УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ


Марко Вељковић

# КРЕИРАЊЕ ВИЦЕТА У ПРОГРАМСКОМ ЈЕЗИКУ SWIFT

мастер рад

Београд, 2022.

**Ментор**:

др Милена Вујошевић Јаничић, ванредни професор
Универзитет у Београду, Математички факултет

**Чланови комисије**:

др Филип Марић, ванредни професор
Универзитет у Београду, Математички факултет

др Мирко Спасић, доцент
Универзитет у Београду, Математички факултет

**Датум одбране**: 15. јануар 2022.

**Наслов мастер рада**: Креирање вицета у програмском језику Swift

**Резиме**: Употреба програмског језика *Swift* за израду вицета у *iOS* оперативном систему

**Кључне речи**: *widget*, *Swift*, *iOS*, *Apple*, програмски језик, програмирање

# Садржај

# Глава 1

# Увод

# Глава 2

# Програмски језик Swift

*Swift* је модеран програмски језик, првенствено намењен развоју апликација на *Apple* платформама (*iOS*, *iPadOS*, *macOS*, *tvOS* и *watchOS*). Настао је као резултат истоименог пројекта унутар компаније *Apple*, чији је циљ био креирање програмског језика који ће бити сигуран, концизан и ефикасан. Резултати пројекта *Swift* биће приказани у наставку.

## 2.1 Настанак и развој

Развој програмског језика *Swift* започео је Крис Латнер (енг. Chris Lattner) [1] у јулу 2010. године. У јуну 2014. године објављена је прва апликација комплетно написана у *Swift*-у, названа „Apple Worldwide Developers Conference" (WWDC) по истоименој годишњој конференцији информационих технологија компаније *Apple*. На конференцији те године, кроз предавање и интерактивну демонстрацију представљена је бета верзија језика, „The Swift Programming Language" [6] - бесплатно упутство и званична веб страница [7]. Прва званична верзија језика *Swift* 1.0, постала је доступна 9. септембра 2014. године.

Званично објављивање апликација (на *App Store*-у) писаних у *Swift*-у постало је могуће од верзије програмског језика 2.0. Језик је у оквиру анкете коју организује *Stack Overflow* [8] проглашен за омиљени програмски језик 2015. године, док је 2016. године заузео друго место у тој категорији. Децембра 2015. изворни код језика, подржаних библиотека, дибагер и менаџер

[1] Софтверски инжењер, најпознатији по развоју *LLVM* технологија, *Clang* компајлера и *Swift* програмског језика

пакета постали су отвореног кода, под лиценцом *Apache* 2.0, доступни на *GitHub*-у [3].

На годишњој конференцији 2016. године представљена је апликација за уређај *iPad* под називом *Swift Playgrounds* [4], која је намењена учењу програмирања у *Swift*-у. Касније је ова апликација развијена и за оперативни систем *macOS*.

Током конференције 2019. године представљено је радно окружење *SwiftUI* [5], које омогућава декларативно програмирање апликација за све *Apple* платформе. У време писања рада, последња званична верзија језика је *Swift* 5.6.

## 2.2 Основна намена и карактеристике


Слика 2.1: *Swift лого*

*Swift* је моћан и интуитиван језик за програмирање апликација намењних *Apple* платформама (*iOS*, *iPadOS*, *macOS*, *tvOS* и *watchOS*). Писање кода у *Swift*-у је забавно и лако, синтакса је веома концизна, али у исто време веома изражајна. Програмски језик *Swift* је безбедан, брз и интерактиван, и као такав погодан за људе који уче основе програмирања. Код писан у *Swift*-у се преводи и оптимизује тако да извуче максимум из хардверских компоненти. Детаљнији опис особина и примери кодирања биће дати у наредним одељцима.

## 2.3 Концепти

*Swift* је наследник програмских језика *C* и *Objective-C*, па је самим тим одређене концепате преузео из ових језика, али истовремено постоје концепти у *Swift*-у који нису присутни у *C*-у и *Objective-C*-у. Објашњење најбитнијих концепата у *Swift*-у дат је у наставку, док се потпуна листа може наћи на званичком сајту програмског језика. [7]

## Основе

*Swift* пружа своје типове основних променљивих, *Int* за целобројне вредности, *Float* и *Double* за бројеве са основом у покретном зарезу, *Bool* за Булове вредности, *String* за текстуалне променљиве, као и три основна типа колекција *Array*, *Set* и *Dictionary* о којима ће бити више речи у делу 2.3 Колекције.

Поред ових основних типова који су наслеђени из *Objective-C*-а, постоје и неколико ново уведених, као што су *Tuples* који омогућују креирање и прослеђивање груписаних вредности и *Optionals* помоћу којих рукујемо *nil* вредношћу на безбедан начин.

Константе декларишемо коришћењем кључне речи *let*, након чега следи име константе и затим њена иницијализација. Декларисање променљиве постижемо употребом кључне речи *var*. Када декларишемо променљиво можемо је одмах и иницијализовати или коришћењем анотације јој само доделити тип. Конкретна примена се може видети у примеру 2.1 - *Декларисања променљивих и константи*.

```
1 // Deklarisanje konstantne
2 let spageteKarbonaraRecept = Recept("Spagete Karbonara")
3
4 // Deklarisanje promenljive uz inicijalizaciju
5 var raspolozivoNovca = 1000
6
7 // Deklarisanje promenljive koriscenjem samo anotacije
8 var brojPorcija: Int
```

Listing 2.1: *Декларисања променљивих и константи*

Целобројне променљиве могу бити написане у облику децималних, бинарних, октални и хексадецималних бројева. Пример следи у наставку 2.2 - *Целобројне променљиве*.

```
1 let mojiBroj = 25
2 let binarniBroj = 0b11001     // 25 u binarnom obliku
3 let oktalniBroj = 0o31        // 25 u oktalnom obliku
4 let heksadecimalniBroj = 0x19 // 25 u heksadecimalnom obliku
```

Listing 2.2: *Целобројне променљиве*

*Tuple* група се користи за груписање променљивих било ког типа. Група може садржати променљиве различитих типова. У примеру 2.3 - *Tuple*, могу

се видети два начина креирања *Tuple* група и различити начини приступања члановима групе.

```
1  // Definisanje tuple-a (String, Int)
2  let namirnicaSaCenom = ("Jaja 10 komada", 100)
3
4  // Pristupanje clanovima tuple-a:
5  let namirnica = namirnicaSaCenom.0
6  let cena = namirnicaSaCenom.1
7
8  // Definisanje tuple-a sa imenovanim clanovima
9  let urednijaNamirnicaSaCenom = (namirnica: "Jaja 10 komada", cena:
       100)
10
11 // Pristupanje imenovanim clanovima tuple-a::
12 let urednijaNamirnica = urednijaNamirnicaSaCenom.namirnica
13 let urednijaCena = urednijaNamirnicaSaCenom.cena
```

Listing 2.3: *Tuple*

*Assertions*

## Оператори

Као и у већини програмских језика, постоје три основне врсте оператора:

- Унарни
  - Префиксни унарни оператори (на пример, '-' за бројевне вредности и '!' за логичке вредности)
  - Постфиксни унарни оператори (на пример, '?' и '!' који се користе за опционе променљиве)
- Бинарни
  - Оператор доделе '=', који за разлику од језика C, не враћа повратну вредност
  - Артиметички оператори, '+', '-', '*', '/', '%'
  - Сложени оператори доделе, '+=', '-=', '*=', '/='
  - Оператори поређења, '==', '!=', '<', '>', '<=', '>='

- Тернарни
  Једини тернари оператор који постоји у језику је оператор '?:'. Код овог оператора израз са крајње леве стране мора бити типа *Bool*, док израз или променљива која се налази у средини оператора мора бити истог типа као израз или променљива са крајње десне стране, без ограничења типа. Пример примене тернарног оператора, као и приказ блока кода који једна линија са тернарним оператором може заменити приказани су у примеру 2.4 - *Тернарни оператор*.

```
1  // Ternarni operator:
2  a ? b : c
3
4  // Izraz ternarnog operatora
5  let visinaCelije = celijaImaSliku ? 100 : 50
6
7  // Prethodni primer koriscenjem uslovnog grananja
8  let visinaCelije: Int
9  if celijaImaSliku {
10     visinaCelije = 100
11 }
12 else {
13     visinaCelije = 50
14 }
```

Listing 2.4: *Тернарни оператор*

Поред основних оператора у *Swift*-у постоје и специјалне врсте оператора

- Оператор 'nil-сједињавања' ('??'), бинарни оператор који се користи над опционим променљивима. Уколико израз са леве стране оператора садржи вредност (није *nil*), та вредност ће бити резултат оператора, док уколико је лева страна оператора једнака *nil*, резултат оператора биће израз са десне стране, који мора садржати неку вредност. Пример примене '??' оператора и приказ његовог расписивања помоћу тернарног оператора дат је у наставку 2.5 - *nil-сједињавање*.

```
1      var a: Int?
2      if trenutnoVremeUMilisekundama % 2 == 0 {
3          a = 10
4      }
```

```
5     // Ukoliko promenljiva 'a' ne sadrzi vrednost, promenljivoj 'b
          ' bice dodeljena vrednost 5
6     let b = a ?? 5
7
8     // Raspisivanje operator 'nil-sjedinjavanja' koriscenjem
          ternarnog operatora:
9     let b = (a != nil ? a! : 5)
```

Listing 2.5: *nil-сједињавање*

- Оператори распона
  - Затвореног распона (a...b), распон од 'a' до 'б' укључујући оба броја
  - Полу-отвореног распона (a..<b), распон од 'a' до 'б' укључујући само 'a'
  - Распони једне стране [a...], распон од 'a' па надаље докле год је то могуће (Ову врсту оператора треба користити опрезно!)

## Карактери и стрингови

Стринг је низ карактера, као што је „Здраво, свете". У *Swift*-у се стрингови представљају помоћу класе *String*, која омогућава брз, ефикасан и *Unicode*-компатибилан начин рада са текстом. Креирање и операције са стринговима (конкатенација, рад са карактерима и интерполација) биће приказане кроз пример 2.6 - *Операције над стринговима*.

```
1     var prazanString = ""
2     var drugiPrazanString = String()
3     var treciPrazanString: String?
4
5     if !prazanString.isEmpty {
6         prazanString = "Zdravo"
7     }
8
9     // Konkatenacija stringova
10    drugiPrazanString += ", svete"
11
12    // Rad sa karakterima
13    for k in prazanString {
14        print(k)
```

```
15      }
16      // Ispisace:
17      // Z
18      // d
19      // r
20      // a
21      // v
22      // o
23
24      // Interpolacija stringova
25      print(\(prazanString) + \(drugiPrazanString) + "!")
26      // Ispisace 'Zdravo, svete!'
```

Listing 2.6: *Операције над стринговима*

## Колекције

*Swift* дефинише три примарна типа колекција: низове, скупове и речнике. Сва три типа су дефинисана као генеричке[2] колекције. Уколико се дефинисана колекција додели промењливој, она се може мењати (додавање, брисање и измена чланова у њој); међутим уколико се она додели некој константи, манипулација њеним члановима неће бити могућа.

Низови се користе за уређено чување елемената истог типа. Један елемент се може појавити у низу више пута, на различитим индексима. Конкретан пример дефинисања, иницијализације и управљањем података низа може се видети у коду 2.7 - *Рад са низовима.*

```
1       // Definisanje niza sa elementima tipa 'Recept'
2       var recepti: [Recept] = []
3       // Dodavanje novog elementa
4       recepti.append(Recept("Domaca kafa"))
5       // Pristupanje prvom clanu niza
6       var prviRecept = recepti[0]
7       // Kreiranje niza sa 3 inicijalna elementa tipa 'String'
8       var koraci = Array(repeating: "", count: 3)
9       // Foreach petlja kojom prolazimo kroz niz uz pamcenje indeksa
10      for (index, korak) in prviRecept.koraci.enumerated() {
```

[2]Генерички код омогућава писање флексибилних и поновно искористивих функција и типова; помоћу њих се избегава непотребно дуплирање кода

```
11          if index < 3 {
12              koraci[index] = korak
13          }
14          else {
15              koraci.append(korak)
16          }
17      }
18      // Brisanje prvog clana niza, ukoliko niz nije prazan
19      if !koraci.isEmpty {
20          koraci.remove(at: 0)
21      }
```

Listing 2.7: *Рад са низовима*

Скупови су колекције које не гарантују чување редоследа елемената и у којима један елемент може да се појави највише једанпут. Тип елемента скупа мора бити могуће кодирати[3] (енг. hashable). Рад са скуповима приказан је у примеру 2.8 - *Рад са скуповима.*

```
1       // Kreiranje skupa sa elementima tipa 'String'
2       var sastojci = Set<String>()
3       // Dodavanje novog elementa
4       sastojci.insert("Mlevena kafa")
5
6       // Provera broja elemenata skupa
7       if sastojci.count == 1 {
8           sastojci.insert("Obicna voda")
9       }
10
11      // Provera da li odredjeni element postoji u skupu
12      if sastojci.contains("Secer") {
13          sastojci.remove("Secer")
14      }
15      else {
16          sastojci.insert("Mleko")
17      }
18
19      var dodatniSastojci = Set<String>()
20      dodatniSastojci.insert("Mleko")
21
22      // Rad sa skupovnim operacijama
```

[3]Тип који се може кодирати мора имати дефинисану функцију за одређивање hash вредности за сваку инстанцу, два елемента могу имати исту hash вредност акко су једнаки

```
23
24      // Unija
25      sastojci.union(dodatniSastojci)
26      // Mlevena kafa, Obicna voda, Mleko
27
28      // Presek
29      sastojci.intersection(dodatniSastojci)
30      // Mleko
31
32      // Razlika
33      sastojci.subtracting(dodatniSastojci)
34      // Mlevena kafa, Obicna voda
35
36      // Disjunktivna unija
37      sastojci.symmetricDifference(dodatniSastojci)
38      // Mlevena kafa, Obicna voda
```

Listing 2.8: *Рад са скуповима*

Речници се користи за чување скупа парова кључ - вредност, без очувања редоследа. Свака вредност је додељена јединственом кључу, који мора бити погодан за кодирање (енг. hashable). Речници се најчешће користе за чување података којима је могућ брз приступ помоћу одговарајућег кључа. Рад са речницима приказан је у прмеру 2.9 - *Рад са речницима.*

```
1       // Kreiranje recnika tipa [Int : String]
2       var tipoviHTTPStatusa: [Int : String] = [:]
3
4       // Dodeljivanje recnika promenljivoj 'tipoviHTTPStatusa'
5       tipoviHTTPStatusa = [200: "OK", 201: "Resurs je kreiran", 202: "
            Zahtev je prihvacen"]
6
7       // Dodavanje novog elementa ukoliko ne postoji, odnosno promena
            postojeceg
8       tipoviHTTPStatusa[404] = "Stranica nije pronadjena"
9
10      // Brisanje elementa iz recnika
11      if let izbrisanaVrednost = tipoviHTTPStatusa.removeValue(forKey:
            201) {
12          print("Vrednost izbrisana iz recnika: \(izbrisanaVrednost)")
13      }
14
15      // Razlicite vrste iteracija kroz recnik
```

```
16
17      for kod in tipoviHTTPStatusa.keys {
18          // TODO: iteriraj kroz kodove
19      }
20
21      for status in tipoviHTTPStatusa.values {
22          // TODO: iteriraj kroz statuse
23      }
24
25      for (kod, status) in tipoviHTTPStatusa {
26          // TODO: iteriraj kroz elemente
27      }
```

Listing 2.9: *Рад са речницима*

## Контрола тока

Наредбе контроле тока које се користе у *Swift*-у су: *if*, *guard*, *switch* и петље: *for-in*, *while*. *If* наредба приказана је у примеру 2.10 - *If наредба контроле тока*.

```
1       var recept = Recept("Cezar salata")
2       recept.sastojci = ["Zelena salata", "Pilece grudi", "Slanina", "
            Paradajz", "Hleb", "Cezar premaz"]
3       // If naredba kondtrole toka
4       var brojSastojaka = recept.sastojci.count
5       if brojSastojaka < 6 {
6           print("Neki sastojak nedostaje")
7       }
8       else if brojSastojaka > 6 {
9           print("Broj sastojaka ne odgovara originalu, ali samo napred,
                eksperimentisi")
10      }
11      else {
12          print("Broj sastojaka je odgovarajuci")
13      }
```

Listing 2.10: *If наредба контроле тока*

*Switch* наредба је слична као у другим програмским језицима, једна битна разлика је да ће се увек извршити тачно један од случајева унутар наредбе, па није потребно експлицитно навођење *break* наредбе. *Break* наредба се по

конвенцији наводи само када је неки од случајева *Switch* наредбе празан, јер сваки случај мора бити извршив (енг. executable). Наведене ствари приказане су у примеру 2.11 - *Switch наредба контроле тока.*

```
1   enum Zacin {
2       case vegeta, kari, kurkuma, origano, biber
3   }
4
5   var mojiZacin: Zacin = .vegeta
6
7   // Switch naredba
8   switch mojiZacin {
9       case .vegeta:
10          print("Vegeta")
11      case .biber:
12          print("Nije vegeta, nego biber")
13      default:
14          print("Nije vegeta ni biber")
15  }
16
17  // Ukoliko ne navedemo sve slucajeve, moramo dodati slucaj '
        default' slucaj
18  switch mojiZacin {
19      case .origano:
20          print("Moze i to")
21      default:
22          break
23  }
```

Listing 2.11: *Switch наредба контроле тока*

*For-in* је наредба понављања која се користи за пролаз кроз елементе неке колекције (низа, скупа, речника). Променљива која се користи за пролаз кроз колекцију је увек константа и није је могуће мењати у телу наредбе. Један од начина на који можемо мењати елементе колекције (уколико је колекција дефинисана као променљива, коришћењем кључне речи *var*) је истовременим пролажењем кроз елементе и њихове индексе колекције и променом елемента колекције на одређеној позицији. Описани начине употребе петље могу се видети у примеру 2.12 - *For-in наредба контроле тока.*

```
1   let sastojci = ["Jaja", "Pecenica", "Maslinovo ulje", "Persun"]
2   // For-in naredba
3   for sastojak in sastojci {
```

```
4           print("Potreban sastojak: \(sastojak)")
5       }
6
7       let sastojciSaKolicinom = ["Jaja": "3 komada", "Pecenica": "50
            grama", "Maslinovo ulje": "Koliko je potrebno da pokrije tiganj
            ", "Persun": "Prstohvat"]
8       // For-in naredba za prolaz kroz recnik
9       for (sastojak, kolicina) in sastojciSaKolicinom {
10          print("Potreban sastojak: \(sastojak), u kolicini: \(kolicina)
                ")
11      }
12
13      var promenljiviElementi = ["Jaja", "Pecenica", "Maslinovo ulje", "
            Persun"]
14      // For-in naredba sa indeksiranjem
15      for (indeks, element) in promenljiviElementi.enumerated() {
16          if element == "Persun" {
17              promenljiviElementi[indeks] = "Origano"
18          }
19      }
```

Listing 2.12: *For-in наредба контроле тока*

Постоје два типа *while* наредбе понављања. *While* наредба која прво проверава да ли је задати услов испуњен и онда извршава тело петље и *repeat-while* наредба која прво извршава тело наредбе након чега проверава услов и уколико је он задовољен наставља са следећом итерацијом. Употреба је приказана у примеру 2.13 - *While наредба контроле тока*.

```
1       let nasumicniBrojevi = [3, 12, 5, 18, 11, 99]
2       var i = 0
3       // While naredba
4       while i < nasumicniBrojevi.count, nasumicniBrojevi[i] < 15 {
5           print("Broj \(nasumicniBrojevi[i] je manji od 15")
6           i += 1
7       }
8
9       let nasumicniBroj = nasumicniBrojevi[2] // 5
10      // Repeat-while petlja
11      repeat {
12          print("Zdravo, svete!")
13      } while nasumicniBroj != 5 // Uvek netacno
```

Listing 2.13: *While наредба контроле тока*

```
1    //TODO: continue, break, fallthrough, return, throw
```

Listing 2.14: *Додаци наредба контроле тока*

## Функције и затворења

Фунцкије су делови кода, који обично имају само једну, специфичну намену. Свака функција је идентификована својим именом које се користи да би се конкретна функција позивала у коду. Поред имена, функција може имати повратну вредност (уколико није дефинисана, подразумевана повратна вредност је *Void*) и пареметре (именоване или неименоване).

```
1    // Definisanje fukncije
2    func ispisiSastojke(sastojci: [String]) {
3        for sastojak in sastojci {
4            print(sastojak)
5        }
6    }
7
8    let sastojci = ["Jaja", "Sira"]
9    // Pozivanje f-je 'ispisiSastojke'
10   ispisiSastojke(sastojci: sastojci)
```

Listing 2.15: *Дефинисање и позивање функције са параметром*

```
1    // Definisanje funkcije koja ima povratnu vrednost 'Int' i jedan
         neimenovani parametar
2    func izracunajCenu(_ proizvodi[String: Int]) -> Int {
3        int ukupnno = 0
4        for (proizvod, cena) in proizvodi {
5            ukupno += cena
6        }
7        return ukupno
8    }
9
10   let proizvodi = ["Jaja": 10, "Sira": 200]
11   // Kada f-ja ima neimenovane parametre, njihova imena(labele) ne
         navodimo prilikom pozivanja te funkcije
12   let ukupnaCena = izracunajCenu(proizvodi)
```

Listing 2.16: *Дефинисање и позивање функције са повратном вредношћу*

```
// Definisanje fukncije koja nema povratnu vrednost, jedan imenovani parametar i jedan imenovani parametar (cija se labela razlikuje od imena promenljive) sa podrazumevanom vrednoscu
func func ispisiSastojkeSaDvaparametra(sastojci: [String], ispisati ispisatiCeloIme: Bool = true) {
    for sastojak in sastojci {
        if ispisatiCeloIme {
            print(sastojak)
        }
        else {
            print(sastojak.prefix(3))
        }
    }
}

// Iskoristicemo primer niza sastojaka kao za f-ju 'ispisiSastojke'
// Pozivanje f-je 'ispisiSastojkeSaDvaparametra' prosledjivanjem oba paremetra
// Ovde mozemo videti da se prilikom poziva f-je, ukoliko je neki parametar imenovan, navodi njegova labela
ispisiSastojkeSaDvaparametra(sastojci: sastojci, ispisati: false)
ispisiSastojkeSaDvaparametra(sastojci: sastojci, ispisati: true)

// Ukoliko ne navedemo drugi parametar 'ispisati', on ce u f-ji imati podrazumevanu vrednost, u ovom slucaju 'true'
ispisiSastojkeSaDvaparametra(sastojci: sastojci)
```

Listing 2.17: *Дефинисање и позивање функције са параметрима са подразумеваним вредностима*

```
// Definisanje funkcije koja ima povratnu vrednost 'Int' i jedan neimenovani parametar
func izracunajCenu(_ proizvodi[String: Int]) -> Int {
    int ukupnno = 0
    for (proizvod, cena) in proizvodi {
        ukupno += cena
    }
    return ukupno
}
```

```
9
10      let proizvodi = ["Jaja": 10, "Sira": 200]
11      // Kada f-ja ima neimenovane parametre, njihova imena(labele) ne
            navodimo prilikom pozivanja te funkcije
12      let ukupnaCena = izracunajCenu(proizvodi)
```

Listing 2.18: *Дефинисање и позивање функције са повратном вредношћу*

```
1       // Definisanje genericke funkcije koja vraca 2 vrednosti
2       func minMax<T>(niz: [T]) -> (min: T, max: T)? {
3           guard !niz.isEmpty else {
4               return nil
5           }
6           trenutniMin = niz[0]
7           trenutniMax = niz[0]
8
9           for element in niz[1..<niz.count] {
10              if element < trenutniMin {
11                  trenutniMin = element
12              }
13              else if element > trenutniMax {
14                  trenutniMax = element
15              }
16          }
17
18          return (trenutniMin, trenutniMax)
19      }
20
21      let nizBrojeva = [5, 12, -4, 19, -99]
22      let minimumIMaksimum = minMax(niz: nizBrojeva)
23      // Pristupamo povratnim vrednostima po labelama koje se nalaze u
            deklaraciji f-je
24      // Odnosno za minimum: minimumIMaksimum?.min, za maksimum:
            minimumIMaksimum?.max
```

Listing 2.19: *Дефинисање и позивање функције са више повратних вредности*

```
1       // Definisanje genericke funkcije koja prima 2 parametra i
            zamenjuje njihove vrednosti
2       // Osnovna stvar kod definisanja f-je je dodavanje kljucne reci '
            inout' pre tipa parametra za koji zelimo da omogucimo menjanje
            njegove vrednosti u f-ji
3       func zameniDvaParametra<T>(prvi: inout T, drugi: inout T) {
```

```
4           var pomocna = prvi
5           prvi = drugi
6           drugi = pomocna
7       }
8
9       // Moramo proslediti promenljive kao parametre, konstane (let) se
            ne mogu menjati
10      var prviString = "Ja sam prvi"
11      var drugiString = "Ja sam drugi"
12
13      print(prviString + ", " + drugiString)
14      // Bice ispisano: Ja sam prvi, Ja sam drugi
15
16      // Navodimo '&' pre imena promenljive, da naznacimo da njihove
            vrednosti mogu biti promenjene u telu f-je
17      zameniDvaParametra(prvi: &prviString, drugi: &drugiString)
18
19      print(prviString + ", " + drugiString)
20      // Bice ispisano: Ja sam drugi, Ja sam prvi
```

Listing 2.20: *Дефинисање и позивање функције са променљивим параметрима*

Затворења су самостални блокови кода који се могу прослеђивати и користити у коду. Слични су ламбда изразима у другим модерним језицима.

Изрази затворења представљају начин за писање затворења у једној линији (енг. inline), притом пружајући неколико синтаксних оптимизација, у виду кратке форме, разумљивости и изражајности. Показаћемо на примеру *Swift* метода *'sorted'*, како се једно затворење може написати на неколико начина, од целе ф-је, па све до само једног карактера.

```
1       func sortirajBrojeve(_ broj1: Int, _ broj2: Int) -> Bool {
2           return broj1 < broj2
3       }
4
5       let nasumicniBrojevi = [2, 10, 5, 18, 100, -11, -25, 55, 72]
6
7       var sortiraniBrojevi = nasumicniBrojevi.sorted(by: sortirajBrojeve
```

Listing 2.21: *Затворење кроз ф-ју*

Синтаксу израза затворења можемо генерално представити као:

```
1   { (parametri) -> tip povratne vrednosti in
2       naredbe
```

```
3 }
```

На основу овога, пример са сортирањем бројева можемо написати и овако:

```
1     sortiraniBrojevi = nasumicniBrojevi.sorted(by: { (broj1: Int,
          broj2: Int) -> Bool in
2         return broj1 < broj2
3     })
4
5     // Tipovi promenljivih u zatvorenju nemoraju biti eksplicitno
          navedeni, zato sto se odredjuju na osnovu tipa elemenata niza
          nad kojim se radi
6     sortiraniBrojevi = nasumicniBrojevi.sorted(by: {broj1, broj2 in
          return broj1 <  broj2})
7
8     // Kada u zatvorenju postoji samo jedna naredba, nije potrebno
          navodjenje kljucne reci 'return', povratna vrednost bice
          vrednost izvrsenja te naredbe
9     sortiraniBrojevi = nasumicniBrojevi.sorted(by: {broj1, broj2 in
          broj1 < broj2})
10
11    // Swift omogucava i kratka imena parametara, za pruzanje
          izrazajnije sintakse
12    // Imena su oznacena kao $0, $1... u zavisnosti od broja parametra
          u f-ji
13    sortiraniBrojevi = nasumicniBrojevi.sorted(by: { $0 < $1 })
14
15    // Kada koristimo tipove za koje je vec definisano ponasanje
          prilikom poredjenja, mozemo proslediti samo kako zelimo da
          sortiramo clanove niza
16    sortiraniBrojevi = nasumicniBrojevi.sorted(by: <)
```

Listing 2.22: *Израз затворења за сортирање*

Затворења се могу проследити и као параметри ф-је. Једино ограничење је да затворење мора ићи као последњи параметар ф-је. Најчешћи разлог за овакву употребу затворења је, да би били сигурни да ће се наредбе у затворењу извршити након што се заврши извршавање ф-је. Оваква врста затворења, назива се затворење трага (енг. Trailing closures).

```
1     func ucitajSliku(sa url: URL, completition: (Image?) -> Void) {
2         if let slika = skini("Omlet.jpg", sa: url) {
3             completition(slika)
4         }
```

```
5           else {
6               completition(nil)
7           }
8       }
9
10      ucitajSliku(sa: lokalniUrl) { slika in
11          if let slika = slika {
12              celija.image = slika
13          }
14          else {
15              celija.image.backgroundColor = .gray
16          }
17      }
```

Listing 2.23: *Затворење тра̄а*

## Класе и структуре

Класе и структуре су конструкције опште намене које имају своја својства и методе. За разлику од већине других програмских језика, класе и структуре у *Swift*-у су много сличније што се функционалности тиче, па се често за инстанцу класе, као и структуре, користи назив инстанца, уместо уобичајеног, објекат.

У поређењу класа и структура, ствари које се могу радити са обе: дефинисање својстава, дефинисање метода, дефинисање иницијализатора, могу се надограђивати коришћењем проширења (енг. extensions), имплементирати протоколе. Оно у чему се разликују, односно функционалности које поседују само класе су: наслеђивање друге класе, провера типа инстанце у времену извршавања програма, деиницијализација.

Уколико неко својство нема унапред дефинисану вредност, оно мора бити:

- Део иницијализације ако је константно
- Део иницијализације или бити опционог типа ако је промењиво

Дефинисање класе и структуре, инстанцирање као и приступање својствима и методама коришћењем тачка синтаксе (eng. dot syntax) може се видети у примеру 2.24 - *Дефинисање класе и структуре*.

```
1   // Definisanje strukture
```

```
2     struct OkvirPozadine {
3         var visina = 0
4         var sirina = 0
5         var boja: UIColor?
6     }
7
8     // Definisanje klase
9     class GlavniIzgled {
10        var okvir = OkvirPozadine()
11        var slika: UIImage?
12        var ponovitiSliku = false
13    }
14
15    // Instanciranje strukture
16    let okvir = OkvirPozadine()
17    // Instanciranje klase
18    let glavniIzgeld = GlavniIzgled()
19
20    // Pristupanje clanovima instance
21    let okvirGlavnogIzgleda = glavniIzgled.okvir
22    let sirinaOkviraPozadine = okvir.sirina
23
24    // Strukture imaju automatski generisane inicijalizatore za sva
          svojstva
25    let maliSiviOkvir = OkvirPozadine(visina: 50, sirina: 50, boja: .
          gray)
```

Listing 2.24: *Дефинисање класе и структуре*

Уколико унутар класе имамо инстанцу неке друге класе, али не желимо да се инстанцирање обави одмах на почетку, већ непосредно пре употребе те инстанце, инстанци можемо додати лењо својство (енг. lazy propertie). Лења својства можемо користити када инстанцирање класе зависи од других параметара који нису познати у тренутку иницијализације главне класе или када инстанцирање може узети много времена и добро је одложити га док не буде апсолутно неопходно (можда у неким случајевима не буде уопште искоришћено). Пример употребе лењог својства налази се у примеру 2.25 - *Лења својства.*

```
1     class UcitavanjeFajla {
2         var imeFajla = "recepti.txt"
3     }
```

```
4
5       class MenadzerPodataka {
6           lazy var ucitavanje = UcitavanjeFajla()
7           var podaci: [String] = []
8       }
9
10      var menadzer = MenadzerPodataka()
11      menadzer.podaci.append("Prvi podatak")
12      menadzer.podaci.append("Drugi podatak")
13      // U ovom trenutku klasa 'UcitavanjeFajla' i dalje nije
            instancirana
14
15      // Pre izvrsenja f-je 'print', instancira se klasa '
            UcitavanjeFajla'
16      print(menadzer.ucitavanje.imeFajla)
```

Listing 2.25: *Лења својства*

Методе су функције које су везане за одређени тип, било класе, структуре или набрајања (енг. enumerations). Методе инстанце су функције које припадају одређеној инстанци и подржавају функционалности те инстанце. Креирање и коришћење метода инстанце приказано је у примеру 2.26 - *Методе*.

```
1       class Recept {
2           var ime: String
3
4           // Koriscenjem kljucne reci 'self', naglasavamo da hocemo da
                pristupimo svojstvu klase
5           init(ime: String) {
6               self.ime = ime
7           }
8
9           // Metod koji ispisuje svojstvo 'ime'
10          func ispisiIme() {
11              print(ime)
12          }
13
14          // Metod koji menja svojstvo 'ime', parametrom 'ime'
15          func promeniIme(novo ime: String) {
16              self.ime = ime
17          }
18      }
```

```
19
20      var recept = Recept("Bolonjeze")
21      recept.ispisiIme()
22      // Ispisuje Bolonjeze
23
24      recept.promeniIme(novo: "Karbonara")
25      recept.ispisiIme()
26      // Ispisuje Karbonara
```

Listing 2.26: *Методе*

Као и у свим објектно оријентисаним језицима, и у *Swift*-у постоји наслеђивање класа. Класа која наследи другу класу, наслеђује сва њена својства и методе које нису дефинисане као приватне, и може их и мењати, односно преписати (енг. override). Свака класа која не наслађује ниједну другу назива се основна класа. Пример наслеђивања класе може се видети у делу 2.27 - *Наслеђивање класа*.

```
1       class Pravougaonik {
2           var sirina = 0
3           var duzina = 0
4
5           func izracunajPovrsinu -> Int {
6               return sirina * duzina
7           }
8       }
9
10      class Kvadrat : Pravougaonik {
11          override func izracunajPovrsinu -> Int {
12              return sirina * sirina
13          }
14      }
```

Listing 2.27: *Наслеђивање класа*

## Опционе промењиве и рад са њима

Опционе промењиве се користе у ситуацијама када није сигурно да ли ће промењива имати неку вредност и желећи да се избегне приступање таквој промењивој јер би дошло до одређених грешака у раду програма (када промењива нема вредност, њена подразумевана вредност је *nil* и са њом се мора

пажљиво руковати).

Када се дефинише опциона променљива експлицитно се наводи ког је типа након чега иде знак '?' и након тога може се, а не мора, извршити иницијализација. Уколико се променљива иницијалзијује без експлицитног навођења опционог типа, неопходно је да израз којим се иницијализује променљива буде опционог типа, у супротном променљивој не би био додељен опциони тип и она не би могла да се користи као опциона променљива. Пример дефинисања опционе променљиве уз иницијализацију и експлицитно навођење опционог типа, као и два начина иницијализације без навођења типа може се видети у делу 2.28 - *Дефинисање опционе променљиве.*

```
1    // Opciona promenljiva tipa 'Int?'
2    var opciona: Int? = 42
3    // Promenljiva tipa 'String'
4    var brojUOblikuStringa = "55"
5    // Opciona promenljiva tipa 'Int?'
6    var konvertovaniBroj = Int(brojUOblikuStringa)
```

Listing 2.28: *Дефинисање опционе променљиве*

У неким ситуацијама не може се радити са опционим променљивима, на пример када се прослеђују као параметри функције која очекује конкретну вредност, па опциону променљиву морамо одмотати и узети вредност која се налази у њој. Да при томе не дође до грешке тако што би било покушано одмотавање *nil* вредности, постоје два начина за безбедно одмотавање опционе променљиве и руковање са *nil* вредношћу.

Први начин је одмотавање опционе променљиве помоћу условног гранања (*if* или *guard*) приказан у примеру 2.29 - *Одмотавање опционе променљиве коришћењем услова*, а други начин задавањем подраз умеване вредности односно коришћењем оператора *if* сједињавања (енг. nil coalescing) приказан у примеру 2.30 - *Одмотавање опционе променљиве задавањем подразумеване вредности.*

```
1    func saberiDvaBroja(_ prvi: Int, _ drugi: Int) -> Int {
2        return prvi+drugi
3    }
4
5    var opcioniBroj: Int? = 42
6    var broj = 25
7
```

```
8     // saberiDvaBroja(opcioniBroj, broj) - greska, 'opcioniBroj' je
          tipa 'Int?' dok funkcija ocekuje parametar tipa 'Int'
9
10    // 1. nacin koriscenjem if-a
11    if let raspakovaniBroj = opcioniBroj {
12        saberiDvaBroja(raspakovaniBroj, broj) // 'raspakovaniBroj' je
              tipa 'Int'
13    }
14    else {
15        print("Prvi broj nema vrednost, ne moze se sabrati")
16    }
17
18    // 2. nacin koriscenjem if-a
19    if opcioniBroj != nil {
20        saberiDvaBroja(opcioniBroj!, broj)
21    }
22    else {
23        print("Prvi broj nema vrednost, ne moze se sabrati")
24    }
25
26    // 3. nacin koriscenjem guard-a
27    guard opcioniBroj != nil else {
28        print("Prvi broj nema vrednost, ne moze se sabrati")
29        return
30    }
31    saberiDvaBroja(opcioniBroj!, broj)
```

Listing 2.29: *Одмотавање опционе променљиве коришћењем услова*

```
1     func saberiDvaBroja(_ prvi: Int, _ drugi: Int) -> Int {
2         return prvi+drugi
3     }
4
5     var opcioniBroj: Int? = 42
6     var broj = 25
7
8     saberiDvaBroja(opcioniBroj ?? 5, broj)
```

Listing 2.30: *Одмотавање опционе променљиве задавањем подразумеване вредности*

## 2.4 Особине

Као што је већ неколико пута наведено, програмски језик *Swift* је креиран са намером да буде безбедан, модеран и ефикасан. Детаљан опис ових као и неких такође важних особина дат је у наставку поглавља.

### Модеран језик

*Swift* је настао као резултат најновијих истраживања програмских језика. Именовани параметри су изражајни, у једноставној синтакси, што код чини лаким за читање и разумевање. Као и у свим модерним језицима, употреба знака тачка-зарез на крају наредби није неопходна и по установљеној конвенцији се не пише. Претпостављање типова променљивих, чини код чистијим и отпорнијим на грешке. Меморијом се управља аутоматски, коришћењем аутоматског бројача референци (енг. Automatic Reference Counting, ARC).

Приликом дефинисања класе конвенција у *Swift*-у је да се за једноставније структуре користи кључна реч *Struct*, а не кључна реч *Class*. Надоградња типа се користи да наше типове одвојимо у смислене целине, на пример наслеђивање неке надкласе или имплементација интерфејса, што се може видети у примеру 2.31 - *Надоградња постојећег типа (класе, структуре)*, док истовремено можемо надоградити већ постојеће типове и имплементирати неке наше функције да не бисмо морали имплементирати исту логику више пута у коду приказано у примеру 2.32 - *Надоградња Swift класе*. Ограничење код екстензије је да не можемо додавати нова поља и променљиве унутар типа који надограђујемо.

```
1  struct Recept {
2      var ime: String
3      var vremePripreme = 30
4      var sastojci: [String] = []
5      var slika: UIImage
6
7      init(_ name: String) {
8          self.name = name
9      }
10 }
11
12 var recept = Recept("Omlet")
13
```

```
14 extension Recept {
15     func dodajSastojak(_ sastojak: String) {
16         self.sastojci.append(sastojak)
17     }
18 }
19 
20 self.recept.dodajSastojak("2 jaja")
```

Listing 2.31: *Надоградња постојећег типа (класе, структуре)*

```
1  extension UIImage {
2      func slikaRecepta(_ imeSlike: String) -> UIImage {
3          var image = UIImage(named: imeSlike)
4          var okvirSlike = image.view.frame
5          okvirSlike.width = 50
6          okvirSlike.height = 55
7          image.view.frame = okvirSlike
8          image.backgroundColor = .gray
9          return image
10     }
11 }
12 
13 self.recept.slika = UIImage.slikaRecepta(self.recept.ime)
```

Listing 2.32: *Надоградња Swift класе*

Многе могућности које нису поменуте у овом делу чине *Swift* тако моћним и модерним језиком, као што су: трансформације коришћењем затворења (енг. closures), моћни генерички типови који се лако користе, функције као грађани првог реда, брза итерација кроз распон или колекцију, *tuple* и вишеструке повратне вредности функција, енуми, функционално програмирање, уграђене функције за рад са изузецима и многе друге.

## Безбедан начин програмирања

У току развијања језика, уложени су огромни напори да би он био што безбеднији. Променљиве су увек иницијализоване, низови и целобројне променљиве се увек проверавају да не дође до прекорачења, меморијом се управља аутоматски и много других карактеристика. Још једна безбедоносна одлика је да *Swift* објекти подразумевано никада не могу бити *nil*. *Swift* компајлер ће спречити покушај да се направи или искористи *nil* објекат, избацивањем

грешке у време превођења програма. Међутим, постоје случајеви када је коришћење *nil* валидно. За те случајеве, користе се опционе променљиве[4]. Да би могле да се користе опционе променљиве, прво морају бити пажљиво одмотане, коришћењем оператора '?'.

У примеру 2.33 - *Коришћење опционе променљиве* може се видети употреба опционе променљиве, провера да ли је њена вредност различита од *nil* и уколико јесте њена вредност ће бити додељена новој променљивој која неће бити опционог типа јер не може имати вредност *nil*. Након тога нова променљива се може користи са сигурношћу да програм неће избацити грешку о насилном одмотавању променљиве чија је вредност *nil* (енг. unexpectedly found nil while unwrapping an Optional value).

```
1  func tableView(_ tableView: UITableView, cellForRowAt indexPath:
       IndexPath) -> UITableViewCell {
2
3      // Kreiranje nove celije
4      let celija = self.tableView.dequeueReusableCell(withIdentifier: "
           receptCelija") as? UITableViewCell
5
6      // Bezbedno odmotavanje vrednosti
7      guard let proverenaCelija = celija else {
8          return UITableViewCell()
9      }
10
11     // Provera broja elemanata niza
12     guard self.recepti.count > indexPath.row else {
13         return UITableViewCell()
14     }
15
16     // Promena teksta u polju celije
17     proverenaCelija.textLabel?.text = self.recepti[indexPath.row].ime
18
19     return proverenaCelija
20 }
```

Listing 2.33: *Коришћење опционе променљиве*

[4]Променљиве које уколико немају вредност, су једнаке *nil*

### Ефикасно извршавање

*Swift* је наследник програмских језика *C* і *Objective-C* и као такав од почетка је дизајниран да буде концизан и ефикасан. Коришћењем технологије *LLVM* компајлера, *Swift* код се трансформише у оптимизовани изворни код, који извлачи највише из модерног хардвера. Синтакса и стандардна библиотека су такође направљени тако да учине да се најочитији начин кодирања извршава најбоље, безобзира на ком је уређају покренут програм.

### Одличан језик за почетнике

*Swift* је дизајниран тако да може бити свачији први програмски језик. У циљу подучавања, *Apple* је направио бесплатан наставни план и програм који може свако користити [2]. Најбоља апликација за почетнике је *Swift Playgrounds* [4], апликација намењена уређајима *iPad*, развијена од стране *Apple*-а.

### Отвореног кода

#### *Playground*

Попут апликације *Swift Playground iPad*, однедавно постоји апликација *Playground* за оперативни систем *macOS* која је одлична за почетнике, али и за искусније програмере који желе да испробају део кода или се само мало забаве. Резултат извршавања ће бити одмах приказан; резултати извршавања могу бити приказани графички, унутар листе резултата или помоћу графа током времена.

#### *Package manager*

*Swift package manager* је више платформски алат за израду, покретање, тестирање и груписање ваших *Swift* библиотека и извршних датотека. Помоћу пакет менаџера могу се најлакше поделити библиотеке и изворни кодови. Конфигурација самог менаџера, као и сам *Swift* менаџер података, су такође писани у *Swift*-у, чинећи конфигурацију циљаних извршних датотека и управљање зависностима међу пактима веома једноставним.

### Компатибилан са *Objective-C*-ом

Цела апликација може бити написана у Swift-у, или се може користити за додавање нових функционалности у већ постојећи програм. Swift и Objective-C могу узајамно постојати у апликацији, и корисник без проблема може користити делове кода написаног у једном језику унутар другог и обратно, уз само мало додатног подешавања пројекта. [1]

## 2.5 Xcode

*Xcode* је интегрисано развојно окружење (ИРО) развијено од стране компаније *Apple*, намењен оперативном систему *macOS*. Софтвер је бесплатан и могуће је преузети га на *Mac App Store*-у[5].

### Основно

ИРО *Xcode* се користи за равој софтвера намењених оперативним системима *iOS*, *iPadOS*, *watchOS*, *tvOS* и *macOS*. Прва верзија *Xcode*-а, објављена је 2003. године, а последња стабилна верзија је *Xcode13*. *Xcode* укључује алат командне линије (енг. Command Line Tools, CLT) који омогућава *UNIX* стил развоја софтвера помоћу терминала.

Овај ИРО се састоји од неколико алата који помажу програмеру приликом развоја апликација за *Apple* платформе, од креирања апликације, преко тестирања и оптимизације, до прослеђивања на *App Store*-у. Најзначанији алати који су део *Xcode*-а су симулатор и инструменти.

### Симулатор

Симулатор се користи за тестирање апликације у току развоја, уколико не постоји могућност употребе физичког уређаја. Тестирање на симулатору у неким ситуацијама може бити и боље, јер пружа могућност тестирања апликације на више различитих уређаја (симулатора) одједном (на пример, различите генерације телефона *iPhone*, као и различите верзије оперативног система).

Као што је већ истакнуто, симулатор је део *Xcode*-а; инсталира се уз њега,

[5]Платформа која служи за дигиталну дистрибуцију апликација намењених оперативном систему *macOS*

а покреће се и понаша као обична апликација оперативног система *macOS* и омогућава симулацију свих уређаја са *Apple* платформе (*iPhone*, *iPad*, *Apple Watch*, *Apple TV*). Приликом тестирања могуће је и покретање више симулатора за различите платформе да би се тестирала њихова компатибилност, као на пример сарадња апликације на *iPhone*-у и *Apple Watch*-у. Поред овога, још неке погодности које пружа симулатор су: интеракција са апликацијама коришћењем миша и тастатуре, одстрањивање неисправности у апликацији, оптимизација графичког приказа.

### Инструменти

Инструменти су моћан алат, део *Xcode*-а, који служе за анализу перформанси апликације, као помоћ при њеном тестирању, да би се боље разумело понашање апликације и омогућила додатна оптимизација перформанси. Коришћење инструмената од почетка развијања апликације доприноси раном откривању појединих грешака и олакшава њихово решавање. Неке од функција које инструменти омогућавају су:

- Истраживање понашања апликације или процеса
- Испитивање карактеристика специфичних за уређаје као што су *Bluetooth* и *Wi-Fi*
- Профајлирање апликације у симулатору или на физичком уређају
- Анализа перформанси апликације
- Проблеми са меморијом
  - Цурење меморије
  - Напуштена меморија (енг. abandoned memory)
  - Зомби објекти (деалоцирани објекти који се још увек чувају)
- Оптимизовање апликације ради боље енергетске ефикасности

## 2.6 SwiftUI

Део развоја програмског језика *Swift* је усмерен ка поједностављивању процеса израде корисничког интерфејса и увођењу декларативне синтаксе у

језик. У том контексту настало је радно окружење *SwiftUI* које одликује могућност брзог креирања концизних и ефикасних решења.

## Уопштено

*SwiftUI* је радно окружење које служи за израду апликација са одличним графичким приказом, погодних за све *Apple* платформе, користећи моћ програмског језика *Swift* са што мање кода. Омогућава креирање бољих и разноврснијих апликација уз само један скуп алата и програмског интерфејса апликације (енг. Application Programming Interface, API).

## Основна структура

У склопу овог радног окружења добијамо велики број погледа (енг. views), контрола и распоредних структура (енг. layout structures) који олакшавају процес израде корисничког интерфејса апликације. Уз то садржи и алате за управљање током података од модела до погледа и контролера, које корисник види и може интераговати са њима преко додира, гестова и других типова улазних података у апликацији који се обрађују помоћу обрађивача догађаја.

Структура апликације се дефинише помоћу протокола *App* и попуњава се сценама које садрже погледе чији скуп чини кориснички интерфејс апликације. *SwiftUI* омогућава и креирање нових погледа, једини услов је да тај поглед имплементира протокол *View*. Нови поглед се може комбиновати са другим, корисничким или погледима радног окружења, као што су текстуална поља, слике и многи други да би се направили комплекснији погледи који ће бити погодни за све кориснике апликације.

## Карактеристике

Основна карактеристика која издваја *SwiftUI* од *UIKit*-а је другачија програмска парадигма, конкретно декларативна синтакса. Више о разликама ова два радна окружења биће описано у поглављу 2.6 - Разлика SwiftUI и UIKit.

Декларативна синтакса омогућава програмерима да што једноставније опишу понашање корисничког интерфејса. Код је много једноставнији за читање и разумевање, као и за писање, чиме је обезбеђена значајна уштеда

времена приликом писања новог кода и одржавања већ постојећег. Пример једноставног кода у *SwiftUI*-у приказан је у делу 2.34 - *Пример SwiftUI кода*. Модификатор *@State* биће објашњен у делу 2.6 - Стање и ток података.

```
1   // Ucitavanje SwiftUI radnog okruzenja
2   import SwiftUI
3
4   // Kreiranje strukture koja ce sadrzati glavni pogled
5   struct Content : View {
6
7       // Definisanje promenljive 'recepti'
8       @State var recepti = RecepModel.listaRecepata
9
10      // Definisanje tela pogleda
11      var body: some View {
12          // Izlistavanje svih recepata kroz tabelu
13          List(recepti.stavke, action: recepti.izabranaStavka) {
                recept in
14              // Prikaz slike
15              Image(recept.slika)
16              // Definisanje vertikalnog skupa elemenata
17              VStack(alignment: .leading) {
18                  // Prikaz teksta
19                  Text(recept.ime)
20                  // Prikaz teksta sive boje
21                  Text(recept.vremePripreme)
22                      .color(.gray)
23              }
24          }
25      }
26  }
```

Listing 2.34: *Пример SwiftUI кода*

## Стање и ток података

Декларативно програмирање омогућује да се за погледе вежу одговарајући модели података. Када год се неки од података промени, *SwiftUI* аутоматски поново учита све погледе за који су промењени подаци везани и прикаже их кориснику, тако да програмер не мора да брине и о томе. Ово се постиже променљивим стањима и везивањем, чиме се подаци везују за конкретне погледе.

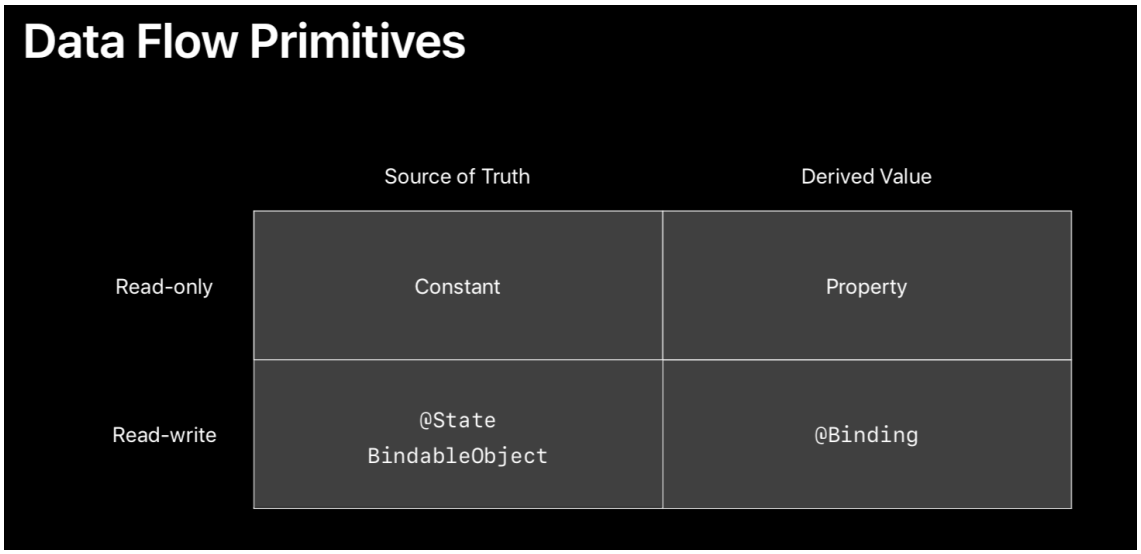
Тиме се остварује једини извор истине[6] (енг. single source of truth, SSOT) за све податке и олакшава одржавање тачности података у сваком тренутку.

У зависности од конкретне потребе у тренутној ситуацији, постоји више начина за остваривање јединог извора истине:

- *State* - Омогућава локално управљање стањем корисничког интерфејса, пример 2.35 - *Омотачи података - State*
- *BindableObject* - Користећи *ObservedObject* омотач својства, може се приступити спољашној референци на модел података који имплементира *ObservableObject* протокол. Уколико је променљива смештена у спољашње окружење, може јој се приступити користећи *EnvironmentObject* омотач својства. Инстанцирање посматрајућег (енг. observable) објекта директно у погледу, постиже се коришћењем *StateObject*
- *Binding* - Користи се за дељење референце на једини извор истине, пример 2.36 - *Омотачи података - Binding*
- *Environment* - Подаци сачувани у *Environment-y* се могу делити кроз целу апликацију, пример 2.37 - *Омотачи података - Environment*
- *PreferenceKey* - Прослеђивање података уз хирархију погледа, од детета ка родитељу
- *FetchRequest* - Управљање трајним подацима који се чувају унутар *Core Data*

Графички приказ модификатора може се видети на слици 2.2 - *Различити омотачи података.*

[6]Једини извор истине је начин структуирања информационих модела и шеме података тако да се сваки податак обрађује и мења на само једном месту




Слика 2.2: *Различити омотачи података*

```
1   struct Recept: View {
2       var recept: ReceptPodatak
3       @State private var daLiJeOmiljen: Bool = false
4
5       var body: some View {
6           VStack {
7               Text(recept.ime)
8               // 'OmiljenRecept' je pogled koji sadrzi zvezdicu koja
                    oznacava da li je recept medju omiljenima (puna
                   zvezdica - jeste, prazna - nije)
9               // Prosledjivanjem promenljive 'daLiJeOmiljen' uz
                   prefiks '$' omogucava se promena promenljive '
                   daLiJeOmiljen' u pogledu 'OmiljenRecept'
10              OmiljenRecept(daLiJe: $daLiJeOmiljen)
11          }
12      }
13  }
```

Listing 2.35: *Омотачи података - State*

```
1   struct Recept: View {
2   // Promenljiva 'isPlaying' je definisana u jednom pogledu, a moze
       se menjati u drugom
3   var recept: ReceptPodatak
4   @Binding var daLiJeOmiljen: Bool
```

```
5
6       var body: some View {
7           Button(action: {
8               // Akcija dugmeta koja menja promenljivu 'daLiJeOmiljen'
9               self.daLiJeOmiljen.toggle()
10          }) {
11              // Provera promenljive 'daLiJeOmiljen' i prikaz
                    odgovarajuce slike
12              Image(systemName: daLiJeOmiljen ? "star.fill" : "star.
                    empty")
13          }
14      }
15  }
```

Listing 2.36: *Омотачи података - Binding*

```
1       struct Kulinarstvo_widgetEntryView : View {
2       var entry: Provider.Entry
3
4       // Citamo podatke za 'widgetFamily' iz okruzenja aplikacije (eng.
            Environment) i smestamo ih u promenljivu 'widgetFamily'
5       @Environment(\.widgetFamily) var widgetFamily
6
7       @ViewBuilder
8       var body: some View {
9           // U zavisnosti od promenljive 'widgetFamily' prikazujemo
                odgovarajuci widget
10          switch widgetFamily {
11          case .systemSmall:
12              RecipeView(recipe: entry.recipe)
13                  .widgetURL(entry.recipe.url)
14          case .systemMedium:
15              RecipeMediumView(recipe: entry.recipe, ingredients: entry.
                    recipe.ingredients.count > 3 ? Array(entry.recipe.
                    ingredients.dropLast(entry.recipe.ingredients.count -
                    3)) : entry.recipe.ingredients)
16          default:
17              Text("")
18          }
19      }
20  }
```

Listing 2.37: *Омотачи података - Environment*

## Разлика SwiftUI и UIKit

*UIKit* и *SwiftUI* су радна окружења развијена од стране *Apple*-а, а која помажу приликом израде корисничког интерфејса апликације. Генерално, највећа разлика између ова два радна окружења је у начину размишљања, како доћи до решења и како то решење касније имплементирати. Ову разлику ће најбоље бити показана на једном конкретном примеру; Форма за пријављивање на одређени сајт, креирање вертикалног скупа елемената, хоризонтално и вертикално центрираних у том скупу, док се скуп састоји од два текстуална поља(корисничко име и лозинка) и дугмета(са акцијом провере података).

Са *UIKit*-ом мора се водити рачуна о свим ситним детаљима као што су: креирање вертикалног скупа елемената, његово додавање у главни поглед, креирање текстуалног поља, додавање текстуалног поља у скуп елемената, додавање аутоматског ограничења распореда како би се центрирало текстуално поље, понављање поступка за друго текстуално поље и поновно понављање поступка за дугме.

За разлику од тога, као што је напоменуто, *SwiftUI* се базира на декларативном начину програмирања, и овде је довољно да навести груписање два текстуална поља и дугмета у вертиклани скуп елемената и у ком погледу ће се приказати. Све ситне детаље ће радно окружење одрадити уместо нас онако како је то уобичајено дефинисано. Наравно не мора се придржавати свих детаља које окружење одради, већ се могу по потреби променити.

Креирање корисничког интерфејса у *UIKit*-у коришћењем само *Swift* кода је веома компликовано, и за веће пројекте готово немогуће, јер се мора водити рачуна о сваком детаљу. Најчешћи начин израде корисничког интерфејса је коришћењем *Storyboards-a* і *Interface Builder-a*, помоћу којих програмер креира кориснички интерфејс превлачењем, спуштањем и конфигурацијом елемената. У *SwiftUI*-у се кориснички интерфејс изграђује помоћу *Swift* кода. Једноставно се изјасни шта ће бити креирано и радно окружење то уради. Да би процес креирања био бржи и приступачнији, од верзије *Xcode*-а 11, која је изашла у исто време када је представљен *SwiftUI*, постоји могућност прегледа уживо сваког појединачног погледа који је креиран или скупа више погледа одједном. О овоме ће бити више речи у поглављу 2.6 - Xcode - преглед уживо.

Уколико се сагледају архитектуре образаца, може се приметити да се *UIKit*

првенствено базира на *MVC*[7] обрасцу, док за разлику од њега, *SwiftUI* користи *MVVM*[8] образац. За заинтересоване читаоце, постоји могућност комбиновања два радна окружења и коришћење *SwiftUI*-а унутар *UIKit* кода, или обратно, али ово се оставља читаоцима да сами истраже како се то може постићи.

## Xcode - преглед уживо

Са представљањем *SwiftUI*-а, *Apple* је представио и нову верзију њиховог ИРО-а, *Xcode11*, у коме је додато својство рада у новом радном окружењу као и могућност прегледа уживо сваког погледа. Предност оваквог начина писања кода је пре свега у могућности брзог прегледа измена и то без поновног обнављања (енг. rebuilding) апликације, поготово уколико се ради на додавању или измени погледа који се налази дубоко у навигацији апликације и за који је потребно више кликова и/или превлачења да би се до њега дошло.

Преглед уживо помаже да се и у *SwiftUI*-у користи метод превлачења и пуштања за креирање корисничког интерфејса, који се разликује од претходног који је коришћен унутар *Storyboard*-а, јер сваки елемент се превлачи у део где се пише код и када се испусти тај елемент постаје део кода. Да би се омогућило коришћење приказа уживо, инстанца жељеног погледа се смешта унутар тела структуре која имплементира протокол *PreviewProvider*, а која служи за живи приказ погледа или групе погледа. Пример употребе структуре која имплементира протокол *PreviewProvider* може се видети у делу 2.38 - *Xcode - преглед уживо*.

```
1
2     struct PlaceholderView : View {
3         var body : some View {
4             Kulinarstvo_widgetEntryView(entry: SimpleEntry(date: Date
                  (), configuration: ConfigurationIntent(), recipe:
                  RecipeModel.testData[0]))
5         }
6     }
```

[7]Архитектурни образац Модел-Поглед-Контролер (енг. Model-View-Controller) који се заснива на подели на три целине, модел - структура података, поглед - приказ података у корисничком окружењу, контролор - управљање моделом

[8]Архитектурни образац Модел-Поглед-Модел погледа (енг. Model-View-ViewModel) који се заснива на подели на три целине, модел - структура података, поглед - приказ података у корисничком окружењу, модел погледа - стање података у моделу

```
7
8       // Struktura u kojoj se konfigurise prikaz uzivo
9       struct Kulinarstvo_widget_Previews: PreviewProvider {
10          static var previews: some View {
11              // Grupisanje vise pogleda
12              Group {
13                  // Prikaz malog widget-a sa prvim elementom iz liste
14                  Kulinarstvo_widgetEntryView(entry: SimpleEntry(date:
                        Date(), configuration: ConfigurationIntent(),
                        recipe: RecipeModel.testData[0]))
15                      .previewContext(WidgetPreviewContext(family: .
                            systemSmall))
16
17                  // Prikaz srednjeg widget-a sa skrivenim sadrzajem
18                  PlaceholderView()
19                      .previewContext(WidgetPreviewContext(family: .
                            systemMedium))
20                      .redacted(reason: .placeholder)
21              }
22          }
23      }
```

Listing 2.38: *Xcode - преглед уживо*

Након сваке измене која се направи у коду који је везан за поглед(е) који се налази у прегледу уживо, *Xcode* ће изнова направити нову верзију и покренути је у прозору за преглед уживо. Као што је виђено у примеру кода изнад, преглед уживо не мора приказивати само један поглед, већ се могу груписати различити погледи и све бити приказати одједном. Предност оваквог приступа је могућност истовременог прегледа старог и новог изгледа погледа, више величина *widget*-а, истих погледа са светлом и тамном бојом позадине, погледа на различитим језицима...

За оне који желе да истраже више о овој теми, препорука је да одгледају два одлична клипа са *Apple*-ове конференције за програмере из 2019. и 2020. године респективно. Клипови су: *'Mastering Xcode Previews'* и *'Structure your app for SwiftUI previews'*.

# Глава 3

# Улога и развој Widget-а

*Widget*, као део мобилне апликације, се налази на почетном екрану уређаја (телефона или таблета) и кориснику приказује одабране важне информације из те апликације. За разлику од *widget*-а у оперативном систему *Android*, који су присутни више од десет година, *widgeti* на Apple платформама су уведени 2020. године, тако да је и сама технологија која подржава њихово креирање и даље у активном развоју.

## 3.1 Основно

*Widgeti* на уређајима са *Apple* платформом узимају један од кључних делова апликације за коју је развијени и приказују га крајњим корисницима тамо где ће га најлакше уочити, на *iPhone*-у и *iPad*-у се може налазити на почетном екрану или у делу *Today View*-а, док се на *Mac* уређајима налазе у центру за нотификације. Величина *widget*-а није флексибилна као на *Android* уређајима, па тако постоји могућност креирања малих (величина 2x2 места на почетном екрану *iPhone*-а), средњих (2x4), великих (4x4) и од верзије *iPadOS15* екстра великих (4x8), само за *iPad* уређаје, *widget*-а.

Скуп свих тренутно доступних *widget*-а на уређају налази се у галерији *widget*-а (енг. widget gallery), која помаже корисницима приликом одабира конкретне величине и типа *widget*-а (Једна апликација може испоручити више типова *widget*-а исте величине). Унутар галерије такође постоји опција за измену *widget*-а у којој корисници могу да контролишу и мењају своје *widget*-е и тиме их што више прилагоде себи, али само уколико је у току конструисања *widget*-а од стране програмера то омогућено. Више речи о овоме биће у делу

3.2 - Развој Widget-а.

На оперативним системима *iOS* и *iPadOS*, галерија има могућност додавања паметних гомила (енг. smart stack), које могу садржати до 10 различитих *widget*-а исте величине. Паметна гомила приказује само један од *widget*-а који се налазе у њој. Корисник може сам да мења који ће *widget* бити приказан једноставним померањем (енг. scrolling). Временом, паметна гомила може научити који *widget* корисник ставља на почетак гомиле у току дана (или недеље) и сама мењати примарне *widget*-е у одређеном тренутку (на пример, након гашења аларма прво се приказује *widget* са временском прогнозом, па најновије вести, стање у саобраћају...)

*Siri*[1] може и сама додати *widget*-е у паметну гомилу, уколико претпостави да постоји неки *widget* који би кориснику био користан. Након тога, корисник сам одлучује да ли жели да новододати *widget* остане у паметној гомили или не.

### WidgetKit

*WidgetKit* је радно окружење, које уз *widget API* из *SwiftUI*-а служи за израду *widget*-а, од његовог изгледа преко временског ажурирања па све до омогућавања конфигурације *widget*-а од стране крајњих корисника и управљања паметном гомилом приликом ротације *widget*-а од стране самог система. Још једна могућност коју ово радно окружење пружа је повезивање апликације и самог *widget*-а, што омогућава кориснику да отвори апликацију притиском на *widget* и аутоматски оде на одговарајући поглед из *widget*-а када жели да види детаљније податке. Пажња код ових ствари је да *widget* не би смео да служи само као пречица за покретање апликације, више о томе биће објашњено у делу 3.3 - Дизајн *Widget*-а.

## 3.2 Развој Widget-а

*Widget* је ништа друго до заправо само један *SwiftUI* поглед. *Widgeti* су тренутно једини део оперативних система *Apple*-а који у потпуности морају бити написани коришћењем радног оквира *SwiftUI*. *Apple* је отпочетка развоја *widget*-а имао на уму овакву идеју, због начина приказивања података,

[1]Интелигентни лични асистент на уређајима са *Apple* платформом

повременог ажурирања података и немогућности корисничке интеракције са самим *widget*-има (осим једноставног клика којим се отвара одређени део аплиакције).

## Додавање widget додатка апликацији

Шаблон за *widget* додатак креира основне ставке потребне за његову израду. Унутар овог додатка корисник креира све потребне *widget*-е за апликацију, независно од њиховог броја и величине. У посебним ситуацијама различити *widgeti* могу бити одвојени у посебним додатцима, ово се најчешће односи када један тип *widget*-а захтева одређене дозволе од стране корисника, док за други тип оне нису потребне (на пример, приступ тренутној локацији корисника).

Кораци за креирање *widget* додатка:

1. Отворити пројекат у *Xcode*-у и изабрати *File -> New -> Target*
2. Из групе *Application Extension*, изабрати *Widget Extension* и кликнути *Next*
3. Унети име додатка
4. Уколико ће *widget* подржавати конфигурацију од стране корисника, штиклирати поље *Include Configuration Intent*
5. Кликнути на *Finish*

## Додавање детаља конфигурације

Као што је већ напоменуто, шаблон *widget* додатка пружа иницијалну имплементацију *widget*-а која имплементира *Widget* протокол. Два могућа начина конфигурације *widget*-а су статичка (енг. StaticConfiguration) и конфигурација са сврхом (енг. IntentConfiguration). Статичка конфигурација се користи за *widget*-е који немају параметре који могу бити конфигурисани од стране корисника (на пример, системска апликација *Screen time* која води статистику о времену проведеном на одговарајућем уређају). Конфигурација са сврхом се користи за *widget*-е чији одређени параметри могу бити конфигурисани од стране корисника (на пример, системаска аплиакција за временску прогнозу где корисник може наместити одређени град за који жели да добија

податке). Ова конфигурација ће бити укључена и конфигурациони фајл ће бити додат уколико је корисник приликом додавања *widget* додатка, штиклирао поље *Include Configuration Intent*.

Да би програмер спровео почетну конфигурацију *widget*-а потребно је да проследи следеће параметре:

- Тип (енг. Kind), стринг који идентификује *widget*, требао би да казује шта *widget* представља
- Снабдевач (енг. Provider), објекат класе која имплементира протокол *TimelineProvider* и кроз временску линију коју производи одређује у ком тренутку ће *widget* бити поново изрендерован и нови подаци бити приказани. Више о овом протоколу и свеукупној причи о временој линији у делу 3.2 - Временска линија
- Затворење садржаја (енг. Content Closure), затворење које садржи поглед *SwiftUI* и које *WidgetKit* позива када дође време за поновно рендеровање садржаја *widget-a*
- Прилагођена сврха (енг. Custom Intent), фајл који дефинише параметре које корисник може мењати и прилагођавати себи, више о овоме у делу: 3.2 - *Intent*

## Временска линија

Снабдевач временске линије генерише временску линију која се састоји од уноса (енг. entries), а сваки унос садржи датум и време када је потребно ажурирати садржај *widget*-а. Када се датум и време из уноса подударе са реалним временом, *WidgetKit* позива затворење садржаја које потом приказује ажуриране податке.

Да би *widget* био приказан у *widget* галерији, *WidgetKit* захтева од снабдевача, преглед снимка (енг. Preview snapshot). Дохватање прегледа снимка се разрешава провером промењљиве *isPreview* којом се проверава да ли снабдевач прегледа снимка шаље тренутни снимак за приказ у галерији или за приказ *widget*-а на почетном екрану (или *Today* погледу, или центру за обавештења). Када је параметар *isPreview* тачан, *widget* се приказује у галерији. Уколико за приказ *widget*-а треба да буду приказани и одређени подаци, а подаци нису пристигли са серверске стране, постоје два решења. Можемо

приказати подразумеване, унапред одређене податке, или можемо користити податке које чувају место правим подацима (енг. placeholder).

Када пристигну подаци са сервера, снабдевач добија обавештење, сакупља реалне податке и приказује *widget* са њима. Након што корисник дода *widget* на почетни екран и буде приказан иницијални снимак изгледа *widget-a*, *WidgetKit* позива функцију *getTimeline* из провајдера, чиме захтева временску линију.

### *Intent*

*Widget*-і предтављају погледе који не интерагују са корисницима, односно не подржавају интерактивне елементе, као што су поглед *scroll* и дугме *switch*. Једна врста интеракције корисника са *widget*-ом постиже се омогућавањем конфигурације *widget*-а од стране корисника коришћењем конфигурације *Intent*, у којој се наводе сви параметри које корисник може да промени (и дозвољене вредности за те параметре).

Да би се додали параметри које корисник може да конфигурише, постоје предуслови који се морају испунити:

- Додавање дефиниције *intent*-а који дефинише конфигурабилне параметре
- Коришћење протокола *IntentTimelineProvider* уместо протокола *TimelineProvider* као провајдера временске линије, да би конфигурација параметара од стране корисника била сачувана у уносима временске линије
- Уколико параметри зависе од динамичких података потребно је имплементирати екстензију *intent*-а

### Везе унутар *widget*-а

Једини начин директне комуникације између корисника и *widget*-а остварена је везама (енг. links) унутар *widget*-а. Када корисник кликне на *widget* отвара се апликација којој тај *widget* припада, и може се конфигурисати који део апликације ће бити приказан кориснику у зависности од елемента унутар *widget*-а на који је кликнуо. Свим величинама *widget*-а може бити додат модификатор *widgetURL(_:)*, којим се одређује у који део апликације ће корисник бити одведен када кликне на *widget*.

За све величине *widget*-а, осим малих, могу се користити и везе које су додате једном елементу унутар *widget*-а и којим је одређено место у апликацији које ће бити отворено (на пример, један *widget* средње величине који садржи листу са 3 рецепата, сваки елемент листе има везу која води ка детаљној страни о рецепту који тај елемент представља). Иако *widget* користи везе унутар својих елемената, може користити и модигикатор *widgetURL(_:)* из претходног примера. Овај модификатор ће бити активиран уколико корисник кликне на елеменат у *widget*-у који нема дефинисану везу.

### Више *widget*-а у једном проширењу

Уколико желимо да користимо више различитих типова *widget*-а у једном проширењу, то можемо лако урадити уз само пар измена главног дела проширења, означеног атрибутом *@main*. Уместо протокола *Widget* главна структура мора имплементирати протокол *WidgetBundle*. Тело структуре сада имплементира протокол *Widget* и додата му је анотација *@WidgetBundleBuilder*. Приказ употребе више типова у једном проширењу може се видети у примеру 3.1 - *Више widget-а у једном проширењу*.

```
1    @main
2    // Umesto protokola 'Widget', koristimo 'WidgetBundle'
3    struct ReceptiWidgets: WidgetBundle {
4        // Definisemo atribut '@WidgetBundlerBuilder'
5        @WidgetBundleBuilder
6        // Definisemo telo, koje ovog puta implementira protokol '
             Widget'
7        var body: some Widget {
8            DetaljanPrikazReceptaWidget()
9            ListaRecepataWidget()
10           SpisakZaKupovinuWidget()
11       }
12   }
```

Listing 3.1: *Више widget-а у једном проширењу*

## 3.3 Дизајн *Widget*-а

Главна улога *widget*-а је приказивање садржаја који кориснику пружа корисне информације без покретања апликације. Самим тим подаци морају

бити тачни и релевантни за корисника, сам *widget* би требао бити конфигурабилан како би кориснику допустио одређену врсту слободе прилком коришћења *widget*-а и дизајниран тако да одговара апликацији којој припада.

## Фокус *Widget*-а

Подаци које приказује треба да буду минималистички, да одговарају величини *widget*-а (већа величина треба да повлачи и већу количину података) и да буду временски и кориснички релевантни. Први корак у дизајну *widget*-а је избор једног дела аплиакције који ће тај *widget* представљати.

Свака величина *widget*-а која је омогућена за додавање из галерије треба да садржи одређену количину информација која је пропорционална тој величини. Не сме се дозволити да више величина *widget*-а приказују исте податке, али истовремено приликом додавања нових података се мора водити рачуна о почетној идеји, односно делу апликације које тај *widget* треба да представља. Уколико не постоји довољна количина података за веће *widget*-е (на пример, за *widget* величине *large*), одређене величине могу бити и искључене из понуде кориснику.

*Widget* не би смео да служи само као пречица за покретање аплиакције. Корисници очекује од сваког *widget*-а да им покаже корисне информације, у супротном неће наићи на добар одзив и истовремено може бити штетно самој апликацији (мањи број корисника, лошија оцена у продавници).

## Ажурни подаци

Да би *widgeti* могли да пружају корисне и прецизне информације у скоро сваком тренутку, морају повремено бити ажурирани. *Widgeti* не подржавају ажурирање у реалном времену, а и сам систем може ограничити ажурирање *widget*-а у завиности од корисничког понашања и интеракције са њим, па се морамо потрудити да нађемо начин на који ће подаци у нашем *widget*-у увек бити релевантни.

Потребно је пронаћи оптимално време за ажурирање података у *widget*-у, узимајући у обзир колико се сами подаци често мењају и колико често корисницима може бити корисно да виде те податке. Уколико након публикације *widget*-а уочимо да су корисницима чешће или ређе потребни новији подаци, можемо променити време између два ажурирања и тиме побољшати кори-

сничко искуство. Још једна корисна ствар код информација које су временски зависне (на пример, тренутно стање неког индекса или акције на берзи), можемо у *widget*-у додати и поље које ће представљати датум и време када су подаци последњи пут ажурирани. Иако не можемо да приказујемо податке у реалном времену, за неке податке можемо искористити помоћ система за одређивање датума и времена, па тако уколико би имали *widget* који приказује време у које ће се огласити аларм, истовремено можемо имати и ажуран податак о томе колико је времена остало до оглашавања аларма.

## Конфигурабилност и интеракција

У већини случајева *widget* треба да омогући кориснику конфигурабилност како би пружио корисне информације (на пример, књига коју корисник тренутно чита и његов прогрес у апликацији *Apple Books*), док поједине апликације могу то да изоставе (на пример, најновије вести). Уколико је *widget* конфигурабилан, потребно је да подешавања буду што једноставнија и да се не траже превише информација од корисника. Кориснички интерфејс за измену *widget*-а је унапред одређен и исти за све, као што је показано у делу 3.2 - *Intent*.

Када корисник кликне на *widget* или део унутар њега, за веће величине, очекује да буде одведен на жељену страницу, да не мора да претражује и даље апликацију (на пример, када кликне на одређени рецепт у *widget*-у, желеће да му се отвори страница са детаљним приказом тог рецепта, а не почетна страна). Истовремено, треба се избећи превише веза на малом простору, где се лако може десити да због густине распореда, корисник кликне грешком на један елемент, а желео је да притисне сасвим други.

## Дизајн прилагођен свима

*Widgeti* треба да имају јарке боје како би се истицали на екрану, али истовремено и јасно видљив текст како би корисник могао да види све потребне информације „бацањем погледа” након откључавања или непосредно пре закључавања уређаја. *Widget* треба прилагодити апликацији коју представља (боје, фонт текста, јединствени елементи...), док истовремено не треба претерати са обележјима.

Количина информација коју ће бити приказана у *widget*-у мора бити оп-

тимална. Уколико се прикаже премало информација *widget* неће имати превелики значај за кориснике, док превише информација на мало простора отежава читање и разумевање података.

Једна од битнијих ствари која се не сме заборавити у данашње време је дизајн *widget*-а за обе врсте боја системске позадине (светле и тамне). Дизајн оба *widget*-а се не сме разликовати од системске боје позадине јер ниједан корисник не жели видети таман текст на светлој боји позадине уколико је изабрао тамну системску боју позадине. Приликом израде обе врсте дизјна може помоћи *Xcode preview* који омогућава истовремено сагледавање оба дизајна, упоређивање и исправљање евентуалних недостатака.

*Apple* саветује да се никад не користи фонт текста мањи од 11 поена[2]. Коришћење мањег фонта би корисницима знатно отежало употребу *widget*-а. Поред овога, увек се требају користити званични елементи за приказ текста, како би се омогућила скалабилност као и системско читање текста.

Пажњу треба обратити на дизајн прегледа *widget*-а унутар галерије, за све типове и величине који *widget* подржава, као и приказ чувара места уместо реалних података уколико они нису пристигли на време са сервера и непостоје подразумевани подаци. Уколико се исти елементи налазе у апликацији и истовремено на *widget*-у потребно је да имају исту функционалност јер би у супротном корисници били збуњени.

Потребно је искористити могућност приказа описа *widget*-а у галерији и саставити кратак и јасан опис функционалности *widget*-а. Груписање свих величина једног типа *widget*-а са јединственим описом је погодно корисницима апликације, пре свега због једноставности разумевања коришћења *widget*-а.

Скалабилност елемената унутар *widget*-а је веома важна јер систем аутоматски прилагођава величину *widget*-а величини екрана уређаја, зато је потребно обратити пажњу приликом додавања елемената. Најбоље је користити проверене елементе који пружају флексибилност, у овом случају било који основни *SwiftUI* елемент или њихова комбинација.

[2] *Apple*-ов израз за „број који треба уписати у поље", универзална мера у дизајну на *Apple* платформама

# Глава 4

# Опис апликације

# Глава 5

# Закључак

# Библиографија


[1] Apple Inc. Apple Developer. on-line at: https://developer.apple.com/swift/.

[2] Apple Inc. Swift Education. on-line at: https://www.apple.com/education/k12/teaching-code/.

[3] Apple Inc. Swift on GitHub. on-line at: https://github.com/apple/swift.

[4] Apple Inc. Swift Playground. on-line at: https://www.apple.com/swift/playgrounds/.

[5] Apple Inc. SwiftUI. on-line at: https://developer.apple.com/xcode/swiftui/.

[6] Apple Inc. The swift programming language (swift 5.5), 2014.

[7] Apple Inc. Swift.org, 2021. on-line at: https://www.swift.org/.

[8] StackOverflow. Stack overflow. on-line at: https://stackoverflow.com/.