SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

**Marijan Hranj**

Izgradnja Web aplikacije pomoću Spring Boot-a i Kotlina

DIPLOMSKI rad

Varaždin, 2020.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Marijan Hranj

Matični broj: 46330/17–I

Studij: Informacijsko i programsko inženjerstvo

Izgradnja Web aplikacije pomoću Spring Boot-a i Kotlina

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Danijel Radošević

Varaždin, kolovoz 2020.

*Marijan Hranj*

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

U ovom diplomskom radu obraditi će se upotreba razvojnog okvira Spring Boot i programskog jezika Kotlin pri izgradnji Web aplikacije. Obradit će se specifičnosti jezika Kotlin te usporediti njegove karakteristike s programskim jezikom Java. Izraditi će se Web aplikacija za planiranje te rezerviranja odmora na raznim turističkim destinacijama.

Ključne riječi: SpringBoot; Java; Kotlin; Web; Aplikacija

**Sadržaj**

[1. Uvod 1](#_Toc47273440)

[2. Spring 2](#_Toc47273441)

[2.1. Povijest 3](#_Toc47273442)

[2.2. Programski okvir Spring 4](#_Toc47273443)

[2.2.1. Injektiranje ovisnosti i inverzija kontrole 4](#_Toc47273444)

[2.2.1.1. Spring implementacija injektiranja ovisnosti 5](#_Toc47273445)

[2.2.2. Arhitektura programskog okvira Spring 6](#_Toc47273446)

[2.2.2.1. Core Container 7](#_Toc47273447)

[2.2.2.2. AOP, Aspects i Instrumentation 7](#_Toc47273448)

[2.2.2.3. Messaging 7](#_Toc47273449)

[2.2.2.4. Data Access/Integration 8](#_Toc47273450)

[2.2.2.5. Web 8](#_Toc47273451)

[2.2.2.6. Test 9](#_Toc47273452)

[2.3. Spring Boot 9](#_Toc47273453)

[3. Kotlin 10](#_Toc47273454)

[3.1. Jezik 11](#_Toc47273455)

[3.1.1. Osnovna sintaksa 11](#_Toc47273456)

[3.1.1.1. Rad s paketima 11](#_Toc47273457)

[3.1.1.2. Funkcije 12](#_Toc47273458)

[3.1.1.3. Varijable 12](#_Toc47273459)

[3.1.1.4. String predlošci 13](#_Toc47273460)

[3.1.1.5. Uvjetni izrazi 13](#_Toc47273461)

[3.1.1.6. Provjeravanje null vrijednosti 14](#_Toc47273462)

[3.1.1.7. Provjeravanje te pretvorba tipova podataka 15](#_Toc47273463)

[3.1.1.8. For i While petlje te rasponi 15](#_Toc47273464)

[3.1.1.9. When izraz 16](#_Toc47273465)

[3.1.1.10. Kreiranje objekata 17](#_Toc47273466)

[3.2. Usporedba s programskim jezikom Java 17](#_Toc47273467)

[3.2.1. Java problemi koji su riješeni u Kotlinu 17](#_Toc47273468)

[3.2.1.1. Null reference 18](#_Toc47273469)

[3.2.1.2. Nepostojanje sirovih tipova podataka 18](#_Toc47273470)

[3.2.1.3. Nizovi su nepromjenjivi 19](#_Toc47273471)

[3.2.1.4. Pravi funkcijski tipovi 19](#_Toc47273472)

[3.2.1.5. Nekorištenje Wildcard izraza prilikom specificiranja tipa 19](#_Toc47273473)

[3.2.1.6. Nepodržavanje označenih iznimki 20](#_Toc47273474)

[3.2.2. Koncepti koje Java posjeduje a Kotlin ne 20](#_Toc47273475)

[3.2.2.1. Označene iznimke 20](#_Toc47273476)

[3.2.2.2. Primitivni tipovi podataka 20](#_Toc47273477)

[3.2.2.3. Statični članovi 21](#_Toc47273478)

[3.2.2.4. Korištenje Wildcard izraza prilikom specificiranja tipa 21](#_Toc47273479)

[3.2.2.5. Ternarni operator 21](#_Toc47273480)

[3.2.3. Koncepti koje Kotlin posjeduje a Java ne 22](#_Toc47273481)

[3.2.3.1. Inline funkcije 22](#_Toc47273482)

[3.2.3.2. Proširujuće funkcije 23](#_Toc47273483)

[3.2.3.3. Null sigurnost 24](#_Toc47273484)

[3.2.3.4. Pametna pretvaranja 24](#_Toc47273485)

[3.2.3.5. String predlošci 25](#_Toc47273486)

[3.2.3.6. Polja klasa 25](#_Toc47273487)

[3.2.3.7. Primarni konstruktor 26](#_Toc47273488)

[3.2.3.8. Delegacija 27](#_Toc47273489)

[3.2.3.9. Automatsko prepoznavanje tipova podataka 27](#_Toc47273490)

[3.2.3.10. Singletoni 28](#_Toc47273491)

[3.2.3.11. Nekorištenje Wildcard izraza prilikom specificiranja tipa 28](#_Toc47273492)

[3.2.3.12. Rasponi 28](#_Toc47273493)

[3.2.3.13. Preopterećivanje operatora 28](#_Toc47273494)

[3.2.3.14. Companion objekti 29](#_Toc47273495)

[3.2.3.15. Data klase 30](#_Toc47273496)

[3.2.3.16. Posebna sučelja za promjenjive i nepromjenjive objekte 30](#_Toc47273497)

1. Uvod

U današnjem brzorastućem IT (skraćeno od „Informacijska tehnologija“) svijetu svako malo pojavljuju se novi trendovi a samim time i prateće tehnologije, alati, programski jezici i slično. Zbog tog razloga teško je opstati u IT svijetu ukoliko ne dolazi do konstantnog unaprijeđenja i učenja. Spring se može pohvaliti da je već duži niz godina široko korištena tehnologija mnogih IT stručnjaka. Dokaz toga dobio sam i osobno uvidom u trenutnu potražnju u IT svijetu, kod mnoštva Java stručnjaka traži se upravo znanje Spring-a. U ovom radu fokus će najviše biti na Spring Boot-u kao jednom projektu nastalom na temelju programskog okvira Spring.

Popularnošću stvari može se pohvaliti i programski jezik Kotlin. Programski jezik Kotlin spada u mlade programske jezike, njegova prva stabilna verzija objavljena je ne tako davne 2016-te godine, točnije 15. 02. 2016 (Breslav, 2015). Tada ga je Breslav opisao kao pragmatični programski jezik koji kombinira objektno orijentirani te funkcionalni pristup. Kotlin se mogao koristiti svugdje gdje i programski jezik Java, tj. na svakom sklopovlju na kojem se može pokrenuti Java Virtualna Mašina (eng. *Java Virtual Machine - JVM*). Tako se Kotlin mogao koristiti u web aplikacijama, stolnim aplikacijama (eng. *Desktop app*) te mobilnim aplikacijama. Pod korištenjem Kotlina u razvoju mobilnih aplikacija podrazumijeva se razvoj za operativni sustav Android. Prema smjernicama od strane (Google, 2020), za razvoj Andorid aplikacija preporučuju se Java ili Kotlin. To se može vidjeti u njihovim uputama za kreiranje novog projekta za razvoj Android aplikacije gdje su predložili da se za korišteni programski jezik koristi ili Java ili Kotlin. Na temelju vlastitog iskustva mogu reći da primjećujem sve veću potražnju za Android developerima koji znaju Kotlin pa dolazim do zaključka kako su tvrtke počele shvaćati koliko je Kotlin dobar programski jezik.

Sve navedeno daje naslutiti kako su i Spring, tj. Spring Boot i Kotlin vrlo atraktivna rješenja koja su i te kako aktualna u današnjoj IT industriji. U ovom radu pokušat ću dati odgovore na pitanje: „Zašto su upravo te dvije tehnologije trenutno među popularnijima?“.

1. Spring

Prema službenoj Spring dokumentaciji (Pivotal, 2020b) navodi se kako Spring olakšava razvoj poslovnih aplikacija. Jezici koje Spring podržava su: Java, Kotlin te Groovy. Java je primarni izbor a Kotlin i Groovy služe kao alternative. U ovom radu glavna riječ će biti upravo o navedenoj alternativi, Kotlinu. Često sam se susretao sa krivom percepcijom koja vlada među ljudima a to je da je Spring samo jedan u nizu od tehnologija koje olakšavaju razvoj web aplikacija. Ta tvrdnja je samo djelomično točna. Naime, Spring se može koristiti za ravoj bilo kakve vrste aplikacije koja se može pokrenuti na JVM to znači od stolnih (eng. *desktop)* pa sve do web aplikacija. Dakle ruši se kriva pretpostavka kako Spring služi samo za razvoj web aplikacija.

Prema službenoj Spring dokumentaciji (Pivotal, 2020b) navodi se da je pod pojmom Spring, ovisno o kontekstu, moguće pomisliti na dva različita pojma, na sami programski okvir Spring (eng. *Spring framework)* te na obitelj projekata nastalih na temelju programskog okvira Spring. Kada ljudi govore o Springu, najčešće pomisle upravo na obitelj projekata nastalih na temelju programskog okvira Spring. Pošto će u ovom radu najviše biti riječ o Spring Bootu te Spring Frameworku, ostale projekte iz obitelji ću samo nabrojati (VMware, 2020a):

|  |  |
| --- | --- |
| * Spring Framework * Spring Boot * Spring Data * Spring Cloud * Spring Cloud Data Flow * Spring Security * Spring Session * Spring Integration * Spring HATEOS * Spring REST Docs * Spring Batch * Spring AMQP | * Spring za Android * Spring CredHub * Spring Flo * Spring za Apache Kafku * Spring LDAP * Spring Mobile * Spring Roo * Spring Shell * Spring Statemachine * Spring Vault * Spring Web Flow * Spring Web Services |

Kao što se može vidjeti, projekata je zaista mnogo te je pokriveno veliko područje potreba odnosno zahtjeva s istima.

* 1. Povijest

Prema službenoj Spring dokumentaciji (Pivotal, 2020b) spominje se da Spring prvi puta nastupa na tržište 2003. godine. Iako mnogi smatraju kako je Spring nastao kao konkurent Javi izdanju za poduzeća (eng. *Java Enterprise Edition – JAVA EE)*, stvarnost je zapravo takva da Spring nadopunjuje JAVA EE. Spring integrira neke od JAVA EE specifikacija na svoj jedinstven način, te specifikacije su sljedeće:

* Aplikacijsko programsko sučelje za Servlet (eng. *Servlet Application Programming Interface – Servlet API*)
* API Web utičnice (eng. *WebSocket API*)
* Uslužni programi za istodobno izvršavanje procesa (eng. *Concurrency Utilities*)
* API za povezivanje na JavaScript objekt notaciju (eng. *JavaScript Object Notation Binding API – JSON Binding API*)
* Validacija zrna (eng. *Bean validation*)
* Java API za postojanost (eng. *Java Persistance API - JPA*)
* Java servis za poruke (eng. *Java Message Service - JMS*)

Također po potrebi integriraju se i Java Arhitektura konektora (eng. Java Connector Architecture - JCA) te Java Transakcijski API (eng. Java Transaction API – JTA).

Od verzije 5.0, programski okvir Spring zahtjeva minimalno verziju 7 JAVA EE. Prema (VMware, 2020c), trenutna verzija programskog okvira Spring je 5.2.8.

A close up of a logo

Description automatically generated

Slika 1. Spring logo (izvor: (VMware, 2020d))

* 1. Programski okvir Spring

Programski okvir Spring (eng. *Spring framework*) je prvi projekt u obitelji projekata Spring. Od navedenog projekta nastali su ranije spomenuti projekti. Na službenoj dokumentaciji programskog okvira Spring (Pivotal, 2016b), navodi se kako je programski okvir Spring rješenje koje ne zahtjeva mnogo računalnih resursa i primjereno je za izgradnju poslovnih aplikacija. Programski okvir spring je modularan, tj. u prijevodu, sastoji se od više dijelova ili modula. Kasnije ću obraditi koji su to moduli. Zbog modularne arhitekture, moguće je koristiit samo one module, tj. dijelove programskog okvira Springa koji su potrebni, samim time dodatno se smanjuje opterećenost računalnih resursa. Na primjer, programer može odabrati da će koristiti modul potreban za razvoj *Web* aplikacija dok mu modul za komunikacijom sa bazom podataka neće biti potreban.

* + 1. Injektiranje ovisnosti i inverzija kontrole

Pojmovi koji se vrlo često vežu sa programskim okvirom Spring su „Injektiranje ovisnosti“ (eng. *Dependency injection - DI*) te „Inverzija kontrole“ (eng. *Inversion of control - IoC*). Programski ovkir spring je programski okvir koji koristi paradigmu inverzije kontrole. Pod pojmom inverzija kontrole, u širokom smislu, podrazumijeva se da tok izvršavanja programa preuzima sami programski okvir, a ne programer odnosno aplikacija. Navedeno omogućuje da programski okvir samostalno poziva implementacije koje je programer razvio. Dakle ukoliko programer želi prilagoditi ponašanje programskog okvira, on treba proširiti klase programskog okvira ili dodati svoje klase (Crusoveanu, 2020).

Drugi pojam, odnosno injektiranje ovisnosti predstavlja uzorak dizajna kojim se programski okvir Spring služi za postizanjem ranije opisane inverzije kontrole. Slijedi objašnjenje ovisnosti u programskom kodu te na koji se način uzorak dizajna injektiranje ovisnosti odnosi s istim (Crusoveanu, 2020).

public class Racunalo {

private Procesor procesor;

public Racunalo() {

procesor = new ProcesorImpl();

}

}

public class Racunalo {

private Procesor procesor;

public Racunalo(Procesor procesor) {

this.procesor = procesor;

}

}

U prvom primjeru klasa *Racunalo* „ovisna“ je o klasi *Procesor* te će se uvijek koristiti jedno te ista implementacija. U drugom primjeru koristio se princip injektiranja ovisnosti, te se na taj način postiglo da klasa *Racunalo* više nije ovisna o klasi *Procesor* pošto programer ima mogućnosti proslijediti klasi *Racunalo* bilo koju implementaciju koju on želi. Na taj način daje se veća sloboda samom programeru te postiže se da ovisnosti nisu usko vezane.

* + - 1. Spring implementacija injektiranja ovisnosti

Programski okvir Spring implementira uzorak dizajna injektiranje ovisnosti na način da koristi tzv. Spring IoC kontejner (eng. Spring *IoC Container*). Sa Spring IoC kontejnerom programer ima mogućnost označiti klase koje želi da se učitaju u IoC kontejner te ih kasnije samo „injektirati“ u svoj programski kod. Slijedi primjer Spring injektiranja ovisnosti:

public interface Servis { ... }

@Service

public class ServisImpl implements Servis { ... }

@Service

public class KlasaKojaKoristiServis {

@Autowired

private Servis servis;

...

}

U primjeru se može primjetiti ranije navedeno. Dakle programer je samo trebao označiti klasu koju želi učitati u IoC kontejner, u ovom slučaju je to bila klasa *ServisImpl* te ju istu mogao „injektirati“ u klasu *KlasaKojaKoristiServis* koja je također učitana u IoC kontejner. Kako bi se klasa učitala u IoC kontejner, mogu se koristiti neke od anotacija, u ovom primjeru koristila se anotacija *@Service*. U kodu aplikacije pokazati ću da se isto može postići korištenjem anotacije *@Repository* te *@Controller*. Razlika između tih anotacija je da se koriste za različite slojeve aplikacije. Zajedničko im je to da sve sadrže *@Component* anotaciju koja služi upravo za učitavanje u Spring IoC container. Dakle klase koje su učitane u IoC kontejneru, mogu injektirati druge klase iz kontejnera.

* + 1. Arhitektura programskog okvira Spring

Kao što je već bilo spomenuto, programski okvir Spring ima višedjelnu, odnosno modularnu arhitekturu.

A close up of a green screen

Description automatically generated

Slika 2. Arhitektura programskog okvira Spring

(izvor: (Pivotal, 2016a))

Iz slike se može vidjeti kako programski okvir Spring sadrži relativno velik broj modula od kojih su neki svrstani u tzv. slojeve (eng. *layers*): jezgreni kontejner (eng. *core container*), pristup podacima/integracija (eng. *data access/integration*) te web (Pivotal, 2016b).

* + - 1. Core Container

Prema službenoj dokumentaciji (Pivotal, 2016b) core kontejner predstavlja sloj koji se sastoji od sljedećih modula:

* *spring-core*
* *spring-beans*
* *spring-context*
* *spring-context-support*
* *spring-expression*

Spring-core te spring-beans moduli predstavljaju osnovu programskog okvira koji pružaju temeljne funkcionalnosti poput inverzije kontrole te injektiranje ovisnosti.

Najvažniji dio spring-context modula predstavlja sučelje ApplicationContext. Pomoću navedenog sučelja programer može pristupiti klasama učitanim u IoC kontejner.

Spring-context-support modul pruža mogućnost integriranja stranih biblioteka u Spring za upravljanje mailovima (eng. mailling), međuspremanjem (eng. caching), planiranjem (eng. scheduling) te predlošcima (eng. templates).

Spring-expression modul pruža jedan moćni jezik izraza (eng. expression language - EL) koji predstavlja proširenje jedinstvenog jezika izraza (eng. unified EL) opisanog u JSP 2.1 specifikaciji (Pivotal, 2016b).

* + - 1. AOP, Aspects i Instrumentation

Prema službenoj dokumentaciji (Pivotal, 2016b) navodi se kako se pomoću *spring-aop* modula mogu koristiti značajke aspektno orijentiranog programiranja (eng. *Aspect oriented programming – AOP*). Također postoji odvojeni modul *spring-aspects koji* pruža mogućnost integracije sa *AspectJ*.

*Spring-instrument* modul sadrži konfiguracije potrebne da se aplikacija koristi na određenim aplikacijskim serverima.

* + - 1. Messaging

*Spring-messaging* modul predstavlja modul koji pruža temelje za razvoj aplikacija koje su temeljene na porukama (eng. *message based application*) (Pivotal, 2016b).

* + - 1. Data Access/Integration

Prema službenoj dokumentaciji (Pivotal, 2016b) sloj pristupa podacima odnosno integracije sastoji se od sljedećih modula:

* *spring-jdbc*
* *spring-tx*
* *spring-orm*
* *spring-oxm*
* *spring-jms*

*Spring-jdbc* modul pruža mogućnost korištenja sučelja koje programer može koristiti za olakšano pisanje koda za java konekciju sa bazom podataka (eng. *Java Database Conectivity - JDBC*).

Uloga *spring-tx* modula je pružanje mogućnosti korištenja transakcija u programskom unutar aplikacije.

*Spring-orm* modul daje programeru na mogućnost korištenje objektno-relacijskog mapiranja (eng. *Object-relational mapping – ORM*). Sličnu stvar pruža i modul *spring-oxm*, no za razliku od *spring-orm* modula, *spring-oxm* modul daje mogućnost mapiranja objektnog-XML mapiranja.

*Spring-jms* modul pruža metode za kreiranje te konzumiranje poruka. Od verzije programskog ovkira Spring 4.1, navedeni modul se također može integrirati sa *spring-messaging* modulom (Pivotal, 2016b).

* + - 1. Web

Prema službenoj dokumentaciji (Pivotal, 2016b) web predstavlja sloj koji se sastoji od sljedećih modula:

* *spring-web*
* *spring-webmvc*
* *spring-websocket*
* *spring-webmvc-portlet*

*Spring-web* modul pruža temelj da aplikacija sadrži osnovne web funkcionalnosti poput HTTP (skraćeno od eng. *HyperText Transfer Protocol*) klijenta i slično (Pivotal, 2016b).

*Spring-webmvc* modul sadrži Springovu implementaciju uzorka Model-Pogled-Kontroler (eng. *Model-View-Controller - MVC*) te web servisa temeljenih na reprezentacijskom prijenosu stanja (eng. *Representational state transfer - REST*). Posljedica MVC-a je odvajanje poslovne logike, pogleda i podataka. Modul je također poznat pod nazivom *Web-Servlet* modul jer se temelji na Java Servletima (Pivotal, 2016b).

*Spring-webmvc-portlet* modul pruža istu funkcionalnost poput *spring-webmvc* modula samo što je u ovom slučaju u fokusu Java Portleti a ne Servleti kao kod *spring-webmvc* (Pivotal, 2016b).

* + - 1. Test

*Spring-test* modul daje na mogućnost pisanje jediničnih (eng. *unit*) te integracijskih testova (eng. *integration*) za testiranje Spring komponenti (Pivotal, 2016b).

* 1. Spring Boot

Spring Boot predstavlja jedan od projekata iz obitelji projekata nastalih na temelju programskog okvira Spring. U ovom radu biti će pokazano kako na temelju upravo toga projekta izgraditi jednu funkcionalnu web aplikaciju. Na službenoj stranici Spring Boot projekta (VMware, 2020b) navodi se kako Spring Boot olakšava razvoj samostalnih aplikacija nastalih na temelju Springa koje su zrele za produkciju. Programer može početi razvijati prema svojim potrebama vrlo brzo. Naime, za razliku od nekih ostalih programskih okvira, Spring Boot dolazi sa nekim već unaprijed učitanima konfiguracijama kako bi što više olakšali posao programeru. Također se navodi kako većinu Spring Boot aplikacija treba dodati tek minimalno konfigurarati sa Spring opcijama.

Neke od funkcionalnosti koje se na službenoj stranici spominju su:

* Kreiranje samostalih Spring aplikacija
* Uklapanje (eng. *Embed*) servera poput: Tomcat, Jetty ili Undertow kako bi se Spring Boot aplikacija direktno instalirala, nema potrebe za manualnom instalaciju .war datoteke nastale prilikom izgradnje (eng. *build*) Spring Boot aplikacije
* Opcija za korištenjem tzv. Početnih (eng. *Starter*) ovisnosti (eng. *Dependency*) koje pružaju širok spektar funckionalnosti
* Automatska konfiguracija Springa te također i nekih ostalih biblioteka (eng. *Library*) stranih proizvođača kada god je to moguće
* Pružanje produkcijskih funckionalnosti poput metrika, provjera zdravlja sustava i slično
* Za razliku od Springa gdje se konfiguracije pišu u XML (eng. *Extensible Markup Language*) formatu, u Spring Bootu se konfiguracije pišu u programskom kodu, što je po mom mišljenju puno bolje rješenje

Trenutna verzija Spring Boota je 2.3.1. Upravo tu verziju koristiti ću i u ovome radu, tj. aplikaciji. Na službenoj Spring Boot dokumentaciji (Pivotal, 2020a) se navodi kako navedna verzija zahtjeva Java verziju 8 kao minimum te je kompatibilno i sa Javom 14. Također uvjet je i verzija programskog okvira Spring 5.2.7 ili veća. U mojoj aplikacija upravo ta verzija programskog okvira Spring je korištena. Također se navodi kako se alati poput Apache Maven-a (verzija minimalno 3.3) ili Gradle (verzija minimalno 6.3) podržavaju izgradnju Spring Boot aplikacija. Kasnije će biti više riječ o Apache Mavenu.

1. Kotlin

U uvodu sam već napomenio kako je Kotlin relativno mlad programski jezik koji stječe sve veću popularnost. Na Kotlinovoj službenoj stranici (JetBrains s.r.o., 2020t) navodi se kako je Kotlin trenutno prikladan za sljedeće:

* Razvoj programskog koda koji može biti pokrenut na više mobilnih platformi, dakle iOS i Android

A close up of electronics

Description automatically generated

Slika 2. Prikaz korištenja Kotlina više mobilinih platformi

(izvor: slika zaslona preuzeta sa:

(JetBrains s.r.o., 2020w))

* Razvoj Kotlin programskog koda koji se pretvara u izvorni kod u binarnom obliku kako bi se izbjegla potreba za korištenjem dodatnog okruženja. Dakle dodaje se mogućnost pokretanja koda napisanog u Kotlinu na više platformi, neke od tih platformi su (JetBrains s.r.o., 2020l) iOS, macOS, watchOS, tvOS, Android, Windows, Linux, WebAssembly
* Korištenje Kotlina u području znanosti o podacima (eng. *Data Science*)
* Korištenje Kotlina za razvoj na serverskoj strani, upravo ovu funckionalnost Kotlina koristiti ću u svome radu, upariti ću Kotlin sa Spring Bootom na serverskoj strani
* Korištenje Kotlina za razvoj na Webu, tj. mogućnost pretvorbe Kotlin koda u JavaScript kod te programiranje na klijentskoj strani (JetBrains s.r.o., 2020j)
* Možda i najpoznatija funkcionalnost, korištenje Kotlina za razvoj Android aplikacija
  1. Jezik

Mogu primjetiti da se jezik kao takav ne razlikuje mnogo od programskoj jezika Java. Štoviše Kotlin čak podržava pozivanje Java biblioteka unutar Kotlin koda. Tvorac Kotlin-a, tvrtka JetBrains, u svom je razvojnom okruženju naziva Intellij čak dodao mogućnost pretvorbe Java datoteke u Kotlin datoteku. U ovom radu upravo će Intellij biti korišteno razvojno okruženje. Iako programski jezik Java znam od prije te opcija pretvorbe Java koda u Kotlin zvuči vrlo zanimljivo, nisam ju koristio pošto sam htio samostalno pisati Kotlin kod te naučiti sintaksu istoga. Kasnije u radu ću opisati sličnosti i razlike sa programskim jezikom Java.

* + 1. Osnovna sintaksa

Slijedi prikaz sintakse jezika te nekih osnovnih elemenata prema službenoj Kotlin dokumentaciji (JetBrains s.r.o., 2020o)

* + - 1. Rad s paketima

Definicija paketa je identična onoj u javi. Dakle koristi se ključna riječ ***package*** te naziv paketa. Na primjer: package hranj.marijan.diplomskirad

Ukoliko programer želi koristiti značajke nekih drugih paketa, mora koristiti ključnu riječ ***import*** te naziv paketa i naziv klase ili funkcije koju želi koristiti. Na primjer:

import org.springframework.boot.runApplication

znači da programer može koristiti funckiju *runApplication* iz navedenog paketa.

* + - 1. Funkcije

Početni dio svake Kotlin aplikacije je tzv. *main* funkcija. Kako bi označili da je odabrana funkcija *main*, ona mora imati naziv ***main***, slijedi primjer takve funkcije:

fun **main**() {

println("Ja sam main funkcija!")

}

Kao što se već može zaključiti, ključna riječ za definiranje funkcije je ***fun***. Nakon ključne riječi slijede opcionalni parametri te opcionalni povratni tip funkcije.

Na sljedećem primjeru objasniti ću navedene koncepte. Ukoliko se promotri funkcija koja množi dva broja:

**fun** pomnozi(mnozenik: **Int**, mnozitelj: **Int**): **Int** {

return mnozenik \* mnozitelj

}

Za razliku od main funkcije koja nema parametara niti povratnog tipa, funckija pomnozi prima dva parametra. Spomenuti parametri su: *mnozenik* te *mnozitelj*. Oba parametra su podatkovnog tipa *Int* što u Kotlinu predstavlja cijeli broj. Također kao povratni tip funkcije je zadan *Int*, što upravo i predstavlja umnožak dva cijela broja. Kako bi funkcija nešto „vratila“, koristi se ključna riječ ***return*** a zatim izraz koji treba biti „vraćen“.

* + - 1. Varijable

Prethodno spomenuti koncepti su, možemo reći, vrlo slični Javi. Prva veća promjena koju sam uočio u odnosu na Javu je deklaracija odnosno inicijalizacija varijabli. Naime ovdje do izražaja dolaze dvije ključne riječi: ***val*** i ***var***. Ukoliko se ispred deklaracije odnosno inicijalizacije varijable stavi ključna riječ ***val***, daje se na znanje Kotlinu kako se vrijednost te varijable više neće mjenjati. Prilikom označavanja polja (eng. *field*) klase sa ***val*** generira se zadana (eng. *default)* *„getter“* funkcija za navedeno polje.

Za razliku od ***val***, ključna riječ ***var*** dopušta kasniju promjenu vrijednosti varijable. Prilikom označavanja polja sa ***var***, generiraju se zadane i „*getter“* i „*setter“* funkcije za navedeno polje.

**val** a: Int = 1 // polje tipa Int, getter generiran

val b = 2 // Kotlin prepoznaje tip bez potrebnog navođenja

b = 3 // greška u kodu, val varijabla se ne može mjenjati

**var** a = 2 //polje tipa Int, generirani getter i setter

a = 3 // nema greške u kodu

* + - 1. String predlošci

Vrlo korisna stvar koja postoji u Kotlinu naziva se String predložak (eng. *String template*). Naime korištenjem navedenog koncepta, moguće je izbjeći po meni vrlo nepregledno spajanje Stringova sa operatorom ***+***. String u Kotlinu predstavlji identični tip podataka kao u Javi, dakle skup znakova. Slijedi primjer korištenja String predložaka:

val danas = LocalDate.now()

println("Danas je datum: **$**danas") // korištenje String predložaka

println("Danas je datum:" + danas) // korištenje operatora "+"

println("Sutra je datum: **${**danas.plusDays(1)**}**") // izraz

Kao što se iz navedenih primjera može vidjeti, za korištenje String predložaka koristi se operator **$** te potom ime varijable. Također je moguće navesti neki izraz umjesto varijable, no u tom slučaju izraz je potrebno staviti unutar vitičastih zagrada. Prava korist ove funckionalnosti ističe se ponajviše onda kada treba sastaviti dugačak String sa mnoštvom vrijednosti koje se čitaju iz varijabli ili izraza.

* + - 1. Uvjetni izrazi

Poput velike većine programskih jezika i Kotlin podržava grananje u programskom kodu. Uvjetni blok, tj. kondicionalni blok ili kako se često u žargonu spominje „*if uvjeti*“ u Kotlinu izgledaju identično kao u programskom jeziku Java uz neke sitne dodatne mogućnosti. Dodatna funkcionalnost je ta što *if*, odnosno uvjetniblok koda može „vratiti“ određenu vrijednost.

**if** (a > b) {

println("Varijabla a je veća od varijable b")

} **else if** (b > a) {

println("Varijabla b je veća od varijable a")

} **else** {

println("Varijable imaju jednaku vrijednost")

} // prikaz klasičnog korištenja *if* uvjeta

val najmanjiBroj = if (a < b) a else b

U zadnjem primjeru vidi se kako za razliku od programskog jezika Java *if* blok može iskazati kao izraz koji može „vratiti“ određenu vrijednost. U konkretnom primjeru ta vrijednost se pridružuje varijabli *najmanjiBroj* i to na način da će varijabla poprimiti vrijednost varijable *a* ukoliko je uvjet ispunjen a u suprotnom vrijednost varijable *b*.

* + - 1. Provjeravanje null vrijednosti

Kao i u programskom jeziku Java, tako i u Kotlinu varijabla može biti ne-inicijalizirana, tj. imati vrijednost ***null*** ili referencirati ***null***. Kada varijabla referencira ***null***, to znači da ista ne pokazuje niti na jednu adresu u memorijskom prostoru u kojem bi bila spremljena neka odgovarajuća vrijednost. U programskom jeziku Kotlin, programer mora eksplicitno naglasiti da određena vrijednost, tj. vrijednost varijable ili funkcije može poprimiti vrijednost ***null***. U tu svrhu koristi se operator **?**. Slijedi primjer korištenja istoga (JetBrains s.r.o., 2020m) :

var neNullVarijabla: String = "Nisam null!"

neNullVarijable = null // greška u kodu

var nullVarijabla: **String?** = "Mogu biti null!"

nullVarijabla = null // regularno

Kada bi programer htio pozvati neku od funkcija varijable imena *neNullVarijabla*, to nebi predstavljao nikakav problem, međutim ukoliko bi htio učiniti isto sa varijablom imena *nullVarijabla* morao bi prije toga provjeriti da vrijednost *nullVarijabla* ne iznosi ***null***. Ukoliko nebi učinio isto, kompajler (eng. *Compiler*) bi javljao grešku. Nekoliko je načina kako izbjeći pozivanje operacija nad ***null*** vrijednosti i samim time izbjegavanje iznimku null pokazivača (eng. *NullPointerException – NPE*) (JetBrains s.r.o., 2020m):

var nullVarijabla: String? = "Mogu biti null!"

var brojZnakova = if (nullVarijabla != null) nullVarijabla.length

else -1 // korištenje if – else kao izraz

-----------------------------------------------------------------

if (nullVarijabla != null) {

brojZnakova = nullVarijabla.length

} else {

brojZnakova = -1

} // klasično korištenje if - else

-----------------------------------------------------------------

println(**nullVarijabla?**.length) // tzv. siguran poziv (eng. *safe*

*call*), ukoliko je vrijednost varijable null, vraća se null*, u*

*protivnom odabrana vrijednost*

-----------------------------------------------------------------

brojZnakova = **nullVarijabla?**.length **?:** -1 //identično prvom

primjeru, umjesto if – else, koristi se tzv. Elvis operator ?:

Ukoliko je programer siguran da vrijednost varijable koja može poprimiti ***null*** vrijednost nije ***null***, može koristiti **!!** operator. Operator pretvara vrijednost u *ne-null* vrijednost ukoliko je moguće, u suprotnom baca *NPE*.

var nullVarijabla: String? = "Mogu biti null!"

val a = **nullVarijabla!!**.length // regularno

nullVarijabla: String? = null

val a = **nullVarijabla!!**.length // NPE

* + - 1. Provjeravanje te pretvorba tipova podataka

Za razliku od programskog jezika Jave, gdje sve klase nasljeđuju baznu klasu *Object*, u Kotlinu sve klase nasljeđuju *Any* klasu (JetBrains s.r.o., 2020a). Kako bi programer provjerio da je neki objekt tipa *Any* instanca neke druge klase, može koristiti ključnu riječ ***is***. Ukoliko je programer siguran da vrijednost spremljena u objektu tipa klase roditelja predstavlja objekt tipa klase djeteta, može koristiti ključnu riječ ***as*** kako bi pretvorio objekt tipa klase roditelj u objekt tipa klase djeteta (JetBrains s.r.o., 2020p). Ukoliko pretvorba nije moguća dogoditi će se iznimka.

fun vratiDuljinuRijeci(objekt: Any): Int {

if (objekt **is** String) {

return objekt.length // unutar ovog if bloka objekt

automatski pretvoren u tip podataka String

}

return -1 // ovdje je objekt tipa Any

}

-----------------------------------------------------------------

val objekt: Any = "Neki string"

val rijec = objekt **as** String // rijec je tipa podataka String

* + - 1. For i While petlje te rasponi

*For* i *While* petlje su vrlo slične programskom jeziku Java. Jedina razlika koju sam mogao uočiti je da klasična *for* petlja sa iteratorom više nije podržana. Umjesto toga može se koristiti nešto što iz Kotlina nazivaju raspon (eng. *range*). Kako bi se pristupalo elementu kolekcije, koristi se ključna riječ ***in***. Slijedi primjer *for* petlje:

**for** (elem **in** listOf("jedan", "dva", "tri"))

println(elem) // ispis: jedan, dva, tri

**for** ((indeks, elem) **in** listOf("jedan", "dva", "tri").withIndex()) {

println("Element: $elem sa indeksom: $indeks")

}

Ukoliko je programeru potreban indeks unutar for petlje, može pozvati funkciju *withIndex()* nad kolekcijom (JetBrains s.r.o., 2020c). Na taj način programer ima dostupno i indeks elementa te samu vrijednost elementa. *ListOf()* je funkcija koja je dostupna standardno u Kotlinu (JetBrains s.r.o., 2020u). Navedena funkcija kreira listu od navedenih elemenata. Slijedi primjer klasične Java *for* petlje te kako bi se isto kod napisao u Kotlinu koristeći raspone. Za definiranje raspona kombiniraju se ključna riječ ***in*** te operator ***..*** (dvije točke).

for (int i = 0; i < 10; i = i + 2){

System.out.println(i) // java ispis: 0, 2, 4, 6, 8

}

-----------------------------------------------------------------

for (i **in** 0..10 step 2) {

println(i) // Kotlin ekvivalent gornjeg Java koda

}

-----------------------------------------------------------------

val i = 1

if (i **!in** 2..i + 6) {

println("Varijabla nije u rasponu brojeva")

} // izraz sa in operatorom može također vratiti i logičku

vrijednost

Slijedi primjer *while* petlje koja kao što sam već spomenuo vrlo nalikuje *while* petlji u programskom jeziku Java:

val elementi = listOf("jedan", "dva", "tri")

val i = 0

**while** (i < elementi.size){

println("Element: ${elementi[i]} sa indeksom: $i")

} // ekvivalent gornje *for* petlje sa funkcijom *withIndex()*

* + - 1. When izraz

Nešto što je po mome mišljenju vrlo koristan koncept u Kotlinu naziva se *when* izraz. *When* izraz nalikuje na switch izraz u programskome jeziku Java no mnogo je moćniji. *When* izraz se defnira korištenjem ključne riječi ***when.*** Prikazati ću moć ovog koncepta na primjeru sa službene Kotlin dokumentacije (JetBrains s.r.o., 2020o):

fun opisiParametar(obj: Any): String =

**when**(obj){

1 -> "Jedan"

"Bok" -> "Pozdrav"

is Long -> "Long!!!"

!is String -> "Nije String!"

**else** -> "Ne znam opisati parametar."

}

println(opisiParametar("Bok")) // ispis: Pozdrav

println(opisiParametar(2L)) // ispis: Long!!!

println(opisiParametar("Nesto") // ispis: Ne znam opisati parametar

* + - 1. Kreiranje objekata

Objekti se u Kotlinu kreiraju na vrlo jednostavan način. Naime, jedino što je potrebno je specifirati naziv klase te u zagrade staviti parametre konstruktora ukoliko postoje. Za razliku od programskog jezika Jave gdje se za inicijalizaciju novog objekta mora koristiti ključna riječ ***new***, u Kotlinu to nije potrebno. Slijedi primjer inicijalizacije dva objekta različitog tipa:

val covjek = Covjek() // konstruktor nema parametara

val krug = Krug(2) // konstruktor ima parametar radijus tipa Int

* 1. Usporedba s programskim jezikom Java

Osobno smatram da programski jezik Kotlin i programski jezik Java, kojeg već od prije znam, dijele mnoge sličnosti. Te sličnosti mogle su se vidjeti u prethodnom odjeljku gdje se govorilo o sintaksi programskog jezika Kotlin. Naravno moraju postojati i stvari koje su različite u tim programskim jezicima. U ovom odjeljku upravo te stvari ću proći, tj. navesti ću što to programski jezik Kotlin ima a programski jezik Java ne i vice-versa. Na službenoj dokumentaciji Kotlina može se upravo naći odjeljak koji uspoređuje ta dva programska jezika. Svoju interpretaciju temeljiti ću upravo na tome (JetBrains s.r.o., 2020s).

* + 1. Java problemi koji su riješeni u Kotlinu

Prema službenoj dokumentaciji programskog jezika Kotlin (JetBrains s.r.o., 2020s) navodi se kako je nekoliko problema prisutnih u programskom jeziku Java riješeno u programskom jeziku Kotlin. Slijedi pregled nekih od tih problema.

* + - 1. Null reference

Kao što je već ranije spomenuto, vrijednost *null* u Kotlinu mogu poprimiti samo one varijable koje se za to unaprijed određene. U Kotlinovom slučaju, to su one varijable označene sa **?**. Ukoliko programer želi pristupiti vrijednosti varijable koja referencira *null*, mora prvo napraviti *null* provjere. Ukoliko to ne napravi kompajler će javljati grešku. Upravo to drastično smanjuje mogućnosti događanja NPE, što je jedan od ciljeva programskog jezika Kotlin. Dakle NPE se u Kotlinu rješavaju pomoću tipova podataka (JetBrains s.r.o., 2020m). To u programskom jeziku Java nije slučaj, ondje svaka varijabla može poprimiti vrijednost *null* te kompajler neće javljati da može doći do NPE. NPE se u Kotlinu može dogoditi korištenjem **!!** operatora čija se uloga ranije opisala. Na Kotlinovoj dokumentaciji navedeni operator opisuju da je za ljubitelje NPE (JetBrains s.r.o., 2020m).

* + - 1. Nepostojanje sirovih tipova podataka

U programskom jeziku Java raspoznaje se pojam sirovi tip podataka (eng. *raw type*). Prema službenoj Oracle stranici Jave u odjeljku za sirove tipove podataka (Oracle, 2020b), navodi se kako sirovi tip podataka predstavlja onu generičku klasu koja nema pridružen argument tipa. Objasniti ću to na primjeru generičkog sučelja *java.util.List*

public interface List<E> extends Collection<E>

Kao što se vidi iz priloženog sučelje *List* predstavlja generičko sučelje koje ima argument tipa *E*. Ukoliko bi programer inicijalizirao objekt tipa *List* bez navođenja argumenta tipa, u tom slučaju radilo bi se i sirovom tipu podataka. Problem sirovih tipova podataka je taj da se elementi koje generička klasa ili sučelje obgrljuju uvijek moraju pretvarati (eng. *castati*). U programskom jeziku Kotlin ne postoje sirovi tipovi podataka.

List lista1 = new ArrayList(); // sirovi tip podataka - nije

pridružen argument tipa, uzima se zadani - *Object*

List<String> list2 = new ArrayList<>(); // nije sirovi tip

Podataka – pridružen argument tipa *String*

lista1.add("String1");

lista2.add("String2");

String vrijednost = "";

vrijednost = lista1.get(0); // greška, nemoguće pretvoriti tip

podataka *Object* u tip *String*

vrijednost = (String) lista1.get(0); // nema greške

vrijednost = lista2.get(0); // nema greške

* + - 1. Nizovi su nepromjenjivi

U programskom jeziku Java moguće je pridružiti vrijednost niza (eng. *array*) tipa klase dijeteta nizu tipa klase roditelja. Uz neopreznost navedeno može dovesti do pogrješaka tijekom izvršavanja programskog koda. Nizovi Kotlinu su nepromjenjivi. Dakle ne može se učiniti ovo što sam upravo naveo da se može u programskom jeziku Java (JetBrains s.r.o., 2020r). Slijedi primjer:

String[] javaStringNiz = {"elementNiza"};

Object[] javaObjektNiz = javaStringNiz; // regularno

-----------------------------------------------------------------

val kotlinStringNiz = arrayOf("elementNiza")

val kotlinAnyNiz: Array<Any> = kotlinStringNiz // greška

* + - 1. Pravi funkcijski tipovi

Programski jezik Java u svojoj verziji 8 uveo je pojam funkcijskih sučelja (eng. *Functional Interface*) kojima su se htjeli dotaknuti područja funkcijskog programiranja. Funkcijsko sučelje je svako Java sučelje koje sadrži točno jednu apstratknu metodu, dakle metodu koju treba implementirati. Za razliku od Kotlina, gdje su funkcije na istoj razini poput varijabli, to u Javi nije slučaj. Da su funkcije na istoj razini u strukturi kao varijable znači da se funkcija u Kotlinu izravno može proslijediti kao argument drugoj funkciji više razine, spremiti u drugu varijablu, vratiti iz neke druge funkcije više razine i slično (Zhang, 2019). U Javi se slično ponašanje može također postići, no ondje se sve funkcije prvo moraju pretvoriti u odgovarajuće funkcijsko sučelje. Dakle zaključak je da u Kotlinu postoje pravi funkcijski tipovi dok se u Javi funkcije najprije moraju pretvoriti u odgovarajuće funkcijsko sučelje.

* + - 1. Nekorištenje Wildcard izraza prilikom specificiranja tipa

Ukoliko programer u programskom jeziku Java želi koristiti generički tip podataka koji kao argument tipa može poprimiti više izraza, može koristiti tzv. *Wildcard* izraze. Istu stvar može postići u programskom jeziku Kotlin samo uz korištenje ključnih riječi ***in*** i ***out,*** dakle nešto jednostavnije (JetBrains s.r.o., 2020h). Opet ću na primjeru generičkog sučelja *java.util.List* (Java) te *kotlin.collections.List* (Kotlin)objasniti ovaj primjer.

List<String> lista = new ArrayList<>(); // samo Stringovi mogu biti

pohranjeni

val lista: List<String> = arrayListOf() // Kotlin ekvivalent

List<? extends String> lista = new ArrayList<>(); // mogu biti

pohranjeni i Stringovi i klase koje nasljeđuju String klasu

val lista: List<out String> = arrayListOf() // Kotlin ekvivalent

List<? super String> lista = new ArrayList<>(); // mogu biti

pohranjeni i Strinovi i klase koje String klasa nasljeđuje

val niz: Array<in String> = arrayOf() // Kotlin ekvivalent sa nizom

Izrazi oblika *"? extends String"* i *"? super String"* predstavljaju *Wildcard* izraze.

* + - 1. Nepodržavanje označenih iznimki

Označene iznimke (eng. *checked exceptions*) su one iznimke koje se moraju obraditi u programskom kodu, ukoliko se ne obrade kompajler će javljati grešku. Kotlin za razliku od Jave ne podržava označene iznimke. U Kotlin dokumentaciji referenciraju se na nekoliko izvora u kojima se navodi kako označene iznimke zapravo loša praksa (JetBrains s.r.o., 2020v).

Pošto postoji mogućnost pozivanja Kotlin koda iz nekih drugih programskih jezika koji podržavaju označene iznimke poput Jave, moguće je obavijestiti programera koji poziva Kotlin kod kako je moguće da pozivajuća funkcija može baciti iznimku. To se radi tako da se Kotlin funkcija anotira sa sa *@Throws* anotacijom te specificira klasa iznimke (JetBrains s.r.o., 2020v).

* + 1. Koncepti koje Java posjeduje a Kotlin ne

Postoje koncepti koji su podržani unutar programskog jezika Java, međutim nisu u programskom jeziku Kotlin. Po mome mišljenju, izostajanje tih funkcionalnosti nije toliko bitno jer su nekako već nadoknađene na određeni način. Slijedi popis takvih funkcionalnosti.

* + - 1. Označene iznimke

Kao što je bilo spomenuto u prethodnom odjeljku, programski jezik Java za razliku od programskog jezika Kotlin podržava tzv. označene iznimke.

* + - 1. Primitivni tipovi podataka

Programski jezik Java podržava primitivne tipove podataka koji nisu klase. Za razliku od Jave, programski jezik Kotlin kao takav ne podržava primitivne tipove podataka na kakve su programeri navikli u Javi. Jednostavne vrijednosti poput brojeva, slova i slično spremaju se u Kotlinu u odgovarajuću obgrljenu klasu (eng. *wrapper class*). Java primitivni tipovi su: *byte*, *short*, *int*, *long*, *float*, *double*, *boolean* i *char* (Oracle, 2020a). Kotlin klase koje služe poput primitivnih tipova podataka u Javi su: *Byte*, *Short*, *Int*, *Long*, *Float*, *Double*, *Boolean* i *Char* (JetBrains s.r.o., 2020q).

* + - 1. Statični članovi

Programski jezik Java podržava tzv. statične članove (metode, varijable i slično). To su oni članovi koji pripadaju klasi a ne instanci klase, tj. objektu. Dakle takav član se može pozivati direktno iz klase, bez potrebe za instanciranjem objekta. Programski jezik Kotlin ne podržava statične članove, no zato postoje neki drugi koncepti koji ih mogu zamjeniti. Jedan od tih koncepata naziva se *companion* *object*. Za korištenje navedenog koncepta, koriste se ključne riječi ***companion*** i ***object***. Ključna riječ ***object*** označava kako se radi o *Singleton (uzorak dizajna)* objektu. Ključna riječ ***companion*** omogućuje da se članovi *companion object-a* pozivaju direktno preko klase (JetBrains s.r.o., 2020e). Dakle postiže se isti učinak kao sa Java statičnim članovima. Slijedi primjer:

class Klasa {

**companion** **object** Objekt {

fun zbroji(x: Int, y: Int) :Int {

return x + y

}

}

}

val zbroj = **Klasa.zbroji**(1, 2)

* + - 1. Korištenje Wildcard izraza prilikom specificiranja tipa

Kao što je već ranije bilo spomenuto, Java podržava specificiranje tipova podataka koristeći *Wildcard* izraze. Kako bi postigao sličnu funkcionalnost, programski jezik Kotlin služi se ključnim riječima ***out*** i ***in*** kako je ranije opisano.

* + - 1. Ternarni operator

Programski jezik Java podržava koncept koji se naziva ternarni operator. Iz samog naziva se da zaključiti da operator treba imati nešto u vezi s brojem tri. Uistinu to je i slučaj, naime ovako izgleda korištenje ternarnog operatora:

logickiIzraz ? vrijednostJedan : vrijednostDva

Ukoliko vrijednost logičkog izraza istinita vratiti će se *vrijednostJedan* a u suprotnom *vrijednostDva*. Kao alternativa u programskom jeziku Kotlin, mogu se koristi *if* izrazi o kojima je ranije u radu bilo riječ.

int veciBroj = a > b ? a : b; // Java ternarni operator

val veciBroj = if (a > b) a else b // Kotlin alternativa

* + 1. Koncepti koje Kotlin posjeduje a Java ne

Kako postoje stvari koje su podržane u programskom jeziku Java a nisu programskom jeziku Kotlin, tako je za očekivati i suprotno, tj. da postoje neki koncepti koje Kotlin posjeduje a Java ne. Po mome mišljenju upravo te stvari koje razlikuju Kotlin od Jave ostavljaju vrlo moderan utisak i moguće je da bi bile presudne za programere koji se dvoume između korištenja navednih programskih jezika. Slijedi popis navedenih koncepata.

* + - 1. Inline funkcije

Linijske (eng. inline) funkcije predstavljaju prvi koncept koji programski Jezik Kotlin podržava u odnosu na programski jezik Java. Pozivanje funkcije iz funkcije više razine u nekim slučajevima može donjeti neke posljedice. Naime, svaki put kada funkcija više razine poziva funkciju niže razine, odnosno funkciju prosljeđenu kao parametar, dolazi do kreiranja novog objekta iste te pozivajuće funkcije. Kao što se može zaključiti, u tom slučaju potrebno je alocirati dodatni memorijski prostor što nije uvijek povoljno (JetBrains s.r.o., 2020i). Slijedi pojednostavljeni primjer temeljen na verziji sa (GeeksforGeeks, 2019):

fun funkcijaViseRazine( funkcijaManjeRazine :() -> Unit) {

funkcijaManjeRazine()

}

fun main(args: Array<String>) {

println("Tekst 1")

funkcijaViseRazine { println("Tekst 2") }

println("Tekst 3")

}

Funkciji pod nazivom *funkcijaViseRazine* kao argument je pridružen tzv. lambda izraz. Kao i Java kod, Kotlin kod se prvo prevodi u Java bajt-kod (eng. *byte-code*). Kada bi se gornji odsječak koda preveo u bajt-kod te zatim dekompajlirao, pojavio bi se kod sličan sljedećem (iz koda su izostavljene nebitne stvari) (GeeksforGeeks, 2019):

public void funkcijaViseRazine(Function funkcijaManjeRazine) {

funkcijaManjeRazine.invoke()

}

Linija koda: *funkcijaManjeRazine.invoke();* bi zatim u pozadini izvršavala sljedeće:

funkcijaManjeRazine(new Function() {

@Override

public void invoke() {

System.out.println("Tekst 2");

}

});

Kao što se može primjetiti, od prosljeđenog lambda izraza kreirala bi se nova instanca Function objekta. Ukoliko bi se pozivalo mnoštvo funkcija na taj način došlo bi do primjetne potrošnje memorije i slabljenja performansi (GeeksforGeeks, 2019).

Kako bi Kotlin riješio ovaj problem, uveden je pojam linijskih odnosno *inline* funkcija. Inline funkcije sprječavaju kreiranje novih objekata prosljeđene funkcije, već samo pozivaju kod navedene funkcije na mjestu gdje se ista poziva. Sve što je potrebno napraviti kako bi se funkcija deklarirala kao inline jest ispred definicije funkcije staviti ključnu riječ ***inline***. Slijedi primjer kako bi ranije navedeni primjer izgledao sa inline funkcijom (GeeksforGeeks, 2019).

**inline** fun funkcijaViseRazine( funkcijaManjeRazine :() -> Unit) {

funkcijaManjeRazine()

}

Kada bi se u ovom slučaju kod preveo u bajt-kod te dekompajlirao, izgledao bi otprilike ovako:

public void funkcijaViseRazine(Function funkcijaManjeRazine) {

System.out.println("Tekst 2");

}

Kao što se može primjetiti, korištenjem inline funkcija programski kod *funkcijaManjeRazine* je kopiran na mjesto gdje se ista pozivala.

* + - 1. Proširujuće funkcije

Jedan po meni vrlo zanimljiv koncept predstavljaju funkcije proširenja odnosno ekstenzije (eng. *extension*). Naime, Kotlin omogućuje proširenje funkcionalnosti klase bez potrebe za pisanjem vlastitih klasa koje iste proširuju. U primjeru ću prikazati kako se može proširiti funkcionalnost klase *kotlin.collections.MutableList* dodavanje funkcije *zamjeni()* (JetBrains s.r.o., 2020b)*:*

fun MutableList<String>.zamjeni(index1: Int, index2: Int) {

val privremeno = this[index1] // 'this' referencira listu

this[index1] = this[index2]

this[index2] = privremeno

}

Dakle programer prije naziva funkcije mora definirati koju klasu želi proširiti, u ovom primjeru *kotlin.collections.MutableList<String>*. Sada programer ima na mogućnost korištenje funkcije zamjeni na svakom objektu tipa *kotlin.collections.MutableList<String>*.

val lista: MutableList<String> = mutableListOf("jedan", "dva", "tri")

println(lista[0]) // jedan

lista.zamjeni(0, 1)

println(lista[0]) // dva

* + - 1. Null sigurnost

Ovo poglavlje je već ranije bilo obrađeno u radu stoga ću samo ukratko ponoviti najbitnije. Kotlin za razliku od Jave raspoznaje tipove podataka koji mogu poprimiti vrijednost ***null*** od onih koji ne mogu. Zbog istoga kompajler upozorava programera, ukoliko se koristi takvim tipovima podataka, da provjeri referencira li objekt ***null*** vrijednost. U programskom jeziku Java svi ne-primitivni tipovi podataka mogu poprimiti vrijednost ***null*** te kompajler ne upozorava programera o mogućim NPE. Za ophodnju sa tipovima podataka koji mogu referencirati ***null*** u Kotlinu koriste se operatori ***?*** i ***!!*** (JetBrains s.r.o., 2020m).

* + - 1. Pametna pretvaranja

Pametna pretvaranje (eng. *smart casts*) su također jedan koncept koji sam već ranije napomenuo stoga ću samo ukratko ponoviti. U programskom jeziku Kotlin sve klase nasljeđuju baznu klasu *Any* (JetBrains s.r.o., 2020a)*.* Dakle objekt tipa *Any* može kao vrijednost poprimiti instancu bilo koje klase. Kako bi programer utvrdio radi li se o specifičnoj klasi, treba koristiti operator ***is***. Unutar bloka koda u kojem se pomoću operatora ***is*** utvrdi da objekt tipa *Any* ili bilo koja klasa roditelja zapravosadrži objekt dijeteta, dolazi do pametnog pretvaranja. Dakle u navedenom bloku kompajler zna da se radi o klasi djeteta. Ukoliko je programer siguran da se radi o specifičnoj klasi dijeteta sadržanoj u klasi roditelja, može koristiti ***as*** operator kako bi „ručno“ pretvorio objekt u odgovarajući tip (JetBrains s.r.o., 2020p). Ponovno ću navesti primjer od ranije radi lakšeg prisjećanja:

fun vratiDuljinuRijeci(objekt: Any): Int {

if (objekt **is** String) {

return objekt.length // unutar ovog if bloka objekt

automatski pretvoren u tip podataka String

}

return -1 // ovdje je objekt tipa Any

}

-----------------------------------------------------------------

val objekt: Any = "Neki string"

val rijec = objekt **as** String // rijec je tipa podataka String

* + - 1. String predlošci

Kotlin podržava koncept koji se naziva String predlošci. Pošto je o ovoj temi također bilo riječ ranije, samo ću ukratko ponoviti. Kako bi Kotlin riješio problem spajanja tipova podataka *String* pomoću operatora ***+***, koje je znalo izgledati dosta nepregledno, uveo je pojam String predložaka. Unutar *Stringa* pomoću operatora ***$*** mogu se pozivati varijable te izrazi (JetBrains s.r.o., 2020o). Slijedi primjer korištenja istih:

val danas = LocalDate.now()

println("Danas je datum: **$**danas") // korištenje String predložaka

println("Danas je datum:" + danas) // korištenje operatora "+"

println("Sutra je datum: **${**danas.plusDays(1)**}**") // izraz

* + - 1. Polja klasa

Polja (eng. *field*) u programskom jeziku Kotlin mogu biti ili promjenjiva (***var***) ili primjerena samo za čitanje (***val***). Ukoliko se polje označi sa ***val***, automatski se generira zadani „*getter“* za odgovarajuće polje. Međutim ako se polje označi sa ***var***, automatski se dodaju zadani i „*getter“* i „*setter“*. „*Getter“* i „*setter“* imaju isti naziv poput naziva polja. Programer ima mogućnost promjeniti ponašanje zadanih „*gettera*“ i „*settera*“ polja na sljedeći način, fokus je na polju *godina* (JetBrains s.r.o., 2020g):

class Osoba {

val ime: String = "Pero"

var prezime: String = "Peric"

var godina: Int = 1995

get() = field + 1

set(value) {

field = value + 1

}

}

val osoba = Osoba()

osoba.ime = "Ime" // greska

println(osoba.ime) // poziva se zadani getter

osoba.prezime = "Prezime" // poziva se zadani setter

osoba.godina = 2020 // poziva se prilagođeni setter

println(osoba.godina) // poziva se prilagođeni getter

* + - 1. Primarni konstruktor

Za razliku od programskog jezika Java, programski jezik Kotlin podržava koncept koji se naziva primarni konstruktor. Pomoću primarnog konstruktora moguće je već u samom zaglavlju klase navesti parametre koje klasa treba imati za instanciranje objekta. Takav primarni konstruktor zamjeniti će zadani konstruktor kojem za instanciranje objekta nisu potrebni parametri. Klasa može imati jedan primarni i više sekundarnih konstruktora. Sekundarni konstruktor zadaje se specificiranjem ključne riječi ***constructor***. Primarni konstruktor ne može sadržavati programski kod, zbog tog slučaja ukoliko je potrebno obaviti radnje prije instanciranja, koristi se *init* blok. Za specificiranje *init* bloka koristi se ključna riječ ***init*** (JetBrains s.r.o., 2020k). Slijedi primjer korištenja primarnog i sekundarnog konstruktora:

class Osoba**(ime: String, prezime: String)** {

**init** {

println("$ime $prezime")

}

**constructor**(ime: String) {

println(ime)

}

}

var osoba = Osoba("Ime", "Prezime")

osoba = Osoba("Ime")

Klasa iz navedenog primjera ima dva konstruktora, jedan primarni koji kao parametre prima *ime* i *prezime* i drugi sekundarni koji kao parametar prima *ime*.

* + - 1. Delegacija

Programski jezik Kotlin podržava sam po sebi korištenje uzorka dizajna „Delegacija“ bez potrebe za pisanjem dodatnog koda. Ključnu ulogu u postizanju navedenog ima ključna riječ ***by*** (JetBrains s.r.o., 2020f). Slijedi primjer korištenja iste kako bi se postigla delegacija:

interface Baza {

fun ispisiBroj()

}

class BazaImpl(val broj: Int) : Baza {

override fun ispisiBroj() {

println(broj)

}

}

class KlasaKojaKoristiBaznuKlasu(baza: Baza) : Baza **by** baza

val baza = BazaImpl(5)

KlasaKojaKoristiBaznuKlasu(baza).ispisiBroj() // 5

U navedenom primjeru može se primjetiti korištenje ključne riječi ***by***. Također se primjećuje delegacija, odnosno kako se funkcija *ispisiBroj()* klase *Baza* delegira u klasu *KlasaKojaKoristiBaznuKlasu* te je ista mogla pozvati implementaciju iz *BazaImpl*.

* + - 1. Automatsko prepoznavanje tipova podataka

Programski jezik Kotlin, za razliku od Java podržava automatsko prepoznavanje tipova podataka. Dakle izraz s lijeve strane će se automatski prepoznati, nije potrebno navođenje tipa podataka prilikom deklaracije (JetBrains s.r.o., 2020r). Slijedi primjer automatskog prepoznavanja tipova podataka:

val longBroj = 1.2L // tip podatka Long

val intBroj = 1 // tip podatka Int

val doubleBroj = 2.2 // tip podatka Double

val osoba = Osoba() // tip podatka Osoba

* + - 1. Singletoni

Već je bilo viđeno kako Kotlin olakšava korištenje uzoraka dizajna bez pisanja dodatnog koda u primjeru sa delegacijom. Slična stvar je i sa uzorkom dizajna *singleton*, naime Kotlin omogućava programeru da vrlo jednostavno ključnom riječi ***object*** specificira *singleton* objekt (JetBrains s.r.o., 2020e). Slijedi primjer korištenja istoga:

**object** MojSingleton {

...

}

Dakle umjesto specificiranje klase, koristila se ključna riječ ***object*** i time se postiglo da *MojSingleton* predstavlja uzorak dizajna *singleton*. Po meni vrlo elegantno i jednostavno rješenje.

* + - 1. Nekorištenje Wildcard izraza prilikom specificiranja tipa

O ovoj temi bilo je riječ u više navrata no pošto se u ovom odjeljku govori i stvarima koje programski jezik Kotlin podržava, a programski jezik Java ne, ponoviti ću. Programski jezik Kotlin umjesto korištenja Wildcard izraza za specificiranje tipa podataka koristi se ključnim riječima ***in*** i ***out*** (JetBrains s.r.o., 2020h).

* + - 1. Rasponi

O rasponima je bila riječ u poglavlju o osnovnoj sintaksi programskog jezika Kotlin. Slijedi kratki podsjetnik raspona u programskom jeziku Kotlin. Za korištenje raspona kombiniraju se ključna riječ ***in*** te operator ***..*** (dvije točke) (JetBrains s.r.o., 2020o).

for (i **in** 0..10 step 2) {

println(i) // 0, 2, 4, 6, 8

}

* + - 1. Preopterećivanje operatora

Preopterećivanje operatora (eng. *operator overloading*) je po meni vrlo zanimljiva funkcionalnost dostupna u programskom jeziku Kotlin. Naime, programski jezik Kotlin omogućuje samostalno implementiranje ponašanja bilo kojeg operatora kada se koristi na odgovarajućem tipu podataka. Primjera radi, moguće je definirati ponašanje operatora ***+*** kada se koristi sa tipom podataka *Osoba*. Svaki operator ima svoju odgovarajuću metodu koja se treba implementirati kako bi unutar klase nad kojom će se pozivati operator. Na primjer za operator ***+*** postoji funkcija *unaryPlus()* koja se treba implementirati u klasi *Osoba* ukoliko se navedeni operator želi koristiti uz navedenu klasu. Bitna stvar koju treba naglasiti je da deklaracija navedene funkcija kojom se implementira ponašanje operatora mora započinjati sa ključnom riječi ***operator*** (JetBrains s.r.o., 2020n). Slijedi primjer preopterećivanja operatora ***+*** nad klasom *Osoba*.

class Osoba {

var godina = 20

**operator** fun unaryPlus() {

this.godina++

}

}

val osoba = Osoba()

println(osoba.godina) // 20

+osoba

println(osoba.godina) // 21

* + - 1. Companion objekti

Kao što je već ranije bilo spomenuto, programski jezik Kotlin ne podržava statične metode poput programskog jezika Java. Međutim ista funkcionalnost, tj. pozivanje članova koji pripadaju klasi a ne instanci klase postiže se korištenjem *companion* objekata. Kako bi objekt definirali kao *companion* koristi se ključna riječ ***companion*** (JetBrains s.r.o., 2020e). Slijedi primjer korištenja istoga:

Class Klasa {

**companion** **object** Objekt {

fun zbroji(x: Int, y: Int) :Int {

return x + y

}

}

}

val zbroj = **Klasa.zbroji**(1, 2)

* + - 1. Data klase

Također jedan vrlo koristan koncept koji smanjuje potrebu za pisanje ponavljajućeg koda jesu Data klase. U programiranju stalno se susreće potreba za pisanjem klasa čija je jedina uloga da čuvaju podatke (eng. *data*). Programski jezik Kotlin olakšava pisanje takvih klasa i to na način da se iste mogu označiti ključnom riječi ***data***. Klasi kojoj se pridruži navedena ključna riječ, automatski se generiraju funkcije koje bi svaka klasa čija je jedina uloga čuvanje podataka trebala imati. Spomenute funkcije su:

* *equals()/hashcode()*
* *toString()*
* *component****N****()* funkcije – N odgovara rednom broju parametra u primarnom konstruktoru
* *copy()*

Navedene funkcije generiraju se na temelju parametara primarnog konstruktora navedene *data* klase. Primarni konstruktor sa barem jednim parametrom označenim sa ključnim riječima ***val*** ili ***var*** uvijet je *data* klasa (JetBrains s.r.o., 2020d). Slijedi primjer deklariranja *data* klase:

**data** class Osoba(val ime: String) {

**...**

}

* + - 1. Posebna sučelja za promjenjive i nepromjenjive kolekcije