Razvoj i testiranje mobilnih aplikacija pisanih u programskom jeziku Kotlin

Grbavac, Vedran

Master's thesis / Diplomski rad

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:630519

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Vedran Grbavac

Razvoj i testiranje mobilnih aplikacija pisanih u programskog jeziku Kotlin

DIPLOMSKI RAD

Varaždin, 2020.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Vedran Grbavac

Matični broj: 44052-15/R

Studij: Informacijsko i programsko inženjerstvo

Razvoj i testiranje mobilnih aplikacija pisanih u programskog jeziku Kotlin

DIPLOMSKI RAD

Mentor:

Doc. dr. sc. Zlatko Stapić

Varaždin, rujan 2020.

Vedran Grbavac

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Ovaj rad se bavi razvojem i testiranjem mobilnih aplikacija pisanih u programskom jeziku Kotlin. Rad je podijeljen u nekoliko velikih cjelina, a započinjemo za predstavljanjem Kotlina kao programskog jezika, objašnjavajući zašto bi ga trebali koristiti te koje su njegove prednosti i mane. Nastavljamo sa objašnjavanjem njegovih osnova kao što su funkcije, varijable, klase, svojstva, tokovi podataka, itd. Osim toga, započinjemo pisati stvarni programski kôd, odnosno, učimo napisati poznatu uzrečicu „Hello world“ u Kotlinu. Potom ulazimo u dubinu ovog programskog jezika te se upoznajemo sa naprednim alatima i tehnikama koje se koriste u današnjem svijetu Android programiranja. Neke od njih su: MVVM, LiveData, Data Binding, Room, Coroutines, itd. Nakon toga slijedi testiranje programskog kôda pisanog u Kotlinu. U ovom dijelu rada objašnjava se što je jedinični test, instrumentacijski test te što je to automatizirano testiranje. Također, objašnjava se zašto bi trebalo koristiti testiranja te kada koristimo navedene testove i koje su razlike između njih. Kraj rada sastoji se od praktičnog dijela u kojem razvijamo i testiramo stvarno aplikaciju koristeći sve što smo prethodno objasnili u teoretskom dijelu rada. Kroz cijeli rad možemo pronaći sve tvrdnje potkrijepljene sa stvarnim primjerima programskog kôda.

Ključne riječi: Android; Kotlin; Razvoj; Testiranje; Aplikacija; Programski jezik; Appium; Android Studio

Sadržaj

1. Uvod

U ovom radu ćemo se upoznati sa novim načinom programiranja za Android mobilne aplikacije u Kotlin programskom jeziku te uz pomoć njega naučiti nove vrijednosti u svijetu Androida i to primijeniti nad svojim stvarnim projektima. Za čitanje i razumijevanje ovog rada nisu potrebna prethodna iskustva i znanja u razvoju aplikacija za Android mobilne uređaje, no ukoliko ih imate oni će vam uvelike pomoći kako bi lakše i brže shvatili tekst ovog rada.

Mnogi autori danas Kotlin stavljaju ispred Jave kada se radi o razvoju mobilnih aplikacija, a razlog tome su bezbrojne značajke Kotlina koje omogućuju sažetiji i razumljiviji kôd od Jave, bez žrtvovanja performansi ili sigurnosti. Također, Kotlin dolazi iz industrije, a ne akademske zajednice te rješava probleme s kojima se danas suočavaju programeri.

Ovaj rad je raspodijeljen na dva velika dijela, a to su: teoretski dio i praktični dio. Teoretski dio započinje sa nekoliko poglavlja koja opisuju Kotlin kao programski jezik te nam oni služe kao motivacija za daljnje korištenje Kotlina u razvoju naših mobilnih aplikacija. Nastavlja se sa osnovama programiranja na apstraktnoj razini te se potom polako ulazi u stavke specifične za Kotlin. Objašnjavaju se razlike između Kotlina i ostalih jezika te se ukazuje na benefite koje Kotlin donosi te razloge zašto bi baš njega trebali koristiti u današnje vrijeme. Nakon toga, naučiti ćemo napisati našu prvu liniju u Kotlinu te potom nastaviti sa slaganjem našeg znanja kako bi mogli izraditi složenije i kompliciranije aplikacije. I na kraju teoretskog dijela, obraditi ćemo napredne tehnike i alate koje se koriste u današnje vrijeme razvoja i testiranja aplikacija za Android operacijski sustav. Praktični dio rada pratiti će sve ono što je napisano u teoretskom. Pokazati ćemo sve od najosnovnijih osnova kao na primjer izrada projekta unutar *Android Studia* pa sve do toga da ćemo napraviti integraciju *Android Studia* sa aplikacijom *Appium* te ćemo izvoditi automatizirane testove nad aplikacijom koja će prethodno biti razvijena. Cijeli praktični dio biti će javno dostupan na autorovom *GitHub* profilu te će ga svatko moći pogledati u svako vrijeme.

Nadam se da će ovaj rad doprijeti do velike publike koja je zainteresirana za ove temu te da će se kroz godine usavršavati kako bi dao najbolji primjer teorije i prakse koji je zastupljen u svijetu razvoja i testiranja aplikacija za Android mobilne uređaje.

2. Što je Kotlin?

Prije nego što počnemo sa samim značajkama ovog programskog jezika, upoznajmo se malo sa pozadinom njega te kako je uopće nastao i postao tako popularan u današnjem svijetu Android programiranja.

U današnjem svijetu stvari se rapidno mijenjaju za Android programere, ali uvijek idu na bolje. Jedna od tih bolji promjena je bila i ta da je 2017. Googleov Android team proglasio da je Kotlin od tada njihov službeni jezik za razvoj Android mobilnih aplikacija. To znači da je i dalje moguće razvijati aplikacije uz pomoć programskog jezika Java, ali od tada Google se brine da Kotlin bude potpuno podržan i da sve nove značajke (eng. features), okviri (eng. frameworks), integrirana razvojna sučelja (eng. IDE) i sve biblioteke rade besprijekorno sa novim jezikom. Ljudi su godinama tažili od Googlea da prizna Kotlin kao njihov službeni jezik i na kraju ih je Google ipak poslušao.

Kotlin je JVM baziran jezik koji je razvijen od strane JetBrainsa[1], tvrtke poznate po razvoju moćnih integriranih razvojnih sučelja za programski jezik Java pod nazivom „IntelliJ IDEA“. Android Studio, službeno integrirano razvojno sučelje za razvoj Android aplikacija je bazirano na IntelliJ.

Kotlin je napravljen tako da se uvijek razmišljalo kako olakšati Java programerima, a prema A. Levia dvije najbitnije značajke Kotlin programskog jezika su:

- Veoma je intuitivan i jednostavan za naučit ukoliko imate iskustva u Javi – veliki dio ovog jezika je sličan onome što već od prije znamo, razlike u osnovnim konceptima se nauče u vrlo kratkom roku.
- Kotlin sadrži totalnu integraciju sa integriranim razvojnim sučeljem za besplatno – Android studio može razumjeti, kompilirati te pokrenuti Kotlin kôd. Podrška za ovaj jezik dolazi iz tvrtke koja razvija integrirana razvojna sučelja, pa možemo reći da su Android programeri građani prve klase.

[1] Službena stranica gdje možete pročitati više o njima: https://www.jetbrains.com/

2.1. Razlike između Kotlina i Jave 6

Za ovo poglavlje i ovu usporedbu pripremljena je tablica kako bi se lakše moglo razumjeti razlike između navedenih programskih jezika. Prema Leviai napravljena je tablica u kojoj su navedene prednosti Kotlina nad Javom te objašnjenja iz kojeg razloga je to prednost (Levia, 2017).

Tablica 1: Razlika između Kotlina i Jave 6

Prednost	Razlog zašto je to prednosti
Ekspresivnost	Jedna od najvažnijih prednosti i kvaliteta ovog jezika. Možemo pisati puno više stvari sa puno manje kôda što u krajnosti dovodi i do lakšeg čitanja, testiranja, shvaćanja tuđeg kôda, itd.
Sigurnost	Puno ljudi će reći da je Kotlin „null safe“ što znači da se sa mogućim *null*[2] situacijama možemo boriti u vrijeme kompiliranja da bi spriječili iznimke u vrijeme izvršenja kôda. Moramo eksplicitno specificirati da objekt može biti *null* i onda provjeriti njegovu ništavosti prije njegova korištenja.
Funkcionalnost	Kotlin je u osnovi objektno orijentiran jezik, a ne čisti funkcionalni. Međutim, kao i u mnogim drugim jezicima, koristi se mnogim konceptima iz funkcionalnog programiranja, poput lambda izraza, da bi neke probleme riješio na puno lakši način.
Koristi funkcije proširenja	Ovo znači da možemo proširiti bilo koju klasu sa novim značajkama, čak i ako nemamo pristup izvornom kôdu.
Vrlo je interoperabilan	Možemo nastaviti koristiti većinu biblioteka i kôdova napisanih u Javi jer je interoperabilnosti između ova dva jezika izvrsna. Čak je moguće i kreirati mješovite projekte, s datotekama Kotlina i Jave istodobno.

[2] Što zapravo „null“ znači pročitajte na: https://www.merriam-webster.com/dictionary/null

Iz tablice se može izvući neke od značajki jezika koje ćemo u nastavku još dodatno i detaljnije objasniti, ova tablica je služila samo da bi se uvidjele razlike, odnosno, prednosti Kotlina nad Javom.

2.2. Što dobijemo s Kotlinom?

Bez da ulazimo preduboko u srž Kotlina jer ćemo do toga doći u nastavku rada, volio bih samo izdvojiti neke interesantne značajke Kotlina koje nam definitivno nedostaju u Javi.

2.2.1. Ekspresivnost

Sa Kotlinom puno je jednostavnije izbjeći tzv. *n* kôd(eng. boilerplate code) [3] zbog toga što su najčešći obrasci pokriveni zadanim jezikom. Također, omogućuje nam da se usredotočimo na izražavanje svojih ideja što utječe na pisanje manje kôda. Manje napisanog kôda također znači manje kôda za održavanje i testiranje. Pokažimo to na primjeru kako bi lakše shvatili o čemu se priča. Primjer iz (Levia, 2017) gdje u Javi želimo stvoriti jedan model, točnije jednu podatkovnu klasu, to bi morali napraviti na sljedeći način. Klasa je naziva „Person“, a sadrži atribute ime, prezime, broj godina te broj osobne iskaznice.

```
public class Person {
    private String firstName;
    private String lastName;
    private int age;
    private int numberOfId;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public int getAge() {
        return age;
```

[3] Više pročitajte na: https://en.wikipedia.org/wiki/Boilerplate_code

```
    }

    public void setAge(int age) {
        this.age = age;
    }

    public int getNumberOfId() {
        return numberOfId;
    }

    public void setNumberOfId(int numberOfId) {
        this.numberOfId = numberOfId;
    }
}
```

Isječak kôda 1: Klasa "Person" u Javi

Kada bi tu istu klasu pisali u Kotlin programskom jeziku, ona bi izgledala ovako.

```
data class PersonKotlin(
    val firstName: String,
    val lastName: String,
    val age: Int,
    val numberOfId: Int
)
```

Isječak kôda 2: Klasa "Person" u Kotlinu

Ovakva klasa podataka automatski generira sva polja te sve pristupnike entitetima, kao i neke korisne metoda kao što je *toString*(), *equals*() i *hashCode*().

2.2.2. Null Safety

Prema A. Levia, kada koristimo Javu velika količina našeg kôda je obrambena, točnije, moramo jednom provjeriti dali je nešto *null* prije nego što uopće to počnemo koristiti kako ne bi dobili neočekivani *NullPointerException*. Tipovi *@Nullable* i *@NotNull* su u Kotlinovu tipu sustava kako bi nam pomogli da izbjegnete *NullPointerException* te je Kotlin, kao i mnogi drugi moderni jezici, *null* siguran, zbog toga što sam tip objekta jasno definira može li on biti *null* korištenjem operatora sigurnog poziva koji se poziva tako da na kraju tipa objekta dodamo znak „?“. Primjer je recimo `val firstName:String?` gdje *String?* znači da je objekt tipa *String* te da može biti ništavan pa zbog toga možemo raditi stvari poput sljedeće: `var notNullPerson : PersonKotlin = null`. Ovo se neće kompilirati zbog toga što „PersonKotlin“ ne može biti *null*. Dok u ovom primjeru: `var person : PersonKotlin? = null,` „PersonKotlin“ može biti *null*. Razlika je u znaku „?“ na kraju tipa podataka.

Sljedeći primjer, person.print() se također neće kompilirati jer „PersonKotlin“ može biti *null* pa se s time moramo najprije suočiti pa potom zvati preostale metode. Jer ukoliko da pozovemo metodu „print()“, a objekt je *null*, aplikacije neće imati što ispisati te će baciti NullPointerException. Dok će redak person?.print() ispisati objekt „PersonKotlin“ samo ukoliko nije *null*. Ne trebamo koristiti operator sigurnog poziva ukoliko smo već prethodno provjerili njegovu ništavost, pogledajmo sljedeći primjer.

```
if(person != null){
    person.print()
}
```

Isječak kôda 3: Provjera ništavosti u Kotlinu

Sljedeću liniju kôda `person!!.print()` trebamo koristiti samo kada smo sigurno da objekt nije ništava, u suprotnom će baciti iznimku.
U Kotlinu još postoji tzv. Elvis operator koji će nam dati alternativu ukoliko je objekt ništavan.

```
val name = person?.firstName ?: "No name"
```

Isječak kôda 4: Elvis operator

2.2.3.Ekstenzije

U Kotlinu možemo dodati nove funkcije u bilo koju klasu. To je mnogo čitljiviji nadomjestak za uobičajene korisne klase koje svi mi imamo u našim projektima. Pokažimo opet kako to izgleda na primjeru. Možemo recimo dodati novu metodu fragmentima kako bi prikazali „toast“ poruku.

```
fun Fragment.toast(message: CharSequence,
                   duration: Int = Toast.LENGTH_SHORT) {
    Toast.makeText(getActivity(), message, duration).show()
}
```

Isječak kôda 5: Ekstenzija na metodu „toast“

I onda možemo ispisati poruku na sljedeći način:

```
fragment.toast("Hello world!")
```

Isječak kôda 6: Poziv ekstenzije

2.2.4. Lambde

Što kada bi rekli da umjesto da svaki put moramo dekalirati anonimnu klasu svaki puta kada trebamo implementirati slušatelj klikova (eng. Click listener), možemo samo definirati što želimo raditi. Navedenu stavku i još mnogo toga dobivamo zahvaljujući lambdama. Sljedeći primjer pokazuje to:

```
view.setOnClickListener { toast("Hello world!") }
```

Isječak kôda 7: Primjer lambde u Koltinu

Ovo je samo mali dio toga što Kotlin može napraviti za nas kako bi pojednostavili kôd. Sada kada smo se upoznali sa nekim od brojnih zanimljivih značajki jezika, nastaviti ćemo u tom smjeru te krenuti čitati ovaj rad do kraja i usput primjenjivati sve što nađemo u njemu kako bi u budućnosti napredovali u svakom segmentu modernog programiranja (Levia, 2017; Kotlinlang.org, 2020).

3. Osnove Kotlina

U ovom poglavlju obraditi ćemo osnove elemente Kotlina, točnije što ti elementi predstavljaju, kako ih napisati, što rade itd. Neke od osnovnih elementa koje ćemo proći su:

- Funkcije
- Varijable
- Klase
- Tokovi podataka
- Iznimke
- Rad s podacima

Uz to, upoznati ćemo se sa konceptima svojstava u Kotlinu, obraditi ćemo korištenje raznih upravljačkih struktura. Uglavnom, većina ovih stvari slična je onima u Javi, ali su značajno poboljšane. Na kraju ovog poglavlja, moći ćemo zasigurno koristiti osnove jezika Kotlin kako bi napisali neki osnovni kôd koji će sigurno raditi, no možda neće biti najbolja izvedba istog.

3.1. Funkcije i varijable

Ove dvije stavke nabrojene u naslovu su osnovni elementi koji čine svaki Kotlin program. Kroz ovo poglavlje upoznati ćemo se kako Kotlin omogućuje izostavljanje mnogih vrsta deklaracija i kako nas potiče da koristimo nepromjenjive(eng. immutable), a ne promjenjive podatke(eng. mutable)

3.1.1. „Hello world“

Započnimo sa najklasičnijim primjerom u svijetu programiranja, a to je program koji ispisuje „Hello World“. U Kotlinu dovoljna je samo jedna funkcija kako bi to ispisali. Pogledajmo u sljedećem primjeru kako to izgleda.

```
fun main(args: Array<String>){
    println("Hello, world!")
}
```

Isječak kôda 8: Ispis "Hello World" teksta

Što sve možemo uočiti iz ovog kratkog kôda su njegove značajke i dijelove sintakse jezika. Prema D.Jemerov i S.Isakova širi popis toga što se može vidjeti proćitajmo u nastavku:

- Ključna riječ *fun* koristi se kako bi deklarirali funkcije
- Tip parametra se piše nakon njegovog naziva, također to se odnosi i na deklaracije varijable, ali o tome više u nastavku
- Funkcija se može deklarirati na gornjoj razini datoteke, ne mora biti unutar klase
- Polja su samo klase, za razliku od Jave, Kotlin nema posebnu sintaksu za deklariranje tipova polja
- Umjesto *System.out.println* upisujemo samo *println* jer Kotlinova standardna biblioteka nudi mnogo omotača (eng. wrapper) oko standardnih funkcija iz Java biblioteke sa veoma sažetom sintaksom
- Točka-zarez možemo izostaviti s kraja retka, kao i u mnogim drugim modernim jezicima

3.1.2. Funkcije

U prethodnom poglavlju mogli smo vidjeti kako deklarirati funkciju koja ne vraća ništa. Sigurno se sada postavlja pitanje, gdje bi se trebao staviti tip povratnog parametra. Možemo pogađati, ali otkriti ćemo da ide nakon ulaznih parametara te je odvojen dvotočkom. Ovako to izgleda za funkciju koja nam vraća veći rezultat od dva unesena:

```
fun max(a: Int, b: Int): Int {
    return if (a > b) a else b
}
```

Isječak kôda 9: Funkcija koja vraća veći broj

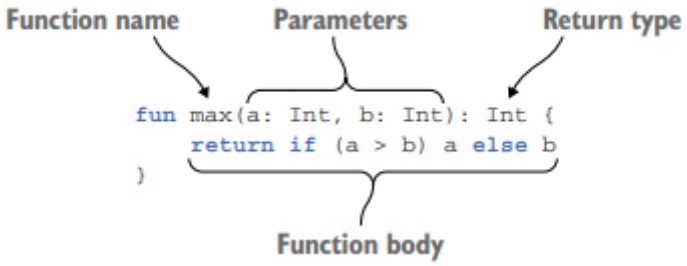
Kada bi željeli pozvati tu funkciju, odnosno, kada bi željeli ispisati njezin rezultat, to bi onda izgledalo ovako:

```
println(max(1, 2))
```

Isječak kôda 10: Ispis funkcije

Deklaracija funkcije počinje sa ključnom riječi *fun*, a zatim slijedi njezin naziv, u našem slučaju „max“. Nakon toga dolazi popis ulaznih parametara unutar oblih zagrada te na

kraju dolazi vrsta povratnog parametra odvojen dvotočkom, u našem slučaju, to je *Int*. Na sljedećoj slici, preuzetoj iz (Jemerov i Isakova, 2017), su prikazani navedeni koncepti koji mogu poslužiti kao podsjetnik.



Slika 1: Deklaracija funkcije

Prethodnu funkcije mogli smo pojednostaviti još više. Zato što se tijelo funkcije sastoji od jednog izraza možemo ga koristiti kao cijelo tijelo funkcije, uklanjajući vitičaste zagrade i povratnu izjavu, ta ista funkcija sada izgleda ovako:
`fun max(a: Int, b: Int): Int = if (a > b) a else b`. Ako je tijelo funkcije napisano unutar vitičastih zagrada kažemo da ta funkcija ima blok tijelo(eng, blok body), a ako vraća izraz izravno nazad onda kažemo da ima izrazno tijelo(eng. expression body). U IntelliJ IDEA platformama imamo mogućnost da nam ona napravi pretvorbu između ta dva stila, a funkcije glase: „Pretvori u izrazno tijelo“(eng. Convert to expression body) i „Pretvori u tijelo bloka“(eng. Convert to block body). Funkcije sa izraznim tijelo se često mogu pronaći u Kotlinovom kôdu. Ovaj stil koristi se ne samo za trivijalne jedno-linijske funkcije, već i za funkcije koje predstavljaju jedan složeniji izraz, kao što su *if*, *when* ili *try*, o tim izrazima detaljnije u nastavku poglavlja. Ona našu prvotnu funkciju za vraćanje maksimuma možemo još više pojednostaviti te izostaviti vrstu povratka te ona tada izgleda ovako: `fun max(a: Int, b: Int) = if (a > b) a else b`.

Zašto uopće postoje funkcije bez deklaracije povratnog tipa? Ne zahtjeva li Kotlin kao jezik statički tip da svaki izraz ima tip u vrijeme kompiliranja? Doista i je tako, jer svaka varijabla i svaki izraz ima tip dok svaka funkcije ima povratni tip. Ali za funkcije koje imaju izrazno tijelo, kompajler može analizirati izraz koji koristi kao tijelo funkcije te iskoristi njen tip kao povratni tip funkcije, čak i kada nije izričito tako napisano. Napomena da izostavljanje povratnog tipa dopušteno samo za funkcije koje imaju izrazno tijelo. Za funkcije koje imaju blok tijelo te koje vraćaju neku vrijednost, moramo

navesti vrstu povratnog tipa te eksplicitno napisati povratne izjave. Funkcije u stvarnom svijetu često su komplicirane i duge te mogu sadržavati i nekoliko povratnih izjava. Imajući povratni tip i eksplicitno napisanu povratu izjavu pomaže vam da brzo shvatite što se točno može vratiti tom funkcijom (Jemerov i Isakova, 2017; Kotlinlang.org, 2020).

3.1.3. Varijable

U Javi kada deklariramo varijablu, nju započinjemo pisati sa njezinim tipom, no to ne bi uspjelo u Kotlinu. Kotlin omogućuje izostavljanje vrsta iz mnogih varijabli deklaracije. Tako u Kotlinu započinjemo varijablu sa ključnom riječju, a možemo ili ne moramo unijeti vrstu nakon naziva varijable. Ključne riječi za varijable su „val“ ili „var“, no o tome više u nastavku. Uzmimo primjer dvije varijable kako bismo lakše shvatili.

```
val question = "Koliko zgrada ima FOI?"
val answer = 2
```

Isječak kôda 11: Deklariranje varijabli bez unesenog tipa

Gornji primjer izostavlja deklaraciju tipa, no ukoliko mi to želimo eksplicitno naglasiti onda bi napisali ovako:

```
val answer: Int = 2
```

Isječak kôda 12: Deklariranje varijabli sa tipom

Baš kao i kod funkcija s izraznim tijelom, ako ne odredimo vrstu varijable, kompajler će analizirati inicijalizatorski izraz te koristiti njegov tip kao tip varijable. U našem slučaju, inicijalizator(2), ima *Int* tip, pa će varijabla imati isti tip. Ako koristimo konstantu s pomičnom točkom onda će varijabla imati tip *Double*.

```
val doubleNumber = 4.6e6
```

Isječak kôda 13: Deklariranje varijabli sa tipom i početnim stanjem

Ako varijabla nema inicijalizator, morate izričito navesti njen tip:

```
val answer: Int
answer = 2
```

Isječak kôda 14: Varijabla koja nema inicijalizator

Kompajler ne može zaključiti vrstu varijable ukoliko ne damo podatke o vrijednostima koje mogu biti dodijeljene ovoj varijabli (Jemerov i Isakova, 2017; Kotlinglang.org, 2020).

Kao što je već napomenuto u prethodnom dijelu poglavlja, postoje dvije ključne riječi za deklariranje varijabli:

- Val (dolazi od engleske riječi *Value* što znači vrijednost) – ona je nepromjenjiva referenca (eng. immutable reference). Varijabla deklarirana s ovom ključnom riječju ne može biti promijenjena nakon što je prvi put inicijalizirana. Odgovara konačnoj (eng. final) varijabli u Javi.
- Var (dolazi on engleske riječi *Variable* što znači varijabla) – ona je promjenjiva referenca (eng. mutable reference). Vrijednosti ove varijable mogu biti promijenjene nakon prvotne inicijalizacije. Ova deklaracija odgovara uobičajenoj (eng. non-final) Java varijabli

Prema uobičajenim načinom programiranja, trebali bi nastojati proglasiti sve varijable u Kotlinu pomoću ključne riječi val. U tome nam uvelike pomaže IntellJ IDEA platforme koje nam uvijek nude mogućnosti da promjenjive varijable promijenimo u nepromjenjive ukoliko se one nigdje u kôdu ne mijenjaju. Ključnu riječ var koristimo samo kada je to nužno. Pomoću nepromjenjivih referenci, nepromjenjivih objekta te funkcija bez nuspojava naš kôd je bliže funkcionalnom stilu. Val varijabla se mora inicijalizirati točno jednom tijekom izvođenja bloka gdje je definirana, ali ju možemo inicijalizirati s različitim vrijednostima, ovisno o uvjetima, ako kompajler može osigurati da će samo jedna od inicijalizatorskih izjava biti izvršena:

```
val message: String
if (someBooleanFunction()) {
    message = "Uspješno!"
// ... Obavi radnju
    }
else {
    message = "Neuspješno!"
    }
}
```

Isječak kôda 15: Inicijalizacija *val* varijable kroz *if* grananje

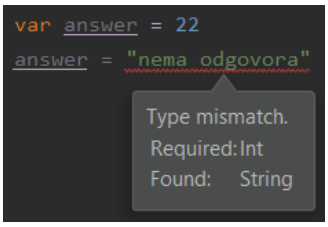
Imajmo na umu da, iako je val referenca sama po sebi nepromjenjiva, objekt na koji ukazuje može biti izmjenjiv. Na primjer sljedeći kôd je savršeno validan:

```
val subjects = arrayListOf("Ekonomija", "Organizacija")
subjects.add("Matematika")
```

Iako ključna riječ var omogućuje varijabli da promijeni svoju vrijednost, njezin tip je fiksiran. Sljedeći kôd nećemo moći kompilirati:

```
var answer = 22
answer = "Nema odgovora"
```

Došlo je do pogreške jer njegova vrsta je *String*, a on nije dobio tip koji je očekivao nego je dobio tip *Int*, slika 2.



Slika 2: Krivi tip varijable

Kompajler zaključuje vrstu varijable samo iz inicijalizatora i ne uzima naknadne dodjele u obzir prilikom određivanja vrste. Ako trebate pohraniti vrijednost nepodudarne vrste u varijablu, morate ručno pretvoriti ili prisiliti vrijednost u pravi tip. Više o tome ćemo poslije ali pokazati ćemo samo kratki primjer kako bi to trebalo izgledati.

```
var answer = "Nema odgovora"
answer = 42.toString()
```

Ovo poglavlje pomoglo nam je da shvatimo neke od osnovnih elemenata Kotlina te da lakše obradimo poglavlja koja slijede (Jemerov i Isakova, 2017).

3.2. Osnovno o klasama i svojstva

Klase u Kotlinu slijede veoma jednostavnu strukturu. Međutim, postoje neke sitnice koje se razlikuju od Jave s kojima ćemo se upoznati prije nego što nastavimo sa ovim poglavljem. Kotlinovi koncepti klasa će sigurno biti poznati, ali otkriti ćemo da se mnoštvo uobičajenih zadataka može ostvariti sa puno manje kôda.

3.2.1. Deklariranje klasa

Ako želimo deklarirati klasu, sve što trebamo učiniti je koristiti ključnu riječ *class*, a to izgleda ovako:

```
class MyFirstClass {
   //… kôd klase
}
```

Isječak kôda 16: Deklaracija klase

U Javi tijelo konstruktora često sadrži kôd koji se u potpunosti ponavlja (eng. boilerplate), točnije, pridružujemo parametre poljima sa korespondirajućim imenima. U Kotlinu, ova logika se može napisati sa puno manje ponavljajućeg kôda. Klase imaju jedinstveni zadani konstruktor, a u nastavku ćemo vidjeti da možemo dodati sekundarni konstruktor, ali u većini slučajeva dovoljan nam je samo ovaj zadani. Parametri se pišu odmah nakon naziva. Vitičaste zagrade nisu potrebne ukoliko klasa nema sadržaj pa to onda izgleda ovako: `class Person(name: String, surname: String)`. Sigurno se sada pitate gdje je tijelo konstruktora, za taj problem možemo deklarirati *init* blok.

```
class Person(name: String, surname:  String){
    init {

    }
}
```

Isječak kôda 17: Deklaracije klase sa eksplicitnim pozivom konstruktora

Za razliku od Jave, Kotlinove deklaracije su konačne (eng. final) i javne (eng. public). Uz to, ugniježđene (eng. nested) klase nisu unutarnje (eng. inner) prema zadanim postavkama, ne sadrže implicitnu referencu na njihovu vanjsku klasu.

3.2.2. Nasljeđivanje klasa

Prema zadanim postavkama, klasa se uvijek nasljeđuje od *Any* (slično kao u Javi), ali nju možemo proširiti i drugim klasama. Klase su zatvorene (konačne) prema zadanim postavkama, tako da klasu možemo naslijediti samo ako je izričito deklarirana kao otvorena (eng. open) ili apstraktna (eng. abstract).

```
open class Person(name: String, surname:  String)
class Student(name: String, surname:  String,
              studentId: Int): Person(name, surname)
```

Isječak kôda 18: Deklaracija otvorene (eng. open) klase

Napomena da prilikom korištenja nomenklature pojedinog konstruktora moramo specificirati parametre koje koristimo za nadređeni konstruktor. To je ekvivalent pozivu super() u Javi.

3.2.3. Svojstva

Kao što već znamo, ideja klase je da kapsulira podatke i kôd koji djeluju nad tim podacima kao jedanu cjelinu. U Javi podaci se obično pohranjuju u polja koja su privatna (eng. private). Ako klijentima klase trebate dopustiti pristup tim podacima, pružate metode pristupnika (eng. accessor), odnosno *gettere* i *settere*[4]. Primjer toga smo mogli vidjeti u poglavlju 2.2.1. Setteri također mogu sadržavati dodatnu logiku za provjeru poslane vrijednosti, slanje obavijesti o promjeni i tako dalje. U Javi kombinaciju polja i njegovih dodataka nazivamo svojstva (eng. property), a mnogi moderni okviri (eng. frameworks) često koriste ovaj koncept. U Kotlinu svojstva (eng. property) su prvorazredna jezična značajka, koja u potpunosti zamjenjuje polja i metode pristupa. Mi deklariramo svojstva u klasama isto kao što deklariramo i varijablu, sa var ili val ključnim riječima. Svojstvo koje smo napravili s ključnom riječi val je samo za čitanje dok svojstvo napravljeno sa var je promjenjivo.

```
class Person(
// svojstvo samo za čitanje, generira polje i   trivijalni getter
    val name: String,
// promjenjivo svojstvo, generira polje, getter i setter
    var isMarried: Boolean
)
```

Isječak kôda 19: Svojstva u Kotlinu

U osnovi kada deklariramo svojstvo, zapravo deklariramo i odgovarajuće pristupnike, za val svojstva to je samo *getter*, a za var svojstvo to su *getter* i *setter*. Po zadanim postavkama, implementacija pristupnika je trivijalna, kreira se polje za pohranjivanje vrijednosti, a *getter* i *setter* vraćaju i ažuriraju svoju vrijednost. Ali ukoliko mi želimo, možemo napraviti prilagođeni pristupnik koji za računanje ili ažuriranje vrijednosti svojstva koristi drugu logiku. U gornjem primjeru vidimo sažeti prikaz deklaracije klase „Person“ koja ima istu temeljnu implementaciju kao i izvorni Java kôd, to je klasa sa privatnim poljima koja se inicijalizira u konstruktoru i može joj se pristupiti putem odgovarajućeg *gettera*. To znači da možemo koristiti ovu klasu iz Jave i iz Kotlina na isti način, neovisno o mjestu gdje je deklarirana. Uporaba izgleda identično, samo je kôd drugačiji. Kako bi izgledalo kada bi tu klasu koristili iz Jave pogledajmo u nastavku.

[4] O getterima i setterima pročitajte više na: https://www.codejava.net/coding/java-getter-and-setter-tutorial-from-basics-to-best-practices

```
/* Java */
    Person person = new Person("Danijela", true);
    System.out.println(person.getName());
    System.out.println(person.isMarried());

// Ispis
    Danijela
    true
```

Isječak kôda 20: Pozivanje svojstva u Javi

Imajmo na umu da to izgleda isto kada je klasa „Person“ definirana u Javi i u Kotlinu. Kotlinovo ime svojstva izloženo je Javi kao *getter* metoda, točnije ona bi predstavljala metodu getName(). Pravilo o imenovanju *gettera* i *settera* ima jednu iznimku, a to je ako ime svojstva započinje sa *is*, ne dodaje se dodatni prefiks za *getter*, a u *setteru* riječ *is* se zamjenjuje sa „set“ pa zbog toga u Javi pozivate isMarried(). Ako tu klasu pozivamo iz Kotlina to bi izgledalo onda ovako:

```
/* Kotlin */
val person = Person("Danijela", true);
println(person.name)
println(person.isMarried)

// Print
Danijela
true
```

Isječak kôda 21: Pozivanje svojstva u Kotlinu

Sada, umjesto da pozivamo *gettere*, pozivamo izravno referencu na svojstvo. Logika ostaje ista, ali kôd je kraći. *Setteri* promjenjivih svojstava djeluju na isti način, odnosno, kada u Javi koristimo `person.setMarried(false)` da bismo rekli da se razveo, u Kotlinu možemo samo napisati `person.isMarried = false`.

U većini slučajeva svojstvo ima odgovarajuće sigurnosno polje u kojem se pohranjuje vrijednost. Ali ako se vrijednost može izračunati usput, na primjer iz drugih svojstava, to se može izraziti pomoću prilagođenog *gettera* (Levia, 2017; Jemerov i Isakova, 2017; Kotlinglang.org, 2020).

3.3. Tok podataka i rasponi

Već smo otprije spomenuli neke od tokova podataka te ih koristili u kôd, no sada je vrijeme da i njih konačno objasnimo detaljnije. Iako je u Kotlinu obično potrebno puno manje mehanizama za kontrolu toka podataka nego u proceduralnom programiranju,

svejedno su i dalje veoma korisni. Također, postoje nove snažne ideje koje će pojedine probleme riješiti puno lakše (Levia, 2017).

3.3.1. Enum, if i when

3.3.1.1. Enum

Započnimo odmah sa jednim primjerom *Enum* klase kako bi sve ovo mogli lakše shvatiti.

```
enum class Boje {
    CRVENA, ZELENA, PLAVA, CRNA, BIJELA, SIVA
}
```

Isječak kôda 22: Definiranje enum klase

Ovo je jedan od rijetkih slučajeva gdje Kotlinova deklaracija koristi više ključnih riječi od Jave: „enum class“ u Kotlinu dok je u Javi samo „enum“. U Kotlinu, „enum“ je takozvana meka (eng. soft) ključna riječ što znači da ona ima posebno značenje kada se napiše prije klase, ali možemo ju koristiti kao obična imena na drugim mjestima. S druge strane, klasa je i dalje ključna riječ i morat ćemo nastaviti deklarirati varijable pod nazivom „aClass“ ili „myClass“. Kao i u Javi, *enumi* nisu popisi vrijednosti, odnosno, možemo deklarirati svojstva i metode nad *enum* klasama. Ovako to funkcionira:

```
// Deklarira svojstva enum klase
enum class Boje(val r: Int, val g: Int, val b: Int) {
CRVENA(255, 0, 0),        // Specificiramo vrijednosti svojstava kada
NARANCASTA(255, 165, 0), // su one kreirane
ZUTA(255, 255, 0),
ZELENA(0, 255, 0),
PLAVA(0, 0, 255);            // Ovdje moramo imati točku-zarez
// Definiramo metodu nad enum klasom
fun rgb() = (r * 256 + g) * 256 + b }
// Print
println(Boje.PLAVA.rgb())
255
```

Isječak kôda 23: Primjer enum klase

Enum konstante koriste istu sintaksu konstruktora i deklaracije svojstava kao što ste vidjeli ranije za normalne klase. Kada deklariramo da je svaki *enum* konstanta onda morate navesti vrijednost svojstava za tu konstantu. Imajmo na umu da ovaj primjer prikazuje jedino mjestu u Kotlin sintaksi gdje trebate koristiti točku-zarez, u

ostalim stvarima ona je proizvoljna. Točka-zarez u ovo slučaju odvaja popis konstanti *enuma* od definicija metoda. Za nekoliko poglavlja ćemo još prikazati niz briljantnih ideja kako možemo iskoristiti *enum* u svom kôdu kako bi ga još više poboljšali ili pojednostavili.

3.3.1.2. If

Gotovo sve u Kotlinu je izraz (eng. expression), to znači da sve vraća nekakvu vrijednost. *If* uvjeti nisu iznimka (eng. exception), više o njima u nastavku, tako učeni možemo koristi *if* izraz kao što smo i navikli, pogledajmo primjer.

```
if (x > 0) {
    toast("x je veći od 0")
} else if (x == 0) {
    toast("x je jednak 0")
} else {
    toast("x je manji od 0")
}
```

Isječak kôda 24: *If* grananje

Rezultat *if* izraza također možemo dodijeliti varijabli, već smo to prikazali u kôdu na početku, ali ponoviti ćemo još jedan primjer.

```
val res = if (x != null && x.size() >= brojDana) x else null
```

Isječak kôda 25: *If* izraz dodjeljujemo varijabli

To također podrazumijeva da nam ne treba ternarna opcija slična onoj u Javi, jer to možemo lako riješiti na sljedeći način: `val z = if (condition) x else y`

Na kraju zaključujemo da *if* izraz uvijek vraća vrijednosti. Ako jedna grana vrati „Unit()“(slično kao u Javi „void()“), čitav izraz će vratiti „Unit“, koja se može zanemariti, a u tom slučaju će raditi kao i obični *if* izraz u Javi.

3.3.1.3. When

When izraz je sličan „switch-case“ izrazu u Javi, ali daleko moćniji. Ovaj izraz će pokušati uskladiti svoj argument protiv svih mogućih grana uzastopno, dok ne pronađe onu koja je zadovoljavajuća, odnosno, onu koja ispunjava uvjete njegovog argumenta. Kada ga je pronašla, primijeniti će desnu stranu izraza, točnije, odraditi će sve ono što piše s desne strane kod argumenta koji je ispunio uvjete. Za razliku od „switch-case“ u Javi, argument ovdje može biti doslovno bilo što, a uvjeti za grane

također. Za zadane opcije možemo dodati još jednu granu koja će se izvršiti ukoliko niti jedna prethodna grana nije zadovoljila početno postavljene uvjete. Kôd koji se izvršava kada je uvjet zadovoljen također može biti blok kôd. Kako to izgleda na primjeru pogledajmo u nastavku.

```
when (x) {
    1 -> println("x == 1")
    2 -> println("x == 2")
    else -> {
        println("Ovo je blok")
        println("x nije 1 niti 2")
    }
}
```

Isječak kôda 26: *when* grananje

Iz razloga što je *when* izraz, on može dovesti i do rezultata. Uzmimo u obzir kada se koristi kao izraz, on mora pokriti sve moguće slučajeve ili implementirati *else* granu, inače se neće kompilirati.

```
val result = when (x) {
    0, 1 -> "binarno"
    else -> "error"
}
```

Isječak kôda 27: Primjer *when* grananja

Kao što možemo vidjeti, uvjet može biti skup vrijednosti odvojenih zarezima, ali i puno više od toga. Na primjer, mogli bi provjeriti vrstu argumenta i na temelju toga donijeti odluku, provjerimo primjer ispod.

```
when (view) {
    is TextView -> view.text = "Ja sam TextView"
    is EditText -> toast("EditText vrijednost: ${view.getText()}")
    is ViewGroup -> toast("Broj djece: ${view.getChildCount()} ")
    else -> view.visibility = View.GONE
}
```

Isječak kôda 28: Napredno *when* grananje

Argument je automatski pretvoren (eng. cast) u desnom dijelu uvjeta, tako da ne moramo raditi nikakvo eksplicitno pretvaranje. Moguće je provjeriti je li argument unutar raspona, ali o tome ćemo u nastavku kada budemo objašnjavali raspone. Također, možemo provjeriti čak ako se nalazi unutar kolekcije.

```
val cost = when(x) {
    in 1..10 -> "jeftino"
    in 10..100 -> "normalno"
    in 100..1000 -> "skupo"
    in specialValues -> "specijalna vrijednost!"
    else -> "nema vrijednost"
}
```

Isječak kôda 29: Definiranje varijable uz pomoć *when*

Uz sve ovo, još se možemo riješiti argumentacije i obaviti sve provjere koja nam možda zatrebaju. Lako možemo zamijeniti *if-else* izraz sa *when*, a to izgleda ovako.

```
val res = when {
    x in 1..10 -> "jeftino"
    s.contains("Bok") -> "Ovo je pozdrav!"
    v is ViewGroup -> "broj djece: ${v.getChildCount()}"
    else -> ""
}
```

Isječak kôda 30: Zamjena *if-else* sa *when*

3.3.2. Petlje

Od svih značajki koje ćemo proći u ovom radu, iteracije u Kotlinu su vjerojatno najsličnije Javi. *While* petlja je identična kao ona u Javi pa je zbog tog razloga nećemo detaljno objašnjavati nego samo ju spomenuti. *For* petlja postoji samo u jednoj jedinoj formi koja je ekvivalentna Javinoj „for-each“ petlji. Napisana je za stavke unutar elemenata (izvorno: „<item> in <elements>“) kao u C# programskom jeziku. Najčešća primjena ove petlje je za iteraciju kolekcija, baš kao i u Javi (Jemerov i Isakova, 2017).

3.3.2.1. For petlja

Iako ju vjerojatno nećemo pretjerano koristiti ako koristite funkcionalne operatore u kolekcijama ipak *for* petlja može biti korisna u nekim situacijama pa zbog toga su i dalje dostupne. Mogu raditi sa svime što ima iterator.

```
for (stavka in kolekcija) {
    println(stavka)
}
```

Isječak kôda 31: *for* petlja

Ako želimo postići regularnu iteraciju nad indeksima, možemo to učiniti pomoću raspona iako za to obično postoje pametnija i bolja rješenja.

```
for (index in 0..viewGroup.getChildCount() - 1) {
    val view = viewGroup.getChildAt(index)
    view.visibility = View.VISIBLE
}
```

Isječak kôda 32: Korištenje raspona u *for* petlji

Kada iteriramo po nizu ili polju, skup indeksa može biti zatražen od strane objekta, pa tako prethodni artefakt nije potreban.

```
for (i in polje.indices)
   print(polje[i])
```

Isječak kôda 33: *For* petlja bez artefakta

3.3.2.2. While petlja

Možemo također nastaviti koristiti i *while* petlje, ali to isto tako nije baš uobičajeno u Kotlinu jer obično postoje puno jednostavnija i vizualnija rješenja za taj problem. U Kotlinu postoje *while* i *do-while* petlje, a njihova sintaksa se ne razlikuje odgovarajućim petljama u Javi. Sintaksa im je sljedeća.

```
while (uvjet) {
    /*...*/            // izvršava se kada je uvjet istinit
}

do {
    /*...*/
// prvi put se izvršava bez obzira je li uvjet istinit ili ne
} while (uvjet)
// nakon toga se nastavlja izvršavati samo ako je istinit
```

Isječak kôda 34: *while* i *do-while* petlja

Kotlin ne donosi ništa novo u ove jednostavne petlje, tako da nećemo dužiti ovdje.

3.3.3. Rasponi

Teško je objasniti tok podataka u Kotlinu ako ne objasnimo što su rasponi. Njihov opseg je puno širi nego samo da djeluju na tok podataka. Izraz raspona koriste operator u obliku „..“(dvije uzastopne točke) koji je definiran implementacijom *rangeTo* funkcije, točnije, taj operator govori od duljini raspona, odnosno, od koliko do koliko ide

raspon. Na primjer, ukoliko bi željeli da raspon ide od 1 do 10, to bi upisali na ovaj način „1 .. 10“. Rasponi pomažu pojednostaviti naš kôd na mnogo kreativnih načina. Uzmimo sljedeći primjer i pojednostavimo ga pomoću raspona (Levia, 2017).

```
if (i >= 0 && i <= 10)
   println(i)
```

Probajmo prvo sami razmisliti kako bi ga pojednostavili, a nakon toga pogledajmo rješenje navedeno ispod.

```
if (i in 0..10)
   println(i)
```

Isječak kôda 35: Raspon u Kotlinu

Raspon je definiran bilo kojom vrstom koja se može uspoređivati, ali za numeričke tipove kompajler će ga optimizirati pretvarajući ga u analogni kôd u Javi kako bi izbjegao dodatne pretpostavke. Najjednostavnija stvar koju možemo učiniti sa brojčanim rasponima je petlja nad svim vrijednostima. Ako možemo ponoviti sve vrijednosti u nekom rasponu onda se takav raspon naziva progresija (eng. progression). Brojčani rasponi se također mogu iterirati, a petlje se također optimiziraju pretvaranje u isti bajt kôd (eng. bytecode) koji bi koristili u Javi.

```
for (i in 0..10)
   println(i)
```

Isječak kôda 36: Operator *in* unutar *for* petlje

U gornjem primjeru vidite operator *in* koji ćemo često koristiti u for petlji, ali o njemu više u zasebnom poglavlju.
Prema zadanim postavkama rasponi su inkrementalni pa možemo napraviti nešto ovako.

```
for (i in 10..0)
   println(i)
```

Isječak kôda 37: Obrnuto korištenje operatora *in*

No možemo također koristiti i funkciju *downTo*.

```
for (i in 10 downTo 0)
   println(i)
```

Isječak kôda 38: Korištenje *downTo* metode

Osim toga, možemo definirati i razmak različit od jedan između vrijednosti koristeći funkcije *step* te kombinacije više funkcija. Također, vrijednost *step* funkcije može biti i negativna ukoliko želimo ići unatrag, a ne prema naprijed.

```
for (i in 1..4 step 2) println(i)

for (i in 4 downTo 1 step 2) println(i)
```

Isječak kôda 39: Korištenje funkcije *step*

Ukoliko želite napraviti otvoreni raspon, znači onaj koji ne uključuje posljednju stavku, onda možemo koristiti funkciju *until*.

```
for (i in 0 until 4) println(i)
```

Isječak kôda 40: Otvoreni raspon

Prethodni primjer ispisati će brojeve od 0 do 3 jer će preskočiti zadnju vrijednost. Što znači da je „0 until 4“ zapravo „0..3“. Za iteracije nizova moglo bi lakše biti za razumjeti ako umjesto „for(i in 0 until list.size)“ koristimo „for(i in 0..list.size – 1)“. Kao što je već bilo spomenuto, postoje zaista kreativni načini korištenja raspona. Jedan od njih bi bio jednostavan način prikazivanja popisa pregleda unutar ViewGroup-a.

```
val views = (0 until viewGroup.childCount).map {
              viewGroup.getChildAt(it) }
```

Isječak kôda 41: Prikazivanje popisa unutar *ViewGroupe* uz pomoć raspona

Korištenje raspona i funkcionalnih operatora sprječava korištenje eksplicitne petlje za iteraciju kolekcija i kreaciju eksplicitnog niza kojem dodajemo poglede (eng. views). Sve to smo napisali u jedno retku i to doista radi (Levia, 2017).

Već smo spomenuli *in* operator u ovom poglavlju, ali u nastavku poglavlja ćemo ga malo detaljnije objasniti jer se veoma često koristi u programiranju. Pomoću operatora *in* provjeravate je li vrijednost u rasponu ili je suprotna. Također možemo koristit preinaku ovog operatora kao *!in* da biste provjerili da vrijednost nije u rasponu. U nastavku slijedi jedan primjer (Jemerov i Isakova, 2017).

```
fun jeSlovo(c: Char) = c in 'a'..'z' || c in 'A'..'Z'
fun nijeSlovo(c: Char) = c !in '0'..'9'
//Print
println(jeSlovo('q'))
true                              // vraćena vrijednost
```

```
println(nijeSlovo('x'))     // vraćena vrijednost
true
```

Isječak kôda 42: Korištenje *in* operatora

Ova tehnika provjere je li uneseni znak slovo ili nije izgleda veoma jednostavno, no nije baš tako. U pozadini ovog programa ne događa se ništa škakljivo, mi i dalje provjeravamo je li kod unesenog znaka negdje između kôda prvog i kôda posljednjeg slova. Ali ta se logika sakrila u implementaciji klasa raspona u standardnoj biblioteci.

```
c in 'a'..'z'        // pretvara se u a<=c i c<=z
```

Operatori *in* i *!in* rade također i sa *when* izrazom.

```
fun prepoznaj(c: Char) = when (c) {
    in '0'..'9' -> "Ovo je broj!"
    in 'a'..'z', in 'A'..'Z' -> "Ovo je slovo!"
    else -> "Ne znam..."
}

// print
println(recognize('8'))
It's a digit!            //vraćena vrijednost
```

Isječak kôda 43: Korištenje *in* operatora sa *when*

Raspon nije ograničeni ni na znakove također. Ako imamo bilo koju klasu koja podržava uspoređivanje instance, možemo stvoriti nizove objekata te vrste. Ako imamo takav raspon, ne možemo nabrojati (eng. enumerate) sve objekte u rasponu. Provjeriti pripada li objekt nekom rasponu možemo učiniti pomoću operatora *in* na sljedeći način `println("Kotlin" in "Java".."Scala")` te dobijemo rezultat `true`. Napomena da se znakovni nizovi (eng. Strings) ovdje uspoređuju abecednim redom jer tako klasa „String“ implementira njezino sučelje za usporedbu. Ista usporedba vrijedi i za kolekcije, a nju upisujemo ovako `println("Kotlin" in setOf("Java", "Scala"))` te dobivamo rezultat `false`.

3.4. Iznimke(eng. Exceptions)

Rukovanje iznimkama u Kotlinu slično je načinu u Javi i u mnogim drugim modernim jezicima. Funkcija se može dovršiti na uobičajen način ili baciti iznimku ako se dogodi pogreška. U Kotlinu, sve iznimke implementiraju *Throwable* sučelje, imaju

poruku te nisu označene. To znači da nismo dužni upotrebljavati „try-catch“ na nijednom od njih. To nije slučaj u Javi, gdje metode koje recimo bacaju *IOException*, trebaju biti okružene „try-catch“ blokom. Nakon mnogo godina suočavanja s njima, praksa je pokazala da ipak nisu dobra ideja. Način korištenja, odnosno, bacanja iznimki je vrlo sličan Javi, a ide ovako: `throw MyException("Neka nasumična poruka").` Također, „try-catch“ je isto identičan, no o njemu više u posebnom poglavlju.

```
try {
    // neki kôd
} catch (e: SomeException) {
    // ovdje upravljamo iznimkom
} finally {
    // opcionalan finally blok
}
```

Isječak kôda 44: *try-catch-finally* u Kotlinu

Oboje i „throw“ i „try-catch“ su izrazi u Kotlinu, što znači da se mogu dodijeliti varijabli. Kao i kod svih drugih klasa, ne moramo koristi novu ključnu riječ da bismo stvorili instancu iznimke.

```
val postotak =
    if (broj in 0..100)
        broj
    else
        throw IllegalArgumentException(
            "Postotak mora biti broj izmedu 1 i 100: $broj")
```

Isječak kôda 45: *throw* u Kotlinu

U ovom primjeru, ako je uvjet zadovoljen, program se ponaša onako kako bi i trebao, a varijabla postotka se inicijalizira brojem. U suprotnom, baca se izuzetak i varijabla se ne inicijalizira.

Kao i u Javi, koristimo „try“ zajedno sa „catch“ i „finally“ da bi upravljali iznimkama. U sljedećem primjeru možemo vidjeti kôd koji čita liniju iz dane datoteke, pokušava ju raščlaniti kao broj i vraća ili broj ili nulu ako redak nije važeći broj.

```
fun ocitajBroj(citac: BufferedReader): Int? {
     try {
         val linija = citac.readLine()
         return Integer.parseInt(linija)
        }
     catch (e: NumberFormatException) {
         return null
        }
     finally {
```

```
            citac.close()
           }
       }
// pokretanje programa
val reader = BufferedReader(StringReader("239"))

// print
println(readNumber(reader))

//rezultat
 239
```

Isječak kôda 46: Primjer funkcije koja koristi *try-catch-finally*

Najveća razlika od Jave je ta što klauzula bacanja ne postoji u kôdu, kada bismo napisali ovaj kôd u Javi, morati ćemo izričito napisati „throw IOException“ nakon deklaracije funkcije. Trebamo to učiniti jer je *IOException* označena iznimka (eng. checked exception). Označene iznimke su one koje se provjeravaju u vrijeme kompiliranja kôda, ako neki kôd unutar metode baci označenu iznimku, tada metoda mora ili obraditi iznimku ili mora navesti iznimku pomoću ključne riječi *throw*.[5] U Javi to je iznimka s kojom se treba eksplicitno postupati. Moramo deklarirati sve provjerene iznimke koje naša funkcija može baciti, a ako pozovete neku drugu funkciju, moramo riješiti njezine provjerene iznimke ili deklarirati da naša iznimka može baciti iznimku također.

Kao i mnogi drugi moderni jezici, ni Kotlin ne razlikuje provjerene i neprovjerene iznimke. Ne specificirane iznimke bačene od strane funkcije možemo, ali ne morate obraditi. Ova dizajnerska odluka temelji se na praksi korištenja provjerenih iznimki u Javi. Iskustvo je pokazalo da Java pravila često zahtijevaju puno besmislenog kôda radi ponovnog bacanja ili ignoriranja iznimke, a pravila vas neće dosljedno štititi od pogrešaka koje se mogu dogoditi.

U gornjem primjeru, *NumberFormatException* nije provjerena iznimka. Zbog toga nas Java kompajler ne prisiljava da ju uhvatimo, a iznimku možemo lako vidjeti tijekom izvođenja programa. Ulazni podaci često su nevažeći i s njima se treba postupati oprezno. U isto vrijeme, metoda *BufferedReader.close* može baciti *IOException* iznimku, što je provjerena iznimka i nju treba obraditi. Većina programa ne može poduzeti nikakve smislene radnje ako zatvaranje protoka (eng. stream) podataka ne uspije pa potreban kôd za hvatanje iznimke je u ovom slučaju višak,

[5] Više on označenim iznimkama (eng. checked exceptions) na: https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/

odnosno tzv. „boilerplate“ kôd, koji smo već objasnili. Kotlin nema nikakvu posebno sintaksu za ovo, sve je implementirano kao biblioteka funkcija (Levia, 2017; Jemerov i Isakova, 2017).

3.5. Klase, objekti, sučelja

U ovom poglavlju ćemo objasniti klase te rad s njima puno detaljnije. Već neke osnove rada s njima te njihovu sintaksu možemo pročitati u poglavlju 3.2. Također, već znamo kako deklarirati metode i svojstva te koristiti jednostavni primarni konstruktor i raditi s *enumima*, ali još puno toga slijedi. Kotlinove klase i sučelja se malo razlikuju od onih u Javi, na primjer jedna razlika je da sučelja mogu sadržavati deklaracije svojstava. Još jedna od razlika je da Kotlinova deklaracija je konačna (eng. final) i javna (eng. public). Osim toga, ugniježđene klase nisu unutarnje (eng. inner) prema normalnim postavkama te ne sadrže implicitnu referencu na njihovu vanjsku (eng. outer) klasu. Za konstruktore, kratka sintaksa primarnog konstruktora izvrsno funkcionira u većini slučajeva, ali postoji i potpuna sintaksa koja vam omogućuje proglašavanje konstruktora sa netrivijalnom logikom inicijalizacije. Isto to možemo primijeniti i na svojstva. Sažeta sintaksa je uvijek lijepa. U ovom poglavlju također ćemo se upoznati sa novom ključnom riječju *object* koja deklarira klasu, ali također odmah i stvara instancu te klase, no više o tome poslije (Jemerov i Isakova, 2017; Tutorialspoint.com, 2020).

3.5.1. Klase

Ovo poglavlje najvećim dijelom govori o hijerarhiji klasa u Kotlinu u usporedbi s Javom. Proći ćemo kroz Kotlinove modifikatore pristupa koji su veoma slični onima u Javi, ali imaju različite zadane postavke. Također ćemo pričati od zapečaćenim modifikatorima (eng. sealed modifier) koji ograničavaju moguće podklase od klase. (Jemerov i Isakova, 2017)

Java nam omogućuje da stvorimo podklase bilo koje klase i nadjačamo (eng. override) bilo koju metodu, osim ako nije izričito označena konačnom (eng. final) ključnom riječi. To je često prikladno, ali je i problematično. U Javi javlja se problem takozvane krhke bazne klase (eng. fragile base class) koja nastaje kada modifikacije bazne klase mogu prouzrokovati pogrešno ponašanje podklasa zbog toga što promijenjeni kôd osnovne klase više ne odgovara pretpostavkama u njegovim

podklasama. Da bi se zaštitili od ovog problema jedna od poznatih metoda je metoda napisana od strane Joshuae Blocha(2008) gdje je rekao:

„design and document for inheritance or else prohibit it“

Što u biti znači da sve klase i metode koje nisu specifično namijenjene za podklase trebaju biti izričito označene kao konačne(eng. final). Tu filozofiju slijedi i Kotlin. Dok su u Javi klase i metode otvorene prema zadanim postavkama, u Kotlinu su konačne. Ako želimo omogućiti stvaranje podklasa klase, morati označiti klasu sa *open* ključnom riječi. Pored toga, moramo dodati *open* modifikator za svako svojstvo ili metodu koja se može nadjačati. Kako to izgleda pogledajmo u nastavku (Jemerov i Isakova, 2017).

```
open class Gumb : Clickable { // otvorena klasa, druge je mogu nasljeđivati
    fun iskljuci() {}       // funkcija je konačna, ne možemo ju nadjačati
    open fun animiraj() {}  // funkcija je otvorena, možemo ju nadjačati
    override fun click() {} // funkcija nadjačava funkciju koja je otvorena
}
```

Isječak kôda 47: Različiti modifikatori klasa i metoda

Imajmo na umu ako nadjačamo člana osnovne klase ili sučelja, nadjačani član će također, prema osnovnim postavka, biti označen kao otvoren. Ako to želimo promijeniti i zabraniti podklasama našim klasama da nadjačaju našu implementaciju, možemo eksplicitno označiti prevladavajućeg člana kao konačnog, pogledajmo u sljedećem primjeru.

```
open class Gumb : Clickable {
    final override fun click() {}
}
```

Isječak kôda 48: Otvorena klasa *Gumb*

I u Kotlinu kao i u Javi, klasu možemo proglasiti apstraktnom i takve klase ne mogu biti instancirane. Apstraktna klasa obično sadrži apstraktne članove koji nemaju implementacije i moraju biti nadjačane u podklasama. Apstraktni članovi su uvijek otvoreni, tako da ne moraju eksplicitno koristiti ključnu riječ *open*, kako to izgleda pogledajmo ispod.

```
abstract class Animirano {          // apstraktna klasa
    abstract fun animiraj()        // apstraktna funkcija
    // nema implementaciju i mora biti nadjačana
    open fun prestaniAnimirat() {
    // nije apstraktna funkcija, ali je otvorena
    }
    fun animirajDvaput() { // nije apstraktna funkcija, nije otvorena
```

```
        }
    }
```

Isječak kôda 49: Apstraktni članovi klase

Kako bi olakšali razumijevanje i kako bi uvijek mogli ponoviti modifikatore pristupa, napravljena je tablica prema D.Jemerov i S.Isakova sa komentarima kada što i gdje ide.

Tablica 2: Modifikator pristupa

Modifikator	Pripadajući član	Komentar
Final	Ne može biti nadjačan	Prema zadanim postavkama svi članovi klase su tog pristupa
Open	Može biti nadjačan	Mora se eksplicitno napisati i specificirati
Abstract	Mora biti nadjačan	Može biti korišten samo u apstraktnim klasama, apstraktni članovi ne mogu imati implementaciju
Override	Nadjačava člana u roditeljskoj klasi ili sučelju	Nadjačan član je otvoren prema zadanim postavkama, osima ako nije označen sa „final“

Pričajući o modifikatorima pristupa koji upravljaju nasljeđivanjem, prelazimo na drugu vrstu modifikatora, a to su modifikatori vidljivosti(eng. visibility modifiers).

Modifikatori vidljivosti pomažu u kontroli pristupa deklaracijama u bazi vašeg kôda. Ograničavajući vidljivost pojedinosti o implementaciji klase osiguravamo da ih možemo promijeniti bez rizika od kršenja kôda koji ovisi o klasi. U osnovi, modifikatori vidljivosti u Kotlinu slični su onima u Javi. Postoje javni (eng. public), zaštićeni (eng. protected) i privatni (eng. private) modifikatori, ali zadana vidljivost je drugačija, točnije u Kotlinu ako izostavimo modifikator vidljivosti, deklaracija će biti javna dok je za Javu to privatna za paket (eng. package-private). Kotlin pakete koristi samo za način organiziranja kôda, ne upotrebljava ih za kontrolu vidljivosti. Kao neku alternativu, Kotlin nudi novi modifikator vidljivost s imenom „unutarnji“ (eng. internal) koji označava vidljivost jedinu unutar istog modula, a modul je skup Kotlin datoteka kompiliranih zajedno. Prednosti interne vidljivosti je u tome što pruža stvarnu kapsulu za detalje implementacije vašeg modula. U tablici ispod možemo vidjeti sve modifikatore vidljivosti u Kotlinu koja je napravljena prema D.Jemerov i S.Isakova.

Tablica 3: Modifikatori vidljivosti

Modifikator	Član klase	Deklaracija najviše razine
Public	Vidljiv svuda	Vidljiv svuda
Internal	Vidljiv unutar modula	Vidljiv unutar modula
Protected	Vidljiv unutar potklasa	-
Private	Vidljiv unutar klase	Vidljiv u datoteci

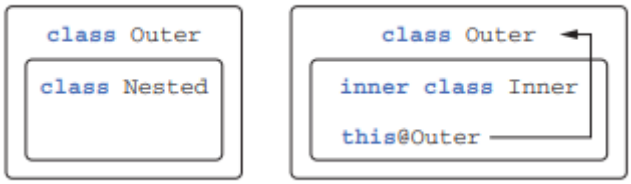
U Javi možemo pristupiti zaštićenom članu iz istog paketa, no u Kotlinu to nije moguće. U Kotlinu su pravila vidljivosti veoma jednostavna, a zaštićeni član vidljiv je samo u klasi i njezinim podklasama. Također, moramo imati na umu da funkcije proširenja klase ne mogu pristupiti njenim privatnim ili zaštićenim članovima. Posljednja razlika u pravilima vidljivosti između Kotlina i Jave koju ćemo spomenuti je ta da vanjska klasa ne vidi privatne članove svojih unutarnjih (ili ugniježđenih) klasa te s tom razlikom prelazimo na diskusiju o unutarnjim (eng. inner) i ugniježđenim (eng. nested) klasama.

Kao i u Javi pa tako i u Kotlinu možemo klasificirati klasu u drugoj klasi. Ovaj način koristan je za enkapsulaciju pomoćnih klasa ili za postavljanje kôda bliže mjestu gdje će ih se koristiti radi lakšeg čitanja i shvaćanja kôda. Razlika je u tome što u Kotlinu ugniježđene klase nemaju pristup vanjskoj instanci klase, osim ako to posebno ne zatražimo. Ugniježđena klasa u Kotlinu bez eksplicitnih modifikatora ista je kao i statična (eng. static) ugniježđena klasa u Javi. Da bismo ju pretvorili u unutarnju klasu tako da sadrži reference na vanjsku klasu moramo koristiti unutarnji modifikator. Tablica ispod opisuje razlike u ponašanju između Kotlina i Jave, a napravljena je prema D.Jemerov i S.Isakova.

Tablica 4: Korespondencija između ugniježđenih i unutarnjih klasa u Javi i Kotlinu

Klasa „A“ deklarirana sa drugom klasom „B“	**Java**	**Kotlin**
Ugniježđena klasa → nema referencu na vanjsku klasu	Statična klasa „A“	Klasa „A“
Unutarnja klasa → ima referencu na vanjsku klasu	Klasa „A“	Unutarnja klasa „A“

Razliku između ugniježđenih i unutarnjih klasa pogledajmo na slici ispod. Ugniježđene klase nemaju referencu na njihovu vanjsku klasu, dok unutarnje imaju (Jemerov i Isakova, 2017).



Slika 3: Razlika između unutarnje i ugniježđene klase

Sada kada smo objasnili razliku između ugniježđenih i unutarnjih klasa, vrijeme je da objasnimo detaljnije ugniježđene klase, odnosno, njihov slučaj upotrebe u stvaranju hijerarhije koja sadrži ograničen broj klasa, a to je veoma korisno u Kotlinu.

Sjetimo se primjera kada smo objašnjavali *when* izraz te toga kako smo svaki put morali definirati *else* granu, odnosno, granu koja će se izvršiti ukoliko niti jedna prijašnja grana nije zadovoljila uvjete. Ukoliko to ne učinimo Kotlinov kompajler će nas upozoriti na tu grešku. Dodavanje te grane je nezgodno ako se to mora raditi često. Štoviše, dodate li novu podklasu, kompajler neće otkriti da se išta promijenilo. Ako zaboravite dodati novu granu, biti će odabrana zadana, što može dovesti do suptilnih pogrešaka. No, Kotlin nudi rješenje za ovaj problem, a on se naziva zatvorene klase (eng. sealed classes). Roditeljeve klase označavamo zatvorenim modifikatorom, što ograničava stvaranje podklasa. Sve izravne podklase moraju biti ugniježđene u roditeljevoj klasi. Ako obradimo sve podklase zatvorene klase u izrazu *when*, tada ne moramo navesti *else* granu. Moramo imati na umu da zatvoreni modifikator podrazumijeva da je klasa otvorena, ne treba vam eksplicitni otvoreni modifikator, proučimo sljedeći primjer.

```
sealed class Expr {              // označimo baznu klasu kao "sealed"
    // dodamo sve moguće podklase kao "nested" klase
    class Num(val value: Int) : Expr()
    class Sum(val left: Expr, val right: Expr) : Expr()
}
fun eval(e: Expr): Int =
    when (e) {
          //u "when" izrazu prođemo kroz sve moguće slučajeve pa nam
          ne treba "else" grana
          is Expr.Num -> e.value
          is Expr.Sum -> eval(e.right) + eval(e.left)
}
```

Isječak kôda 50: *sealed* klasa

3.5.2.Sučelja

Kotlin sučelja slična su onima u Javi, mogu sadržavati definicije apstraktnih metoda kao i implementacije ne apstraktnih metoda, ali ne mogu sadržavati nijedno stanje (eng. state). Da bismo deklarirali sučelje u Kotlinu, koristiti ćemo ključnu riječ „interface“. Jednostavno sučelje prikazano je u nastavku (Baeldung.com, 2020).

```
interface Clickable {
    fun click()
}
```

Isječak kôda 51: Definiranje sučelja u Kotlinu

Ovako se deklarira sučelje s jednom apstraktnom metodom koja se zove „click“. Sve ne apstraktne klase koje implementiraju sučelje trebaju osigurati implementaciju ove klase. U nastavku pogledajmo kako implementirati sučelje.

```
class Button : Clickable {
     override fun click() = println("Pritisnut sam!")
 }

 Button().click()

// print
 Pritisnut sam
```

Isječak kôda 52: Implementacija sučelja

Kotlin koristi dvotočku iza naziva klase kako bi zamijenio i *extend* i *implements* ključne riječi koje se koriste u Javi, a sam njihov naziv nam govori što rade. Kao i u Javi, klasa može implementirati (eng. implements) onoliko sučelja koliko želi, ali može proširiti (eng. extend) samo jednu klasu.

Modifikator nadjačavanja (eng. override), sličan je anotaciji *@Override* u Javi i koristi se za označavanje metoda i svojstava koja nadjačavaju one iz superklase ili sučelja. Za razliku od Jave, u Kotlinu korištenje modifikatora nadjačavanja je obavezno. To nas sprečava od slučajnog nadjačavanja metode ako je dodana nakon što smo napisali implementaciju, u tom slučaju kôd se neće kompilirati osim ako metodu izričito ne označimo kao nadjačavanje ili ju preimenujemo.

Metoda sučelja može imati zadanu implementaciju. Za razliku od Jave, koja zahtijeva da takve implementacije označite zadanom ključnom riječi, Kotlin nema posebne napomene za takve metode, samo joj damo tijelo metode. Ako pokušamo

implementirati dva sučelja koja sadrže isti naziv metode u našu klasu, kompajler će nam izbaciti grešku osim ako izričito ne navedenom koju metodu implementiramo. Također, Kotlinov kompajler nas prisiljava da osiguramo vlastitu implementaciju. Da bismo pozvali naslijeđenu implementaciju, koristimo ključnu riječ „super“ isto kao i u Javi, ali sintaksa za odabir određene implementacije je različita. Dok u Javi možemo staviti naziv osnovnog tipa prije „super“ ključne riječi, kao na primjer `Clickable.super.show()`, u Kotlinu moramo staviti osnovni tip u izlomljene zagrade („< >“), na primjer `super.<Clickable>.show()` (Jemerov i Isakova, 2017).

3.5.3. Objekti

Object ključna riječ javlja se u velikom broju slučajeva, ali svi ti slučajevi imaju istu osnovnu ideju, a to je da ključna riječ definira klasu i istovremeno stvara njezinu instancu, drugim riječima, objekt te klase. Ovo su neke od situacija gdje koristimo objekt:

- Definiranje Singleton uzorka dizajna
- „Companion“ objekti koji mogu sadržavati tvorničke metode i druge metode koje su povezane sa ovom klasom, ali ne zahtijevaju instancu klase. Njihovim članovima može se pristupiti preko imena klase
- Objekti izraza(eng. expression) koriste se umjesto Javinih anonimnih unutarnjih klasa.

3.5.3.1. Singleton

Prilično uobičajena pojava u dizajnu objektno orijentiranih sustava je klasa za koju vam je potrebna samo jedna instanca. U Javi to se obično provodi pomoću Singleton uzorka dizajna. Da bi ga izradili potrebno je definirati klasu s privatnim konstruktorom i statičkim poljem koji drži jedinu postojeću instancu klase. Kotlin za to nudi prvoklasnu jezičnu podršku pomoću značajke deklariranja objekata. Deklaracija objekta kombinira deklaraciju klase i deklaraciju jedne instance te klase.

```
class Singleton private constructor() {

    private object HOLDER {
        val INSTANCE = Singleton()
    }

    companion object {
        val instance: Singleton by lazy { HOLDER.INSTANCE }
    }
}
```

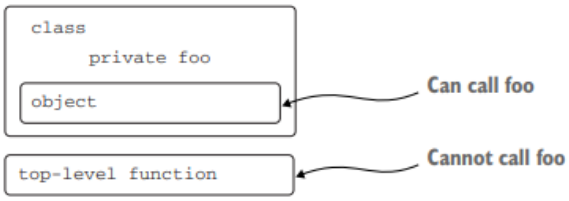
Isječak kôda 53: Primjer Singletona u Kotlinu

Baš kao i klasa, objektna deklaracija može sadržavati deklaracije svojstava, metoda, blokove inicijalizatora, itd. Jedine stvari koje nisu dopuštene su konstruktori (primarni i sekundarni). Za razliku od slučajeva redovnih klasa, objektne deklaracije kreiraju se odmah na točki definicije, a ne putem poziva konstruktora s drugih mjesta u kôdu. Prema tome, definiranje konstruktora za deklaraciju objekta nema smisla. I baš kao i varijabla, deklaracija objekta vam omogućuje pozivanje metoda i pristupa svojstvima koristeći ime objekta s lijeve strane znaka točke(' . '). To onda izgleda ovako: `uplataPlaca.sviZaposlenici.add(Osoba(ime, prezime, IBAN))`.

Deklaracije objekata također se mogu naslijediti iz klasa i sučelja. To je često korisno kada okvir koji koristite zahtijeva da implementiramo sučelje, ali naša implementacija ne sadrži niti jedno stanje. Koristimo Singleton objekte u bilo kojem kontekstu gdje se može koristit obični objekt(instanca klase). Također, možemo deklarirati objekte u klasi. Takvi objekti također imaju samo jednu instancu, nemaju zasebnu instancu po instanci klase koja ih sadrži.

3.5.3.2. „Companion“ objekti

Klase u Kotlinu ne mogu imati statičke članove, Java *static* ključna riječ nije sastavni dio Kotlinovog programskog jezika. Kao zamjena tome, Kotlin se oslanja na funkcije na razini paketa i deklaracije objekata. U većini slučajeva preporuča se upotreba funkcija najviše razine, ali funkcije najviše razine ne mogu pristupiti privatnim članovima klase, kao što prikazuje slika 4 koja je preuzeta iz (D.Jemerov i S.Isakova, 2017)



Slika 4: Privatni članovi ne mogu biti korišteni u funkcijama više razine

Prema tome, ako trebamo napisati funkciju koja se može pozvati bez instance klase, ali joj je potreban pristup unutrašnjosti klase, možemo napisati člana objektne deklaracije unutar te klase. Primjer takve funkcije bila bi tvornička metoda.

Jedan od objekata definiranih u klase može se označiti posebnom ključnom riječi Compaion. Ako to učinimo, dobivamo mogućnost pristupa metodama i svojstvima tog objekta izravno kroz naziv klase koja sadrži, bez izričitog navođenja imena objekta. Rezultirajuća praksa izgleda upravo kao pozivanje statičke metode u Javi, slijedi primjer.

```
class A {
    companion object {
        fun bar() {
            println("Companion objekt pozvan")
        }
    }
}

A.bar()

// print
 "Companion objekt pozvan"
```

Isječak kôda 54: *companion* objekt

„Companion“ objekt ima pristup svim privatnim članovima klase, uključujući i privatni konstruktor te iz tog razloga idealan je kandidat za primjeru tvorničkog uzorka dizajna (eng. factory design pattern). Tvorničke metode vrlo su korisne. One se mogu

imenovati prema svojoj namjeni. Pored toga, tvornička metoda može vratiti podklase klase u kojoj je metoda deklarirana. Također, možemo izbjeći stvaranje novih objekata kada to nije potrebno.

```
class User {
    val nickname: String
    constructor(email: String) {
        nickname = email.substringBefore('@')
    }
    constructor(facebookAccountId: Int) {
        nickname = getFacebookName(facebookAccountId)
    }
}
```

Isječak kôda 55: Izbjegavanje stvaranja novih objekata

Na primjer, možemo osigurati da svaka e-pošta odgovara jedinstvenoj korisničkoj instanci i vratimo postojeću instancu umjesto nove kada se tvornička metoda poziva e-poštom koja je već u predmemoriji. Ako trebamo proširiti takve klase, korištenjem nekoliko konstruktora može biti bolje rješenje, jer se „comapion“ objekti ne mogu nadjačati u podklasama, u nastavku pogledajmo kako izgleda taj primjer kôda.

```
// označavamo primarni konstuktor kao privatni
class User private constructor(val nickname: String) {
// deklaracija "companion" objekta
    companion object {
// deklariramo imenovani "companion" objekt
        fun newSubscribingUser(email: String) =
            User(email.substringBefore('@'))
// tvornička metoda za kreiranje Usera prema Facebook ID
        fun newFacebookUser(accountId: Int) =
            User(getFacebookName(accountId))
    }
}
```

Isječak kôda 56: Proširivanje klase

3.6. Rad s podacima

3.6.1. Dohvaćanje s API-a

U ovom dijelu poglavlja najviše ćemo se bazirati na stvarni primjer dohvaćanja podataka preko API-a. API dolazi od engleske izvedenice *application programming interface* što u prijevodu znači aplikacijsko programskog sučelje, ali u daljnjem tekstu koristiti ćemo skraćenicu API. API je skup određenih pravila i specifikacija koje programeri slijede tako da se mogu služiti uslugama ili resursima operacijskog sustava ili nekog drugog složenog programa kao standardne biblioteke rutina(funkcija, procedura, metoda), struktura podataka, objekata i protokola. Korištenje API-ja uvelike štedi vrijeme programerima jer mogu koristiti rad drugih programera iz razloga što svi koriste iste standarde (Levia, 2017)

Koristit ćemo jedan od veoma poznatih i veoma korišten API, a to je „OpenWeatherMap“ API jer je jedan od vodećih pružatelja digitalnih informacija o vremenu. Taj API koristit ćemo za dohvaćanje podataka, a neke druge klase ćemo koristiti za zahtjeve prema njemu. Budući da je Kotlinova interoperabilnost izuzetno snažna, za zahtjeve poslužitelja možemo koristiti bilo koju biblioteku, a jedna od najkorištenijih je svakako „Retrofit 2.0“[6] koju ćemo mi koristiti. Jedini razlog korištenja ove biblioteke je njezina jednostavnost jer ne moramo koristiti biblioteku treće strane da bi postigli ovaj jednostavni cilj. Osim toga, Kotlin pruža i neke funkcije proširenja koji će zahtjeve učiniti mnogo jednostavnijim. Napomena da ćemo se u ovom dijelu rada držati veoma jednostavnih koncepata samo da shvatimo kako ovo funkcionira i koja je razlika između Jave, a u poglavlju 6 ćemo koristiti napredne koncepte te uvidjeti kako to zapravo izgleda u stvarnom svijetu programiranja. Primjer nad kojim ćemo objašnjavati preuzet je iz literature (Levia, 2017), ali ima nekoliko preinaka. Krenimo sa izradom klase zahtjeva.

```
class Request(val url: String) {

    fun run() {
        val forecastJsonStr = URL(url).readText()
        Log.d(javaClass.simpleName, forecastJsonStr)
    }
}
```

Isječak kôda 57: Klasa zahtjeva

[6] Više o Retrofit-u pročitajte na: https://square.github.io/retrofit/

Konstruktor jednostavno prima URL (skraćenica od engleske izvedenice „Uniform Resouce Locator“)[7], nakon toga funkcija *run* čita rezultate i ispisuje JSON[8] format u *Logcat*. *Logcat* je alat naredbenog retka koji ubacuje dnevnik sistemskih poruka, uključujući tragove snopa kada uređaj baca pogrešku i poruke koje smo napisali iz svoje aplikacije uz pomoć *Log* klase. Implementacija je vrlo jednostavna kada se koristi funkcija *readText* i to je jedna od funkcija proširenja iz Kotlinove standardne biblioteke. Ova metoda se ne preporučuje kod velikih odgovora, odnosno kada dohvaćamo puno podataka s API-a odjednom, ali biti će dovoljno dobra za naš slučaj (Levia, 2017).

Ako ovaj kôd usporedimo s onim koji vam je potreban u Javi, vidjet ćemo da smo već uštedili puno vremena i truda samo pomoću standardne biblioteke. Kako bi mogli izvršiti ovaj zahtjev, aplikacija mora moći koristiti internet, a to dobivamo tako da sljedeću liniju kôda dodajemo u AndroidManifest.xml.

```
<uses-permission android:name="android.permission.INTERNET" />
```

Ako smo već radili sa sličnim stvarima u Javi onda sigurno znamo da HTTP zahtjevi nisu dopušteni na glavnoj dretvi, u suprotnom aplikacija će baciti iznimku. To je zato što blokiranje UI dretve je stvarno loša praksa. Uobičajeno rješenje za Android je korištenje *AsyncTaks*, to je rješenje za korištenje više asinkronih dretvi da bi izvršili neki zadatak. Primjerice, za dohvaćanje ovih podataka u asinkronu dretvu bi stavili da se dohvaćaju podaci i spremaju u bazu, dok bi na glavnoj petlji stavili nekakvu animaciju ili poruku da se dohvaćaju podaci te blokirali da korisnik može kliktati dalje po aplikaciji. *AsyncTasks* su jako opasni ako ih se ne koristi pažljivo jer do trenutka dok ne dođe do *postExecute*, aktivnost je već mogla biti uništena i aplikacija će se srušiti. Anko, o kojem ćemo pričati više poslije, pruža vrlo jednostavan DSL za rješavanje sinkronizacije, koje će odgovarati u većini slučajeva. U osnovi on pruža *doAsync* funkciju koja će svoj kôd izvršiti u drugoj dretvi s mogućnošću da se vrati u glavnu dretvu pozivom UI dretve. Izvršavanje zahtjeva u sekundarnom nizu je jednostavno, a izvodi se ovako:

```
val url = "http://api.openweathermap.org/data/2.5/forecast/daily?" +

"APPID=15646a06818f61f7b8d7823ca833e1ce&q=94043&mode=json&units=metri
c&cnt="

doAsync() {
```

[7] Više o URL pročitajte na: https://hr.wikipedia.org/wiki/URL
[8] Više o JSON pročitajte na: https://www.json.org/json-en.html

```
        Request(url).run()
        uiThread { longToast("Zahtjev obavljen") }
    }
```

Isječak kôda 58: Izvršavanje zahtjeva

Lijepa stvar za *uiThread* je ta što ima različite implementacije, ovisno o objektu pozivatelja. Ako ga koristi aktivnost (eng. Activity), kôd *uiThreada* neće se izvršiti ako metoda *activity.isFinished()* vrati istinitu vrijednost i neće se srušiti ako aktivnost više nije valjana. Možemo koristiti i vlastitog izvršitelja. Metoda *doAsync* vraća Java Future, u slučaju da želite raditi sa budućnosti, a ako želite raditi samo sa budućim rezultatom onda možemo koristiti *doAsyncResult*. Ovo je puno jednostavnije i lakše za čitati nego *AsyncTasks*. U ovom primjeru smo samo slali statični URL na zahtjev da provjerimo je li sadržaj pravilno primljen i da li smo u mogućnosti primiti ga u aktivnost. Kasnije ćemo raditi JSON raščlanjivanje i pretvorbu u podatkovne klase, no prije toga važno je razumjeti što su to podatkovne klase (eng. Data class) (Levia, 2017).

3.6.2. Data klase

Data klase ili ako želimo to reći hrvatskim jezikom, podatkovne klase su snažna vrsta klasa koje izbjegavaju dosadni i ponavljajući kôd koji moramo koristiti u Javi da bi stvorili POJO (eng, plain old Java object)[9]. Klase koje se koriste za održavanje stanja su veoma jednostavne u operacijama koje obavljaju. Obično pružaju samo *gettere* i *settere* da bi pristupile njihovim poljima. Definiranje nove data klase je vrlo jednostavno.

```
data class Forecast(
    val date: Date,
    val temperature: Float,
    val details: String
)
```

Isječak kôda 59: Data klasa za vremensku prognozu

Uz data klasu, osim svojstava o kojima smo već pričali, dobivamo i nekoliko zanimljivih funkcija:

- Equals() – uspoređuje svojstva oba objekta kako bi se osiguralo da su identični
- hashCode() – dobivamo hash kôd koji je također izračunat iz svojstava
- copy() – možemo kopirati objekt, mijenjajući potrebna svojstva

[9] Više o POJO objektima na: https://en.wikipedia.org/wiki/Plain_old_Java_object

- toString() – za generiranje reprezentacije niza koja pokazuju sva polja u redoslijedu deklaracije
- Skup numeriranih funkcija koje su korisne za mapiranje objekta u varijable

Ako koristimo nepromjenjivost (eng. immutability), o čemu smo govorili prije nekoliko poglavlja, ustanoviti ćemo da ako želimo stanje objekta, potrebna je nova instanca klase s promijenjenim jednim ili više njegovih svojstava. Podatkovne klase uključuju metodu *copy()*, što će postupak učiniti vrlo jednostavnim i intuitivnim. Na primjer, ako trebamo mijenjati temperaturu prognoze, to možemo jednostavno učiniti na sljedeći način.

```
val f1 = Forecast(Date(), 27.5f, "Suncan dan")
val f2 = f1.copy(temperature = 30f)
```

Isječak kôda 60: Mijenjanje varijabli koristeći *copy()* metodu

Na ovaj način kopiramo prvu prognozu i mijenjamo samo temperaturno svojstvo bez promjene stanja izvornog objekta.

Mapiranje objekta u varijablu je postupak poznat kao destrukturiranje deklaracije i sastoji se od mapiranja svakog svojstva objekta u varijablu. Pogledajmo primjer na osnovi prethodnog primjera.

```
val date = f1.component1()
val temperature = f1.component2()
val details = f1.component3()
```

Isječak kôda 61: Mapiranje objekta u varijablu

Logika koja stoji iza ove značajke je vrlo moćna i može nam pomoći u pojednostavljivanju kôda u mnogim situacijama. Na primjer, klasa *Map* ima implementirane neke funkcije koje omogućuju obnavljanje ključeva i vrijednosti u ponavljanju (Kotlinlang.org, 2020; Baeldung.com, 2020).

```
for ((key, value) in map) {
    Log.d("map", "Kljuc:$key, vrijdnost:$value")
}
```

Isječak kôda 62: Obnavljanje ključeva i vrijednosti

3.6.3. Parsiranje podataka

Sada kada znamo kako stvoriti data klase, spremni smo započeti parsiranje, točnije, analizu podataka. Ako u pregledniku otvorimo URL koji smo koristili u 3.6.1. poglavlju „Dohvaćanje s API-a“, možemo vidjeti strukturu JSON datoteke. U osnovi se sastoji od objekta koji sadrži grad i popisa predviđanja vremenskih prognoza. Grad ima id, ime, svoje koordinate i zemlju kojoj pripada. Svaka prognoza dolazi s dobrim nizom informacija kao što su datum, različite temperature i vremenski objekt s opisom i ID-om (Levia, 2017).

Obično, kada koristimo takav neki javni API, većinom nećemo trebati sve podatke koje on vraća pa iz tog razloga moramo razdvojiti one podatke koji nam trebaju od onih koji nam ne trebaju. Također, ponekad je dobro raščlaniti sve dostupne podatke po klasama, u slučaju da nam budu trebale u budućnosti. U našem konkretnom primjeru, to bi izgledalo ovako.

```
data class ForecastResult(val city: City, val list: List<Forecast>)

data class City(
    val id: Long, val name: String, val coord: Coordinates,
    val country: String, val population: Int
)

data class Coordinates(val lon: Float, val lat: Float)

data class Forecast(
    val dt: Long, val temp: Temperature, val pressure: Float,
    val humidity: Int, val weather: List<Weather>,
    val speed: Float, val deg: Int, val clouds: Int,
    val rain: Float
)

data class Temperature(
    val day: Float, val min: Float, val max: Float,
    val night: Float, val eve: Float, val morn: Float
)

data class Weather(
    val id: Long, val main: String, val description: String,
    val icon: String
)
```

Isječak kôda 63: Raščlanjivanje podataka po klasama

Za ovaj primjer koristiti ćemo Moshi[10] biblioteku za raščlanjivanje JSON datoteke u naše klase, više o Moshi i kasnijem dijelu rada. Da bi mogli uspješno raščlaniti podatke, svojstva moraju imate isti naziv kao i oni u JSON datoteci ili im

[10] Više pročitajte na: https://github.com/square/moshi/

eksplicitno odrediti serializirano ime. Dobra praksa koja se objašnjava u većini softverskih arhitektura je korištenje različitih modela za različite slojeve u našoj aplikaciji kako bi se razdvojili jedan od drugog, zbog toga dobro je pojednostaviti deklaraciju tih klasa jer ćemo ih u ostatku aplikacije pretvarati prije nego što ih budemo koristili. Da bi ovaj primjer uspješno radio i parsirao podatke, treba nad njim napraviti puno izmjena te dodati nekoliko stvari. U ovom dijelu rada nije predviđeno da se ide u detalje kako se implementira pa ćemo taj dio sačuvati za poglavlje 6. nad stvarnom aplikacijom. No, ako želimo isprobati ovaj API te ovu aplikaciju, primjer možemo pronaći u literaturi u (Leiva, 2017) na stranicama od 42 do 47.

3.7. Delegati

U ovom poglavlju ćemo obraditi skup standardnih delegata uključenih u Kotlinovu standardnu biblioteku. Također, proći ćemo kroz najčešće situacije u kojima je delegat zaista koristan.

3.7.1. Lazy

U računalnom programiranju, lijena inicijalizacija (eng. lazy initialization) je taktika odgađanja stvaranja objekta, izračuna vrijednosti ili nekog drugog skupog postupka dok prvi put to nije potrebno. To je vrsta lijene procjene koja se posebno odnosi na instanciranje objekata ili drugih resursa. *Lazy()* je funkcija koja uzima lambdu i vraća instancu *Lazy<T>* koja može poslužiti kao delegat za implementaciju lijenog svojstva. Prvi poziv *get()* izvršava lambdu proslijeđenu u *lazy()* i pamti rezultat , sljedeći pozivi za *get()* jednostavno vraćaju zapamćeni rezultat. S njime možemo uštedjeti memoriju i preskočiti inicijalizaciju sve dok entitet nije potreban. Ovaj delegat često se koristi kod korištenja i inicijalizacija baza u aplikaciji pa tako pogledajmo sljedeći primjer.

```
class App : Application() {
    val database: SQLiteOpenHelper by lazy {
        MyDatabaseHelper(applicationContext)
    }

    override fun onCreate() {
        super.onCreate()
        val db = database.writableDatabase
    }
}
```

Isječak kôda 64: Primjer *lazy* delegata

U ovom se primjeru baza podataka zapravo ne inicijalizira dok se ne pozove prvi put *onCreate*. U tom trenutku smo sigurni da kontekst aplikacije postoji i da je spreman za upotrebu. *Lazy* delegat je sigurna za dretveni rad(eng. thread safe). Možemo također koristiti i *LazyThreadSafetyMode.NONE* ako se brinete o višestrukoj dretvenosti i želite postići dodatne performanse (Kotlinlang.org, 2020).

3.7.2. Observable

Ovaj delegat pomoći će nam da otkrijemo promjene na bilo kojem entitetu koji moramo promatrati. Izvršiti će deklarirani lambda izraz svaki put kada se postavi zadana funkcija. Nakon što dodijelimo novu vrijednost, dobivamo delegirano svojstvo, staru vrijednost i novu vrijednost.

```
class ViewModel(val db: MyDatabase) {
    var myProperty by Delegates.observable("") { _, _, new ->
        db.saveChanges(this, new)
    }
}
```

Isječak kôda 65: Primjer *observable* delegata

Ovaj primjer predstavlja lakšu verziju ViewModel klase, o njima ćemo više govoriti naknadno u radu, koja je svjesna promjena u *myProperty* i sprema ih u bazu svaki put kada se dodijeli nova vrijednost (Kotlinlang.org, 2020).

3.7.3. Vetoable

Ovo je posebna vrsta promatranja koja omogućava odlučivanje treba li vrijednost biti spremljena ili ne. Može se koristiti za provjeru nekih uvjeta prije nego što spremite vrijednosti.

```
var positiveNumber = Delegates.vetoable(0) { _, _, new ->
    new >= 0
}
```

Isječak kôda 66: Primjer *vetoable* delegata

Prethodni delegat omogućit će spremanje nove vrijednosti samo ako je pozitivan broj. Unutar lambda, zadnja linija predstavlja povratnu vrijednosti. Ne trebamo koristiti povratnu riječ *return* inače se neće kompilirati (Kotlinlang.org, 2020).

3.7.4. Lateinit

Ponekad nam je potreban određeni entitet kako bi mogli inicijalizirati neki drugi entitet, ali u konstruktoru nemamo potrebno stanje ili čak do njih ne možemo ni pristupiti. Ovaj drugi slučaj događa se s vremena na vrijeme u Androidu, to zna biti u aktivnostima, fragmentima, uslugama, itd. Međutim, u apstraktnom entitetu potrebna je vrijednost prije nego što konstruktor završi izvršavanje. Ne možemo čekati dokle želimo da bismo dodijelili vrijednost entitetu. Postoji nekoliko opcija za to. Prva je da koristimo *nullable* varijablu i postavimo ju na *null*, dok ne dobijemo pravu vrijednost, ali tada svuda moramo provjeravati da li je to svojstvo *null* ili ne. Ako smo sigurno da ovaj entitet neće biti ništav prvi put kada ga budemo koristili, možda ćemo za to morati pisati nepotreban kôd. Druga mogućnost je upotreba *lateinit*, koji identificira da svojstvo treba imati *non-nullable* vrijednosti, ali njegovo dodjeljivanje će se odgoditi. Ako se vrijednost zatraži prije dodjeljivanja, baciti će iznimku koja jasno identificira svojstvo kojem se pristupa. *Lateinit* nije baš pravi delegat, već je modifikator svojstva i mora biti napisan prije svojstva, pogledajmo primjer.

```
class App : Application() {

    companion object {
        lateinit var instance: App
    }

    override fun onCreate() {
        super.onCreate()
        instance = this
    }
}
```

Isječak kôda 67: Primjer *lateinit* delegata

Lateinit je također nezamjenjiv za rad s naknadnim dodavanjem ovisnosti (eng. dependency injection) kao što je *Dagger* ili *Koin*, a vrlo je koristan i za testove (Kotlinlang.org, 2020).

3.8. Rad s bazom podataka

Kao što već znamo, Android koristi *SQLite* kao sustav za upravljanje bazom podataka. *SQLite* je baza podataka ugrađena u aplikaciju i zaista je lagana te iz tog razloga je dobra opcija za mobilne aplikacije. Međutim, API za rad s bazama podataka u Androidu je prilično sirov. Vidjet ćemo da trebamo napisati mnogo SQL rečenica da

bi uspješno obavili neke od zadataka. Na svu sreću tu je Anko kojeg smo već spomenuli, ali detaljnije će biti obrađen u poglavlju 4.3.1., mješavinom njega i Kotlina te zadatke mnogo pojednostavljujemo. Njegova glavna svrha je generiranje izgleda sučelja pomoću kôda umjesto XML-a, no uvelike nam može pomoći i pri radu s bazom. Naravno, postoji veliki broj biblioteka za rad s bazama podataka u Androidu, a sve one rade zahvaljujući interoperabilnosti koju Kotlin posjeduje. Ali ne moramo ih koristiti, dapače za neke jednostavnije aplikacije one nisu ni preporučljive.

Anko nudi snažan *SqliteOpenHelper* koji uvelike pojednostavljuje stvari. Kada koristimo uobičajeni *SqliteOpenHelper*, moramo pozvati metodu *getReadableDatabase*() ili *getWritableDatabase*(), a zatim možemo izvoditi naše upite nad objektom koji dobijemo. Nakon toga ne bismo trebali zaboraviti pozvati metodu *close*(). *SqliteDatabase* sadrži puno korisnih funkcija proširenja koje ćemo kasnije koristiti, ali sada ćemo definirat naše tablice i implementirat *SqliteOpenHelper*.

Stvaranje nekoliko objekata koji će predstavljati naše tablice iz baze podataka biti će korisno za izbjegavanje pogrešno napisanih naziva tablica ili stupaca i ponavljanja. Za ovaj primjer koristit ćemo dvije tablice: jedna će spremiti podatke grada, a druga prognozu za pojedine dane. Druga tablica će imati vanjski ključ (eng. foreign key) na prvu tablicu. *CityForecastTable* prvo daje naziv tablice, a zatim skup stupaca, odnosno atributa koji su joj potrebni: id, naziv grada, zemlja, ovako to trenutno izgleda.

```
object CityForecastTable {
    val NAME = "CityForecast"
    val ID = "_id"
    val CITY = "city"
    val COUNTRY = "country"
}
```

Isječak kôda 68: Objekt *CityForecastTable*

DayForecast će imati malo više podataka, pa ćemo trebati skup stupaca koje možemo vidjeti dolje. Posljednji stupac *cityId* će biti vanjski ključ koji smo spomenuli prije.

```
object DayForecastTable {
    val NAME = "DayForecast"
    val ID = "_id"
    val DATE = "date"
    val DESCRIPTION = "description"
    val HIGH = "high"
    val LOW = "low"
    val ICON_URL = "iconUrl"
```

```
        val CITY_ID = "cityId"
    }
```

Isječak kôda 69: Objekt *DayForecastTable*

SQLiteOpenHelper će u osnovi upravljati stvaranjem i nadogradnjom naše baze podataka te će pružiti *SqliteDatabase* kako bismo mogli raditi s njom. Upiti se uobičajeno pišu u nekoj drugoj klasi. Kod implementacije ove klase možemo koristiti *lazy* delegat koji je opisan u prethodnom poglavlju kako ne bi stvarali nepotrebne objekte. Također, *lazy* delegat bi spriječio stvaranje nekoliko instanci iz različitih dretvi, to bi se dogodilo samo ukoliko dvije dretve istovremeno pokušaju pristupiti instanci, što je teško, ali može se dogoditi ovisno o vrsti aplikacije koju implementirate. Ako ne koristimo nikakve biblioteke onda se stvaranje tablica vrši pomoću pisanja sirovog „Create table“ upita gdje definiramo sve stupce i njihove vrste. Međutim, Anko pruža jednostavnu funkciju proširenja koje prima naziv tablice i skup *Pair* objekata koji identificiraju ime i vrstu stupca. Uz ovo, postoji još biblioteka *Room* koja će biti objašnjena detaljnije u poglavlju 4.5.6., a onu pruža još lakšu i jednostavniju implementaciju baze podataka. Još moramo znati da sve podatke koje spremamo u našu bazu podataka se nalaze u predmemoriji te ukoliko se aplikacija obriše ili isprazni predmemoriju, naših podataka više neće biti. Kada radimo ažuriranja naših aplikacija, možemo napraviti migracije koje će reći aplikaciji što da radi sa pojedinim stupcima i elementima (Levia, 2017).

3.8.1. Kreiranje modela

Do sada smo već pokrili stvaranje *SQLiteOpenHelper*-a, ali sada nam još treba način da ga upotrijebimo za uvrštavanje naših podataka unutar baze te ih vraćamo kada su nam potrebni. Da bi to učinili prvo trebamo stvoriti modele za našu bazu podataka. Jedan od uobičajenih slučajeva pohranjivanja vrijednosti je *map*. To se često događa u aplikacijama poput raščlanjivanja JSON-a ili obavljanja drugih "dinamičnih" stvari. U tom slučaju možemo koristiti samu instancu *map* kao delegat za delegirano svojstvo (eng. delegated property)[11]. *Map* delegat ćemo koristiti za mapiranje tih polja izravno u bazu podataka, ovako bi otprilike trebao izgledati jedan model.

[11] Više o *map* delegatu na: https://kotlinlang.org/docs/reference/delegated-properties.html#storing-properties-in-a-map

```
class CityForecast(
    val map: MutableMap<String, Any?>,
    val dailyForecast: List<DayForecast>
) {
    var _id: Long by map
    var city: String by map
    var country: String by map

    constructor(
        id: Long, city: String, country: String,
        dailyForecast: List<DayForecast>
    )
            : this(HashMap(), dailyForecast) {
        this._id = id
        this.city = city
        this.country = country
    }
}
```

Isječak kôda 70: Model *CityForecast*

Zahvaljujući delegatima vrijednosti će se preslikati u odgovarajuća svojstva na temelju imena ključa. Ako želimo da mapiranje savršeno funkcionira, imena svojstava moraju biti jednaka imenima stupaca u bazi podataka. Umjesto korištenja mapiranja, izvlačenje vrijednosti iz svojstava bi bilo prikladnije te zbog toga imamo sekundarni konstruktor u gornjem primjeru. Prosljeđujemo mu prazan *map*, ali opet zahvaljujući delegatu, kada postavimo vrijednost nekom svojstvu, on automatski dodaje novu vrijednost *map*-u. Na taj način, imamo naš *map* spreman za dodavanje u bazu podataka. Ovaj primjer također je preuzet iz literature (Leiva Antoinio, 2017) te je korišten samo za prikaz osnovnih stvari, ukoliko želite potpuni funkcionalni primjer, potražite ga pod navedenom literaturnom.

3.8.2. Unos i ispis podataka

SQLiteOpenHelper je samo alat, odnosno, kanal između objektno orijentiranog i SQL svijeta. Koristiti ćemo ga za traženje podataka koji su već pohranjeni u bazi te također za spremanje novih podataka u bazu. Prateći prethodni primjer, definirati ćemo jednu metodu koja će tražiti podatke o vremenskoj prognozi baziranoj na poštanskom broju i datumu.

```
fun requestForecastByZipCode(zipCode: Long, date: Long) =
forecastDbHelper.use {
    ...
}
```

Isječak kôda 71: Metoda za traženje podataka prema poštanskom broju

Prvi zahtjev koji treba učiniti je dnevna prognoza jer nam je ona potrebna za kreiranje *City* objekta, no Anko nam pruža jednostavni alat za izradu tog zahtjeva pa ćemo to i iskoristiti.

```
val dailyRequest = "${DayForecastTable.CITY_ID} = ? " +
        "AND ${DayForecastTable.DATE} >= ?"

val dailyForecast = select(DayForecastTable.NAME)
    .whereSimple(dailyRequest, zipCode.toString(), date.toString())
    .parseList { DayForecast(HashMap(it)) }
```

Isječak kôda 72: Jedan on načina pisanja zahtjeva

Ovo je samo jedan on načina pisanja zahtjeva, postoji još mnogo njih koje možemo pronaći na internetu, u knjigama ili slijedeći neke od internetskih udžbenika (eng. tutorials). Ovaj primjer je veoma jednostavan jer „select“ funkcija samo traži da joj se proslijedi ime tablice. Kada se upoznamo sa nekim naprednim metodama i alatima u Kotlinu, sve ovo će biti mnogo jednostavnije ili kraće, ali prvo je bitno da shvatimo osnovu rada s bazom kako bi mogli lakše razumjeti napredne metode te što se događa u pozadini njihovog rada. Sada kada smo shvatili koncepte dohvaćanja podataka, pređimo na rad s njima te njihov upis i ispis iz baze podataka. Napraviti ćemo također jednu metodu koja će spremati vremensku prognozu u bazu. Ona će očistiti sve podatke iz baze tako da možemo spremiti one svježe tj. nove. Također, ona će domenski model vremenske prognoze pretvoriti u model baze podataka. Struktura je slična prethodnom primjeru, a ide ovako:

```
fun saveForecast(forecast: ForecastList) = forecastDbHelper.use {
    ...
}
```

Kao što je već rečeno, najprije ćemo očistiti podatke iz tablica pa potom spremiti nove. Da bismo olakšali taj dio sa čišćenjem. Možemo napraviti ekstenziju funkcije za *SQLiteDatabase* koja će okinuti potreban SQL upit za nas, ona izgleda ovako.

```
fun SQLiteDatabase.clear(tableName: String) {
    execSQL("delete from $tableName")
}
```

Isječak kôda 73: Ekstenzija funkcije za čišćenje baze podataka

Ova ekstenzija funkcije samo prima ime tablice te na principu tog imena briše sve iz istoimene tablice u bazi podataka. Pozivamo ju na sljedeći način.

```
clear(CityForecastTable.NAME)
clear(DayForecastTable.NAME)
```

Isječak kôda 74: Poziv ekstenzije za čišćenje podataka

Sada je vrijeme da pretvorimo podatke te taj rezultat spremimo u bazu pomoću *insert* upita. Pretvorba iz domenskog modela u onaj od baze podataka je jednosmjerna ulica pa ne možemo tu ništa pogriješiti, ali ovako bi izgledao jedan primjer toga.

```
fun convertFromDomain(forecast: ForecastList) = with(forecast) {
    val daily = dailyForecast.map { convertDayFromDomain(id, it) }
    CityForecast(id, city, country, daily)
}
```

Isječak kôda 75: Pretvaranje iz domenskog modela u model baze podataka

Nakon te konverzije, sve što trebamo je koristiti još jednu Anko funkciju za upisivanje podataka u bazu, ali ona za svoje parametre traži ime tablice u koju unosimo i podatke strukturirane tako da su tipa *Pair<String, Any>*, odnosno svakom podatku moramo dati i njegovo ime kako bi znala pod koji stupac ga spremiti unutar tablice. Tu pretvorbu možemo učiti na više načina, jedan od njih opet je koristeći ekstenzije, no nećemo ulaziti do te dubine. Nakon konverzije trebamo pozvati sljedeću funkciju gdje nam drugi parametar predstavlja pretvorene podatke:

```
insert(CityForecastTable.NAME, *map.toVarargArray())
```

Isječak kôda 76: Unos podataka u bazu

On upisuje novi redak u CityForecast tablicu. Isto to možemo koristiti za dnevne prognoze.

```
dailyForecast.forEach { insert(DayForecastTable.NAME,
*it.map.toVarargArray()) }
```

Isječak kôda 77: Unos liste podataka u bazu

Dakle, pomoću mapiranja uspjeli smo pretvoriti klase u registre baze podataka i obrnuto na vrlo jednostavan način, ali postoje još jednostavniji i bolji. Nakon što smo pripremili sve ekstenzije funkcija, možemo ih koristiti i u drugim projektima. Kako na kraju izgleda potpuni kôd za spremanje vremenske prognoze u bazu pogledajmo u nastavku.

```
fun saveForecast(forecast: ForecastList) = forecastDbHelper.use {

    clear(CityForecastTable.NAME)
    clear(DayForecastTable.NAME)

    with(dataMapper.convertFromDomain(forecast)) {
        insert(CityForecastTable.NAME, *map.toVarargArray())
        dailyForecast forEach {
            insert(DayForecastTable.NAME, *it.map.toVarargArray())
        }
    }
}
```

Isječak kôda 78: Funkcija za spremanje vremenske prognoze

Puno teorije o bazama podataka i o njihovom radu je obrađeno u ovom poglavlju te vjerojatno postoji mogućnost da neke stvari još nisu sjele na svoje mjesto. Čim počnemo primjenjivati ove stvari nad stvarnim projektom, shvatit ćemo zapravo koliko je sve to jednostavno i intuitivno. U nastavku rada upoznati ćemo se sa još puno drugih metoda i alata koji će nam rad na stvarnim projektima uvelike olakšati, posebno u dijelu razvoja i testiranja mobilnih aplikacija (Levia, 2017).

3.9. Null Safety

Null safety, *nullability* ili ništavost je značajka Kotlinovog sustava tipova koja nam pomaže izbjeći *NullPointerException* pogreške (eng. errors). Već smo ju spomenuli na početku ovog rada kao jednu od glavnih značajki zašto bi trebali koristiti Kotlin. Ako ste ikada koristili aplikacije, vjerojatno ste vidjeli poruku poput „Nažalost, dogodila se pogreška: *java.lang.NullPointerException*“, bez dodatnih detalja. Osim te pogreške postoji još puno njih sličnih, a većinom su posljedica iste pogreške te takve pogreške su problem i za korisnike i za programere. Pristup suvremenih jezika, uključujući i Kotlin je da se te pogreške pretvore iz pogrešaka u vrijeme izvršavanja (eng. runtime errors) u pogreške kompiliranja (eng. compile-time errors). Podržavajući ništavost kao dio topološkog sustava, prevoditelj može otkriti mnoge moguće pogreške tijekom kompilacije i smanjiti mogućnost bacanja iznimki za vrijeme izvođenja (Jemerov i Isakova, 2017).

Većina modernih jezika to pitanj rješava na neki svoj način pa tako i Kotlin. Kotlin koristi znak upitnika kako bi identificirao ništave tipove. Na taj način, ako varijabla može biti *null*, prevoditelj će nas prisiliti da se nosimo s njom. Kako je u Kotlinu sve objekt, sve može biti ništavo pa tako možemo imati da je i cijeli broj ništav.

```
val a: Int? = null
```

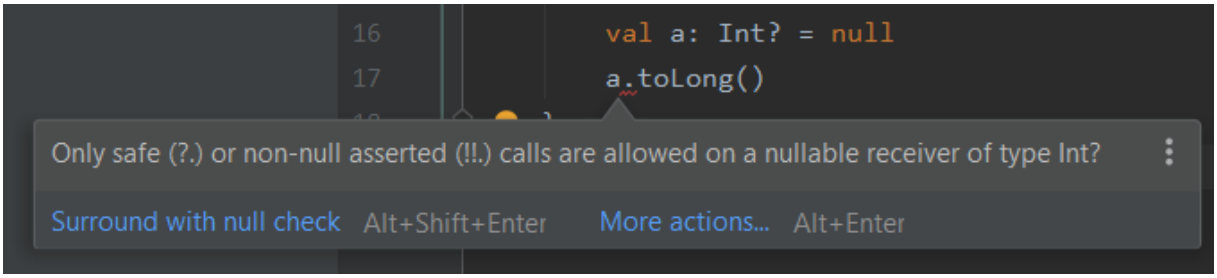
Isječak kôda 79: Ništavost Kotlinovog objekta

Ne možemo direktno raditi sa ništavim tipovima bez da prije toga odradimo nekoliko provjera. Primjerice, sljedeće linije kôda se neće kompilirati.

```
val a: Int? = null
a.toLong()
```

Isječak kôda 80: Neispravan kôd

Android Studio će nam pokazati sljedeću grešku.



Slika 5: Greška ništavog tipa

Ta bi varijabla mogla biti *null*, u našem slučaju zaista i jest, a kompajler je toga svjestan pa tako sve dok ništavost ne bude provjerena, nećemo ju moći koristiti. Ovdje nam je opet Kotlinov *smart cast* od velikog značaja, ako provjeravamo ništavost objekta, od tog trenutka automatski ga pretvara u njego *non-nullabe* tip, pogledajmo primjer.

```
val a: Int? = null
...
if (a != null) {
    a.toLong()
}
```

Isječak kôda 81: Automatsko pretvaranje ništavog tipa

Unutar *if* grananja, varijabla „a“ postaje *Int* umjesto *Int?*, pa zbog toga možemo ju koristiti bez provjeravanja njezine ništavosti ili pretvarajući ju u neki drugi tip. Kôd izvan *if* grananja dakako i dalje mora se nositi s tim.

Ovo radi samo onda kada varijabla ne može biti modificirana, jer inače vrijednost bi mogla biti zamijenjena od strane neke druge dretve i prethodna provjera bi bila netočna u tom trenutku. Podržano je nad *val* svojstvima ili lokalnim (var ili val) varijablama.

Da bi pojednostavili prijašnji kôd, možemo napisati nešto poput sljedećeg.

```
val a: Int? = null
...
a?.toLong()
```

Isječak kôda 82: Pojednostavljenje pretvaranja ništavog tipa

Ovdje koristimo poziv sigurnog operatora (eng. safe call operator, „?.“). Prethodna linija će samo biti izvršena ako varijabla nije *null*. Inače, neće napraviti ništa, kao da ju samo preskoči. Uz to možemo napisati i alternativu ukoliko vrijednost varijable bude *null* koristeći Elvisov operator(?:), pa to izgleda ovako.

```
val a: Int? = null
...
val myLong = a?.toLong() ?: 0L
```

Isječak kôda 83: Primjer *Elvisovog* operatora

Kako su *throw* i *return* također izrazi u Kotlinu i njih možemo koristit sa desne strane Elvis operatora. Bilo kako bilo, postoji puno situacija za koje smo sigurni da koristimo varijablu koja nije *null*, ali njezin tip je *nullable*. Možemo forsirati kompajler da se nosi s tim *nullable* tipovima preskakujući restrikcije koristeći „!!“ operator.

```
val a: Int? = null
a!!.toLong()
```

Isječak kôda 84: Preskakanje restrikcija koristeći „!!“ operator

Prethodni kôd će se kompilirati, ali očigledno će se i srušiti te iz tog razloga moramo biti sigurni da ovaj operator koristimo samo u specifičnim situacijama. Obično možemo birati alternativne solucije. Kôd koji je pun „!!“ operatora je znak da nešto nije napravljeno na pravi način (Levia, 2017).

4. Napredni Kotlin

Do sada smo predstavili i opisali osnove Kotlin programskog jezika te uz pomoć njega možemo napraviti osnovnu mobilnu aplikaciju. Također, trebali bi biti upoznati sa korištenjem Kotlina za pristup postojećem API-u. U ovom poglavlju, krenut ćemo sa nešto naprednijim metodama, počevši od nekih jednostavnijih koje smo već spomenuli pa prema sve složenijim i naprednijim. Saznati ćemo princip konvencija koje se koriste u Kotlinu za provođenje preopterećenja operatora i druge tehnike apstrakcije, poput delegiranih svojstava. Zatim slijedi poglavlja o anotacijama, Anko i ekstenzijama, o svima kojima smo već prije govorili i već ponešto znamo. Nakon toga, dotaknuti ćemo se lambda te ćemo vidjeti kako možemo proglasiti vlastite funkcije koje uzimaju lambde kao parametre. Na kraju dolaze već puno naprednije metode koje se veoma često koriste u stvarnim sustavima i aplikacijama te su u skladu sa vremenom pisanja ovoga rada.

4.1. Preopterećenje operatora

Ukoliko u primjeru naša klasa definira posebnu metodu koja se zove plus, prema dogovoru možemo koristiti operator „+“ na instancama ove klase te zbog toga prema Jemerov D. i Isakova S. ovu tehniku nazivamo konvencijama (eng. convention). U ovom poglavlju razmotriti ćemo različite konvencije koje podržava Kotlin i kako se one mogu koristiti.

Kotlin koristi princip konvencija, umjesto da se oslanja na tipove kao što to čini Java, jer programerima omogućuje prilagođavanje postojećih Java klasa zahtjevima Kotlin jezičnih značajki. Skup sučelja koje implementira klasa je fiksna, a Kotlin ne može mijenjati postojeću klasu tako da implementira dodatna sučelja. S druge strane, definiranje novih metoda za klasu moguće je kroz mehanizam funkcija proširenja. Možemo definirati bilo koje načine konvencije kao proširenja i na taj način prilagoditi bilo koju postojeću Java klasu bez izmjene njezinog kôda.

Najizravniji primjeri uporabe konvencija u Kotlinu su aritmetički operatori. U Javi se cijeli skup aritmetičkih operacija može koristiti samo s primitivnim tipovima, a dodatno se „+“ operator može koristiti s *String* vrijednostima. Na primjer, ako radimo sa brojevima putem *BigInteger* klase, puno lakše i bolje ih je zbrojiti pomoću znaka „+“ nego što eksplicitno pozivate metodu dodavanja. Za dodavanje elementa u kolekciju,

možda biste trebali koristiti operator „+ =“. Kotlin vam dopušta da to učinite, pogledajmo primjer.

```
data class Point(val x: Int, val y: Int) {
// definira operator funkcije "plus"
    operator fun plus(other: Point): Point {
//dodaje koordinate i vraća  novu   točku
         return Point(x + other.x, y + other.y)
 }
}

// implementacija klase
val p1 = Point(10, 20)
val p2 = Point(30, 40)
println(p1 + p2)       // poziv funkcije koristeći "+" znak
Point(x=40, y=60)
```

Isječak kôda 85: Dodavanje elemenata u kolekciju

U sljedećoj tablici možemo pronaći sve binarne aritmetičke operatore koji se mogu preopteretiti.

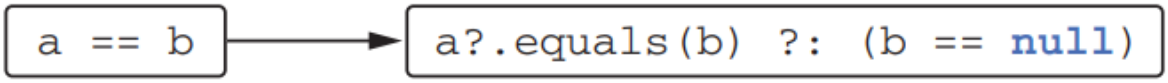
Tablica 5: Operatori koji se mogu preopteretiti

Ekspresija	**Ime funkcije**
a * b	Množenje
a / b	Dijeljenje
a % b	Mod(ostatak od cjelobrojnog dijeljenja)
a + b	Zbrajanje
a – b	Oduzimanje

Operatori za vlastite tipove uvijek koriste isti prioritet kao i standardni numerički tipovi. Na primjer, ako napišemo „a + b * c“, množenje će se uvijek izvršiti prije dodavanja, čak i ako ste sami definirali te operatore. Operatori *, / i % imaju isti prioritet, koji je viši od prioriteta operatora + i -.

Kao i kod aritmetičkih operatora, Kotlin nam omogućuje korištenje operatora usporedbe s bilo kojim objektom, a ne samo s primitivnim tipovima. Operatori usporedbe su ==, !=, <, >, itd. Umjesto da zovemo metodu *equals* ili *compareTo* kao u Javi, možemo izravno koristiti operatore usporedbe što je puno intuitivnije i sažetije. Možemo već zaključiti da koristeći operator „==“ zapravo pozivamo metodu *equals*. Također, operator „!=“ isto je preveden u poziv metode *equals* s očiglednom razlikom, da je rezultat zapravo invertiran, odnosno, suprotan od onog kojeg daje operator „==“.

Jedna veoma važna napomena je da operatori „==“ i „!=“ mogu biti korišteni sa ništavim (eng. Nullable) operatorima zato što oni u pozadini provjere je li određeni operator null,



Slika 6: Operator "==" u Kotlinu

a kako se operator „==“ zapravo prevodi u Kotlinu, pogledajmo na sljedećoj slici.

U Javi, klase mogu implementirati *Comparable* sučelje kako bi koristili algoritme za uspoređivanje vrijednosti, kao što je na primjer traženje maksimuma ili sortiranje. Metoda *compareTo* definirana u tom sučelju koristi se za utvrđivanje je li jedan objekt veći od drugoga, ali u Javi ne postoji kratka sintaksa pozivanja ove metode. Samo vrijednosti primitivnih tipova mogu se usporediti pomoću operatora „<“ i „>“, sve ostale vrste zahtijevaju da eksplicitno napišemo „element1.compareTo(element2)“. Kotlin podržava isto to *Comparable* sučelje, ali metoda *compareTo* definirana u tom sučelju može se nazvati konvencijom, a upotrebe operatora(<, >, <= i >=) prevode se u pozive *compareTo*, kao što je prikazanu u sljedećoj slici.

Slika 7: Operator ">=" u Kotlinu

Povratni tip metode *comapreTo* mora biti *Int.*. Svi ostali operatori usporedbe rade na identičan način.

Neke od najčešćih operacija za rad s kolekcijama su dobivanje i postavljanje elementa po indeksu, kao i provjera pripada li neki element kolekciji. Sve ove operacije podržane su pomoću sintaksa operatora, recimo da biste dobili ili postavili element po indeksu koristite sintaksu „a[b]“ koji se naziva indeks operator. Operator se može koristiti i za provjeru nalazi li se element u kolekciji ili rasponu. Već znamo da u Kotlinu možemo pristupiti elementima u *map* slično kao što pristupate nizovima u Javi - putem kvadratnih zagrada: `val value = map[key]`

Pomoću istog operatora možemo promijeniti vrijednost ključa u *map* koji se može promijeniti: `mutableMap[key] = newValue`

U Kotlinu je operator indeksa još jedna konvencija. Čitanje elementa pomoću indeksa operatora prevodi se u poziv metode *get*, a pisanje elementa postaje poziv metode *set*.

Jedan drugi operator koji podržava kolekcije je operator *in*, koji se koristi za provjeru pripada li neki objekt kolekciji. Odgovarajuća funkcija se zove *contains*.

Da bismo stvorili nekakav raspon, možemo koristiti sintaksu „ .. “, na primjer „1..10“ će stvoriti sve brojeve koji se nalaze između 1 i 10. Ovo smo već vidjeli, a ovdje ćemo se upoznati sa konvencijom koja pomaže da to stvorimo. Operator „ .. “ je sažet način pozivanja funkcije *rangeTo* koja vraća raspon, slika 8.

Slika 8: Operator ".."

Možemo sami definirati ovaj operator za našu vlastitu klasu, ali ako naša klasa implementira sučelje *Comparable* onda ne moramo to činiti. Možemo samo napraviti raspon bilo kojih usporedivih elemenata pomoću Kotlinove standardne biblioteke. Biblioteka definira funkciju *rangeTo* koja se može pozvati na bilo koji usporedivi element, ovo je jedan od mnogih primjera.

```
operator fun <T: Comparable<T>> T.rangeTo(that: T): ClosedRange<T>
```

Isječak kôda 86: Primjer *rangeTo* funkcije

Ova funkcija vraća raspon koji vam omogućuje provjeru pridaju li različiti elementi njoj (Jemerov i Isakova, 2017).

4.2. Anotacije

Do sada smo vidjeli mnoge značajke za rad s klasama i funkcijama, ali sve one zahtijevaju da navedete točna imena klasa i funkcija koje upotrebljavate kao dio izvornog kôda programa. Da biste pozvali funkciju, moramo znati klasu u kojoj je definirana, kao i njeno ime te vrste parametara. Anotacije nam daju snagu da nadiđete to i pišete kôd koji se bavi proizvoljnim klasama koje se ne znaju unaprijed. Anotacije možemo upotrebljavati za dodjeljivanje semantike specifične za biblioteku tim klasama. Primjena anotacija je jednostavna. Ali pisanje vlastitih anotacija, a posebno

pisanje kôda kojim se rukuje nije nimalo jednostavno. Sintaksa za korištenje anotacija potpuno je ista kao u Javi, dok je sintaksa za proglašavanje vlastitih klasa anotacija malo drugačija.

Većina modernih Java okvira (eng. framework) općenito koristi anotacije, tako da smo se sigurno susreli s njima tijekom rada na Java aplikacijama. Osnovni koncept u Kotlinu je isti. Anotacija nam omogućuje pridruživanje dodatnih metapodataka deklaraciji. Metapodacima se tada može pristupiti pomoću alata koji rade s izvornim kôdom, sa sastavljenim datotekama klase ili u vrijeme izvođenja, ovisno o konfiguraciji anotacije (Jemerov i Isakova, 2017).

4.2.1. Primjena anotacija

Anotacije u Kotlinu koristite se na isti način kao i u Javi. Da bismo primijenili anotaciju, na početak deklaracije anotacije stavimo njeno ime te znak „@“ stavimo ispred. Možemo anotirati različite elemente kôda, poput funkcija i klasa. Na primjer, ako koristimo okvir *JUnit*, možemo označiti metodu ispitivanja anotacijom @Test, o čemu ćemo pričati detaljnije u sljedećem poglavlju koje će se baviti testiranjem aplikacija, no sada pogledajmo kratki primjer kako bi to trebalo izgledati.

```
class MyTest {
    @Test fun testTrue() {     // @Test anotacija upućuje JUnit okvir
        Assert.assertTrue(true)// da ovu metodu pozove kao test
    }
}
```

Isječak kôda 87: Primjer anotacije

Anotacije mogu imati samo parametre sljedećih vrsta:

- primitivne vrste
- Strings
- enumi
- reference klasa
- ostale klase napomena i nizovi istih.

Sintaksa za specificiranje argumenata anotacija malo se razlikuje od Javinih:

- Da bismo odredili klasu kao argument anotacije, moramo staviti „::class“ nakon naziva klase
    - Primjer: `@MyAnnotation(MyClass::class)`

- Da bismo kao argument naveli drugu anotaciju, ne stavljamo znak „@“ prije naziva anotacije
- Da bismo odredili niz kao argument, koristimo funkciju *arrayOf*. Ako je klasa anotacija deklarirana u Javi, vrijednost naziva parametara automatski se pretvara u *vararg* parametar ako je potrebno, pa se argumenti mogu navesti i bez korištenja funkcije *arrayOf*
    - Primjer: `@RequestMapping(path = arrayOf(„/foo“, „/bar“))`

Argumenti anotacije moraju biti poznati u vrijeme kompiliranja, tako da arbitrarna svojstva ne možemo koristiti kao argumente. Da bismo koristili svojstvo kao argument anotacije, moramo ga označiti modifikatorom *const*, koji kompajleru kaže da je svojstvo konstanta u vrijeme kompiliranja. Ako pokušamo koristiti regularni entitet kao argument napomene, pojaviti će se pogreška „Only 'const val' can be used in constant expressions.'', što u prijevodu znači da samo konstantne varijable se mogu koristiti (Jemerov i Isakova, 2017).

4.2.2. Cilj anotacije

U mnogim slučajevima jedna deklaracija u izvornom kôdu Kotlina odgovara većem broju deklaracija u Javi i svaka od njih može nositi anotacije. Na primjer, Kotlin svojstvo (eng. property) odgovara Java polju, getteru, a možda i seteru i njegovom parametru. Svojstvo prijavljeno u primarnom konstruktoru ima još jedan odgovarajući element: parametar konstruktora. Stoga bi moglo biti potrebno odrediti koji od tih elemenata treba anotirati. Element koji treba anotirati odredimo odredištem cilja upotrebe na web mjestu. Cilj mjesta upotrebe nalazi se između znaka „@“ i imena anotacija i od imena je odvojen dvotočkom. U *JUnit* možemo odrediti pravilo koje se izvršava prije svake metode ispitivanja. Na primjer, standardno pravilo *TemporaryFolder* koristi se za stvaranje datoteka i mapa koje se brišu kada se metoda ispitivanja završi. Da bismo odredili pravilo, u Javi izjavljujete javno polje ili metodu označenu s @Rule. Ali ako samo označimo mapu svojstava u Kotlin testnom razredu s @Rule, dobit ćemo iznimku. To se događa jer se na polje primjenjuje @Rule, koje je prema zadanim postavkama privatno. Da biste ga primijenili na getter, to moramo eksplicitno napisati.

Ako anotiramo entitet anotacijom deklariranom u Javi, prema zadanom se primjenjuje na odgovarajuće polje. Kotlin vam također omogućuje da izjavite anotacije koje se mogu izravno primijeniti na svojstva.

Potpuni popis podržanih ciljeva upotrebe web mjesta je sljedeći:

- Svojstvo – Java anotacije se ne mogu primijeniti s ovim ciljem upotrebe web mjesta
- Polje (eng. field) – polje generirano za entitet
- Get – svojstvo *getter-a*
- Set – svojstvo *setter-a*
- Prijemnik(eng. receiver) – parametar primatelja funkcije proširenja ili svojstva
- Param – parametar konstruktora
- Setparam – parametar postavljanja svojstva
- Delegat – polje za pohranjivanje delegatske instance za delegirani entitet
- Datoteka – klasa koja sadrži funkcije i svojstva najviše razine deklarirane u datoteci

Anotacija s ciljem datoteke mora se postaviti na gornju razinu datoteke, prije direktive o paketu. Jedna od anotacija koja se obično primjenjuje na datoteke je @JvmName, koja mijenja naziv odgovarajuće klase. Još jedna česta anotacija je @Suppress koju možemo koristiti za suzbijanje određenog upozorenja prevoditelja u kontekstu izraza anotacije (Jemerov i Isakova, 2017).

4.2.3. Deklariranje anotacija

U ovom ćemo odjeljku naučiti kako proglasiti anotacije pomoću primjesa iz *JKida* kao primjera. Anotacija *@JsonExclude* ima najjednostavniji oblik, jer nema parametara: klasa anotacija *JsonExclude*. Sintaksa izgleda kao uobičajena deklaracija klase, s dodanim modifikatorom anotacija prije ključne riječi klase. Budući da se klase anotacija upotrebljavaju samo za definiranje strukture metapodataka povezanih s deklaracijama i izrazima, ne mogu sadržavati nijedan kôd. Stoga prevoditelj zabranjuje određivanje tijela za klasu anotacija. Za komentare koji imaju parametre, parametri su deklarirani u primarnom konstruktoru klase:

```
annotation class JsonName(val name: String)
```

Isječak kôda 88: Parametri anotirane klase

Koristimo redovitu sintaksu deklaracije primarnog konstruktora. Ključna riječ *val* obvezna je za sve parametre razreda anotacija. Za usporedbu, evo kako bismo istu naznaku naveli u Javi:

```
/* Java */
public @interface JsonName {
    String value();
}
```

Isječak kôda 89: Anotirana klasa u Javi

Obratimo pažnju na to kako Java anotacija ima metodu koja se naziva *value*, dok Kotlinova anotacija ima svojstvo *name*. *Value* metoda posebna je u Javi: kada primijenimo anotaciju, moramo navesti izričita imena za sve atribute koje navedemo osim *value*. S druge strane, u Kotlinu je primjena anotacija redoviti poziv konstruktora. Možemo upotrijebiti sintaksu imenovanog argumenta da bismo nazive argumenata učinili eksplicitnima ili ih možemo izostaviti: *@JsonName (name = "first_name")* znači isto što i *@JsonName ("first_name"),* jer je naziv prvi parametar konstruktor *JsonName*. Ako trebamo primijeniti anotaciju deklariranu u Javi na Kotlin element, morat ćemo koristiti sintaksu imenovanog argumenta za sve argumente osim vrijednosti, koje Kotlin također prepoznaje kao posebne (Jemerov i Isakova, 2017).

4.3. Anko i ekstenzije

Tim koji je razvijao Kotlin programski jezik također su razvili i neke sjajne alate koji olakšavaju razvoj i testiranje mobilnih aplikacija u programskom jeziku Kotlin. U ovom poglavlju pokazati ćemo što su ti alati i kako ih možemo započeti koristiti.

4.3.1. Anko

Prije nego što uđemo dublje u ovaj alat, želio bi napomenuti da u vrijeme pisanja ovog rada Anko je zastario(eng. deprecated) te se više ne preporučuje njegovo korištenje, ali njega objašnjavam zato da bi shvatili moć Kotlina kao programskog jezika, no više o svemu u nastavku ovog poglavlja.

Anko je moćna biblioteka koju je razvio JetBrains, neke djelomične funkcionalnosti već smo mogli pronaći u ovom radu. Omogućuje razvoj aplikacija za Android operacijski sustav bržim i lakšim tako da čini naš kôd čistim i lakim za čitanje. Njegova glavna svrha je generiranje izgleda sučelja pomoću kôda umjesto XML-a. Međutim, to nije jedina značajka koju možemo dobiti iz ove biblioteke. Anko uključuje puno izuzetno korisnih funkcija i svojstava koja će nam pomoći u izbjegavanju *boilerplate* kôda. Već smo vidjeli puno primjera u ovom radu gdje ga koristimo, ali vidjeti

ćemo još mnogo toga u nastavku. Budući da je Anko biblioteka napisana posebno za Android, u svakom trenutku možemo doći do njegovog izvornog kôda koristeći prečac (tipka ctrl + klik miša na funkciju) kako bi mogli lakše shvatiti što se događa iza kulisa. Anko-implementacija je izvrstan primjer za učenje korisnih načina kako najbolje iskoristit Kotlin programski jezik (Levia, 2017; GitHub.com, 2020).

Kad god koristimo nešto iz Anko-a, to će uključiti uvoz s nazivom entiteta ili funkcije u datoteku. To je iz razloga što Anko koristi proširenja za dodavanje novih značajki Androidovom okviru. Anko nam može pomoći da, među ostalim, pojednostavnimo izradu namjera, navigaciju između aktivnosti, stvaranje fragmenata, pristup bazi podataka, stvaranje upozorenja, itd. Pronaći ćemo još puno zanimljivih primjera kada ga počnemo koristiti.

Anko se sastoji od nekoliko dijelova:

- Anko Commons – lagana biblioteka puna pomagača za namjere, dijaloga, evidentiranje i slično
- Anko Layouts – brz i siguran način pisanja dinamičkih izgleda za Android
- Anko SQLite – zbirka upita DSL i parsera za Android SQLite
- Anko Coroutines – alati na temelju kotinx.couroutines biblioteke

Iako je Anko dugo godina bio poprilično popularan među Kotlinovim korisnicima, ipak iskustvo korištenja istog nije bilo 100% savršeno. Donedavno su Android View API-ji bili vrlo optimizirani za inflaciju, a ponekad nije bilo moguće programsko postaviti neke atribute. Kao rezultat toga, DSL se morao oslanjati na zaobilaznice. Također nije bilo trivijalno oponašati reflektirajući pristup učitavanja *widgeta* potreban za podršku *AppCompat*u. Nije imao dovoljno resursa da na vrijeme riješimo sve rubne slučajeve. Međutim, stvari su se bitno promijenile u posljednjih nekoliko godina. Kada je Google službeno podržao Kotlin te kasnije kada je čak učinio Kotlin preferiranim jezikom za razvoj aplikacija za Android, zahvaljujući njegovom *JetPacku*, opsežnom skupu biblioteka, grubi rubovi SDK-a su izglađeni i tu Anko gubi svoj smisao. Na kraju pročitajte što su rekli njegovi osnivači kada su služeno izjavili da Anko ide u zastaru (GitHub.com, 2020)

„Anko is a successful project, and it has played its role in establishing a better Android developer experience with Kotlin. However, there are modern alternatives today, and we feel it's time to say goodbye to Anko.“ [7b]

Neke alternative za korištenje koju sugeriraju njegovi vlasnici su:

- Za Layout DSL
    - Jetpack Compose – reaktivni DSL za Kotlin kojeg podržava Google
    - Splitties – pregledi DSL-a, DSL koji se može proširivati nalikuje Anku
- Za generičke usluge
    - Android KTX – skup proširenja Kotlina za različite svrhe, također ga podržava Google
    - Splitties – Puno mikro biblioteka za sve prigode
- SQLite pomagači
    - Room – okvir temeljen na anotacijama za pristup bazi podataka, također podržan od strane Google-a
    - SQLDelight – sigurnosni API za SQL upite

4.3.2. Ekstenzije

Ekstenzije ili funkcije proširenja su funkcije koje dodaju novo ponašanje klasi, čak i ako nemamo pristup izvornom kôdu te klase. To je način za proširivanje klasa kojima nedostaju neke korisne funkcije. U Javi se to obično provodi u uslužnim klasama koje uključuju skup statičnih metoda. Prednost upotrebe ekstenzija u Kotlinu je da ne trebamo proslijediti objekt kao argument. Ekstenzija djeluje kao dio klase i možemo ju implementirati koristeći nju i sve njezine javne metode. Na primjer, možemo napraviti *toast* funkciju koja ne treba kontekst kao parametar, a koju bi mogli koristiti svi *Context* objekti ili oni koji proširuju(eng. extends) *Context* klasu, kao što su *Activity* ili *Service*, pogledajmo primjer ispod (Levia, 2017).

```
fun Context.toast(message: CharSequence, duration: Int =
Toast.LENGTH_SHORT) {
    Toast.makeText(this, message, duration).show()
}
```

Isječak kôda 90: Primjer ekstenzije u Kotlinu

Ova funkcija bi se mogla koristi unutar aktivnosti, na primjer ovako.

```
toast("Hello world!")
toast("Hello world!", Toast.LENGTH_LONG)
```

Isječak kôda 91: Poziv ekstenzije

Naravno da Anko već uključuje vlastitu funkciju proširenja *toast* metode, vrlo sličnu ovoj. Biblioteka pruža funkcije i za *CharSequence* i za resurse te različita imena

za duge i krake *toast* poruke. Ekstenzije također mogu biti i svojstva. Tako možemo stvoriti ekstenzije svojstava na sličan način.

Ekstenzije zapravo ne mijenjaju originalnu klasu, ali funkcija se dodaje kao statički uvoz tamo gdje se koristi. Ekstenzije se mogu deklarirati u bilo kojoj datoteci, tako da je uobičajena praksa stvaranje datoteka koje uključuju skup povezanih funkcija, a to je čar iza mnogo Ankovih karakteristika pa od sada možemo početi koristit sve njezine prednosti (Levia, 2017).

4.4. Lambda

Lambda izrazi, ili jednostavnije lambde, su jednostavan način definiranja anonimne funkcije. Vrlo su korisne jer nas sprječavaju da specifikaciju funkcije moramo zapisati u apstraktnu klasu ili sučelje, a zatim i implementaciju klase. U Kotlinu su lambde prvoklasni građani, što znači da se funkcija ponaša kao tip, pa se može proslijediti kao argument drugoj funkciji, funkcija može vratiti, spremiti u varijablu ili svojstvo (Leiva A., 2017).

Pomoću lambda možemo lako izvući uobičajene strukture kôdova u funkcije biblioteka. Jedna od najčešćih upotreba lambda je rad s kolekcijama, što ste već mogli prije vidjeti, ali također vidjeti ćemo još puno toga u nastavku rada (Jemerov D. i Isakova S., 2017).

Uvođenje lambda u Java 8 bila je jedna od dugo očekivanih promjena u evoluciji programskog jezika. Zašto je to toliko očekivano iz zbog koje razloga, otkriti ćemo u ovom dijelu rada te također zašto su lambde tako korisne i kako izgleda sintaksa lambda izraza u Kotlinu. Pronalaženje i spremanje dijelova ponašanja u našem kôdu veoma je čest zadatak. S lambda izrazima ovo je veoma jednostavno napraviti, a kôd je veoma sažet. Ne trebamo deklarirati funkciju, umjesto toga, učinkovito možemo proslijediti blok kôda izravno kao funkcijski parametar, pogledajmo sljedeći primjer kako bi lakše shvatili. Zamislite da trebamo definirati ponašanje za klik na gumb. Dodajemo slušatelja koji je odgovoran za rukovanje klikom. Ovaj slušatelj implementirati odgovarajuće *OnClickListener* sučelje s jednom metodom, *onClick.*

```
/* Java */
    button.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View view) {
/* Akcija na klik */
```

```
        }
    });
```

Isječak kôda 92: Lambda izraz u Javi

Verbalizam potreban za proglašavanje anonimne unutarnje klase postaje iritantan ako se ponavlja više puta. Oznaka kojom se izražava samo ponašanje, odnosno, ono što treba učiniti kada korisnik „klikne“, pomaže u uklanjaju suvišnog kôda. U Kotlinu, kao i u Javi 8, možemo koristiti lambde: `button.setOnClickListener { /* akcija na klik */ }`

Ovaj Kotlin kôd čini istu stvar kao anonimna klasa u Javi, ali kôd je sažetiji i čitljiviji. Ovdje smo mogli vidjeti kako se lambda može koristiti kao alternativa anonimnom objektu putem jedne metode. U nastavku ćemo objasniti najčešću upotrebu lambda, rad s kolekcijama (Levia, 2017).

4.4.1. Rad s kolekcijama uz pomoć lambde

Jedno od glavnih načela dobrog stila programiranja jest izbjegavanje dupliciranja u našem kôdu. Većina zadataka koje obavljamo sa kolekcijama slijedi nekoliko uobičajenih obrazaca, tako da kôd koji ih provodi treba biti sadržan u bibliotekama. Ali bez lambda je teško osigurati dobru, prikladnu biblioteku za rad s kolekcijama. Dakle, ako smo svoj kôd napisali u Javi (prije Java 8), najvjerojatnije imamo naviku implementirati sve sami. Ta se navika mora mijenjati s Kotlinom. Pretpostavimo da imamo popis ljudi te unutar njega moramo pronaći najstarije osobe. Ako nismo imali iskustva s lambdama, sve ćemo morati ručno implementirati. Uz dovoljno iskustva, takve petlje možemo napisati poprilično brzo, ali puno kôda se piše te rizik od pogreške je veoma velik. Na primjer, možemo dobiti pogrešku usporedbe i pronaći minimalni element umjesto maksimalnog. U Kotlinu, postoji puno bolji način, a on se sastoji od korištenja funkcije iz biblioteke kao što je prikazano u nastavku. (Levia, 2017)

```
val people = listOf(Person("Alice", 29), Person("Bob", 31))
println(people.maxBy { it.age })// traži maksimum uspoređujući godine
Person(name=Bob, age=31)
```

Isječak kôda 93: Rad s kolekcijama uz pomoć lambde

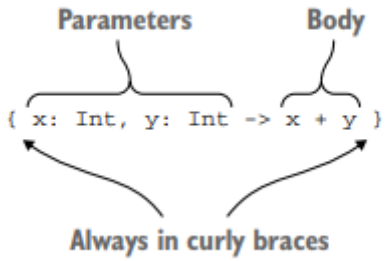
Funkcija *maxBy* može se pozvati u bilo koju kolekciju i uzima jedan argument, funkcija koja određuje koje vrijednosti treba usporediti da bi se pronašao maksimalni element.

Kôd u vitičastim zagradama je lambda izraz koji implementira tu logiku. Dobiva element kolekcije kao argument i vraća vrijednost za usporedbu. U ovom primjeru element kolekcije je *Person* objekt, a vrijednost za usporedbu je njegova dob (eng. age), pohranjena u *age* svojstvu. Ako se lambda samo delegira na funkciju ili svojstvo, možemo ju zamijeniti referencom člana: `people.maxBy(Person::age)`.

Ovaj kôd znači istu stvar kao i prethodno napisani kôd. Većinu stvari koju radimo sa kolekcijama u Javi može se puno bolje izraziti funkcijama iz biblioteka koristeći lambde ili članske reference. Krajnji kôd mnogo je kraći i lakši za razumijevanje (Levia, 2017).

4.4.2. Sintaksa lambda izraza

Kao što smo spomenuli, lambda šifrira mali djelić ponašanja koje možemo prenijeti kao vrijednost. Može se deklarirati neovisno i pohraniti u varijablu, ali češće se deklarira izravno kad se proslijedi u funkciju. Na slici 9 prikazana je sintaksa za deklariranje lambda izraza.



Slika 9: Sintaksa lambda izrada

Lambda izraz u Kotlinu uvijek je okružen vitičastim zagradama. Imajmo na umu da nema zagrada oko argumenata. Strelica odvaja popis argumenata od tijela lambde. Možemo pohraniti lambda izraz u varijablu, a zatim tretirati ovu varijablu kao normalnu funkciju.

```
val sum = { x: Int, y: Int -> x+y}
println(sum(1, 2))
// PRINT
3
```

Isječak kôda 94: Pohranjivanje lambda izraza u varijablu

Ako želimo možemo i izravno pozvati lambda izraz, ovako bismo to učinili `{println(42)}()`, ali takva sintaksa nije čitljiva i nema mnogo smisla. Ako moramo u blok staviti komad kôda, možemo upotrijebiti funkciju *run* koja izvršava lambda izraz koji joj je proslijeđen, primjer naveden ispod.

```
run { println(42) }
// PRINT
42
```

Isječak kôda 95: Primjer *run* funkcije

U prethodnom poglavlju pokazali smo primjer kako možemo sortirati osobe prema njihovoj dobi, ako napišemo taj primjer bez upotrebe sintaktičkih prečaca, dobiti ćemo sljedeće `people.maxBy({ p: Person -> p.age })`. Trebalo bi biti jasno što se ovdje događa, komad kôda u vitičastim zagradama je lambda izraz, a mi ga prenosimo kao argument funkciji. Lambda izraz uzima jedan argument tipa *Person* i vraća njegovu dob. Prvo, postoji previše interpunkcijskih znakova, što šteti čitljivosti. Drugo, vrsta se može zaključiti iz konteksta i stoga izostaviti. Na kraju, ne trebamo dodijeliti ime argumentu lambda u ovom slučaju. Napravimo ta poboljšanja, počevši od zagrada. U Kotlinu, sintaktička konvencija omogućuje vam da pomaknete lambda izraz iz zagrada ako je to posljednji argument u pozivu funkcije. U ovom primjeru, lambda je jedini argument, pa se može staviti iza zagrada te dobijemo `people.maxBy() { p: Person -> p.age }`. Ako je lambda jedini argument funkcije, iz poziva možemo ukloniti i prazne zagrade onda dobijemo `people.maxBy { p: Person -> p.age }`.

Sva tri sintaktička oblika znače istu stvar, ali zadnji je najlakši za čitanje. Ako je lambda jedini argument, definitivno ćemo ju htjeti napisati bez zagrade. Kad imamo nekoliko argumenata, možemo naglasiti da je lambda argument tako da je ostavite u zagradama ili ih možemo staviti izvan njih, obje su opcije važeće.
Nastavimo s pojednostavljivanjem sintakse i riješimo se vrste parametara.

```
people.maxBy { p: Person -> p.age } // Tip parametra eksplicitno
napisan
people.maxBy { p -> p.age }   // Tip parametra zaključen
```

Isječak kôda 96: Pojednostavljivanje sintakse lambde

Kao i kod lokalnih varijabli, ako se može zaključiti vrsta lambda parametra, ne moramo ga eksplicitno određivati. Posljednje pojednostavljenje koje možemo napraviti u ovom primjeru je zamjena parametra sa zadanim nazivom parametra: *it*. To se ime

generira ako kontekst očekuje lambda sa samo jednim argumentom i može se zaključiti o njegovoj vrsti: `people.maxBy { it.age }`.

Ovo zadano ime generira se samo ako izričito ne navedete naziv argumenta. Do sada smo vidjeli samo primjere s lambdama koje se sastoje od jednog izraza ili izjave. Ali lambde nisu ograničene na tako malu veličinu i mogu sadržavati više iskaza (Jemerov i Isakova, 2017).

4.5. Napredni alati i tehnike

4.5.1. Coroutines

Coroutines su bile najveći uvod u Kotlin 1.1. Stvarno su sjajne zbog toga što su moćne, a zajednica još uvijek otkriva kako izvući maksimum iz njih. Jednostavno rečeno, *coroutines* su postupci načina pisanja asinkrona kôda uzastopno. Kako ne bi previše vremena potrošili na povratne veze, možemo pisati retke kôda jedan za drugim. Neki od njih će moći obustaviti izvršenje i pričekati dok rezultat ne bude dostupan. Ako imamo nekih znanja iz C# programskog jezika, onda ih lako možemo povezati sa C# *await/async* konceptom, no postupci u Kotlinu su snažniji jer umjesto da su konkretna implementacija ideje, oni su jezična značajka koja se može implementirati na različite načine za rješavanje različitih problema. Možemo napisati vlastitu implementaciju ili koristiti neku od nekoliko opcija koje je sastavio Kotlin tim i ostali neovisni programeri. Ovo poglavlje i nastavak ovog rada možemo uzeti kao primjer što se sve može učiniti s *coroutinama*, ali nije nužno i pravilo jer se one veoma brzo mijenjaju, a dok se ovaj rad bude čitalo, moguće da će dio toga već biti promijenjeno.

Cilj ovog poglavlja je da steknemo neke osnovne koncepte i upotrijebimo jednu od postojećih biblioteka, a ne da gradimo vlastite implementacije. Također, smatra se da je važno razumjeti stvari koje se događaju u pozadini, kako ne bi slijepo koristili ono što nam je dano.

Coroutines se temelje na ideji obustave funkcija, odnosno, funkcije koje mogu zaustaviti izvršenje kad se pozivaju i natjerati funkciju da nastavi nakon što završi sa pozadinskim zadatkom. Funkcije obustave označene su rezerviranom riječi *suspend* i može se pozvati samo unutar ostalih funkcija obustave ili unutar programa. To znači

da ne možemo svugdje pozvati funkciju zaustavljanja. Za to treba postojati okolna funkcija koja gradi program i pruža potrebni kontekst za to.

Coroutine nisu baš dretve, ali djeluju na sličan način koji je puno lakši i učinkovitiji. Možemo imati milijune koraka koji se prikazuju na nekoliko dretvi, što otvara novi svijet mogućnosti. Prema Levia A., postoje tri načina na koje možemo koristiti ovu značajku:

- Sirova implementacija (eng. raw implementation) – znači izgraditi vlastiti način korištenja postupaka. To je poprilično složeno i obično se uopće ne zahtjeva. Primjer ove implementacije pogledajmo u nastavku.

```
suspend fun login(request: LoginRequest): FirebaseUser =
    suspendCoroutine { continuation ->
 firebaseAuth.signInWithEmailAndPassword(request.email,request.password)
            .addOnCompleteListener { task ->
                if (task.isSuccessful && firebaseAuth.currentUser!=null)
{
                    val currentUser = firebaseAuth.currentUser
                    continuation.resume(currentUser!!)
                } else {
                    continuation.resumeWithException(task.exception ?:
                    AppError.Unknown)
                }
            }
    }
```

Isječak kôda 97: Primjer sirove implementacije *coroutina*

- Implementacija na niskoj razini (eng. low-level implementation) – Kotlin pruža skup biblioteka koje možemo naći u kotlinx.coroutines[12] repozitoriju, koji rješavaju neke najteže dijelove i pružaju specifičnu implementaciju za različite scenarije. Primjer niske razine implementacije pogledajmo na ovoj fusnoti[13].
- Implementacije na višoj razini (eng. higher-level implementation) – ako samo želimo imati rješenje koje pruža sve što je potrebno za početak korištenja, postoji nekoliko biblioteka koje obavljaju sav težak posao za nas, a njihov popis neprestano raste, pogledajmo primjer u nastavku.

```
async(UI) {
    val r1: Deferred<Result> = bg { fetchResult1() }
    val r2: Deferred<Result> = bg { fetchResult2() }
```

[12] https://github.com/Kotlin/kotlinx.coroutines
[13] Primjer implementacije Coroutine na niskoj razini:
https://github.com/Kotlin/kotlinx.coroutines/tree/master/ui/kotlinx-coroutines-android

```
    updateUI(r1.await(), r2.await())
}
```

Isječak kôda 98: Primjer implementacije *coroutina* na višoj razini

4.5.2. Data Binding

Data binding ili kada bi ga preveli na hrvatski, povezivanje podataka u računalnom programiranju je opća tehnika koja povezuje izvore podataka davatelja i potrošača zajedno te ih sinkronizira. To se obično izvodi s dva izvora podataka/informacija s različitim jezicima kao u povezivanju XML podataka i povezivanju podatka u korisničkom sučelju. U povezivanju podataka u korisničkom sučelju, podaci i informacijski objekti istoga jezika, ali različite funkcije su povezani zajedno, na primjer elementi Java UI na Java objekte.

U procesu povezivanja podataka svaka se promjena podataka automatski odražava na elemente koji su vezani za te podatke. Izraz vezivanje podataka također se koristi u slučajevima kada se vanjski prikaz podataka u elementu mijenja, a temeljni se podaci automatski ažuriraju kako bi odrazio ovu promjenu, primjerice promjena u *TextBox* elementu može izmijeniti temeljnu vrijednost podataka.

Android nudi podršku za pisanje rasporeda elemenata korisničkog sučelja (eng. layout) pomoću povezivanja podataka. To nam umanjuje potrebni kôd u logici naše aplikacije za povezivanje s elementima korisničkog sučelja. Korištenje *data bindinga* zahtijeva promjene u rasporedu elemenata korisničkog sučelja. Takvi rasporedi elemenata korisničkog sučelja počinju s korijenima izgleda, nakon čega slijedi podatkovni element i element korijena vlasničkog pregleda. Elementi podataka opisuju podatke koji su dostupni za vezivanje. Ovaj element pogleda sadrži našu korijensku hijerarhiju sličnu datotekama izgleda koji se ne koriste s povezivanjem podataka. Upućivanja na podatkovne elemente ili izraze unutar izgleda pišu se svojstvima atributa pomoću „@{}“ ili „@={}“.

Na primjer, kôd ispod poziva *findViewById()* metodu da bi pronašao *TextView* i povezao ga sa svojstvom *userName* od *viewModel* varijable.

```
findViewById<TextView>(R.id.sample_text).apply {
    text = viewModel.userName
}
```

Isječak kôda 99: Pronalazak *textView* elementa pomoću *findViewById()* metode

Sljedeći primjer prikazuje kako se pomoću biblioteke povezivanja podataka dodjeljuje tekst *widgetu* izravno u datoteci izgleda. Ovo uklanja potrebu za pozivanjem bilo kojeg Java kôda prikazanog gore. Imajmo na umu upotrebu sintakse „@{}“ u izrazu dodjele.

```
<TextView
android:text="@{viewmodel.userName}" />
```

Isječak kôda 100: Povezivanje *textView* elementa sa *data bindingom*

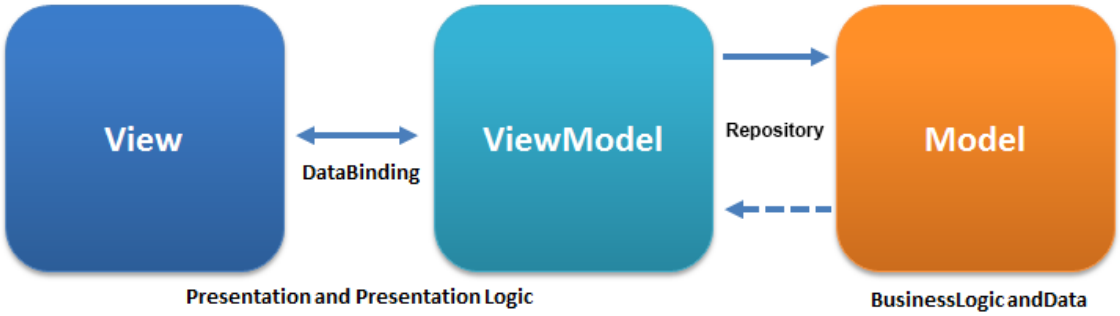
Vezivanje komponenata u datoteci izgleda omogućava nam uklanjanje mnogih okvira UI poziva u svojim aktivnostima, čineći ih jednostavnijim i lakšim za održavanje. To također može poboljšati performanse naše aplikacije i pomoći u sprječavanju curenja memorije i dovođenja do *null pointer* iznimki (Developers.android.com, 2020; Vogella.com, 2019).

4.5.3. MVVM

MVVM kratica je od „Model View ViewModel“ te je on uzorak dizajna arhitekture koji olakšava odvajanje razvoja grafičkog korisničkog sučelja od razvoja poslovne logike ili obrnuto.

Ovo je jedan od najboljih uzoraka dizajna koji se mogu koristiti za razvoj aplikacije za Android operacijski sustav, ali ono što čini MVVM moćniji jesu komponente iz *Android Jetpacka*, o kojima ćemo pričati u poglavlju 6, gdje će biti prikazan MVVM nad stvarnim projektom. Kao što mu i sam naziva kaže MVVM se sastoji od tri komponente koje su prikazane na slici 10, a one su:

1. Model – sadrži sve klase podataka, klase za baze podataka, API i repozitorije
2. View – je dio UI koji predstavlja trenutno stanje informacija koje je vidljivo korisniku
3. ViewModel – sadrži podatke potrebne u *View* i prevodi podatke pohranjene u *Model* koji mogu biti prisutni u *View*. ViewModel i View povezani su putem *data* bindinga i *Live Data*



Slika 10: Dijelovi MVVM arhitekture

Prednosti MVVM-a:

- Odvaja poslovnu logiku pomoću logike prezentacije
- *View* nije svjestan svih izračuna koji se događaju iza scene, što čini *ViewModel* ponovno iskoristivim
- Kada koristimo repozitorij između *ViewModela* i *Modela*, *ViewModel* zna samo kako podnijeti zahtjev za podacima, a repozitorij je taj koji će se brinuti o načinu preuzimanja podataka, bilo da su iz API-ja ili lokalne baze podataka.
- Budući da koristimo *ViewModel* i *LiveData* koji su svjesni životnog ciklusa aktivnosti, to će dovesti do manje rušenja i curenja memorije

Primjer jedne vrlo jednostavne MVVM arhitekture koja je zadužena za ispis korisničkih podataka na ekranu koji također još sadrži i jedan gumb pomoću kojeg se odjavljujemo iz aplikacije. Sastoji se od jednog modela *User, viewModel* klase *ProfileViewModel* te od jednog pogleda, odnosno *view* naziva *layout_profile.*

Ovako izgleda model.

```
data class DBUser(
    val firstName: String,
    val lastName: String,
    @PrimaryKey
    val email: String
)
```

Isječak kôda 101: Podatkovna klasa *DBUser*

Primjer jedne vrlo jednostavne *viewModel* klase, a ona izgleda ovako.

```
class ProfileViewModel(
    private val repository: ProfileRepository,
    private val authRepository: AuthRepository
){
```

```
    val user = repository.user
    val newPassword = MutableLiveData<String>()

    fun logout(removeDestination: Int) {
        authRepository.logout()
    }

    private fun changePassword(newPassword: String) {
        suspendCall {
            repository.changePassword(newPassword)
        }
    }

    fun onClick(view: View) {
        when (view.id) {
            R.id.btnChangePassword -> {
                changePassword(newPassword.value!!)
            }
        }
    }
}
```

Isječak kôda 102: Jednostavna *viewModel* klasa

I na kraju, njegov pripadajući ekran, odnosno *view* izgleda ovako.

```
<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/colorWhite"
    android:padding="@dimen/padding_default"
    tools:context=".ui.fragments.ProfileFragment">

    <com.google.android.material.textfield.TextInputEditText
        style="@style/InputEditText"
        android:drawableStart="@drawable/ic_user"
        android:drawablePadding="@dimen/size16"
        android:focusable="false"
        android:text="@{viewModel.user.firstName + ' ' +
viewModel.user.lastName}"/>

    <com.google.android.material.textfield.TextInputEditText
        style="@style/InputEditText"
        android:drawableStart="@drawable/ic_mail"
        android:drawablePadding="@dimen/size16"
        android:focusable="false"
        android:text="@{viewModel.user.email}"/>

    <com.google.android.material.textfield.TextInputEditText
        style="@style/InputEditText"
        android:hint="@string/enter_new_password"
        android:text= "@={viewModel.newPassword}"
        android:drawableStart="@drawable/ic_key"
        android:drawableTint="@color/colorPrimary"
        android:drawablePadding="@dimen/size16"
        android:inputType="textPassword"/>
</com.google.android.material.textfield.TextInputLayout>

<androidx.appcompat.widget.AppCompatButton
        android:id="@+id/btnChangePassword"
```

```
            style="@style/blueButton"
            android:onClick="@{(v) -> viewModel.onClick(v)}"
            android:background="@drawable/background_button_gray"
            android:enabled = "false"
            android:text="@string/save_changes"
            android:layout_marginBottom="@dimen/size8"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintBottom_toBottomOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Isječak kôda 103: XML *layout view* elementa

4.5.4. Naknadno dodavanje ovisnosti (Koin)

U softverskom inženjerstvu naknadno dodavanje ovisnosti (eng. dependency injection) je tehnika programiranja koja klasu čini neovisnom o njezinim ovisnostima (eng. dependencies). To postiže razdvajanjem upotrebe predmeta od njegovog stvaranja.

Klase često zahtijevaju reference na druge klase. Na primjer, klasa *Car* možda će trebati referencu na klasu *Engine*, njihova potreba naziva se ovisnost, a u ovom primjeru klasa *Car* ovisi o postojanju instance klase *Engine*. Postoje tri načina da klasa dobije objekt koji joj je potreban:

1. Klasa konstruira ovisnost koja joj je potrebna. U gornjem primjeru *Car* bi stvorio i inicijalizirao vlastiti primjerak *Enginea*.
2. Dohvati je s nekog drugog mjesta. Neki Android API-ji, poput *Context getteri* i *getSystemService()*, rade na ovaj način.
3. Neka se isporuči kao parametar. Aplikacija može pružiti ove ovisnosti kada je klasa konstruirana ili ih proslijediti funkcijama koje trebaju svaku ovisnost. U gornjem primjeru, konstruktor automobila primio bi *Engine* kao parametar.

Treća opcija je naknadno dodavanje ovisnosti! Ovim pristupom uzimamo ovisnosti klase i pružamo ih umjesto da ih instanca klase sama dobije.

Dependency injection je jedan oblik šire tehnike inverzije kontrole. Klijent koji želi nazivati neke usluge ne bi trebao znati kako konstruirati te usluge. Umjesto toga, klijent delegira odgovornost pružanja svojih usluga vanjskom kôdu (injektoru). Klijent ne smije pozivati kôd injektora, injektor je taj koji konstruira usluge. Injektor tada ubrizgava (prosljeđuje) usluge klijentu koje mogu već postojati ili ih injektori također mogu konstruirati. Klijent tada koristi usluge, to znači da klijent ne treba znati o

injektoru, kako konstruirati usluge ili čak i koje stvarne usluge koristi. Klijent mora znati samo unutrašnja sučelja usluga jer oni definiraju kako klijent može koristiti usluge. To razdvaja odgovornost „uporabe“ od odgovornosti za „izgradnju“.

Postoje dva glavna načina naknadnog dodavanja ovisnosti u Androidu:

- **Injektiranje konstruktora** (eng. Constructor Injection) – ovisnost klase prosljeđujemo njezinu konstruktoru
- **Injektiranje polja** (eng. Field injection) – Sustav instancira određene klase Android okvira (eng. framwork), poput aktivnosti i fragmenata, pa ubrizgavanje konstruktora nije moguće. S injektiranjem polja, ovisnosti se pokreću nakon stvaranja klase, a kôd možemo pogledati u nastavku.

```
class Car {
    lateinit var engine: Engine

    fun start() {
        engine.start()
    }
}

fun main(args: Array) {
    val car = Car()
    car.engine = Engine()
    car.start()
}
```

Isječak kôda 104: Primjer naknadne dodavanje ovisnosti

Sigurno se sada pitate zašto je naveden *Koin* u naslovu, ali ubrzo ćemo to objasniti. *Koin* je pragmatičan lagani okvir za *dependency injection* posebno napravljen za Kotlin programere. Napisan u čistom Kotlinu, samo pomoću funkcionalne rezolucije, laički rečeno to znači da nema proxyja, nema stvaranja kôda, nema refleksije. *Koin* je DSL, lagani spremnik i pragmatični API. Kako bi uz pomoć *Koina* naknadno dodavali ovisnosti prvo moramo deklarirati modul, a to izgleda ovako.

```
// Dodane neke klase
class Controller(val service : BusinessService)
class BusinessService()

// deklariranje modula
val myModule = module {
    single { Controller(get()) }
    single { BusinessService() }
}
```

Isječak kôda 105: Deklariranje modula za *Koin*

Isti taj modul moramo i pokrenuti unutar aplikacije koristeći *startKoin* funkciju, pogledajmo u nastavku.

```
class MyApplication : Application() {
    override fun onCreate(){
        super.onCreate()
        // pokreni Koin!
        startKoin {
            // deklariranje Android konteksta
            androidContext(this@MyApplication)
            // deklariranje modula
            modules(myModule)
        }
    }
}
```

Isječak kôda 106: Pokretanje *Koina* sa metodom *startKoin()*

I na kraju potrebno je samo još provesti naknadno dodavanje ovisnosti unutar Android klasa.

```
// Samo injektiraj u jednostavnu klasu
class MyActivity() : AppCompatActivity() {

    // lijeno injektiranje (eng. lazy inject) BusinessService
    // u svojstvo (eng. property)
    val service : BusinessService by inject()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // ili direktno dohvati bilo koju instancu
        val service : BusinessService = get()
    }
}
```

Isječak kôda 107: Naknadno dodavanje ovisnosti unutar klase koristeći *Koin*

Više o *Koinu* možemo pronaći u poglavlju 6 ili u literaturi s rednim brojem 11.

4.5.5. LiveData

LiveData je klasa nositelja podataka koja se može promatrati (eng. observe). Za razliku od običnog promatranja, *LiveData* je svjestan životnog ciklusa, što znači da poštuje životni ciklus ostalih komponenti aplikacije, poput aktivnosti, fragmenata ili usluga. Ta njegova svijest osigurava da *LiveData* ažurira samo promatrače komponenata aplikacije (eng. app component observers) koji su u aktivnom stanju životnog ciklusa.

LiveData smatra da je promatrač, kojeg predstavlja klasa *Observer*, u aktivnom stanju ako je njegov životni ciklus u stanju „početak“ (eng. Started) ili „nastavljen“ (eng. resumed). *LiveData* samo obavijesti aktivne promatrače o ažuriranjima. Neaktivni promatrači registrirani za gledanje *LiveData* objekata nisu obaviješteni o promjenama.

Možemo registrirati promatrača uparenog s objektom koji implementira *LifecycleOwner* sučelje. Ovaj odnos omogućuje uklanjanje promatrača kada se stanje odgovarajućeg životnog ciklusa promijeni u „uništeno“ (eng. destroyed). To je posebno korisno za aktivnosti i fragmente jer mogu sigurno promatrati *LiveData* objekte i ne brinuti se o curenju, aktivnosti i fragmenti se trenutno odjavljuju kada im se unište životni ciklusi.

Uz sve ovo u nastavku pogledajmo tablicu napravljenu prema (Developers.android.com, 2020) sa svim prednostima koje nam pruža LiveData i njihovim objašnjenjima.

Tablica 6: Prednosti korištenja LiveData

Prednost	Objašnjenje
LiveData slijedi obrazac promatrača(eng. observer pattern)	LiveData obavještava promatračke objekte o promjeni stanja životnog ciklusa. Možemo objediniti svoj kôd za ažuriranje korisničkog sučelja svaki put kada se podaci o aplikaciji promijeni, vaš promatrač može ažurirati korisničko sučelje svaki put kada dođe do promjene
Nema curenja memorije	Promatrači su vezani za predmete životnog ciklusa i čiste nakon sebe kad im se uništi povezani životni ciklus
Nema pada sustava zbog zaustavljenih aktivnosti	Ako je promatračev životni ciklus neaktivan, kao što je slučaj u aktivnosti na stražnjem snopu, rada ne prima nikakve događaje *LiveData*.

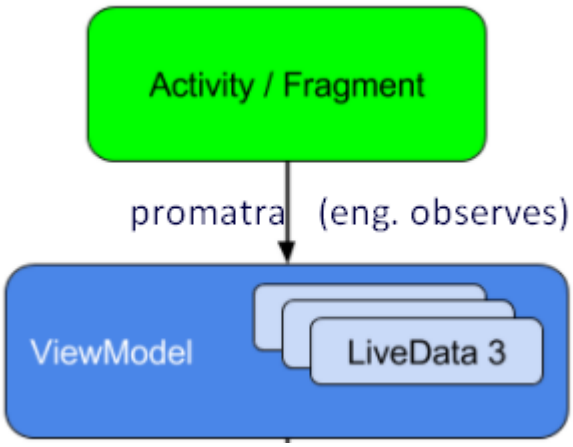
Uvijek ažurni podaci	Ako životni ciklus postane neaktivan, nakon što ponovno postane aktivan, prima najnovije podatke. Na primjer, aktivnost koja je bila u pozadini dobiva najnovije podatke odmah nakon što se vrati u prvi plan.
Pravilne promjene konfiguracije	Ako se neka aktivnost ili fragment ponovno stvori zbog promjene konfiguracije, poput rotacije uređaja, odmah dobiva najnovije dostupne podatke.
Dijeljenje resursa	Možemo proširiti *LiveData* objekt pomoću *singleton* uzorka za ogrtanje sistemskih usluga tako da se mogu dijeliti u vašoj aplikaciji. Objekt *LiveData* povezuje se na sistemsku uslugu jednom, a tada svaki promatrač kojem je resurs potreban, može samo gledati objekt *LiveData*.

Objekt *LiveData* obično se pohranjuje u objekt *ViewModela* i pristupa mu se putem *gettera*, kao što je prikazano u sljedećem primjeru.

```
class NameViewModel : ViewModel() {

    // Kreiranje LiveDate sa znakovnim nizom(String)
    val currentName: MutableLiveData<String> by lazy {
        MutableLiveData<String>()
    }

    // Ostatak viewModela
}
```

Isječak kôda 108: Primjer *LiveData* unutar *viewModela*

Za lakše razumijevanje kako *LiveData* funkcionira pripremljen je sljedeći graf.



Slika 11: Dijagram LiveDate

4.5.6. Room

Room biblioteka ili punim imenom „Room persistence library“ pruža sloj apstrakcije preko *SQLitea* kako bi se omogućio čvršći pristup bazi podataka uz istovremene korištenje pune snage *SQLitea*. Biblioteka pomaže da se stvori predmemorija podataka od aplikacije na uređaju koji pokreće istu aplikaciju. Ta predmemorija, koja služi kao jedinstveni izvori podataka vaše aplikacije, omogućava korisnicima da pregledavaju ključne podatake unutar vaše aplikacije, bez obzira na to imaju li internetsku vezu. Važno je napomenuti da je Room dio *Android Architecture Components*[14], a to je zbirka biblioteka koje pomažu u dizajniranju robusnih, provjerljivih i održivih aplikacija. *Room* će vam olakšati rad s *SQLiteDatase* objektima, smanjujući količinu *boilerplate* kôda i provjeru SQL upita u vrijeme kompiliranja.

Ukoliko već posjedujemo Android projekt u kojemu niste koristili Room veoma lako se možemo prebaciti na njega. Ovdje ćemo ukratko opisati što je sve potrebno učiniti za tu preobrazbu, a u poglavlju 6 možemo pogledati kako sve to izgleda na stvarnom projektu. Moramo ažurirati ovisnosti (eng. dependencies) u *gradleu*, stvoriti svoje entitete, DAO-ove i bazu podataka, zamijenite *SQLiteDatabase* pozive sa pozivima na DAO metode, testirajmo sve što ste stvorili ili izmijenili te uklonimo neiskorištene klase i to je to.

Neke od prednosti, odnosno, razloga zbog kojih bi trebali početi koristi Room su:

[14] Više o *Android Architecture Components* na:
https://developer.android.com/topic/libraries/architecture

- Provjera SQL upita u vrijeme kompiliranja – svaki *@Query* i *@Entitiy* se provjerava u vrijeme kompiliranja, što vašu aplikaciju čuva od problema s padom tijekom izvođenja, a ne samo da provjerava samo sintaksu, već i tablice koje nedostaju.
- Rješava nas *boilerplate* kôda
- Lako se integrira s ostalim *Android Architecture Components*, poput *LiveDate*

Ponekad želimo izraziti entitet ili objekt podataka kao kohezivnu cjelinu u logici baze podataka, čak i ako objekt sadrži samo nekoliko polja. U tim situacijama koristit ćemo anotaciju *@Embedded* da predstavimo objekt koji želimo razgraditi u njegova potpolja unutar tablice. U sljedećem primjeru možemo vidjeti model jednog grada koji sadrži ime, državu te koordinate koje predstavlja zasebni objekt i unutar toga imamo geografsku širinu (eng. latitude) i geografsku dužinu (eng. longitude). Slijedi podatkovna klasa za koordinate.

```
data class Coordinates(
    val lon: Double,
    val lat: Double
)
```

Isječak kôda 109: Podatkovna klasa *Coordinates*

Dok podatkovna klasa za objekt grad je prikazana u nastavku.

```
data class City(
    val name: String,
    @Embedded(prefix = "coord_")
    val coordinates: Coordinates,
    val country: String
)
```

Isječak kôda 110: Podatkovna klasa *City*

Za promjenu, dodavanje ili brisanje podatka također koristimo SQL upite koje dodajemo u parametar anotacije *@Query*. Pogledajmo u nastavku kako izgledaju spomenuti upiti nad podacima koji se odnose na prethodni objekt *City*.

```
@Dao
interface CityDao{
    // Dohvaćanje podataka iz tablice baze podataka City
    @Query("Select * from City")
    suspend fun getCity(): City

    // Dodavanje novog podatka u tablicu City
```

```
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertCity(model: City)

    // Brisanje modela City iz tablice
    @Delete
    suspend fun deleteCity(model: City)

    // Promjena podataka jednog retka u tablici City
    @Update(onConflict = OnConflictStrategy.REPLACE)
    suspend fun updateCity(model: City)
}
```

Isječak kôda 111: *Dao* sučelje *CityDao*

Kako bi poboljšali korištenje i funkcionalnost naše aplikacije, možemo definirati jedno bazno sučelje za sva *@Dao* sučelja koje prima određeni model te na principu njih može dodavati, brisati ili ažurirati podatke. Također, možemo dodati metode da dodavanje, ažuriranje i brisanje više modela odjednom, odnosno listu modela. Za implementaciju svega navedenog pogledajmo sljedeći Isječak kôda.

```
interface BaseDao<T> {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertModel(model: T)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertModels(models: List<T>)

    @Delete
    suspend fun delete(model: T)

     @Delete
    suspend fun deleteMultiple(models: List<T>)

    @Update(onConflict = OnConflictStrategy.REPLACE)
    suspend fun updateMultiple(models: List<T>)

    @Update(onConflict = OnConflictStrategy.REPLACE)
    suspend fun update(model: T)

}
```

Isječak kôda 112: *BaseDao* sučelje

Kako bi koristili ovo sučelje, trebamo ga implementirati unutar odgovarajućih sučelja te mu proslijediti model s kojim želimo raditi, a to činimo na sljedeći način.

```
interface CityDao : BaseDao<City?>
```

Isječak kôda 113: Implementacija *BaseDao* sučelja

4.5.7. Navigation

Navigation tj. prevedeno navigacija se odnosi na interakcije koje korisnicima omogućuju navigaciju, umetanje i povratak iz različitih dijelova sadržaja vašoj aplikaciji. Navigacijska komponenta također je dio *Android Architecture Components* koja nam pomaže u implementaciji navigacije, poput aplikacijskih traka i navigacijske ladice (eng. navigation drawer). Navigacijska komponenta također osigurava dosljedno i predvidljivo korisničko iskustvo pridržavanjem uspostavljenog skupa načela.

Navigacijska komponenta sastoji se od tri ključna dijela:

1. Navigacijski graf – XML resurs koji sadrži sve informacije vezane uz navigaciju na jednom centraliziranom mjestu. To uključuje sva pojedinačna područja sadržaja u našoj aplikaciji, koja se nazivaju odredišta, kao i moguće puteve koje korisnik može proći kroz vašu aplikaciju
2. NavHost – prazan spremnik(eng. container) koji prikazuje odredišta našeg navigacijskog grafikona. Navigacijska komponenta sadrži zadanu NavHost implementaciju, *HavHostFragmenta*, koja prikazuje odredišta fragmenta.
3. NavController – objekt koji upravlja navigacijom po aplikaciji unutar *NavHosta*. On orkestrira izmjenu odredišnog sadržaja u *NavHostu* dok se korisnici kreću kroz vašu aplikaciju.

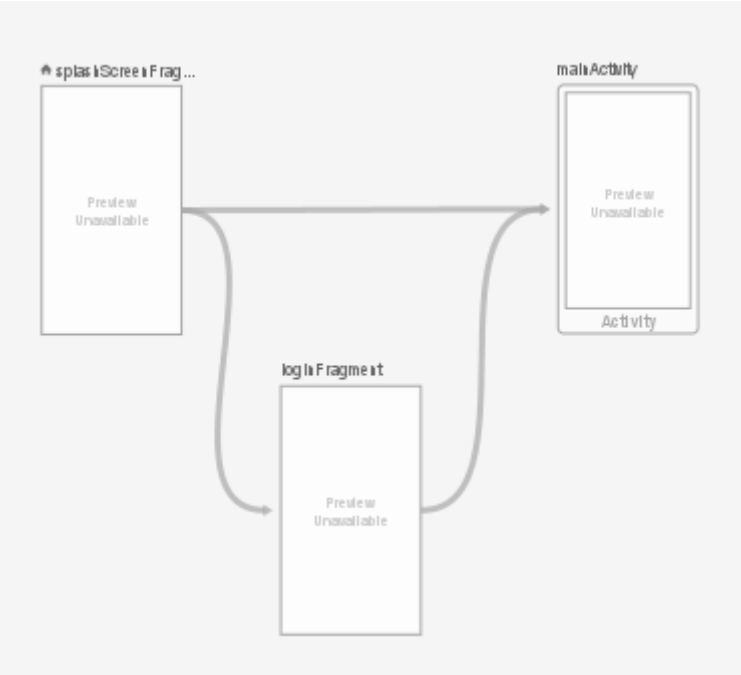
Dok se krećemo kroz našu aplikaciju, kažemo *NavControlleru* da želimo navigirati bilo kojom određenom stazom vašeg navigacijskog grafa ili izravno do određenog odredišta. *NavContoller* tada prikazuje odgovarajuće odredište u *NavHostu*.

Navigacijska komponenta pruža niz drugih prednosti, uključujući i sljedeće:

- Upravljanje transakcijama između fragmenata
- Ispravno rukovanje radnjama naprijed(eng. Up) i nazad(eng. back)
- Pružanje standardiziranih resursa za animacije i prijelaze
- Primjena i rukovanje dubokim povezivanjem
- Safe Args – dodataka *Gradleu* koji pruža sigurnost tip tijekom navigacije i prijenosa podataka između odredišta
- Podrška za *ViewModel*

Primjer jedne navigacijske komponente koja prvotno pokreće ekran sa logotipom aplikacije te dok se taj ekran pokazuje korisniku, u pozadini se provjerava je li korisnik već prijavljen u aplikaciju ili nije. Ukoliko je prijavljen aplikacija ga vodi na početni ekran

unutar aplikacije, a ako nije onda ga vodi na ekran za prijavu u aplikaciju. Navigacijski graf prikazan je na sljedećoj slici.



Slika 12: Primjer navigacijskog grafa

Xml Isječak kôda prikazan je u nastavku.

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
android:id="@+id/auth_nav_graph"
    app:startDestination="@id/splashScreenFragment">
    <fragment
        android:id="@+id/loginFragment"
        android:name="com.uhp.sanner.ui.fragments.LoginFragment"
        android:label="LoginFragment" >
        <action
            android:id="@+id/action_loginFragment_to_mainActivity"
            app:destination="@id/mainActivity" />
    </fragment>
    <activity
        android:id="@+id/mainActivity"
        android:name="com.uhp.sanner.ui.activities.MainActivity"
        android:label="MainActivity" />
    <fragment
        android:id="@+id/splashScreenFragment"
        android:name="com.uhp.sanner.ui.fragments.SplashScreenFragment"
        android:label="SplashScreenFragment" >
        <action
            android:id="@+id/action_splashScreenFragment_to_loginFragment"
            app:destination="@id/loginFragment" />
        <action
            android:id="@+id/action_splashScreenFragment_to_mainActivity"
            app:destination="@id/mainActivity" />
```

```
    </fragment>
</navigation>
```

Isječak kôda 114: XML kôd navigacijske komponente

Kako bi navigirali iz jednog ekrana unutar drugi koristimo metodu *navigate()* kojoj prosljeđujemo id akcije. U prethodnom slučaju na sljedeći način bi navigirali iz ekrana za prijavu u glavni ekran aplikacije.

```
        view.findNavController().navigate(R.id.action_loginFragment_to_main
Activity)
```

Isječak kôda 115: Korištenje metode *navigate()* za navigaciju unutar aplikacije

4.5.8. Glide

Glide je brz i učinkovit okvir za upravljanje medijem te za učitavanje slika za Android koji omotava dekodiranje medija, memoriju i predmemoriju diska, a udružuje resurse u jednostavno sučelje koje je lako za uporabu. *Glide* podržava dohvaćanje, dekodiranje i prikaz videozapisa, slika i animiranih GIF-ova. *Glide* uključuje fleksibilan API koji omogućava programerima da se priključe na gotovo bilo koji mrežni skup. *Glide* prema zadanim postavkama koristi prilagođeni snop temeljen na *HttpUrlConncetion*, ali uključuje i uslužne biblioteke koji su umjesto toga uključeni u Googleov projekt *Volley* ili *OkHttp* biblioteku.

Glide je primarno usredotočen na što brže i ugodnije pomicanje bilo kojeg popisa slika, ali *Glide* je također učinkovit za gotovo svaki slučaj u kojem morate dohvatiti, promijeniti veličinu i prikazati udaljeno (eng. remote) sliku.

Glide koristi jednostavan tečni API koji korisnicima omogućuje postavljanje većine zahtjeva u jednom retku. U nastavku je prikazan primjer kôda kako učitati određenu sliku preko URL-a unutar određenog elementa *imageView* (GitHub.com, 2020).

```
Glide.with(fragment)
.load(url)
.into(imageView)
```

Isječak kôda 116: Učitavanje slike uz pomoć *Glidea*

5. Testiranje Kotlin aplikacija

Kako se približavamo kraju ovog rada, tako sve više i više znamo o Kotlinu te proširujemo svoje programerske vještine. Već smo obradili mnoštvo Kotlinovih značajki koje će nam biti od koristi za izradu i razvoj mobilnih aplikacija, ali sigurno se pitamo što je sa testiranjem tih istih aplikacija i možemo li testirati svoje aplikacije isključivo pomoću Kotlina. Odgovor na sva ta pitanja je jednostavno „da“. U Androidu postoji nekoliko različitih testova kao što su jedinični testovi (eng. unit test) i instrumentacijski testovi (eng. instrumentation tests). U ovom radu to ćemo još proširiti sa jednom bibliotekom „Mockk“ te ukratko objasniti i definirati automatizirano testiranje. Ovo poglavlje obraditi će osnove o testiranju, ali naravno da to nije dovoljno da postanete vrsni tester ili QA stručnjak, za ovo područje postoje čitave knjige koje su posvećene samo tome. Cilj ovog poglavlja je objasniti kako pripremiti svoje okruženje za pisanje nekih testova i pokazati da Kotlin također dobro radi testiranje (Levia, 2017).

5.1. Testiranje aplikacije

5.1.1. Jedinični test

Za jedinične testove postoji mnogo različitih definicija koje u sebi sadrže male razlike, nećemo ulaziti u svaku od njih. Općenita ideja bi mogla biti da su to jedinični testovi koji potvrđuju pojedinačnu jedinicu izvornog kôda. Ono što uključuje ta „jedinica“ prepušta se čitatelju ovog rada. U našem slučaju pokretati ćemo testove koji ne trebaju pokretati uređaj. IDE će moći pokrenuti testove i pokazati rezultat koji identificira koji su testovi uspjeli, a koji nisu. Ispitivanje jedinice obično se vrši pomoću biblioteke *JUnit*, no postoje i mnoge druge koje možemo koristiti. Kada dodajemo ovisnosti u *build.gradle* umjesto da upišemo „implementation“ možemo upisati „testImplementation“ i na taj se način biblioteka izostavlja iz redovnih kompilacija, smanjujući veličinu konačne aplikacije. U nastavku ćemo napisati jedan veoma jednostavan test kako bi shvatili o čemu se radi te vidjeli kako to zapravo izgleda (Levia, 2017).

```
class SimpleTest {

    @Test
    fun unitTestingWorks() {
        assertTrue(true)
    }
}
```

Isječak kôda 117: Jednostavni jedinični test

Pomoću *@Test* anotacije odredimo funkciju kao test. Obavezno koristimo *org.unit.Test*. Zatim dodajemo jednostavnu tvrdnju, da vidimo funkcionira li sve ispravno. Sve što ovaj test radi je provjerava je li istina (eng. true) jednaka istini, znači veoma besmislen test, ali radimo ga samo kako bi shvatili koncept te sve što treba napraviti za napisati najobičniji jedinični test. Za provođenje testova, desnom tipkom miša kliknemo na mapu te odaberemo „Pokreni sve testove“ na engleskom će pisati „Run All Tests“. Kada kompilacija završi, pokrenuti će naš test te nakon završetka moći ćemo vidjeti sažetak koji prikazuje konačni rezultat. Za ovaj konkretni test trebali bismo vidjeti da je test prošao (eng. pass). U nastavku ćemo pisati stvarne testove, ali njihova složenost neće biti pretjerana.

```
@Test
fun testLongToDateString() {
    assertEquals("Oct 19, 2015", 1445275635000L.toDateString())
}
```

Isječak kôda 118: Jedinični test pretvaranja *Long* u *Date*

Ovaj test provjerava je li *Long* instanca ispravno pretvorena u *String*. Konkretno ovaj test testira zadano ponašanje, odnosno hoće li se zadani datum napisan u *Long* tipu, točnije u broj Unix sekundi pretvoriti u *DateFormat.MEDIUM* format. Ako smo već pisali jedinične testove u Javi, možemo primijetiti da nema puno razlike. Opisan je vrlo jednostavan primjer, ali odavde možemo stvoriti složenije testove za provjeru nekih drugih dijelova aplikacije (Levia, 2017).

5.1.2. Mocking

Ukoliko želimo testirati neke kompliciranije i složenije metode ili klase koje u sebi koriste neke druge klase, može biti veoma teško jer ne možemo lako doći do određenih klasa. Na primjer kada imamo klasu koja nam upravlja bazom podataka, a mi želimo testirati neku od metoda druge klase koja koristi tu bazu podataka. U takvim

situacijama mi te potrebne klase možemo oponašati (eng. mock). Za oponašanja klasa postoji puno raznih biblioteka koje na vrlo jednostavan način to odrađuju umjesto nas, jedna on najpoznatijih je svakako „Mockito“. Mockito je okvir za oponašanje (eng. mocking framework) koji nudi kompatibilnost s testiranje Android jedinica (eng. Android unit testing). Pomoću Mockita možemo konfigurirati oponašajuće objekte tako da vraćaju specifičnu vrijednost nakon pozivanja.

Za dodavanje okvira za oponašanje u testiranje lokalne jedinice pomoću svog okvira slijediti ćemo ovaj programski model:

1. Uključimo ovisnost biblioteke Mockito u svoju datoteku *build.gradle*
2. Na početku definicije testne klase jedinice dodajmo anotaciju *@RunWith(MockitoJUnitRunner.class)*. Ova anotacija govori Mockito test pokretaču da potvrdi da je naša upotreba okvira ispravna i pojednostavljuje inicijalizaciju naših oponašajućih objekta
3. Da bismo stvorili oponašajući objekt za ovisnost od Androidu, dodajte *@Mock* anotaciju prije deklaracije polja
4. Da bismo omeli ponašanje ovisnosti, možemo odrediti uvjet i vratiti vrijednost kada je uvjet ispunjen, koristeći metode *when()* i *thenReturns()*

Sljedeći primjer prikazuje kako se može kreirati jedinični test koji koristi oponašajući objekt *Context* objekta, primjer je preuzet sa službene Android stranice.

```
private const val FAKE_STRING = "HELLO WORLD"

@RunWith(MockitoJUnitRunner::class)
class UnitTestSample {

    @Mock
    private lateinit var mockContext: Context

    @Test
    fun readStringFromContext_LocalizedString() {
     // Given a mocked Context injected into the object under test...
        `when`(mockContext.getString(R.string.hello_word))
            .thenReturn(FAKE_STRING)
        val myObjectUnderTest = ClassUnderTest(mockContext)
     // ...when the string is returned from the object under test...
        val result: String = myObjectUnderTest.getHelloWorldString()
     // ...then the result should be the expected one.
        assertThat(result, `is`(FAKE_STRING))
    }
}
```

Isječak kôda 119: Primjer *mockiranja* klasa koristeći *Mockito*

5.1.3. Instrumentacijski test

Instrumentacijski testovi (eng. Intrumentation tests) su malo drugačiji. Obično se koriste za testiranje interakcija u korisničkom sučelju, gdje nam treba da se primjerak aplikacije koji se izvodi do trenutka kada su testovi svi izvršeni. Da bismo to učinili, trebat ćemo pokrenuti aplikaciju na uređaju, može biti fizički uređaj ili emulator, te potom pokrenuti sve testove na uređaju. Ova vrsta testova mora biti uključena u mapu *androidTest*,. Službena biblioteka za provođenje testova instrumentacije je „Espresso“, koja će nam pomoći lako kretati se kroz našu aplikaciju pisanjem akcija (eng. actions) te filtriranjem i provjerom rezultata pomoću *ViewMatchers* i *Matchers*. Konfiguracija je malo teža od prethodne. Potrebna nam je hrpa dodatnih biblioteka i *Gradle* konfiguracije. Dobra stvar je što Kotlin ne dodaje dodatne napore, tako da ako već znamo kako konfigurirati Espresso, bit će nam lako. Za primjer instrumentacijskog testa koristiti ćemo jedan test koji je otprije napisan, a sve što on radi je da provjerava je li aplikacija dohvatila ispravne podatke iz aplikacije. Aplikacija služi kao nekakav portfolio određene tvrtke u kojoj piše s čime se oni bave, prijašnji projekt, kontakt, lokacija, itd. Također, sastavni dio aplikacije je i kontakt forma preko koje se možemo njima javiti ukoliko želimo više informacija ili pak dogovoriti poslovanje s njima. Aplikacija se koristila na događajima kao što su sastanci, poslovni razgovori, predstavljanje tvrtke na sajmovima i slično. Pogledajmo kako izgleda taj instrumentacijski test, malo je duži, ali nije kompliciran (Levia, 2017).

```
class ContactDatabaseTest {

    @JvmField
    @Rule
    val instantTaskExRule = InstantTaskExecutorRule()

    private lateinit var contactDao: ContactDao
    private lateinit var db: AppDb
    private lateinit var contact: Contact

    @Before
    fun createDb() {
        runBlocking {
            val context =
InstrumentationRegistry.getInstrumentation().targetContext
            db = Room.inMemoryDatabaseBuilder(context,
AppDb::class.java).allowMainThreadQueries().build()
            contactDao = db.contactDao
            contact = Contact("Vedran", "Grbavac", "UHP",
"grbavac@uhp-digital.com", "0912312321", "Android", "Bla bla")
            contactDao.insertModel(contact)
        }
    }
```

```
    @After
    @Throws(IOException::class)
    fun closeDb() {
        db.close()
    }

    @Test
    fun getContact() = runBlockingTest {
        assertEquals(contactDao.getNumberOfContacts(), 1)
        assertEquals(contactDao.getAllContacts().blockingObserve()?
                        .get(0)?.firstName, contact.firstName)
        assertEquals(contactDao.getAllContacts().blockingObserve()?
                        .get(0)?.lastName, contact.lastName)
        assertEquals(contactDao.getAllContacts().blockingObserve()?
                        .get(0)?.company, contact.company)
        assertEquals(contactDao.getAllContacts().blockingObserve()?
                        .get(0)?.email, contact.email)
        assertEquals(contactDao.getAllContacts().blockingObserve()?
                        .get(0)?.phone, contact.phone)
        assertEquals(contactDao.getAllContacts().blockingObserve()?
                        .get(0)?.service, contact.service)
        assertEquals(contactDao.getAllContacts().blockingObserve()?
                        .get(0)?.description, contact.description)
    }
}
```

Isječak kôda 120: Primjer instrumentacijskog testa

U testu možemo vidjeti da imamo *@Before* i *@After*. U *@Before* dijelu klase imamo metodu *createDb()* kao što joj i samo ime kaže stvara bazu podataka te ubacuje jedan model *Contact* u nju, dok u *@After* dijelu klase imamo metodu *closeDb()* koja samo zatvara zadanu bazu podataka. Metodu koju trebamo proučiti je metoda *getContact()* koja je anotirana s *@Test*. Sve što ona radi je da prvo dohvaća sve kontakte unutar baze te provjerava je li njihov iznos jedan, nakon toga idemo atribut po atribut kontakta i provjeravamo je li isti kao i od kontakta kojeg smo definirali na početku. Ovdje zapravo testiramo hoće li se baza podataka ispravno stvoriti prilikom pokretanja aplikacije te unosi li i ispisuje li točne podatke iz nje.

5.2. MockK

Mockk je biblioteka koja nudi podršku za Kotlinove jezične značajke i konstrukcije. Mockk kreira zamjene za oponašajuće klase (eng. mocked vlasses). To uzrokuje određenu degradaciju performansi, ali ukupne prednosti koje na MockK pruža su vrijedne toga. U posljednjih nekoliko godina MockK se sve češće pojavljuje i vrti u svijetu Kotlina. Korisnici aktivno pomažu u poboljšanju programa tako što šalju

probleme i sugeriraju poboljšanja. Početkom prošle godine predstavio je veoma moćne značajke što ga je dovelo do vrhunca u njegovom korištenju te možemo sa sigurnošću reći da je zamijenio Mockito u svijetu Kotlina. Već znamo da je Mockito puno pomogao u jediničnim testovima te je bio najbolji način za kreiranje oponašajućih objekta u Javi i Kotlinu, no ipak znamo da je on zapravo napravljen za Javu te ima određena ograničenja kada ga koristimo u Kotlinu i primarno zbog tog razloga dolazio do razvoja *MockKa* (Mockk.io, 2020).

Značajke koje čine razliku između *MockKa* i *Mockita* su:

- Snimanje (eng. capturing) – snimanje argumenata može nam olakšati život ako trebamo dobiti vrijednost argumenta u *every* ili *verify* bloku. Postoje dva načina za hvatanje argumenta: korištenjem *CapturingSlot<Int>* ili korištenjem *MutableList<Int>*. *CapturingSlot* omogućava hvatanje samo jedne vrijednosti, tako da je jednostavniji za upotrebu
- Opuštene imitacije (eng. relaxed mocks) – prema zadanim postavkama oponašanje objekata je strogo. Prije prenošenja modela na kôd koji se testira, trebamo postaviti ponašanje sa *every* blokom. U slučaju da ne pružimo očekivano ponašanje, a poziv se obavlja, biblioteka će baciti iznimku. Ovo se razlikuje od onoga što *Mockito* radi po zadanim postavkama. *Mockito* nam omogućuje da preskočite specificiranje očekivanog ponašanja i odgovore s nekom osnovnom vrijednošću kao null ili 0. To isto, pa i više, možemo postići u *Mockku* deklariranjem opuštene imitacije.
- Špijuni (eng. spies) – špijuni daju mogućnost postavljanja očekivanog ponašanja i provjere ponašanja dok još uvijek izvršavaju izvorne metode objekta.
- Anotacije – biblioteka podržava anotacije *@MockK*, *@SpyK* i *@RelaxedMockK*, koje se mogu koristiti kao jednostavniji način na stvaranje odgovarajućeg oponašajućeg objekta, špijuna ili opuštene imitacije

Kako bi koristili oponašajuće klase unutar testiranja aplikacije prvo ih moramo definirati, a to činimo na sljedeći način.

```
val testCaseMocksModule: Module
    get() = module {
        single<SharedPreference> { mockk(relaxed = true) }
    }
```

Isječak kôda 121: Definiranje oponašajućih klasa

Nakon toga, moramo učitati te module, a za to koristimo sljedeću metodu.

```
loadKoinModules(testCaseMocksModule)
```

Isječak kôda 122: Učitavanje *Koin* modula

Kada smo to sve postavili, možemo koristiti oponašajuću klasu u našim testovima, a ovako izgleda primjer jednog testa koji provjerava jesmo li ispravno učitali aplikaciju na traženom jeziku.

```
class MainViewModelTest{
    private val mainViewModel = MainViewModel(get())
    private val sharedPrefs = SharedPreference(get())

    @Test
    fun getLanguage() {
        every { get<SharedPreference>().getValueString(KEY_LANG_CODE)
} returns LANG_CODE_ENGLISH
        assert(mainViewModel.getLanguage() == LANG_CODE_ENGLISH)
        assert(mainViewModel.getLanguage() != LANG_CODE_GERMAN)
        assert(sharedPrefs.getValueString(KEY_LANG_CODE) ==
                LANG_CODE_ENGLISH)
        assert(sharedPrefs.getValueString(KEY_LANG_CODE) !=
                LANG_CODE_GERMAN)
    }
}
```

Isječak kôda 123: Primjer jediničnog testa koristeći *Mockk*

5.3. Automatizirano testiranje

Automatizirano testiranje znači korištenje alata za automatizaciju izvršavanja skupa testnih slučajeva. Nasuprot tome je ručno testiranje. Za automatizirano testiranje postoji mnogo različitih definicija koje se razlikuju u pojedinim detaljima. Za našu svrhu držati ćemo se da je automatizirano testiranje proces korištenja softvera za kontrolu izvršenja testova, uspoređujući stvarne s predviđenim ishodima i za druge funkcije izvješćivanja o testiranju i kontroli (Ammann i Offutt, 2017).

Testiranje softvera može biti dug i skup posao te zbog toga cilj je automatizirati sto veći broj funkcionalnosti sustava. Budući da se automatsko testiranje provodi

pomoću alata za automatizaciju, potrebno je manje vremena za istraživačke testove i potrebno je više vremena za održavanje skripte za testiranje uz istovremeno povećanje ukupne pokrivenosti testom (Smartbear.com, 2020)

Uzastopni razvojni ciklusi zahtijevat će više puta izvršavanje istog testnog skupa. Pomoću alata za automatizirano testiranje moguće je snimiti ovaj testni paket i ponovo ga reproducirati prema potrebi. Jednom kada se testni paket automatizira, nije potrebna nikakva ljudska intervencija.

Sad već znamo vrijednost testiranja softvera, ali okruženja s brzim tempom softvera mogu stvoriti ograničenja vremena i troškova zbog kojih je teško temeljito testirati aplikaciju prije izdavanja. Ako se kvarovi ne primijete u proizvodnom okruženju, rezultat može biti nezadovoljstvo kupaca i povećani troškovi održavanja. Automatizacija testiranja omogućuje našem timu da izvrši više testova za manje vremena, povećavajući pokrivenost i oslobodivši ljudske testere da urade više istražne radnje na visokoj razini. Automatizacija je posebno korisna za testne slučajeve koji se ponavljaju, poput onih za kompatibilnost s pretraživačima i one koji su dio punog ili djelomičnog regresijskog skupa (Ranorex.com, 2020).

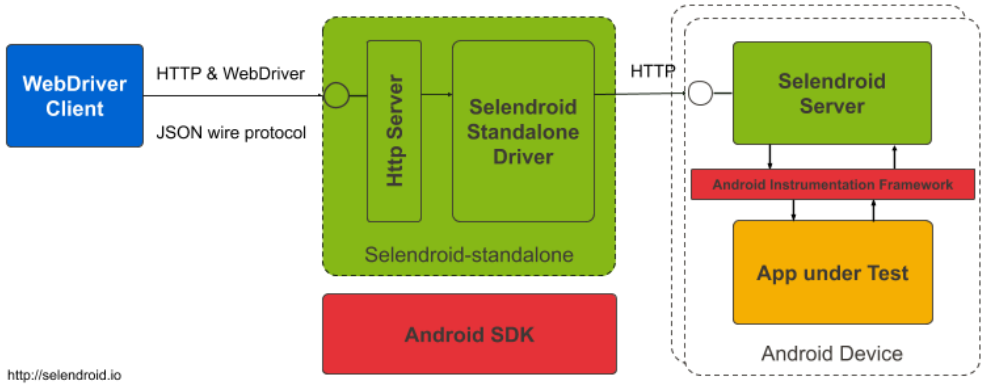
5.3.1. Alati za automatizirano testiranje Android aplikacija

5.3.1.1. Selendroid

Prema (Selendroid.io, 2020) *Selendroid* je testni okvir za automatizaciju koji odbacuje korisničko sučelje Androidovih izvornih i hibridnih aplikacija te mobilnog weba. Temelji se na Androidovom instrumentacijskom okviru (eng. instrumentation framework), pa je podržana samo jedna aplikacija. Selendroid sadrži četiri glavne komponente:

1. Selendroid-klijent – biblioteka java klijenta (zasnovana na Selenium Java klijentu)
2. Selendroid-Server – koji se izvodi uz vašu aplikaciju na Android uređaju.
3. AndroidDriver-App – ugrađena aplikacija za pregled web uređaja sa sustavom Android za testiranje mobilnog weba.
4. Selendroid-Standalone – upravlja različitim Android uređajima instaliranjem Selendroid-Servera i aplikacije koja se testira.

U nastavku pogledajmo njegovu arhitekturu.



Slika 13: Arhitektura Selendroid-a

U sljedećem primjeru, pokazano je kako bi testirali verifikaciju korisnika koji ispunjava nekoliko podataka o njemu.

```
@Throws(Exception::class)
private fun verifyUser(user: UserDO) {
    Assert.assertEquals(
        driver.findElement(By.id("label_username_data")).getText(),
        user.username
    )
    Assert.assertEquals(driver.findElement(By.id("label_email_data"))
          .getText(), user.email)
    Assert.assertEquals(
        driver.findElement(By.id("label_password_data")).getText(),
        user.password
    )
    Assert.assertEquals(driver.findElement(By.id("label_name_data"))
          .getText(), user.name)
    Assert.assertEquals(
        driver.findElement(
          By.id("label_preferedProgrammingLanguage_data")
          .getText(), user.programmingLanguage
    )
    Assert.assertEquals(driver.findElement(
          By.id("label_acceptAdds_data")).getText(), "true")
}
```

Isječak kôda 124: Primjer automatiziranog testa pisanog u *Selendroid*

5.3.1.2. Appium

Appium je okvir za automatizaciju otvorenog kôda (eng. open source) za upotrebu s izvornim, hibridnim i mobilnim web aplikacijama. Podržava simulatore (iOS), emulatore (Android) i stvarne uređaje (iOS, Android, Windows, Mac). Važno je da je Appium "cross-platform": omogućava nam pisanje testova na više platformi (iOS, Android, Windows), koristeći isti API. To omogućuje ponovnu upotrebu kôda između testova iOS, Android i Windows.

Appium je osmišljen kako bi zadovoljio potrebe mobilne automatizacije u skladu s filozofijom koju su navela sljedeća četiri načela:

1. Ne bismo trebali ponovno kompilirati aplikaciju ili je na bilo koji način mijenjati kako biste je automatizirali
2. Ne bismo trebali biti zaključani u određeni jezik ili okvir za pisanje i pokretanje testova
3. Okvir za mobilnu automatizaciju ne bi trebao ponovo „izumiti kotač“ kada je u pitanju API za automatizaciju
4. Okvir za mobilnu automatizaciju trebao bi biti otvoren izvor (eng. open source)

Neke od prednosti *Appiuma*, odnosno razlozi zašto bi ga trebali koristiti su:

- Ne moramo kopirati aplikaciju ili je na bilo koji način mijenjati zbog upotrebe standardnih API-ja za automatizaciju na svim platformama
- Testove možemo pisati s omiljenim alatima za razvoj, upotrebom bilo kojeg jezika koji je kompatibilan sa WebDriverom, kao što su Java, Objective-C, JavaScript (Node), PHP, Python, Ruby, C #, Clojure ili Perl
- Možemo koristiti bilo koji okvir za testiranje
- Appium ima ugrađenu podršku za mobilni web i hibridne aplikacije. Unutar iste skripte možemo se jednostavno prebacivati s nativne automatizacije aplikacija i automatizacije web pregledavanja, a sve koristeći model *WebDriver* koji je već standard za web automatizaciju.

Na primjer, s Googlovim UiAutomator-om ili Espressom možemo pisati testove samo u Javi i Kotlinu, dok Appium otvara mogućnost istinske automatizacije različitih platforma, za mobilne uređaje i šire (Appium.io, 2020).

U sljedećem primjeru bit će pokazan test koji šalje određenu poruku unutar Android elementa te onda verificira je li ispravno poruka ispisana, u ovom slučaju šalje se „Hello world“.

```
@Test
fun testSendKeys() {
    driver.startActivity(Activity(PACKAGE, SEARCH_ACTIVITY))
    val searchBoxEl =
        driver.findElementById("txt_query_prefill") as AndroidElement
    searchBoxEl.sendKeys("Hello world!")
    val onSearchRequestedBtn =
        driver.findElementById("btn_start_search") as AndroidElement
```

```
    onSearchRequestedBtn.click()
    val searchText = WebDriverWait(driver, 30)
        .until(
            ExpectedConditions.visibilityOfElementLocated(
                By.id(
                    "android:id/search_src_text"
                )
            )
        ) as AndroidElement
    val searchTextValue = searchText.text
    Assert.assertEquals(searchTextValue, "Hello world!")
}
```

Isječak kôda 125: Primjer automatiziranog testa pisanog u *Appium*

6. Izrada vlastite aplikacije i automatizirano testiranje

U ovom poglavlju će biti primijenjeno sve što smo do sada naučili i prošli. Ova aplikacija izrađena je u potpunosti za svrhu ovog rada te je u potpunosti u vlasništvu autora. Kôd ove aplikacije biti će javno dostupan te će svatko imati pravo na njegov uvid.

Aplikacija je na temu vođenja vlastitih financija, odnosno, aplikacija nam pomaže da pratimo vlastitu financijsku potrošnju. Ona prati potrošnju korisnika tako da korisnik najprije unese koliko je potrošio, kojeg datuma, što je kupio te u koju kategoriju potrošnje to spada. Nakon toga, aplikacija skuplja te podatke te radi izvješća do kojih korisnik u svakom trenutku može doći. Aplikacija je simbolično nazvana *BudgetApp*, samo kako bi bilo lakše i jednostavnije razumjeti o čemu se radi, te je za njenu upotrebu napravljen osnovni logo koji prikazuje vreću novaca te ime aplikacije. Osim toga, u aplikaciju je dodan i osnovni dizajn samo kako bi lakše bilo ju koristiti i opisivati.

U razvoju ove aplikacije koristili su se još neke dodatne tehnike i alate koji su u vrijeme pisanja ovog rada aktualni i koriste ih velike većine IT tvrtki, odnosno one su u vidu jednog standarda kojeg prati IT svijet, no ti se standardi rapidno mijenjaju u ovom sektoru poslovanja. Neki od tih standarda je i Git tok(eng. flow) o kojem je bilo malo riječi tijekom ovog rada, no ništa detaljno. Ova aplikacija je također koristila određeni Git flow, no on nije preporučen za korištenje u obrađivanju te teme, nego je korišten samo kako bi autoru olakšao razvoj ove aplikacije. Postoji mnoštvo literature na tu temu pa bolje njih koristite. Također, ovdje su primijenjeni još neki alati i tehnike koji nisu navedeni u ovom radu te preporučujem da njih ne uzimate u obzir prilikom obrađivanja teme ovog rada jer svrha ovog rada, kao što mu i naslov govori, prikazati razvoj i testiranje mobilne aplikacije u programskom jeziku Kotlin.

Tablica 7: Popis funkcionalnih i nefunkcionalnih zahtjeva aplikacije

Funkcionalni zahtjevi	Nefunkcionalni zahtjevi
Korisnik se mora moći registrirati i prijaviti u aplikaciju	Aplikacija mora raditi na Android verziji 6.0. ili većoj
Korisnik mora moći pregledavati i unositi nove transakcije	Zastoj u radu aplikacije ne smije preći 10 sekundi
Korisnik mora moći pregledati svoje korisničke podatke	Autentifikacija korisnika se vrši putem *Firebase* servisa

Korisnik mora moći pogledati statistiku svojih podataka na tri načina: • Ukupna statistika po kategoriji • Statistika prema datumu • Statistika prema kategoriji	Korištenje Kotlin programskog jezika pri razvoju i testiranju
Svaka transakcija se sprema u lokalnu bazu podataka	Uporaba propisanog standardnog procesa razvoja
Korisnik se u svakom trenutku može odjaviti iz aplikacije	Omogućiti konzistentan način rada sa sustavom u svrhu olakšanja korištenja. Ekrani sustava bi se trebali moći upotrebljavati na sličan način kroz cijeli sustav.
Popis transakcija se može sortirati prema njezinim atributima	Sustav bi se u budućnosti trebao moći nadograditi eventualnim dodatnim funkcionalnostima

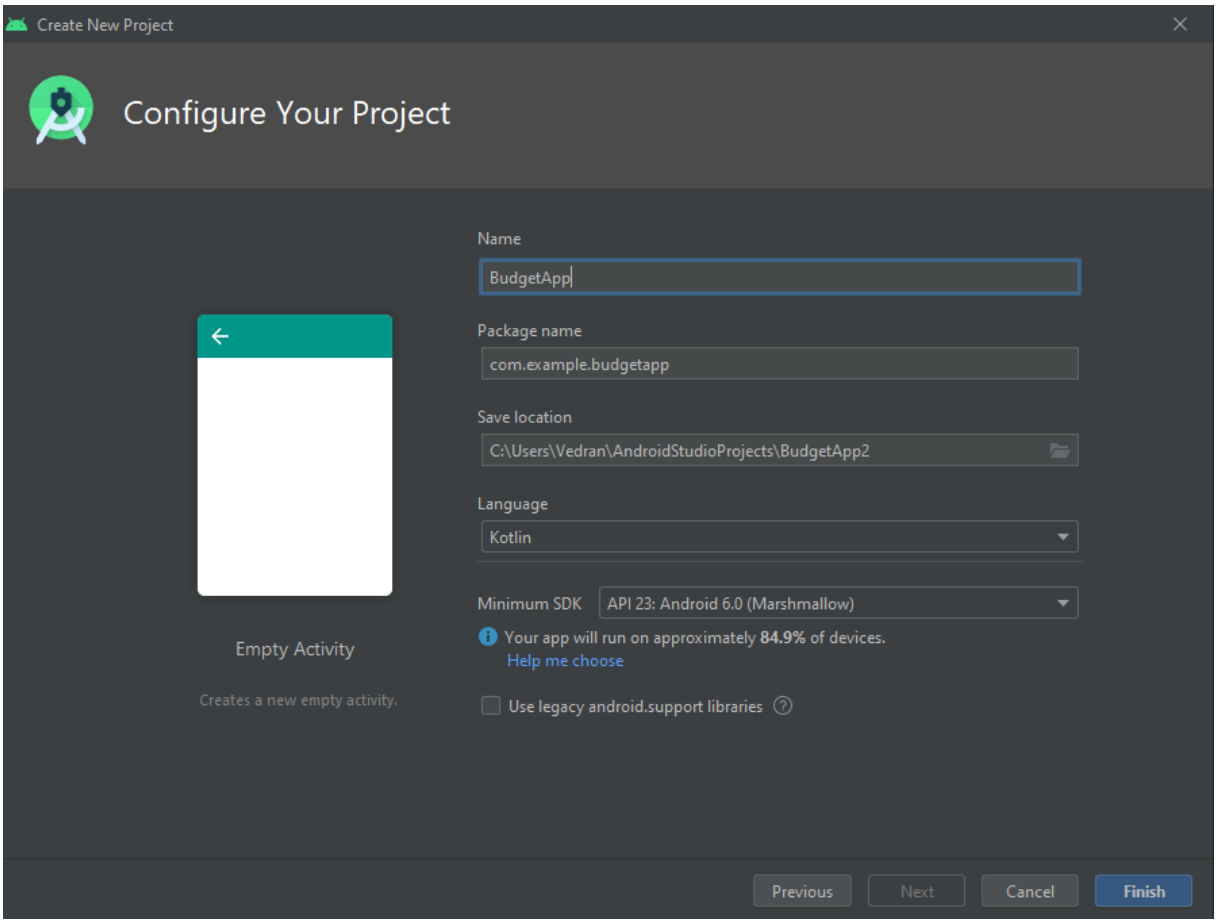
6.1. Razvoj aplikacije

Razvoj aplikacije započet je sa izrađivanjem repozitorija na Github.com stranici kako bi mogli pravilo izvršavati verzioniranje kôda aplikacije[15]. Nakon toga uslijedio je početni commit (eng. initial commit) te smo spremni za rad. Za korištenje git flow korišten je alat Sourcetree[16].

6.1.1. Kreiranje novog projekta

Nakon toga, krećemo sa kreiranjem novog projekta tako da pokrenemo Android Studio te unutar njega odaberemo File → New → New project… → odaberemo „Empty Activity“ te onda dobijemo prozor kao na slici 12.

[15] Link na GitHub repozitorij: https://github.com/vgrbavac/BudgetAPP
[16] Više o Sourcetree na: https://www.sourcetreeapp.com/



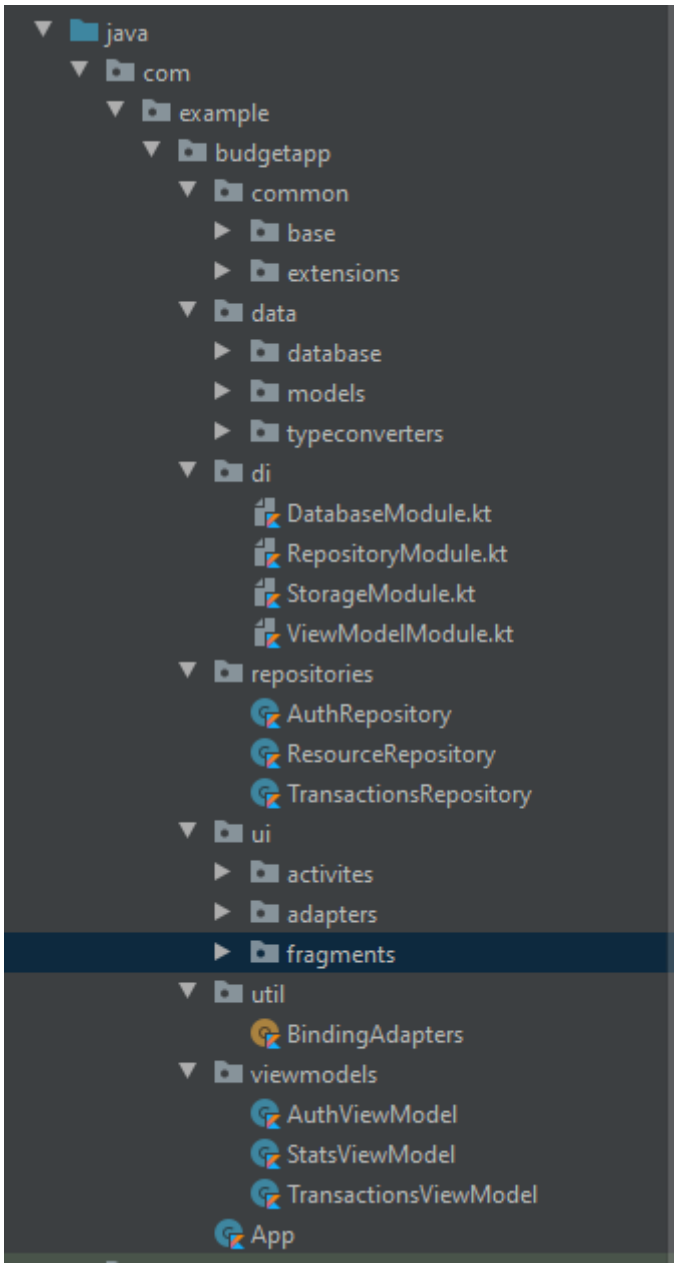
Slika 14: Konfiguracija novog projekta

Unutar prozora unesemo ime aplikacije, u našem slučaju BudgetApp te pod *language* odaberemo Kotlin. Minimum SDK za ovu aplikaciju je API 23 koji predstavlja Android 6.0(Marshmallow) te zastupljenost ove verzije androida i veće je 85% što je veoma dobar postotak, na kraju odaberemo *Finish* i izradili smo novu aplikaciju. Možemo odmah probati pokrenuti aplikaciju pritiskom na Run → Run 'app' ili kombinacijom tipki Shift + F10, kako bi provjerili da sve ispravno radi.

6.1.2. Okvirna arhitektura projekta

Krećemo sa pisanjem prvih linija kôda, za ovaj projekt prvo je postavljena okvirna arhitektura projekta koja prati MVVM standarde te je napravljena okvirna organizacija klasa unutar paketa (eng. package). Paket izrađujemo tako da kliknemo desnu tipku miša na mapu našeg projekta te odaberemo New → Package te unesemo ime novog paketa. Za početak smo napravili *ui* paket koji će skladištiti sve klase koje su se odnosile na korisničko sučelje, a to su aktivnosti, fragmenti i adapteri. Nakon toga smo napravili paket *viewmodels* koji će skladištiti sve *viewModele*, potom paket

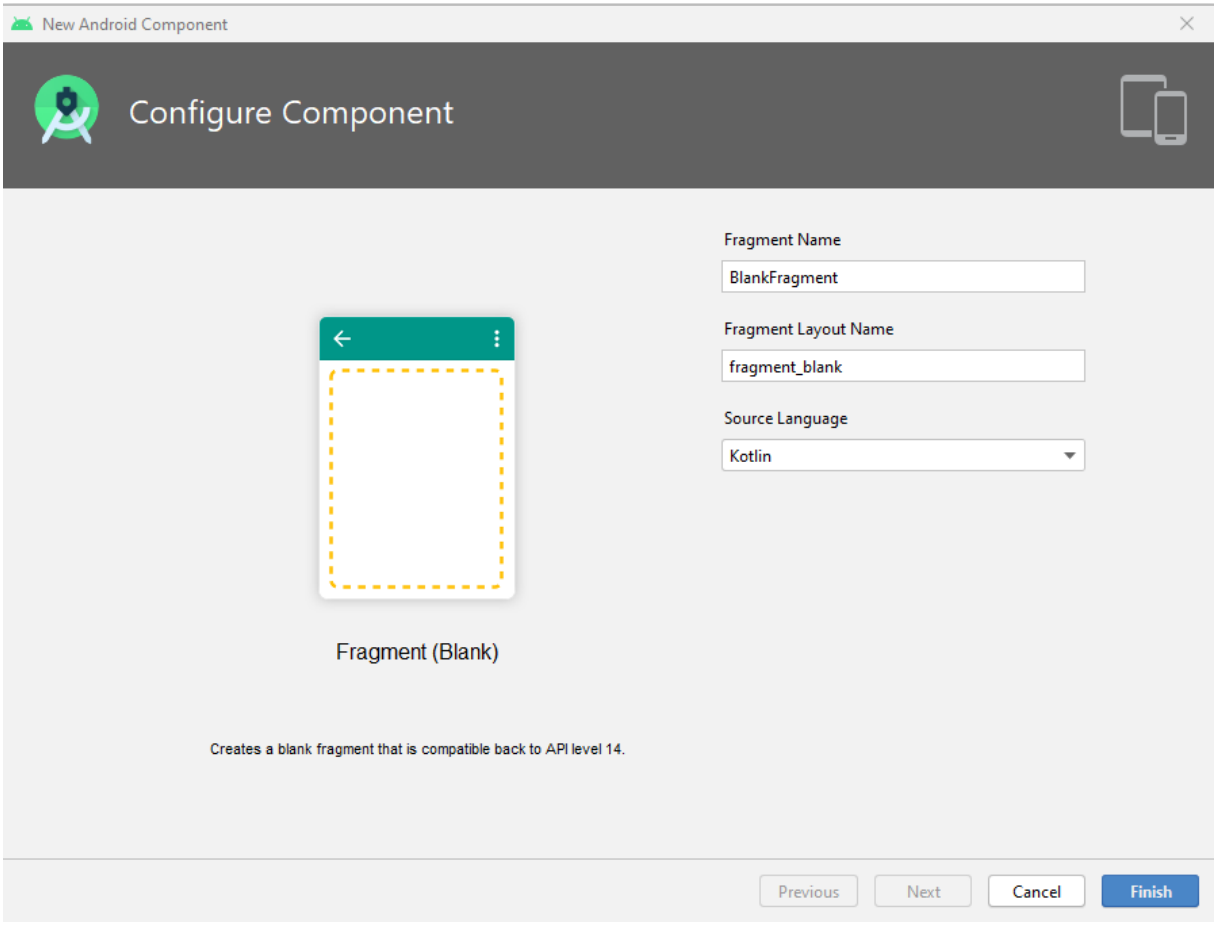
data u kojem će biti smještene sve klase koje će raditi sa podacima kao što su modeli, *dao* sučelja, klase za bazu podataka itd. Postoji puno vrsta organizacija kôda te svaki pojedinac radi nešto drugačije, odnosno prilagođava sebi kako mu najviše odgovara. Organizacija kôda ne utječe na rad aplikacije nego je tu samo kako bi što lakše i što brže mogli doći do određenog dijela kôda kako bi ga promijenili, popravili ili nadopunili. Ova verzija organizacije nekome će odgovarati dok drugome neće, kako je finalna organizacija dokumenata izgledala pogledajmo na sljedećoj slici.



Slika 15: Organizacija kôda

6.1.3. Kreiranje prvih ekrana i navigacije

Osim organizacije kôda napravljeni su osnovni ekrani (eng. screens) koje ćemo koristiti u aplikaciji, odnosno, napravljeni su fragmenti i njihovi *layouti*. Fragment možemo dodati tako da kliknemo desni klik miša na mapu projekta te odaberemo New → Fragment → Fragment(Blank) te dobijemo prozor kao na sljedećoj slici.



Slika 16: Dodavanje novog fragmenta

Uz sve to, unutar *MainActivitya* dodana je alatna traka (eng. toolbar) i donja navigacijska traka (eng. bottom navigation) koja koristi Navigation komponentu opisanu u poglavlju 4.5.7.

Za dodavanje alatne trake dodajemo sljedeće linije kôda unutar našeg *layouta* povezanog sa *MainActivity.*

```
<androidx.appcompat.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:theme="@style/ToolbarTheme"
    app:titleTextColor="@android:color/darker_gray"
```

```
app:layout_constraintStart_toStartOf="parent"
app:layout_constraintTop_toTopOf="parent" />
```

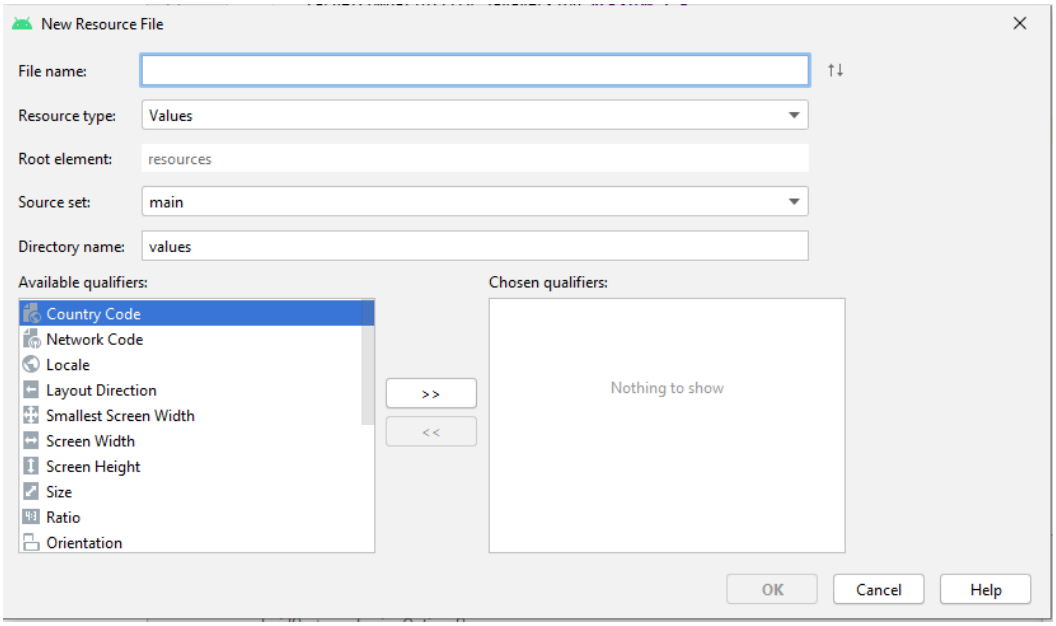
Isječak kôda 126: XML kôd alatne trake (eng. toolbar)

Možemo primijetiti da je dodana nova tema za navedenu alatnu traku. Izradu nove teme vršimo tako da unutar mape *values* nađemo datoteku *styles.xml* te u nju dodamo novi stil. Novi stil dodajemo tako da upišemo „<style>“ i pritisnemo tipku enter. Ovaj stil se sasvim jednostavan, samo mijenjamo pozadinu u bijelu boju, a to izgleda ovako.

```
<style name="ToolbarTheme"
parent="Theme.AppCompat.Light.DarkActionBar">
    <item name="android:background">@android:color/white</item>
</style>
```

Isječak kôda 127: Stil za alatnu traku

Osim alatne trake potrebno je u isti *layout* dodati i kôd za donju navigacijsku traku, no prije nego što nju dodamo, napraviti ćemo novi resurs koji će sadržavati elemente navedene trake. Dodajemo novu resursnu datoteku (eng. resource file) tako da kliknemo desni klik na mapu *res* te odaberemo New → Android Resource File te dobijemo novi prozor, slika 17. Unutar prozora pod „Directory name“ upišemo „menu“, a za „File name“ upišemo „menu_navigation“.



Slika 17: Dodavanje novog resursa u aplikaciju

Sada unutar tog resursa dodajemo elemente našeg navigacijske trake. Dodajemo ih tako da počnemo pisati „<item>“ te pritiskom na tipku enter kreiramo novi elementa. Unutar elementa dodajemo mu atribute *id, icon* i *title* gdje *id* predstavlja jedinstvenu

oznaku elementa, *icon* je referenca na ikonu koju želimo koristiti za prikaz tog elementa, a *title* je tekst koji će pisati uz ikonu elementa. Pogledajmo kako izgleda ovaj resurs u sljedećem isječku kôda.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/graphFragment"
        android:icon="@drawable/ic_graph_24"
        android:title="@string/statistics" />

    <item
        android:id="@+id/transactionsFragment"
        android:icon="@drawable/ic_baseline_monetization_on_24"
        android:title="@string/transactions" />

    <item
        android:id="@+id/profileFragment"
        android:icon="@drawable/ic_profile_24"
        android:title="@string/profile" />

</menu>
```

Isječak kôda 128: XML kôd elemenata izbornika (eng. menu)

Kada smo to napravili, vraćamo se u naš *layout* gdje dodajemo sljedeći Isječak kôda.

```
<com.google.android.material.bottomnavigation.BottomNavigationView
    android:id="@+id/bottomNav"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="@color/colorWhite"
    app:labelVisibilityMode="labeled"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:menu="@menu/menu_navigation" />
```

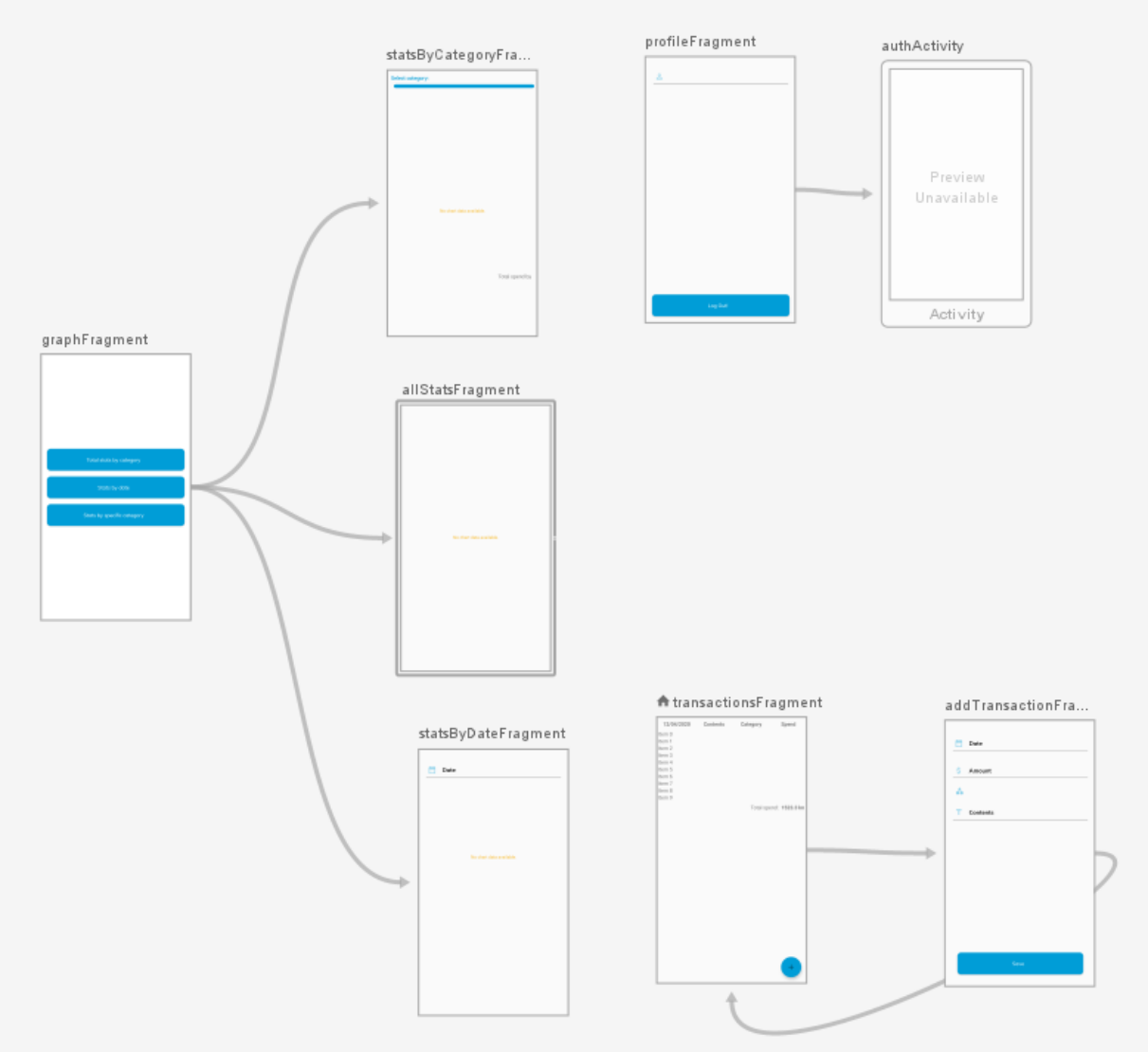
Isječak kôda 129: XML kôd donje navigacijske trake

Možemo primijetiti da smo za atribut „app:menu“ dodali naš prethodno izrađeni resurs te sada koristimo sve elemente koji su u tom resursu. Ako želimo dodavati ili brisati elemente naše navigacijske trake, sve što trebamo napraviti je unutar *menu_navigation.xml* dodati ili obrisati elemente i oni će se automatski ažurirati unutar navigacijske trake. Atribut „app:labelVisibilityMode“ nam govori hoćemo li prikazati *title* unutar elementa ili ne, u našem slučaju stavljeno je „labeled“ što označava da će se prikazivati *title* uz ikonu.

Kako bi dodali Navigation komponentu, prvo u *app/build.grade* moramo dodati ovisnost (eng. dependency) te moramo sinkronizirati (eng. sync) *gradle.* Kada smo to

napravili dodajemo novu resursnu datoteku isto kao i za „menu“(Slika 17) u kojemu dodamo ime „nav_graph“ te pod „Resource type“ odaberemo *Navigation*.

Sada smo napravili navigacijsku komponentu, ulaskom na nju dobivamo vizualni pregled naše aplikacije. U gornjem desnom kutu imamo ikonu mobitela sa zelenim plusom, kada ju kliknemo možemo dodavati fragmente tj. ekrane koji će se koristiti u aplikaciji. Dodamo sve fragmente koje imamo te ih možemo povezivati tako da na sredini pojedinog ekrana kliknemo na kružić i povučemo mišem na ekran koji želimo otvoriti. Kako izgleda ovaj navigacijski graf pogledajmo na sljedećoj slici.



Slika 18: Navigacijski graf aplikacije

Svakom ekranu ili vezi između ekrana možemo dati određeni *id* te naslov. Osim toga preko navigacijske komponente možemo prosljeđivati i argumente između fragmenata ili aktivnosti. Kada unutar kôda želimo navigirati s jednog ekrana na drugi, dovoljno je

samo da pozovemo metodu navigacijskog kontrolera *navigate()* te mu kao parametar damo *id* određene akcije, odnosno veze.

```
view.findNavController()
.navigate(R.id.action_loginFragment_to_registrationFragment)
```

Isječak kôda 130: Navigacija u aplikaciji uz pomoć navigacijske komponente

Opširniji i bolji priručnik kako postaviti navigacijsku komponentu možemo pronaći u („Using The Navigation Architecture Component in Andorid Jetpack(Kotlin)“, 2018.)

Da bi naša navigacija i alatna traka bila vidljiva i radila kako spada, moramo ih pozvati unutar naše aktivnosti, u ovom slučaju to je *MainActivity*. Za navigacijsku traku kreirati ćemo metodu koja će preko dodanog navigacijskog kontrolera mijenjati ekrane. Uz to dodati ćemo dvije nove varijable koje će predstavljati navigacijskog domaćina (eng. navigation host) te navigacijski kontroler (eng. navigation controller). Obje varijable će biti pozvane sa *lazy* delegatom koji je opisan u poglavlju 3.7.1. Još ćemo dodati jednu metodu za slušaoca destinacije (eng. destination listener) u kojem ćemo definirati da ukoliko je destinacija profil, statistika ili transakcije da u alatnoj traci ne prikazuje *home* gumb. U sve to nadjačavamo metodu *onSupportNavigateUp()* koja nam definira što će se dogoditi prilikom pritiska gumba za nazad. U sljedećem isječku kôda pogledajmo kako konačno izgleda naša aktivnosti.

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

    private val navHostFragment: NavHostFragment
          by lazy { getNavHost() as NavHostFragment }
    protected val navController: NavController
          by lazy { navHostFragment.navController }

    fun getNavHost(): Fragment = drawerNavHost

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding =
          DataBindingUtil.setContentView(this, R.layout.activity_main)

        setSupportActionBar(toolbar)
        setupBottomNavigation()
        setupActionBar()
        setupDestinationListener()
    }

    private fun setupBottomNavigation() {
```

```
        bottomNav?.let {
            NavigationUI
                 .setupWithNavController(bottomNav, navController)
        }
    }

    private fun setupActionBar() {
        NavigationUI
                 .setupActionBarWithNavController(this,
                 navController, drawer_layout)
    }

    private fun setupDestinationListener() {
        navController.addOnDestinationChangedListener
{ _, destination, _ ->
            if (destination.id == R.id.profileFragment ||
                destination.id == R.id.graphFragment ||
                destination.id == R.id.transactionsFragment
            ) {
                supportActionBar?.setDisplayHomeAsUpEnabled(false)
            }
        }
    }

    override fun onSupportNavigateUp(): Boolean {
        return NavigationUI.navigateUp(
            Navigation.findNavController(this, R.id.drawerNavHost),
            drawer_layout
        )
    }
}
```

Isječak kôda 131: Aktivnost *MainActivity*

Također, na početku dodane su i neke osnovne ikone koje ćemo koristi kroz aplikaciju, ali kroz razvoj neke od njih su promijenjene, a neke su zamijenjene novima. Sve ikone korištene osim za logo aplikacije su iz Android assetsa te su korištene vektorske ikone. Za dodavanje ikone kliknemo desni klik na *res* datoteku te odaberetmo New → Vector asset i dobijemo novi prozor unutar kojeg klikom na „Clip Art“ odaberemo ikonu koju želite potom joj postavimo ime zatim kliknemo *Next* pa *Finish* i dobili smo vektorsku ikonu koja se nalazi unutar *drawable* mape i tamo ju možemo preuređivati ukoliko želite. Također, važno je napomenuti da smo uz dodavanje fragmenata dodavali i njihove *viewModele* pa je tako dodan *TransactionViewModel* koji je za početak bio prazan. Ovo sve je sadržavao prvi *commit*.

6.1.4. Dodavanje *data bindinga* u aplikaciju

Sljedećih nekoliko *commitova* odnosili su se na poboljšanje trenutne arhitekture te njezino proširenje. Važna stvar za napomenuti je ta da su svi *xml layouti* promijenjeni da budu *data binding*. To možemo napraviti tako da unutar *app/build.gradle* dodamo sljedeće linije kôda unutar android blok naredbi.

```
dataBinding {
    enabled = true
}
```

Isječak kôda 132: Postavljanje *data bindinga* u aplikaciji

Nakon što to dodamo unutar svakog *xml layouta* možemo pritisnuti kombinaciju tipki Alt + Enter i odabrati opciju „Convert to data binding layout“ te će Android studio sve odraditi za nas. Pogledajmo kako izgleda početak *layouta* koji se odnosi na ekran transkackija, a sadrži jednu varijablu *viewModel* koja je *TransactionViewModel* tip.

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <data>
        <variable
            name="viewModel"

type="com.example.budgetapp.viewmodels.TransactionsViewModel" />
    </data>
</layout>
```

Isječak kôda 133: Primjer *data binding layouta*

Sve što u prethodnom *layoutu* trebamo dodati jesu elementi našeg ekrana, a njih dodajemo ispod bloka koji se odnosi na „<data>“.
Zatim trebamo unutar klase fragmenta inicijalizirati taj *binding* tako da upišemo sljedeće, ovo je primjer za TransactionFragment.

```
private lateinit var binding: FragmentTransactionsBinding
```

Isječak kôda 134: Inicijalizacija *data bindinga*

Nakon toga, postoji više načina da pozovemo tu varijablu *binding,* jedan od načina koji je korišten u razvoju ove aplikacije je unutar metode *onCreateView()* dodamo sljedeće linije.

```
binding =
    DataBindingUtil
.inflate(inflater, R.layout.fragment_transactions, container, false)

return binding.root
```

Isječak kôda 135: Pozivanje *data bindinga*

Ako želimo dodavati varijable unutar *bindinga* onda mu ju moramo pridružiti unutar metode *onViewCreated()* te mu također trebamo dati novi *lifecyclerOwner*, u ovom fragmentu prosljeđivali smo varijablu *viewModel* pa je to izgledalo ovako.

```
binding.lifecycleOwner = viewLifecycleOwner
binding.viewModel = viewModel
```

Isječak kôda 136: Pridruživanje varijable *data bindingu*

6.1.5. Rad s transakcijama

Nakon još nekoliko *commitova* u kojima smo dorađivali i poboljšavali dosadašnju strukturu i kôd, slijedi izrada prve i najvažnije funkcionalnosti, a to je unos i ispis transakcija. Prije nego što smo započeli s razvojem ove funkcionalnosti napravili smo novu granu (eng. branch) na *development* grani kako bi odvojili funkcionalnost u zasebnu granu te je po završetku ponovno vratili (eng. merge) u *development* granu.

6.1.5.1. Priprema ispisa transakcija

Prvo smo dodali osnovni element ove funkcionalnost, a to je *recyclerView*[17] u koji ćemo puniti listu s podacima i on će nam ispisivati sve podatke.

Za njega smo izradili i posebni Android resurs koji će predstavljati jedan element unutar liste. Kako bismo mogli njemu pridružiti *data binding* koji će unutar podataka imati varijablu koja se odnosi na našu transakciju, prvo moramo stvoriti model transakcije unutar baze podataka, nazvati ćemo ju *DBTransaciton*.

[17] Više o *recyclerView* možemo pronaći na:
https://developer.android.com/jetpack/androidx/releases/recyclerview

```
@Entity
data class DBTransaction(
    @PrimaryKey(autoGenerate = true)
    val id: Int,
    val contents: String,
    val date: LocalDate,
    val category: String,
    val totalPrice: Double
) {
    constructor(
        contents: String,
        date: LocalDate,
        category: String,
        totalPrice: Double
    ) : this(0, contents, date, category, totalPrice)
}
```

Isječak kôda 137: Model transakcije *DBTransaction*

U ovu *data* klasu smo dodali još anotaciju *@Entity* kako bi *Room* znao da se radi o entitetu baze podataka. Također postavili smo da je *id* atribut auto generiran te izradili konstruktor u kojem prosljeđujemo sve atribute osim *id* tako da se ne moramo brinuti o njemu jer je on auto generiran. Svaki element *recyclerViewa* se sastojao od *LinearLayouta* kojem je orijentacija postavljena na horizontalnu te unutar njega smo dodali datum, opis, kategoriju i cijenu te smo sve te elemente ravnomjerno rasporedili koristeći atribut *weight*, pogledajmo u nastavku.

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <data>
        <variable
            name="transaction"

type="com.example.budgetapp.data.models.persistance.DBTransaction" />

    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="horizontal"
            android:gravity="center_vertical"
            android:background="@color/colorPrimary"
            android:padding="8dp"
            android:layout_marginTop="4dp"
```

```
                android:weightSum="4"
                app:layout_constraintTop_toTopOf="parent"
                app:layout_constraintStart_toStartOf="parent">

                <androidx.appcompat.widget.AppCompatTextView
                    android:id="@+id/tvDate"
                    style="@style/elementInList"
                    app:dateToText="@{transaction.date}"
                    tools:text="12/04/2020" />

                <androidx.appcompat.widget.AppCompatTextView
                    android:id="@+id/tvTitle"
                    style="@style/elementInList"
                    android:text="@{transaction.contents}"
                    tools:text="Groceries for house" />

                <androidx.appcompat.widget.AppCompatTextView
                    android:id="@+id/tvCategory"
                    style="@style/elementInList"
                    android:text="@{transaction.category}"
                    tools:text="Food" />

                <androidx.appcompat.widget.AppCompatTextView
                    android:id="@+id/tvTotalPrice"
                    style="@style/elementInList"
                    app:doubleToText="@{transaction.totalPrice}"
                    tools:text="145,99 kn" />

            </LinearLayout>

        </androidx.constraintlayout.widget.ConstraintLayout>
    </layout>
```

Isječak kôda 138: XML kôd jednog elementa *recyclerViewa*

Možemo vidjeti da i ovaj *layout* koristi *data binding* te sadrži jedu varijablu tipa *DBTransaction* koja je zapravo model naše transakcije. Putem te varijable možemo ispisivati podatke direktno unutar naših elemenata ekrana. Također, vidljivo je da su tu korištena dva vrlo jednostavna *binding* adaptera, jedan je pretvarao tip varijable *Date* u znakovni niz (eng. String), a drugi je pretvarao tip varijable *Double* u znakovni niz. Pogledajmo u nastavku njihov Isječak kôda, a kako ih pozivamo mogli smo vidjeti u prethodnom isječku kôda.

```
@RequiresApi(Build.VERSION_CODES.O)
@BindingAdapter("dateToText")
@JvmStatic
fun dateTimeToString(textView: TextView, localDate: LocalDate?) {
    val formatter = DateTimeFormatter.ofPattern("dd MMM yyyy")
    textView.text = localDate?.format(formatter)
}

@SuppressLint("SetTextI18n")
@BindingAdapter("doubleToText")
@JvmStatic
```

```
fun doubleToStringWithTwoDecimals(textView: TextView, value: Double?)
{
    textView.text = """${value?.format(2).toString()} kn"""
}
```

Isječak kôda 139: *Binding* adapteri za *Date* i *Double*

Nakon toga na ekran transakcija dodali smo zaglavlje koje je sadržavalo naslove podataka koje je bilo statičko. Zaglavlje je bilo napravljeno isto kao i *layout* koji služi za renderiranje svakog elementa unutar *recyclerView*, jedino što su tekstovi bili podebljani i statički.

Poslije svega dodajemo adapter koji nam služi za renderiranje podataka unutar *recyclerViewa.* Kako bismo poboljšali kôd, neke dijelove iz te klase smo izvukli i napravili *BaseAdapter* koji sadrži osnovne stvari za svaki *recyclerView* adapter te tako olakšavamo daljnji razvoj aplikacije, a on je izgledao ovako.

```
typealias ItemClickListener<T> = ((T, Int) -> Unit)?

fun <T> createDefaultDiffCallback() = object :
DiffUtil.ItemCallback<T>() {
    override fun areItemsTheSame(oldItem: T, newItem: T): Boolean {
        return oldItem?.equals(newItem) ?: false
    }
    @SuppressLint("DiffUtilEquals")
    override fun areContentsTheSame(oldItem: T, newItem: T): Boolean
{
        return oldItem?.equals(newItem) ?: false
    }
}
abstract class BaseAdapter<M, VH : AbstractViewHolder<M>>(
    private val listener: ItemClickListener<M>,
    diffCallback:
          DiffUtil.ItemCallback<M>=createDefaultDiffCallback()
) : ListAdapter<M, VH>(diffCallback) {

    protected abstract val itemLayout: Int

    protected abstract fun createViewHolder(view: View): VH

    override fun onCreateViewHolder(
          parent: ViewGroup, viewType: Int): VH {
        val view = LayoutInflater.from(parent.context)
                        .inflate(itemLayout, parent, false)
        return createViewHolder(view)
    }

    override fun onBindViewHolder(holder: VH, position: Int) {
        holder.bind(getItem(position), position, listener)
    }

}
```

Isječak kôda 140: *BaseAdapter* za pomoć pri stvaranju *recyclerViewa*

Dok je adapter za renderiranje podataka unutar *recyclerView*, izgledao ovako.

```
class TransactionsRecyclerAdapter(itemClickListener: ((DBTransaction,
Int) -> Unit)? = null) :
    BaseAdapter<DBTransaction,
TransactionsViewHolder>(itemClickListener) {
    override val itemLayout:
         Int = R.layout.layout_transactions_list_item

    override fun createViewHolder(view: View): TransactionsViewHolder
{
        val layoutInflater = LayoutInflater.from(view.context)
        val binding = LayoutTransactionsListItemBinding
                .inflate(layoutInflater, view as ViewGroup?, false)
        return TransactionsViewHolder(binding)
    }

    override fun getItemId(position: Int): Long {
        return position.toLong()
    }
}

class TransactionsViewHolder(view: View) :
AbstractViewHolder<DBTransaction>(view) {

    lateinit var binding: LayoutTransactionsListItemBinding

    constructor(binding: LayoutTransactionsListItemBinding) :
this(binding.root) {
        this.binding = binding
    }

    @RequiresApi(Build.VERSION_CODES.O)
    override fun bind(model: DBTransaction, position: Int,
                listener: ItemClickListener<DBTransaction>) {
        binding.transaction = model
        binding.executePendingBindings()

    }
}
```

Isječak kôda 141: Adapter za *recyclerView*

Unutar nadjačane metode *bind* možemo vidjeti da smo na *binding* od *layouta* pridružili naš model *DBTransaction* i s tim smo dobili da mi samo dodajemo nove modele, a naš adapter će ih proslijediti u *layout* dok će on ih ispisati unutar elemenata onako kako smo ih mi zadali. Ukoliko bi sada izrađivali još jedan adapter bilo bi puno lakše i jednostavnije.

Također, morali smo pripremiti i još jedan *binding* adapter koji će nam služiti kako bi putem *data binding* mogli ubaciti listu elemenata u naš *recyclerView.* Najprije smo u *layout* dodali varijablu tipa *TransactionViewModel* koji je dodan u poglavlju 6.1.3. Prije dodavanja *binding* adaptera unutar *viewModel* klase napravili smo jednu

varijablu tipa *MediatorLiveData* putem koje ćemo kasnije dodavati listu transakcija za ispis na ekran. Uz ovu varijablu dodana je još jedan koja će služiti za dohvaćanje transakcija iz baze podataka preko repozitorija. Ovako je to izgledalo na kraju.

```
class TransactionsViewModel{
    val transactionsLiveData: LiveData<List<DBTransaction>?>
        get() {
            TODO()
        }

    val filteredData = MediatorLiveData<List<DBTransaction>>().apply
{
     listOf(
        transactionsLiveData
    )
 }
}
```

Isječak kôda 142: Početna faza *TransactionsViewModela*

Potom nastavljamo sa dodavanjem *binding* adaptera, a on je izgledao ovako.

```
@BindingAdapter("items")
@JvmStatic
fun <T> setData(recyclerView: RecyclerView, data: List<T>?) {
    if (recyclerView.layoutAnimation == null) {
        recyclerView.layoutAnimation =
            AnimationUtils.loadLayoutAnimation(
                recyclerView.context,
                R.anim.layout_animation_fall_down
            )
    }
    with(recyclerView.adapter as? ListAdapter<T, *>) {
        this?.run {
            submitList(data)
        }
    }
}
```

Isječak kôda 143: *Binding* adapter za dodavanje animacije i elemenata u *recyclerView*

Ovaj adapter je primao poslane podatke kroz *data binding* te je njih pridružio *recyclerView* adapteru koji je prethodno napravljen te uz sve to je još dodao animaciju nad elementima radi boljeg korisničkog iskustva, taj dio nije potreban za funkcioniranje aplikacije.

Na kraju, spremi smo za dodavanje *recyclerViewa* koji je zaslužan za ispis svih naši transakcija unutar aplikacije, a on izgleda ovako.

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/rvListOfTransactions"
    android:layout_width="match_parent"
```

```
        android:layout_height="0dp"
        android:layout_marginTop="8dp"
        app:items="@{viewModel.filteredData}"

    app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"
        app:layout_constraintTop_toBottomOf="@id/llHeader"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintEnd_toEndOf="parent" />
```

Isječak kôda 144: XML kôd *recyclerViewa*

S ovim korakom napravljeno je da se podaci mogu ispisivati unutar aplikacije kroz *recyclerView* koje je trenutno najbolje rješenje za ispisivanje velikog broja podataka. Također, napravljeni su brojni adapteri koji pomažu i ubrzavaju daljnji razvoj aplikacije. Kada bi sada radili novi *recyclerView* na nekom drugom ekranu, postupak njegovog kreiranja bi bio znatno lakši nego od prvog jer veliki dio se pisao kao ponovno iskoristiv kôd (eng. reusable code).

6.1.5.2. Izrada lokalne baze podataka i rad s njom

Potom krećemo u razvoj baze podataka za koju smo koristili *Room* biblioteku koja je opisana u poglavlju 4.5.6. Prvo smo krenuli sa izradom baze podataka tj. dodajemo klasu *AppDb* koja predstavlja našu bazu podataka, a ona je na početku izgledala ovako.

```
@Database(
    entities = [DBTransaction::class],
    version = 1,
    exportSchema = false
)
@TypeConverters(
    DateTypeConverter::class
)
abstract class AppDb : RoomDatabase() {
}
```

Isječak kôda 145: Klasa *AppDb*

Kôd je toliko jednostavan da sam pogled na njega može nam reći o čemu se radi i tu je ta prednost *Room* biblioteke, ne moramo ni znati *SQLite* da bi napravili vlastitu lokalnu bazu podataka. Nakon ovoga, sve što treba je postepeno dodavati *DAO* sučelja i nadopunjavati ovu klasu. Prvo *DAO* sučelje koje je napravljeno je *TransacitionsDao* koje je sadržavalo nekoliko metoda za ispis podataka, a izgledalo je ovako.

```
@Dao
interface TransactionsDao: BaseDao<DBTransaction> {
    @Query("Select * from DBTransaction")
    fun getTransactionsLiveData() : LiveData<List<DBTransaction>>

    @Query("Select * from DBTransaction")
    suspend fun getTransactionsAsync() : List<DBTransaction>

    @Query("Select * from DBTransaction ORDER BY date")
    suspend fun getTransactionsAsyncSortedByDate() :
                List<DBTransaction>

    @Query("Select * from DBTransaction ORDER BY contents")
    suspend fun getTransactionsAsyncSortedByContents() :
                List<DBTransaction>

    @Query("Select * from DBTransaction ORDER BY category")
    suspend fun getTransactionsAsyncSortedByCategory() :
                List<DBTransaction>
    @Query("Select * from DBTransaction ORDER BY totalPrice")
    suspend fun getTransactionsAsyncSortedBySpend() :
                List<DBTransaction>
}
```

Isječak kôda 146: Sučelje *TransactionDao*

Može se primijetiti da u funkcijama koje dohvaćaju podatke asinkrono koristi se ključna riječ *suspend,* a ona označava da su korištene *cooroutines* koje su objašnjene u poglavlju 4.5.1., a mogu se pronaći na više mjesta unutar ovog projekta. Osim ovog *DAO* sučelja dodano je i *BaseDao* sučelje koje svako *DAO* sučelje implementira te uz pomoć njega možemo raditi određene funkcije koje su iste za sva *DAO* sučelja, a to su na primjer dodavanje modela, dodavanje liste modela, brisanje jednog ili više modela, itd. Detalje vezane u *BaseDao* sučelje možemo pronaći u poglavlju 4.5.6. Nakon toga, krećemo sa izradom skladišta (eng. storage) koji će prenositi metode između repozitorija i *DAO* sučelja. Unutar njega možemo raditi razne manipulacije i dorade nad podacima kako bi što bolje odgovarali podacima unutar aplikacije, a ovako izgleda naš *storage.* Također, unutar njega možemo vidjeti kako koristimo metode *BaseDao* sučelja.

```
class TransactionsStorage(private val dao: TransactionsDao) {

    fun getTransactionsLiveData() : LiveData<List<DBTransaction>>{
        return dao.getTransactionsLiveData()
    }

    suspend fun getTransactionsAsync(): List<DBTransaction>{
        return dao.getTransactionsAsync()
    }

    suspend fun getTransactionsAsyncSortedByDate():
```

```
List<DBTransaction>{
        return dao.getTransactionsAsyncSortedByDate()
    }
    suspend fun getTransactionsAsyncSortedByContents():
List<DBTransaction>{
        return dao.getTransactionsAsyncSortedByContents()
    }

    suspend fun getTransactionsAsyncSortedByCategory():
List<DBTransaction>{
        return dao.getTransactionsAsyncSortedByCategory()
    }

    suspend fun getTransactionsAsyncSortedBySpend():
List<DBTransaction>{
        return dao.getTransactionsAsyncSortedBySpend()
    }

    suspend fun save(model: DBTransaction) {
        dao.insertModel(model)
    }
}
```

Isječak kôda 147: Skladište (eng. storage) *TransactionStorage*

Poslije *storage* izrađujemo repozitorij unutar kojeg ćemo skupljati podatke iz *storage* i servisa, no u našem slučaju ništa ne primamo s servera pa tako imamo samo podatke iz lokalne baze. Kada bi imali oboje, unutar repozitorija bi mogli raditi sinkronizaciju podataka tako da lokalna baza uvijek bude ažurirana kao i baza na serveru. Ovaj repozitorij je veoma jednostavan i izgleda ovako.

```
class TransactionsRepository(private val storage:
TransactionsStorage) {

    val transactions: MutableLiveData<List<DBTransaction>?> =
        MutableLiveData(storage.getTransactionsLiveData().value)

    suspend fun refreshTransactions() {
        this.transactions.postValue(storage.getTransactionsAsync())
    }

    suspend fun saveTransaction(model: DBTransaction) {
        storage.save(model)
    }

    suspend fun getTransactionsAsyncSortedByDate() {
        this.transactions
          .postValue(storage.getTransactionsAsyncSortedByDate())
    }

    suspend fun getTransactionsAsyncSortedByContents() {
        this.transactions
          .postValue(storage.getTransactionsAsyncSortedByContents())
    }
```

```
    suspend fun getTransactionsAsyncSortedByCategory() {
        this.transactions
          .postValue(storage.getTransactionsAsyncSortedByCategory())
    }

    suspend fun getTransactionsAsyncSortedBySpend() {
        this.transactions
          .postValue(storage.getTransactionsAsyncSortedBySpend())
    }
}
```

Isječak kôda 148: Repozitorij *TransacitionsRepository*

6.1.5.3. Naknadno dodavanje ovisnosti

Sada kad imamo sve spremno za unošenje i ispis podataka, vrijeme je da dodamo *dependency injection* unutar aplikacije kako bi mogli lakše baratati sa određenim funkcijama aplikacije. Svi *dependency injectioni* smješteni su unutar jednog paketa pod nazivom „di“, a unutar njega imamo nekoliko modula koji su označeni imenom onog čega predstavljaju, pogledati sliku 15. Za *dependency injectio*n korištena je *Koin* biblioteka koja je objašnjena u poglavlju 4.5.4. Primjer jednog modula koji se odnosi na *viewModele* izgleda ovako, svaki drugi modul je sličan samo injektira različite klase.

```
val viewModelModule = module {
    viewModel { TransactionsViewModel(get()) }
    viewModel { StatsViewModel(get()) }
    viewModel { AuthViewModel(get()) }
}
```

Isječak kôda 149: Modul *viewModel* klasa za *Koin*

Jedina razlika između modula je modul za bazu podataka. Unutar tog modula izrađujemo početnu bazu podataka koja je kreira prilikom prvog pokretanja aplikacije. Ukoliko se aplikacija deinstalira ili se obrišu njezini podaci, zajedno s njima nestaje i baza podataka. Također, unutar modula se mogu dodavati i različite metode koje želimo da se pokrenu prilikom pokretanja aplikacije. Tako smo unutar modula za bazu podataka dodali metodu koja će nam ubaciti nekoliko podataka u bazu podataka kako bi mogli provjeriti da li ispis podataka radi, prije nego što zapravo imamo stvarni unos podataka, kako izgleda modul za bazu podataka pogledajmo u nastavku.

```
val databaseModule = module {
    single {
        Room.databaseBuilder(androidApplication(), AppDb::class.java,
```

```
"transaction_database")

.allowMainThreadQueries().fallbackToDestructiveMigration()
            .build()
    }

    single { get<AppDb>().transactionsDao }
    single { get<AppDb>().userDao }
    /**
     * Dodavanje podataka prije instalacije aplikacije. Može biti
korisno kada želimo testirati ispis podataka iz baze prije nego što
imamo stvarni upis
     */
    @RequiresApi(Build.VERSION_CODES.O)
    fun prepopulateDb(db: AppDb) {
        val dateFormat =
DateFormat.getDateInstance(DateFormat.MEDIUM, Locale.GERMAN)
        val data =
            listOf(
                DBTransaction("Groceries from Lidl", LocalDate.now(),
                              "Food", 157.31),
                DBTransaction("Coffee with friends", LocalDate.now(),
                              "Social life", 11.51),
                DBTransaction("Groceries from Konzum",
                              LocalDate.now(), "Food", 67.11),
                DBTransaction("New PC", LocalDate.now(), "Job",
                              2578.24)
            )

        GlobalScope.launch {
            db.transactionsDao.insertModels(data)

        }
    }
}
```

Isječak kôda 150: Modul za bazu podataka koristeći *Koin*

Kako bi nam svi moduli bili dostupni odmah prilikom pokretanja aplikacije, izrađujemo novu klasu imena „App“ koja nasljeđuje Kotlinovu klasu *Application* te unutar nje pozivamo metodu *startKoin* koja pokreće (eng. load) sve module koje joj proslijedimo, a ona izgleda ovako.

```
class App : Application() {
    override fun onCreate() {
        super.onCreate()
        startKoin {
            androidContext(this@App)
            loadKoinModules(
                listOf(
                    viewModelModule,
                    repositoryModule,
                    databaseModule,
                    storageModule
                )
            )
        }
    }
}
```

Isječak kôda 151: Klasa *App*

Kada smo nju definirali potrebno ju je postaviti unutar *Android manifesta* kako bi se pokretala odmah prilikom pokretanja aplikacije, a to činimo tako da unutar *Android manifesta* dodamo sljedeću liniju kôda: `android:name=".App".`

Sada kada želimo koristiti određenu klasu potrebno je samo injektirati njezin modul u odabranu klasu, recimo da bi koristili *TransactionsViewModel* unutar *TransactionsFragment* napišemo samo sljedeću liniju.

```
private val viewModel: TransactionsViewModel by inject()
```

Isječak kôda 152: Injektiranje *viewModela* koristeći *Koin*

Ovo je samo jedan primjer pozivanja modula, može se pozvati na još mnogo drugih načina, ali više o tome možemo pronaći u Koin službenoj dokumentaciji.

6.1.5.4. Spajanje upisa i ispisa podataka transakcija

Nakon ovog dijela, podešavanje arhitekture aplikacije te sveukupna priprema projekta za rad je velikim dijelom gotova, naravno s razvojem aplikacije trebati će se i cjelokupna arhitektura širiti, ali od sada je to samo više-manje uvijek isti proces. Kada smo sve to podesili, vrijeme je da nastavimo sa izradom stvarne funkcionalnosti te ju dovesti u stanje da se može koristiti i vidjeti. Kako je za ispis transakcija skoro sve pripremljeno, potrebno je samo nadopuniti *viewModel* sa metodama za dohvaćanje podataka iz repozitorija te ih ispisivati na našem ekranu. Uz to dodali smo i statični *LiveData* koji je predstavljao kategorije te sve *EditText-ove* sa *layouta* povezali smo putem *LiveData* kako bi mogli raditi s njima. Napomena da ovaj *viewModel* ne služi samo za ispis i dohvaćanje transakcija nego isto i za dodavanje novih transakcija bez obzira što se to radi u drugom fragmentu, fragmenti su tu samo kako bi prikazali korisniku aplikacije što je potrebno, odnosno, traženo od njega. Također preko *MediatorLiveData* je napravljeno da se popis transakcija može sortirati prema određenom atributu, a dohvaćanje sortiranih podataka već smo mogli vidjeti u *TransactionDao* sučelju. Dok nije implementirano dodavanja transakcija koristili smo lažne podatke, odnosno ručno unesene podatke. Nakon još nekoliko sitnih preinaka dobili smo gotovu prvu funkcionalnost aplikacije, a kako ona izgleda pogledajmo na slici 18, a kôd njezinog *viewModela* koji sadržava svu pozadinsku logiku možemo pronaći u prilogu 1.

Slika 19: Ispis transakcija

6.1.5.5. Dodavanje transakcije

Da bi u potpunosti imali gotovu funkcionalnost rada sa transakcijama treba nam još dodavanje transakcija te to razvijamo sljedeće. Sada je puno lakše razvijati dalje aplikaciju, za ovu funkcionalnost dovoljno je dodati novi fragment sa pripadajućim *layoutom* te unutar postojećeg *viewModela* dodati metodu za spremanje transakcija koja će pozivati metodu već pripremljenu unutar repozitorija i proslijediti joj potrebne parametre, a ona izgleda ovako.

```
private fun saveTransaction(dbTransaction: DBTransaction) {
    suspendCall {
        repository.saveTransaction(dbTransaction)
    }
}
```

Isječak kôda 153: Metoda za spremanje transakcija

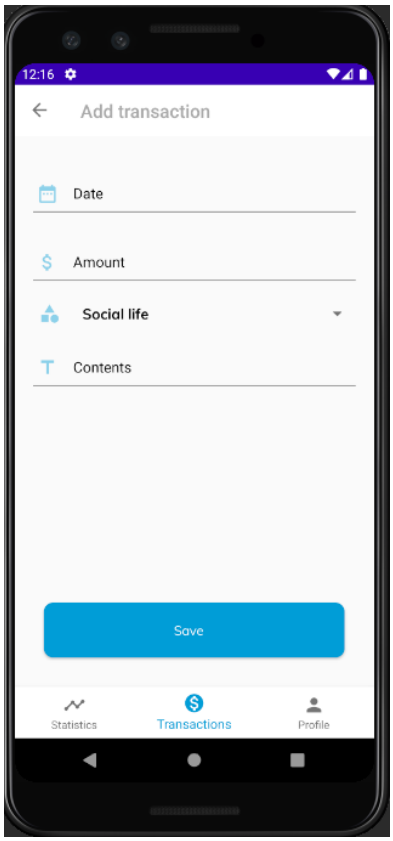
Kada smo to izradili na lebdeći gumb(pogledati sliku 18) pridružimo da se otvara novi izrađeni fragment te unutar njega na pritisak gumba „Save“ pozivamo pripremljenu metodu za spremanje transakcije te joj proslijedimo sve podatke iz popunjenih polja. Kako bi pridružili određenu akciju na gumb moramo mu unutar njegovih atributa dodati sljedeću liniju: `android:onClick="@{(v) -> viewModel.onClick(v)}"` .Ova linija u *viewModel* prosljeđuje *view* od elementa te onda unutar *viewModela* putem *id* elementa pozivamo potrebni kôd, kako to izgleda u *viewModelu* pogledajmo u nastavku.

```
@RequiresApi(Build.VERSION_CODES.O)
fun onClick(view: View) {
    when (view.id) {
        R.id.btnSave -> {
            val formatter = DateTimeFormatter.ofPattern(
                 "dd/MM/yyyy",
                 Locale.GERMAN
                 )
             val ld = LocalDate.parse(
                 dateLiveData.value.toString(),
                 formatter
                 )
            Log.d("Date", ld.toString())
            saveTransaction(
                DBTransaction(
                    contentsLiveData.value.toString(),
                    ld,
                    selectedCategory.value.toString(),
                    amountLiveData.value!!.toDouble()
                )
            )
            view.findNavController(.navigate(
                 R.id.action_addTransactionFragment_
                 to_transactionsFragment
                 )
                 }
}
```

Isječak kôda 154: *OnClick()* metoda unutar *viewModela*

Kada smo osnovni dio napravili, još smo funkcionalnost proširili, odnosno, poboljšali tako da smo na odabir kategorije dodali *Spinner* dok je na odabir datuma dodan *DatePicker*. Kako izgleda konačni *layout* ove funkcionalnosti možemo pronaći u prilogu 2 ili na GitHub repozitoriju.

Na slici 19 možemo vidjeti kako ovaj ekran na kraju izgleda.



Slika 20: Dodavanje transakcije

Uz još nekoliko sitnih popravaka kôda i izgleda ekrana završavamo sa izradom prve i glavne funkcionalnost ove aplikacije te krećemo sa sljedećom koja nam služi za ispis statistike, a direktno je povezana sa prethodno izrađenom aplikacijom.

6.1.6. Statistika unutar aplikacije

Za ovu funkcionalnost koristili smo novu biblioteku s imenom „MPAndroidChart“ koja je javno dostupna na *GitHubu*[18], a pomaže nam u ispisivanju podataka putem raznih grafova. Prvo smo započeli sa dodavanjem ovisnosti unutar *gradle* aplikacije.

```
//MP charts
implementation 'com.github.PhilJay:MPAndroidChart:v3.1.0-alpha'
```

Isječak kôda 155: Dodavanje ovisnosti za *MPAndroidChart*

[18] Više o njoj na: https://github.com/PhilJay/MPAndroidChart

Zatim smo dodali jedan tortni graf (eng. pie chart) sa ručno unesenim podacima samo kako bi shvatili na koji način ova biblioteka radi te što sve sa njom možemo postići. Nakon toga napravili smo jednu izrazito složenu funkciju koja služi za dohvaćanje podataka iz baze podataka, ali tako da ih slaže prema kategorijama te da svaku istu kategoriju zbraja s prethodnima tako da dobijemo na kraju kategorije samo jednom ispisane. Važno je napomenuti da smo za ovo koristili novi *viewModel* u kojem smo pisali sve metode za ovu funkcionalnost, ali smo koristili stari repozitorij za dohvaćanje podataka jer smo dohvaćali iste podatke kao i prošla funkcionalnost, samo smo radili druge stvari nad njima. Kako izgleda funkcija te novo dodani *viewModel* pogledajmo u nastavku.

```
class StatsViewModel(
private val repository: TransactionsRepository) : BaseViewModel() {

    private val transactionsLiveData: LiveData<List<DBTransaction>?>
         get() = repository.transactions

    fun getAllStatsData() {
        val transactions = transactionsLiveData.value
        var pieEntriesTemp = pieEntries.toMutableList()

        transactions?.forEach { dbt ->
            if (pieEntriesTemp.isEmpty()) {
                pieEntries.add(PieEntry(
                 dbt.totalPrice.toFloat(), dbt.category.trim()))
            } else {
                var categoryAlreadyExists = false
                pieEntriesTemp.forEach { peTemp ->
                    if (peTemp.label.trim() == dbt.category.trim())
                        categoryAlreadyExists = true
                }
                if (!categoryAlreadyExists) {
                    pieEntries.add(PieEntry(
                     dbt.totalPrice.toFloat(), dbt.category.trim()))
                } else {
                    var newValue = dbt.totalPrice.toFloat()
                    pieEntriesTemp.forEach { pe ->
                        if (pe.label.trim() == dbt.category.trim()) {
                            newValue += pe.value
                            pieEntries.remove(pe)
                            pieEntries.add(
                             PieEntry(newValue, dbt.category.trim()))
                        }
                    }
                }
            }
            pieEntriesTemp = pieEntries.toMutableList()
        }
    }
}
```

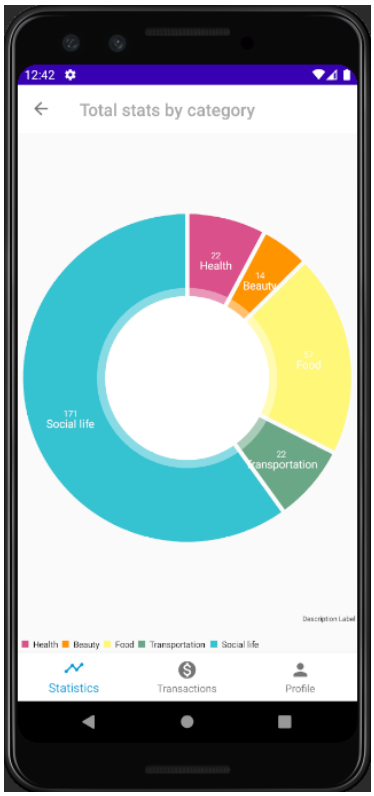
Isječak kôda 156: Početak razvoja *StatsViewModela*

Nakon toga smo još napravili jednu metodu koja nam je služila na inicijalizaciju tortnog grafa, odnosno, njoj smo proslijedili graf iz fragmenta te nad tim grafom dodali smo sva svojstva koja su nama trebala. Funkciju za inicijalizaciju tortnog grafa pogledajmo u nastavku.

```
fun initializePieChart(pieChart: PieChart) {
    Log.d("pieEntires", pieEntries.toString())
    pieDataSet = PieDataSet(pieEntries, "")
    pieData = PieData(pieDataSet)
    pieChart.legend.isEnabled = false
    pieChart.description.isEnabled = false
    pieChart.data = pieData
    pieDataSet!!.setColors(*ColorTemplate.JOYFUL_COLORS)
    pieDataSet!!.sliceSpace = 2f
    pieDataSet!!.valueTextColor = Color.WHITE
    pieDataSet!!.valueTextSize = 10f
    pieDataSet!!.sliceSpace = 5f
    // pieChart.setUsePercentValues(true)
    pieChart.animateXY(1500, 1500)
}
```

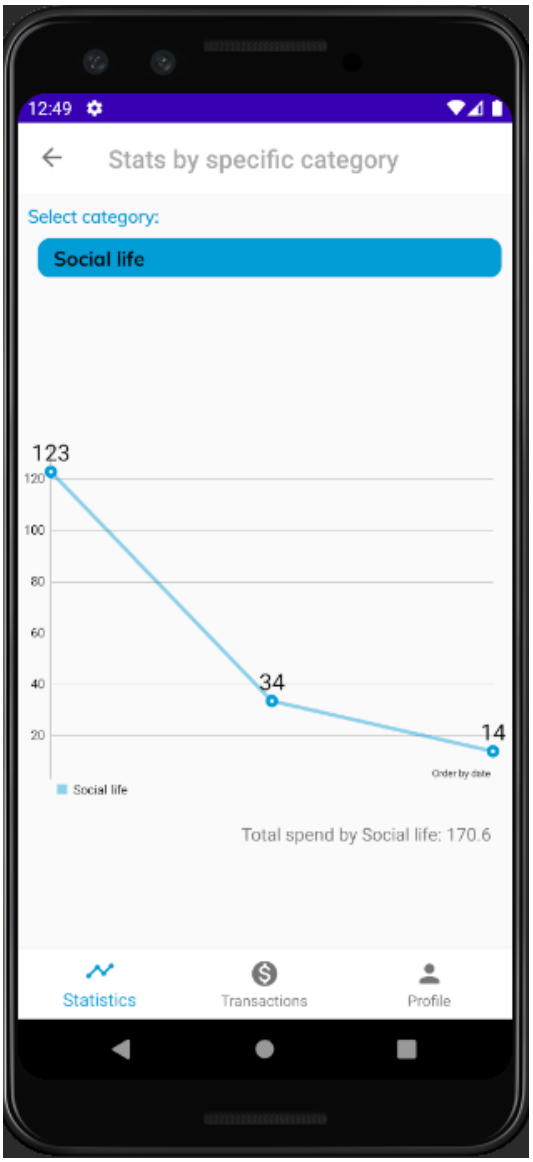
Isječak kôda 157: Metoda za inicijalizaciju tortnog grafa

Ova biblioteka stvarno ima velikih mogućnosti te se svašta može raditi s njom, ovdje je isprobano samo nekoliko osnovnih opcija koju su nama bile potrebne. Kako je ova funkcionalnost izgledala prikazano je na sljedećoj slici.

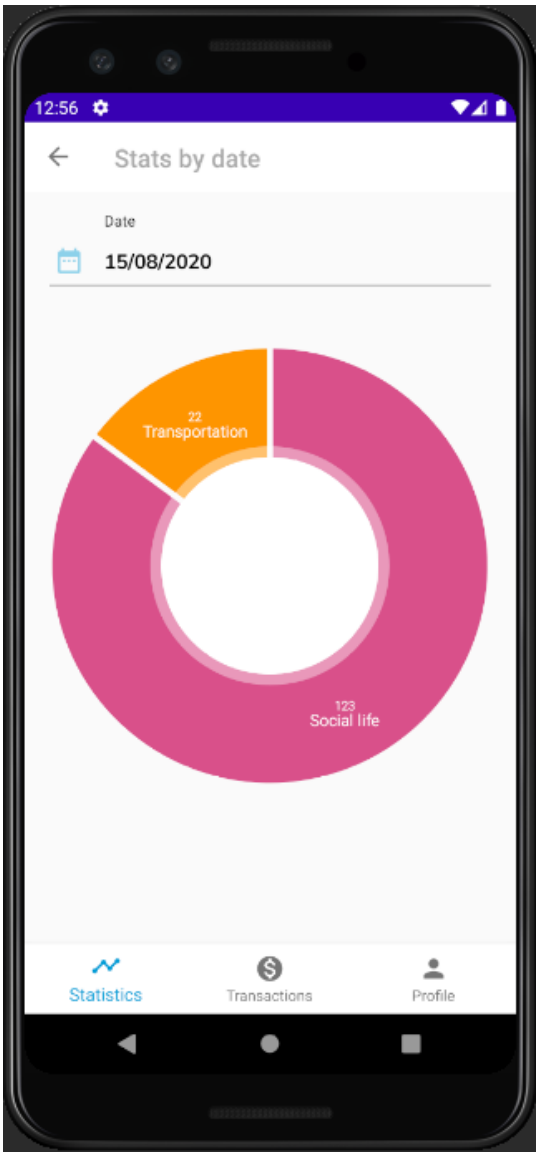


Slika 21: Tortni graf za sve transakcije sortirane prema kategorijama

Nakon što je ovo napravljeno shvaćeno je da bi ovu funkcionalnost mogli proširiti na još nekoliko grafova pa je tako napravljen novi fragment koji je sadržavao tri gumba, od koji je svaki služio za otvaranje različitih grafova. Svaki graf je imao i svoj fragment tako da je na prethodno spomenute gumbe dodana navigacijska akcija da ih vodi u određeni fragment, a gumb za nazad unutar alatne trake nas je vraćao na ekran sa izborom grafova. Od grafova su dodani još linijski graf koji pokazuje potrošnju određene kategorije sortirano prema datumu te na toj krivulji može se vidjeti kada se trošak za određenu kategoriju povećavao, a kada smanjivao. Drugi dodan graf je tortni graf koji je na odabir određenog datuma pokazao koliko smo na taj dan potrošili na određenu kategoriju, kako su oni izgledali pogledajmo na sljedećim slikama.



Slika 23: Linijski graf za pregled potrošnje prema odabranoj kategoriji

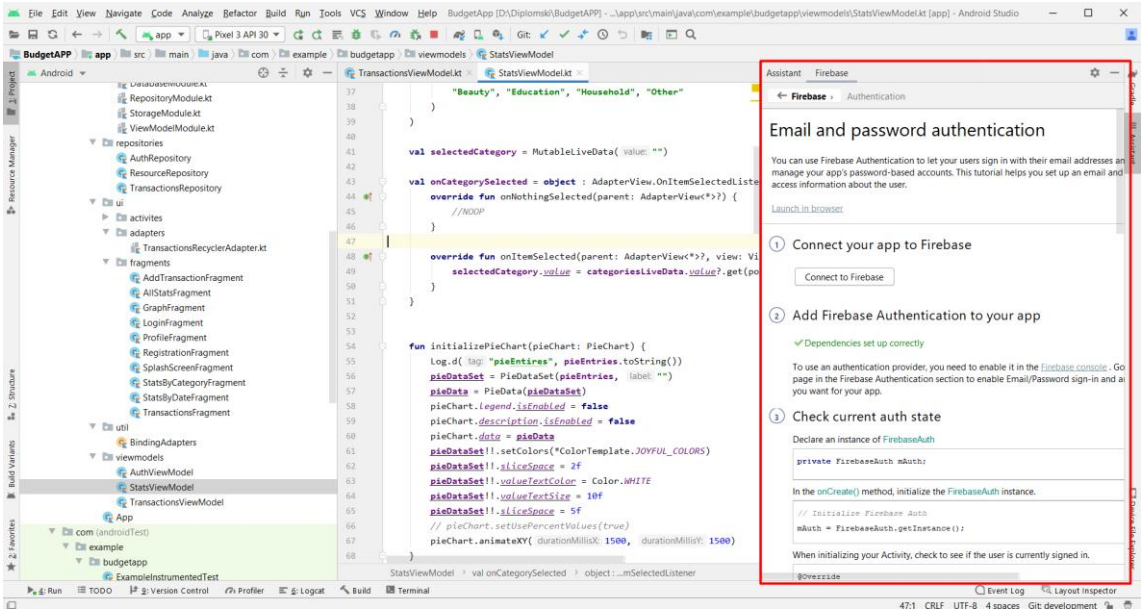


Slika 22: Tortni graf za prikaz potrošnje na određeni datum prema kategorijama

Kako je konačni *StatsViewModel* izgledao može se pronaći u prilogu pod brojem 3. Fragmenti i *layouti* za ovo su bili veoma jednostavni pa ih nećemo objašnjavati, oni su samo sadržavali osnovni kôd za ispis grafa, no ipak ih možemo pronaći na GitHub repozitoriju.

6.1.7. Prijava i registracija

Kao što je već napisano, ova aplikacija nema nikakav server na koji bi spremala i sa kojeg bi dohvaćala podatke, no ipak je napravljen određeni *Login* koji prikazuje jednostavni način prijave kako bi to trebalo izgledati unutar MVVM arhitekture. Kao što možemo pretpostaviti, sljedeća funkcionalnost ujedno je i zadnja. Za prijavu i registraciju ne koristimo nikakav vlastiti server, nego samo *Firebase* servis. Kako bi započeli implementaciju ove funkcionalnosti, najprije trebamo dodati *Firebase* servis unutar našeg projekta, a to radimo tako da kliknemo na Tools → Firebase → Authentication te potom pratimo upute koje nam direktno daje Android Studio, pogledajmo sljedeću sliku.



Slika 24: Dodavanje *Firebase* servisa

Osim autentifikacije, *Firebase* servis nudi puno raznih drugih stvari koje su veoma dobre i veoma korištene u današnje vrijeme, no više o samom servisu možemo pronaći na njegovoj službenoj stranici[19]. Za izradu prijave i registracije uz pomoć *Firebase* servisa internet je prepun članaka i video zapisa koji to objašnjavaju te u

[19] Više o *Firebase* pročitajte na: https://firebase.google.com/docs/

ovom radu neće biti toliko detaljno objašnjeno, ukoliko želimo više o tome pročitati možemo pronaći neku drugu i specifičniju literaturu za to. Kada smo sve pripremili napraviti ćemo posebnu aktivnost (eng. activity) za autentifikaciju koju smo nazvali *AuthActivity* te smo u njezin *layout* dodali fragment koji će se učitavati preko nove navigacijske komponente koju smo također izradili kao i prvu te smo ju nazvali „auth_nav_graph“. Nakon toga dodajemo dva nova fragmenta, jedan za prijavu i jedan za registraciju. Stavljamo da se fragment za prijavu automatski učitava kao početni pogleda unutar *AuthActivitya*, dok fragment za registraciju pozivamo pritiskom na *TextView* koji se nalazi na ekranu. U *LoginFragment* dodajemo varijablu *fbAuth* koja predstavlja instancu *Firebase Authentication* te uz pomoć nje pravimo novu metodu koja nam služi za prijavu u aplikaciju.

```
var fbAuth = FirebaseAuth.getInstance()
```

Isječak kôda 158: Dohvaćanje *Firebase* instance

Ova metoda uzima podatke iz polja, točnije, uzima email adresu i lozinku te ju šalje na *Firebase* server kroz već gotovu metodu *signInWithEmailAndPassword()* te ukoliko ta metoda vrati da je uspješno (eng successful) provedena onda pozivamo navigacijski kontroler koji nas šalje na *MainActivity*, odnosno, u aplikaciju. Ako ta metoda vrati da je neuspješno provedena prijava, onda ispisujemo korisniku poruku što je neispravno te da ponovi unos podataka, pogledajmo isječak kôda u nastavku.

```
private fun signIn(view: View, email: String, password: String) {
    snackbar("Authenticating...")
    fbAuth.signInWithEmailAndPassword(email, password)
        .addOnCompleteListener(requireActivity(),
         OnCompleteListener<AuthResult> { task ->
            if (task.isSuccessful) {
                viewModel.saveUser(DBUser(email))
                view.findNavController()
                 .navigate(R.id.action_loginFragment_to_mainActivity)
            } else {
                snackbar("Error: ${task.exception?.message}")
            }
        })
}
```

Isječak kôda 159: *Firebase* metoda za prijavu

Unutar *RegistrationFragmenta* imamo sve isto kao u kod *LoginFragmenta* samo što ovdje umjesto prijave korisnika, šaljemo registraciju korisnika, odnosno,

koristimo gotovu metodu *createUserWithEmailAndPassword()*, kôd možemo pogledati u nastavku.

```
private fun createUser(view: View, email: String, password: String) {
    snackbar("Registering...")
    fbAuth.createUserWithEmailAndPassword(email, password)
        .addOnCompleteListener(requireActivity(), OnCompleteListener
{ task ->
            if (task.isSuccessful) {
                viewModel.saveUser(DBUser(email))
                view.findNavController()
                 .navigate(
                  R.id.action_registrationFragment_to_mainActivity)
            } else {
                snackbar("Error: ${task.exception?.message}")
            }
        })
}
```

Isječak kôda 160: *Firebase* metoda za registraciju

Također, osim toga proširili smo funkcionalnost sa još jednim fragmentom koji će predstavljati SplashScreen. Što on točno radi je da prilikom pokretanja aplikacije prvo se pokazuje logo aplikacije dok se svi podaci ne učitaju. To poboljšava korisničko iskustvo jer korisnik neće dobiti prikaz crnog ekrana nego će znati da se aplikacija ispravno pokrenula. Također, u ovom dijelu napravljeno je da aplikacija provjeri je li korisnik ulogiran od prije te ukoliko je, automatski ga prebacuje u *MainActivity*, odnosno ne mora ponovno izvršiti prijavu. Za njega također smo koristili *AuthViewModel* u koju smo dodali sljedeću metodu.

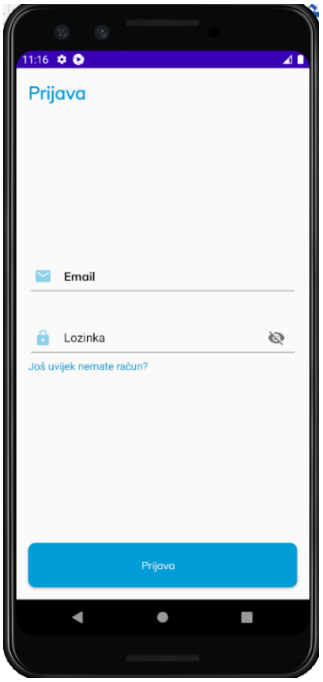
```
fun decideStartingScreen(view: View) {
    suspendCall {
        delay(1000)
        if (repository.isUserLoggedIn()) {
            try {
                view.findNavController()
                 .navigate(
                  R.id.action_splashScreenFragment_to_mainActivity)
            } catch (e: Exception) {
                repository.logout()
                view.findNavController()
                 .navigate(
                  R.id.action_splashScreenFragment_to_loginFragment)
              }
          } else {
              view.findNavController()
                  .navigate(
                   R.id.action_splashScreenFragment_to_loginFragment)
          }
         }
  }
```

Isječak kôda 161: Metoda za odlučivanje o početnom ekranu

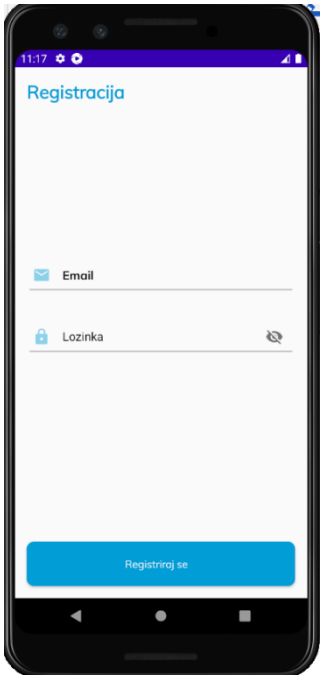
Uz sve to, prilikom uspješne prijave mi spremamo i korisnika unutar lokalne baze na mobitelu te kada otvorimo ekran sa profilom možemo vidjeti koji je korisnik trenutno prijavljen, odnosno koja se email adresa nalazi u lokalnoj bazi podataka. Također, dodan je gumb za odjavu iz aplikacije na isti ekran, a kako izgleda konačni izgled ove funkcionalnosti možemo pogledati na sljedećim slikama, kôd za odjavu koji je dodan u *AuthRepository* možemo pronaći u nastavku.

```
fun logout() {
    fbAuth.signOut()
    authStorage.logout()
}
```

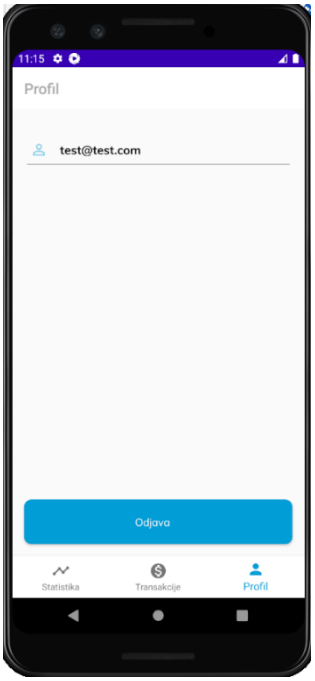
Isječak kôda 162: Metoda za odjavu iz aplikacije



Slika 24: *Prijava u aplikaciju*



Slika 25: Registracija u aplikaciji



Slika 26: Ekran profila

Ovo trenutno nema neku preveliku funkciju, ali je pripremljeno za budući razvoj aplikacije. Ukoliko bi u budućnosti napravili pravi server, odnosno, naš vlastiti *backend* servis mogli bi vrlo lako unutar ovog ekrana mijenjati korisnička svojstva kao što su npr. lozinka, broj mobitela, adresa, neke posebne značajke specijalizirane za korisnika, itd. Uz još nekoliko dodatnih provjera, dodanih nekoliko poruka korisniku te dorađenih dijelova kôda i ova funkcionalnost je gotova.

6.1.8. Lokalizacija i dizajn

Kada smo ove tri glavne funkcionalnosti završili sve ih dodajemo na *Development* granu unutar *Git* projekta te na njemu nastavljamo razvoj, odnosno, doradu aplikacije. Ukoliko bi htjeli neku novu funkcionalnost vrlo laku ju možemo dodati jer je već sve pripremljeno za daljnji razvoj aplikacije. U ovoj fazi odlučeno je da ćemo malo poboljšati dizajn koji ne pridonosi ništa funkcionalnosti aplikacije, ali veoma je bitan za korisničko iskustvo te uvelike olakšava korištenje aplikacije. Također, sav tekst unutar aplikacije dodavan je preko *strings.xml* vrijednosti te smo se odlučili za uvođenje lokalizacije aplikacije. Točnije, aplikacija će se prema zadanim postavkama pokretati na engleskom jeziku, no ako je naš mobitel postavljen na hrvatski jezik tada će i aplikacija biti na hrvatskom. Svi stilovi unutar aplikacije dodani su u datoteku *styles.xml* te su korišteni na više mjesta, to uvelike olakšava razvoj te također omogućava nam da budemo konstanti u dizajnu naše aplikacije. Uz sve to, korištene su i zadane dimenzije unutar datoteke *dimens.xml* te boje aplikacije unutar *colors.xml.*

6.2. Testiranje aplikacije

6.2.1. Jedinično testiranje

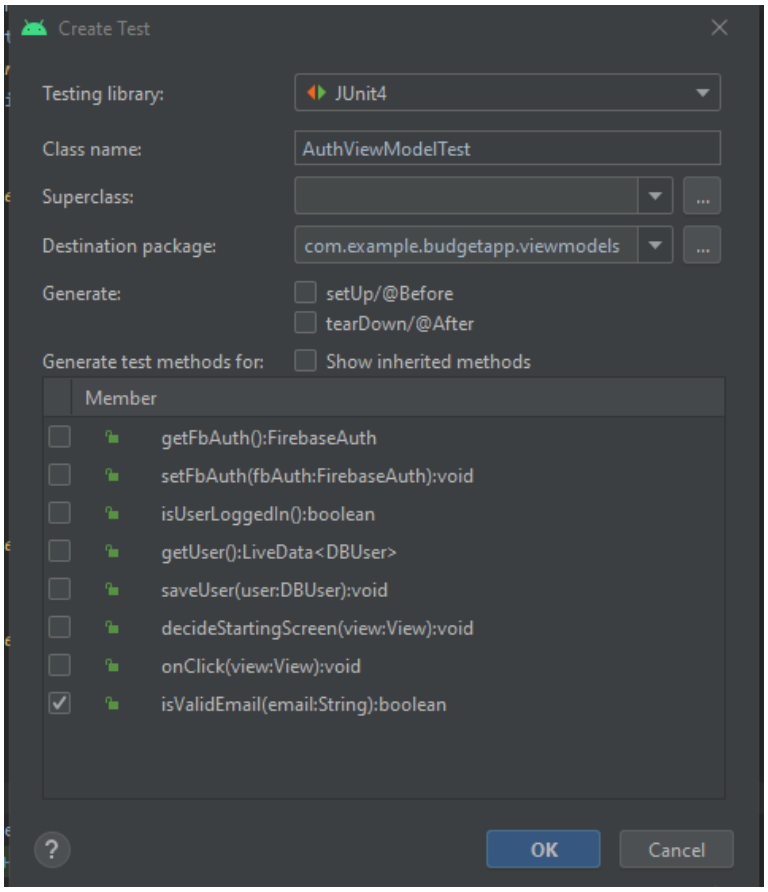
Za potrebu jediničnog testiranja (eng. unit test) pripremiti ćemo jednu metodu koja će uz pomoć *RegEx* kôda označavati je li email ispravno upisan. Iako, *Firebase* već u svojoj metodi ima tu provjeru, napraviti ćemo to samo da bi pokazali kako se pišu jedinični testovi u programskom jeziku Kotlin, a kako izgleda metoda koju ćemo testirati pogledajmo u nastavku.

```
fun isValidEmail(email: String): Boolean {
    val pattern = Pattern.compile(
        "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,64}"
    )
    return pattern.matcher(email).matches()
}
```

Isječak kôda 163: Funkcija za validaciju email adrese

Jedinični test možemo kreirati ručno, ali možemo ga generirati i uz pomoć Android Studia. Kako bi ga generirali potrebno je osjenčati odabranu klasu te na nju pritisnuti desni klik miša potom odabrati „Go To“ → „Test“, nakon toga ukoliko test već

ne postoji Android Studio će nam ponuditi da napravimo novi te dobijemo prozor kao na slici ispod.



Slika 25: Prozor za dodavanje jediničnog testa

Unutar prozora pod „Testing library“ odabrati ćemo JUnit4 jer ćemo tu biblioteku koristit za pisanje testova, naravno tu se može odabrati i neka druga. Kada smo to napravili, dobiti ćemo samo okvir testne klase te trebamo nastaviti pisati naše testove. Pošto mi testiramo metodu koja se nalazi unutar *viewModela* moramo napraviti i instancu tog istog *viewModela* kako bi mogli koristiti njezine metode. U testiranju kada nam trebaju instance nekih klasa, a one zahtijevaju određene resurse onda koristimo tzv. oponašajuće varijable (eng. mock), a u naše slučaju za tu svrhu koristimo biblioteku Mockk koja je opisana u poglavlju 5.2. No, prije toga napraviti ćemo jednu apstraktnu testnu klasu koja će nam pokretati *Koin* biblioteku za *dependancy injection* te ju na kraju i zaustaviti. Drugim riječima, unutar *init* bloka naredbe pozivamo *startKoin()* metodu kojoj prosljeđujemo apstraktnu varijablu *testCaseMocksModule* koja predstavlja skup svih modula koje testna klasa treba proslijediti, a unutar metode *tearDown()* koja je anotirana sa *@After* pozivamo metodu *stopKoin()*. Uz to sve, još smo omogućili da testna klasa koja bude nasljeđivala ovu apstraktnu klasu može

dodati još svoje *before()* i *after()* naredbe te da može pozivati *dispacher* kada budemo testirali *LiveData* i *Cooroutine,* zvuči veoma komplicirano, ali na kraju to izgleda ovako sasvim jednostavno.

```
abstract class TestCase: KoinTest {

    @JvmField @Rule
    val instantTaskExRule = InstantTaskExecutorRule()

    abstract val testCaseMocksModule: Module

    open fun before() {}
    open fun after() {}

    init {
        startKoin {
            androidContext(mockk())
            loadKoinModules(testCaseMocksModule)
        }
    }

    @Before
    fun setUp() {
        before()
    }

    @After
    fun tearDown() {
        stopKoin()
        after()
    }

}
```

Isječak kôda 164: Abstraktna klasa *TestCase*

Nakon toga, u našoj početnoj testnoj klasi naslijedimo ovu klasu te također naslijedimo i testnu klasu za *Koin* koja se zove *KoinTest.* Zatim implementiramo varijablu *testCaseMocksModule* te joj napravimo *getter* u kojem ćemo dodati *dependency injectione* koji nam trebaju, u ovom slučaju to su *AuthViewModel* i *AuthRepository*. Deklariramo novu varijablu koja će nam predstavljati instancu *AuthViewModela*. Pošto nam *AuthViewModel* traži parametar, točnije trebamo mu proslijediti repozitorij, dovoljno je samo da umjesto parametra pozovemo metodu *get()* i *KoinTest* će automatski proslijediti klasu koja je potreba, ali ona mora biti dodana unutar *testCaseMockModule* varijable. Sve je spremno za pisanje naših testova, ovdje smo ukratko napisali nekoliko situacija kao što su neispravna email adresa, ispravna email adresa i proslijedili prazan *String*, naša testna klasa ovako izgleda.

```
class AuthViewModelTest : TestCase(), KoinTest {

    override val testCaseMocksModule: Module
        get() = module {
            single<AuthViewModel> { mockk(relaxed = true) }
            single<AuthRepository> { mockk(relaxed = true) }
        }
    private val authViewModel = AuthViewModel(get())

    @Test
    fun isValidEmailTest() {
        assert(!authViewModel.isValidEmail("wrongemail#f"))
        assert(authViewModel.isValidEmail("test2@email.com"))
        assert(!authViewModel.isValidEmail(""))
    }
}
```

Isječak kôda 165: Testna klasa *AuthViewModelTest*

Sada ćemo provesti još jedno testiranje za *TransactionsViewModel* da se može vidjeti koliko je sada lako dodati jedinični test nakon što je sve pripremljeno. Testirati ćemo metodu koja dohvaća popis transakcija iz lokalne baze podataka, s tim da ćemo joj kreirati novu bazu podataka i u nju unijeti statičke podatke kako bi točno znali što imamo u bazi i na taj način mogli točno provesti testiranje. Ponovno prvo postavljamo module koji su nam potrebni, u ovo slučaju to su: *TransactionsRepository, AppDb, TransactionsDao* te postavljamo *dispatchera* pri pokretanju testa te ga resetiramo prilikom završetka testa. Izrađujemo jednu statičku varijablu tipa *DBTransaction* koju ćemo dodati u bazu podataka, odnosno, namjestiti ćemo da repozitorij uvijek vraća listu sa tom stavkom u njoj. Kreiramo testnu metodu koju stavljamo da se pokreće u *coroutine* području rada te u njoj izrađujemo jedan *observer* koji će nam predstavljati naš pravi *observer*, odnosno, njega ćemo testirati. Pozivamo metodu iz *viewModela* koja nam dohvaća sve transakcije te potom provjeravamo je li se napravljeni *observer* promijenio ili je ostao isti, kako to izgleda u kôd pogledajmo ispod te također i komentare na pojedinu liniju.

```
class TransactionsViewModelTest : TestCase() {
    // inicijaliziramo viewModel
    private val transactionsViewModel = TransactionsViewModel(get())
    // inicijaliziramo repozitorij
    private val transactionsRepository get() =
         get<TransactionsRepository>()

    override val testCaseMocksModule: Module //lista potrebnih modula
        get() = module {
            single<TransactionsRepository> { mockk(relaxed = true) }
            single<AppDb> { mockk(relaxed = true) }
```

```
                single<TransactionsDao> { mockk(relaxed = true) }
            }

        private val testDispatcher = TestCoroutineDispatcher()

        override fun before() {
            super.before()
            Dispatchers.setMain(testDispatcher)
        }

        override fun after() {
            super.after()
            testDispatcher.cleanupTestCoroutines()
            Dispatchers.resetMain()
        }
        // statički podaci
        val dbTransactionTemp = DBTransaction("Groceries",
                                    LocalDate.now(), "Food", 15.68)

        @Test
        fun getTransactionsLiveData() = runBlockingTest {
            // svaki poziv na tu metodu će uvijek vratiti tu listu
            // "every" je metoda biblioteke Mockk
            every { transactionsRepository.transactions }
                      returns MutableLiveData(
                listOf(
                    dbTransactionTemp
                )
            )
            // kreiramo observer koji promatramo
            val observer =
              mockk<Observer<List<DBTransaction>?>>(relaxed = true)

            // poziv metode iz viewModela
     transactionsViewModel.transactionsLiveData.observeForever(observer)
            // provjeravamo je li se promijenio u vremenu od 5 sekundi
            verify(timeout = 5000) { observer.onChanged(any()) }
        }
    }
```

Isječak kôda 166: Testna klasa *TransactionsViewModelTest*

Možemo vidjeti koliko je jednostavno sada pisati prave jedinične testove te koliko nam ovakav način pisanja kôda olakšava testiranje. Arhitekture pisanja kôda kao MVC, MVVM i MVP primarno su napravljene za Android baš zato da odvoje poslovnu logiku aplikacije od onoga što pokazujemo korisniku kako bi lakše mogli testirati i time umanjili vrijeme pisanja testova i poboljšali cjelokupni rad aplikacije. Kada bi sada htjeli napraviti test za neku drugu metodu ovog istog *viewModela* samo mi dodali novu testnu metodu, definirali što želimo da nam repozitorij vraća te što promatramo. Na kraju, samo provjeravamo što smo dobili i je li se to poklapa sa stvarnom specifikacijom aplikacije. U ovoj aplikaciji nismo nikako povezani sa nikakvim servisom, ali na ovaj način bi isto testirali i servise koje koristimo. Možemo pogledati

sljedeće prikazani kôd koji nije dio ove aplikacije nego jedne druge koja koristi javno dostupne API servise, ovaj primjer samo služi kako bi lakše mogli shvatiti koliko je to zapravo sada jednostavno.

```
@Test
fun createMessageOnSuccess() = runBlockingTest {
    coEvery { repositoryContact.createMessage(any()) }
                returns Response.success(Unit)
    val observer = mockk<Observer<String>>(relaxed = true)
    contactViewModel.errorMessage.observeForever(observer)
    contactViewModel.callCreateMessage(contactTest)
    verify(timeout = 5000) { observer.onChanged(any()) }
}
```

Isječak kôda 167: Testiranje API servisa

6.2.2. Instrumentacijsko testiranje

Sada kada je pokazano kako izvršiti jedinično testiranje nastaviti ćemo sa instrumentacijskim testom u kojem ćemo se najviše usredotočiti na testiranje baze podataka. Ovu testnu klasu napraviti ćemo isto kao i prve dvije, samo što ćemo ovu smjestiti u drugu mapu naziva „androidTest“, a nalazi se unutar „java“ mape. Ovu testnu klasu nazvat ćemo *TransactionDatabaseTest* te ćemo unutar nje dodati instancu klase *InsatantTaskExecutorRule()* te instancu naše baze, *TransactionDao* te modela *DBTransaction*. Prije svega pokrenuti ćemo metodu *createDb()* koja će nam dohvatiti kontekst aplikacije, kreirati bazu podataka te dodati jedan model u bazu podataka nad kojim ćemo izvršavati testove. Ovu metodu ćemo anotirati sa *@Before* jer želimo da se onda odmah izvrši. Osim nje napraviti ćemo i metodu *closeDb()* koja će nam zatvarati našu bazu podataka na kraju testiranja te ćemo nju anotirati sa *@After*. Nakon svega toga možemo dodavati testove na metode koje želimo testirati. Prvo ćemo testirati metodu za dohvaćanje svih transakcija iz baze te ćemo provjeriti da li broj tih transakcija odgovara broju modela koje smo mi dodali u bazu, pa onda ta cijela klasa izgleda ovako.

```
class TransactionDatabaseTest {
    @JvmField
    @Rule
    val instantTaskExRule = InstantTaskExecutorRule()

    private lateinit var transactionsDao: TransactionsDao
    private lateinit var db: AppDb
    private lateinit var transaction: DBTransaction
```

```
    @Before
    fun createDb() {
        runBlocking {
            val context =
                 InstrumentationRegistry
                 .getInstrumentation().targetContext
            db = Room.inMemoryDatabaseBuilder(context,
                 AppDb::class.java).allowMainThreadQueries().build()
            transactionsDao = db.transactionsDao
            transaction = DBTransaction("Coffee with friends",
                           LocalDate.now(), "Social life", 25.19)
            transactionsDao.insertModel(transaction)
        }
    }

    @After
    fun closeDb(){
        db.close()
    }

    @Test
    fun getTransactions() = runBlockingTest {
        assertEquals(transactionsDao.getTransactionsAsync().size, 1)
     assertEquals(transactionsDao
                 .getTransactionsAsync()[0].category, "Social life")
        assertNotEquals(transactionsDao
                        .getTransactionsAsync()[0].totalPrice, 25)
        assertTrue("It's false!", transactionsDao
          .getTransactionsAsync()[0].contents == transaction.contents)
    }
}
```

Isječak kôda 168: Instrumentacijski test *TransactionDatabaseTest*

Sada možemo nastaviti samo dodavati metode koje želimo testirati te ih anotirati sa *@Test* i imamo odmah napravljeni test. Unutar tih metoda dodavamo provjere koje god mi želimo, sa *org.junit.Assert* bibliotekom stvarno ima provjera na pretek te svaki djelić našeg kôda možemo provjeriti da li se poklapa sa onime što se od aplikacije traži, točnije sa onime što piše unutar specifikacija aplikacije. Važno je napomenuti da za instrumentacijske testove potreban je uređaj ili emulator nad kojim ćemo ih izvršavati dok jedinične testove možemo izvršavati samo nad kôdom, bez pokretanja aplikacije. Za više testova i provjera nad ovom aplikacijom možemo pronaći u izvorni kôd koji je dostupan na autorom GitHub profilu.

6.2.3. Automatizirano testiranje

Kada smo i s ovim dijelom testiranja završili, zadnji korak u testiranju naše aplikacije je dakako automatizirano testiranje. Za automatizirano testiranje odlučili smo se za *Appium* jer prema našem mišljenju ima puno prednosti nad *Selendroidom*, a i

veoma je lakši i jednostavniji za korištenje. Kako bi proveli ovaj dio testiranja, prvo trebamo instalirati *Appium* i sve njegove komponente. Ovaj korak nećemo objašnjavati u ovom radu, ali možemo ga pronaći u literaturi („Appium tutorial, 2020) gdje točno piše i kako se može spojiti sa Android Studiom. Uz prethodno spomenutu literaturu potrebno je još dodati nekoliko stavki kako bi mogli koristiti Kotlin u pisanju automatskih testova, a taj dio možemo pronaći literaturu pod brojem („Writting an Appium test in Kotlin“, 2020).

Za provođenje automatskog testiranja odlučeno je da će se testirati funkcionalnost registracije i prijave, s tim da će se više vremena posvetiti prijavi. Ova funkcionalnost je izabrana iz razloga jer je to funkcionalnost koja je veoma bitna unutar aplikacije, a nalazi se gotovo u svim današnjim aplikacijama. Također, jedan od razloga je i taj što je to neprestano ponavljajuća akcija koju automatizirani test može s lakoćom izvoditi, a nama developerima ili testerima će uvelike uštedjeti vremena.

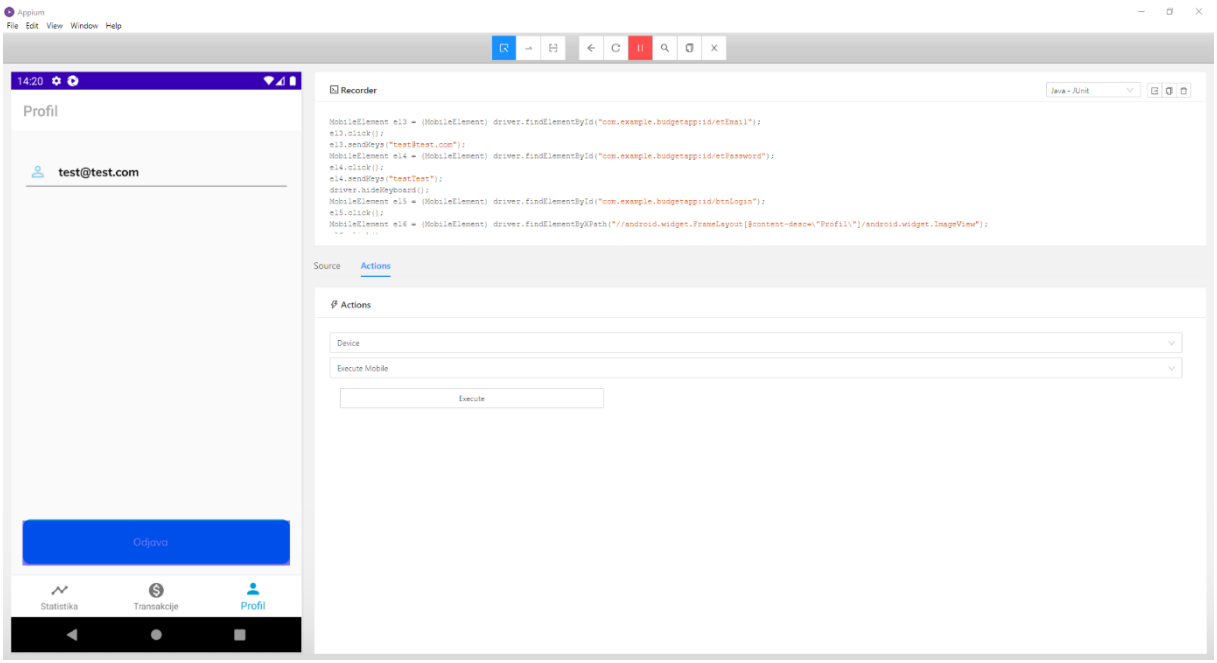
Test možemo napraviti pomoću *Appium* korisničkog sučelja ili jednostavno pišući test kao i sve prethodne, ovdje ćemo pokazati prvi način. Prvo ćemo pripremiti jednu testnu klasu u kojoj ćemo postaviti sve parametre kako bi *Appium* uopće mogao pokrenuti aplikaciju putem emulatora, ukoliko radite testiranje putem mobitela pogledajmo literaturu („Appium tutorial“, 2020), te službenu dokumentaciju *Appiuma*. Nećemo previše ulaziti u dubinu ove klase jer je ona samo *boilerplate* kôd koji moramo imati kako bi sve normalno funkcioniralo, a to izgleda ovako.

```
open class AppiumTest {

    var driver: AppiumDriver<MobileElement>? = null

    @Before
    fun setup() {
        val capabilities = DesiredCapabilities()
        val userDir = System.getProperty("user.dir")
        val serverAddress = URL("http://127.0.0.1:4723/wd/hub")
        capabilities.setCapability(
                MobileCapabilityType.APPIUM_VERSION, "1.7.1")

capabilities.setCapability(MobileCapabilityType.PLATFORM_NAME,
"Android")
        capabilities.setCapability(
                MobileCapabilityType.DEVICE_NAME, "Android")

        val localApp = "release\\app-release.apk"
        val appPath = Paths.get(userDir, localApp)
                        .toAbsolutePath().toString()
        capabilities.setCapability(MobileCapabilityType.APP, appPath)

        driver = AndroidDriver(serverAddress, capabilities)
```

```
            driver?.let {
                it.manage()?.timeouts()?
                      .implicitlyWait(30, TimeUnit.SECONDS)
            }
        }

        @After
        fun tearDown() {
            driver?.quit()
        }
    }
```

Isječak kôda 169: Otvorena klasa za *Appium* testiranje

Potom izrađujemo novu testnu klasu koja će ovu klasu nasljeđivati, a u njoj ćemo pisati sve ostale testove. Za test koristiti ćemo *Appium* korisničko sučelje te uz pomoć njega napisati test. Pokrećemo snimanje unutar aplikacije te sa lijeve strane možemo vidjeti emulator koji projicira naču aplikaciju. Sve što trebamo je pritiskati na tipke ili elemente naše aplikacije te izvoditi potrebne operacije. Ako želimo pritisnuti na neki element prvo odaberemo element potom kliknemo s desne strane na gumb „Tap“. Ako želimo upisati neke podatke unutar polja odaberemo „Send keys“ te potom unesemo što želimo upisati. Također, imamo i uobičajene akcije nad mobilnim uređajem koje možemo pogledati kada odaberemo „Actions“, a one mogu biti npr. prikazivanje notifikacija, spuštanje i dizanje tipkovnice, gašenje interneta, resetiranje aplikacije, slanje aplikaciju u pozadinu, itd. Sada samo redoslijedom odabiremo elemente te unosimo sve što nam je potrebno za prijavu. Kada smo se uspješno prijavili, još odlazimo na ekran sa podacima o profilu kako bi potvrdili uspješnu prijavu. Kada smo završili sa našim testom, sve naše korake možemo pronaći u dijelu prozora gdje piše „Recorder“. Sve što trebamo je odabrati jezik da bude „Java-JUnit“ te kopirati sve te elemente, kako prozor izgleda pogledajmo na sljedećoj slici.



Slika 26: *Appium* prozor za snimanje testa

Kada kopiramo ovaj kôd unutar Android Studia, on će nam ponuditi da automatski prevede kôd u Kotlin programski jezik i dobili smo naš prvi automatski test. Napomena, test može biti napisan i u Javi, svejedno će normalno raditi, no mi ćemo se i dalje držati Kotlina. Na kraju ćemo samo dodati komandu za zaustavljanje dretve na pet sekundi kako bi stigli pogledati zadnji ekran, a test nam izgleda ovako.

```
class LoginAutomationTest : AppiumTest() {
    @Test
    fun login() {
        val el3 =
         driver!!.findElementById(
          "com.example.budgetapp:id/etEmail") as MobileElement
        el3.click()
        el3.sendKeys("test@test.com")
        val el4 =
            driver!!.findElementById(
          "com.example.budgetapp:id/etPassword") as MobileElement
        el4.click()
        el4.sendKeys("testTest")
        driver!!.hideKeyboard()
        val el5 =
            driver!!.findElementById(
          "com.example.budgetapp:id/btnLogin") as MobileElement
        el5.click()
        val el6 =
            driver!!.findElementByXPath(
                "//android.widget.FrameLayout[@content-
                        desc=\"Profil\"]/android.widget.ImageView")
                as MobileElement
        el6.click()
        // samo da bi mogli pogledati zadnji ekran
```

```
            Thread.sleep(5000)

        }
    }
```

Isječak kôda 170: Automatizirani test *LoginAutomationTest*

Sve što nam je preostalo da pokrenemo test i gledamo kako *Appium* radi sve za nas. Test se pokreće normalno kao i svi prethodni testovi. Ukoliko bi htjeli napisati vlastiti test bez pomoći aplikacije, samo trebate pozorno pratiti korake koji se događaju u aplikaciji i opisivati ih kroz programski kôd kao u gore navedenoj metodi. Znači dohvatite element sa ekrana, popunite radnju koju on treba obavljati te idete na sljedeći element, kada sve elemente završite napravite određenu radnju i to je to. Možda trenutno mislite kako je to puno posla kada je aplikacija veoma složena i treba puno elemenata definirati, no uzmite u obzir da kada jednom napišemo test više ništa ne moramo ručno testirati nego sve *Appium* obavi za nas. Također, testovi se mogu obavljati u pozadini ili na nekom drugom računalu ili serveru bez da ometaju naš rad.

7. Zaključak

Pred Kotlinom kao programskom jezikom za razvoj mobilnih aplikacija velika je budućnost. Kotlin je već u vrijeme pisanja ovog rada veoma popularan programski jezik u Android svijetu te u puno tvrtki i za puno aplikacija polako postaje i standard. Također, otkako je prihvaćen od strane *Googlea* te otkad ima njegovu potpunu podršku, njegovo korištenje i popularnost naglo je skočila. Za sve što Kotlin trenutno nema, tu je uvijek Java koja je savršeno kompatibilna sa njim te sve što ne možemo napraviti u Kotlinu možemo napraviti u Javi.

U ovom radu obrađen je veliki dio teorije Kotlina kao programskog jezika, ali i njegovih nadopuna koje se koriste u današnjem svijetu Android programiranja. Pokazane su dodatne biblioteke koje su veoma korištene te koje prate sve najmodernije svjetske trendove unutar razvoja Android aplikacija. Također, odrađeni su osnovni i najčešći oblici testiranja mobilnih aplikacija gdje je uz sve to uvršteno u automatizacija testova te sve pokazano na primjeru stvarne aplikacije.

Osim teorijskog segmenta ovog rada, nalazi se i praktični dio[20] koji obrađuje razvoj i testiranje jedne Android mobilne aplikacije. Razvoj počinje od najosnovnijih dijelova aplikacije te se postepeno proširuje i komplicira. Unutar razvoja aplikacije primjenjivane su sve metode te svi alati obrađeni u teoretskom dijelu aplikacije. Također, aplikacija je objašnjena apstraktno u smislu njezinog funkcioniranja, ali je objašnjena i kroz detalje programskog kôda koji su posebno istaknuti i objašnjeni. Aplikacija je također provedena kroz sve korake testiranja, točnije, ima ručno testiranje, jedinično testiranje, instrumentacijsko testiranje te napravljen automatski test koristeći *Appium*. Sva testiranja su detaljno opisana i objašnjena u poglavlju 6 ovog rada.

Glavna svrha ovog rada je obrada teme koja se može iščitati iz naslova, ali također može služiti kao uvod za početnike u svijet Android programiranja te im može pokazati kako se danas u svijetu to radi. Korisnici koji ovaj rad pročitaju i obrade sve primjere navedene ovdje imati će veoma dobru podlogu za daljnje učenje i napredovanje u svijetu razvoja i testiranja Android aplikacija. Također, ovaj rad se može koristiti kao paravan za izradu prve aplikacije u Kotlin programskom jeziku za sve ljude koji su upoznati sa razvojem aplikacija za Android operacijski sustav, ali nikad nisu imali prilike raditi u Kotlinu.

[20] Link na GitHub repozitorij: https://github.com/vgrbavac/BudgetAPP

Popis literature

[1] Leiva Antoinio. (2017). *Kotlin for Android Developers*. Leanpub

[2] Developers.android.com. (2020).

a. *Develop Android Apps with Kotlin*. Preuzeto 22.04.2020. s https://developer.android.com/kotlin
b. *LiveData Overview*. Preuzeto 27.05.2020. s https://developer.android.com/topic/libraries/architecture/livedata
c. *Room Persistence Library*. Preuzeto 28.05.2020. s https://developer.android.com/topic/libraries/architecture/room
d. *Get started with the Navigation component*. Preuzeto 28.05.2020. s https://developer.android.com/guide/navigation/navigation-getting-started
e. *Build local unit tests*. Preuzeto 02.06.2020. s https://developer.android.com/training/testing/unit-testing/local-unit-tests
f. *Data Binding Library*. Preuzeto 25.05.2020. s https://developer.android.com/topic/libraries/data-binding

[3] Jemerov D. i Isakova S. (2017). *Kotlin in Action.* Mannaning Publications Co.

[4] Kotlinlang.org. (2020).

a. *Using Kotlin for Android Development.* Preuzeto 25.04.2020. s https://kotlinlang.org/docs/reference/android-overview.html
b. *Comparison to Java Programming Language.* Preuzeto 25.04.2020. s https://kotlinlang.org/docs/reference/comparison-to-java.html
c. *Extensions.* Preuzeto 26.04.2020. s https://kotlinlang.org/docs/reference/extensions.html
d. *Higher-Order Functions and Lambdas.* Preuzeto 26.04.2020. s https://kotlinlang.org/docs/reference/lambdas.html
e. *Functions.* Preuzeto 26.04.2020. s https://kotlinlang.org/docs/reference/functions.html
f. *Declaring variables.* Preuzeto 27.04.2020. s https://kotlinlang.org/docs/tutorials/kotlin-for-py/declaring-variables.html

g. *Properties and Fields,* preuzeto 28.04.2020. s https://kotlinlang.org/docs/reference/properties.html
h. *Interfaces.* Preuzeto 28.04.2020. s https://kotlinlang.org/docs/reference/interfaces.html
i. *Data Classes.* Preuzeto 28.04.2020. s https://kotlinlang.org/docs/reference/data-classes.html
j. *Delegated Properties.* Preuzeto 28.04.2020. s https://kotlinlang.org/docs/reference/delegated-properties.html
k. *Coroutines for asynchronous programming and more,* preuzeto 21.05.2020. sa https://kotlinlang.org/docs/reference/coroutines-overview.html

[5] Tutorialspoint.com. (2020). *Kotlin - Class & Object* Preuzeto 29.04.2020. s https://www.tutorialspoint.com/kotlin/kotlin_class_and_object.htm

[6] Baeldung.com. (2020).
a. *Guide to Kotlin Interfaces.* Preuzeto 02.05.2020. s https://www.baeldung.com/kotlin-interfaces
b. *Data Classes in Kotlin.* Preuzeto 02.05.2020. s https://www.baeldung.com/kotlin-data-classes

[7] GitHub.com. (2020).
a. *Anko.* Preuzeto 05.05.2020. s https://github.com/Kotlin/anko
b. *Discontinuing Anko.* Preuzeto 05.05.2020. s https://github.com/Kotlin/anko/blob/master/GOODBYE.md
c. *What is KOIN?.* Preuzeto 26.05.2020. s https://github.com/InsertKoinIO/koin
d. *Glide.* Preuzeto 29.05.2020. s https://github.com/bumptech/glide

[8] Voggella.com. (2019). *Using data binding in Android – Tutorial.* Preuzeto 22.05.2020. s https://www.vogella.com/tutorials/AndroidDatabinding/article.html

[9] Medium.com. (2019). *MVVM (Model View ViewModel) + Kotlin + Google Jetpack.* Preuzeto 24.05.2020 s https://medium.com/@er.ankitbisht/mvvm-model-view-viewmodel-kotlin-google-jetpack-f02ec7754854

[10] ProAndroidDev.com. (2018). *MVVM with Kotlin — Android Architecture Components, Dagger 2, Retrofit and RxAndroid* preuzeto 25.05.2020. s

https://proandroiddev.com/mvvm-with-kotlin-android-architecture-components-dagger-2-retrofit-and-rxandroid-1a4ebb38c699

[11] Inser-Koin.io. (2020). *What is Koin.* Preuzeto 26.05.2020. s https://insert-koin.io/

[12] Mockk.io. (2020). *MockK.* Preuzeto 01.06.2020. s https://mockk.io/

[13] P. Ammann i J. Offutt. (2017). Introduction to software testing. Cambridgeshire: Cambridge University Press

[14] Smartbear.com. (2020). *What is Automated Testing?.* Preuzeto 09.06.2020. s https://smartbear.com/learn/automated-testing/what-is-automated-testing/

[15] Guru99.com. (2020). *AUTOMATION TESTING Tutorial: What is, Process, Benefits & Tools* preuzeto 09.06.2020. s https://www.guru99.com/automation-testing.html

[16] Ranorex.com. (2020). *What are the benefits of automated testing?*. Preuzeto 11.06.2020. s https://www.ranorex.com/why-test-automation/

[17] Selendroid.io. (2020). *„Selendorid, selenium for Android“*. Preuzeto 12.06.2020. s http://selendroid.io/

[18] Appium.io. (2020). *Appium*. Preuzeto 15.06.2020. s http://appium.io/

[19] Medium.com. (2018). *Using The Navigation Architecture Component in Android Jetpack (Kotlin).* Preuzeto 18.08.2020. s https://medium.com/android-nuggets/using-the-navigation-architecture-component-in-android-jetpack-kotlin-48d4167ec9e5

[20] Javapoint.com. (2020). *Appium tutorial.* Preuzeto 20.08.2020. s https://www.javatpoint.com/appium

[21] Headspin.io. (2020). *Writing an Appium test in Kotlin.* Preuzeto 20.08.2020. s https://www.headspin.io/blog/appium/writing-an-appium-test-in-kotlin/

Popis slika

Popis tablica

Popis isječaka kôdova

Prilog 1. *TransactionViewModel*

```
class TransactionsViewModel(private val repository: TransactionsRepository)
: BaseViewModel() {

    val transactionsLiveData: LiveData<List<DBTransaction>?> get() =
repository.transactions


    val categoriesLiveData = MutableLiveData<List<String>>(
        listOf(
            "Social life", "Food", "Self development", "Transportation",
"Health",
            "Beauty", "Education", "Household", "Other"
        )
    )

    val amountLiveData = MutableLiveData<String?>()
    val contentsLiveData = MutableLiveData<String?>()
    val dateLiveData = MutableLiveData<String?>()

    val selectedCategory = MutableLiveData("")

    val onCategorySelected = object : AdapterView.OnItemSelectedListener {
        override fun onNothingSelected(parent: AdapterView<*>?) {
            //NOOP
        }

        override fun onItemSelected(parent: AdapterView<*>?, view: View?,
position: Int, id: Long) {
            selectedCategory.value =
categoriesLiveData.value?.get(position)
        }
    }

    fun refreshTransactions() {
        suspendCall {
            repository.refreshTransactions()
        }
    }

    val filteredData = MediatorLiveData<List<DBTransaction>>().apply {
        listOf(
            transactionsLiveData
        ).forEach { it ->
            addSource(it) {
                val filteredValue = transactionsLiveData.value
                value = filteredValue
            }
        }
    }

    private fun saveTransaction(dbTransaction: DBTransaction) {
        suspendCall {
            repository.saveTransaction(dbTransaction)
        }
    }
```

```
    @RequiresApi(Build.VERSION_CODES.O)
    fun onClick(view: View) {
        when (view.id) {
            R.id.fabAddTransaction -> {
                view.findNavController()

.navigate(R.id.action_transactionsFragment_to_addTransactionFragment)
            }
            R.id.btnSave -> {
                val formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy",
Locale.GERMAN)
                val ld = LocalDate.parse(dateLiveData.value.toString(),
formatter)
                Log.d("Date", ld.toString())
                saveTransaction(
                    DBTransaction(
                        contentsLiveData.value.toString(),
                        ld,
                        selectedCategory.value.toString(),
                        amountLiveData.value!!.toDouble()
                    )
                )
                view.findNavController()

.navigate(R.id.action_addTransactionFragment_to_transactionsFragment)
            }
            R.id.tvDate ->{
                suspendCall {
                    repository.getTransactionsAsyncSortedByDate()
                }
            }
            R.id.tvContents -> {
                suspendCall {
                    repository.getTransactionsAsyncSortedByContents()
                }
            }
            R.id.tvCategory -> {
                suspendCall {
                    repository.getTransactionsAsyncSortedByCategory()
                }
            }
            R.id.tvSpend -> {
                suspendCall {
                    repository.getTransactionsAsyncSortedBySpend()
                }
            }

        }
    }
}
```

Prilog 2. *fragment_add_transaction.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <data>
        <variable
            name="viewModel"
            type="com.example.budgetapp.viewmodels.TransactionsViewModel"
/>
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:padding="16dp"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:context=".ui.fragments.AddTransactionFragment">

        <com.google.android.material.textfield.TextInputLayout
            android:id="@+id/tilDate"
            style="@style/InputTil"
            app:hintEnabled="true"
            app:hintAnimationEnabled="true"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintTop_toTopOf="parent">

            <com.google.android.material.textfield.TextInputEditText
                android:id="@+id/etDate"
                style="@style/InputEditText"
                android:drawableStart="@drawable/ic_date_range_24"
                android:drawablePadding="16dp"
                android:focusable="false"
                android:hint="@string/date"
                android:text="@={viewModel.dateLiveData}" />

        </com.google.android.material.textfield.TextInputLayout>

        <com.google.android.material.textfield.TextInputLayout
            android:id="@+id/tilAmount"
            style="@style/InputTil"
            app:hintEnabled="true"
            app:hintAnimationEnabled="true"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/tilDate">

            <com.google.android.material.textfield.TextInputEditText
                android:id="@+id/etAmount"
                style="@style/InputEditText"
                android:hint ="@string/amount"
                android:text="@={viewModel.amountLiveData}"
                android:drawableStart="@drawable/ic_money_24"
                android:drawablePadding="@dimen/size16"
                android:inputType="numberDecimal"/>

        </com.google.android.material.textfield.TextInputLayout>
```

```
        <androidx.appcompat.widget.AppCompatImageView
            android:id="@+id/ivCategory"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="16dp"
            android:paddingStart="@dimen/size10"
            android:src="@drawable/ic_category_24"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/tilAmount" />

        <androidx.appcompat.widget.AppCompatSpinner
            android:id="@+id/spCategories"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:spinnerMode="dialog"
            android:theme="@style/SpinnerTheme"
            android:layout_marginStart="16dp"
            android:prompt="@string/category"
            items="@{viewModel.categoriesLiveData}"
            onItemSelected="@{viewModel.onCategorySelected}"
            app:layout_constraintTop_toTopOf="@+id/ivCategory"
            app:layout_constraintBottom_toBottomOf="@+id/ivCategory"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toEndOf="@+id/ivCategory" />

        <com.google.android.material.textfield.TextInputLayout
            android:id="@+id/tilContents"
            style="@style/InputTil"
            app:hintEnabled="true"
            app:hintAnimationEnabled="true"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintTop_toBottomOf="@+id/spCategories">

            <com.google.android.material.textfield.TextInputEditText
                android:id="@+id/etContents"
                style="@style/InputEditText"
                android:hint ="@string/contents"
                android:text="@={viewModel.contentsLiveData}"
                android:drawableStart="@drawable/ic_title_24"
                android:drawablePadding="@dimen/size16"
                android:inputType="textAutoComplete"/>

        </com.google.android.material.textfield.TextInputLayout>

        <Button
            android:id="@+id/btnSave"
            style="@style/blueButton"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            android:layout_margin="@dimen/size16"
            android:onClick="@{(v) -> viewModel.onClick(v)}"
            android:text="@string/save" />


    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Prilog 3. *StatsViewModel*

```
class StatsViewModel(private val repository: TransactionsRepository) :
BaseViewModel() {

    private val transactionsLiveData: LiveData<List<DBTransaction>?> get()
= repository.transactions

    val dateLiveData = MutableLiveData<String?>()

    private var pieData: PieData? = null
    private var pieDataSet: PieDataSet? = null
    private val pieEntries: MutableList<PieEntry> = arrayListOf()

    var lineEntries: MutableList<Entry> = arrayListOf()

    val categoriesLiveData = MutableLiveData<List<String>>(
        listOf(
            "Social life", "Food", "Self development", "Transportation",
"Health",
            "Beauty", "Education", "Household", "Other"
        )
    )

    val selectedCategory = MutableLiveData("")
    val onCategorySelected = object : AdapterView.OnItemSelectedListener {
        override fun onNothingSelected(parent: AdapterView<*>?) {
            //NOOP
        }
        override fun onItemSelected(parent: AdapterView<*>?, view: View?,
position: Int, id: Long) {
            selectedCategory.value =
categoriesLiveData.value?.get(position)
        }
    }

    fun initializePieChart(pieChart: PieChart) {
        Log.d("pieEntires", pieEntries.toString())
        pieDataSet = PieDataSet(pieEntries, "")
        pieData = PieData(pieDataSet)
        pieChart.legend.isEnabled = false
        pieChart.description.isEnabled = false
        pieChart.data = pieData
        pieDataSet!!.setColors(*ColorTemplate.JOYFUL_COLORS)
        pieDataSet!!.sliceSpace = 2f
        pieDataSet!!.valueTextColor = Color.WHITE
        pieDataSet!!.valueTextSize = 10f
        pieDataSet!!.sliceSpace = 5f
        // pieChart.setUsePercentValues(true)
        pieChart.animateXY(1500, 1500)
    }

    fun initializeLineChart(lineChart: LineChart, selectedCategory: String)
{
        val vl = LineDataSet(lineEntries, selectedCategory)
        vl.setDrawCircles(true)
        vl.lineWidth = 3f
```

```
        vl.circleRadius = 5f
        vl.color = Color.parseColor("#70009dd7")
        vl.setCircleColor(Color.parseColor("#009dd7"))
        vl.valueTextSize = 18f
        lineChart.xAxis.labelRotationAngle = 0f
        lineChart.xAxis.isEnabled = false
        lineChart.data = LineData(vl)
        lineChart.axisRight.isEnabled = false
        lineChart.setTouchEnabled(true)
        lineChart.setPinchZoom(true)
        lineChart.setDrawGridBackground(false)
        lineChart.description.text = "Order by date"
        lineChart.setNoDataText("No data yet!")
        lineChart.animateX(1500, Easing.EaseInExpo)
    }

    fun getStatsByDate(date: LocalDate) {

        val transactions = transactionsLiveData.value
        var pieEntriesTemp = pieEntries.toMutableList()

        transactions?.forEach { dbt ->
            if (dbt.date == date) {
                if (pieEntriesTemp.isEmpty()) {
                    pieEntries.add(PieEntry(dbt.totalPrice.toFloat(),
dbt.category.trim()))
                } else {
                    var categoryAlreadyExists = false
                    pieEntriesTemp.forEach { peTemp ->
                        if (peTemp.label.trim() == dbt.category.trim())
                            categoryAlreadyExists = true
                    }
                    if (!categoryAlreadyExists) {
                        pieEntries.add(PieEntry(dbt.totalPrice.toFloat(),
dbt.category.trim()))
                    } else {
                        var newValue = dbt.totalPrice.toFloat()
                        pieEntriesTemp.forEach { pe ->
                            if (pe.label.trim() == dbt.category.trim()) {
                                newValue += pe.value
                                pieEntries.remove(pe)
                                pieEntries.add(PieEntry(newValue,
dbt.category.trim()))
                            }
                        }
                    }
                }
                pieEntriesTemp = pieEntries.toMutableList()
            }
        }
    }

    fun getAmountByCategory(selectedCategory: String) {
        val transactions = transactionsLiveData.value
        var x = 1
        val lineEntriesTemp = ArrayList<Entry>()
        Log.d("Primljeno", selectedCategory)
        transactions?.sortedBy { it.date }?.forEach { dbt ->
            if (dbt.category.trim() == selectedCategory) {
                lineEntriesTemp.add(Entry(x.toFloat(),
dbt.totalPrice.toFloat()))
```

```
                x++
            }
        }
        lineEntries = lineEntriesTemp.toMutableList()
    }

    fun getAllStatsData() {
        val transactions = transactionsLiveData.value
        var pieEntriesTemp = pieEntries.toMutableList()

        transactions?.forEach { dbt ->
            if (pieEntriesTemp.isEmpty()) {
                pieEntries.add(PieEntry(dbt.totalPrice.toFloat(),
dbt.category.trim()))
            } else {
                var categoryAlreadyExists = false
                pieEntriesTemp.forEach { peTemp ->
                    if (peTemp.label.trim() == dbt.category.trim())
                        categoryAlreadyExists = true
                }
                if (!categoryAlreadyExists) {
                    pieEntries.add(PieEntry(dbt.totalPrice.toFloat(),
dbt.category.trim()))
                } else {
                    var newValue = dbt.totalPrice.toFloat()
                    pieEntriesTemp.forEach { pe ->
                        if (pe.label.trim() == dbt.category.trim()) {
                            newValue += pe.value
                            pieEntries.remove(pe)
                            pieEntries.add(PieEntry(newValue,
dbt.category.trim()))
                        }
                    }
                }
            }
            pieEntriesTemp = pieEntries.toMutableList()
        }
    }

    fun onClick(view: View) {
        when (view.id) {
            R.id.btnTotalStatsGraph -> {

view.findNavController().navigate(R.id.action_graphFragment_to_allStatsFrag
ment)
            }
            R.id.btnStatsByDate -> {
                view.findNavController()

.navigate(R.id.action_graphFragment_to_statsByDateFragment)
            }
            R.id.btnStatsBySpecificCategory -> {
                view.findNavController()

.navigate(R.id.action_graphFragment_to_statsByCategoryFragment)
            }

        }
    }
}
```