**Sadržaj:**

[1. poglavlje: Nove funkcionalnosti programskog jezika Java od verzije 1.7 3](#_Toc519433428)

[1.1 Unaprijeđena naredba *try* 4](#_Toc519433429)

[1.2 Upravljanje višestrukim *catch*-blokovima 4](#_Toc519433430)

[1.2.1 Odvojeni *catch*-blokovi 4](#_Toc519433431)

[1.2.2 Grupiranje *catch*-blokova 5](#_Toc519433432)

[1.3 Korištenje *stringova* u *switch*-izjavi 5](#_Toc519433433)

[1.4 Novi operatori u radu s generičkim tipovima 6](#_Toc519433434)

[2. poglavlje: Funkcionalnosti Javina programskog jezika od verzije 1.8 7](#_Toc519433435)

[2.1 Lambda-izrazi i funkcionalna sučelja 8](#_Toc519433436)

[2.1.1 Funkcionalna sučelja 8](#_Toc519433437)

[2.1.2 Referenciranje lambda-izrazima 9](#_Toc519433438)

[2.2 Java tokovi (engl. *streams*) 10](#_Toc519433443)

[2.2.1 Stream.filter() 10](#_Toc519433444)

[2.2.2 Stream.min() i Stream.max() 11](#_Toc519433445)

[2.2.3 Stream.count() 11](#_Toc519433446)

[2.2.4 Stream.map() 12](#_Toc519433447)

[2.2.5 Stream.average i Stream.sum() 12](#_Toc519433448)

[2.2.6 Stream.reduce() 13](#_Toc519433450)

[2.2.7 Stream.sorted() 14](#_Toc519433451)

[2.2.8 Stream.findFirst() i Stream.orElse() 16](#_Toc519433452)

[2.2.9 Stream.allMatch(), Stream.noneMatch(), Stream.anyMatch() 16](#_Toc519433453)

[2.3 *Date-Time API* 17](#_Toc519433454)

[2.3.1 Rad sa sistemskim vremenom i zonama 17](#_Toc519433455)

[2.3.2 Rad s datumom i vremenom 18](#_Toc519433456)

[2.3.3 Korištenjeklase *Duration* 18](#_Toc519433458)

[2.3.4 Korištenje klase *Period* 19](#_Toc519433459)

## Nove funkcionalnosti programskog jezika Java od verzije 1.7

* Koristiti unaprijeđenu naredbu *try*
* Upravljanje višestrukim *catch*-blokovima
* Kako koristiti objekt *string* u izjavi *switch*

### Unaprijeđena naredba *try*

U prijašnjim verzijama Jave, bilo je moguće koristiti samo *try/catch/finally* blokove, gdje bismo morali paziti na to da sve resurse koje smo instancirali po potrebi oslobodimo iz memorije. Dobar primjer toga je čitanje i pisanje datoteka, gdje bismo u *finally* bloku morali osloboditi resurse nakon što pročitamo datoteku ili je zapišemo na disk. U protivnome, datoteka bi bila nedostupna za daljnje korištenje jer bi tok za pisanje prema njoj ostao otvoren. Preduvjet za ovakav pristup je da resursi koje koristimo implementiraju sučelja java.lang.Autoclosable.

Korištenjem unaprijeđene naredbe *try* (*try with resources* izjava*)*, ne moramo se brinuti o oslobađanju resursa, već će to biti napravljeno automatski. Moguće je instancirati jedan ili više resursa.

//UpravljanjeDatotekama.java

public class UpravljanjeDatotekama

{

public void procitajDatoteku(String putanja)

{

try (BufferedReader br = new BufferedReader(new FileReader(putanja)) {

// Daljnja obrada datoteke...

}

}

}

U ovom primjeru, napravljena je instanca klase BufferedReader unutar *try* izjave. Primijetimo da nema *finally* bloka u kojem zatvaramo tok BufferedReader. Od Javine verzije 1.7 pa nadalje, svi resursi instancirani unutar *try with resources* izjave, bit će automatski oslobođeni iz memorije.

### Upravljanje višestrukim *catch*-blokovima

Obrađivanje iznimaka moguće je raditi korištenjem jednoga *catch*-izraza po bloku, a da bismo obradili više tipova iznimki, moramo navesti *catch*-blok za svaki tip posebno. U verzijama Jave 1.7 i nadalje, moguće je grupirati više tipova iznimki u jedan *catch*-blok.

#### Odvojeni *catch*-blokovi

//ViseVrstiIznimki.java

public class ViseVrstiIznimki

{

public static void main(String[] args)

{

int broj = 0;

String unos = "234d";

int[] polje = {1,2,3};

try {

// 1. greška

broj = new Integer(unos);

// 2. greška

broj = polje[5];

}

catch (NumberFormatException e) {

System.out.println("Greška pri konverziji znakovnog niza u broj");

}

catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Greška pri pristupu elementu iz polja");

}

}

}

#### Grupiranje *catch*-blokova

Primjer 1.2.1 moguće je modificirati na sljedeći način:

//ViseVrstiIznimki.java

public class ViseVrstiIznimki

{

public static void main(String[] args)

{

String unos = "234d";

int[] polje = {1,2,3};

try {

// 1. greška

broj = new Integer(unos);

// 2. greška

broj = polje[5];

}

catch (NumberFormatException | ArrayIndexOutOfBoundsException e) {

System.out.println("Znakovni niz mora biti ispravan, a pristup poljima unutar dozvoljenog raspona.");

}

System.out.println("Završetak programa.");

}

}

Ovim načinom dobili smo manje linija i čitljiviji kôd. Primijetimo da se sada nekoliko tipova iznimaka obrađuje na jednome mjestu, pa s ovakvim pristupom treba biti umjeren.

### Korištenje *stringova* u *switch*-izjavi

Još jedna novost proizašla iz nadogradnji Javina programskog jezika je mogućnost korištenja String tipa unutar *switch*-izjave. Do sada smo u *switch*-izjavi mogli koristiti samo integralne tipove kao što su *byte*, *short*, *int*, *long*, *char*. Specifičnost integralnih tipova podataka je ograničenost brojeva koje mogu prikazati zbog fiksne veličine memorije koju zauzimaju.

U idućem primjeru možemo vidjeti kako izgleda *switch*-izjava koja radi s objektom tipa *String*. Rezultat ovog programa ispisat će se na standardni izlaz broj 3, jer smo u *switch*-izjavi definirali srijedu kao treći dan u tjednu.

public class SwitchStringPrimjer

{

public static void main(String[] args)

{

String dan = "Srijeda";

System.out.println(kojiJeDanUTjednu(dan));

}

public static int kojiJeDanUTjednu(String nazivDana)

{

switch (dan) {

case "Ponedjeljak":

return 1;

case "Utorak":

return 2;

case "Srijeda":

return 3;

case "Četvrtak":

return 4;

case "Petak":

return 5;

case "Subota":

return 6;

case "Nedjelja":

return 7;

default:

return -1;

}

}

}

### Novi operatori u radu s generičkim tipovima

Generički tipovi, a jedan od njih je ArrayList, koji implementira sučelje List, morali su se instancirati specificiranjem tipa koji koristimo. Instanciranje ovoga generičkog tipa napisali bismo prema primjeru:

public static void main(String[] args)

{

List<String> daniUTjednu = new ArrayList<String>();

}

Sada možemo koristiti skraćenu verziju, korištenjem *diamond* operatora, koji pišemo kao <> (znakovi za manje/veće). Tako isti kôd možemo napisati prema sljedećem primjeru, a instanca generičkog tipa bit će automatski prepoznata:

public static void main(String[] args)

{

List<String> daniUTjednu = new ArrayList<>();

}

## Funkcionalnosti Javina programskog jezika od verzije 1.8

* Koristiti lambda-izraze
* Kako se koriste tokovi
* Novi API za rad s datumom i vremenom

### Lambda-izrazi i funkcionalna sučelja

U Javinu programskom jeziku prvi puta pojavili su se u verziji 1.8, a također predstavljaju i prvi puta da je Java zakoračila u funkcionalno programiranje.

Lambda-izraz je funkcija koja može biti kreirana bez da pripada nekoj klasi, a često se koristi za implementaciju jednostavnog mehanizma pogonjenog događajima (engl. *event handling mechanism*) i u funkcionalnom programiranju korištenjem Javina programskog sučelja za tokove (engl*. Java streams API*).

#### Funkcionalna sučelja

Funkcionalno sučelje ili sučelje s jednom apstraktnom metodom može se pretvoriti u lambda-izraz, ako sučelje zadovoljava ove uvjete:

* Ima li sučelje samo jednu apstraktnu metodu?
* Odgovaraju li parametri lambda-izraza parametrima funkcionalnog sučelja?
* Odgovara li povratni tip lambda-izraza povratnom tipu funkcionalnog sučelja?

Uzmimo za primjer jedno sučelje koje ima samo jednu apstraktnu metodu, ActionListener. Ovo sučelje ima metodu actionPerformed, koja se mora implementirati, a koristi se u mehanizmu pogonjenom događajima. Ovaj slučaj koristimo kada želimo nekoj kontroli, kao što je gumb, dodijeliti akciju u slučaju da korisnik klikne gumb mišem.

Izgled sučelja ActionListener:

public interface ActionListener extends EventListener

{

public void actionPerformed(ActionEvent ae);

}

Ovo sučelje do sada smo mogli implementirati korištenjem anonimne unutarnje klase:

//AnonimnaKlasaDogadajiPrimjer.java

public class AnonimnaKlasaDogadajiPrimjer

{

JButton btnPohraniPodatke;

public void registrirajDogadaj()

{

btnPohraniPodatke.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent ae) {

//Kod za pohranu podataka...

System.out.println("Podaci uspješno pohranjeni.");

}

});

}

}

Ista implementacija moguća je korištenjem lambda-izraza, a u primjeru ćemo vidjeti da ovim pristupom implementaciju možemo napraviti jednostavnijom, a kôd lakše čitljivim:

//LambdaDogadajiPrimjer.java

public class LambdaDogadajiPrimjer

{

JButton btnPohraniPodatke;

public void registrirajDogadaj()

{

btnPohraniPodatke.addActionListener((ActionEvent ae) -> {

//Kod za pohranu podataka...

System.out.println("Podaci uspješno pohranjeni.");

});

}

}

Pogledajmo razlike u odnosu na prvi način implementacije. Uočimo da nema instance anonimne klase koja implementira sučelje ActionListener kao i metode actionPerformed. Korištenjem lambda-izraza nije potrebno navoditi metodu koju implementiramo što je, sjetimo se pravila, moguće jer funkcionalno sučelje ima jednu apstraktnu metodu koju smo dužni implementirati. Jedino što smo zadržali iz prethodne implementacije je parametar ActionEvent.

Jedna novost ovdje je znak strelice -> (minus jednako), koja odvaja parametre s lijeve strane od izraza s desne strane, a parametar odgovara parametru u apstraktnoj metodi sučelja.

#### Referenciranje lambda-izrazima

Pristupanje svojstvima, metodama ili konstruktorima neke klase korištenjem lambda-izraza zovemo referenciranje. Ono nam pomaže kako bismo na jednostavniji način pristupili elementima s kojima radimo.

Moguće je referencirati sljedeće tipove metoda:

* statičke metode
* instance metoda na parametrima objekata
* metode na instancama
* konstruktore.

Referenciranje metoda pomoću lambda-izraza kada treba pozvati samo jednu metodu s parametrima. Najprije smo napravili vlastito sučelje, a zatim primjer kako lambda-instanca može implementirati MojaLambdaPrimjer sučelje:

public interface MojaLambdaPrimjer

{

public void ispisi(String s);

}

// Primjer implementacije sučelja

MojaLambdaPrimjer primjer = (s) -> { System.out.println(s); }

Referenciranje metoda lambda-izrazima: Object :: imeMetode, kraći je način zapisivanja lambda-izraza (m) -> m.imeMetode()

Referenciranje konstruktora instanciranjem sučelja Function:

java.util.function.Function<String, String> func = String::new;

System.out.println(func.apply("ovo je tekst"));

### Javini tokovi (engl. *streams*)

Javini tokovi su novi način rada s Javinim kolekcijama i nemaju vezu s ulazno-izlaznim tokovima, iako je ideja slična – koristi se tok objekata umjesto toka bajtova ili znakova. Tokovi su dizajnirani tako da se mogu kombinirati s lambda-izrazima, a možemo reći da predstavljaju proširenje funkcionalnosti za kolekcije te omogućavaju ulančavanje izraza. U narednih nekoliko primjera vidjet ćemo kako se pomoću tokova može obavljati razne operacije nad kolekcijama.

#### Stream.filter()

//LambdaTokoviPrimjer.java

public class LambdaTokoviPrimjer

{

public void filtrirajeDane()

{

List<String> daniUTjednu = new ArrayList<>();

daniUTjednu.add("Ponedjeljak");

daniUTjednu.add("Utorak");

daniUTjednu.add("Srijeda");

daniUTjednu.add("Četvrtak");

daniUTjednu.add("Petak");

daniUTjednu.add("Subota");

daniUTjednu.add("Nedjelja");

List<String> daniKojiPocinjuSlovomP = daniUTjednu

.stream()

.filter(d -> d.startsWith("P"))

.collect(Collectors.toList());

}

}

Pogledajmo što se u metodi filtrirajDane događa. Inicijalno imamo kolekciju String objekata, u koju smo dodali dane u tjednu. Nakon toga napravili smo novu kolekciju daniKojiPocinjuSlovomP, te joj filtriranjem kopirali vrijednosti iz prve kolekcije prema zadanom uvjetu.

Da bismo filtrirati kolekciju, najprije moramo pristupiti toku metodom stream. Nakon što smo pristupili toku kolekcije, ulančavanjem pozivamo metodu filter, u koju smo proslijedili lambda-izraz. Lambda-izraz u metodi filter je samo zamjena za implementaciju funkcionalnog sučelja Predicate. Implementacija Predicate sučelja mogla se napisati i bez lambda-izraza, ali nije praktično i manje je čitljivo.

Sučelje Predicate prima parametar te vraća vrijednost tipa boolean, ovisno o tome je li uvjet zadovoljen ili ne. U našem slučaju uvjet zadovoljavanja je da tekst počinje znakom „P“. Varijabla d u lambda-izrazu predstavlja naziv parametra, preko kojeg pristupamo svojstvima i metodama objekta nad kojim radimo ovu operaciju (u našem slučaju to je String), a mogli smo napisati bilo koji naziv.

Posljednja ulančana metoda collect jedna je od metoda na Stream sučelju, a njezin je zadatak pokupiti sve elemente koji su zadovoljili uvjet u metodi filter te vratiti objekte u novu kolekciju.

#### Stream.min() i Stream.max()

Za istu kolekciju iz prethodnog primjera, pogledajmo kako možemo pomoću toka pronaći koji dan u tjednu ima najmanje znakova. Za element s najviše znakova, samo primijenimo metodu max.

//LambdaTokoviPrimjer.java

public class LambdaTokoviPrimjer

{

public void ispisiDanSNajmanjeZnakova(List<String> daniUTjednu)

{

String danSNajmanjeZnakova

= daniUTjednu

.stream()

.min(Comparator.comparing(d -> d.length()))

.get();

}

}

Metode min i max vraćaju instancu klase Optional koja u sebi ima metodu get, a služi da bismo dohvatili svoju vrijednost. Metodemini max primaju Comparator kao parametar.

Metoda Comparator.comparing kreira instancu tipa Comparator ovisno o lambda-izrazu koji smo proslijedili. Zapravo, metoda comparing prima tip Function koji je funkcionalno sučelje, a zadovoljava uvjete za pretvorbu u lambda-izraz.

#### Stream.count()

Metoda count korisna je kada želimo vratiti broj elemenata unutar toka nakon što primijenimo filtriranje.

//LambdaTokoviPrimjer.java

public class LambdaTokoviPrimjer

{

public void ispisiDanSNajmanjeZnakova(List<String> daniUTjednu)

{

long brojDanaKojiPocinjuSlovomP

= daniUTjednu

.stream()

.filter(d -> d.startsWith("P"))

.count();

}

}

Rezultat izraza u ovom primjeru vraća broj elemenata koji zadovoljavaju uvjet prema zadanom filteru. Prolaskom kroz kolekciju sačuvaju se svi elementi koji počinju slovom „P“ i na kraju vrati njihov broj. Možemo primijetiti da rezultat ima povratni tip long.

#### Stream.map()

Još jedna važna funkcionalnost tokova je mapiranje objekata. Mapiranje objekata znači da prolaskom kroz kolekciju svaku njezinu stavku možemo pretvoriti u novi objekt temeljem te stavke. Pogledajmo jednostavan primjer mapiranja:

//LambdaTokoviPrimjer.java

public class LambdaTokoviPrimjer

{

public void pretvoriUVelikaSlova(List<String> daniUTjednu)

{

List<String> daniVelikimSlovima

= daniUTjednu

.stream()

.map(d -> d.toUpperCase())

.collect(Collectors.toList());

}

}

Ovaj primjer mapira stavke iz prve kolekcije u velika slova, a na kraju kao i kod filter-metode, pokupimo sve elemente i dodamo ih u novu kolekciju. Prema primjeru možemo vidjeti da je mapiranjem vraćena kolekcija stringova. Osim stringova, moguće je vratiti i brojeve korištenjem metoda mapToDouble, mapToInt te mapToLong. U idućem primjeru pokazat ćemo kako ih možemo iskoristiti.

#### Stream.average i Stream.sum()

Prosjek i zbroj nekih vrijednosti možemo napraviti korištenjem metoda average i sum, u slučaju kada su kolekcije nekog numeričkog tipa. Kolekcija ne mora nužno biti numerički tip, ali ako kolekcije dohvate neko svojstvo koje možemo mapirati na numerički tip, također možemo primijeniti ove metode.

//LambdaTokoviPrimjer.java

public class LambdaTokoviPrimjer

{

public double averagePrimjer(List<String> daniUTjednu)

{

return daniUTjednu

stream()

.mapToInt(d -> d.length())

.average()

.getAsDouble();

//rezultat: 7.28571428571

}

public int sumPrimjer(List<String> daniUTjednu)

{

return daniUTjednu

stream()

.mapToInt(d -> d.length())

.sum();

//rezultat: 51

}

}

U averagePrimjer metodi najprije smo iz svoje kolekcije stringova za svaki element umjeste njegove vrijednosti uzeli duljinu znakova. Da smo imali objekt s nekim svojstvom tipa int ili double, mogli smo uzeti to svojstvo i pozvati metodu average. Na primjer, godine korisnika, pa izračunati njihov prosjek. Na kraju smo samo metodom getAsDouble vratili rezultat tipa double.

Metoda sum radi slično, ali nismo morali koristiti getAsDouble već vraćamo onaj tip koji smo postavili u metodi map, u ovom slučaju to je int.

#### Stream.reduce()

Metoda *reduce* može spojiti sve elemente jednog toka u jednu vrijednost. Primjerom ćemo pokazati kako sve dane u tjednu ovom metodom možemo spojiti u jedan objekt tipa String.

//LambdaTokoviPrimjer.java

public class LambdaTokoviPrimjer

{

public void spojiDane(List<String> daniUTjednu)

{

String sviDani = daniUTjednu

.stream()

.reduce((dani, el) -> dani + ", " + el)

.get();

}

}

Metoda *reduce* prima tip BinaryOperator kao parametar, a sadržava metodu *apply* koju implementiramo korištenjem lambda-izraza, te dva parametra: *dani*, koji predstavljaju akumulirane, odnosno spojene elemente, i *el* koji predstavlja element unutar toka. Ako tok nema elemenata, metoda *get* vratit će *null* kao vrijednost.

Rezultat ispisa varijable sviDaniizgledao bi u našem slučaju ovako: Ponedjeljak, Utorak, Srijeda, Četvrtak, Petak, Subota, Nedjelja

#### Stream.sorted()

Još jedna u nizu korisnih metoda u Javinim tokovima je metoda sorted, koja može sortirati kolekcije uzlazno, silazno, te prema svojstvu nekog objekta, a u ovom primjeru ćemo koristiti klasu Korisnik kako bismo mogli prikazati sortiranje na bilo kojem objektu.

Za sljedeće primjere koristit ćemo kolekciju objekata tipa Korisnik*,* koju ćemo sami napraviti. U kolekciju dodajemo varijable ime i prezime, a zatim napravimo za njih get i setmetode. Radi jednostavnosti, u istu klasu ćemo dodati statičku metodu dohvatiKorisnike, a zatim ju u glavnom programu pozvati i prosljeđivati njezin rezultat kao izvor podataka za sortiranje.

//Korisnik.java

public class Korisnik

{

private String ime, prezime;

private int id;

public int getId() {

return this.id;

}

public void setId(int id) {

this.id = id;

}

public String getIme() {

return this.ime;

}

public void setIme(String ime) {

this.ime = ime;

}

public String getPrezime() {

return this.prezime;

}

public void setPrezime(String prezime) {

this.prezime = prezime;

}

//Radi jednostavnijeg ispisa u konzolu

@Override

public String toString() {

return this.ime + " " + this.prezime;

}

public static List<Korisnik> dohvatiKorisnike()

{

List<Korisnik> korisnici = new ArrayList<>();

Korisnik k1 = new Korisnik();

k1.setId(1);

k1.setIme("Miro");

k1.setPrezime("Mirić");

korisnici.add(k1);

Korisnik k2 = new Korisnik();

k2.setId(2);

k2.setIme("Ivo");

k2.setPrezime("Jurić");

korisnici.add(k2);

Korisnik k3 = new Korisnik();

k3.setId(3);

k3.setIme("Miro");

k3.setPrezime("Jurić");

korisnici.add(k3);

return korisnici;

}

}

Glavni program:

//LambdaTokoviPrimjer.java

public class LambdaTokoviPrimjer

{

public static void main(String[] args)

{

List<Korisnik> korisnici = Korisnik.dohvatiKorisnike();

sortirajKorisnike(korisnici);

sortirajKorisnikePrezimeIme(korisnici);

}

public void sortirajKorisnike(List<Korisnik> korisnici)

{

List<Korisnik> sortiranoPoPrezimenuUzlazno

= korisnici

.stream()

.sorted(Comparator.comparing(k -> k.getPrezime()))

.collect(Collectors.toList());

System.out.println(sortiranoPoPrezimenuUzlazno);

List<Korisnik> sortiranoPoPrezimenuSilazno

= korisnici

.stream()

.sorted(Comparator.comparing(Korisnik::getPrezime).reversed())

.collect(Collectors.toList());

System.out.println(sortiranoPoPrezimenuSilazno);

}

}

U metodi sortirajKorisnike nalaze se dvije kolekcije u koje dodajemo sortirane elemente iz proslijeđene kolekcije korisnici. Prvi primjer sortira kolekciju uzlazno po prezimenu. Metoda sorted prima kao parametar Comparator koji smo imali priliku upoznati u primjeru s metodama min i max. Comparator uspoređuje elemente kolekcije po zadanom ključu, u našem primjeru po prezimenu. Druga metoda je vrlo slična, ali ovaj puta kolekcija je sortirana silazno po prezimenu.

Sljedeća metoda demonstrira kako se sortirati može korištenjem više ključeva. Ovaj puta kolekciju sortiramo po prezimenu, a zatim nakon metode comparingpozovemo metodu thenComparingi proslijedimo parametar getIme.

//LambdaTokoviPrimjer.java

public class LambdaTokoviPrimjer

{

...

public void sortirajKorisnikePrezimeIme(List<Korisnik> korisnici)

{

List<Korisnik> sortiranoPoPrezimenuImenu

= korisnici

.stream()

.sorted(Comparator.comparing(Korisnik::getPrezime)

.thenComparing(Korisnik::getIme))

.collect(Collectors.toList());

System.out.println(sortiranoPoPrezimenuSilazno);

}

}

#### Stream.findFirst() i Stream.orElse()

Metoda findFirst pronalazi prvi element unutar toka prema zadanom kriteriju metodom filter. Nakon metode findFirst, možemo dohvatiti traženi element metodom get, ali u slučaju da nije pronađen niti jedan element, dobit ćemo iznimku java.util.NoSuchElementException. Bolji pristup je koristiti metodu orElsei kao parametar proslijediti vrijednost null. U tom slučaju, ako element nije pronađen, kao rezultat dobit ćemo *null*-vrijednost umjesto iznimke.

//LambdaTokoviPrimjer.java

public class LambdaTokoviPrimjer

{

...

public Korisnik pronadiKorisnikaPoIdu(List<Korisnik> korisnici, int id)

{

Korisnik korisnik = korisnici

.stream()

.filter(k -> k.getId() == id)

.findFirst()

//.get()

.orElse(null);

return korisnik;

}

}

#### Stream.allMatch(), Stream.noneMatch() , Stream.anyMatch()

Metoda allMatch provjerava odgovaraju li svi elementi zadanom uvjetu unutar predikata. Metoda noneMatch provjerava nema li elemenata koji zadovoljavaju zadani uvjet predikata. Ako je tok elemenata prazan, izvršavanje se ne provodi, a true dobijemo kao rezultat. Posljednja od ove tri metode, anyMatch provjerava postoji li barem jedan element koji zadovoljava zadani uvjet. Sve tri metode imaju povratni tip boolean.

//LambdaTokoviPrimjer.java

public class LambdaTokoviPrimjer

{

...

public boolean allMatchPrimjer(List<Korisnik> korisnici)

{

return korisnici

.stream()

.allMatch(k -> k.getPrezime().length() > 0);

}

public boolean noneMatchPrimjer(List<Korisnik> korisnici)

{

return korisnici

.stream()

.noneMatch(k -> k.getPrezime().length() > 100);

}

public boolean anyMatchPrimjer(List<Korisnik> korisnici)

{

return korisnici

.stream()

.anyMatch(k -> k.getPrezime().startsWith("M"));

}

}

### *Date-Time API*

Datum i vrijeme u prijašnjim izdanjima Javina jezika nisu bili najprikladniji za upotrebu. Stoga je napravljen novi API, koji rad s datumom i vremenom čini prikladnijima. U iduća dva prmjera prikazat ćemo način korištenja sistemskog sata/zona i rad s datumom/vremenom.

Za rad s klasama koje upravljaju datumom i vremenom potrebno je uključiti paket *java.time.\**

#### Rad sa sistemskim vremenom i zonama:

public class DatAndTimeAPIPrimjer {

public static void main(String[] args) {

// sistemsko vrijeme postavljeno na UTC-u

Clock clockUtc = Clock.systemUTC();

// sistemsko vrijeme

Clock clockDefault = Clock.systemDefaultZone();

// vrijeme u milisekundama od 1.1.1970.

long vrijemeMs = clockDefault.millis();

// Vremenska zona prema sistemu

ZoneId zona = ZoneId.systemDefault();

// Postavljanje sata u određenu vremensku zonu

Clock clockSaZonom = Clock.system(zona);

// Traženje id-a određene zone

ZoneId zonaBerling = ZoneId.of("Europe/Berlin");

}

}

Za rad s datumom i vremenom koristimo klase LocalDate (rad s datumima), LocalTime (rad s vremenom) i LocalDateTime (rad s datumom i vremenom).

#### Rad s datumom i vremenom

public class DatAndTimeAPIPrimjer {

public static void main(String[] args) {

LocalDate datum = LocalDate.now();

System.out.println("Godina: " + datum.getYear());

System.out.println("Mjesec: " + datum.getMonthValue());

System.out.println("Dan: " + datum.getDayOfMonth());

}

}

Metoda LocalDate.now dohvaća današnji datum, a rezultat ispisivanja ovog programa bi ispisao godinu, mjesec te dan u mjesecu za aktualni datum. Za datum 22. 2. 2022. na izlazu bismo dobili rezultat:

Godina: 2022

Mjesec: 2

Dan: 22

Pretvorba (engl. *parse*) teksta u datum ili vrijeme. Kod pretvorbe u tip LocalDateTimemožemo koristiti klasu DateTimeFormatter kako bismo jasno naznačili format koji parser može očekivati.

public class DatAndTimeAPIPrimjer {

public static void main(String[] args) {

LocalTime vrijeme = LocalTime.parse("11:33:45");

System.out.println(vrijeme);

DateTimeFormatter format = DateTimeFormatter.ofPattern("dd.MM.yyyy HH:mm");

LocalDateTime datumVrijeme = LocalDateTime.parse("22.05.2013 12:22:05", format);

System.out.println(datumVrijeme); }

}

#### Korištenjeklase *Duration*

public static void main(String[] args) {

Duration jedanSat = Duration.ofHours(1);

// Koliko sekundi ima jedan sat

System.out.println(jedanSat.getSeconds());

// Trajanje u sekundama između dva vremenska razdbolja

LocalDateTime pocetak = LocalDateTime.of(2010, 5, 15, 17, 55);

LocalDateTime kraj = LocalDateTime.of(2018, 5, 22, 14, 33);

Duration trajanje = Duration.between(pocetak, kraj);

System.out.println(trajanje.getSeconds()); // rezultat: 253053480

}

#### Korištenje klase *Period*

public static void main(String[] args) {

LocalDate pocetak = LocalDate.of(1982, Month.AUGUST, 31);

LocalDate kraj = LocalDate.of(2016, Month.NOVEMBER, 9);

Period period = Period.between(pocetak, kraj);

System.out.println(period.getYears() + " godine");

System.out.println(period.getMonths() + " mjeseca");

System.out.println(period.getDays() + " dana");

// rezultat ispisa: 34 godine

2 mjeseca

9 dana

}

U primjeru možemo vidjeti kako je na jednostavan način moguće računati s vremenom. Između dva datuma kao rezultat dobili smo razliku u godinama, mjesecima i danima.