

УНИВЕРЗИТЕТ У БЕОГРАДУ  
МАТЕМАТИЧКИ ФАКУЛТЕТ



Марко Вельковић

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 Марич, 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 виџета у програмском језику Swift

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

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

# Садржај

<b>1 Увод</b>	<b>1</b>
<b>2 Програмски језик Swift</b>	<b>2</b>
2.1 Настанак и развој . . . . .	2
2.2 Основна намена и карактеристике . . . . .	3
2.3 Концепти . . . . .	3
2.4 Особине . . . . .	27
2.5 Xcode . . . . .	35
2.6 SwiftUI . . . . .	38
<b>3 Улога и развој Widget-a</b>	<b>47</b>
3.1 Основно . . . . .	47
3.2 Развој Widget-a . . . . .	48
3.3 Дизајн Widget-a . . . . .	55
<b>4 Опис апликације</b>	<b>59</b>
<b>5 Закључак</b>	<b>60</b>
<b>Библиографија</b>	<b>61</b>

# **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)<sup>1</sup> у јулу 2010. године. У јуну 2014. године објављена је прва апликација комплетно написана у *Swift*-у, назvana „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, дигагер и менаџер

---

<sup>1</sup>Софтверски инжењер, најпознатији по развоју *LLVM* технологија, *Clang* компајлера и *Swift* програмског језика

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

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* лоџо

*Swift* је моћан и интуитиван језик за програмирање апликација намењених *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 концептне преузете 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 званичном 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 се декларишу коришћењем кључне речи *let*, након чега следи име константе и затим њена иницијализација. Декларисање променљиве се постиже употребом кључне речи *var*. Када се декларише променљива може се одмах и иницијализовати или јој може бити додељен тип коришћењем анотације. Конкретна примена се може видети у примеру 2.1 - *Декларисања променљивих и константи*.

```
1 // Deklarisanje konstantne
2 let cenaJela = 200
3
4 // Deklarisanje promenljive uz inicijalizaciju
5 var raspolozivoNovca = 1000
6
7 // Deklarisanje promenljive koriscenjem anotacije, bez inicijalizacije
8 var brojPorcija: Int
```

Listing 2.1: *Декларисања променљивих и константи*

Целобројне променљиве могу бити написане у облику децималних, бинарних, окталних 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

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

се видети два начина креирања *Tuple* група и два начина приступања члановима *Tuple-a* (неименованим и именованим члановима).

```
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:
10    100)
11
11 // Pristupanje imenovanim clanovima tuple-a
12 let urednijaNamirnica = urednijaNamirnicaSaCenom.namirnica
13 let urednijaCena = urednijaNamirnicaSaCenom.cena
```

Listing 2.3: *Tuple*

*Assertions* су провере унутар кода која се дешавају у време извршавања програма. Обично се користе за проверу критичног дела кода и уколико она задовољава услов (вредност израза у *assert*-у је *true*) програм наставља своје извршавање. У супротном, извршавање апликације ће бити прекинуто и дибагер ће означити место у коду у коме је дошло до прекида програма. Пример примене биће приказан у коду 2.4 - *Assertions*.

```
1 var cenaPrvogProizvoda = 100
2 var cenaDrugogProizvoda = -100
3 assert(cenaPrvogProizvoda >= 0, "Cena proizvoda ne moze biti negativna
4   ") //true
5 assert(cenaDrugogProizvoda >= 0, "Cena proizvoda ne moze biti
6   negativna") //false
```

Listing 2.4: *Assertions*

## Оператори

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

- Унарни

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

- 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 2.5 - *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 granaanja  
8 let visinaCelije: Int  
9 if celijaImaSliku {  
10     visinaCelije = 100  
11 }  
12 else {  
13     visinaCelije = 50  
14 }
```

Listing 2.5: *Тернарни оператор*

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

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 2.6 - *nil-сједињавање*.

```
1 var a: Int?  
2 let trenutnoVremeUMilisekundama = Int.random(in: 1...100)  
3 if trenutnoVremeUMilisekundama % 2 == 0 {  
4     a = 10  
5 }  
6  
7 let b = a ?? 5  
8 // Raspisivanje koriscenjem ternarnog operatora:  
9 let c = (a != nil ? a! : 5)
```

Listing 2.6: *nil-сједињавање*

- Оператори распона

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

## Каректори и стрингови

Стринг је низ карактера, као што је „Здраво, светe”. У *Swift*-у се стрингови представљају помоћу класе *String*, која омогућава брз, ефикасан и *Unicode*-компабилан начин рада са текстом. Креирање и операције са стринговима (конкатенација, рад са карактерима и интерполација) биће приказане кроз пример 2.7 - *Операције нај са стринговима*.

```
1 var prazanString = ""  
2 var drugiPrazanString = String()
```

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

```
3  var trećiPrazanString: 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 2.7: Операције најсупротивнијима

## 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 2.8 - *If* наредба контроле тока.

```
1  var recept = Recept("Cezar salata")
2  recept.sastojci = ["Zelena salata", "Pilece grudi", "Slanina", "
3      Paradajz", "Hleb", "Cezar premaz"]
4  var brojSastojaka = recept.sastojci.count
5
6  if brojSastojaka < 6 {
7      print("Nisu svi sastojci nabavljeni")
8  }
9  else if brojSastojaka > 6 {
10      print("Broj sastojaka ne odgovara originalu, ali samo napred
11          eksperimentisi")
```

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

```
10    }
11    else {
12        print("Broj sastojaka je odgovarajuci")
13    }
```

Listing 2.8: 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). Уколико случајевима *switch* наредбе нису обухаћени све могуће вредност променљиве, на крају се мора навести *default* наредба која ће се извршити уколико се ниједан од случајева не поклапа са вредношћу променљиве. Наведена правила приказана су у примеру 2.9 - *Switch* наредба контроле тока.

```
1 enum Zacin {
2     case vegeta, kari, kurkuma, origano, biber
3 }
4 var mojiZacin: Zacin = .vegeta
5
6 switch mojiZacin {
7     case .vegeta:
8         print("Vegeta")
9     case .biber:
10        print("Nije vegeta, nego biber")
11    default:
12        print("Nije ni vegeta ni biber")
13 }
```

Listing 2.9: 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 константа и њену вредност није 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. Још једна могућност *for-in*

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

петље је пролаз кроз задати интервал бројева. Описани начине употребе петље могу се видети у примеру 2.10 - *For-in наредба концроле 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 // For-in naredba nad intervalom
7 for i in 0..
```

Listing 2.10: *For-in наредба концроле тока*

Постоје два типа наредбе понављања *while*. Наредба *while* прво проверава да ли је задати услов испуњен и онда извршава једну итерацију тела петље и наредба *repeat-while* која прво извршава једну итерацију тела наредбе некон чега проверава услов и уколико је он задовољен наставља са следећом итерацијом. Употреба је приказана у примеру 2.11 - *While наредба концроле тока*.

```
1 let nasumicniBrojevi = [3, 12, 5, 18, 11, 99]
2 var i = 0
3
```

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

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

Listing 2.11: *While* наредба контроле тока

Поред наведених основних наредба контроле тока, постоје додатне помоћне наредбе које се користе заједно са основним наредбама и тиме њихову употреби чине лакшом. Наредба *continue* се користи за прескакање једне итерације унутар петље. *Break* је наредба која прекида извршавање наредбе унутар које се налази, а може се користити унутар *switch* случајева и унутар тела петље. Пример употребе ових наредби приказан је у делу 2.12 - *Догаџи наредба контроле тока*.

```
1 let nizBrojeva = [1, 2, 3, 4, 5, 6, 7, 8, 9]
2 print("Parni brojevi iz niza manji od 7:")
3 for broj in nizBrojeva {
4     if broj % 2 != 0 {
5         continue
6     }
7     if broj > 7 {
8         break
9     }
10    print(broj)
11 }
```

Listing 2.12: *Догаџи* наредба контроле тока

## Колекције

*Swift* дефинише три примарна типа колекција: низове, скупове и речнике. Сва три типа су дефинисана као генеричке<sup>2</sup> колекције. Уколико се дефинисана колекција додели променљивој, она се може мењати (додавање, брисање

<sup>2</sup>Генерички код омогућава писање флексибилних и поновно искористивих функција и типова; помоћу њих се избегава непотребно дуплирање кода

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

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 2.13 - *Pag 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 inicialna 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() {
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 2.13: *Pag 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<sup>3</sup> (енг. hashable). Рад са скуповима приказан је у примеру 2.14 - *Pag са скуповима*.

```
1 // Kreiranje skupa sa elementima tipa 'String'
2 var sastojci = Set<String>()
3 // Dodavanje novog elementa
4 sastojci.insert("Mlevena kafa")
```

<sup>3</sup>Тип који се може кодирати мора имати дефинисану функцију за одређивање hash вредности за сваку инстанцу, два елемента могу имати исту hash вредност ако су једнаки

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

```
5 // Provera broja elemenata skupa
6 if sastojci.count == 1 {
7     sastojci.insert("Obicna voda")
8 }
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
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 2.14: Pag sa 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 примеру 2.15 - *Pag sa 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: "
6     Zahtev je prihvacen"]
7
8 // Dodavanje novog elementa ukoliko ne postoji, odnosno promena
9     postojeceg
10 tipoviHTTPStatusa[404] = "Stranica nije pronađena"
11
12 // Brisanje elementa iz recnika
13 if let izbrisanaVrednost = tipoviHTTPStatusa.removeValue(forKey:
14     201) {
15     print("Vrednost izbrisana iz recnika: \(izbrisanaVrednost)")
16 }
17
18 // Razlicite vrste iteracija kroz recnik
19
20 for kod in tipoviHTTPStatusa.keys {
21     // TODO: iteriraj kroz kodove
22 }
23
24 for status in tipoviHTTPStatusa.values {
25     // TODO: iteriraj kroz statuse
26 }
27
28 for (kod, status) in tipoviHTTPStatusa {
29     // TODO: iteriraj kroz elemente
30 }
```

Listing 2.15: Pag sa речницима

### Функције и затворења

Функције 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 тип повратне вредности (уколико није дефинисан, подразумевани тип повратне вредности је *Void*) и пареметре (именоване или неименоване). Оп-

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

шти потпис дефиниције функције је:

`func име_функције(параметри_функције) { -> тип_повратне_вредности}`

док се параметри могу представити на следећи начин:

{лабела\_параметра} име\_параметра: тип\_параметра {= подразумевана\_вредност}

Као што је већ наведено, параметри функције могу бити именовани и неименовани. Приликом позивања функције са именованим параметром потребно је навести лабелу тог параметра уз конкретну вредност. Приликом дефинисања функције лабела параметра се наводи пре имена параметра, или се наводи карактер '`_`' за неименоване параметре. Уколико лабела параметра није експлицитно наведена сматраће се да име параметра представља и лабелу. Пример дефинисања и позивања функције са именованим параметром и без повратног типа представљен је у коду 2.16 - *Дефинисање и позивање функције са параметром*, док пример 2.17 - *Дефинисање и позивање функције са повратном вредношћу* показује дефинисање и позивање функције са неименованим параметром и повратним типом `Int`.

```
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
10 ispisiSastojke(sastojci: sastojci)
```

Listing 2.16: *Дефинисање и позивање функције са параметром*

```
1 func izracunajCenu(_ proizvodi[String: Int]) -> Int {
2     int ukupno = 0
3     for (proizvod, cena) in proizvodi {
4         ukupno += cena
5     }
6     return ukupno
7 }
8
9 let proizvodi = ["Jaja": 10, "Sira": 200]
10 let ukupnaCena = izracunajCenu(proizvodi)
```

Listing 2.17: *Дефинисање и позивање функције са повратном вредношћу*

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

Поред лабеле уз параметар може стојати и подразумевана вредност параметра, која ће представљати вредност параметра приликом извршавања тела функције, уколико приликом позива функције наведени параметар није прописан. Пример је приказан у коду 2.18 - *Дефинисање и позивање функције са параметрима са подразумеваним вредностима.*

```
1 func ispisIsastojkeSaDvaparametra(sastojci: [String], ispisati
2     ispisatiCeloIme: Bool = true) {
3     for sastojak in sastojci {
4         if ispisatiCeloIme {
5             print(sastojak)
6         }
7         else {
8             print(sastojak.prefix(3))
9         }
10    }
11    let sastojci = ["Jaja", "Sira"]
12    ispisIsastojkeSaDvaparametra(sastojci: sastojci, ispisati: false)
13    ispisIsastojkeSaDvaparametra(sastojci: sastojci)
14 // Parametar 'ispisati' ce imati vrednost 'true'
```

Listing 2.18: *Дефинисање и позивање функције са параметрима са подразумеваним вредностима*

Као и други објектно оријентисани програмски језици и *Swift* пружа могућност дефинисања генеричких функција<sup>4</sup> које се могу користити над различитим конкретним типовима. Поред овога функције у *Swift*-у омогућавају дефинисање функција са две или више повратне вредности. Конкретан пример генеричке функције са две повратне вредности приказан је у коду 2.19 - *Дефинисање и позивање функције са више повратних вредности.*

```
1 // Definisanje generickie funkcije sa dvema povratnim vrednostima
2 func minMax<T>(niz: [T]) -> (min: T, max: T)? {
3     guard !niz.isEmpty else {
4         return nil
5     }
6     let minimum = niz.min()
7     let maksimum = niz.max()
```

<sup>4</sup>Генеричка функција је функција са генеричким параметрима или генеричким повратним вредностима. Генерички тип може представљати више различитих типова одједном који задовољавају одређене услове дефинисане од стране програмера

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

```
8     return (minimum, maksimum)
9 }
10
11
12 let nizBrojeva = [5, 12, -4, 19, -99]
13 let minimumIMaksimum = minMax(niz: nizBrojeva)
14 let minimum = minimumIMaksimum?.min
15 let maksimum = minimumIMaksimum?.max
```

Listing 2.19: Дефинисање и позивање функције са више повратних вредносћи

Прослеђени параметри унутар функција су константе и њихова вредност се не може мењати у телу функције. Да би се омогућило заобилажење овог правила, *Swift* је увео кључну реч '*inout*' која се наводи приликом дефинисања функције, а пре типа сваког од параметра за који ће бити омогућена промена вредности у телу функције. Још једна промена коју је потребно применити је употреба карактера '&' приликом прослеђивања променљиве у позиву функције. Употреба овако прослеђених параметара приказана је на примеру генеричке функције која замењује вредности два прослеђена параметра 2.20 - *Дефинисање и позивање функције са променљивим параметрима*.

```
1 func zameniDvaParametra<T>(prvi: inout T, drugi: inout T) {
2     var pomocna = prvi
3     prvi = drugi
4     drugi = pomocna
5 }
6
7 // Prosledjeni parametri moraju biti promenljive
8 var prviString = "Ja sam prvi"
9 var drugiString = "Ja sam drugi"
10
11 print(prviString + ", " + drugiString)
12 // Ja sam prvi, Ja sam drugi
13
14 zameniDvaParametra(prvi: &prviString, drugi: &drugiString)
15 print(prviString + ", " + drugiString)
16 // 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.

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

Изрази затворења представљају начин за писање затворења у једној линији (енг. inline), притом пружајући неколико синтаксних оптимизација у виду кратке форме, разумљивости и изражајности. Синтакса израза затворења приказана је у примеру 2.21 - *Синтакса израза затворења*. На примеру 2.22 - *Израз затворења за сортирање* показана је Swift метода 'sorted' и како се једно затворење може написати на неколико начина, од целе функције па све до само једног карактера.

```
1  (parametri) -> tip_povratne_vrednosti in  
2      naredbe
```

Listing 2.21: Синтакса израза затворења

```
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  // Prosledjivanjem funkcije  
7  var sortiraniBrojevi = nasumicniBrojevi.sorted(by: sortirajBrojeve  
8      )  
9  
9  // Koriscenjem zatvorenja  
10 sortiraniBrojevi = nasumicniBrojevi.sorted(by: { (broj1: Int,  
11     broj2: Int) -> Bool in  
12         return broj1 < broj2  
13     })  
13  
14 // Bez eksplicitnog navodjenja tipa parametra  
15 sortiraniBrojevi = nasumicniBrojevi.sorted(by: {broj1, broj2 in  
16     return broj1 < broj2  
17     })  
18  
19 // Kada u zatvorenju postoji samo jedna naredba, nije potrebno  
20     navodjenje ključne reci 'return', povratna vrednost bice  
21     vrednost izvršenja te naredbe  
22 sortiraniBrojevi = nasumicniBrojevi.sorted(by: {broj1, broj2 in  
23     broj1 < broj2  
24     })  
24  
25 // Swift omogucava i kratka imena parametara, za pruzanje  
26     izrazajnije sintakse  
27 sortiraniBrojevi = nasumicniBrojevi.sorted(by: {
```

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

```
26     $0 < $1
27 }
28
29 // Kada koristimo tipove za koje je vec definisano ponasanje
   prilikom poredjenja, mozemo proslediti samo kako zelimo da
   sortiramo clanove niza
30 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 сигурност да ће се наредбе у затворењу извршити након што се заврши извршавање функције. Оваква врста затворења назива се затворење трага (енг. trailing closures), пример у коду 2.23 - Затворење трага.

```
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 трага

## Класе и структуре

Класе и структуре су конструкције опште намене које имају 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

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

инстанцу класе као и структуре користи назив инстанца уместо уобичајеног, објекат.

У поређењу класа и структура, ствари које се могу радити са обе: дефинисање својства, 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 променљиво

Дефинисање класе и структуре, инстанцирање као и прступање 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 glavniIzgled = GlavniIzgled()
19
20 // Pristupanje clanovima instance
21 let okvirGlavnogIzgleda = glavniIzgled.okvir
22 let sirinaOkviraPozadine = okvir.sirina
23
```

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

```
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 property). 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 struct 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
14 // Pre izvršenja f-je 'print', instancira se klasa '
     UcitavanjeFajla'
15 print(menadzer.ucitavanje.imeFajla)
```

Listing 2.25: Лења својства

За разлику од својстава и лењих својстава који у себи чувају одређене вредности, рачунајућа својства (енг. computed properties) не чувају вредности, већ садрже блок дохватача (енг. getter) (који је обавезан) и опционо блок постављача (енг. setter) којима се дохватат и постављају вредности других променљивих. Приликом дефинисања рачунајућег својства уколико својство садржи и дохватач и постављач потребно је експлицитно навести блокове за

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

оба параметра *get* и *set*, док у случају да садржи само дохватач може се дефинисати само један блок за који ће компајлер знати да је блок дохватања. У примеру 2.26 - *Рачунајућа својсћва* приказано је рачунајуће својство '*centar*' које садржи и дохватач и постављач, као и рачунајуће својство '*povrsina*' које има само блок дохватача.

```
1 struct Tacka {
2     var x = 0.0
3     var y = 0.0
4 }
5 struct Dimenzije {
6     var stranica = 0.0
7 }
8 struct Kvadrat {
9     var pocetak = Tacka()
10    var dimenzije = Dimenzije()
11    var centar: Tacka {
12        get {
13            let centarX = pocetak.x + dimenzije.stranica/2
14            let centarY = pocetak.y + dimenzije.stranica/2
15            return Tacka(x: centarX, y: centarY)
16        }
17        set(noviCentar) {
18            pocetak.x = noviCentar.x - dimenzije.stranica/2
19            pocetak.y = noviCentar.y - dimenzije.stranica/2
20        }
21    }
22    var povrsina: Double {
23        return pow(dimenzija.stranica, 2)
24    }
25 }
26 var kvadrat = Kvadrat(pocetak: Tacka(), dimenzije: Dimenzije(
27     stranica: 10.0))
28 let inicialniCentar = kvadrat.centar // Tacka(5.0, 5.0)
29 kvadrat.centar = Tacka(x: 15.0, y: 15.0)
30 print("Novi pocetak: \(kvadrat.pocetak.x) \(kvadrat.pocetak.y)")
31 // Novi pocetak: 10.0 10.0
32 print("Povrsina kvadrata: \(kvadrat.povrsina)")
33 // Povrsina kvadrata: 100.0
```

Listing 2.26: *Рачунајућа својсћва*

Посматрачи својства посматрају и реагују на промену вредности својства

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

унутар кога су имплементирани. Посматрачи реагују увек када је нова вредност додељена својству, чак иако је нова вредност једнака старој. Могу се користити унутар кориснички дефинисаних својстава, наслеђених својстава и наслеђених рачунајућих својстава. Посматрачи својства који се могу дефинисати унутар својства су *willSet* - које се позива пре постављања нове вредности и *didSet* које се позива након постављања нове вредности својства. Функцији *willSet* се аутоматски прослеђује параметар у којој је смештена нова вредност својства и уколико програмер не наведе експлицитно име параметра, оно ће бити *newValue*. Исто важи и за функцију *didSet*, којој се прослеђује параметар старије вредности својства са називом *oldValue*. Пример употребе посматрача приказан је у делу 2.27 - *Посматрачи својстава*.

```
1 struct Namirnica {  
2     var ime: String  
3     var cena: Double {  
4         willSet {  
5             print("Nova cena: \(newValue)")  
6         }  
7         didSet(staraCena) {  
8             if cena < staraCena {  
9                 print("Popust na namirnici: \(ime)")  
10            }  
11        }  
12    }  
13 }  
14 var mleko = Namirnica(ime: "Mleko", cena: 105.0)  
15 mleko.cena = 98  
16 // Nova cena: 98  
17 // Popust na namirnici: Mleko
```

Listing 2.27: *Посматрачи својстава*

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. Дефинисање метода класе и позивање тих метода над конкретном инстанцом класе приказано је у примеру 2.28 - *Методе*.

```
1 class Recept {  
2     var ime: String  
3 }
```

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

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

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 наслеђује ниједну другу назива се основна класа. Пример наслеђивања класе може се видети у делу 2.29 - *Наслеђивање класа*.

```
1 class Pravougaonik {
2     var sirina = 0
3     var duzina = 0
4
5     func izracunajPovrsinu -> Int {
6         return sirina * duzina
7     }
8 }
```

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

```
10 class Kvadrat : Pravougaonik {  
11     override func izracunajPovrsinu -> Int {  
12         return sirina * sirina  
13     }  
14 }
```

Listing 2.29: Наслеђивање класа

## Опционе променљиве и рад са њима

Опционе променљиве се користе у ситуацијама када није сигурно да ли ће променљива имати неку вредност и желећи да се избегне приступање таквој променљивој јер би дошло до одређених грешака у раду програма (када променљива 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 опциона 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 2.30 - *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 2.30: 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

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

променљиве и руковање са *nil* вредношћу.

Први начин је одмотавање опционе променљиве помоћу условног гранања (*if* или *guard*) приказан у примеру 2.31 - *Одмотавање опцијоне променљиве коришћењем услова*, а други начин задавањем подраз умеване вредности односно коришћењем оператора *if* сједињавања (енг. nil coalescing) приказан у примеру 2.32 - *Одмотавање опцијоне 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
9 // tipa 'Int?' dok funkcija ocekuje parametar tipa 'Int'
10
11 // 1. nacin koriscenjem if-a
12 if let raspakovaniBroj = opcioniBroj {
13     saberiDvaBroja(raspakovaniBroj, broj) // 'raspakovaniBroj' je
14     // tipa 'Int'
15 } else {
16     print("Prvi broj nema vrednost, ne moze se sabrati")
17 }
18
19 // 2. nacin koriscenjem if-a
20 if opcioniBroj != nil {
21     saberiDvaBroja(opcioniBroj!, broj)
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 2.31: Одмошавање оиционе 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 2.32: Одмошавање оиционе 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).

ARC се користи за праћење и управљање меморије коју апликација користи, тако што аутоматски ослобађа меморију коју је користила инстанца која више није потребна. Бројач референци води рачуна само о инстанцама класе, док инстанце структуре и набрајања игнорише јер оне представљају тип вредности, а не референтни тип. Сваки пут када се креира нова инстанца класе ARC одвоји део меморије потребан за смештање информација о инстанци. За сваку инстанцу ARC води рачуна о броју својстава, константи и променљивих

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

које реферишу на посматрану инстанцу. *ARC* ће деалоцирати део меморије у којој је смештена инстанца тек када све јаке (енг. strong) референце ка њој нестану. Када се инстанца класе додели својству, константи или променљивој уколико се екплицитно не наведе другачије, аутоматски ће бити креирана јака референца ка инстанци.

Ретка ситуација где програмер мора водити рачуна о управљању меморијом је избегавање креирања две инстанце неких класа које међусобно држе јаке референце једна ка другој јер у овој ситуацији долази до формирања циклуса јаких референци (енг. strong reference cycle) што онемогућује *ARC* да деалоцира меморију у којој се налазе ове две инстанце (увек ће једна ка другој имати јаку референцу) и долази до цурења меморије. Циклус јаких референци се може спречити употребом *weak* или *unowned* референци које *ARC* не узима у обзир. Пример употребе слабе (енг. weak) референце приказан је у коду 2.33 - *Слаба референца*.

```
1 protocol NoviReceptDelegate : AnyObject {
2     func receptJeKreiran(_ viewController: NoviReceptController, -
3                           receipt: Recept)
4 }
5
6 class NoviReceptController() : UIViewController {
7     weak delegate: NoviReceptDelegate?
8     ...
9     func sacuvajReceipt(_ receipt: Recept) {
10         ...
11         self.delegate.receptJeKreiran(self, receipt)
12     }
13 }
14 class ReceptController : UIViewController {
15     var novRecept = NoviReceptController()
16     override viewDidLoad() {
17         ...
18         self.novRecept.delegate = self
19     }
20 }
21 extension ReceptController : NoviReceptDelegate {
22     func receptJeKreiran(_ viewController: NoviReceptController, -
23                           receipt: Recept) {
```

24 | }

Listing 2.33: Слаба референца

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 2.34 - *Надоградња постојећег шија (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 2.35 - *Надоградња Swift класе*. Ограниччење 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 2.34: Надоградња постојећег шија (класе, структуре)

```
1 extension UIImage {
2     func slikaRecepta(_ imeSlike: String) -> UIImage {
3         var image = UIImage(named: imeSlike)
4         var okvirSlike = image.view.frame
```

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

```
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 2.35: *Нагођања 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 више-струке повратне вредности функција, енуми, функционално програмирање, уgraђене функције за рад са изузетима 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 безбедносна 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<sup>5</sup>. Да би могле да се користе опционе променљиве, прво морају бити пажљиво одмотане, коришћењем оператора '?'.

У примеру 2.36 - 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 насиљном

<sup>5</sup>Променљиве које уколико немају вредност, су једнаке *nil*

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

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 elemenata 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 2.36: Коришћење опцијоне променљиве

## Ефикасно извршавање

*Swift* је наследник програмских језика *C* и *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

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

који може свако користити [2]. Најбоља апликација за почетнике је *Swift Playgrounds* [4], апликација намењена уређајима *iPad*, развијена од стране *Apple*-а.

### Отвореног кода

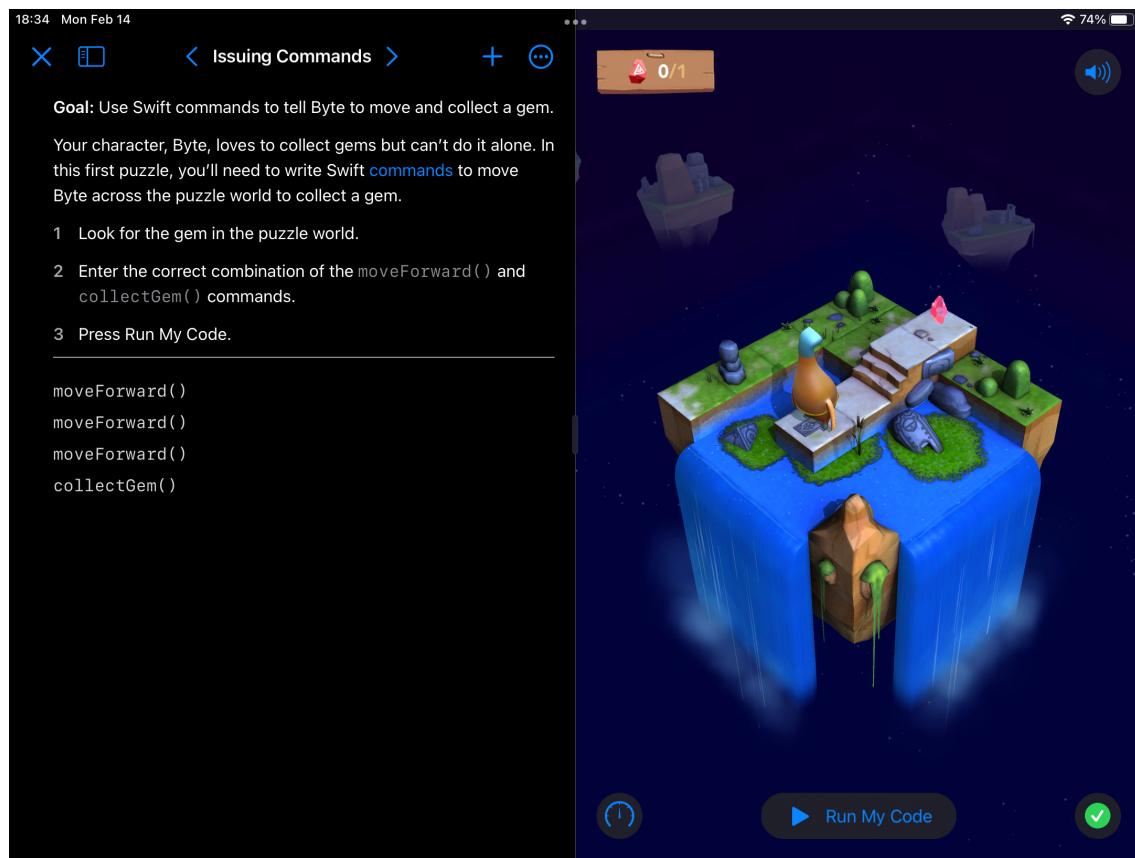
Крајем 2015. године програмски језик *Swift* је под лиценцом *Apache 2.0* постао пројекат отвореног кода. Заједно са пројектом језика отвореног кода су постале и пратеће библиотеке, дебагер и менаџер пакета. Изворни код се налази на *GitHub*-у где је свакоме лако доступан за преузимање и евентуалну дораду и допуну. Пројекат *Swift* се састоји од неколико засебних пројеката:

- *Swift* компајлер
- Стандардна библиотека
- *Core* библиотека
- *LLDB* дебагер
- *Swift* менаџер података
- *Xcode* подршка за *playground*

### *Swift Playgrounds*

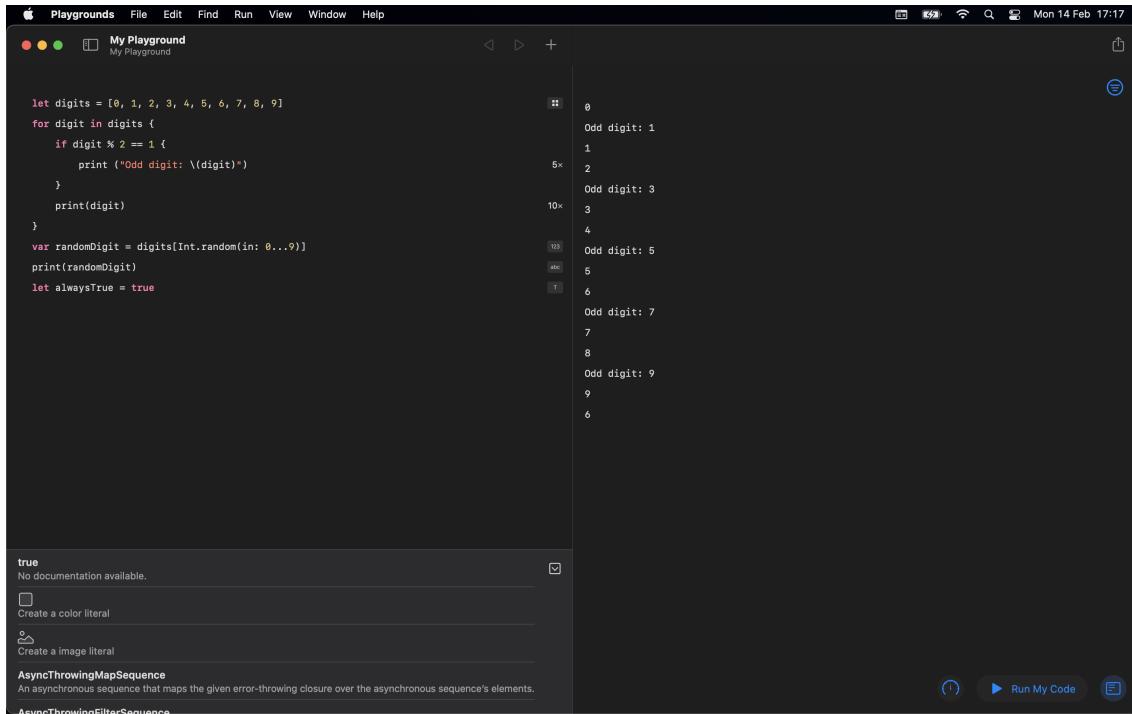
Попут апликације *Swift Playgrounds* *iPad* чији је један од пројеката намењен почетницима призан на слици 2.2, однедавно постоји апликација *Swift Playgrounds* за оперативни систем *macOS* која је одлична за почетнике, али и за искусније програмере који желе да испробају део кода или се само мало забаве. Резултат извршавања ће бити одмах приказан, као што се може видети на слици 2.3. Када пролази кроз петљу за сваки израз који утиче на рад програма биће исписано колико пута се извршио. Поред променљивих је видљив њихов конкретан тип, а уколико је променљива иницијализована и њена вредност.

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT



Слика 2.2: Приказ айлайкације Swift Playgrounds на iPad-у

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT



Слика 2.3: Приказ апликације *Swift Playgrounds* на *macOS*-у

## 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.

## Компабилан 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 које се може пронаћи на *Apple*-вом

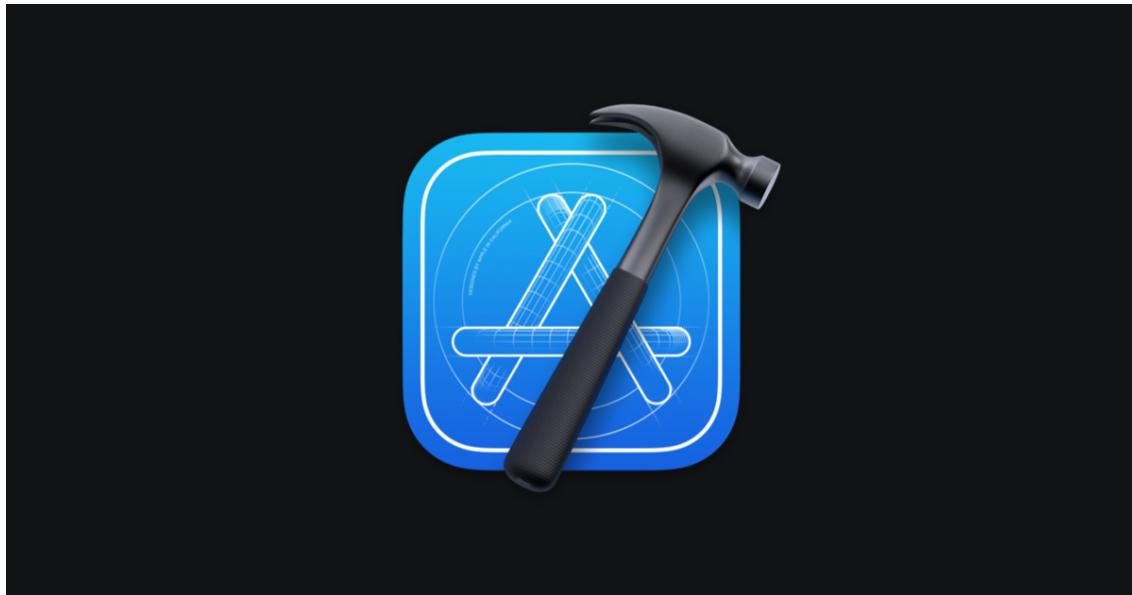
сајту посвећеном развијаоцима софтвера (енг. developers) [1], конкретно на адресама *Importing Objective-C into Swift* и *Importing Swift into Objective-C*.

## 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<sup>6</sup>.

### Основно

ИРО *Xcode* се користи за развој 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. На слици 2.4 може се видети званични лого ИРО-а *Xcode13*.



Слика 2.4: Званични лоѓо ИРО-а *Xcode13*

---

<sup>6</sup>Платформа која служи за дигиталну дистрибуцију апликација намењених оперативном систему *macOS*

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

*Xcode* 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, 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. На слици 2.5 може се видети истовремена употреба симулатора телефона *iPhone 13 Pro Max* са оперативним системом *iOS 15.0* и подешеном тамном бојом приказивања (енг. dark appearance), телефона *iPhone SE - 2nd generation* такође са оперативним системом *iOS 15.0*, али светлом бојом приказивања, таблета *iPad Pro (9.7-inch)* са оперативним системом *iOS 12.0* и паметног сата *Series 7 - 45mm* са оперативним системом *watchOS 8.0*.

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT



Слика 2.5: Приказ неколико симулатора

## 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 значјна уштеда времена приликом писања новог кода и одржавања већ постојећег. Пример једноставног кода у *SwiftUI*-у приказан је у делу 2.37 - *Пример SwiftUI кога*. Модификатор `@State` биће објашњен у делу 2.6 - Станje 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) {
14            recept in
15                // Prikaz slike
16                Image(recept.slika)
17                // Definisanje vertikalnog skupa elemenata
18                VStack(alignment: .leading) {
19                    // Prikaz teksta
20                    Text(recept.ime)
21                    // Prikaz teksta sive boje
22                    Text(recept.vremePripreme)
23                        .color(.gray)
24                }
25            }
26        }
```

Listing 2.37: *Пример SwiftUI кога*

## 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<sup>7</sup> (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:

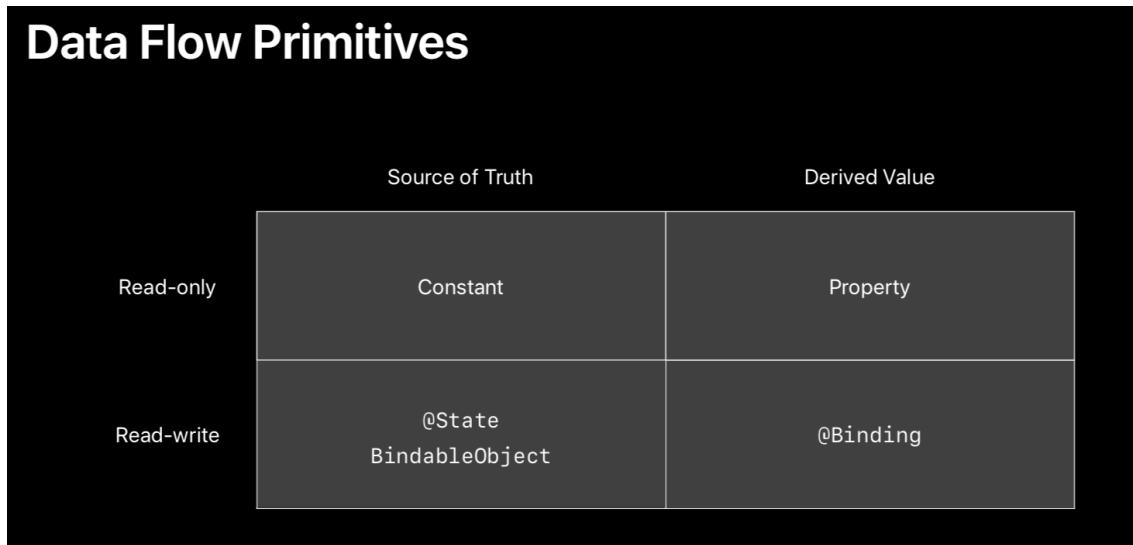
- *State* - Омогућава локално управљање стањем корисничког интерфејса, пример 2.38 - *Омотачи података - State*. Када је променљива означена као *State* другом погледу се мора проследити са префиксом '\$' уколико се жели омогућити промена њене вредности
- *BindableObject* - Користећи *ObservedObject* омотач својства, може се приступити спољашњој 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 2.39 - *Омотачи података - Binding*
- *Environment* - Подаци сачувани у *Environment-y* се могу делити кроз целу апликацију, пример 2.40 - *Омотачи података - Environment*
- *PreferenceKey* - Прослеђивање података уз хијархију погледа, од детета ка родитељу
- *FetchRequest* - Управљање трајним подацима који се чувају унутар *Core Data*

---

<sup>7</sup>Једини извор истине је начин структуирања информационих модела и шеме података тако да се сваки податак обрађује и мења на само једном месту

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

Графички приказ модификатора може се видети на слици 2.6 - *Различићи омоћачи уогаћака.*



Слика 2.6: *Различићи омоћачи уогаћака*

```
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  
9             // označava da li je recept medju omiljenima (puna  
10            // zvezdica - jeste, prazna - nije)  
11            OmiljenRecept(daLiJe: $daLiJeOmiljen)  
12        }  
13    }  
14}
```

Listing 2.38: *Омоћачи уогаћака - State*

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

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

```
5  var body: some View {
6      Button(action: {
7          // Akcija dugmeta koja menja promenljivu 'daLiJe0miljen'
8          self.daLiJe0miljen.toggle()
9      }) {
10         // Provera promenljive 'daLiJe0miljen' i prikaz
11         // odgovarajuce slike
12         Image(systemName: daLiJe0miljen ? "star.fill" : "star.
13             empty")
14     }
15 }
```

Listing 2.39: Омогућачи уогађаја - Binding

```
1 struct Kulinarstvo_widgetEntryView : View {
2     var entry: Provider.Entry
3
4     // Cita podatke za 'widgetFamily' iz okruzenja aplikacije i smesta
5     // ih u promenljivu 'widgetFamily'
6     @Environment(\.widgetFamily) var widgetFamily
7
8     @ViewBuilder
9     var body: some View {
10        // U zavisnosti od promenljive 'widgetFamily' prikazuje se
11        // odgovarajuci widget
12        switch widgetFamily {
13            case .systemSmall:
14                RecipeView(recipe: entry.recipe)
15                    .widgetURL(entry.recipe.url)
16            case .systemMedium:
17                RecipeMediumView(recipe: entry.recipe, ingredients: entry.
18                    recipe.ingredients.count > 3 ? Array(entry.recipe.
19                        ingredients.dropLast(entry.recipe.ingredients.count -
20                            3)) : entry.recipe.ingredients)
21            default:
22                Text(" ")
23        }
24    }
25 }
```

Listing 2.40: Омогућачи уогађаја - Environment

## Разлика SwiftUI и UIKit

*UIKit* и *SwiftUI* су радна окружења развијена од стране *Apple-a*, 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 доволно 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* и *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-a 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*

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

првенствено базира на *MVC*<sup>8</sup> обрасцу, док за разлику од њега, *SwiftUI* користи *MVVM*<sup>9</sup> 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

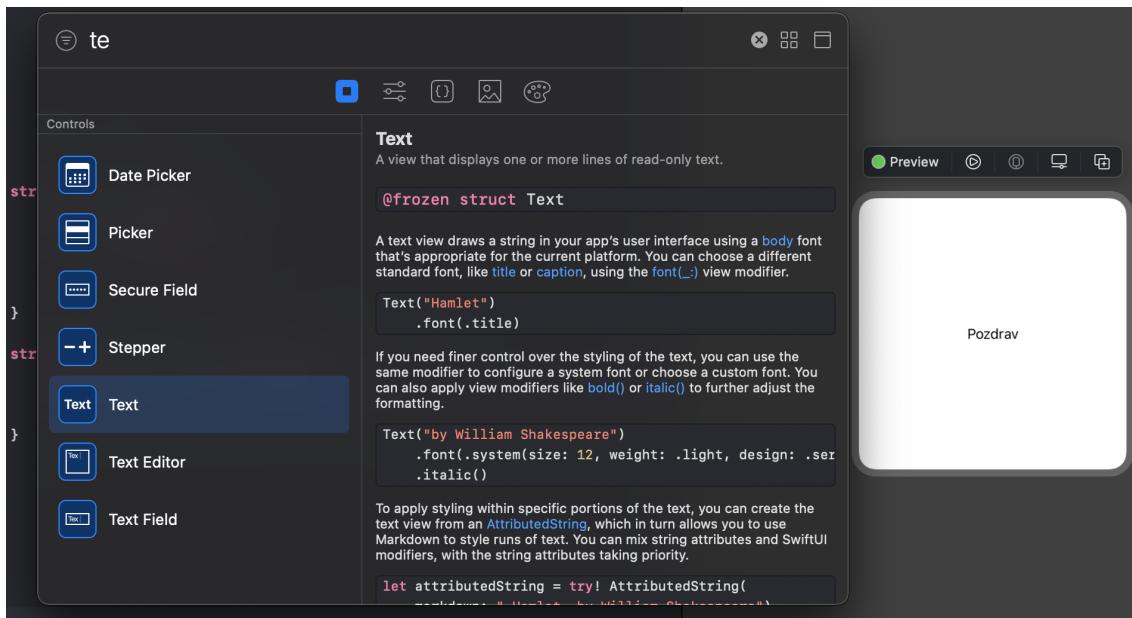
Са представљањем *SwiftUI*-а, *Apple* је представио и нову верзију њиховог ИРО-а, *Xcode11*, у коме је додато својство рада у новом радном окружењу као и могућност прегледа уживо сваког погледа. Предност оваквог начина писања кода је пре свега у могућности брзог прегледа измена и то без поновног обнављања (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 испуштања 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. Избор графичких елемената може се видети на слици 2.7, док је код програма и приказ уживо након испуштања елемента '*Text*' приказан на слици 2.8.

<sup>8</sup>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

<sup>9</sup>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

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT



Слика 2.7: Приказ једнога графичког елемената



Слика 2.8: Код је приказан у предваритељу

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 2.41 - *Xcode - уређајег ујаснио.*

```
1
2 struct PlaceholderView : View {
3     var body : some View {
```

## ГЛАВА 2. ПРОГРАМСКИ ЈЕЗИК SWIFT

---

```
4     Kulinarstvo_widgetEntryView(entry: SimpleEntry(date: Date
5         (), configuration: ConfigurationIntent(), recipe:
6             RecipeModel.testData[0]))
7 }
8
9 // Struktura u kojoj se konfigurise prikaz uzivo
10 struct Kulinarstvo_widget_Previews: PreviewProvider {
11     static var previews: some View {
12         // Grupisanje vise pogleda
13         Group {
14             // Prikaz malog widget-a sa prvim elementom iz liste
15             Kulinarstvo_widgetEntryView(entry: SimpleEntry(date:
16                 Date(), configuration: ConfigurationIntent(),
17                 recipe: RecipeModel.testData[0]))
18                 .previewContext(WidgetPreviewContext(family: .
19                     systemSmall))
20
21             // Prikaz srednjeg widget-a sa skrivenim sadrzajem
22             PlaceholderView()
23                 .previewContext(WidgetPreviewContext(family: .
24                     systemMedium))
25                 .redacted(reason: .placeholder)
26         }
27     }
28 }
```

Listing 2.41: Xcode - уређај 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, виште величине *widget*-а, истих погледа са светлом и тамном бојом позадине, погледа на различитим језицима...

За оне који желе да истраже виште 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-a* у оперативном систему *Android*, који су присутни више од десет година, *widget-i* 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

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

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

## ГЛАВА 3. УЛОГА И РАЗВОЈ WIDGET-A

---

### 3.2 - Развој Widget-a.

На оперативним системима *iOS* и *iPadOS*, галерија има могућност додања 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 не-деље) 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*<sup>1</sup> 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-a

*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,

---

<sup>1</sup>Интелигентни лични асистент на уређајима са *Apple* платформом

## ГЛАВА 3. УЛОГА И РАЗВОЈ WIDGET-A

---

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

### 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 и изабрати тим који ради на пројекту
4. Уколико *widget* подржава 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, системска апликација за временску прогнозу где корисник може наместити одређени град за који жели да добија

## ГЛАВА 3. УЛОГА И РАЗВОЈ WIDGET-A

---

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-a* 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*

Пример почетне статичке конфигурације *widget-a* може се видети у коду 3.1 - *Widget - почетна конфигурација*. Да би се подесила боја шеме прослеђује се променљива *colorScheme* којом се боја *widget-a* усклађује са системском бојом уређаја. Променљива *kind* служи за јединствену идентификацију типа *widget-a*.

```
1 struct KulinarstvoSecondWidget: Widget {
2     @Environment(\.colorScheme) var colorScheme
3
4     let kind: String = "KulinarstvoSlasnoIEfikasnoSecondWidget"
5
6     var body: some WidgetConfiguration {
7         StaticConfiguration(kind: kind, provider: SecondProvider()
8             ) { entry in
9                 KulinarstvoSecondWidgetEntryView(entry: entry)
10            }
11            .configurationDisplayName("Recept na klik")
```

## ГЛАВА 3. УЛОГА И РАЗВОЈ WIDGET-A

```
11     .description("Dodaj svoje omiljene recepte na pocetni  
12         ekran")  
13     .supportedFamilies([.systemLarge])  
14 }
```

Listing 3.1: Widget - иначејна конфигурација

### 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-a*. 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 коришћењем променљиве *isPreview* којом се проверава да ли снабдевач прегледа снимка шаље тренутни снимак за приказ у галерији или за приказ *widget-a* на почетном екрану (или *Today* погледу, или центру за обавештења). Када је параметар *isPreview* тачан, *widget* се приказује у галерији. Уколико за приказ *widget-a* треба да буду приказани и одређени подаци, а подаци нису пристигли са серверске стране, постоје два решења. Могу се приказати подразумевани, унапред одређени подаци, или се могу користити подаци који чувају место правим подацима (енг. placeholder). У примеру 3.2 - *Widget - placeholder* може се видети креирање погледа чувара места (енг. placeholder view) коришћењем статичких података који су увек доступни, и конкретан приказ тог чувара места у прегледу уживо са сакривеним подацима (енг. redacted data) - приказ како ће корисник видети *widget* док не пристигну конкретни подаци у неким ситуацијама.

```
1 struct PlaceholderView : View {  
2     var body : some View {  
3         Kulinarstvo_widgetEntryView(  
4             entry: SimpleEntry(date: Date(), configuration:  
5                 ConfigurationIntent(), recipe: Datafeed.shared.  
6                 favRecipes[0], parameterToShow: MainParameter.  
7                 Sastojci.rawValue))  
8     }  
9 }
```

## ГЛАВА 3. УЛОГА И РАЗВОЈ WIDGET-A

---

```
6     }
7
8     struct Kulinarstvo_widget_Previews: PreviewProvider {
9         static var previews: some View {
10            PlaceholderView()
11                .previewContext(WidgetPreviewContext(family: .
12                                systemLarge))
13                .redacted(reason: .placeholder)
14        }
}
```

Listing 3.2: *Widget - 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-i* представљају 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-a* од стране корисника коришћењем конфигурације *Intent*, у којој се наводе сви параметри које корисник може да промени (и дозвољене вредности за те параметре).

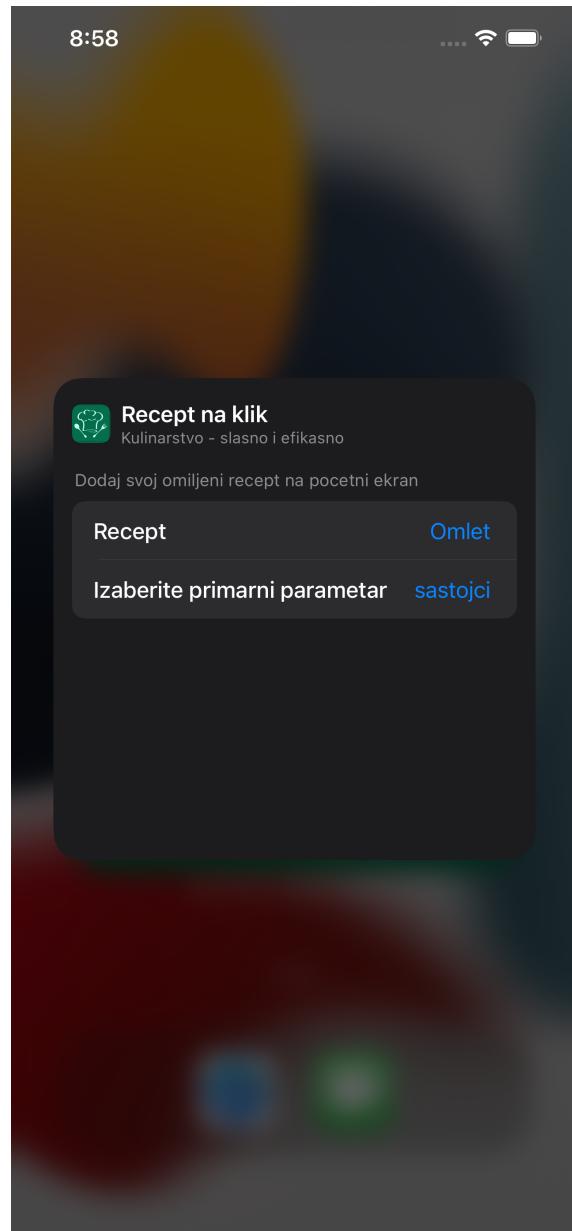
Да би се додали параметри које корисник може да конфигурише, постоје предуслови који се морају испунити:

- Додавање дефиниције *intent-a* 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 екstenзију *intent-a*

## ГЛАВА 3. УЛОГА И РАЗВОЈ WIDGET-A

---

На слици 3.1 се може видети како изгледа конфигурација *widget-a* са два параметра, први за избор рецепта који ће у *widget-y* бити приказан и избор примарног параметра уз опис рецепта (састојци или припрема).



Слика 3.1: Конфигурација *widget-a*

## Везе унутар *widget-a*

Једини начин директне комуникације између корисника и *widget-a* остварена је везама (енг. links) унутар *widget-a*. 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-a* на који је кликнуо. Свим величинама *widget-a* може бити додат модификатор *widgetURL(\_ :)*, којим се одређује у који део апликације ће корисник бити одведен када кликне на *widget*.

За све величине *widget-a*, осим малих, могу се користити и везе (енг. Link) које су додате једном елементу унутар *widget-a* 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 модификатор *widgetURL(\_ :)* из претходног примера. Овај модификатор ће бити активиран уколико корисник кликне на елеменат у *widget*-у који нема дефинисану везу. У примеру кода 3.3 - *Везе унутар widget-a* приказана је употреба модификатора *widgetURL(\_ :)* за мале *widget-e*, као и употреба веза за средње и велике *widget-e*.

```
1 struct Kulinarstvo_widgetEntryView : View {
2     var entry: Provider.Entry
3     @Environment(\.widgetFamily) var widgetFamily
4
5     @ViewBuilder
6     var body: some View {
7         switch widgetFamily {
8             case .systemSmall:
9                 ImageRecipeView(recipe: entry.recipe, isSmallView:
10                     true)
11                     .widgetURL(entry.recipe.url)
12             case .systemMedium:
13                 Link(destination: entry.recipe.url ?? URL(
14                     fileURLWithPath: ""))
15                     RecipeMediumView(recipe: entry.recipe, listName:
16                         entry.parameterToShow)
17             case .systemLarge:
18                 Link(destination: entry.recipe.url ?? URL(
```

## ГЛАВА 3. УЛОГА И РАЗВОЈ WIDGET-A

---

```
17         fileURLWithPath: "") {
18             RecipeLargeView(recipe: entry.recipe,
19                             mainParameter: entry.parameterToShow)
20         }
21     default:
22         Text(" ")
23     }
}
```

Listing 3.3: Везе унутар *widget-a*

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

Уколико желимо да користимо више различитих типова *widget-a* 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`. Пrikaz употребе више типова у једном проширењу може се видети у примеру 3.4 - *Више widget-a у једном проширењу*.

```
1 @main
2 struct ReceptWidgets: WidgetBundle {
3     @WidgetBundleBuilder
4     var body: some Widget {
5         DetaljanPrikazReceptaWidget()
6         ListaRecepataWidget()
7         SpisakZaKupovinuWidget()
8     }
9 }
```

Listing 3.4: Више *widget-a* у једном проширењу

### 3.3 Дизајн *Widget-a*

Главна улога *widget-a* 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 конфигу-

## ГЛАВА 3. УЛОГА И РАЗВОЈ WIDGET-A

---

рабилан како би кориснику допустио одређену врсту слободе прилком коришћења *widget-a* и дизајниран тако да одговара апликацији којој припада.

### Фокус *Widget-a*

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

Свака величина *widget-a* 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-a* 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-e* (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 апликације. Корисници очекује од сваког *widget-a* да им покаже корисне информације, у супротном неће наћи на добар одзив и истовремено може бити штетно самој апликацији (мањи број корисника, лошија оцена у продавници).

### Ажуарни подаци

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

Потребно је пронаћи оптимално време за ажурирање података у *widget-u*, узимајући у обзир колико се сами подаци често мењају и колико често корисницима може бити корисно да виде те податке. Уколико након публикације *widget-a* уочимо да су корисницима чешће или ређе потребни новији подаци, можемо променити време између два ажурирања и тиме побољшати корисничко искуство. Још једна корисна ствар код информација које су временен-

## ГЛАВА 3. УЛОГА И РАЗВОЈ WIDGET-A

---

ски 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 истичали 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 оптимална. Уколико се прикаже премало информација *widget* неће имати пре-

## ГЛАВА 3. УЛОГА И РАЗВОЈ WIDGET-A

---

велики 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-a* за обе врсте боја системске позадине (светле и тамне). Дизајн оба *widget-a* 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<sup>2</sup>. Коришћење мањег фонта би корисницима знатно отежало употребу *widget-a*. 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-a* 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-u* потребно је да имају исту функционалност јер би у супротном корисници били збуњени.

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

Скалабилност елемената унутар *widget-a* је веома важна јер систем аутоматски прилагођава величину *widget-a* 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.

---

<sup>2</sup>*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/>.