Univerzitet u Beogradu

Matematički fakultet

Master rad

Analiza karakteristika programskog jezika Kotlin

Student:
Jana Protić

Mentor:
dr Vladimir Filipović

Beograd, 2022.

Mentor: **prof. dr Vladimir Filipović**
Matematički fakultet, Univerzitet u Beogradu

Članovi komisije: **prof. dr Filip Marić**
Matematički fakultet, Univerzitet u Beogradu

dr Jovana Kovačević
Matematički fakultet, Univerzitet u Beogradu

Datum odbrane: _____

Analiza karakteristika programskog jezika Kotlin

Apstrakt - Kotlin je statički tipiziran, objektno-orijentisan programski jezik razvijan od strane kompanije JetBrains. Pojavio se 2011. godine i jezik je otvorenog koda. U početku je osmišljen kao jezik koji će raditi na JVM (Java Virtual Machine) platformi, ali koji, za razliku od Jave, neće morati da bude kompatibilan sa starijim verzijama, pa će moći da omogući industriji sve one dugo očekivane karakteristike koje je iščekivala od Jave. U međuvremenu je napravljen napredak od te početne ideje, pa sada Kotlin može da se kompajlira i u JavaScript i u mašinski kod (eng. native code). Pored toga, Kotlin je podržan i od strane kompanije Google, pa sada omogućava razvoj Android aplikacija. Zbog svoje interoperabilnosti sa Javom Kotlin ima sve one biblioteke koje su dostupne i u Javi, tako da podrška zajednice već postoji. Ovaj jezik je doneo dugo očekivane karakteristike, kao što su: interpolacija stringova, zaključivanje tipova, pametna konverzija tipova, imenovani i podrazumevani argumenti i mnoge druge.

Programski jezik Kotlin je u ovom trenutku široko zastupljen u industriji. On često predstavlja izabrani programski jezik za aplikacije koje se tek kreiraju (mnoge nove aplikacije su u celosti napisane u njemu - Corda), ali se koristi i kao podrška već postojećem programskom kodu (npr. Pinterest, Uber, Gradle, Evernote, Atlassian, Coursera itd.).

Cilj rada je istraživanje i prikazivanje karakteristika jezika Kotlin kroz primere. Biće upoređeni isti kod u Kotlinu i kod u Javi radi prikaza čitljivosti i sažetosti koju Kotlin pruža. Po potrebi će se izvršiti i poređenja sa nekim drugim programskim jezicima koji imaju karakteristike najsličnije Kotlinu. Programski kod tokom izrade ovog master rada je javno dostupan kao softver otvorenog koda i nalazi se u GitHub repozitorijumu na adresi: https://github.com/proticjana/memories.

Sadržaj

1. Uvod

Jezik Kotlin je razvijen 2011. godine za potrebe kompanije Džetbrejns (*eng. JetBrains*). Bio im je potreban programski jezik koji će se brzo prevoditi i imati sve neophodne karakteristike modernih jezika. Scala nije došla u obzir jer se sporije prevodi od Jave, a Java je imala prespor odgovor na potrebe zajednice. Novi programski jezik je bio Kotlin sa sledećim odlikama:

- **radi na Java virtuelnoj mašini** (*eng. Java Virtual Machine, JVM*); kod se kompajlira u bajtkod (*eng. bytecode*) koji se pokreće na JVM-u što omogućava pravljenje multiplatformskih aplikacija
- slično kao i Java**,** Kotlin je **statički tipiziran objektno-orijentisan** jezik; uz ovo koristi i mnoge koncepte funkcionalnih jezika, pa podržava i funkcionalnu paradigmu, ali takođe ima podršku i za reaktivnu i konkurentnu paradigmu
- **koncizan**, omogućava isto ponašanje koda sa mnogo manje napisanih linija nego što je slučaj sa Javom; ovime se povećava čitljivost koda, što takođe smanjuje i verovatnoću pravljenja greški, pa ga čini i veoma **bezbednim** jezikom
- **potpuna interoperabilnost sa Java kodom**; zbog već pomenutog pokretanja na JVM-u u Kotlin projektima se mogu direktno koristiti sve Java biblioteke; ovim je dobijen jezik koji ima već postojeću podršku industrije
- jezik **otvorenog koda**; ovo je danas još značajnija karakteristika s obzirom da je za Javu od 2019. potrebna licenca za komercijalno korišćenje
- **kompajliranje** Kotlin koda je u proseku 10-15% sporije nego kompajliranje Java koda; ukoliko se uzme u obzir inkrementalno kompajliranje koda, brzina za oba jezika postaje vrlo slična

2. Primene

Kotlin je prepoznat u industriji kao jezik sa mnogo prednosti, pa nije iznenađujuć spektar njegovih primena:

- **Razvoj mobilnih aplikacija za razne platforme** (*eng. Multiplatform mobile*)
    - Omogućava razvijanje zajedničkog backend-a biznis logike iOS i Android aplikacija. Specifičnosti vezane za platformu se pišu samo onde gde je neophodno, npr. nativni UI ili API-evi platforme.
- **Razvoj aplikacija sa serverske strane** (*eng. Server side*)
    - Široko se koristi za aplikacije koje rade na strani servera. Podržava rad sa mnogim već vodećim radnim okvirima u ovom polju kao što su Spring (Java radni okvir), Ktor (Kotlin radni okvir), Micronaut (za mikroservisne aplikacije), Quarkus (aplikacije za rad na oblaku) itd.
- **Nativne aplikacije** (*eng. Native)*
    - Za programiranje na nivou blizu hardvera (*eng. embedded*). Tamo gde ne može ili je isuviše komplikovano koristiti virtuelne mašine.
- **Nauka o podacima** (*eng. Data science)*
    - Ima odličnu podršku za mašinsko učenje. Postoje razne biblioteke u Kotlinu koje olakšavaju rad sa obradom podataka, kao i različiti već poznati alata u ovoj oblasti (Jupyter Notebook, Apache Zeppelin, ...)
- **Programiranje za veb** (*eng. Web frontend*)
    - Kotlin/JS omogućava transpilaciju Kotlin koda u JavaScript. Ovime se omogućava pisanje čitkog i bezbednog koda za veb aplikacije.
- **Mobilne aplikacije za Android operativni sistem**
    - 2019. kompanija Gugl (*eng. Google)* zvanično je dala podršku Kotlinu i od tog momenta on postaje glavni programski jezik za razvijanje Android aplikacija.

3. Osnovne karakteristike

3.1 Sintaksa

Sintaksu Kotlina odlikuju sledeće karakteristike:

- Kao što je već pomenuto, tip svakog izraza je poznat u vreme prevođenja, što čini Kotlin statički tipiziranim jezikom. Za razliku od Jave tip ne mora biti eksplicitno naveden već može biti zaključen iz konteksta (*eng. type inference*)

PRIMER 1	
umesto	`val godine: Int = 27`
piše se	`val godine = 27`

- Kraj naredbe se ne završava simbolom ; i ako to nije striktno pravilo. Nekad, najčešće radi pojačane čitljivosti, može se staviti ; što kompajler neće prepoznati kao grešku, ali će mnoga radna okruženja prijaviti kao upozorenje
- U Kotlinu svi entiteti su objekti, ne postoje primitivni tipovi. I ako su neki tipovi (brojevi, karakteri i bulovski tip) u vreme izvršavanja posmatrani kao primitivni, pri pisanju koda imaju sve odlike objekata, te se na njima mogu pozivati metode
- Promenljive se u Kotlinu označavaju ključnom rečju *var* ili ključnom rečju *val*. One koje su deklarisane sa *var* su mutabilne (*eng. mutable*) tokom izvršavanja programa, tj. može im se menjati vrednost. Sa *val* se deklarišu konstante, tj. imutabilne (*eng. immutable*) vrednosti, koje neće menjati svoju vrednost tokom celog izvršavanja programa
- Karakteri se pišu unutar simbola ’ a nizovi karaktera (*eng. string*) unutar “

- Još jedna odlika koja podržava čitkost koda je i to što se brojevi tipa *int* i *long* mogu pisati sa simbolom _.

PRIMER 2

```
val broj = 100_000_000
```

3.2 Kontrola toka

U Kotlinu se izrazi kontrole toka *if, if/else, else, while* i *do while* koriste slično kao u Javi, pa se njima neće posvetiti pažnja u radu. Ono što je izmenjeno je *for* petlja. Sintaksa istog koda je prikazana u narednom primeru.

PRIMER 3

sintaksa u Javi	`for (int i = 0; i < 10; i++) { ... }`
sintaksa u Kotlinu	`for (i in 1..10) { ... }` `for (i in 1 until 10) { ... }`

Ovakav kod je koncizniji, a samim tim i čitkiji. Pored ova dva načina Kotlin dodaje i dodatne načine korišćenja *for* petlje.

PRIMER 4

```
for (i in 10 downTo 1 step 2) { ... }
```

Vrlo konciznim kodom, bez mnogo napisanih linija, očito je da ova *for* petlja iterira kroz promenljivu koja redom uzima vrednosti 10, 8, 6, 4 i 2.

3.2.1 When naredba

Naredba *switch*, za kontrolu toka pri različitim vrednostima koje može imati neki izraz, a koja je široko korišćena u Javi i mnogim drugim programskim jezicima, ne postoji u programskom jeziku Kotlin. Umesto nje se uvodi naredba ***when*.** Naredba *when* može u potpunosti zameniti svaku *switch* naredbu.

PRIMER 5	

sintaksa u Javi	`switch (mesto) {`
	`  case 1:`
	`  println("Zlato");`
	`  break;`
	`  case 2:`
	`  println("Srebro");`
	`  break;`
	`  case 3:`
	`  println("Bronza");`
	`  break;`
	`  default:`
	`  println("Nije među prva tri mesta.");`
	`}`

sintaksa u Kotlinu	`when (mesto) {`
	`  1 -> println("Zlato")`
	`  2 -> println("Srebro")`
	`  3 -> println("Bronza")`
	`  else -> println("Nije među prva tri mesta.")`
	`}`

U ovom primeru vidimo da su repetativne naredbe prekida (*eng. break*), iz *switch* naredbe, u *when* naredbi izbačene i podrazumevane. Podrazumevana opcija *(eng. default)* je zamenjena ključnom rečju *else*, slično kao i u *if* naredbi.

U narednom primeru, primećuje se kako je sintaksa sa ključnim rečima *case* izmenjena radi postizanja konciznosti (primer 6). Dok se u primeru nakon njega prikazuje da je takvom sintaksom povećana i izražajnost jezika (primer 7).

PRIMER 6

```
when (mesto) {
    1, 2, 3 -> println("Top tri")
    in 4 .. 10 -> println("Top deset")
    in srecnici -> println("Osvojeno srećno mesto.")
        // val srecnici = arrayListOf(13, 23, 100)
    else -> println("Bez nagrade.")
}
```

PRIMER 7

```
when (zivoBice) {
    is Pas -> zivoBice.baciLoptu()
    is Zivotinja -> zivoBice.nahrani()
    is Biljka -> zivoBice.dajVodu()
    else -> println("Možda ipak nije živo biće.")
}
```

3.3 Prazne vrednosti

Ni jedan objekat ne može sadržati praznu vrednost, sem ako je drugačije naznačeno. Ovako se obezbeđuje bezbednost od prazne vrednosti (*eng. null safety*). Prazna vrednost se omogućava pri deklaraciji promenljive gde se tip eksplicitno zadaje sa pratećim znakom pitanja.

PRIMER 8

```
var ime : String?
```

Kada se pristupa poljima objekta koji može imati praznu vrednost obavezno se dodaje znak pitanja nakon promenljive.

PRIMER 9

```
ime?.length
```

Ovako se izbegava veoma čest izuzetak u vreme izvršavanja programa, izuzetak referisanja praznom vrednošću (*eng. Null Pointer Exception*, *NPE*). U slučaju da promenjiva sadrži praznu vrednost rezultat će takodje biti prazna vrednost, te i rezultat mora biti tipa koji prihvata i prazne vrednosti.

Ovo je korak napred u odnosu na Javu. Umesto čestih provera da li su vrednosti prazne u petljama skraćujemo kod i samim time mu povećavamo čitljivost.

PRIMER 10

umesto

```
if (ime != null) {
    duzinaImena = ime.length();
} else {
    duzinaImena = null;
}
```

piše se

```
duzinaImena = ime?.length
```

PRIMER 11

umesto

```
if (ime != null) {
    println("Ime ima dužinu: " + ime.length());
}
```

piše se

```
ime?.let { println("Ime ima dužinu $it.length") }
```

PRIMER 12	

umesto	`if (auto != null) {`
	`  auto.pokreni();`
	`}`

piše se	`auto?.pokreni()`

Ova odlika u jeziku Kotlin je izrazito korisna u slučaju više provera gde se vrši takozvano nadovezivanje koje bi u Javi zahtevalo nekoliko provera.

PRIMER 13	

umesto	`if (korisnik != null) {`
	`  if (skolovanje != null) {`
	`  if (fakultet != null) {`
	`  korisnik.getSkolovanje()`
	`  .getFakultet()`
	`  .getGodinaUpisa();`
	`  }`
	`  }`
	`}`

piše se	`korisnik?.skolovanje?.fakultet?.godinaUpisa`

Kada se radi sa promenljivima koje mogu sadržati i prazne vrednosti, često se koristi tzv. Elvis operator (*?:*) koji predstavlja skraćenje ternarnog operatora uslovnog izraza. Ovaj operator označava korišćenje navedene vrednosti ukoliko ona nije prazna, a ukoliko je prazna onda se postavlja predefinisana vrednost.

PRIMER 14	
umesto	`ime = (imeKorisnika != null) ? imeKorisnika : "Gost"`

piše se	`ime = imeKorisnika ?: "Gost"`

U slučajevima kada je apsolutno sigurno da promenljiva, i ako je dozvoljeno da sadrži praznu vrednost, u momentu pristupanja ne sadrži istu koriste se dva znaka uzvika.

PRIMER 15

```
imeKorisnika!!.toLowerCase()
```

U slučaju da ona ipak sadrži praznu vrednost, u toku izvršavanja programa, javiće se već pomenuti NPE.

3.4 Izuzeci koji se proveravaju

Za razliku od Jave, Kotlin ne podržava izuzetke koji se proveravaju *(eng. checked exceptions)*. Samim time svo rukovanje takvim izuzecima u *try-catch* blokovima nije obavezno.

Stvaraoci jezika Kotlin su se odlučili na ovaj korak vođeni iskustvom Java zajednice i nezadovoljstvom rukovanja izuzecima koji se proveravaju. Ovakvi izuzeci postoje u Javi da bi obezbedili što veću sigurnost koda, ali njihovo rukovanje ipak oduzima neko vreme i generiše se dosta šablonskog koda, pa se gubi i na konciznosti koda. Razlog njihovog izbacivanja u Kotlinu je u mnogome što većina projekata brzo preraste tu granicu gde se trud oko rukovanja jednostavno ne isplati.

4. Klase

U Kotlinu je sve klasa. Svi tipovi i objektni i primitivni (celobrojni, realni, logički, itd.) su klase. Zbog toga pisanje klasa je svedeno na jednostavan nivo gde se veliki deo šablonskog koda (*eng. boilerplate code*) izostavlja i podrazumeva.

4.1 Sintaksa

Kao što je već napomenuto, veliki deo koda pri pisanju klasa u Kotlinu se podrazumeva. U narednom primeru uporedićemo definiciju iste klase u Javi i u Kotlinu.

PRIMER 16

definicija klase u Javi

```
class Osoba {
    private final String ime;
    private Integer godine;

    public Osoba(String ime) {
        this.ime = ime;
        this.godine = 25; }

    public Osoba(String ime, Integer godine) {
        this.ime = ime;
        this.godine = godine; }

    public String getIme() {
        return this.ime; }

    public Integer getGodine() {
        return this.godine; }

    public void setGodine(Integer noveGodine) {
        this.godine = noveGodine; }
}
```

definicija klase u Kotlinu	`class Osoba (val ime: String, var godine: Int? = 25)`

Ove dve definicije klase Person generišu potpuno isti bajtkod te reprezentuju potpuno identičnu klasu. Subjektivno osećaj može biti da je preveliki deo koda podrazumevan, ali ukoliko su poznata sva pravila koja Kotlin definiše, sve je vrlo očito. Može se uočiti sledeće:

- Konstruktori u Kotlinu su podrazumevani i pored primarnog konstruktora koji postavlja sva polja podrazumevan će biti i konstrukor koji postavlja samo neka od atributa klase, a ostala postavlja na podrazumevane vrednosti
- Ključna reč *val* podrazumeva da je taj atribut konstanta i da se neće menjati tokom života objekta, što se u Java kodu definiše ključnom rečju *final*. Za te atribute će se podrazumevati i pristupna metoda za čitanje atributa (*eng. getter*).
- Ključna reč *var* podrazumeva da je taj atribut mutabilan tj. da se njegova vrednost može menjati tokom života objekta. Za te atribute će se podrazumevati i pristupne metode za čitanje, kao i za menjanje vrednosti atributa (*eng. getter and setter methods*).
- U Kotlinu pristupni atribut (*eng. accessor*) koji se podrazumeva je privatan (*eng. private*), ukoliko je potrebno postaviti neki drugi neophodno je u definiciji klase dodati i ključnu reč *constructor*.

PRIMER 17

```
class Osoba constructor(
    val ime: String,
    private var godine: Int? = 25)
```

4.1.1 Any

Kotlin ima klasu koja je natklasa svih klasa, klasu *Any*. Ta klasa nema atribute, već samo metode *equals()*, *hashCode()* i *toString()*. I ako ova klasa podseća na natklasu svih klasa u Javi, *Object*, njih dve ipak nisu identične. Klasa *Object*, s obzirom da potiče iz Java sveta, može sadržati i praznu vrednost. U Kotlinu ovo narušava bezbednost koda, pa se tokom rada programa (*eng. runtime*) kotlin.Any! u stvari tretira isto kao java.lang.Object.

4.1.2 Instanciranje klasa

Kreiranje objekata se vrši pozivanjem konstruktora klase i prosleđivanjem parametara. Na vrlo sličan način kao u Javi, samo bez navođenja ključne reči *new*.

PRIMER 18

```
val osoba1 = Osoba("Lena")
val osoba2 = Osoba("Milan", 27)
```

4.1.3 Nasleđivanje

Da bi klasa mogla da se nasleđuje, mora se navesti ključna reč *open* (primer 19). Ukoliko ona nije navedena, u Kotlinu su sve klase podrazumevano zatvorene za nasleđivanje. U Javi je pristup obrnut. Klase su podrazumevano otvorene za nasleđivanje, a u suprotnom se označavaju ključnom rečju *final*.

PRIMER 19

```
open class Osoba(val ime: String) {
    open fun predstaviSe() {
        println("Zdravo! Ja sam $ime.")
    }
    fun uradiNesto() {  }
}
```

Kada se nasleđuje klasa moraju se prevazići (*eng. override*) svi atributi i metode označene ključnom rečju *open*, u suprotnom će se javiti greška pri kompajliranju. Ostali atributi i metode natklase se koriste regularno, kao da su definisani u potklasi.

PRIMER 20

```
class Student(val ime: String, val smer: String): Osoba(ime) {
    override fun predstaviSe() {
        println("Zdravo! Ja sam $ime. Student sam $smer smera.")
        uradiNesto()
    }
}
```

4.1.4 Konverzija tipova

Konverzija tipova (*eng. typecasting*) u Kotlinu se realizuje pomoću ključne reči *as*, a provera tipa ključnom rečju *is*.

Ključna reč *is* se koristi da se proveri da li je neki objekat tipa neke klase. Rezultat tog izraza je bulovski tip koji pokazuje da li ona jeste (*true*) ili nije (*false*) objekat te klase.

PRIMER 21

```
val listaRaznihTipova: List<Any> =
    listOf("Milan", 25, 2, "Lena", 4.53)

for (element in listaRaznihTipova) {
    when (element) {
        is Int -> println("Integer: $element.")
        is Double ->
              println("Double zaokružen: ${element.roundToInt()}")
        is String -> println("String dužine: ${element.length}")
        else -> println("Nepoznat tip")
    }
}
```

U ovom primeru je takođe prikazana takozvana pametna konverzija (*eng. smart cast*). Nakon što je provereno da je element određenog tipa konverzija više nije neophodna, pa se u tim granama podrazumeva da je tip elementa poznat.

Ključna reč *as* se koristi kada je potrebno izvršiti eksplicitnu konverziju objekta u neki tip. S obzirom da se sa *as* ne proverava samo da li je objekat određenog tipa, već se i vrši sama konverzija, može doći do izuzetka tipa *ClassCastException*.

PRIMER 22

```
var objekat: Any = "Milan"
var string = objekat as String
println(string.length)

objekat = 123
string = objekat as String // Izuzetak tipa ClassCastException
println(string.length)
```

Da ne bi dolazilo do izuzetaka koristi se *as?*. Tada se konverzija vrši u slučaju kada je objekat odgovarajućeg tipa, a u suprotnom je rezultat *null*.

PRIMER 23

```
val objekat = 123
val string: String? = objekat as? String
println(string?.length) // Ispisuje null
```

4.1.5 Klase podataka

Kotlin za klase koje predstavljaju „čiste“ podatke, entitete koji ne treba da imaju neke funkcionalnosti već samo da sadrže podatke, uvodi klase podataka (*eng. data classes*). Najbolji primer za ovakve klase su entiteti baze podataka. One se definišu ključnom rečju *data*.

PRIMER 24

```
data class Osoba (val ime: String, val godine: Int)
```

Za ovako definisane klase se tokom kompajliranja generišu i metode *equals()* i *hashCode()*, metoda *toString()*, metode *componentN()* za svaki od atributa (više o ovome u poglavlju 4.5), kao i funkcija *copy()*. Tačnije, generiše se sve što je neophodno za manipulaciju objektima ovakvih klasa podataka, što je u Javi zahtevalo dosta šablonskog koda.

Java od verzije 14 uvodi podršku za definisanje ovakvih klasa, pomoću ključne reči *record*. Međutim Java 11 koja je stabilna LTS verzija (*eng. long term support*) nema ovu funkcionalnost.

4.1.6 Prateći objekat

U Kotlinu ne postoji ključna reč *static* iz Jave. Sve što je treba da bude dostupno bez instanciranja klase se definiše tako što se "pakuje" u prateći objekat klase (*eng. companion object*) i koristi kao što bi se koristili statički atributi i metode u Java kodu.

PRIMER 25

```
class CentarPodrske {
    companion object {
        var telefonskiBroj: String = "08001234567"
        fun pozovi() = println("Poziva se $ telefonskiBroj")
    }
}

fun main() {
    CentarPodrske.telefonskiBroj = "08007654321"
    println(CentarPodrske.pozovi())
}
```

4.1.7 Nastavci klase

Kotlin uvodi jednu veliku novinu, dugo očekivanu i porebnu u Java svetu: nastavci klasa (*eng. extensions*). Ovo svojstvo daje veliku moć jer omogućava proširivanje već postojećih klasa, često delova nekih standardnih ili popularnih biblioteka, metodama koje su nam potrebne za njihovu manipulaciju.

PRIMER 26

```
fun String.kreirajSifru(): String {
    return this.replace("s", "5").replace("i", "1")
}

fun main() {
    val koren = "Assassin'sCreed"
    println(koren.kreirajSifru()) // ispisuje A55a551n'5Creed
}
```

Da bi se isto postiglo u Javi morala bi se napraviti nova potklasa klase čiji nastavak nam treba, implementirati potrebne nastavke, a potom svuda koristiti potklasu umesto natklase.

4.2 String

Stringovi, tj. nizovi karaktera, u Kotlinu se ponašaju i koriste slično kao i u Javi s tim što je interpolacija stringova značajno olakšana.

Interpolacija stringova u Javi se može uraditi na više načina. Koristeći operator +, metodu *format()*, klase *MessageFormat* ili *StringBuilder* i, od Jave 15, metode *formatted()*. Međutim, ni jedan od ovih načina nije elegantan i čitak kao u Kotlinu.

PRIMER 27

interpolacija stringova u Javi

```
String imeSajta = "Google";
String tip = "pretraživač";

String poruka1 = imeSajta + " is a " + tip;

String poruka2 = String.format("%s is a %s",
       imeSajta, tip);

String poruka3 = MessageFormat.format("{0} is a
       {1}", imeSajta, tip);

StringBuilder poruka4 =
       new StringBuilder(imeSajta)
              .append(" is a ")
              .append(String.valueOf(tip))
```

interpolacija stringova u Kotlinu

```
val imeSajta = "Google"
val tip = "pretraživač"
val poruka = "$imeSajta is a $tip"
println(poruka)
```

4.3 Pristup atributima

Pristup atributima preko *get* i *set* konstrukata se u Kotlinu može raditi implicitno i eksplicitno.

Implicitno, kao što je predstavljeno u poglavlju 4.1, Kotlin će pri definiciji klase postaviti podrazumevane metode *get* za svaki od navedenih atributa i *set* za svaki od promenljivih (*var*) atributa. Koriste se jednostavnim navođenjem polja nad objektom.

PRIMER 28

piše se

```
class Osoba(var ime: String = "Milan")
```

podrazumeva se

```
class Osoba {
    var ime: String = "Milan"
        get() = field
        set(value) {
            field = value
        }
}
```

Metode *get* i *set* se mogu postaviti i eksplicitno ukoliko je potrebna posebna manipulacija podatkom pre njegovog korišćenja ili postavljanja.

PRIMER 29

```
class Osoba(var ime: String = "Milan") {
    var stringReprezentacija: String
        get() = this.toString()
        set(value) {
            setDataFromString(value)
        }
}

fun setDataFromString(osoba: String): Osoba { ... }
```

Tokom eksplicitnog navođenja *get* i *set* metode može se promeniti i pristupni atribut. Ukoliko je potrebno da oba budu privatna, ceo atribut se definiše kao *private*, a ukoliko je potreban samo *private set* dodaje se ključna reč kao u narednom primeru.

PRIMER 30

```
class Osoba(var ime: String = "Milan") {
    var stringReprezentacija: String
        get() = this.toString()
        private set(value) {
            setDataFromString(value)
        }
}

fun setDataFromString(osoba: String): Osoba { ... }
```

4.4 Kasna inicijalizacija

U Kotlinu postoji način za takozvanu kasnu inicijalizaciju atributa. Pri definiciji klase u Kotlinu, kao i u Javi, svi atributi klase moraju biti inicijalizovana. Da bi se izbegla inicijalizacija atributa može se staviti ključna reč *lateinit* koja kompajleru "obećava" da će taj atribut uvek biti inicijalizovan pre njegovog korišćenja.

Ukoliko to obećanje nije ispunjeno biće izazvan izuzetak neinicijalizovanog atributa (*eng. UninitializedPropertyException*). I ako deluje nebezbedno ovo daje veću fleksibilnost kodu.

Primer korišćenja ove karakteristike jezika Kotlin je tokom razvijanja Android aplikacija za koje neki atributi neće biti inicijalizovani sve dok određena aktivnost ne bude dostupna i prikazana u aplikaciji.

4.5 Deklaracija kroz dekonstrukciju

Kotlin podržava takozvanu deklaraciju kroz dekonstrukciju koja skraćuje dužinu koda i samim tim povećava njegovu čitljivost.

PRIMER 31

Java kod

```
String ime = osoba.getIme();
String prezime = osoba.getPrezime();
int godine = osoba.getGodine();
```

Kotlin kod

```
val (ime, prezime, godine) = osoba
```

Deklaracija kroz dekonstrukciju se često koristi za funkcije koje kao povratnu vrednost imaju više vrednosti od značaja. U tom slučaju se te vrednosti smeštaju u atribute neke klase, a potom pri povratku iz funkcije raščlanjavaju u promenljive za dalje korišćenje.

PRIMER 32

```
data class Rezultat(val rezultat: Int, val status: Status)

fun nekaFunkcija(): Rezultat {
    ...

    return Rezultat(rezultat, status)
}

fun main() {
    val (rezultat, status) = nekaFunkcija()
}
```

Pri kompajliranju se ovaj kod prevodi u kod koji koristi metode *componentN()*. Iste one pomenute u poglavlju 4.1.5 gde su obrađivane klase podataka sa kojima se ova dekonstrukcija najčešće i koristi.

Ova funkcionalnost se ne mora koristiti samo sa klasama podataka, već je vrlo korisna i pri radu sa nekim kompleksnijim strukturama. To je prepoznato i pri kreaciji jezika Kotlin, pa su mnoge strukture pripremljene za korišćenje dekonstrukcije. Primer je struktura *Map<K, V>* koja pored iteratora ima definisane i metode *component1()* i *component2()* koje u stvari pozivaju, tim redom, metode *getKey()* i *getValue()*. Nakon toga kroz mapu se može iterirati vrlo jednostavno.

PRIMER 33

```
val mapa: Map<Int, String>

// inicijalizacija mape
for ((kljuc, vrednost) in mapa) {
    ...
}
```

4.6 Singleton

Kotlin ima posebnu sintaksu za kreiranje Singleton objekata radi obezbeđivanja što jednostavnijeg korišćenja, ali i povećanja bezbednosti rada sa ovakvim objektima. Metode i atributi ovako definisanog objekta se koriste kao što se koriste statičke metode i atributi, dakle direktno nad objektom.

PRIMER 34

```
object FajlSistem {
    fun kreirajPrivremeniFajl() { ... }
}

fun main() {
     FajlSistem.kreirajPrivremeniFajl()
}
```

Singleton se može koristiti bez bojazni da će postojati u više instanci, jer će svaki pokušaj instanciranja kompajler prijaviti kao grešku. Takođe, ovakav objekat može nasleđivati neku klasu, ali se on sam ne može naslediti.

5. Funkcije

5.1 Sintaksa

Definisanje funkcija, odnosno metoda se u Kotlinu radi korišćenjem ključne reči *fun*, definisanjem njenog imena i parametara, povratne vrednosti i tela funkcije.

PRIMER 35

```
fun saberi(a: Int, b: Int) : Int {
    return a+b
}
```

Ista funkcija se može napisati i koristeći drugačiju sintaksu, gde se ona koncizno piše u samo jednom redu kao izraz. U tom slučaju možemo izostaviti ključnu reč *return*, pa čak i tip povratne vrednosti koji će biti zaključen implicitno.

PRIMER 36

```
fun saberi(a: Int, b: Int) = a+b
```

Ovakvo pisanje funkcija izraza daje dosta slobode u pisanju koda što je prikazano i u narednom primeru.

PRIMER 37

```
fun transformisi(boja: String): Int = when (boja) {
    "Crvena" -> 0
    "Zelena" -> 1
    "Plava" -> 2
    else -> throw IllegalArgumentException("Nedefinisana boja")
}
```

5.2 Podrazumevane vrednosti

Unutar potpisa funkcije se može postaviti podrazumevana vrednost parametara. Ovo se često koristi kod konstruktora, s obzirom da su i oni funkcije.

PRIMER 38

```
fun kreirajSifru(
    koren: String,
    velikoPocetnoSlovo: Boolean = true,
    dodajBrojeve: Boolean = false) {
    ...
}

val sifra = kreirajSifru("MojaSigurnaSifra", false, true)
```

Ovako definisana funkcija se pri kompajliranju preopterećuje (*eng. overloading*), pa se može pozvati i samo sa prosleđenim parametrima koji nemaju podrazumevanu vrednost (primer 39) ili i sa nekim od parametara čiju podrazumevanu vrednost treba promeniti (primer 40).

PRIMER 39

```
val sifra = kreirajSifru("MojaSigurnaSifra")
```

PRIMER 40

```
val sifra = kreirajSifru("MojaSigurnaSifra", dodajBrojeve = true)
```

Prethodno opisana sintaksa se sme koristiti sa funkcijama pisanim u Kotlinu, ali ne i sa funkcijama iz Java biblioteka. One ne moraju imati ista imena parametara kada se dođe do nivoa bajtkoda, pa može doći do greški prilikom kompajliranja koda.

5.3 Promenljiv broj argumenata

U Kotlinu je moguće pisati funkcije sa promenljivim brojem parametara, odnosno argumenata. Unutar te funkcije parametar se koristi kao običan niz.

PRIMER 41

```
fun saberi(vararg brojevi: Int): Int {
    var suma = 0
    brojevi.forEach { suma += it }
    return suma
}
```

Ovako definisana funkcija se može pozvati kao u prethodnom primeru, ali i na još nekoliko načina. Potpuno je validan poziv bez argumenata (primer 42), kao i poziv koji prosleđuje niz odgovarajućeg tipa i operator širenja * (*eng. spread*) (primer 43).

PRIMER 42

```
val praznaSuma = saberi() // Rezultat je 0
```

PRIMER 43

```
val brojevi = intArrayOf(1, 2, 3, 4)
val sumaBrojeva = saberi(*brojevi) // Rezultat je 10
```

Postoje određena ograničenja pri ovakvom korišćenju funkcija. Ukoliko nakon parametra sa promenljivim brojem argumenata imamo još parametara njih moramo navoditi eksplicitno. U suprotnom će se javiti greška pri kompajliranju.

PRIMER 44

```
fun kreirajKorisnika(
    vararg nivoiPristupa: String, korisnickoIme: String) {
    nivoiPristupa.forEach { println(it) }
    println(korisnickoIme)
}

kreirajKorisnika("admin", "korisnik", korisnickoIme = "desertFox")
```

Ograničenje ilustrovano prethodnim primerom može se prevazići postavljanjem parametra sa promenljivim brojem argumenata na poslednje mesto (primer 45).

PRIMER 45

```
fun kreirajKorisnika(
    korisnickoIme: String, vararg nivoiPristupa: String) {
    println(korisnickoIme)
    nivoiPristupa.forEach { println(it) }
}

kreirajKorisnika("desertFox", "admin", "korisnik")
```

5.4 Preopterećivanje operatora

Za razliku od Jave u Kotlinu se svi operatori, bilo unarni (primer 46) bilo binarni (primer 47), mogu preopteretiti. Ovo se postiže pisanjem funkcija uz ključnu reč *operator*.

PRIMER 46

```
data class Tacka(val x: Int, val y: Int)

operator fun Tacka.not() = Tacka(y, x)

val t = Tacka(4, 2)
println(!t) // Ispisuje Tacka(x=2, y=4)
```

PRIMER 47

```
data class Tacka(val x: Int, val y: Int)

operator fun Tacka.plus(druga: Tacka) =
                Tacka(x + druga.x, y + druga.y)

val t1 = Tacka(0, 1)
val t2 = Tacka(1, 2)
println(t1 + t2) // Ispisuje Tacka(x=1, y=3)
```

5.5 Lambda izrazi

S obzirom na već pomenutu podršku funkcionalnom programiranju, lambda izrazi ne dolaze kao veliko iznenađenje. Kada je Kotlin tek počinjao da se koristi Java nije imala podršku za lambda izraze, pa je ovo bila jedna od velikih prednosti Kotlina.

Lambda izrazi su u stvari anonimne funkcije koje se mogu da koristiti kao izrazi jer uvek imaju povratnu vrednost. Samim time mogu se prosleđivati kao parametri funkcija ili kao povratna vrednost funkcija.

Za razliku od Jave, zbog implicitnog zaključivanja tipova, u Kotlinu se ne mora navoditi tip povratne vrednosti lambde, već su dovoljni samo ulazni parametri i telo funkcije razdvojeni ->.

PRIMER 48

```
val kvadrat = { broj: Int -> broj * broj }
val devet = kvadrat(3)
```

Skraćeni oblik pisanja lambde se može koristiti kada lambda izraz ima samo jedan parametar. Tada se taj parametar obeležava ključnom rečju *it*.

PRIMER 49

umesto

```
val niz = arrayOf(1, 2, 3)
niz.forEach { element -> println(element * 4) }
```

piše se

```
val niz = arrayOf(1, 2, 3)
niz.forEach { println(it * 4) } // Ispisuje 4 8 12
```

Lambde se često koriste pri manipulaciji raznih kolekcija za koje su definisane metode koje kao parametar očekuju neku funkciju.

PRIMER 50

```
val brojevi = setOf(1, 2, 3)
val dupliraniBrojevi = brojevi.map { it * 2 } // [2, 4, 6]
```

6. Opis aplikacije

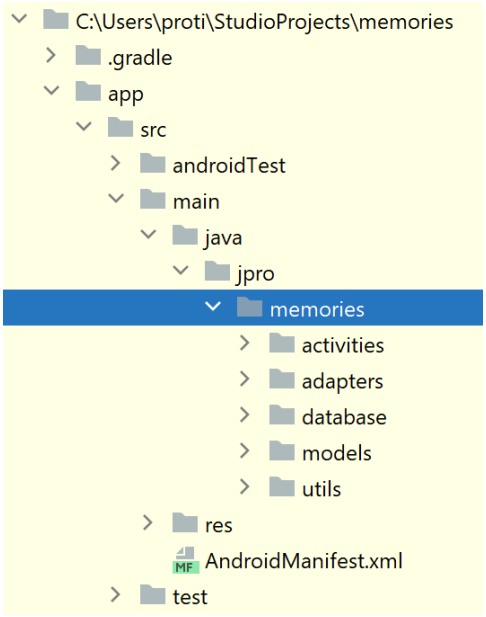
Kao demonstracija upotrebe jezika Kotlin razvijena je jednostavna aplikacija Memories, kao neka vrsta privatnog dnevnika. U ovom poglavlju je prestavljena njena struktura, koja sadrži najmanji skup podešavanja za aplikaciju ove kompleksnosti, zatim funkcionalnosti razvijene aplikacije, kao i značajne delove koda koji prikazuju karakteristike Kotlina već pomenute u ostatku rada.

Aplikacija je razvijana u okruženju AndroidStudio kompanije JetBrains, koja je nadogradnja nad njihovim okruženjem IntelliJ IDEA. Kako je ista kompanija i osnovala jezik Kotlin, to je podrška za jezik Kotlin odlična u oba okruženja.

Za kompajliranje i opšte povezivanje koda korišćen je sistem Gradle, koji predstavlja standard pri razvijanju Android aplikacija.

6.1 Struktura aplikacije

Struktura aplikacije je na narednoj slici. Aplikacija se sastoji iz dve veće podceline: baze podataka i Android aktivnosti (*eng. activity*).



Slika 1. *Struktura aplikacije*

Kao sistem za upravljanje bazom podataka je korišćena biblioteka SQLite. Ova biblioteka otvorenog koda obezbeđuje laku manipulaciju podacima. Da bi se koristila biblioteka definisane su sledeće klase:

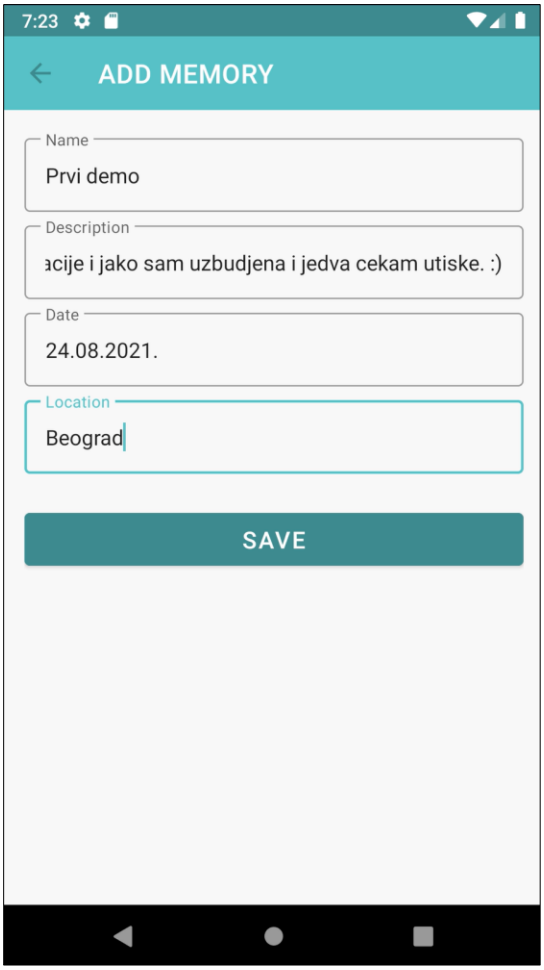
- ***DatabaseHandler*** - potklasa klase *SQLiteHelper* koja služi za upravljanje bazom podataka (kreiranje baze – instance klase *SQLiteDatabase*, kao i upravljanje podacima); nalazi se u paketu *database*
- ***MemoryModel*** – klasa koja služi da predstavi glavni entitet baze podataka Memories; nalazi se u paketu *models* i implementira interfejs *Parcelable* (iz osnovne Android biblioteke)

Aktivnost je jedna od glavnih komponenti u Android programiranju. One predstavljaju stranice sa korisničkim interfejsom. U razvijenoj aplikaciji definisane su u paketu *activities*:

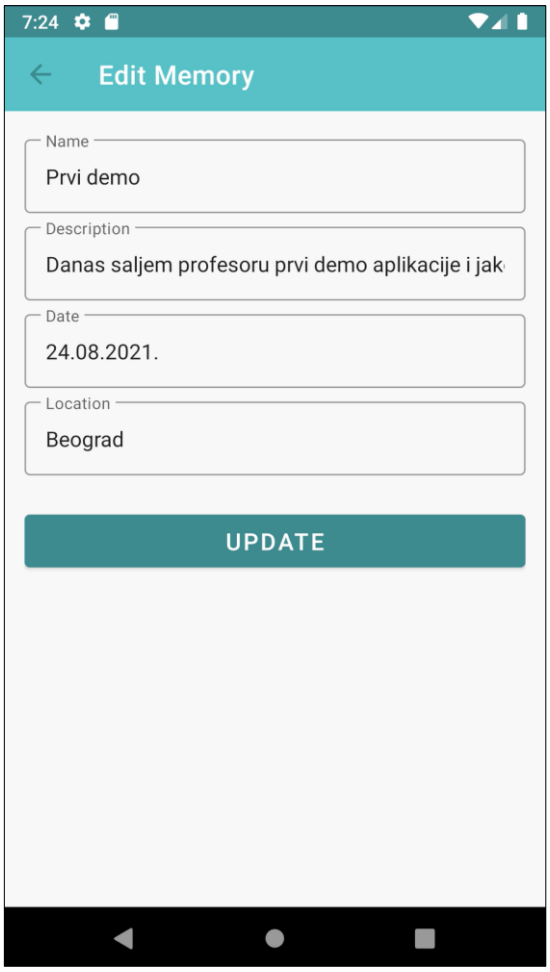
1. ***MainActivity*** – glavna stranica aplikacije gde su prikazane sve sačuvane uspomene (slika 2.)
2. ***AddMemoryActivity*** – stranica koja se koristi za dodavanje novih i izmenu postojećih uspomena (slika 3. i slika 4.)

3. ***MemoryDetailActivity*** – stranica koja detaljnije prikazuje odabranu uspomenu (slika 5.)

Slika 2. *MainActivity stranica*



Slika 3. *AddMemoryActivity stranica tokom dodavanja nove uspomene*



Slika 4. *AddMemoryActivity stranica pri menjanju postojeće uspomene*

Slika 5. *MemoryDetailActivity stranica*

6.2 Funkcionalnosti aplikacije

U razvijenoj aplikaciji podržane su sledeće funkcionalnosti:

1. **Pregled uspomena** – na već pomenutoj *MainActivity* stranici prikazane su sve uspomene koje su skladištene (slika 2.). Za ovo je korišćen standardni Android način koji se koristi klasom *RecyclerView* i pravljenjem adaptera za entitete koji će biti prikazani u listi.

2. **Dodavanje novih uspomena** – sa glavne stranice, klikom na +, otvara se stranica za dodavanje novih uspomena *AddMemoryActivity* (slika 3.). Na njoj se unose ime i opis uspomene, bira se njen datum na odabiraču datuma (*eng. DatePicker*) i lokacija. Klikom na *Save* uspomena se čuva u bazu podataka. Ukoliko neko od polja nije popunjeno, pri kliku na *Save* korisnik će biti obavešten da mora uneti podatke za određeno obavezno polje.

3. **Izmena uspomena** – ukoliko se na *MainActivity* stranici prevuče prstom na desno (slika 6.) otvoriće se stranica *AddMemoryActivity* gde će biti moguća izmena nekog od polja postojeće uspomene (slika 4.). Nakon izmene željenih polja, klikom na *Update* postojeća uspomena će biti promenjena u bazi podataka.

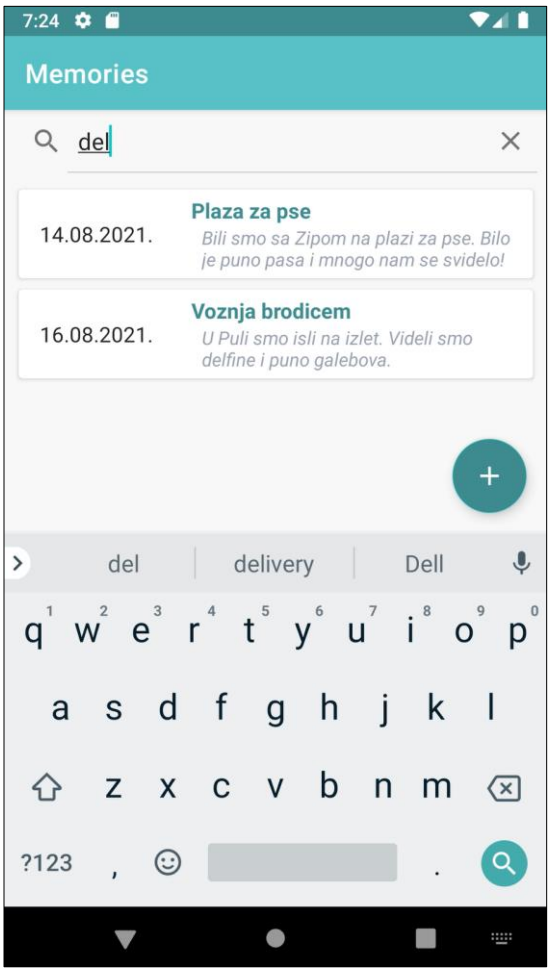
4. **Brisanje uspomena** – slično kao što se prstom prevlači u desno, prevlačenjem prstom na levo (slika 7.) obrisaće se postojeća uspomena iz baze podataka, a samim tim nestaće i sa glavne *MainActivity* stranice.

5. **Pretraga uspomena po ključnoj reči** – pri vrhu glavne stranice *MainActivity* nalazi se polje za pretragu (slika 8.) gde se može vršiti pretraga uspomena po bilo kojoj ključnoj reči (slika 9.), bilo da se ona sadrži u imenu uspomene, opisu, lokaciji ili datumu.

Slika 6. *Demonstracija akcije za izmenu uspomene*

Slika 7. *Demonstracija akcije za brisanje uspomene*

Slika 8. *Polje za pretragu uspomena po ključnoj reči*



Slika 9. *Demonstracija pretrage uspomena po ključnoj reči*

6.3 Demonstracija korišćenja nekih od Kotlinovih karakteristika

U ovom poglavlju prikazana su neka od Kotlinovih karakteristika korišćena u razvijenoj aplikaciji. Ovi primeri bolje prikazuju Kotlin na delu, na realističnim situacijama.

U poglavlju 4.4 je bilo reči o kasnoj inicijalizaciji promenljivih. U klasi *AddMemoryActivity* se ova funkcionalnost koristi za definisanje uvezivanja promenljivih koje će biti dostupno tek nakon što se stranica učita i prikaže na korisničkom interfejsu.

ISEČAK IZ KODA 1

```
    private lateinit var binding: ActivityAddMemoryBinding
```

U narednom isečku iz koda je prikazana interpolacija stringova pri pisanju SQL naredbe. Više o ovoj temi se nalazi u poglavlju 4.2, gde je obrađena klasa String. Pristup kreiranja naredbe je kombinovano interpolacija stringova, ali i konkatenacija stringova, te se može primetiti kako je rezultat vrlo čitak.

ISEČAK IZ KODA 2

Kotlin

```
val createMemoriesTable = ("CREATE TABLE $TABLE_MEMORIES "
        + "($KEY_ID INTEGER PRIMARY KEY,"
        + "$KEY_NAME TEXT,"
        + "$KEY_DESCRIPTION TEXT,"
        + "$KEY_DATE TEXT,"
        + "$KEY_LOCATION TEXT)")
```

Kada su pominjane promenljive koje mogu imati i prazne vrednosti u poglavlju 3.3 predstavljen je rad sa njima. U sledećem isečku iz koda, klase *AddMemoryActivity*, vidi se kako se to primenjuje u praksi.

ISEČAK IZ KODA 3

```
if (mMemoryDetails != null) {
    supportActionBar?.title = "Edit Memory"

    binding.etName.setText(mMemoryDetails!!.name)
    binding.etDescription.setText(mMemoryDetails!!.description)
    binding.etDate.setText(mMemoryDetails!!.date)
    binding.etLocation.setText(mMemoryDetails!!.location)

    binding.btnSave.text =
        resources.getText(R.string.edit_new_memory)
}
```

Pošto je uslov naredbe grananja da promenljiva *mMemoryDetails* nije prazna vrednost u telu naredbe njeni atributi se mogu koristiti bez bojazni da će doći do NPE izuzetka. Takođe kompajleru se simbolom !! daje do znanja da se atributima promenljive koja može sadržati praznu vrednost pristupa svesno i sa namerom.

U poglavlju 4.1.5 pregledano je detaljno korišćenje klasa podataka, a naredni isečak iz koda demonstrira njihovo korišćenje u praksi. I ako je ovo sve što je definsano za klasu *MemoryModel* može se primetiti da je kasnije u kodu korišćen i njen konstruktor (*AddMemoryActivity*) koji nigde nije definisan. Pored njega mogle su biti korišćene i sve ostale metode koje Kotlin generiše npr. *toString(), equals(), componentN()...*

ISEČAK IZ KODA 4

```
data class MemoryModel(
    val id: Int,
    val name: String?,
    val description: String?,
    val date: String?,
    val location: String?
)
```

U poglavlju 4.1.6 je bilo priče o tzv. pratećim objektima klase. Naredni isečak iz koda prikazuje njegovo korišćenje u praksi. U pratećem objeku klase *DatabaseHandler* inicijalizovane su konstante koje su vezane za bazu podataka (njena verzija, ime, tabela, kolone). One su definisane tu da bi bile dostupne statički, bez potrebe da se klasa instancira.

ISEČAK IZ KODA 5

```
class DatabaseHandler(context: Context) :
    SQLiteOpenHelper(context, DATABASE_NAME, null, DATABASE_VERSION) {
        companion object {
            private const val DATABASE_VERSION = 1
            private const val DATABASE_NAME = "MemoriesDatabase"
            private const val TABLE_MEMORIES = "MemoriesTable"

            // Memories table columns
            private const val KEY_ID = "_id"
            private const val KEY_NAME = "name"
            private const val KEY_DESCRIPTION = "description"
            private const val KEY_DATE = "date"
            private const val KEY_LOCATION = "location"
}
```

U narednom isečku iz koda prikazano je korišćenje lambda funkcija, detaljnije opisanim u poglavlju 5.5. Interfejs *OnDateSetListener* je implementiran kao anonimna klasa, odnosno lambda funkcija metode koja mora biti prevaziđena. Funkcija koja treba da se prevaziđe prima parametre *view*, *year*, *month* i *dateOfMonth* pri čemu je prosleđivanje atributa *view* preskočeno, jer se neće ni koristiti u telu funkcije.

ISEČAK IZ KODA 6

```
dateSetListener = DatePickerDialog.OnDateSetListener {
        _, year, month, dayOfMonth ->
                cal.set(Calendar.YEAR, year)
                cal.set(Calendar.MONTH, month)
                cal.set(Calendar.DAY_OF_MONTH, dayOfMonth)
                updateDateInView()
}
```

Ovim je prikazan deo moći koji Kotlin pruža u funkcionalnoj paradigmi.

7. Zaključak

Danas, u izobilju programskih jezika, postoji mnogo načina da se postigne bilo koji cilj. Ono što je teško je odabrati jezik i zajednicu koji će najbolje i najbrže približiti taj zacrtani cilj. Kotlin, koji je relativno nov jezik, kroz vrlo kratak vremnski period je uspeo da zauzme značajno mesto u programerskoj zajednici zahvaljući svojim pogodnim karakteristikama. Najznačajnije od njih su opisane u ovom radu: konciznost u pisanju koda i njegova čitkost, bezbednost napisanog koda, kao i interoperabilnost sa Java kodom. Interoperabilnost sa Java kodom je bila "odskočna daska" za Kotlin zbog mnogih već postojećih biblioteka i projekata napisanih u Javi.

Kotlin iz dana u dan postaje sve popularniji jezik i njegovi osnivači se trude da se vode istim principima dizajna kao i pri njegovom stvaranju:

1. održavanje jezika modernim – redovno unapređivanje već postojećih karakteristika radi izbegavanja zastarelosti koda (*eng. legacy code*)
2. održavanje povratne informacije zajednice – osluškivanje korisnika Kotlina radi dodavanja/izbacivanja karakteristika koda
3. lak prelazak na nove verzije jezika – uz česte promene i unapređenja jezika neophodno je obezbediti i lak prelazak na nove verzije

Zbog svih ovih karakteristika Kotlin ostaje sveprisutan u raznim ograncima IT industrije, a podrškom kompanije Google 2019. godine osigurava svoje mesto u Android zajednici i sve su naznake da će tu i ostati.

8. Korišćeni izvori

1. JetBrains. *Official documentation of Kotlin language*, 2021.
   URL: https://kotlinlang.org/
2. Marcin Moskala. *Effective Kotlin*. Self published, 2019.
3. Dmitry Jemerov, Svetlana Isakova. *Kotlin in Action*. Manning, 2017.
4. Venkat Subramaniam. *Programming Kotlin*. Pragmatic Bookshelf, 2019.
5. Bruce Eckel, Svetlana Isakova. *Atomic Kotlin.* Mindview LLC, 2021.
6. Ken Kousen. *Kotlin Cookbook*. O'Reilly, 2019.
7. Dawn Griffiths, Davod Griffiths. *Head First Kotlin*. O'Reilly, 2019.
8. *Baeldung. Java ecosystem guides and courses* 2021.
   URL: https://www.baeldung.com/kotlin