funkcionalnosti svojim strukturama. Sintaksa dodavanja svojstva i metoda klasama jednaka je sintaksi koju koristimo za definiranje konstanti, varijabli i funkcija.

Klasa predstavlja tip podatka, dok je instanca klase tradicionalno poznata kao objekt. Međutim, Swiftove strukture (*struct*) i klase (*class*) mnogo su bliže u funkcionalnosti nego u drugim jezicima[1]. Stoga, veći dio ovog poglavlja opisuje funkcionalnost koje se odnose i na klase i na strukture u Swiftu.

2.2. Inicijalizatori

Inicijalizacija je proces pripreme instance klase, strukture ili nabrajanja za upotrebu. Proces uključuje postavljanje početne vrijednosti svakom pohranjenom svojstvu instance i izvođenje bilo kojeg drugog postavljanja ili inicijalizacije koji su potrebni prije nego što nova instanca bude spremna za upotrebu[2].[1]

Proces inicijalizacije implementira se definiranjem inicijalizatora, pomoću ključne riječi *init*. Inicijalizatori su posebne metode bez parametra koje ne vraćaju vrijednost, a koje se mogu pozvati za stvaranje nove instance određenog tipa. Njihova je primarna uloga osigurati da su nove instance tipa ispravno inicijalizirane prije nego što se prvi put koriste.

Swift definira dvije vrste inicijalizatora za tipove klasa kako bi se osiguralo da sva pohranjena svojstva dobiju početnu vrijednost:

- Određeni inicijalizatori (*designated initializers*) - ključna riječ *init*
- Pogodni inicijalizatori (*convenience initializers*) - ključna riječ *convenience*

[1] Svim pohranjenim svojstvima klase - uključujući sva svojstva koja klasa nasljeđuje od svoje nadklase - mora se dodijeliti početna vrijednost tijekom inicijalizacije.

2.2.1. Određeni inicijalizatori

Određeni inicijalizatori su primarni inicijalizatori za klasu koji u potpunosti inicijaliziraju sva svojstva klase i pozivaju odgovarajuće inicijalizatore nadklase da nastave proces inicijalizacije prema vrhu hijerarhije klasa[2]. Svaka klasa mora imati barem jedan određeni inicijalizator, a uobičajeno je da ima točno jedan.[2]

U nastavku je prikazan primjer interne klase *Pjesma* s trima pohranjenim svojstvima instance: naziv, trajanje (u sekundama) i opcionalnog tempa (u bpm). Klasa sadrži dva određena inicijalizatora koji omogućavaju kreiranje instanci tipa *Pjesma*. Inicijalizator na liniji 6 omogućava kreiranje instance Pjesme bez tempa, dok inicijalizator na liniji 12 mora primiti *Int* vrijednost tempa koja ne smije biti *nil*.

```
1  class Pjesma {
2      let naziv: String
3      let trajanje: Int
4      let tempo: Int?
5
6      init(naziv: String, trajanje: Int) {
7          self.naziv = naziv
8          self.trajanje = trajanje
9          self.tempo = nil
10     }
11
12     init(naziv: String, trajanje: Int, tempo: Int) {
13         self.naziv = naziv
14         self.trajanje = trajanje
15         self.tempo = tempo
16     }
17 }
```

2.2.2. Pogodni inicijalizatori

Pogodni inicijalizatori su sekundarni inicijalizatori klase. Nisu obavezni, ali se preporučaju u slučajevima kada će omogućiti "prečac" do određenog inicijalizatora te tako uštedjeti vrijeme; učiniti inicijalizaciju klase jasnijom u namjeri[2].

Shodno tome, najčešći primjer korištenja pogodnih inicijalizatora je kada određeni inicijalizator iste klase ima puno parametra pa se u implementaciji pogodnog inicijalizatora poziva određeni inicijalizator iste klase s predefiniranim nekim svojstvima:

[2]U nekim je slučajevima ovaj zahtjev zadovoljen nasljeđivanjem jednog ili više određenih inicijalizatora iz nadklase.

```swift
import Foundation

enum Zanr {
    case rockabilly, blues, jazz, reggae, nesvrstano
}

class Pjesma {
    let naziv: String
    let trajanje: Int
    let tempo: Int?
    let zanr: Zanr

    init(naziv: String, trajanje: Int, tempo: Int?, zanr: Zanr) {
        self.naziv = naziv
        self.trajanje = trajanje
        self.tempo = tempo
        self.zanr = zanr
    }

    // pogodni inicijalizator
    convenience init(naziv: String, trajanje: Int) {
        self.init(
            naziv: naziv,
            trajanje: trajanje,
            tempo: nil,
            zanr: Zanr.nesvrstano
        )
    }
}

let megaHit = Pjesma(naziv: "Unchained Melody", trajanje: 154)

print("Naziv: \(megaHit.naziv)")         // Unchained Melody
print("Trajanje: \(megaHit.trajanje)s") // 154s
print("Tempo: \(megaHit.tempo)")         // nil
print("Zanr: \(megaHit.zanr)")           // nesvrstano
```

Međutim, ako malo bolje sagledamo primjer iznad, možemo uočiti kako smo umjesto pogodnog inicijalizatora mogli samo pridružiti predefinirane vrijednosti svojstvima klase *Pjesma* te bismo postigli isti efekt. Međutim, pogodni inicijalizatori nam pružaju mogućnost naprednije obrade pojedinih parametra inicijalizatora na način da uvedemo dodatne provjere i logiku pri postavljanju vrijednosti svojstva.

2.2.3. Parametrizirani inicijalizatori

Kako bismo što preciznije opisali parametre inicijalizatora, u deklaraciju možemo nadodati oznake parametara. Oznake semantički upotpunjuju i proširuju parametar inicijalizatora te

nam omogućuju bolje prilagođen proces inicijalizacije. Oznake inicijalizacije imaju iste mogućnosti i sintaksu kao i oznake parametra funkcija i metoda[2].

Sljedeći primjer definira klasu *Pjesma*, koja pohranjuje vrijeme trajanje pjesme u sekundama. Klasa implementira dva prilagođena inicijalizatora pod nazivom *init(naziv:minute:)* i *init(naziv:sekunde:)*, koji inicijaliziraju novu instancu klase *Pjesme* s vrijednošću iz različitih vremenskih mjernih jedinica - sekunde i minute:

```
1  import Foundation
2
3  class Pjesma {
4      let naziv: String
5      let trajanje: Int
6
7      init(naziv: String, minute trajanje: Int) {
8          self.naziv = naziv
9          self.trajanje = trajanje * 60
10     }
11
12     init(naziv: String, sekunde trajanje: Int) {
13         self.naziv = naziv
14         self.trajanje = trajanje
15     }
16 }
17
18 let pjesma1 = Pjesma(naziv: "Any Day Now", minute: 3)
19 let pjesma2 = Pjesma(naziv: "Unchained Melody", sekunde: 152)
20
21 print(pjesma1.trajanje) // 180
22 print(pjesma2.trajanje) // 152
```

Prvi inicijalizator (linija 7) ima dva parametra inicijalizacije - *naziv* i *trajanje*. Dodatno, parametar inicijalizacije *trajanje* ima definiranu oznaku *minute*. Međutim, drugi inicijalizator (linija 12) ima identične parametre inicijalizacije. Razlika je samo u oznaci parametra *trajanje* koja je kod drugog inicijalizatora imenovana *sekunde*, umjesto *minute* u slučaju prvog inicijalizatora. Oba inicijalizatora pretvaraju svoj pojedinačni parametar *trajanje* u odgovarajuću vrijednost vremenskog trajanja pjesme u sekundama i pohranjuju tu vrijednost u istoimeno pohranjeno svojstvo instance - *Pjesma.trajanje*.

2.2.4. Inicijalizatori bez oznaka

Oznake parametra inicijalizatora nisu obavezne. Umjesto eksplicitne oznake parametra možemo koristiti podvlaku (_) koja nadjačava predefinirano ponašanje oznake[2]:

```
class Pjesma {
    let naziv: String
    let trajanje: Int

    init(naziv: String, sekunde trajanje: Int) {
        self.naziv = naziv
        self.trajanje = trajanje
    }
}

class AudioKanal {
    var trenutnaPjesma: Pjesma

    init(_ pjesma: Pjesma) {
        self.trenutnaPjesma = pjesma
    }
}

let pjesma = Pjesma(naziv: "Unchained Melody", sekunde: 152)
let audioKanal = AudioKanal(pjesma)
```

Poziv inicijalizatora *AudioKanal(pjesma)* jasan je u svojoj namjeri bez potrebe za oznakom argumenta. Stoga je prikladno napisati ovaj inicijalizator kao *init(_ pjesma: Pjesma)* tako da se može pozvati pružanjem neimenovane vrijednosti tipa *Pjesma*.

2.2.5. Obavezni inicijalizatori

Zbog pravila *ako podklasa ima vlastiti određeni inicijalizator, niti jedan inicijalizator iz nadklase se ne nasljeđuje*, moguće je da podklasa nema niti jedan inicijalizator[2]. Međutim, u nekoliko slučajeva to stvara probleme inicijalizacije same klase. Pogledajmo sljedeći primjer:

```
protocol GlazbenaKompozicija {
    var naziv: String { get set }
    var trajanje: Int { get set }

    init(naziv: String, trajanje: Int)
}

class Pjesma: GlazbenaKompozicija {
    var naziv: String
    var trajanje: Int

    // initializer requirement 'init(naziv:trajanje:)' can only be
    // satisfied by a 'required' initializer in non-final class 'Pjesma'
```

```
14      init(naziv: String, trajanje: Int) {
15          self.naziv = naziv
16          self.trajanje = trajanje
17      }
18  }
```

Iako nam se na prvu može činiti da je sve u redu - klasa *Pjesma* je naslijedila traženi inicijalizator protokola *GlazbenaKompozicija* - inicijalizator klase *Pjesma* javlja grešku kompilacije. Problem je u tome da ako bi neka klasa dalje naslijedila klasu *Pjesma*, ona bi morala ili naslijediti inicijalizator nadklase ili definirati vlastiti inicijalizator da se sačuva *dogovor* protokola. Međutim, to ničim nije zajamčeno upravo zbog prethodno spomenutog pravila.

Sukladno tome postoje dva moguća rješenja. Kao prvo rješenje možemo uzeti sugestiju Swift kompajlera i označiti klasu kao *final* čime garantiramo da će *dogovor* protokola ostati sačuvan jer daljnje nasljeđivanje te klase neće biti moguće. No, ako nam je daljnje nasljeđivanje potrebno, Swift nam pruža drugo rješenje - ključnu riječ *required*. Označavanje inicijalizatora kao *required* obvezuje svaku podklasu da definira traženi inicijalizator te se tako čuva *dogovor* protokola[2].

```
1   protocol GlazbenaKompozicija {
2       var naziv: String { get set }
3       var trajanje: Int { get set }
4
5       init(naziv: String, trajanje: Int)
6   }
7
8   class Pjesma: GlazbenaKompozicija {
9       var naziv: String
10      var trajanje: Int
11
12      // obavezni inicijalizator
13      required init(naziv: String, trajanje: Int) {
14          self.naziv = naziv
15          self.trajanje = trajanje
16      }
17  }
```

Zanimljivo je istaknuti da modifikator nadjačavanja (*override*) nije potreban za obavezne inicijalizatore. Ako i pokušamo označiti nadjačavanje obaveznog inicijalizatora, kompajler će opet bacati grešku kompilacije. Razlog je taj što se nadjačavanje podrazumijeva kada se nadjačava obavezan inicijalizator.

2.2.6. Neuspjeli inicijalizatori

Ne možemo pretpostaviti da će inicijalizacija klase, strukture ili nabrajanja uvijek uspjeti. Na primjer, pokušaj pohrane resursa (slika, video) koji ne postoji jedan je od čestih razloga propadanja inicijalizatora kod razvoja iOS aplikacija. Radi izbjegavanja takvih situacija, već u Swift verziji 1.1 uvedeni su neuspjeli inicijalizatori (*failable initializators*)[2].

Kao što ime sugerira, neuspjeli inicijalizator je inicijalizator koji može propasti. Definira se na način da se iza ključne riječi *init* stavit upitnik (?). Dodatno, neuspjeli inicijalizator mora definirati događaja za koji se smatra da inicijalizacija nije uspjela. Za označavanje točke u kojoj se pokreće neuspjeh inicijalizacije, koristi se *return* izraz koji vraća *nil* vrijednost.[3] Stoga, kada se instanca inicijalizira pomoću neuspjelog inicijalizatora, rezultat je *Optional* objekt koji ili sadrži instancu (kada je inicijalizacija uspjela) ili sadrži *nil* (kada inicijalizacija nije uspjela).

```
1  import Foundation
2
3  class Pjesma {
4      let naziv: String
5      let trajanje: Int
6
7      init(naziv: String, trajanje: Int) {
8          self.naziv = naziv
9          self.trajanje = trajanje
10     }
11 }
12
13 class AudioKanal {
14     var trenutnaPjesma: Pjesma
15     var trenutnoSekunde: Int
16
17     // neuspjeli inicijalizator
18     init?(pjesma: Pjesma, pozicija: Int) {
19         if (0...(pjesma.trajanje)).contains(pozicija) {
20             self.trenutnoSekunde = pozicija
21         } else {
22             return nil
23         }
24         self.trenutnaPjesma = pjesma
25     }
26 }
27
28 let megaHit = Pjesma(naziv: "Unchained Melody", trajanje: 154)
29
30 let audioKanal1 = AudioKanal(pjesma: megaHit, pozicija: 42)
31 print(audioKanal1)  // Optional(AudioKanal)
32
33 let audioKanal2 = AudioKanal(pjesma: megaHit, pozicija: 164)
34 print(audioKanal2)  // nil
```

Pjesmu na zadanoj vremenskoj poziciji (*pozicija*) možemo pustiti na neki audio kanal samo ako se zadana vremenska pozicija nalazi unutar trajanja pjesme.

[3] U Swiftu, inicijalizatori ne vraćaju vrijednosti. Izraz *return nil* služi isključivo za pokretanje greške kod inicijalizacije - ključna riječ *return* ne služi za označavanje uspjeha inicijalizacije.

Koristeći *guard* izraz možemo iskoristiti punu moć Swift jezika te dodatno skratiti implementaciju neuspjelog inicijalizatora:

```
class AudioKanal {
    var trenutnaPjesma: Pjesma
    var trenutnoSekunde: Int

    init?(pjesma: Pjesma, pozicija: Int) {
        guard (0...(pjesma.trajanje)).contains(pozicija) else {
            return nil
        }
        self.trenutnaPjesma = pjesma
        self.trenutnoSekunde = pozicija
    }
}
```

Napomena: neuspjeli inicijalizatori ne mogu imati iste tipove (i imena / labele) parametra kao i inicijalizatori koji garantiraju uspješnu inicijalizaciju instance.

2.3. Svojstva

Svojstva koristimo kada želimo povezati neke vrijednosti s klasom, strukturom ili nabrajanjem. U Swiftu postoje dvije vrste svojstva: pohranjena svojstva (*stored properties*) i izračunata svojstva (*computed properties*)[3].

Instanca određenog tipa, uključujući strukturu, klasu i nabrajanje, povezana je sa svim svojstvima definiranim u tipu. Međutim, umjesto instanci, svojstva se također mogu pridružiti samo tipovima. Takva svojstva nazivaju se svojstva tipa (*type properties*)[3].

2.3.1. Pohranjena svojstva

Pohranjena svojstva mogu se definirati samo u klasama i strukturama, a služe za pohranu varijabilnih vrijednosti (definiranih ključnom riječi *var*) i konstantnih vrijednosti (definiranih ključnom riječi *let*) kao atribut klase[3]. Pohranjenim svojstvima možemo zadati predefinirane vrijednosti u trenutnu deklariranja. U slučaju da se radi o varijabilnom svojstvu, možemo ga promijeniti i kasnije prilikom inicijalizacije klase ili strukture.

U nastavku je dan primjer korištenja pohranjenih svojstva instance, u ovom slučaju klase *Pjesma*.

```
import Foundation

class Pjesma {
    // pohranjena svojstva instance
    let naziv: String       // konstantno svojstvo instance
    var trajanje: Int = 120 // varijabilno predefinirano svojstvo instance
    let tempo: Int?         // konstantno opcionalno svojstvo instance

```

```
9       init(naziv: String) {
10          self.naziv = naziv
11          self.tempo = nil
12      }
13
14      init(naziv: String, trajanje: Int, tempo: Int) {
15          self.naziv = naziv
16          self.trajanje = trajanje
17          self.tempo = tempo
18      }
19  }
20
21  let megaHit = Pjesma(naziv: "Unchained Melody")
22
23  print("Naziv mega hita: \(megaHit.naziv)")          // Unchained Melody
24  print("Trajanje mega hita: \(megaHit.trajanje)")    // 120
25
26  megaHit.trajanje = 174
27  print("Trajanje mega hita: \(megaHit.trajanje)")    // 174
```

Ukoliko bismo pokušali naknadno promijeniti *bpm* svojstvo, dobili bismo poruku o greški: *cannot assign to property: 'bpm' is a 'let' constant*, a koja nam upravo dokazuje da naknadno možemo mijenjati samo varijabilna pohranjena svojstva instance.

2.3.2. Izračunata svojstva

Za razliku od pohranjenih svojstva, izračunata svojstva nemaju pohranjenu vrijednost. Dakle, izračunata svojstva izračunavaju vrijednosti na temelju određenog izraza tijekom njihovog poziva. Možemo ih definirati u klasama, strukturama i nabrajanjima[3].

Način definiranja i korištenja *read-only* izračunatog svojstva instance (klase) prikazan je u nastavku.

```
1   import Foundation
2
3   class Pjesma {
4       let naziv: String
5       var trajanje: Int = 120
6       let tempo: Int?
7
8       init(naziv: String) {
9           self.naziv = naziv
10          self.tempo = nil
11      }
12
13      init(naziv: String, trajanje: Int, tempo: Int) {
14          self.naziv = naziv
15          self.trajanje = trajanje
16          self.tempo = tempo
```

```
17      }
18  }
19
20  class AudioKanal {
21      // pohranjena varijabilna svojstva instance
22      var trenutnaPjesma: Pjesma
23      var trenutnoSekunde: Int = 0
24      // izracunata svojstva instance moraju biti varijabilna
25      var preostaleSekunde: Int {
26          get {
27              return trenutnaPjesma.trajanje - trenutnoSekunde
28          }
29      }
30
31      init(pjesma: Pjesma) {
32          self.trenutnaPjesma = pjesma
33      }
34  }
35
36  let megaHit = Pjesma(naziv: "Unchained Melody")
37  let kanal = AudioKanal(pjesma: megaHit)
38
39  print("Pustam pjesmu \(megaHit.naziv)")
40  print("Preostalo sekundi: \(kanal.preostaleSekunde)")    // 120
41  kanal.trenutnoSekunde = 15
42  print("Preostalo sekundi: \(kanal.preostaleSekunde)")    // 105
```

U početnom stanju instance *kanal*, izračunato preostalo vrijeme trajanja pjesme u sekundama jednako je samom trajanju pjesme, to jest 120 sekundi. Simulacijom prolaska vremena od 15 sekundi (linija 41) te ponovnim dohvaćanjem svojstva *preostaleSekunde* vidimo kako se svojstvo zaista promijenilo - preostalo vrijeme trajanje pjesme je umanjeno za 15 sekundi te iznosi 105 sekundi.

Detaljnijim razmatranjem izračunatih svojstva, nameće se pitanje koja je razlika između izračunatih svojstva instance i metoda instance, pošto su izračunata svojstva ionako interno implementirana kao funkcije.

Semantički gledano, svojstva izražavaju inherentnu vrijednost instance, dok metode izvršavaju radnju nad instancom / instancama. Glavna distinkcija jest da metode mogu imati parametre, a svojstva ne. Metode instance se preferiraju za slučajeve kada izračun vrijednosti dolazi s potencijalno neželjenim učincima poput iznimaka. Ako izračun izvršava bilo kakvu radnju (učitavanje, izračunavanje, ispisivanje), odnosno ima glagolsko ime, to nam daje naznaku da bi ta radnja trebala biti implementirana kao metoda instance.

Također treba imati na umu čitljivost kôda i njegovu složenost. Shodno tome, izračunata svojstva instance trebala bi imati prednost nad metodama instance u slučajevima kada koristimo jednostavne izračune. Idealno bi bilo da je vremenska složenosti izračuna *O(1)*. Dodatno, svojstva nam omogućuju dodavanje promatrača (vidi poglavlje Promatrači svojstva).

2.3.3. Lijena svojstva

Lijena svojstva (*lazy properties*) su svojstva čija se početna vrijednost ne dodjeljuje (izračunava) sve do prvog poziva[3]. Moguće ih je koristiti unutar klasa i struktura, a definiraju se ključnom riječi *lazy*. Lijena svojstva uvijek moraju biti varijabilna (ključna riječ *var*) jer se njihova početna vrijednost možda neće dohvatiti sve dok se ne završi inicijalizacija instance. [4]

Lijena svojstva korisna su kada početna vrijednost ovisi o vanjskim izračunima čije vrijednosti nisu poznate sve dok se inicijalizacija instance ne završi. Lijena svojstva nam mogu poslužiti i kao mehanizam optimizacije jer nam omogućuju da odgodimo vremenski složene, odnosno procesorski-zahtjevne izračune za postavljanje vrijednosti. To znači da izbjegavamo automatsko pokretanje složenog izračuna vrijednosti, odnosno, počinjemo ga izvršavati tek kad je potrebno; ako je potrebno.

Čest slučaj korištenja lijenog svojstva u razvoju iOS aplikacija je kada neko svojstvo instance obrađuje izračune poput poziva prema bazi podataka, poziva prema vanjskom servisu ili pak poziva čitanja datoteke. Svi navedeni izračuni spadaju u vremenski, a potencijalno i procesorski zahtjevne obrade, a to je ono što želimo izbjeći. Želimo korisniku što prije prikazati tražene podatke, a kasnije, ukoliko je uistinu potrebno, dobaviti preostale podatke.

Pretpostavimo da naša iOS aplikacija obrađuje instance *Pjesme*, čija svojstva naziva, izvođača i trajanja dohvaćamo iz baze, a svojstvo tempa putem vanjskog web servisa. Analizom korištenja aplikacije saznali smo da u većini slučajeva korisnici ne koriste svojstvo tempa pa nam ono postaje idealni kandidat za lijeno svojstvo. Shodno tome, primjer u nastavku koristi lijeno svojstvo tempa kako bi se izbjegla *'nepotrebna'* prijevremena inicijalizacija.

```
1  import Foundation
2
3  class Pjesma {
4      let naziv: String
5      let izvodac: String
6      var trajanje: Int
7
8      // lijeno svojstvo
9      lazy var bpm: Int = {
10         sleep(3)    // simulacija slozene obrade
11         return 76
12     }()
13
14     init(naziv: String, izvodac: String, trajanje: Int) {
15         self.naziv = naziv
16         self.izvodac = izvodac
17         self.trajanje = trajanje
18     }
19 }
20
```

[4] Konstantna svojstva uvijek moraju imati vrijednost prije nego što se inicijalizacija instance dovrši pa se stoga ne mogu deklarirati kao lijena.

```
let megaHit = Pjesma(
    naziv: "Unchained Melody",
    izvodac: "Elvis Presley",
    trajanje: 151)

print("Tempo mega hita: \(megaHit.bpm)") // ispis nakon obrade (3s)
print("Tempo mega hita: \(megaHit.bpm)") // instantno vraca vrijednost
```

Također je važno za napomenuti kako postavljanje vrijednosti kod lijenih svojstva ne predstavlja atomičnu operaciju[3]. Odnosno, izračuni unutar bloka lijenog svojstva ne garantiraju siguran pristup resursima kod paralelnog izvođenja više dretvi.

2.3.3.1. Životni vijek

Pri korištenju lijenih svojstva, potrebno je obratiti pažnju na njihov životni vijek. Nasuprot izračunatim svojstvima koja uvijek iznova rade izračun, lijena svojstva se izračunavaju samo jedanput - kod prvog poziva - što može prouzročiti neočekivano ponašanje instance. Dakle, ukoliko lijeno svojstvo radi obradu nad promjenjivim strukturama podataka, uzet će se u obzir samo prvobitno stanje strukture kada se lijeno svojstvo pozvalo. Naknadne izmijene strukture podatka neće utjecati na vrijednost lijenog svojstva:

```
import Foundation

struct Pjesma {
    let naziv: String
    var popularnost: Int
}

class Album {
    var pjesme: [Pjesma]

    lazy var najpopularnijaPjesma: Pjesma? = {
        print("Inicijaliziram svojstvo najpopularnije pjesme")
        return pjesme.max(by: { $0.popularnost < $1.popularnost })
    }()

    init(pjesme: [Pjesma]) {
        self.pjesme = pjesme
    }
}

var moodyBlueAlbum = Album(pjesme: [
    Pjesma(naziv: "Littl' Darlin", popularnost: 952_386),
    Pjesma(naziv: "Moody Blue", popularnost: 19_661_683),
    Pjesma(naziv: "Let Me Be There", popularnost: 1_237_139)
])

print(moodyBlueAlbum.najpopularnijaPjesma)
```

```
28 /**
29  * Inicijaliziram svojstvo najpopularnije pjesme
30  * Pjesma(naziv: "Moody Blue", popularnost: 19_661_683)
31  */
32
33 moodyBlueAlbum.pjesme.append(
34     Pjesma(naziv: "Unchained Melody", popularnost: 38_195_505)
35 )
36
37 print(moodyBlueAlbum.najpopularnijaPjesma)
38 /**
39  * Pjesma(naziv: "Moody Blue", popularnost: 19_661_683)
40  */
```

U primjeru iznad vidimo kako se lijeno svojstvo *Album.najpopularnijaPjesma* inicijaliziralo samo kod prvog poziva. Naknadna izmjena polja *Album.pjesme[Pjesma]* nije uzrokovala ponovno izračunavanje lijenog svojstva.

Ako zaista moramo ponovno izračunati lijeno svojstvo, to možemo učiniti naknadnim direktnim pristupanjem i upisivanjem nove vrijednost. Naravno, pod uvjetom da lijeno svojstvo instance dopušta postavljanje nove vrijednost izvana (prava pristupa svojstva).

2.3.4. Svojstva tipa

Za razliku od svojstva instance koja su pridružena instanci klase, strukture ili nabrajanja, svojstva tipa pridružena su samom tipu - ne pripadaju niti jednoj instanci tipa. U Swiftu možemo definirati svojstvo tipa ključnom riječi *static*, a za izračunata svojstva tipa i za tipove klasa trebamo koristiti ključnu riječ *class*, ali to neće dopustiti nadjačavanje metoda ili varijabli klase roditelja[3]. Za pristupanje svojstvima tipa, koristi se sintaksa točke te ona uvijek trebaju imati predefiniranu vrijednost jer nema inicijalizacije:

```
1  import Foundation
2
3  class AudioKanal {
4      // pohranjena svojstva tipa AudioKanal
5      static let maxLevel = 100
6      static var maxLevelSvihAudioKanala = 0
7      // pohranjeno predefinirano svojstvo instance
8      var trenutanLevel = 0
9  }
10
11 print("Max level AK: \(AudioKanal.maxLevel)")                    // 100
12 print("Max level svih AK: \(AudioKanal.maxLevelSvihAudioKanala)") // 0
```

U primjeru iznad, definirali smo dva svojstva tipa - *maxLevel* i *maxLevelSvihAudioKanala*. Svojstvom *maxLevel* označujemo da sve instance audio kanala dijele zajednički maksimalan level zvuka predstavljen cjelobrojnom vrijednošću 100. Svojstvom *maxLevelSvihAudioKanala* pohranjujemo trenutan maksimalan level koji neka instanca *AudioKanala* sadrži. Na

kraju je dodatno definirano i svojstvo instance *trenutanLevel* kojim mjerimo trenutan level zvuka za svaku instancu *AudioKanala*.

2.3.5. Promatrači svojstva

U Swiftu je svaka varijabla instance automatski svojstvo. To nam omogućava dodavanje takozvanih promatrača svojstva (*property observers*)[3]. U Swiftu postoje dvije vrste promatrača svojstva: jedan se poziva prije dodjele nove vrijednosti, a drugi nakon. Promatrač svojstva koji se poziva prije dodjele, označen je ključnom riječi *willSet*, a promatrač svojstva koji se poziva nakon dodjele, označen je ključnom riječi *didSet*.

Implementacija *willSet* promatrača svojstva prosljeđuje novu vrijednost svojstva kao konstantni parametar. U slučaju da ne definiramo naziv parametra, koristi se predefinirani naziv *newValue*. Slično vrijedi i za *didSet* promatrača svojstva gdje se prosljeđuje konstantni parametar *oldValue* koji sadrži staru vrijednost svojstva.

```
1  import Foundation
2
3  class AudioKanal {
4      static let maxLevel = 100
5      static var maxLevelSvihAudioKanala = 0
6      var trenutanLevel = 0 {
7          // promatrac prije postavljanja nove vrijednosti
8          willSet {
9              print("Zelim postaviti trenutan level na \(newValue).")
10         }
11
12         // nova vrijednost se postavlja
13
14         // promatrac nakon postavljanja nove vrijednosti
15         didSet(stariLevel) {
16             // ogranicava postavljanje levela kanala iznad
17             // maksimalnog levela
18             if trenutanLevel > AudioKanal.maxLevel {
19                 print("Uneseni level je previsok!" +
20                       "Smanjujem na \(AudioKanal.maxLevel).")
21                 trenutanLevel = AudioKanal.maxLevel
22             }
23
24             // uvazavanje promjena nad svojstvom maksimalnog
25             // levela svih audio kanala
26             if trenutanLevel > AudioKanal.maxLevelSvihAudioKanala {
27                 AudioKanal.maxLevelSvihAudioKanala = trenutanLevel
28             }
29
30             if trenutanLevel - stariLevel > 0 {
31                 print("Level povecan za \(trenutanLevel - stariLevel).")
32             } else if trenutanLevel - stariLevel < 0 {
```

```
33                  print("Level smanjen za \(stariLevel - trenutanLevel).")
34              } else {
35                  print("Level ostaje nepromijenjen: \(stariLevel).")
36              }
37          }
38      }
39  }
40
41  let lijeviAK = AudioKanal()
42  let desniAK = AudioKanal()
43
44  lijeviAK.trenutanLevel = 10
45  /**
46   * Zelim postaviti trenutan level na 10.
47   * Level povecan za 10.
48   */
49  print("Level lijevog AK: \(lijeviAK.trenutanLevel)")     // 10
50  lijeviAK.trenutanLevel = 24
51  /**
52   * Zelim postaviti trenutan level na 24.
53   * Level povecan za 14.
54   */
55  print("Level lijevog AK: \(lijeviAK.trenutanLevel)")     // 24
56
57  print("Maksimalan level svih AK: " +
58        "\(AudioKanal.maxLevelSvihAudioKanala)")  // 24
59
60  desniAK.trenutanLevel = 124
61  /**
62   * Zelim postaviti trenutan level na 124.
63   * Uneseni level je previsok! Smanjujem na 100.
64   * Level povecan za 100.
65   */
66  print("Level desnog AK: \(desniAK.trenutanLevel)")  // 100
67
68  print("Maksimalan level svih AK: " +
69        "\(AudioKanal.maxLevelSvihAudioKanala)")  // 100
```

Na linijama 8-10 implementiran je promatrač svojstva instance prije dodjele koji koristi predefiniranu proslijeđenu konstantu nove vrijednosti *newValue*. S druge strane, implementacija promatrača svojstva instance nakon dodjele (linije 15-37) koristi korisnički definiran naziv proslijeđenog konstantnog parametra stare vrijednosti - *stariLevel*. Nova vrijednost svojstva postavlja se između poziva promatrača svojstva prije i nakon dodjele (linija 12). Ipak, konačna vrijednost svojstva instance može se i "nadjačati" u pozivu promatrača nakon dodjele (linija 21).

2.3.6. Omotači svojstva

Omotači svojstva (*property wrappers*) su instance koje enkapsuliraju pohranu svojstva. Kao što ime sugerira, svojstvo je *omotano* omotačem koji kontrolira pristup svojstvu koje omotava. Struktura ili klasa definiraju svojstvo, ali omotač je instanca koja osigurava pristup i samu pohranu definiranog svojstva[3].

Omotač svojstva definiran je označavanjem strukture ili klase atributom *@propertyWrapper* i svojstvom s nazivom *wrappedValue*. To su jedini zahtjevi koje omotač svojstva treba implementirati. Primjena omotača svojstva na svojstvo radi se na način da svojstvo anotiramo koristeći *@NazivOmotačaSvojstva* sintaksu.

U nastavku je prikazana implementacija dvaju omotača svojstva:

- *@Kapitalizirano*: zadužan da svaka riječ u unesenom *Stringu* bude kapitalizirana
- *@Nenegativno*: zadužan da unesena cjelobrojna vrijednost ne bude negativna (što nam može biti vrlo bitno za daljnju obradu tog svojstva u drugim instancama / metodama)

```
1  import Foundation
2
3  @propertyWrapper
4  struct Kapitalizirano {
5      var wrappedValue: String {
6          didSet {
7               wrappedValue = wrappedValue.capitalized
8          }
9      }
10
11     init(wrappedValue: String) {
12         self.wrappedValue = wrappedValue.capitalized
13     }
14 }
15
16 @propertyWrapper
17 struct Nenegativno {
18     var wrappedValue: Int = 0 {
19         didSet {
20             if wrappedValue < 0 {
21                 wrappedValue = 0
22             }
23         }
24     }
25
26     init(wrappedValue: Int) {
27         if wrappedValue >= 0 {
28             self.wrappedValue = wrappedValue
29         } else {
30             self.wrappedValue = 0
```

```
31          }
32      }
33  }
34
35  class Pjesma {
36      @Kapitalizirano var naziv: String
37      @Kapitalizirano var izvodac: String
38      @Nenegativno var trajanje: Int
39
40      init(naziv: String, izvodac: String, trajanje: Int) {
41          self.naziv = naziv
42          self.izvodac = izvodac
43          self.trajanje = trajanje
44      }
45  }
46
47  let megaHit = Pjesma(
48      naziv: "unchained melody",
49      izvodac: "elvis presley",
50      trajanje: 151)
51
52  print("Naziv: \(megaHit.naziv)")         // Unchained Melody
53  print("Izvodac: \(megaHit.izvodac)")     // Elvis Presley
54  print("Trajanje: \(megaHit.trajanje)")   // 151
55
56  megaHit.naziv = "moody blue"
57  megaHit.trajanje = -74
58
59  print("Novi naziv: \(megaHit.naziv)")        // Moody Blue
60  print("Novo trajanje: \(megaHit.trajanje)") // 0
```

Nakon definiranja omotača *@Kapitalizirano* (linije 3 - 14) i omotača *@Nenegativno* (linije 16 - 24), možemo ih koristiti u strukturama i klasama. Primjer korištenja predstavlja klasa *Pjesma* gdje želimo da naziv pjesme i izvođača uvijek budu kapitalizirani, te da trajanja pjesme u sekundama uvijek mora biti nenegativna cjelobrojna vrijednost.

Važno je napomenuti kako omotana svojstva uvijek moraju biti varijabilna (ključna riječ *var*) jer omotači svojstava mogu biti izračunata svojstva pa ih ima smisla ograničiti na *var* deklaracije. Odnosno, vrijednost svojstva se može promijeniti čak i ako je memorija konstantna. Trenutno ne postoji način da kompilator to potvrdi za omotač svojstva bez otkrivanja je li *wrappedValue* pohranjeno svojstvo ili izračunato svojstvo, a što bi narušilo mogućnost autora omotača svojstva da promijeni svoj omotač svojstva.

2.4. Metode

Metode su funkcije koje su povezane s određenim tipom. Klase, strukture i nabrajanja mogu definirati metode instance, koje enkapsuliraju specifične zadatke i funkcionalnost za rad s instancom određenog tipa. Klase, strukture i nabrajanja također mogu definirati i metode tipa, koje su povezane sa samim tipom. Razlika između metoda instance i metoda tipa analogna je svojstvima instanca i svojstvima tipa:

- Metode instance - mogu se pozvati isključivo nad inicijaliziranom instancom tipa
- Metode tipa (*static*) - mogu se pozvati izravno na tipu bez stvaranja instance tog tipa.

2.4.1. Mutirajuće metode

Prema pravilima Swifta, svojstva vrijednosnih tipova (strukture i nabrajanja) se ne mogu mijenjati unutar njihovih metoda instance[4]. Međutim, ako trebamo izmijeniti svojstva strukture ili nabrajanja unutar određene metode, možemo se odlučiti za mutirajuće ponašanje metode. Takve metode nazivamo mutirajuće metode (*mutating methods*).

Mutirajuća metoda može mutirati (tj. promijeniti) svoja svojstva instance unutar metode, a sve promjene koje napravi, pohranjuju se natrag u izvornu instancu kada obrada završi. Mutirajuća metoda također može dodijeliti potpuno novu instancu svom implicitnom svojstvu *self*, a ta nova instanca će zamijeniti postojeću kada metoda završi[4].

Opisano mutirajuće ponašanje metode može se uključiti korištenjem ključne riječi *mutating* ispred ključne riječi *func* za tu metodu:

```
1  import Foundation
2  import Collections
3
4  struct Pjesma {
5      let naziv: String
6      var trajanje: Int
7
8      init(naziv: String, trajanje: Int) {
9          self.naziv = naziv
10         self.trajanje = trajanje
11     }
12 }
13
14 struct AudioKanal {
15     private(set) var trenutnaPjesma: Pjesma
16     private(set) var redCekanja: Deque<Pjesma> = Deque()
17
18     init(_ pjesma: Pjesma) {
19         self.trenutnaPjesma = pjesma
20     }
21
```

```
22      func pokreni() {
23          print("Pustam pjesmu \(self.trenutnaPjesma.naziv).")
24      }
25
26      // mutirajuca metoda
27      mutating func dodajNaKrajReda(_ pjesma: Pjesma) {
28          redCekanja.append(pjesma)
29          print("Pjesma \(pjesma.naziv) dodana je na kraj reda.")
30      }
31
32      // mutirajuca metoda
33      mutating func sljedeci() {
34          guard let sljedecaPjesma = redCekanja.popFirst() else {
35              print("Nema pjesme u redu cekanja.")
36              return
37          }
38          self.trenutnaPjesma = sljedecaPjesma
39          self.pokreni()
40      }
41  }
42
43  let megaHit = Pjesma(naziv: "Unchained Melody", trajanje: 154)
44  var kanal = AudioKanal(megaHit)
45  kanal.pokreni()
46  // Pustam pjesmu Unchained Melody.
47
48  let topHit = Pjesma(naziv: "Moody Blue", trajanje: 170)
49  kanal.dodajNaKrajReda(topHit)
50  // Pjesma Moody Blue dodana je na kraj reda.
51
52  kanal.sljedeci()
53  // Pustam pjesmu Moody Blue.
54  kanal.sljedeci()
55  // Nema pjesme u redu cekanja.
```

2.4.2. Promjenjive metode

Swift 5.4 dao je podršku varijabilnim parametrima u metodama[4]. Metode koje imaju minimalno jedan varijabilni parametar nazivaju se promjenjivim metodama (*variadic methods*).[5]

Promjenjive metode su nam korisne kada želimo odrediti da se nekom (varijabilnom) parametru može proslijediti različiti broj ulaznih vrijednosti kada se metoda pozove - nula ili više vrijednosti određenog tipa. Varijabilni parametar definiramo tako da nakon naziva tipa parametra dodamo trotočje (...).

[5]Metoda može imati više promjenjivih parametara - prvi parametar koji dolazi nakon varijabilnog parametra mora imati oznaku argumenta. Razlog tome je taj da kompajler može nedvosmisleno odrediti koji se argumenti prosljeđuju kojem varijabilnom parametru.

Vrijednosti proslijeđene varijabilnom parametru dostupne su unutar tijela metode kao niz odgovarajućeg tipa. Na primjer, promjenjivi parametar s nazivom *pjesme* i tipom *Pjesma...* dostupan je unutar tijela metode kao konstantni niz koji se zove *pjesme* tipa *[Pjesma]*.

```
1  import Foundation
2  import Collections
3
4  struct Pjesma {
5      let naziv: String
6      var trajanje: Int
7
8      init(naziv: String, trajanje: Int) {
9          self.naziv = naziv
10         self.trajanje = trajanje
11     }
12 }
13
14 struct AudioKanal {
15     private(set) var trenutnaPjesma: Pjesma
16     private(set) var redCekanja: Deque<Pjesma> = Deque()
17
18     init(_ pjesma: Pjesma) {
19         self.trenutnaPjesma = pjesma
20     }
21
22     func pokreni() {
23         print("Pustam pjesmu \(self.trenutnaPjesma.naziv).")
24     }
25
26     // mutirajuca promjenjiva metoda
27     mutating func dodajNaKrajReda(_ pjesme: Pjesma...) {
28         redCekanja.insert(contentsOf: pjesme, at: redCekanja.endIndex)
29         for pjesma in pjesme {
30             print("Pjesma \(pjesma.naziv) dodana je na kraj reda.")
31         }
32     }
33
34     mutating func sljedeci() {
35         guard let sljedecaPjesma = redCekanja.popFirst() else {
36             print("Nema pjesme u redu cekanja.")
37             return
38         }
39         self.trenutnaPjesma = sljedecaPjesma
40         self.pokreni()
41     }
42 }
43
44 let megaHit = Pjesma(naziv: "Unchained Melody", trajanje: 154)
45 var kanal = AudioKanal(megaHit)
```

```
46 kanal.pokreni()
47 // Pustam pjesmu Unchained Melody.
48
49 let topHit = Pjesma(naziv: "Moody Blue", trajanje: 170)
50 let ultraHit = Pjesma(naziv: "Fairytale", trajanje: 165)
51 kanal.dodajNaKrajReda(topHit, ultraHit)
52 /**
53  * Pjesma Moody Blue dodana je na kraj reda.
54  * Pjesma Fairytale dodana je na kraj reda.
55  */
56
57 kanal.sljedeci()
58 // Pustam pjesmu Moody Blue.
59 kanal.sljedeci()
60 // Pustam pjesmu Fairytale.
61 kanal.sljedeci()
62 // Nema pjesama u redu cekanja.
```

3. Upravljanje memorijom

Kada se iOS aplikacija pokrene, cijela datoteka koja sadrži sve izvršne upute bit će učitana u radnu memoriju. U isto vrijeme, aplikacija će od iOSa zatražiti još jedan dio radne memorije koji se naziva hrpa. Hrpa je mjesto u radnoj memoriji gdje će se pohranjivati sve, aplikaciji potrebne, instance klase (sve dok aplikacija radi).

Kada govorimo o upravljanju memorijom, mislimo na proces upravljanja heap memorijom. To znači upravljanje životnim ciklusima instanci na hrpi i osiguravanje da se te instance oslobode (maknu) kada više nisu potrebne kako bi se memorija mogla ponovno koristiti.

Upravljanje memorijom je vrlo važno u bilo kojoj aplikaciji, a posebno u iOS aplikacijama koje imaju stroga memorijska i druga ograničenja.

Swift, za razliku od ostalih mainstream jezika poput C# i Jave, ne koristi automatsko sakupljanje smeća (*automatic garbage collection*, već koristi sustav zvan automatsko brojanje referenci (*automatic reference counting - ARC*). U ovom je poglavlju detaljno objašnjeno kako Swift implementira dealokaciju (deinicijalizaciju) instanci iz radne memorije kada više nisu potrebne.

3.1. Deinicijalizacija

Instance tipova klasa također mogu implementirati deinicijalizator (ključna riječ *deinit*), koji izvodi bilo koje prilagođeno čišćenje neposredno prije nego što se instanca te klase oslobodi iz memorije[5]. Za razliku od inicijalizatora kojih može biti više, deinicijalizator može postojati samo jedan.

Deinicijalizacija instance odvija se kada aplikaciji zadana instanca više nije potrebna (više se ne referencira), odnosno, kada ona sadrži vrijednost *nil*:

```
1  import Foundation
2
3  class Pjesma {
4      let naziv: String
5      var trajanje: Int
6
7      init(naziv: String, trajanje: Int) {
8          self.naziv = naziv
9          self.trajanje = trajanje
10     }
11
12     // deinicijalizator
13     deinit {
14         print("Pjesma \(self.naziv) je obrisana.")
15     }
16 }
17
```

```
18 var megaHit: Pjesma? = Pjesma(naziv: "Unchained Melody", trajanje: 154)
19 megaHit = nil   // Pjesma Unchained Melody je obrisana.
```

3.2. Automatsko brojanje referenci

Dakle, Swift za praćenje i upravljanje memorijom naših aplikacija koristi automatsko brojanje referenci - ABR. U većini slučajeva to znači da upravljanje memorijom "jednostavno radi", bez da moramo pretjerano o tome razmišljati. ABR automatski oslobađa memoriju koju koriste instance klase kada te instance više nisu potrebne.[1]

Međutim, u nekoliko slučajeva ABR zahtijeva više informacija o odnosima između dijelova kôda kako bi umjesto nas upravljao memorijom. Ovo poglavlje opisuje te situacije i pokazuje kako omogućiti ABRu da upravlja cijelom memorijom aplikacije bez naše intervencije.

3.2.1. Način rada

Svaki put kada kreiramo novu instancu klase, ABR dodjeljuje dio memorije za pohranu informacija o toj instanci. Ta memorija sadrži informacije o vrsti instance, zajedno s vrijednostima svih pohranjenih svojstava povezanih s tom instancom[6]. Osim toga, kada instanca više nije potrebna, ABR oslobađa memoriju koju ta instanca koristi, tako da se umjesto nje može koristiti u druge svrhe. To osigurava da instance klase ne zauzimaju prostor u memoriji kada više nisu potrebne.

Međutim, ako bi ABR oslobodio instancu koja je još u upotrebi, više ne bi bilo moguće pristupiti svojstvima te instance ili pozvati metode te instance. Doista, ako bismo pokušali pristupiti instanci, aplikacija bi se najvjerojatnije srušila. Kako bi ABR osigurao da instance ne nestanu dok su još potrebne, ABR prati koliko se svojstava, konstanti i varijabli trenutno odnosi na svaku instancu klase. Dakle, ABR neće poništiti instancu sve dok postoji barem jedna aktivna referenca na tu instancu.

3.3. Jake reference

Svaki put kada instanci klase dodijelimo svojstva, konstante ili varijable, ta svojstva, konstante ili varijable sadrže *jaku* referencu (*strong reference*) na instancu[6]. Referenca se naziva "jakom" referencom jer čvrsto drži tu instancu i ne dopušta da se poništi sve dok ta jaka referenca postoji.

[1] Brojanje referenci primjenjuje se samo na instance klasa. Strukture i enumeracije su vrijednosni tipovi, a ne referentni tipovi, te se ne pohranjuju i ne prosljeđuju referencom.

```
1  import Foundation
2
3  class Pjesma {
4      let naziv: String
5      var trajanje: Int
6
7      init(naziv: String, trajanje: Int) {
8          self.naziv = naziv
9          self.trajanje = trajanje
10         print("Pjesma \(self.naziv) je kreirana.")
11     }
12
13     deinit {
14         print("Pjesma \(self.naziv) je obrisana.")
15     }
16 }
17
18 // Kreiranje triju referenci na tip Pjesma?
19 var megaHit: Pjesma?
20 var topHit: Pjesma?
21 var ultraHit: Pjesma?
22
23 megaHit = Pjesma(
24     naziv: "Unchained Melody",
25     trajanje: 154
26     )   // Pjesma Unchained Melody je kreirana.
27
28 // Stvaranje jakih referenci na megaHit instancu
29 topHit = megaHit
30 ultraHit = megaHit
31
32 megaHit = nil
33 print("megaHit je dealociran")
34 topHit = nil
35 print("topHit je dealociran")
36 ultraHit = nil  // Pjesma Unchained Melody je obrisana.
37 print("ultraHit je dealociran")
```

Iz primjera iznad možemo vidjeti kako se instanca (objekt) klase *Pjesma* kreira tek kada se pozove njezin inicijalizator u linijama 23-26. Ispis "*Pjesma Unchained Melody je kreirana.*" nam potvrđuje da je inicijalizacija završena.

Budući da smo novu instancu *Pjesme* dodijelili varijabli *megaHit*, sada postoji jaka referenca varijable *megaHit* na novu instancu pjesme. A kako postoji barem jedna jaka referenca, ABR osigurava da se ta pjesma zadrži u memoriji; odnosno, da se ne dealocira.

Kada u linijama 29 i 30 dodijelimo istu instancu *Pjesme* još dvjema varijablama - *topHit* i *ultraHit* - ABR uspostavlja još dvije jake reference na tu istu instancu. Dakle, sveukupno imamo tri jake reference na istu instancu jedne pjesme.

Ukoliko pokušamo dealocirati bilo koju od referenci tako da ih postavimo na vrijednost *nil*, primijetit ćemo da se neće pozvati deinicijalizator te instance pjesme. Isto vrijedi i za početnu referencu gdje se instanca kreirala: nakon linije 32, nema ispisa deinicijalizatora u konzolu.

Tek kada treću, ujedno i posljednju, jaku referencu na istu instancu *Pjesme* prekinemo (linija 36), dobivamo ispis da je instanca uspješno dealocirana. U tom je trenutku ABRu jasno da se ta instanca *Pjesme* više nigdje ne referencira pa je može sigurno ukloniti iz memorije.

3.4. Problem jakog referentnog ciklusa

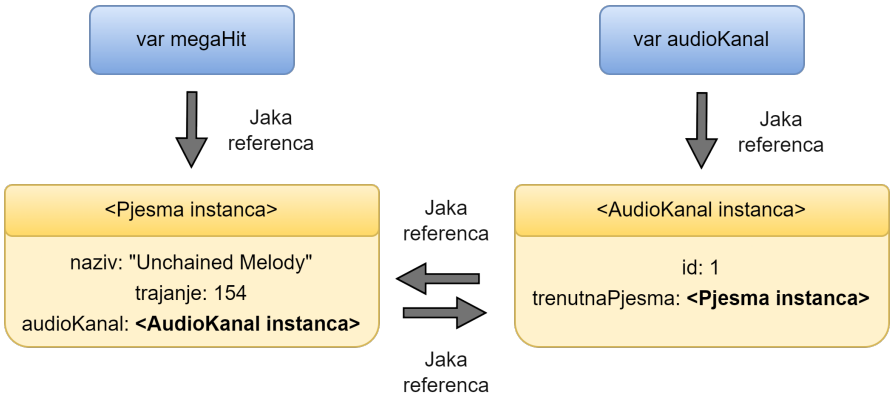
U prethodnom primjeru, ABR može vrlo lako pratiti broj referenci na novo-kreiranu instancu *Pjesme* i jednostavno je dealocirati kada više nije potrebna. Međutim, moguće je napisati kôd u kojem instanca klase nikad ne dođe do točke kada više nema jakih referenci. To se može dogoditi ako dvije instance klasje imaju jaku referencu jedna na drugu, tako da svaka instanca drugu održava na životu. Opisana situacija poznata je pod nazivom problem jakog referentnog ciklusa (*Strong Reference Cycles*)[6].

U nastavku je prikazan primjer kôda kako se slučajno može stvoriti jaki referentni ciklus dviju klasa. Opisani primjer modelira situaciju kada na audio kanalu puštamo, a pjesma zna na kojem se audio kanalu pušta:

```
1  import Foundation
2
3  class Pjesma {
4      let naziv: String
5      var trajanje: Int
6      var audioKanal: AudioKanal?
7
8      init(naziv: String, trajanje: Int) {
9          self.naziv = naziv
10         self.trajanje = trajanje
11         print("Pjesma \(self.naziv) je kreirana.")
12     }
13
14     deinit {
15         print("Pjesma \(self.naziv) je obrisana.")
16     }
17 }
18
19 class AudioKanal {
20     let id: Int
21     var trenutnaPjesma: Pjesma?
22
23     init(id: Int) {
24         self.id = id
25         print("Audio kanal \(self.id) je kreiran.")
```

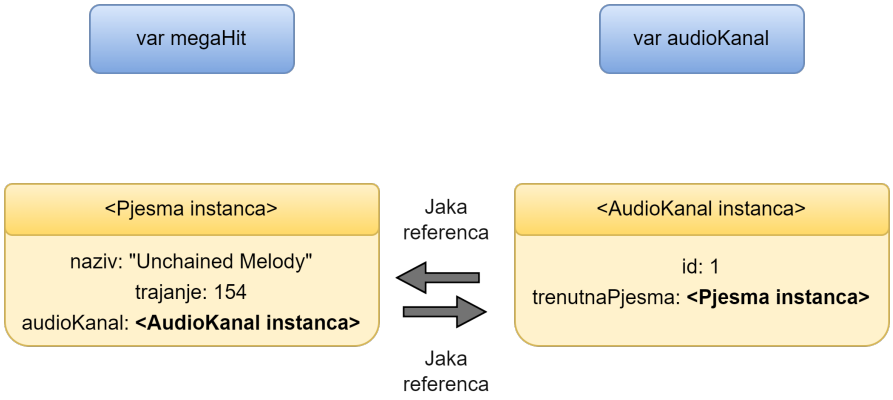
```
26      }
27
28      deinit {
29          print("Audio kanal \(self.id) je obrisan.")
30      }
31  }
32
33  var megaHit: Pjesma? = Pjesma(
34      naziv: "Unchained Melody",
35      trajanje: 154
36      )    // Pjesma Unchained Melody je kreirana.
37
38  var audioKanal: AudioKanal? = AudioKanal(
39      id: 1
40      )    // Audio kanal 1 je kreiran.
41
42  // Audio kanalu pridruzujemo pjesmu koju treba pustiti.
43  audioKanal?.trenutnaPjesma = megaHit
44  // Pjesmi pridruzujemo audio kanal na kojem se pusta.
45  megaHit?.audioKanal = audioKanal
46
47  // Pokusamo dealocirati instance pjesme i audio kanala,
48  // ali bezuspjesno - problem jakog referentnog ciklusa.
49  audioKanal = nil
50  megaHit = nil
```

Radi boljeg razumijevanja, u nastavku je prikazan vizualni dijagram koji opisuje kako ABR vidi jake reference nakon inicijalizacije i dodjele.



Slika 1: Primjer jakog referentnog ciklusa između klasa *(samostalna izrada, 2022)*

Odnosno, nakon dodjeljivanja vrijednosti *nil* varijablama *megaHit* i *audioKanal*:



Slika 2: Pokušaj dealokacije jakog referentnog ciklusa između klasa *(samostalna izrada, 2022)*

Dakle, dealokacijom varijabli *megaHit* i *audioKanal* nismo postigli potpunu dealokaciju i samih instanci u memoriji jer i dalje postoji jaka referenca između instance *Pjesme* i *AudioKanal*.

3.5. Slabe i neposjedovane reference

Swift pruža dva načina rješavanja jakih referentnih ciklusa kod rada sa svojstvima klasa:

1. Slabe reference (*weak references*) i
2. Neposjedovane reference (*unowned references*).

Slabe i neposjedovane reference omogućuju jednoj instanci u referentnom ciklusu da se referira na drugu instancu bez zadržavanja jakog utjecaja na nju. Instance se tada mogu odnositi jedna na drugu bez stvaranja jakog referentnog ciklusa[6].

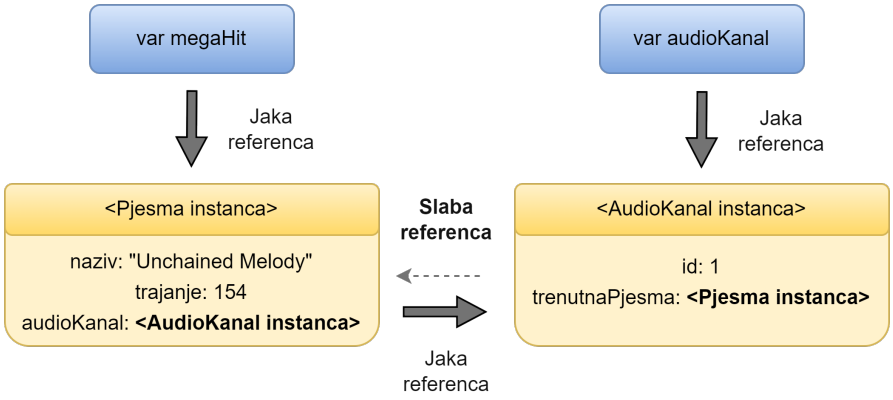
3.5.1. Slabe reference

Slabe reference koristimo u slučajevima kada druga instanca ima k*raći životni vijek* - to jest, kada se druga instanca može prva dealocirati[6]. U prethodnom primjeru audio kanala i pjesme, prikladno je da audio kanal, u nekom trenutku svog životnog vijeka, ne mora nužno imati pjesmu koja se pušta (već ostale samo stoje u redu čekanja) pa je slaba referenca prikladan način za prekid referentnog ciklusa u ovom slučaju.

Dakle, samo dodavanje ključne riječi *weak* ispred svojstva *AudioKanal.trenutnaPjesma* stvaramo slabu referencu koja omogućava ABRu da može prvo dealocirati pjesmu, a zatim i audio kanal.

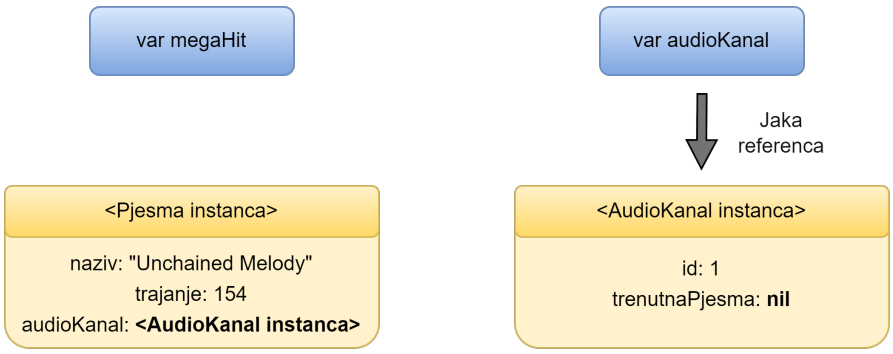
```swift
import Foundation

class Pjesma {
    let naziv: String
    var trajanje: Int
    var audioKanal: AudioKanal?

    init(naziv: String, trajanje: Int) {
        self.naziv = naziv
        self.trajanje = trajanje
        print("Pjesma \(self.naziv) je kreirana.")
    }

    deinit {
        print("Pjesma \(self.naziv) je obrisana.")
    }
}

class AudioKanal {
    let id: Int
    weak var trenutnaPjesma: Pjesma?    // slaba referenca

    init(id: Int) {
        self.id = id
        print("Audio kanal \(self.id) je kreiran.")
    }

    deinit {
        print("Audio kanal \(self.id) je obrisan.")
    }
}

var megaHit: Pjesma? = Pjesma(
    naziv: "Unchained Melody",
    trajanje: 154
    )   // Pjesma Unchained Melody je kreirana.

var audioKanal: AudioKanal? = AudioKanal(
    id: 1
    )   // Audio kanal 1 je kreiran.

audioKanal?.trenutnaPjesma = megaHit
megaHit?.audioKanal = audioKanal

megaHit = nil       // Pjesma Unchained Melody je obrisana.
audioKanal = nil    // Audio kanal 1 je obrisan.
```

Odnosno, vizualno smo postigli sljedeći odnos svojstva klasa:



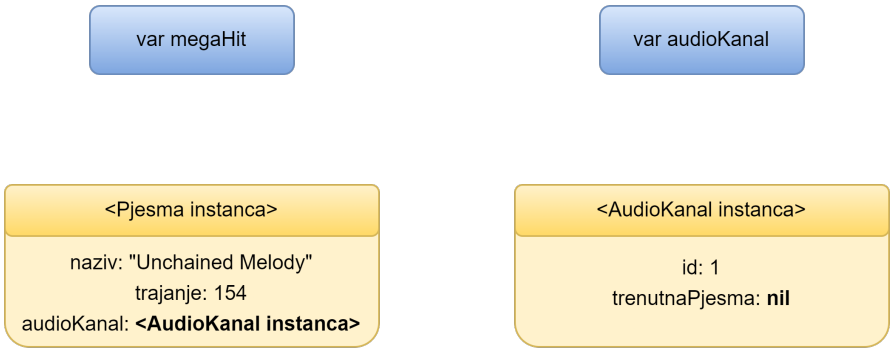
Slika 3: Odnos klasa *Pjesma* i *AudioKanal* pri korištenju slabe reference *(samostalna izrada, 2022)*

Kada u liniji 45, varijabli *megaHit* pridružimo vrijednost *nil* (dealociramo), prekidamo jaku referencu koju drži varijabla *megaHit* i tada više nema jakih referenci na instancu *Pjesme*. Time smo postigli da ABR može dealocirati instancu pjesme, a svojstvo *AudioKanal.trenutnaPjesma* postavlja na *nil*.



Slika 4: Dealokacija instance *Pjesma (samostalna izrada, 2022)*

Konačno možemo dealocirati i instancu audio kanala (linija 46):



Slika 5: Dealokacija instance *AudioKanal (samostalna izrada, 2022)*

čime smo riješili problem referentnog ciklusa i uspješno dealocirali povezane instance.

3.5.2. Neposjedovane reference

Kao i slaba referenca, neposjedovana referenca nema jaki utjecaj na instancu na koju se odnosi. Međutim, za razliku od slabe reference, neposjedovana referenca se koristi kada druga instanca ima *isti ili duži životni vijek* od druge instance[6]. Neposjedovanu referencu označavamo ključnom riječi *unowned* ispred svojstva ili deklaracije varijable.

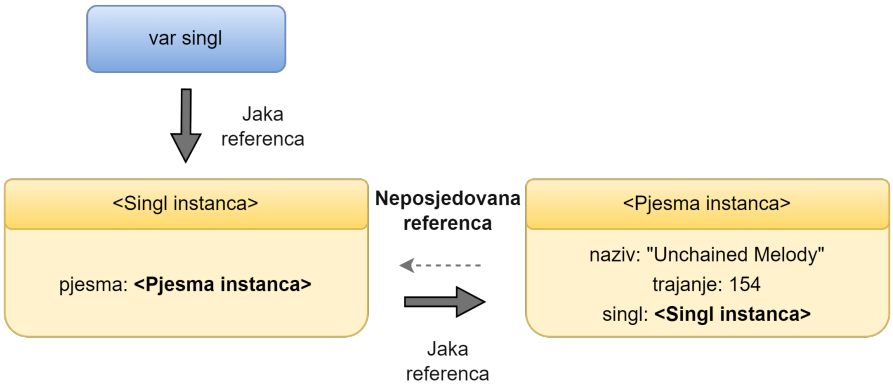
Za razliku od slabe reference, od neposjedovane reference se očekuje da uvijek ima vrijednost. Kao rezultat toga, označavanje svojstva kao neposjedovano ne čini ga izbornim, a što opet znači da ABR nikada neće postaviti vrijednost neposjedovane reference na *nil* kao što je to slučaj sa slabim referencama.

Recimo da želimo modelirati glazbeni singl koji kod kreiranja ne mora sadržavati pjesmu. Međutim, ako singl sadrži *Pjesmu* te obrišemo singl, želimo da se i pjesma automatski obriše s njim. To možemo postići tako da instanca *Singl* ima jaku referencu na pjesmu, a instanca *Pjesma* neposjedovanu referencu na *Singl*:

```
1  import Foundation
2
3  class Pjesma {
4      let naziv: String
5      var trajanje: Int
6      unowned let singl: Singl    // neposjedovana referenca
7
8      init(naziv: String, trajanje: Int, singl: Singl) {
9          self.naziv = naziv
10         self.trajanje = trajanje
11         self.singl = singl
12         print("Pjesma \(self.naziv) je kreirana.")
13     }
14
15     deinit {
16         print("Pjesma \(self.naziv) je obrisana.")
17     }
18 }
19
20 class Singl {
21     var pjesma: Pjesma?
22
23     init() {
24         print("Singl je kreiran.")
25     }
26
27     deinit {
28         print("Singl je obrisan.")
29     }
30 }
31
32 var singl: Singl? = Singl() // Singl je kreiran.
```

```
33 singl?.pjesma = Pjesma(
34     naziv: "Unchained Melody",
35     trajanje: 154,
36     singl: singl!
37     )   // Pjesma Unchained Melody je kreirana.
38 
39 singl = nil
40 // Singl je obrisan.
41 // Pjesma Unchained Melody je obrisana.
```

Stanje prije dealokacije instance *Singl* možemo prikazati sljedećim dijagramom:



Slika 6: Odnos klasa *Singl* i *Pjesma* pri korištenju neposjedovane reference *(samostalna izrada, 2022)*

Nakon što postavimo vrijednost varijable *singl* na *nil* (dealociramo na liniji 39), *Singl* instanca gubi sve jake reference te se dealocira iz memorije uz ispis *"Singl je obrisan."*.

Međutim, kako se *Singl* instanca dealocirala iz memorije, instanca *Pjesma* je posljedično također izgubila sve jake reference pa se i ona dealocira iz memorije uz poruku *"Pjesma Unchained Melody je obrisana."*. Završno stanje, gdje su sve instance uspješno dealocirane prikazano je na dijagramu u nastavku.

Slika 7: Dealokacija instance *Pjesma (samostalna izrada, 2022)*

4. Asinkrono programiranje

Većina modernih iOS aplikacija mora pokretati kôd asinkrono. Asinkroni kôd može se pauzirati i nastaviti kasnije te omogućava da aplikacija zadrži responzivnost svog korisničkog sučelja dok radi na dugim kompleksnim zadacima, poput izvršavanja mrežnih zahtjeva za REST API. Shodno tome, asinkroni kôd često izvodimo paralelno kako bismo najbolje iskoristili resurse kao što su jezgre procesora i internetska propusnost.

Koncept *async await* dio je novih strukturiranih promjena konkurentnosti koje su stigle u Swift 5.5 tijekom WWDC 2021[7]. Konkurentnost u Swiftu znači dopuštanje da se više dijelova kôda izvodi u isto vrijeme. Ovo je vrlo pojednostavljen opis, ali bi vam trebao dati ideju koliko je konkurentnost u Swiftu važna za performanse iOS aplikacija. S novim *async await* konceptom i naredbama čekanja možemo definirati metode koje obavljaju posao asinkrono - asinkrone metode (*asynchronous methods*).

4.1. Asinkrone funkcije

U Swiftu se asinkrone funkcije označavaju ključnom riječi *async* nakon naziva funkcije, ali prije povratnog tipa. Na primjer, ovo je deklaracija *data(from:delegate:)* metode *URLSession* iz iOS SDKa[8]:

```
1 func data(
2     from url: URL,
3     delegate: URLSessionTaskDelegate? = nil
4 ) async throws -> (Data, URLResponse)
```

Asinkrona funkcija može bacati iznimke kao i svaka druga funkcija. U ovom će slučaju, *data(from:delegate:)* metoda baciti iznimku kada postoje problemi s mrežnim prijenosom.

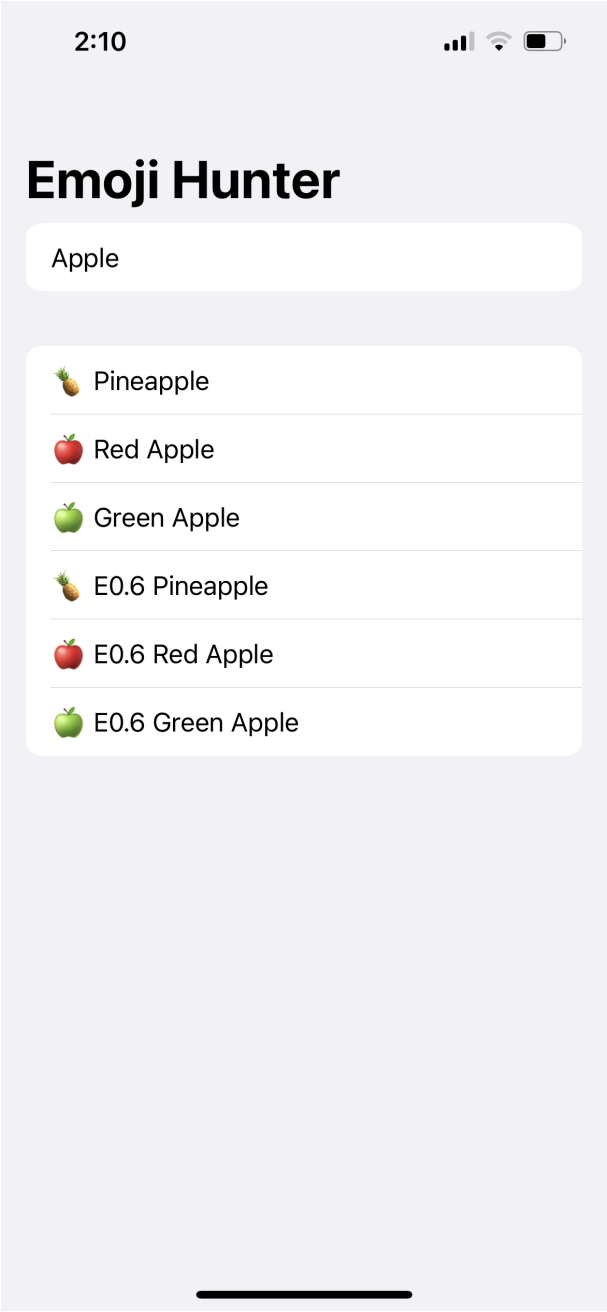
Većinom zapravo koristimo već implementirane asinkrone funkcije iz iOS SDKa ili drugih okvira, tako da se ovdje nećemo fokusirati na samu srž rada asinkronih funkcija (kako su one interno implementirane), već ćemo naglasak staviti na to kako napraviti svoje jednostavne asinkrone funkcije i pozivati već postojeće.

4.2. Emoji Hunter primjer

Zbog same kompleksnosti asinkronih funkcija i paralelnosti koja se događa u pozadini, smatram da je najbolje početi s primjerom, a kojeg ćemo postupno analizirati i objasniti što se točno dešava.

Reprezentativna aplikacija *Emoji Hunter* sastoji se od jednog pogleda gdje je dostupno polje za unos naziv emojija koje, kada se pritisne *enter*, pokreće asinkroni poziv prema vanjskom web servisu. Rezultati pretraživanja emojija po nazivu su zatim asinkrono učitani i prikazani u listu ispod polja pretrage.

Za pretraživanje emojija koristi se vanjski web servis Open Emoji API koji je besplatan (potrebno se samo registrirati za API ključ) i vrlo jednostavan za uporabu. Kada korisnik unese ključne riječi (u ovom slučaju *Apple*) i pritisne *enter* dobiva sljedeći prikaz:



Slika 8: Primjer pretraživanja emojija *(samostalna izrada, 2022)*

4.2.1. Modeliranje odgovora

Najprije je potrebno kreirati strukturu (ili klasu) koja će nam omogućiti mapiranje JSON odgovora s web servisa u naše strukture podataka. Navedena struktura mora implementirati protokol *Codable* kako bi nam omogućila lakše kodiranje i dekodiranje, odnosno mapiranje JSON odgovora s web servisa u naše Swift strukture, to jest svojstva. U nastavku je prikazan

primjer JSON odgovora s Open Emoji API kod pretraživanja pojma *Apple* te Swift struktura *Emoji* koja ga opisuje.


```
[
  {
    "slug": "pineapple",
    "character": "\ud83c\udf4d",
    "unicodeName": "pineapple",
    "codePoint": "1F34D",
    "group": "food-drink",
    "subGroup": "food-fruit"
  },
  {
    "slug": "red-apple",
    "character": "\ud83c\udf4e",
    "unicodeName": "red apple",
    "codePoint": "1F34E",
    "group": "food-drink",
    "subGroup": "food-fruit"
  },
  {
    "slug": "green-apple",
    "character": "\ud83c\udf4f",
    "unicodeName": "green apple",
    "codePoint": "1F34F",
    "group": "food-drink",
    "subGroup": "food-fruit"
  },
  {
    "slug": "e0-6-pineapple",
    "character": "\ud83c\udf4d",
    "unicodeName": "E0.6 pineapple",
    "codePoint": "1F34D",
    "group": "food-drink",
    "subGroup": "food-fruit"
  },
  {
    "slug": "e0-6-red-apple",
    "character": "\ud83c\udf4e",
    "unicodeName": "E0.6 red apple",
    "codePoint": "1F34E",
    "group": "food-drink",
    "subGroup": "food-fruit"
  },
  {
    "slug": "e0-6-green-apple",
    "character": "\ud83c\udf4f",
    "unicodeName": "E0.6 green apple",
    "codePoint": "1F34F",
    "group": "food-drink",
    "subGroup": "food-fruit"
  }
]
```


```
1  struct Emoji: Codable {
2      let slug: String
3      let character: String
4      let unicodeName: String
5      let codePoint: String
6      let group: String
7      let subGroup: String
8  }
```

Slika 9: Primjer JSON odgovora s Open Emoji API web servisa *(samostalna izrada, 2022)*

4.2.2. Asinkrono slanje HTTP GET zahtjeva

Za pretraživanje emojija kreirana je sljedeća asinkrona funkcija unutar pogleda:

```
1  func fetchEmojis(by keywords: String) async -> Void {
2      let keywordParam = keywords
3              .split(separator: " ")
4              .joined(separator: "-")
5      let fullUrl = "\(baseURL)?search=\(keywordParam)&\(accessKey)"
6
7      guard let url = URL(string: fullUrl ) else {
8          return
9      }
10
11     do {
12         let (data, _) = try await URLSession.shared.data(from: url)
13         emojis = try JSONDecoder().decode([Emoji].self, from: data)
14     } catch {
15         return
16     }
17 }
```

Funkcija *fetchEmoji(by:)* prima ključne riječi pretraživanja kao argument. Deklarirana je kao asinkrona i ne vraća ništa, već dohvaćene podatke sprema u *emoji* polje koje se nalazi u samom pogledu:

```
1  @State var emojis: [Emoji] = []
```

Pošto u URLu ne možemo poslati razmak, najprije je potrebno ključne riječi razdvojiti po razmaku i zatim ih spojiti povlakom (linije 2-4). Nadalje kreiramo punu putanju koristeći *baseURL* i *accessKey* vrijednosti, također inicijalizirane u pogledu (linija 5).

```
1  let baseURL = "https://emoji-api.com/emojis"
2  let accessKey = "access_key=e51a0758f01cbf378fa12b0f8a54b667c78de1df"
```

Pomoću *guard* izraza kreiramo i odmah provjeravamo je li putanja važeća, odnosno može li se pretvoriti u *URL* objekt (linije 7-9). Ako pretvorba nije uspjela, zaustavljamo pretraživanje i prekidamo izvođenje funkcije.

Za osnovne zahtjeve, klasa *URLSession* nam pruža dijeljeni *singleton* objekt sesije koji je već predefiniran za slanje jednostavnih zahtjeva[8]. Nad objektom sesije možemo pozvati asinkronu metodu *data(from:)* koja će dohvatiti sadržaj sa zadanog URLa i spremit ga u memoriju (linija 12)[1]. Naravno, kako je metoda asinkrona, potrebno je *pričekati da dohvaćanje završi* koristeći *await* izraz.

Nakon što smo sadržaj HTTP odgovora uspješno spremili u memoriju, potrebno ga je pretvoriti u našu prethodnu definiranu strukturu *Emoji* kako bi nam daljnje rukovanje tim podacima bilo lakše.

[1] Metoda vraća i *URLResponse objekt koji nam u ovom slučaju nije potreban pa ga odbacujemo koristeći donju povlaku (_).*

Pretvaranje *Data* objekta u našu prethodno definiranu strukturu *Emoji* najlakše možemo ostvariti koristeći *JSONDecoder* objekt i njegovu metodu *decode(_:from:)* koja kao prvi argument prima tip podatka u koji želimo *Data* objekt, iz drugog argumenta, pretvoriti (linija 13)[9].

Dohvaćanje sadržaja HTTP GET zahtjeva u memoriju i pretvaranje istog u našu strukturu može *"puknuti"* na puno mjesta - prekid mreže, predugo čekanje odgovora servisa, dohvaćanje nevažećih podataka, neuspjelo pretvaranje u *Emoji* objekt i sl. Zbog svega toga, te su dvije instrukcije smještene unutar *do-try-catch* bloka kojim hvatamo sve iznimke koje se mogu desiti kako ne bismo srušili aplikaciju.

4.2.3. Asinkrono osvježavanje pogleda

U pogledu najprije definiramo praznu String varijablu *searchQuery* u koju ćemo, u realnom vremenu, spremati sadržaj koji korisnik unese u polje za unos.

```
1  @State var searchQuery: String = ""
```

Zatim možemo kreirati samo polje za unos (*TextField*) i nad njim hvatati događaj pritiska tipke enter (*done* / *return*) pomoću metode *onSubmit(of:_:)* koja je dostupna od iOS verzije 15.

Međutim, kada bismo direktno nad uhvaćenim događajem pozvali asinkronu funkciju, blokirali bismo izvođenje programa na duže vrijeme - glavna dretva ne bi više mogla osvježavati ekran i korisnik ne bi mogao ništa kliknuti. Zbog toga i sama metoda *onSubmit(of:_:)* nije asinkrona, to jest, ne omogućava nam izravno pozivanje asinkronih funkcija već nas tjera da koristimo drugu dretvu gdje možemo pozivati asinkrone funkcije i *čekati* ih da završe.

Jedno moguće rješenje je iskoristiti strukturu *Task* koja predstavlja *jedinicu asinkronog rada*[10]. Njoj možemo proslijediti izvršavanje *fetchEmojis(by:)* funkcije sa *searchQuery* parametrom, pričekati da dohvaćanje završi i obrisati polje unosa.

```
1  TextField("Search emoji", text: $searchQuery)
2      .onSubmit {
3          if searchQuery.count != 0 {
4              Task {
5                  await fetchEmojis(by: searchQuery)
6                  searchQuery = ""
7              }
8          }
9      }
```

Nakon uspješnog dohvaćanja emojija, Swift automatski detektira promjenu u varijabli *emojis* i ažurira pogled (listu) s novim podacima:

```
1  ForEach(emojis) { emoji in
2      let emojiDisplay = String(
3          UnicodeScalar(
4              Int(emoji.codePoint.split(separator: " ")[0], radix: 16)!
5          )!
6      )
7      Text("\(emojiDisplay) \(emoji.unicodeName.capitalized)")
8  }
```

5. Komunikacija prema sklopovlju

Za rad sa hardverom iOS uređaja, Swift nam pruža *Core Motion* okvir. Core Motion nam omogućava izravnu komunikaciju s ugrađenim hardverom iOS uređaja[11]. Odnosno, možemo pristupiti svim senzorima iOS uređaja - pedometar, magnetometar, barometar i sl.

Core Motion okvir koristi se za pristup hardverski-generiranim podacima pomoću kojih možemo dodatno poboljšati korisničko iskustvo same iOS aplikacije. Na primjer, igra može koristiti podatke akcelerometra i žiroskopa za upravljanje ponašanjem igre na zaslonu.

Pojedinim senzorima iOS uređaja, poput senzora za vibracije, može se pristupiti i bez korištenja cijelog okvira kao što je Core Motion. Jedan primjer implementacije detekcije pokreta bez korištenja Core Motion okvira prikazan je u nastavku.

5.1. Detekcija pokreta

U ovom poglavlju fokusirat ćemo se na jednostavnu detekciju pokreta trešnje iOS uređaja, bez korištenja Core Motion okvira. Izgradit ćemo jednostavnu aplikaciju koja, kada detektira početak trešnje iOS uređaja, prikazuje obavijest da se iOS uređaj počeo tresti.

Kako bismo detektirali različite pokrete, klasa *UIResponder* nam pruža mnogo različitih metoda. *UIApplication*, *UIViewController*, *UIView* i *UIWindow* su sve primjeri *UIResponder* objekata, što znači da svi mogu detektirati događaje pokreta (*motion events*) bez posebne konfiguracije. Upravo jedan od tih pokreta je i pokret vibracije (*shake gesture*).

UIKit okvir obavještava odgovornu instancu (*responder*) samo kada događaj pokreta počinje i završava - svi podaci između početka i kraja su odbačeni te im ne možemo pristupiti putem UIKit okvira, već moramo koristiti Core Motion okvir[12]. Događaji pokreta isporučuju se inicijalno odgovornoj instanci koja prva reagira i prosljeđuju niz lanac odgovornih instanci, a koje potom mogu reagirati po potrebi.

Tri metode koje su nam ključne za razumijevanje, a koje služe za prosljeđivanje događaja pokreta niz lanac odgovornih instanci su:

1. detekcija početka pokreta,

```
1 func motionBegan(
2     _ motion: UIEvent.EventSubtype,
3     with event: UIEvent?
4 )
```

2. detekcija kraja pokreta i

```
1 func motionEnded(
2     _ motion: UIEvent.EventSubtype,
3     with event: UIEvent?
4 )
```

3. detekcija prekida pokreta.

```
func motionCancelled(
    _ motion: UIEvent.EventSubtype,
    with event: UIEvent?
)
```

Obično aplikacije koriste kraj pokreta na temelju kojeg odrede sljedeću akciju. No, mi ćemo u ovom primjeru iskoristiti sâm početak pokreta vibracije iOS uređaja (u ovom slučaju iPhone) kako bismo odmah obavijestili korisnika da je pokret detektiran.

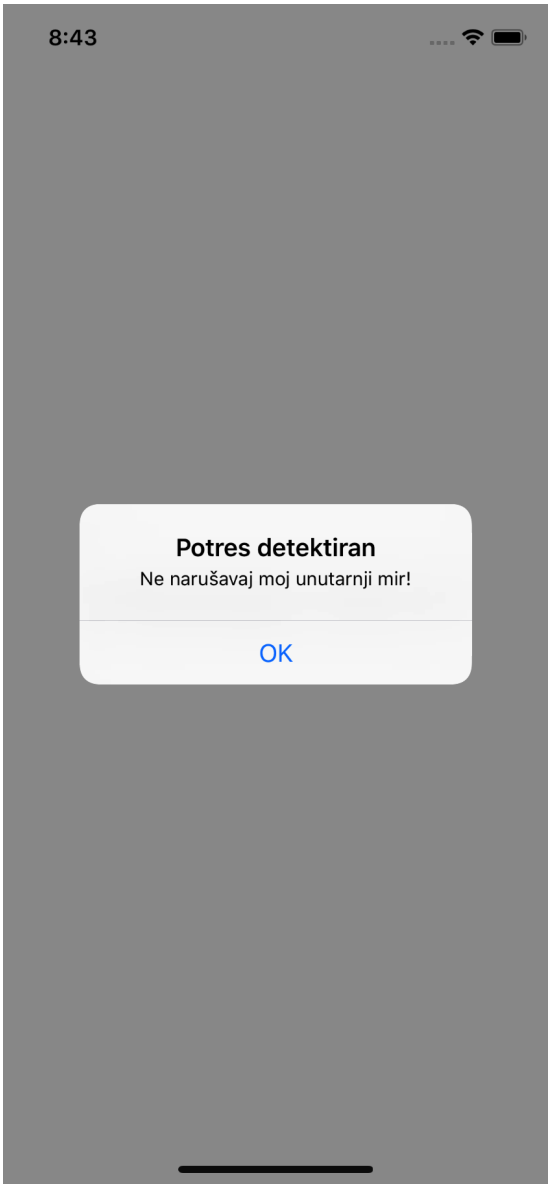
Kao što je prethodno spomenuto, jedna od klasa koje nasljeđuju klasu *UIResponder* je *UIViewController*. Shodno tome, jedino što nam je potrebno implementirati, odnosno nadjačati (*override*), jest metodu *motionBegan(:with:)*. Primjer implementacije prikazan je u nastavku.

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
    }

    // Pokreni ovu metodu kada je detektiran bilo koji pokret mobitela
    override func motionBegan(
        _ motion: UIEventSubtype,
        with event: UIEvent?) {
        // Saznaje vrstu pokreta
        if motion == .motionShake {
            // Kreira instancu alert kontrolera
            let alert = UIAlertController(
                title: "Potres detektiran",
                message: "Ne narusavaj moj unutarnji mir!",
                preferredStyle: .alert
            )

            // Pridruzuje akciju OK (gumb)
            alert.addAction(
                UIAlertAction(
                title: "OK",
                style: UIAlertActionStyle.default,
                handler: nil)
            )

            // Prikazuje alert na trenutnom pogledu
            self.present(alert, animated: true, completion: nil)
```

```
36                 }
37             }
38 }
```

Slika 10 prikazuje početni zaslon kod pokretanja aplikacije, a slika 11 obavijest koja se javila čim je aplikacija, odnosno iPhone, detektirao pokret vibracije.[1]

Slika 10: Početni zaslon *(samostalna izrada, 2022)*



Slika 11: Obavijest o detekciji pokreta trešnje mobitela *(samostalna izrada, 2022)*

[1] Ukoliko koristite iOS emulator, pokret vibracije može se simulirati putem izbornika *Hardware -> Shake Gesture*

6. Zaključak

Programski jezik Swift razvija se i poboljšava iz dana u dan već 8 godina. Stoga je za poznavanje svih konstrukata jezika i najbolje korištenje istih u semantički korektnom načinu potrebno mnogo godina iskustva. U ovom su priručniku izdvojeni samo neki od naprednijih programskih koncepata koji malo više zadiru u samu srž jezika i iskorištavaju njegovog pun potencijal.

Osim spomenutih poglavlja naprednog objektnog programiranja, asinkronog programiranja, upravljanja memorijom i malo dotaknute komunikacije prema sklopovlju uređaja, za nastavak i detaljnije istraživanje mogućnosti jezika i dobrih praksi preporučam samu dokumentaciju Swift jezika. Vrlo je lako čitljiva i napisana od samih kreatora Swift jezika pa sigurno sadrži sve potrebne specifikacije i detalje koji Vas mogu zanimati, odnosno koji Vam mogu zatrebati. Neka od poglavlja koja bih izdvojio su funkcionalno programiranje, konkurentnost i paralelnost te detaljno upravljanje sigurnostima memorije.

Popis literature

[1] Apple. „Structures and Classes.” (4. lipnja 2018.), adresa: https://docs.swift.org/swift-book/LanguageGuide/ClassesAndStructures.html (pogledano 22. 10. 2022.).

[2] Apple. „Initialization.” (4. lipnja 2018.), adresa: https://docs.swift.org/swift-book/LanguageGuide/Initialization.html (pogledano 22. 10. 2022.).

[3] Apple. „Properties.” (4. lipnja 2018.), adresa: https://docs.swift.org/swift-book/LanguageGuide/Properties.html (pogledano 23. 10. 2022.).

[4] Apple. „Methods.” (4. lipnja 2018.), adresa: https://docs.swift.org/swift-book/LanguageGuide/Methods.html (pogledano 24. 10. 2022.).

[5] Apple. „Deinitialization.” (3. lipnja 2014.), adresa: https://docs.swift.org/swift-book/LanguageGuide/Deinitialization.html (pogledano 29. 10. 2022.).

[6] Apple. „Automatic Reference Counting.” (2. lipnja 2014.), adresa: https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html (pogledano 29. 10. 2022.).

[7] Apple. „Meet async/await in Swift - WWDC21.” (8. lipnja 2021.), adresa: https://developer.apple.com/videos/play/wwdc2021/10132/ (pogledano 12. 11. 2022.).

[8] Apple. „Documentation - URLSession.” (7. lipnja 2021.), adresa: https://developer.apple.com/documentation/foundation/urlsession (pogledano 12. 11. 2022.).

[9] Apple. „Documentation - JSONDecoder.” (7. lipnja 2017.), adresa: https://developer.apple.com/documentation/foundation/jsondecoder (pogledano 12. 11. 2022.).

[10] Apple. „Documentation - Task.” (8. lipnja 2021.), adresa: https://developer.apple.com/documentation/swift/task (pogledano 12. 11. 2022.).

[11] Apple. „Core Motion Framework.” (5. lipnja 2017.), adresa: https://developer.apple.com/documentation/coremotion (pogledano 30. 10. 2022.).

[12] Apple. „Documentation - UIResponder.” (5. lipnja 2017.), adresa: https://developer.apple.com/documentation/uikit/uiresponder (pogledano 30. 10. 2022.).

Korišteni alati: Swift (v.*5.7*), XCode (v.*14.1*), Git, GitHub, Diagrams.net, LaTeX

Popis slika