Napredno Java programiranje

# Autori:

# Goran Savić

# Milan Segedinac

# Kreiranje Java anotacija

Ranije smo videli kako se mogu koristiti Java anotacije. Kao što smo objasnili, anotacije se koriste da pruže programeru ili programu dodatne informacije o klasi. Pogledajmo sada kako možemo definisati novu anotaciju. Anotacija se definiše kao interfejs čiji naziv počinje sa @. Kao metode interfejsa navode se atributi koje će anotacija sadržati. Ovo je sve što treba uraditi za definisanje anotacije. Dakle, ne postoji programski kod koji se implementira, jer anotacija sama po sebi nije programski entitet koji obavlja neku operaciju, već služi samo da dodatno opiše neku komponentu programa.

Kreiranje nove anotacije ćemo ilustrovati kroz primer generičkog sistema za pretragu nad proizvoljnim entitetima. Sistem treba da pretražuje entitete po različitim atributima. Uvešćemo novu anotaciju koja će definisati kriterijum pretrage entiteta. Postavljanjem anotacije nad entitetom, definisaćemo po kojim atributima se entitet pretražuje. Pogledajmo, najpre kako možemo definisati ovu anotaciju.

**@Retention(RetentionPolicy.RUNTIME)**

**@Target(ElementType.TYPE)**

**public @interface SearchBy {**

**enum SearchType {EQUALS, CONTAINS};**

**String[] fields();**

**SearchType searchType();**

**}**

Za prikazanu anotaciju @SearchBy je najpre putem @Target anotacije definisano da može biti primenjena nad tipovima podataka u programu (npr. nad klasom). U toku izvršavanja programa, uvidom u strukturu anotirane klase će biti dostupna informacija da je klasa označena ovom anotacijom, obzirom da je kao *RetentionPolicy* postavljena vrednost *RUNTIME*. Vidimo da je u anotaciji moguće specificirati dva atributa. Prvi atribut je niz stringova *fields*, koji predstavlja niz imena polja po kojima će anotirani entitet biti pretraživ. Drugi atribut anotacije je *searchType* koji označava kakva je strategija pretrage za anotirani entitet. Dve varijante pretrage postoje - da se tražena vrednost mora potpuno poklapati sa nekim od atributa entiteta (*SearchType.EQUALS)* ili samo delimično (*SearchType.CONTAINS*).

Pogledajmo sada kako neki entitet možemo anotirati gore kreiranom anotacijom.

**@SearchBy(fields ={"name", "population"}, searchType = SearchType.EQUALS)**

**public class Country {**

**private int id;**

**private String name;**

**private int population;**

**...**

**}**

Vidimo da je za prikazani entitet *Country,* definisano da treba da bude pretraživ po atributima *name* i *population* i da se pri pretrazi zahteva da tražena vrednost bude potpuno jednaka vrednosti u atributu.

# Refleksija

U toku izvršavanja programa, moguće je dobiti informacije o strukturi komponenata Java programa (klasa, interfejsa, atributa, metoda, ...). Na primer, da bismo implementirali generičku pretragu na bazi anotacije prikazane u prethodnom poglavlju, potrebno je da programskim putem preuzmemo informacije navedene u anotaciji. U anotaciji je navedeno po kojim atributima države treba da budu pretražive. Ovo podrazumeva programski uvid u strukturu klase *Country*. Takođe, kada utvrdimo po kojim atributima je država pretraživa, potrebno je na osnovu imena atributa pozvati odgovarajuću get metodu za preuzimanje vrednosti tog atributa. Obzirom da implementiramo sistem za pretragu generički, nije moguće statički u kodu predefinisati koje metode će biti pozvana. Umesto toga, na osnovu imena metode i uvida u strukturu klase, potrebno je dinamički pozvati tu metodu nad odgovarajućim objektom klase. Ovaj mehanizam se naziva **refleksija**i u Javi je podržan korišćenjem skupa klasa koje se nazivaju Java Reflection API.

Upoznaćemo se sa refleksijom u Javi kroz jedan jednostavan primer. Pogledajmo kako bismo za objekat klase *Country*, mogli preuzeti sve atribute koje klasa sadrži i pozvati *get* metodu nad objektom za svaki atribut.

**Country country = new Country(1, "Serbia", 7057000);**

**Field[] fields = Country.class.getDeclaredFields();**

**for (Field field: fields) {**

**Method fieldGetter = Country.class.getMethod("get" +**

**String.valueOf(field.getName().charAt(0)).toUpperCase()**

**+ field.getName().substring(1));**

**System.out.println(field.getName() + ": " + fieldGetter.invoke(country));**

**}**

Programski kod je najpre preuzeo niz svih atributa koje klasa *Country* sadrži. Primetimo da je metoda *getDeclaredFields* pozvana nad objektom koji je dobijen izrazom *Country.class*. Ovaj objekat predstavlja instancu klase *Class*. Objekti ove klase reprezentuju određenu Java klasu tako što imaju informacije o nazivu, paketu i strukturi te klase. Atributi koje metoda *getDeclaredFields* vraća su predstavljeni objektima klase *Field*. Na osnovu imena atributa se za svaki atribut preuzima objekat koji reprezentuje *get* metodu koja preuzima vrednost tog atributa. Metoda je predstavljena objektom klase *Method*. Za samo preuzimanje vrednosti, metoda je pozvana nad objektom *country* putem metode *invoke* objekta klase *Method*.

Na sličan način moguće je preuzimati i druge podatke o klasi. Npr. anotacije kojima je klasa anotirana su dostupne metodom *getAnnotation*.

# Anonimne klase

Kao što smo ranije videli, interfejsi predstavljaju moćan alat za korišćenje polimorfizma u Javi. Česta je praksa da programski kod bude kreiran tako da radi sa interfejsom, kako bi se mogao iskoristiti za bilo koji objekat koji taj objekat implementira. Tako na primer, podrška za sortiranje kolekcije u Javi je implementirana tako da klasa *Collections* kao parametar prima kolekciju bilo kojih objekata koji implementiraju interfejs *Comparable*.

Poznato je da pri izvršavanju programa nije moguće raditi sa interfejsima, nego samo sa instancama klase koja implementira interfejs. Iz tog razloga, neophodno je kreirati novu klasu koja nasleđuje određeni interfejs i u njoj implementirati metode interfejsa. Ovo može biti dosta zahtevan zadatak u situacijama kada kreirana klasa neće biti korišćena na drugim mestima u kodu.

U ovakvim situacijama, efikasna varijanta je korišćenje **anonimnih klasa**. Anonimne klase predstavljaju mehanizam Java jezika u kojem je moguće istovremeno deklarisati klasu i instancirati je. Za ovako kreiranu klasu će postojati taj jedan instancirani objekat i neće biti kasnije moguće iskoristiti kreiranu klasu za kreiranje novih objekata.

Pogledajmo korišćenje anonimnih klasa na primeru. Recimo da postoji interfejs *Displayable* dat u primeru. Klase koje implementiraju ovaj interfejs obavezuju se da mogu svoje podatke da prikažu korisniku.

**public interface Displayable {**

**void displayData(String prefix);**

**}**

Pogledajmo u nastavku kako možemo objekat koji implementira ovaj interfejs napraviti i korišćenjem anonimne klase.

**Displayable d = new Displayable() {**

**String text = "Novi Sad";**

**public void displayData(String prefix) {**

**System.out.println(prefix + " " + text);**

**}**

**};**

**d.displayData("Ispis je: ");**

Praktično, u kodu je napravljena nova klasa koja implementira interfejs *Displayable*. Klasa nema naziv (anonimna je), definiše atribut *text* i implementira metodu *displayData* preuzetu od interfejsa koji implementira. Klasa je odmah i instancirana i referenca na novokreirani objekat je uskladištena u promenljivu *d*. Na kraju listinga vidimo da se objekat *d* koristi kao i objekat bilo koje klase koja na klasičan način implementira interfejs.

Dakle, korišćenje anonimnih klasa štedi programeru vreme, jer ne zahteva kreiranje posebne klase koja implementira interfejs, već omogućuje da se instanciranje objekta i definisanje klase obave u jednom bloku koda. Naglašavamo još jednom da ovako kreirana klasa nije ponovno iskoristiva za kreiranje novih instanci.

# Lambda izrazi

Od Jave verzije 8 postoji i još kraća sintaksa za implementaciju interfejsa bez definisanja posebne klase koja interfejs implementira. Reč je o **lambda izrazima**. Ovi izrazi se mogu koristiti samo za implementaciju interfejsa koji imaju tačno jednu metodu. Ovakve interfejse nazivamo **funkcionalni interfejsi**.

Interfejs *Displayable* iz prethodnog primera je funkcionalni interfejs, obzirom da sadrži jednu metodu. Pogledajmo kako bi izgledala njegova implementacija korišćenjem lambda izraza.

**Displayable d = prefix -> { System.out.println(prefix + " Novi Sad"); };**

Izraz sa desne strane znaka jednakosti je lambda izraz. Njegova funkcionalnost je analogna anonimnoj klasi iz prethodnog primera. Lambda izraz ima dva glavna sintaksna dela, koji su razdvojeni znakom ->. Sa leve strane ovog znaka je niz parametara koje metoda interfejsa prima. Kao što vidimo, tip parametra je izostavljen i generalno se može izostaviti uvek kada kompajler iz konteksta može da zaključi koji je tip parametra. Takođe, nazivi parametara su proizvoljni. Kod sa desne strane je proizvoljna implementacija metode interfejsa, koja može da koristi prosleđeni parametar.

Sintaksa je vrlo slična i ako metoda interfejsa prima više parametara. Pogledajmo izmenjeni interfejs *Displayable* koji prima više od jednog parametra, i odgovarajući lambda izraz za implementaciju ovog interfejsa.

**public interface Displayable {**

**void displayData(String prefix, String sufix);**

**}**

**Displayable d = (px, sx) -> {System.out.println(px + " Novi Sad " + sx);};**

Vidimo da su parametri navedeni u zagradama i razdvojeni zarezom. Zagrade se mogu staviti i kada ima samo jedan parametar. Ako metoda interfejsa ne prima nijedan parametar, obavezno je staviti otvorenu i zatvorenu zagradu kao oznaku za parametre, što je prikazano u primeru.

**Displayable d = () -> { System.out.println("Novi Sad"); };**

Što se tiče tela lambda izraza, ako telo sadrži samo jedan izraz (kao u prethodnom primeru) tada vitičaste zagrade nisu obavezne. Ako metoda interfejsa nije void, telo može da ima return izraz. Ako je return izraz jedini izraz u telu lambda izraza, onda se reč return može i izostaviti. Pogledajmo ovo na primeru funkcionalnog interfejsa koji vraća vrednost.

**public interface Verificator {**

**boolean verify(int a);**

**}**

**Verificator v = number -> number == 0;**

**boolean verificationResult = v.verify(5);**

Vidimo da interfejs vrši verifikaciju prosleđenog celog broja. U konkretnoj implementaciji definisanoj lambda izrazom, verifikacija se vrši poređenjm broja sa nulom. Reč *return* je izostavljena i podrazumeva se da metoda vraća *boolean* vrednost nastalu evaluacijom izraza *number == 0*. Na kraju listinga vidimo kako definisana implementacija interfejsa može biti iskorišćena.

Sintaksno gledano lambda izraz predstavlja implementaciju interfejsa. Obzirom da je reč o funkcionalnom interfejsu koji ima tačno jednu apstraktnu metodu, lambda izraz koji predstavlja implementaciju te metode praktično reprezentuje jednu funkciju. Tako da korišćenje funkcionalnih interfejsa, odnosno lambda izraza omogućuje prosleđivanje funkcija drugim funkcijama (metodama, govoreći Java terminologijom). U korišćenju lambda izraza strogo se preporučuje da oni ne vrše izmene nad objektima sa kojima rade. Ako se kod zasniva na neizmenjivim objektima (eng. *immutable*), tada je podrška za konkurentan pristup jednostavnija, obzirom da ne može doći do štetnog preplitanja pri konkurentnim izmenama objekta. Ovakva programska paradigma koja se zasniva na izvršavanju programa kroz evaluaciju matematičkih funkcija i izbegava izmenjive podatke i zavisnost izvršavanja od stanja podataka, naziva se **funkcionalno programiranje**. Iako postoji od sredine 20. veka, korišćenje principa funkcionalnog programiranja u programiranju informacionih sistema je postalo posebno popularno u skorije vreme u skladu sa povećanim zahtevima za paralelizacijom koda.

# Java Stream API

U skladu sa promovisanjem principa funkcionalnog programiranja, standardna Java biblioteka od verzije Java 8 sadrži Java Stream API koji omogućuje organizovanje podataka u tok, pri čemu se nad podacima u toku mogu primenjivati različite funkcije korišćenjem lambda izraza.

Iako tok podataka može biti formiran od bilo koje sekvence podataka, najčešće se tok dobija iz kolekcije podataka, obzirom da od Java 8 verzije, kolekcija ima metodu *stream()* koja vraća tok podataka predstavljen klasom *Stream*. Pogledajmo primer ispisa svih država korišćenjem Stream API-ja, ako se države nalaze u listi *countries*  i ako objekat država ima metodu *displayCountry* koja ispisuje na konzolu podatke o državi.

**countries**

**.stream()**

**.forEach(c -> c.displayCountry());**

Metoda *forEach* će za svaki podatak izvršiti funkciju definisanu prosleđenim lambda izrazom. U konkretnom primeru to je ispis podataka o državi.

Korišćenjem tokova i lambda izraza moguće je jednostavno izvršiti filterisanje podataka u toku korišćenjem metode *filter*. Pogledajmo prethodni primer proširen tako da se ispisuju samo države koje imaju više od 10 miliona stanovnika.

**countries**

**.stream()**

**.filter(c -> c.getPopulation() > 10000000)**

**.forEach(c -> c.displayCountry());**

Metoda *filter* takođe vraća tok podataka, ali će se u ovom toku naći samo podaci koji zadovoljavaju kriterijum zadat prosleđenim lambda izrazom.

Podaci u toku mogu biti transformisani korišćenjem proizvoljne funkcije tako da dobijemo novi tok podataka. Pogledajmo na primeru kako tok država može biti transformisan u tok naziva država korišćenjem funkcije *map*.

**countries**

**.stream()**

**.filter(c -> c.getPopulation() > 10000000)**

**.map(c -> c.getName())**

**.forEach(name -> System.out.println(name));**

Vidimo da funkcija *map* sada vraća string koji predstavlja naziv države, što rezultuje tokom naziva država. Primetimo da *forEach* funkcija zato sada operiše nad string podacima, umesto nad državama.

Podaci formirani u toku se mogu opet okupiti u neki objekat, npr. u listu. Ovo se vrši operacijom *collect* koja je ilustrovana na primeru okupljanja imena država iz prethodnog primera u listu.

**List<String> names =**

**countries**

**.stream()**

**.filter(c -> c.getPopulation() > 10000000)**

**.map(c -> c.getName())**

**.collect(Collectors.toList());**

Osim klasičnih tokova kakvi su prikazani u prethodnim primerima, postoje i specijalizovani tokovi za rad sa primitivnim tipovima podataka *int*, *long* i *double*. Ovi tokovi su predstavljeni klasama *IntStream*, *LongStream* i *DoubleStream*. Ovi tokovi podržavaju dodatne operacije koje mogu biti primenjene nad brojevima, kao što su zbir podataka, izračunavanje prosečne, maksimalne i minimalne vrednosti. Pogledajmo primer računanja zbira populacije u svim državama.

**int sumPopulation =**

**countries**

**.stream()**

**.mapToInt(c -> c.getPopulation())**

**.sum();**

Vidimo da operacija *mapToInt* vraća tok celih brojeva dobijenih transformacijom koju sprovodi prosleđeni lambda izraz.

Poslednja operacija nad tokom podataka koja će ovde biti pokazana je operacija *flatMap*. Ova operacija za svaki objekat u toku, vraća novi *stream* podataka i najčešće se koristi da se iz određenog objekta preuzme kolekcija koju on sadrži kao 1:N asocijaciju. Pogledajmo kako možemo ispisati gradove u državama preuzimanjem toka gradova iz svake države.

**countries**

**.stream()**

**.flatMap(country -> country.getCities().stream())**

**.forEach(city -> System.out.println(city.getZipCode() + " " + city.getName()));**

Vidimo da operacija *flatMap* vraća za svaku državu tok napravljen od kolekcije gradova. Tako da se *forEach* operacija koja sledi primenjuje nad tim tokom gradova.

Kada smo se upoznali sa glavnim operacijama nad tokom podataka, možemo da izvršimo klasifikaciju ovih operacija u dve grupe. Prva grupa su međuoperacije, koje rezultuju tokom podataka. U ovu grupu spadaju operacije kao što su *filter* i *map*. Važno je naglasiti da se ove operacije ne izvršavaju odmah pri pozivu. Umesto toga, izvršavaju se tek pri pozivu neke od operacija iz druge grupe. Ovu drugu grupu operacija nazivamo terminalne operacije, jer rezultuju konačnom vrednošću. Tek one zapravo izvršavaju lanac operacija definisanih u toku. Primeri terminalnih operacija su *forEach* i *collect*.

Dakle, tek pozivom terminalne operacije tok podataka se procesira i to tako što svaki pojedinačan element toka vertikalno prolazi kroz sve operacije. Na primer, ako imamo lanac *filter*-*map*-*forEach*, za svaki element posebno će biti utvrđeno da li ispunjava uslov definisan u filter operaciji, pa će biti transformisan korišćenjem map operacije, pa će nad njim biti izvršen kod definisan u *forEach* operaciji.

Obzirom da se svaki element odvojeno procesira, posao obrade toka podataka se može paralelizovati na više jezgara procesora. Za paralelizaciju izvršavanja, potrebno je umesto klasičnog toka kreirati paralelni tok podataka putem metode *parallelStream().* Dat je primer paralelnog toka za ispis naziva država koje imaju više od 10 miliona stanovnika.

**countries**

**.parallelStream()**

**.filter(c -> c.getPopulation() > 10000000)**

**.map(c -> c.getName())**

**.forEach(name -> System.out.println(name));**

## Ugrađeni funkcionalni interfejsi

Videli smo da funkcije koje mogu biti primenjene nad tokom, kao što su *forEach, filter* ili *map*, kao parametar primaju lambda izraze. Kao što znamo, lambda izraz se može proslediti na mestu funkcionalnog interfejsa, što znači da su ove funkcije deklarisane tako da primaju funkcionalne interfejse.

Standardna Java biblioteka klasa u paketu *java.util.function* sadrži određen broj funkcionalnih interfejsa opšte namene, koje koriste metode iz Java Stream API-ja, ali koje i programeri mogu da koriste za svoje metode.

Na primer, često je potrebno napisati metodu koja kao parametar prima funkcionalni interfejs čija metoda dobija jedan parametar i vraća boolean vrednost. Ovaj funkcionalni interfejs već postoji u pomenutom paketu i zove se *Predicate.* Metoda *filter* u Stream API-ju očekuje implementaciju upravo ovog interfejsa. Slično, čest je slučaj korišćenja funkcionalnog interfejsa čija metoda očekuje jedan parametar i koja vraća vrednost proizvoljnog tipa. I ovakav funkcionalni interfejs već postoji i njegov naziv je *Function*. U tabeli su navedeni neki često korišćeni predefinisani funkcionalni interfejsi iz Java biblioteke klasa.

|  |  |
| --- | --- |
| Naziv | Opis |
| **Consumer** | Operacija koja prima parametar i ne vraća rezultat. |
| **Supplier** | Operacija koja ne prima parametre i vraća rezultat. |
| **Function** | Operacija koja prima parametar i vraća rezultat |
| **Predicate** | Operacija prima parametar i vraća boolean rezultat |
| **BiConsumer** | Operacija koja prima dva parametra i ne vraća rezultat |
| **BiFunction** | Operacija koja prima dva parametra i vraća rezultat |
| **BiPredicate** | Operacija koja prima dva parametra i vraća boolean rezultat |
| **UnaryOperator** | Operacija nad jednim parametrom koja vraća rezultat istog tipa kao parametar |
| **BinaryOperator** | Operacija nad dva parametra istog tipa koja vraća rezultat istog tipa kao parametri |

# Method References

Možemo primetiti da neki od lambda izraza iz prethodnih primera izvršavaju samo jednu naredbu i to poziv neke postojeće metode. Takav slučaj je sa izrazima

* **name -> System.out.println(name)**
* **c -> c.getName()**
* **c -> c.getPopulation()**
* **c -> c.displayCountry()**

U ovakvim slučajevima, moguće je referencirati metodu koja treba biti pozvana. Ova funkcionalnost je uvedena u Java verziji 8 i naziva se *Method References.* Pogledajmo izmenjenkod za ispis naziva država korišćenjem *method references*.

**countries**

**.parallelStream()**

**.filter(c -> c.getPopulation() > 10000000)**

**.map(Country::getName)**

**.forEach(System.out::println);**

Vidimo da se korišćenjem znaka dve tačke za referencu na metodu navodi klasa kojoj metoda pripada ili objekat nad kojim treba biti pozvana, kao i sam naziv metode.

Na sličan način je moguće koristiti referencu na konstruktor (eng. *constructor references*). Pogledajmo primer kreiranja toka podataka gde se na osnovu naziva države kreiraju objekti koji predstavljaju fudbalske reprezentacije.

**countries**

**.stream()**

**.map(Country::getName)**

**.map(FootballTeam::new)**

**.forEach(t -> System.out.println(t.getName()));**

Vidimo da su objekti klase *FootballTeam* kreirani referenciranjem konstruktora ove klase. Klasi se kao parametar prosleđuje podatak iz toka.

# Optional

Kao što smo ranije objasnili, podaci se u Java programu čuvaju u objektima koji su dostupni preko referenci koje ukazuju na njih. Referenca koja ne ukazuje ni na jedan objekat ima *null* vrednost. Nad takvom referencom nije moguće izvršavati operacije, tj. pokušaj izvršavanja operacije će rezultovati izuzetkom *NullPointerException*. Obzirom da to nije moguće proveriti u trenutku kompajliranja, pri izvršavanju programa mora se voditi računa o tome da li je u referenci uskladištena *null* vrednost.

Od Java verzije 8, postoji ugrađena podrška za efikasnije upravljanje referencama koje mogu imati *null* vrednost. Java 8 uvodi klasu Optional koja predstavlja omotač oko proizvoljnog objekta. Objekat klase *Optional* skladišti referencu na određeni objekat i pruža metode za pristup tom objektu, proveru da li je objekat ima vrednost *null*, kao i metode za rad sa objektom čija funkcionalnost zavisi od toga da li objekat ima vrednost *null* ili skladišti validnu referencu. Koncept *Optional* klase je ilustrovan na slici.

Optional<Country>

Optional<Country>

Country

Objekat Country je *null*

Sadrži objekat Country

Za proizvoljan objekat, možemo dobiti Optional objekat koji ga sadrži statičkom metodom *of* klase Optional, kao u narednom primeru.

**Country country = new Country(1, "Serbia", 7057000);**

**Optional<Country> c1 = Optional.of(country);**

U datom primeru, objekat *country* je klasičan objekat koji skladišti podatke o državi, dok promenljiva *c1* predstavlja objekat klase *Optional* koji je omotač oko objekta *country.*

Ukoliko je prosleđeni objekat *country* *null*, pojaviće se izuzetak. Slično, *Optional* objekat možemo dobiti i metodom Optional.ofNullable(*objekat*), s tim da ova metoda ne izaziva izuzetak ako objekat ima *null* vrednost.

Optional objekat koji ne skladišti nijednu vrednost može se instancirati na sledeći način.

**Optional<Country> c2 = Optional.empty();**

Za *Optional* objekat se može proveriti da li skladišti objekat ili null vrednost metodom *isPresent*. Metodom *orElse* se može dobiti uskladišteni objekat ili neka druga specificirana vrednost u zavisnosti od toga da li je objekat *null.* Ako želimo da preuzmemo vrednost uskladištenu u objektu Optional, to možemo uraditi metodom *get* uz napomenu da vraćena vrednost može biti *null*. Pogledajmo ove metode na primeru ranije kreiranih *Optional* objekata *c1* i *c2*.

**boolean exists1 = c1.isPresent();**

**boolean exists2 = c2.isPresent();**

**Country x1 = c1.orElse(new Country());**

**Country y1 = c1.get();**

**Country x2 = c2.orElse(new Country());**

**Country y2 = c2.get();**

Za prikazani primer, promenljiva *exists1* će imati vrednost true obzirom da *Optional* objekat c1 skladišti vrednost, dok će *exists2* biti false jer objekat *c2* skladišti *null* vrednost. Obzirom da c1 skladišti državu, promenljive x1 i y1 će dobiti referencu na ovu državu. Sa druge strane, u slučaju promenljive *c2* koja skladišti null*,* metoda *orElse* će vratiti novu praznu državu dobijenu pozivom konstruktora koji je prosleđen metodi. Što se tiče promenljive *y2*, ona će sadržati *null* vrednost jer je *Optional* objekat c2 skladišti null vrednost.

Ukoliko je potrebno izvršiti neku operaciju nad objektom, prethodno je potrebno proveriti da li je u objektu *null* vrednost. Korišćenjem *Optional* objekta ovo se može izvršiti pozivom metode *ifPresent* koja kao parametar dobija lambda izraz koji definiše funkciju koju treba izvršiti. Dat je primer ispisa naziva države, ako država nije null. Država je predstavljena ranije kreiranim *Optional* objektom *c1*.

**c1.ifPresent(c -> System.out.println(c.getName()));**

Dobra primena *Optional* objekata je u situacijama kada imamo veze asocijacije između objekata. Kao što znamo, svaka asocijacija je predstavljena referencom na objekat, koja posledično može imati i null vrednost. Pre pristupa asociranom objektu, uvek je neophodno proveriti da li asocijacija ima *null*  vrednost. Pogledajmo ovo na primeru tri klase međusobno povezane vezama asocijacije. Klase redom reprezentuju adresu, grad i državu, pri čemu adresa ima informaciju o gradu u kojem se nalazi, dok grad ima informaciju o državi u kojoj se nalazi. Data je slika ovog objektnog modela.

country

city

Country

City

Address

Ukoliko je potrebno preuzeti naziv države u kojoj se adresa nalazi, korišćenjem klasičnog pristupa kod bi izgledao kao u narednom primeru.

**String name;**

**if (address != null && address.getCity() != null && address.getCity().getCountry() != null &&**

**address.getCity().getCountry().getName() != null) {**

**name = address.getCity().getCountry().getName();**

**} else {**

**name = "Nepoznato";**

**}**

Ista funkcionalnost može biti realizovana korišćenjem *Optional* objekta na sledeći način.

**Optional<Address> a = Optional.of(address);**

**name = a.map(Address::getCity)**

**.map(City::getCountry)**

**.map(Country::getName)**

**.orElse("Nepoznato");**

Vidimo da je najpre dobijen *Optional* objekat koji skladišti adresu. Nad ovim objektom je pozivom metode *map* dobijen novi *Optional* objekat koji skladišti grad u kojem se adresa nalazi. Naredna *map* operacija je rezultovala *Optional* objektom koji skladišti državu u kojoj se grad nalazi. Poslednja *map* metoda je vratila *Optional* objekat koji skladišti naziv države. Obzirom da svaka od prethodno prosleđenih metoda (*getCity, getCountry, getName*) može da vrati i *null* vrednost, konačnu vrednost naziva države preuzimamo metodom *orElse*. Ova metoda će vratiti naziv države, ako je *Optional* objekat sadrži ili reč *Nepoznato* u suprotnom.