OBJEKTNO ORIJENTISANO PROGRAMIRANJE 2

ODGOVORI NA PITANJA ZA USMENI DEO ISPITA

JUN 2021

Git: https://github.com/NikolaVetnic/OOP2

01 Pregled novih mogućnosti koje donosi Java 5. Sintaksa, opis, primena, prednosti i nedostaci (ukoliko postoje) svake mogućnosti.

***ENUMERATED TYPES*: *enum*-i se koriste nad skupovima podataka kod kojih su sve vrednosti poznate u vreme kompajliranja; pre Jave 1.5 *enum*-i su se simulirali korišćenjem promenljivih tipa int:

```
class Dani { public static final int PON = 1; public static final int UTO = 2; ... }
```

Nedostataci ovakvog pristupa: **1)** nemogućnost provere tipa u vremenu kompajliranja, **2)** nemogućnost iteracije kroz vrednosti, **3)** problemi prilikom pridruživanja arbitrarnog tipa podataka nabrojivim vrednostima. Java 1.5 nudi **rešenje** u vidu tipa *enum*-a – referencijalnog tipa čija se polja sastoje od fiksnog skupa konstanti odvojenih zarezima:

```
class Dani { PON, UTO, SRE, CET, PET, SUB, NED }
```

Pored polja *enum*-i mogu sadržati i svojstva (npr. `PON("PON", "MO");`), konstruktore, metode, unutrašnje klase, itd; svako polje *enum*-a može imati sopstvenu implementaciju metoda a time i drugačije ponašanje, a postoji i (ograničena) podrška za nasleđivanje.

***GENERIČKI TIPOVI*:** dozvoljavaju apstrakciju tipova, odnosno parametrizaciju klasa, interfejsa i metoda tipovima, čega je posledica *type-safe* kôd – ako se kôd kompajlira bez grešaka ili upozorenja onda je sigurno da neće biti bačeni *typecasting* izuzeci u vremenu izvršavanja; gen. tipovi se najčešće koriste kod kolekcija objekata, a takođe olakšavaju čitanje (nakon navikavanja na sintaksu). Primer parametrizovane *Stack* klase:

```
class Stack {
  void push(E element) { ... };
  void pop() { ... };
  E top() { ... };
}
```

Klasa se koristi tako što se parametar `E` zameni argumentom konkretnog tipa.

***AUTO(UN)BOXING*:** *box* – instanca *wrapper* klase koja sadrži vrednost prim. tipa (`Integer`, `Float`…), pri čemu je *boxing* kreiranje *box*-a za prim. vrednost, a *unboxing* uklanjanje takve vrednosti; u slučaju da tip podataka zahteva ref. tip (kao što je slučaj sa kolekcijama) prim. tip se mora zapakovati u objekat što se pre Jave 1.5 radilo „ručno“, kao i vraćanje vrednosti:

```
int i = 5;
LinkedList list = new LinkedList();
list.add(new Integer(i)); // boxing
Integer n = list.get(0);
int i = n.intValue(); // unboxing
```

Java 1.5 automatizuje ove akcije i izjednačava `boolean` i `Boolean` u proveri uslova – manje kôda, lakše čitanje; nedostaci: **1)** kod poređenja prim. tipa i *wrapper* objekta učestvuju vrednosti, ali kod dva *wrapper*-a reference (koristiti `equals()`), **2)** odveć čest (un)`boxing` zauzima resurse računara, **3)** `unboxing` `null`-a baca izuzetak. *Suma sumarum*: **1)** *auto(un)boxing* skriva razliku između prim. i ref. tipova ali je ne eliminiše, **2)** `Integer` nije zamena za `int`, **3)** ograničiti upotrebu u kôdu koji zahteva performanse.

***VARARGS*:** reč je o *feature*-u koji omogućava definisanje metoda korišćenjem varijabilnog broja argumenata (nula ili više) koji svi moraju biti istog tipa ali čiji broj ne mora biti predefinisan; pre 1.5 *workaround* je podrazumevao korišćenje niza:

```
void oldPrintAll(int k, String[] s) {
  System.out.println(k);
  for (int i = 0; i < s.length; i++)
    S.O.P.(s[i]);
}
String[] s = { "foo", "bar" };
oldPrintAll(7, s);
```

Vararg parametar je notiran elipsom "...", unutar metoda se njime rukuje kao sa nizom, međutim metod se poziva sa bilo kojim brojem parametara bez pakovanja u niz:

```
void newPrintAll(int k, String... s) {
  System.out.println(k);
  for (int i = 0; i < s.length; i++)
    S.O.P.(s[i]);
}
newPrintAll(6, "foo", "bar");
newPrintAll(7);
```

Vararg se može javiti isključivo kao poslednji parametar u zaglavlju ali ih uopšte govoreći treba izbegavati, pogotovo tamo gde je potreban *overloading* metoda (može biti teško za shvatiti koji metoda se poziva).

POBOLJŠANA `for` PETLJA: nova `for` petlja omogućava lakšu iteraciju kroz kolekciju/niz, bez potrebe da se definiše `Iterator` ili indeksirajuća promenljiva što, naročito pri korišćenju `Iterator`-a za kolekcije, daje kraći i čitljiviji kôd; sintaksa za kolekcije koje implementiraju `Iterator` (ali radi jednako i sa nizovima) – `for (type variable : collection)` :

```
for (Iter<Int> i = c.iter (); i.hasNext(); )          ➔  for (int n : c) ...
for (int i = 0; i < arr.length; i++) ...              ➔  for (int n : arr) ...
```

Nedostatak se naravno ogleda u nemogućnosti pristupa tačno određenom indeksu.

STATIČKI IMPORTI: rešenje problema neophodnosti korišćenja kvalifikovanog imena za svakog statičkog člana referenciranog iz eksternih klasa (`Math.cos` i `Math.PI` uz `import static java.lang.Math.cos` i `.PI` postaju `cos` i `PI`); *wildcard* `*` je i dalje dozvoljen.

ANOTACIJE: u pitanju su jezički konstrukti koji dodeljuju dodatnu semantiku elementima izvornog kôda – klasama, metodima, poljima; anotacija se može procesirati: u vremenu kompajliranja od strane kompajlera, u vremenu kompajliranja od strane eksternog soft. alata koji npr. izbacuje XML, u vremenu izvršavanja, od strane eksternog alata/biblioteke; dev-ovi mogu pisati sopstvene anotacije ali se to retko radi; opšta sintaksa:

```
@annotationName (optionalArguments)
elementThatIsBeingAnnotated
```

Ugrađene anotacije: **1)** `@Override` – obaveštava kompajler da se metoda nadklase *override*-uje trenutnom (preporučljivo, budući da olakšava izbegavanje grešaka – primer `toStrinq()`), **2)** `@Deprecated` – beshrabruje upotrebu elementa, najčešće jer postoje bolje alternative (slučaj starog elementa koji se zadržava samo zbog *back* kompatibilnosti), **3)** `@SuppressWarning( {set_of_warnings} )` – obaveštava kompajler da ignoriše dat skup upozorenja pri obradi anotiranog elementa.

KOVARIJANTNI POVRATNI TIPOVI: sada je moguće da metod koji *override*-uje super vraća objekat čiji je tip podklasa povratnog tipa metoda iz nadklase čime se omogućava pružanje više informacija o povratnom objektu i eliminiše potreba za *cast*-ovanjem:

```
class A {                                   class B extends A {
  public A copy() { return new A(); } }       public B copy() { return new B(); } }
```

02 Pregled novih mogućnosti koje donosi Java 7. Sintaksa, opis, primena, prednosti i nedostaci (ukoliko postoje) svake mogućnosti.

NUMERIČKI LITERALI: binarni literali, donje crte dozvoljene između cifara zarad lakše čitljivosti:

```
byte aByte = (byte) 0b0010001;              long hexBytes = 0xFF_EC_DE_5E;
long creditCardNum = 1234_5678_9012_3456L;  long maxLong = 0x7fff_ffff_ffff_ffffL;
```

***STRING*-OVI U SWITCH NAREDBAMA:** za poređenje kao da se koristi `String.equals()`:

```
String dayOfWeek = "Monday";                switch (dayOfWeek) { ... }
```

INFERENCIJA TIPOVA ZA GENERIČKE INSTANCE: dozvoljeno prosleđivanje praznog skupa parametara sve dok kompajler može da zaključi o kojim se tipovima radi iz konteksta:

```
Map<String, List<String>> myMap = new HashMap<>();
```

***TRY-WITH-RESOURCES* NAREDBA:** reč je o `try` naredbi koja deklariše jedan ili više resursa – objekata koji moraju biti zatvoreni nakon što program završi sa njima, o čemu se brine ovakva varijanta `try` naredbe; bilo koji objekti koji implementiraju `java.lang.AutoClosable` (a to su svi objekti koji implementiraju `java.io.Closable`) mogu se koristiti kao resursi (primer).

HVATANJE VIŠESTRUKIH TIPOVA IZUZETAKA: sada jedan `catch` blok može da obradi više od jednog tipa izuzetka, što je *feature* koji redukuje dupliciranje kôda i obeshrabruje pokušaje hvatanja suviše širokog izuzetka:

```
catch (IOException | SQLException e) { logger.log(ex); throw ex; }
```

03

Pregled novih mogućnosti koje donose Java 8, 9, 10 i 11. Sintaksa, opis, primena, prednosti i nedostaci (ukoliko postoje) svake mogućnosti.

JAVA 8, LAMBDA IZRAZI: anonimna f-ja ili metod, sastoji se od: **1)** nula ili više parametara (zapisanih sa ili bez tipa), **2)** lambda operatora `->`, **3)** tela (u slučaju samo jedne naredbe { } nisu neophodne); lambde se koriste najviše pri obradi nizova/kolekcija, i kao zamena za anonimne unutrašnje klase; Java 8 uključuje i novi *Stream API* sa brojnim metodama koje prihvataju lambda izraze (`forEach()`, `filter()`, `map()`…), a same kolekcije se lako pretvaraju u tok `.stream()` metodom (`Arrays.stream()` za nizove).

JAVA 8, REFERENCE NA METODU: služe za izbegavanje pisanja lambde za koju već postoji metod – umesto prosleđivanja lambde prosleđuje se referenca na metod (primer), nakon čega se prosleđeni metod primenjuje na sve elemente toka:

```
ClassName::method // statički metoda                String::equals // nestatički met. ref. tipa
object::method // metoda objekta                    ClassName::new // konstruktor
```

JAVA 8, FUNKCIONALNI INTERFEJSI: interfejs sa samo jednim metodom, dozvoljava pisanje metoda koji prihvataju lambde (anotiranje je opciono – govori kompajleru o čemu je reč i neće dozvoliti dodavanje drugog metoda):

```
@FunctionalInterface interface Predicate { void apply(String str); }
```

Metod koji prihvata f-ionalni interfejs i listu `String`-ova i primenjuje metod interfejsa na svaki element:

```
void process(Predicate p, List<String> l) {
  for (String s : l) p.apply(s); }
```

Metod se takođe može pozvati ovako (prosleđuje se implementacija metoda `apply`):

```
process(str -> System.out.println(s), l);
```

JAVA 8, *DEFAULT* METODI: interfejsi sada mogu implementirati *default* metode, što je korisno u slučaju potrebe dodavanja metoda postojećim interfejsima unutar velikih projekata (budući da se metoda može dodati bez potrebe za promenom svih klasa koje ga implementiraju); *dev*-ovi Java biblioteka koriste ovaj *feature* za proširenje standardnih interfejsa kako bi se pružila podrška za lambde.

JAVA 9, MODULI: ranije su postojali veliki problemi sa **1)** enkapsulacijom (npr. pristup privatnim poljima je bio jednostavan) i **2)** pouzdanom konfiguracijom (npr. „varalice“ u `CLASSPATH`-u) – Java 9 uvodi mogućnost da JAR fajl sadrži `module-info.java` fajl poput ovog:

```
module mymodule { exports mypackage; requires oneother module; }
```

Samo sadržaj paketa `mypackage` je vidljiv drugim modulima, `mymodule` može da pristupi samo sadržaju modula koji su mu neophodni (`requires`), `MODULE-PATH` sad postoji pored `CLASSPATH` -a, a "Unnamed module" pruža podršku za staro ponašanje.

JAVA 9, PRIVATNI METODI INTERFEJSA: od uvođenja *default* metoda u Javi 8 postojao je problem ponavljanja kôda – ovo je rešeno u Javi 9 uvođenjem `private` *helper* metoda koje se pozivaju iz (više) *default* metoda interfejsa.

JAVA 10 I 11, INFERENCIJA `local-variable` TIPA: mogućnost izbegavanja eksplicitne deklaracije lokalnih varijabli, sa ograničenjem na lokalne varijable sa inicijalizatorima, indekse poboljšanih for petlji i lokalne varijable deklarisane u tradicionalnim for petljama:

```
// infers ArrayList<String>
var list = new ArrayList<String>();
```

```
// infers Stream<String>
var stream = list.stream();
```

04 Pojam toka podataka (*stream*). Problemi sa Javinim I/O sistemom i *Decorator* obrazac. Osnovni tipovi tokova podataka. Zatvaranje *stream*-ova. *Buffered streams*.

Kreiranje dobrog I/O sistema je jedan od najtežih zadataka za dizajnere jezika budući da unutar njega postoje različiti izvori i *sink*-ovi sa kojima želite da komunicirate (fajlovi, konzola, *pipe*-ovi, mrežne konekcije, itd.), a takođe sa svima se mora komunicirate na sijaset različitih načina: sekvencijalno, *random-access*, baferovano, binarno, po karakteru, liniji, reči, itd. Dizajneri Jave prišli su problemu kroz kreiranje mnoštva klasa, tako da rešenje bilo kog pojedinačnog problema nije naročito teško, ali je trik u pronalaženju puta kroz lavirint mogućnosti.

I/O TOKOVI: reč je o ulaznom izvoru ili izlaznoj destinaciji (razlikuje se i starije je od Java *stream* APIja iz Jave 8), može da predstavlja različite vrste izvora i odredišta (fajlove, mrežne konekcije, memorijske lokacije), a takođe podržava i različite tipove podataka (proste bajtove, primitivne tipove, lokalizovane karaktere, objekte); neki tokovi prosto prosleđuju podatke, dok drugi njima manipulišu i transformišu ih na korisne načine; ipak, bez obzira kako rade interno, svi tokovi predstavljaju isti jednostavan model za programe koji ih koriste – to su uvek sekvence podataka. Svi I/O tokovi nalaze se u java.io paketu zajedno sa podržavajućim klasama; ovakvi tokovi su jednosmerni – kreirani su ili za čitanje ili za pisanje podataka, ali nikako za oba; mogu se grupisati na nekoliko načina:

1) tokovi bajtova i karaktera – ovi tokovi vrše operacije nad sirovim 8-bitnim bajtovima i nude metode za čitanje ili pisanje pojedinačnih bajtova ili niza bajtova (korenske klase za tokove bajtova su `InputStream` i `OutputStream`); tokovi karaktera sa druge strane pružaju I/O zasnovan na *Unicode*-u (korenske klase su `Reader` i `Writer`); sve četiri klase su `abstract`; ovakvi tokovi inače predstavljaju jedan vid I/O niskog nivoa i koriste se svi manje-više na isti način, s tim da je razlika u načinu na koji su kreirani; njihovo korišćenje opravdano je u slučajevima kada se vrše operacije nad sirovim binarnim podacima (tj. ako podaci sadrže karaktere najbolji način obrade je korišćenje tokova karaktera); svi ostali tokovi se oslanjaju na ovaj tip;

2) ulazni i izlazni tokovi – ulazni tokovi koriste se za čitanje podataka (korenske klase su `InputStream` i `Reader`), dok su izlazni ili *sink* (ili odredišni) namenjeni pisanju podataka (`OutputStream` i `Writer`);

3) tokovi čvorova (*node*) i filtera – tokovi čvorova su tokovi niskog nivoa koji sadrže osnovnu funkcionalnost za čitanje iz ili pisanje u memorijske lokacije, fajlove, i slično; filteri ili procesirajući tokovi se naslojavaju na tokove čvorova zarad postizanja dodatne funkcionalnosti u vidu izmene ili rukovanja podacima u toku (primer: `ZipInputStream` za čitanje fajlova iz *ZIP* arhiva).

ZATVARNJE TOKOVA: svaki I/O tok daje metodu `close()` za zatvarnje toka što je, nakon što tok više nije potreban, veoma važno – toliko važno zapravo da bi program morao da koristi `finally` blok kako bi garantovao da će se tok zatvoriti čak i ako se desi greška (praksa koja sprečava curenje resursa i gubitak podataka); problem leži u činjenici da `close()` metod može generisati `IOException` (npr. ukoliko program pokušava da zatvori fajl koji je već zatvoren), što je razlog zašto se postavlja `try ... catch` blok unutar `finally` bloka; alternativa je naravno `try with resources` (od Jave 7).

***DECORATOR* OBRAZAC:** sledeća naredba prikazuje kako se dodatna funkcionalnost može primeniti na postojeći objekat na način dodavanja novog sloja povrh postojećeg:

```
reader = new InputStreamReader(new FileInputStream("C:\\sample.txt", "UTF-8"));
```

U primeru FileInputStream vraća sirove bajtove, a onda ih InputStreamReader pretvara u karaktere; u svetu OO dizajna korišćenje naslojenih objekata kako bi se dinamički i transparentno dodale odgovornosti pojedinačnim objektima se naziva *Decorator* obrazac; dekoratori se često koriste kada prosto „podklasiranje" rezultuje velikim brojem klasa kako bi se zadovoljile sve moguće potrebne kombinacije, toliko velikim da to postaje nepraktično; *Decorator* obrazac dodaje mnogo više fleksibilnosti prilikom pisanja programa budući da se atributi mogu lako mešati i kombinovati po potrebi; nedostatak ovakvog pristupa je povećana složenost kôda – razlog zašto je Java I/O biblioteka tako nezgodna za upotrebu je u činjenici da je neophodno kreirati mnoštvo objekata kako bi se dobio jedan jedini I/O objekat.

BAFEROVANI TOKOVI: nebaferovan I/O znači da je svaki zahtev za čitanje/pisanje obrađen direktno od strane OS-a što može učiniti programe mnogo manje efikasnim, budući da svaki takav zahteva okida pristup disku, mrežnu aktivnost ili neku drugu operaciju koja je zaista skupa; kako bi se smanjio trošak Java implementira baferovane I/O tokove koji čitaju podatke iz dela memorije zvanog *buffer*, dok se nativni ulazni API zove samo kada je *buffer* prazan; slično, izlazni tokovi pišu podatke u *buffer* i pozivaju nativni izlazni API samo kada je *buffer* pun; kreiranje baferovanih tokova je lako uz korišćenje dekoratora; za čitanje/pisanje bajtova postoje `BufferedInputStream` i `BufferedOutputStream`, dok za karaktere postoje `BufferedReader` i `BufferedWriter` koji, pored veće efikasnosti pružaju i zgodni `public String readLine()` metoda – svaka od ovih klasa ima konstruktor koji prihvata objekat nebaferovanog toka sa nižeg sloja.

METOD `flush()`: prilikom pisanja u baferovan izlazni tok podaci se čuvaju u *buffer*-u i zaista se zapisuju tek kada je *buffer* pun, zbog čega često ima smisla ispisati *buffer* u kritičnim momentima bez čekanja da se napuni – što se naziva *flush*-ovanje toka; ručno se *flush* obavlja metodom `flush()` – iako je metod deklarisan u `OutputStream` i `Writer` klasama nema nikakav efekat ukoliko je tok nebaferovan; neki tokovi takođe podržavaju automatsko *flush*-ovanje što se može postići opcionim argumentom u konstruktoru – kada je ova opcija omogućena određeni događaji (poput `\n` karaktera) izazvaće *flush*-ovanje *buffer*-a.

05 Klase `InputStream` i `OutputStream` – osobine, upotreba, funkcionalnosti nekoliko karakterističnih metoda, opisi nekoliko najbitnijih podklasa.

APSTRAKTNA KLASA `InputStream`: nadklasa svih klasa koje predstavljaju ulazni tok bajtova; najvažnije metode:

- `public abstract int read()`: čita sledeći bajt podataka (vrednost u opsegu 0-255) iz ulaznog toka; ukoliko se došlo do kraja fajla i nema više bajtova, vraća se vrednost -1;
- `public int read(byte[] b)`: čita određeni broj bajtova iz ulaznog toka u niz `b`; broj bajtova koji će se zaista pročitati se vraća kao `int`;
- `public int read(byte[] b, int off, int len)`: učitava do `len` bajtova podataka iz ulaznog toka u `b` počevši od pozicije `off`.

Sve navedene metode bacaju `IOException` ukoliko se desi neka I/O greška, a pored toga sve metode blokiraju dok ulazni tok ne postane dostupan, ne dođe se do kraja fajla, ili se ne baci izuzetak; kako bi se izbeglo blokiranje koristi se `public int available()` metod koji vraća

procenu broja bajtova koji se mogu pročitati (ili preskočiti) iz tekućeg ulaznog toka bez blokiranja sledećeg pozivanja metoda za tekući ulazni tok; metod `public long skip(long n)` se može koristiti za preskakanje određenog broja bajtova – povratna vrednost je stvarni preskočeni broj bajtova (iz mnoštva razloga može se desiti da preskoči veoma mali broj, pa čak i nijedan), a treba obratiti pažnju i da tok možda ne poseduje sposobnost traženja u kom slučaju se baca izuzetak.

ČVOR `InputStream` KLASA: `FileInputStream` – namenjen čitanju tokova sirovih bajtova (poput slika) iz fajla, `ByteArrayInputStream` – *wrapper* oko internog *buffer*-a koji sadrži bajtove koji se mogu čitati iz toka, `ObjectInputStream` – koristi se za čitanje objekata iz toka tokom procesa deserijalizacije, `AudioInputStream` – ulazni tok sa specifiranim audio formatom i dužinom (dužina je izražena u semplovima, ne milisekundama), kao i druge.

FILTERSKE `InputStream` KLASE: svi filterski ulazni tokovi nasleđuju `FilterInputStream` klasu (direktni potomak `InputStream`-a) i oni usput transformišu osnovne ulazne podatke; `BufferedInputStream` – dodaje mogućnost baferovanja ulaza, `DataInputStream` – dopušta aplikaciji čitanje primitivnih Java tipova iz toka koji se nalazi sloj niže na mašinski nezavistan način (`int`, `float`, `double`, itd.), `CheckedInputStream` – održava *checksum* podataka koji se čitaju koji se kasnije može koristiti za proveru integriteta ulaznih podataka.

APSTRAKTNA KLASA `OutputStream`: nadklasa svih klasa koje predstavljaju izlazni tok bajtova; najvažnije metode (sve bacaju `IOException` ukoliko se desi neka I/O greška):

- `public abstract void write(int b)`: piše prosleđen bajt u ovaj izlazni tok, odnosno piše prvih osam bitova argumenta, dok se ostalih 24 bita ignorišu;
- `public void write(byte[] b)`: piše `b.length` bajtova iz prosleđenog niza bajtova u ovaj izlazni tok;
- `public void write(byte[] b, int off, int len)`: piše `len` bajtova iz prosleđenog niza bajtova počevši od pozicije `off` u ovaj izlazni tok.

Čvor klasa i filterske `OutputStream` klase su pandani klasa zasnovanih na `InputStream`-u: `FileOutputStream`, `ByteArrayOutputStream`, `ObjectOutputStream` – za pisanje objekata u tok tokom procesa serijalizacije; primeri filterskih `OutputStream` klasa uključuju `BufferedOutputStream`, `DataOutputStream`, `CheckedOutputStream`.

06 Klase `Reader` i `Writer` – osobine, upotreba, funkcionalnosti nekoliko karakterističnih metoda, opisi nekoliko najbitnijih podklasa.

KLASE `Reader` I `Writer`: predstavljaju nadklase svih klasa koje obrađuju tokove karaktera, a koji za uzvrat automatski prevode unutrašnji format podataka sa i na lokalni ili specifirani set karaktera; za većinu aplikacija I/O tok karaktera nije složeniji od toka bajtova – štaviše, `Reader` klasa sadrži sve metode koje i `InputStream`, sa tom razlikom da se ovde koristi `char` umesto `byte` kao glavni tip (isto važi i za `Writer` i `OutputStream` klase); tokovi karaktera su zapravo često *wrapper*-i za tokove bajtova – tok karaktera koristi tok bajtova za fizičko izvršavanje I/O operacija, dok se tok karaktera koristi na gornjem sloju za prevod između karaktera i bajtova (parovi tokova bajtova i *wrapper*-a: `InputStream – Reader`, `OutputStream – Writer`, `FileInputStream – FileReader`, `FileOutputStream – FileWriter`, `ByteArrayInputStream – CharArrayReader`, `ByteArrayOutputStream – CharArrayWriter`); postoje dva opšta bajt-karakter „toka za premošćavanje“ – `InputStreamReader` i `OutputStreamWriter` (koriste se za

kreiranje novih tokova karaktera, tj. onda kada ne postoje postojeće klase za tokove karaktera). Metode:

`public int read()`: čita jedan karakter koji ujedno i vraća u vidu 16-bitne celobrojne *unsigned* vrednosti (ukoliko se došlo do kraja toka vraća -1);

`public int read(char[] cbuf)`: učitava karaktere u niz i vraća `int` vrednost;

`public void write(int c / String str)`: upisuje jedan karakter ili string;

`public void write(String str, int off, int len)`: upisuje string dužine `len` počevši od pozicije `off`;

`public int write(char[] cbuf, int off, int len)`: upisuje deo niza karaktera.

KLASA `PrintWriter`: najzgodniji način zapisivanja karaktera iz sledećih razloga: **1)** klasa je baferovana i podržava *auto-flushing*; **2)** nudi višestruke metode za pisanje različitih tipova podataka, linija, formatiranih linija; **3)** nudi nekoliko zgodnih konstruktora koji prihvataju `OutputStream`, `Writer` ili fajl; **4)** metodi ove klase ne bacaju izuzetke – umesto toga oni postavljaju unutrašnji *flag* koji se može preuzeti metodom `checkError()`; imati na umu da `PrintWriter` izvršava *auto-flushing* samo kada se pozovu `println()`, `printf()` ili `format()` metode.

OSTALE KLASE: `BufferedReader` – za baferovano čitanje znakova, `FilterReader` – apstraktna klasa za čitanje filtriranog niza znakova, `BufferedWriter` – za baferovano pisanje karaktera, nizova i stringova u izlazni tok, `FilterWriter` – apstraktna klasa za pisanje filtriranog niza znakova.

07 Motivacija i prednosti upotrebe generičkih tipova. Opšta sintaksa. *Type erasure*. *Generic sub-typing*. *Raw type* i rad sa *legacy* kôdom.

MOTIVACIJA: generički tipovi dozvoljavaju apstrakciju tipova – klase, interfejsi i metode se mogu parametrizovati tipovima što rezultuje *type-safe* kôdom (ako se kôd kompajlira bez grešaka ili upozorenja onda neće baciti ni *typecasting* izuzetak prilikom izvršavanja); generici takođe čine kôd čitljivijim (nakon navikavanja na sintaksu); najčešće se koriste kod kolekcija – primer `Stack` klase koja bi pre Jave 5 koristila `Object` tip što zahteva *typecasting* (iako programer zna koji tip podataka je smešten u *stack* kompajler ne zna, što može dovesti do *run-time* grešaka), dok kod rešenja koje koristi generike kolekcije po definiciji nisu vezane za tip elemenata.

OPŠTA SINTAKSA: definiše se navođenjem tipa parametra u „špicastim“ zagradama nakon naziva klase/interfejsa:

```
class Stack<E> {
  void push(E element) { ... }
  void pop() { ... }
  E top() { ... }  }
```

Prilikom instanciranja generičke klase tip parametra se zamenjuje argumentom tipa što proizvodi parametrizovan tip:

```
Stack<String> stack = new Stack<String>();
```

Po konvenciji parametar tipa navodi se kao jedno veliko slovo, i to: `E` - element (deo kolekcije); `K` - ključ; `N` - broj; `T` - tip; `V` - vrednost; `S`, `U`, `V`, itd. – drugi, treći, … tip. Pojmovi: **1)** generički tip / metod – klasa ili interfejs, ili metoda, sa jednim ili više parametara tipova (npr. `class Stack<E> { ... }`); **2)** parametrizovan tip – tip kreiran od generičkog tipa prosleđivanjem stvarnog argumenta tipa (npr. `Stack<String>`); **3)** parametar tipa – *placeholder* za argument

tipa (npr. `E`); **4)** argument tipa – referencijalni tip koji se koristi u generičkom tipu ili metodu (npr. `String`).

PREDNOSTI: u prethodnom primeru je `Stack` parametrizovan tipom `String` koji je prosleđen kao argument za parametar `E` – ovo se može shvatiti kao da je kompajler zamenio sve pojave `E` u definiciji klase ili interfejsa sa `String` (iako to zapravo nije slučaj); korišćenjem argumenta tipa kompajler može automatski da izvršava *typecasting*, a što je još važnije neće dozvoliti mešanje tipova što sprečava greške u vremenu izvršavanja.

***TYPE ERASURE*:** za razliku od *template*-a u C++, informacija o genericima postoji samo u vremenu kompajliranja – nakon što se kompajler osigura da je kôd *type-safe* informacija o genericima se uklanja iz *byte-code*-a, tj. ona ne postoji u vremenu izvršavanja:

```
LinkedList<Integer> a = new...
LinkedList<String> b = new...
```

```
if (a.getClass() == b.getClass())
  ... // true!
```

***SUB-TYPING*:** kao posledica nasleđivanja sledeće je dozvoljeno:

```
Object a = new Integer(10);
```

```
Object[] x = new String[100];
```

Deluje kao da bi i ovo trebalo da bude dozvoljeno:

```
Stack<Object> s = new Stack<Integer>();
```

```
// error!
```

Iako kontraintuitivno, problem je u tome što generici ne podržavaju nasleđivanje između argumenata tipova pa su i parametrizovani tipovi *invarijantni* – za bilo koja dva tipa `T1` i `T2` `List<T1>` niti je pod- niti nadtip od `List<T2>`; kada navedeno ne bi bilo slučaj kôd ne bi bio *type-safe* a to upravo i jeste razlog uvođenja generika – dakle, naredni kôd nije dozvoljen:

```
Stack<Integer> si = new Stack<Integer>();
Stack<Object> so = si; // kada bi ovo moglo...
```

```
so.push(new Object());
Integer n = si.pop(); // ...run-time exc.
```

Ipak, naredni primer je dozvoljen (programer ručno izvršava *typecasting*):

```
Stack<Object> s = new Stack<Object>();
s.push(new Integer(10)); // allowed
```

```
s.push(new Float(1.0));        // allowed
Integer n = (Integer) s.top(); // typecast
```

A takođe i ovo (nasleđivanje unutar jednog generičkog tipa):

```
List<Integer> list = new LinkedList<Integer>();
```

***RAW TYPE* I RAD SA *LEGACY* KÔDOM:** sirovi tip – generički tip koji je instanciran bez faktičkog argumenta tipa:

```
LinkedList list = new LinkedList<Integer>();
```

Ovo bi tehnički trebala biti greška budući da se gubi informacija o tipu i rezultujući kôd nije *type-safe*, međutim kompajler samo izbacuje upozorenje – ovo je namerna dizajnerska odluka kako bi se obezbedilo funkcionisanje sa postojećim *legacy* kôdom; sirove tipove bi trebalo koristiti samo kada se radi sa *legacy* kôdom, dok bi nov kôd uvek trebao da bude *type-safe* i da se oslanja na generike.

08

Džoker znak „?“ – primena u generičkim tipovima i posledice. Upotreba *bounded* džoker znaka. Generički metodi, sintaksa i primene.

DŽOKER „?“: razmotrimo slučaj pisanja metoda koji štampa sve elemente (bilo koje) kolekcije – bez generičkih tipova ovo bi bilo jednostavno:

```
void printAll(Collection coll) {
```

```
  for (Object o : coll) S.O.P.(o); }
```

Sa genericima deluje da bi funkcionisalo sledeće:

```
void printAll(Collection<Object> coll) {
```

```
  for (Object o : coll) S.O.P.(o); }
```

Ovo ipak nije tačno, budući da će funkcionisati samo sa kolekcijama objekata; rešenje je korišćenje argumenta nepoznatog tipa koji se označava džokerom „?“:

```
void printAll(Collection<?> coll) {
```

```
  for (Object o) S.O.P.(o); }
```

Gorenavedeni metoda prima kolekcije nepoznatog tipa, a njima se pristupa isključivo preko tipa `Object`; štaviše, nemoguće je dodati proizvoljne elemente budući da kompajler ne može da proveri tip, pa samim tim kôd ne bi bio *type-safe*.

***BOUNDED* DŽOKER ZNAK:** pogledajmo primer:

```
interface Shape { void draw(); }

class Circle implements Shape  {
  @Override public void draw() { ... } }

class Square implements Shape  {
  @Override public void draw() { ... } }
```

Treba napisati metoda koji iscrtava listu oblika:

```
void drawAll(List<Shape> shapes) {
  for (Shape s : shapes) s.draw();   }
```

Kako je objašnjeno ranije ovaj metoda se ne može pozvati za `List<Circle>` ili `List<Rectangle>` - regularni džoker u potpunosti gubi informaciju o tipu tako da je neophodno *typecast*-ovanje:

```
void drawAll(List<?> shapes) {
  for (int i = 0; i < shapes.size(); i++) {
    Shape s = (Shape) shapes.get(i);
    s.draw(); } }
```

Rešenje podrazumeva korišćenje <u>džokera sa gornjim ograničenjem</u>:

```
void drawAll(List<? extends Shape> shapes) {
  for (Shape s : shapes) s.draw(); }
```

Jezički konstrukt `<? extends Shape>` označava nepoznat tip koji nasleđuje klasu `Shape` (uključujući i sam `Shape`), što omogućava da se elementima kolekcije može pristupiti pomoću tipa `Shape` umesto isključivo pomoću tipa `Object` (fleksibilniji pristup) – proizvoljni elementi se ipak i dalje ne mogu dodati u kolekciju; analogni konstrukt bio bi `<? super Integer>`, što označava nepoznati tip koji je nadtip tipa `Integer` (uključujući i `Integer`) pa je samim tim ovo način da se zada donje ograničenje džokera – proizvoljni elementi se u ovom slučaju <u>mogu dodavati u kolekciju</u>; pogodnost gornje-ograničenih džokera sa druge strane je mogućnost njihovog navođenja prilikom deklaracije generičkih tipova: `class StackNum<E extends Number> { ... }`).

GENERIČKI METODI, SINTAKSA I PRIMENE: pored klasa i interfejsa i metodi se mogu parametrizovati – tip parametra kod generičkih metoda se navodi u definiciji metoda pre navođenja povratnog tipa; generički metodi se koriste za izražavanje zavisnosti među tipovima jednog ili više svojih argumenata i/ili povratnog tipa, a najčešća upotreba je modifikacija generičkih kolekcija:

```
private <T> void add(T[] a, List<T> l) {
  for (T elem : a) l.add(elem); } }
```

Kao što je ranije rečeno, kompajler se brine o tome da kôd bude *type-safe* pa nije moguće izmeniti kolekciju nepoznatog tipa čak ni generičkom metodom:

```
priv. <T> void add(T[] a, Lst<? ext. T> l) {
  for (T elem : a) l.add(elem); } }
```

Može se definisati i više generičkih tipova tipova u metodu.

09 Ograničenja generičkih tipova.

NEMOGUĆNOST KREIRANJA NIZOVA PARAMETRIZOVANIH TIPOVA: usled fundamentalne razlike između nizova i generika poput kovarijantnosti i invarijantnosti[1], nedozvoljeno je kreiranje niza parametrizovanih tipova (moguć je *workaround* korišćenjem sirovih tipova i *casting*-a):

```
// ukoliko bi ovo bilo dozvoljeno...
List<Str>[] str = new ArrayList<String>[1];

// ...kod ne bi bio type-safe...
List<Integer> ints = Arrays.asList(42);

Object[] objects = str;
objects[0] = ints;
String s = str[0].get(0);
```

[1] Is `f` covariant, contravariant, or invariant? Covariant would mean that a `List<String>` is a subtype of `List<Object>`, contravariant that a `List<Object>` is a subtype of `List<String>` and invariant that neither is a subtype of the other, i.e. `List<String>` and `List<Object>` are inconvertible types.

NEMOGUĆNOST KREIRANJA INSTANCI PARAMETARA TIPOVA: kreiranje instanci parametara tipova operatorom new je zabranjeno (moguć je *workaround* korišćenjem refleksije):

```
static <E> void append(List<E> list) {                E elem = new E(); list.add(elem); }
```

OGRANIČENJA *OVERLOAD*-A METODA: klasa ne može imati dva *overload*-ovana metoda koji imaju isto zaglavlje nakon *type erasure*-a:

```
class Example {
  void print(Set<String> strSet) { ... }              void print(Set<Integer> intSet) { ... } }
```

OSTALA OGRANIČENJA: generici se ne mogu instancirati sa primitivnim tipovima (primitivni tip se ne može koristiti kao argument tipa) – dozvoljeni su samo referencijalni tipovi; klase ne mogu imati statička polja čiji su tipovi parametri tipova (statička polja dele svi članovi klase); *cast*-ovanje i `instanceof` se ne mogu koristiti sa parametrizovanim tipovima (zbog *type erasure*-a); nedozvoljeno je koristiti parametrizovane tipove kao izuzetke (direktno ili indirektno nasleđivanje `Throwable` interfejsa je zabranjeno).

10

Lambda izrazi u Javi. Funkcionalno programiranje i njegove prednosti. Sintaksa lambda izraza, funkcionalni interfejsi, *variable capture*, reference na metode, `default` metodi.

LAMBDA U JAVI: reč je u suštini o metodu bez deklaracije, najčešće zapisanom na sledeći način:

```
(int x, y) -> { return x + y; }          x -> x * x                    () -> x
```

Lambda može imati nula ili više parametara razdvojenih zarezima i njihov tip se može eksplicitno deklarisati ili zaključiti iz konteksta (zagrade nisu neophodne kada figurira jedan parametar); sa `()` se označava nula parametara; telo može da sadrži nula ili više naredbi, a `{}` nisu neophodne kod tela sa jednom naredbom.

FUNKCIONALNO PROGRAMIRANJE: paradigma i stil programiranja koji izračunavanje vidi kao računanje matematičkih f-ja: **1)** eliminiše bočne efekte (*side effects*), **2)** tretira podatke kao *immutable*, **3)** izrazi sada imaju referencijalnu transparentnost, **4)** f-je mogu da primaju f-je kao argumente i da vraćaju f-je kao rezultate, **5)** preferiranje rekurzija umesto eksplicitnih petlji; čemu funkcionalno programiranje: **1)** pisanje programa koji su razumljiviji, deklarativniji i koncizniji nego što to omogućava imperativno programiranje; **2)** fokus je na problemu umesto na kôdu; **3)** oslobađanje *boilerplate*-a i *syntactic sugar*-a; **4)** paralelizacija je olakšana; **5)** omogućava pisanje čistijih API-ja; nova paradigma u Javu dolazi relativno kasno – sa verzijom 1.8 koja je najveća promena u jeziku od početka (ili možda verzije 1.5) – a njena najveća novina su lambde; Java je morala da nadoknadi zaostatak budući da je većina velikih jezika već podržavala lambda izraze, a veličina izazova bila je u tome što se dodatak morao izvesti tako da ne zahteva rekompilaciju postojećih biblioteka.

SINTAKSA LAMBDA IZRAZA: pogledajmo primere:

```
List<Integer> l = Arrays.asList(1, 2, 3);          List<Integer> l = Arrays.asList(1, 2, 3);
l.forEach(x -> S.O.P.(x));                         l.forEach(x -> { x += 2; S.O.P.(x); });
```

U prvom primeru lambda je `x -> S.O.P.(x)` koji definiše anonimnu f-ju sa jednim parametrom `x` tipa `Integer` (kompajler tip promenljive zaključuje iz konteksta); u drugom primeru `{}` su neophodne zbog više-linijskog tela u lambdi; još primera:

```
List<Integer> l = Arrays.asList(1, 2, 3);          List<Integer> l = Arrays.asList(1, 2, 3);
l.forEach(x -> { y += 2; S.O.P.(y); });            l.forEach(Int. x -> { x += 2; S.O.P.(x); });
```

U trećem primeru se definiše lokalna promenljiva u okviru lambde, dok se u četvrtom eksplicitno navodi tip parametra; detalji implementacije su sledeći: kompajler Jave 8 prvo lambdu pretvara u metod, a zatim se generisani metod poziva po potrebi – međutim, postavljaju se pitanja tipa koji bi se trebao generisati za metod, njegovog imena, kao i klase u koju bi ga trebalo smestiti...

FUNKCIONALNI INTERFEJSI: dizajnerskom odlukom su Java 8 lambde dodeljene funkcionalnim interfejsima, što su Java interfejsi sa tačno jednom ne-`default` metodom; naziv metoda nije važan – dovoljno je da se zaglavlje lambde i metoda poklope (u smislu broja i tipova parametara, povratne vrednosti, izuzetaka koji se bacaju ali ne i imena metoda); opciono imaju `@FunctionalInterface` anotaciju. Svojstva generisanih metoda: **1)** metodi generisani iz Java 8 lambda izraza imaju isto zaglavlje kao i metoda funkcionalnog interfejsa, **2)** tip je isti kao tip funkcionalnog interfejsa kojem je lambda izraz dodeljen, a **3)** telo lambda izraza postaje telo metoda u interfejsu. Paket `java.util.function` između ostalog sadrži:

- `Predicate<T>`: predstavlja `boolean`-vrednosnu f-ju sa jednim argumentom „`boolean test(T t)`“ koja vraća `true` ako ulazni argument odgovara predikatu;
- `Supplier<T>`: „dobavljač“ rezultata, funkcionalni metod je „`T get()`“ (vraća rezultat tipa `T`);
- `Consumer<T>`: operacija koja prima jedan input i ne vraća rezultat, metod je „`void accept(T t)`“ i izvršava operaciju na prosleđenom argumentu tipa `T`;
- `Function<T,R>`: f-ja koja prima jedan argument i proizvodi rezultat, metod „`R apply(T t)`“ koji primenjuje funkciju na tip `T` i vraća rezultat tipa `R`;
- `UnaryOperator<T>`: nasleđuje Function i predstavlja operaciju nad jednim operandom koja vraća rezultat istog tipa, metod je „`T apply(T t)`“;
- `Comparator<T>`: poredi dva argumenta, metod je „`int compare(T o1, T o2)`“;
- `Runnable`: za definisanje niti, funkcionalni metod je „`void run()`“.

***VARIABLE CAPTURE*:** lambda izraz se može dodeliti promenljivoj tipa funkcionalnog interfejsa ako su zaglavlje lambda izraza i funkcionalnog metoda kompatibilni – kompajler će „popuniti praznine“ ako tipovi u lambdi nisu specifirani:

```
Consumer<Integer> p = x -> System.out.println(x);
```

Lambde mogu da interahuju sa promenljivima definisanim van tela lambde – korišćenje ovakvih promenljivih naziva se *variable capture*:

```
int var = 10;                                   intList.forEach(x -> S.O.P.(x + var));
```

Promenljive (uključujući parametre metoda) moraju biti (efektivno) `final`.

REFERENCE NA METODE: od Jave 8 reference na metode se mogu koristiti za prosleđivanje postojeće f-je na mestu gde se očekuje lambda (tj. funkcionalni interfejs), pri čemu zaglavlje referenciranog metoda mora da odgovara zaglavlju metoda u interfejsu; konzisnost kod referenci na metode:

```
intList.forEach(x -> S.O.P.(x));                intList.forEach(System.out::println);
```

***DEFAULT* METODE:** sa lambdama pojavila se potreba značajnog proširenja standardne biblioteke, ipak uvođenje novih metoda bi slomilo postojeći kôd budući da bi se morale dodati implementacije novih metoda – glavni razlog uvođenja *default* metoda u Javi 8.

11 Java *stream* API. Dobavljanje *stream*-ova, operacije nad *stream*-ovima, struktura *stream*-ova, životni ciklus. Klasa `Optional`, primeri. Paralelno programiranje sa *stream*-ovima.

POJAM TOKA: tokovi podataka su dizajn obrasci u funkcionalnom stilu programiranja (monade), odnosno tok je sekvenca objekata koja podržava različite metode koje se mogu spojiti u *pipeline* kako bi se dobio željeni rezultat (nije reč o strukturi podataka); operacije nad tokovima vrednosti u funkcionalnom stilu pruža novi `java.util.stream` paket; tokovi se mogu dobiti iz praktično bilo kog smislenog izvora u Java APIju poput individualnih vrednosti (`Stream.of(v1, v2, ...)`), opsega (`IntStream.range(n1, n2)`), nizova (`Arrays.stream(arr)`), kolekcija (`list.stream()`), stringova (`string.chars()`, tok karaktera), fajlova

(`Files.line(path)`, tok linija fajla u vidu stringova); sekvencijalni su ili paralelni, a korisni su za selekciju vrednosti i izvršavanjem akcija nad rezultatima, naročito u slučaju kombinovanja različitih selekcija i akcija.

STRUKTURA I ŽIVOTNI CIKLUS: komponente: **1)** izvor, **2)** *pipeline* od najmanje nula međuoperacija i **3)** terminalna operacija; životni ciklus: **1)** kreiranje – tok iz izvora, **2)** konfiguracija – kroz sekvencu (*pipeline*) međuoperacija, **3)** izvršavanje – pozivanjem terminalne operacije, **4)** čišćenje – kada se jednom izvrši tok je „potrošen" i odbačen (ponovo pokretanje zahteva kreiranje novog).

OPERACIJE NAD TOKOVIMA: tipovi: **1)** međuoperacije (*intermediate*) ostavljaju tok otvorenim za dalje operacije a izračunavaju se „lenjo", **2)** terminalne operacije su konačni korak obrade toka i nakon njih tok je potrošen i dalje se ne koristi; obratiti pažnju da se terminalne operacije navode poslednje ali izvršavaju prve, budući da njihovo pokretanje okida izvršenje međuoperacija i *input* elemenata iz izvora; takođe, operacije nad tokovima ne modifikuju izvor. **Tipovi terminalnih operacija**: **1)** redukcije (vraćaju jedinstvenu vrednost): `count`, `min`, `max`, `reduce`; **2)** *mutable* redukcije (vraćaju višestruk rezultat u vidu kolekcije): `collect` (sa mnoštvom metoda u `Collectors` klasi) `toArray`; **3)** pretrage (vraćaju rezultat čim se nađe poklapanje): `findFirst`, `findAny`, `anyMatch`, `allMatch`; **4)** generičke terminalne operacije (bilo koja vrsta obrade nad elementima toka): `forEach`. **Tipovi međuoperacija**: **1)** *stateless* operacije – ne moraju da znaju išta o rezultatima prethodnih koraka u sekvenci (`filter`, `map`, `mapToInt`, `-Long`, `-Double`, `peek`), **2)** stateful operacije – moraju znati rezultate izvršavanja prethodnih operacija (`distinct`, `limit`, `skip`, `sorted`).

KLASA `Optional`: paket `java.util` u Javi 8 dobija nove klase među kojima je i `Optional<T>` - kontejner koji može sadržati ne-`null` vrednost tipa `T`; `OptionalInt`, `-Long`, `-Double` – varijante za primitivne tipove; česte metode: `boolean isPresent()`, `T get()`, `T orElse (T other)`, `ifPresent(Consumer)` – pokreće *consumer* f-ju na vrednosti ako je prisutna.

PARALELNO PROGRAMIRANJE SA TOKOVIMA: tokovi se mogu izvršavati u sekvencijalnom ili paralelnom režimu, pri čemu se paralelni tok dobija pozivanjem `.parallelStream()` umesto `.stream()` – sve ostalo ostaje isto, paralelizam se postiže automatski.

12 Osobine višenitnih programa. Procesi i niti. Načini kreiranja i pokretanja niti u Javi. Stanja niti, uspavljivanje i interapti. Metod `join()`.

KONKURENTNOST: za softver koji je u stanju da izvršava više operacija istovremeno se kaže da je „konkurentan" (npr. aplikacija skida fajlove, pušta muziku i ažurira displej istovremeno); Java platforma dizajnirana je od temelja da podrži konkurentno programiranje, oslanjajući se pre svega na osnovnu podršku konkurentnosti koja je prisutna u samom jeziku i Java bibliotekama.

PROCES: uopšteno govoreći, proces je instanca programa sa samostalnim okruženjem za izvršavanje i koji ima kompletan, privatan skup osnovnih *run-time* resursa (naročito, svaki proces ima sopstveni memorijski prostor); većina savremenih OS-ova nudi podršku za međuprocesnu komunikaciju (*IPC*) što dozvoljava procesima, koji čak ne moraju biti na istoj mašini, da razmenjuju informacije.

NITI: reč je o *lightweight* procesma koji dele resurse (poput memorije) i otvorene fajlove; svaki proces sastoji se od bar jedne niti koja se često naziva *main thread* – u slučaju Jave, za svaku aplikaciju postoji i dodatna, sistemska nit koja se bavi upravljanjem memorijom,

procesiranjem signala, i slično; u poređenju sa procesima: **1)** niti zahtevaju manje resursa da se kreiraju i izvrše, **2)** među-nitna komunikacija je mnogo brža i bez dodatnog *overhead*-a (*any combination of excess or indirect computation time, memory, bandwidth, or other resources that are required to perform a specific task*), **3)** dok s druge strane deljena memorija zahteva posebno rukovanje kako bi se izbegli problemi pri izvršavanju.

U Javi postoji dva načina definisanja niti: **1)** implementacijom `Runnable` interfejsa, ili **2)** nasleđivanjem `Thread` klase koja zauzvrat obezbeđuje praznu implementaciju `Runnable` interfejsa; drugi način je lakši za upotrebu u jednostavnim aplikacijama ali je ograničen činjenicom da klasa mora naslediti `Thread`; u svim slučajevima kôd koji se izvršava u novoj niti je sadržan u `public void run()` metodu, a pošto je `Runnable` funkcionalni interfejs lambda izrazi se mogu upotrebiti za definisanje `run()` metoda.

KREIRANJE I POKRETANJE NITI: nit se kreira instanciranjem objekta klase koja nasleđuje `Thread`, dok se `Runnable` objekat mora proslediti konstruktoru od `Thread` – stvarno izvršavanje niti počinje tek nakon pozivanja `start()` metoda:

```
class MyThread1 implements Runnable {
  @Override public void run() {
    S.O.P.("Implementing Runnable"); } }
class MyThread2 extends Thread {
  @Override public void run() {
    S.O.P.("Extending Thread"); } }
public class SimpleThreads {
  public static void main(String[] args) {
    new Thread(new MyThread1()).start();
    new MyThread2().start(); } }
```

Nikada se direktno ne poziva `run()` – `start()` izvršava dodatne korake pre pozivanja `run()`; takođe, nije legalno startovati nit više od jednom – naročito, nit se može restartovati nakon završetka izvršavanja.

STANJA NITI: nit može biti u sledećim stanjima: **1)** *Ready* – nit je kreirana ali još nije izvršena (njen `start()` još nije pozvan), **2)** *Running* – nit izvršava `run()` metod, **3)** *Suspended* – izvršavanje je pauzirano, **4)** *Blocked* – nit čeka da se određeni uslovi ispune, **5)** *Terminated* – nit je ugašena nakon što `run()` završi sa radom.

USPAVLJIVANJE: pozivanje statičke `sleep()` metode klase `Thread` suspenduje se izvršavanje trenutne niti (metod prima vreme u milisekundama); ovaj pristup se često koristi za obezbeđenje procesorskog vremena za druge niti aplikacije ili druge aplikacije operativnog sistema; uvek kada nit izvršava petlju sa mnogo iteracija dobra je praksa staviti sleep() u svaku iteraciju (čak i kratko vreme od 5 milisekundi smanjuje korišćenje CPU-a sa 100% na < 1%):

```
class ThreadNoSleep extends Thread {
  @Overrid public void run() {
    while (true) ; } }
class ThreadSleep extends Thread {
  @Override public void run() {
    while (true)
      try { Thread.sleep(1); }
      catch (InterruptedException e) {
        ... } } }
```

***INTERRUPT*-I:** reč je o indikaciji niti da bi trebala da prekine da radi šta god da radi i da uradi nešto drugo – na programeru je da odluči kako će tačno nit da odgovori na *interrupt*, s tim da je gašenje niti veoma često rešenje; podrška niti za *interrupt*-e zavisi od toga šta nit radi: **1)** ako nit često poziva metode koji bacaju `InterruptedException` (poput `sleep()` metoda) ona prosto hvata taj izuzetak, **2)** inače nit periodično poziva `Thread.interrupted()` metod koji vraća `true` ako je primljen *interrupt*; mehanizam *interrupt*-a implementiran je korišćenjem internog *flag*-a zvanog *interrupt status*: kada nit proverava *interrupt*-e prilikom pozivanja statičkog `Thread.interrupted()` metoda ovaj status se raščisti, dok ne-statički metod `isInterrupted()` ne menja status *flag*-a:

```
class IntThread1 extends Thread {
  @Override public void run() {
    while (true) {
      // some data processing...
      try {
        Thread.sleep(5);
      } catch (InterruptedExc e) {
        return; // terminate } } } }
class IntThread2 extends Thread {
  @Override public void run() {
    while (true) {
      // some data processing...
```

```
      if (Thread.interrupted())
        return; // terminate } } }
Thread t1 = new IntThread1(); t1.start();
Thread t2 = new IntThread2(); t2.start();
// let the threads execute for some time
Thread.sleep(10000);
// terminate threads
t1.interrupt(); t2.interrupt();
```

METOD `join()`: dozvoljava jednoj niti da sačeka završetak druge:

```
class OutputProducer extends Thread {
  private String filename;
  private boolean ok;
  public OutputProducer(Str file) {
    this.filename = file; }
  public String getFilename { ... }
  public boolean isOk { retrn ok; }
  @Override public void run() {
    try {
      FileWriter f = new FileW(file);
      f.append("Hello"); f.close();
      ok = true;
    } catch (IOException e) { ... } } }
```

```
class InputConsumer extends Thread {
  private OutputProducer output;
  public InputCons(OutputProd out) {
    this.output = output; }
  @Override public void run() {
    try {
      output.join(); // wait to finish
      if (output.isOk()) {
        BuffRead f = new BuffRead(
          new InputStreamR(
          new
FileInpStream(output.getFile())));
        S.O.P.(f.readLine()); f.close();
      } catch (Exc e) { ... } } }
```

13 *Thread context switch* i *thread interference*. Osnovni načini sinhronizacije u Javi i njihova poređenja.

***THREAD CONTEXT SWITCH*:** paralelno višenitno izvršavanje na *CPU* sa jednim jezgrom nije zaista paralelno već *CPU* izvršava par instrukcija jedne niti a zatim se prebacuje na sledeću; po *default*-u Java niti se izvršavaju na više jezgara ako postoje, ali ako broj niti prevazilazi broj jezgara doći će do prebacivanja; proces čuvanja stanja izvršavanja jedne niti i vraćanje stanja druge se naziva *thread context switch*, što daje utisak simultanog izvršavanja niti iako zapravo to nije slučaj.

Podsetimo se dve stvari: **1)** svaki Java izraz ili naredba se, bez obzira na prividnu jednostavnost, sastoje od nekoliko pod-koraka – npr. i++ se sastoji od čitanja trenutne vrednosti i iz memorije, inkrementacije i čuvanja inkrementirane vrednosti promenljive; **2)** *context switch* se može desiti u bilo kojoj tački između pod-koraka i nikako se ne može predvideti kada će se to desiti – u prethodnom primeru *context switch* se može desiti posle bilo kog od tri pod-koraka (sledi primer sa `Counter` i `CounterThread` sa predavanja, sa različitim rezultatima nakon pokušaja inkrementacije brojača).

INTERFERENCIJA NITI: ukoliko nekoliko niti radi nad istim podacima može doći do interferencije niti – kao što se vidi iz prethodnog scenarija, dolazi do interferencije niti T1 sa instrukcijama za inkrementaciju brojača niti T0, što daje netačan rezultat; kako bi se izbeglo ovakvo ponašanje nit mora „zaštititi" svoje izvršavanje dok radi na deljenim podacima – drugim rečima, niti koje operišu nad istim podacima moraju sinhronizovati svoje izvršavanje kako bi se sprečila interferencija niti i greške u konzistenciji memorije; postoje situacije gde sinhronizacija nije neophodna – npr. čitanje promenljive koja je proglašena za final je uvek bezbedno.

SINHRONIZACIJA: osnovne tehnike uključuju: **1)** metode za sinhronizaciju, **2)** naredbe za sinhronizaciju, **3)** *lock*-ove; pored ovih, Java biblioteke pružaju naprednija rešenja za sinhronizaciju (poput semafora).

METODE ZA SINHRONIZACIJU: `synchronized` metode pružaju jednostavnu strategiju za sprečavanje interferencije niti i grešaka u konzistenciji memorije; nemoguće je pomešati dva poziva `synchronized` metoda nad istim objektom – kada jedna nit izvršava `synchronized` metod objekta, sve ostale niti koje pozivaju bilo koji (ne nužno isti) `synchronized` metod za

isti objekat moraju da zaustave izvršavanje (blokirane su) dok prva nit ne završi svoje; kada postoji `synchronized` metod garantovano je da će sve promene stanja objekta biti vidljive svim nitima, tako da ako je objekat vidljiv za više od jedne niti sve čitanje i pisanje njegovih promenljivih bi trebalo da se radi kroz `synchronized` metode (primer – dodati `synchronized` na sve metode u primeru sa `Counter` i `CounterThread` klasama).

***INTRINSIC LOCKS*:** sinhronizacija je izgrađena oko unutrašnjeg entiteta koji se zove *intrinsic lock* (unutrašnja brava) ili *monitor lock* (ili prosto *monitor*); svaki objekat poseduje *intrinsic lock* koji je povezan sa njim – *lock* obezbeđuje ekskluzivan pristup stanju objekta; kada nit pozove `synchronized` metod ona automatski stiče *intrinsic lock* za objekat tog metoda i otpušta je kada se metod izvrši – otpuštanje *lock*-a se dešava čak i ako se metod završi zbog neuhvaćenog izuzetka; kada nit pokuša da preuzme *lock* koji već poseduje druga nit biće blokirana dok se *lock* ne oslobodi. Iako nit ne može da preuzme *lock* koji poseduje druga nit ona može da preuzme *lock* koji već poseduje; na ovaj način se omogućava *reentrant synchronization* (sinhronizacija pri ponovnom ulasku), bez čega bi sinhronizovani kôd morao da preduzme još mnogo dodatnih predostrožnosti kako bi se izbeglo da nit izazove da se sama blokira (npr. pri pisanju rekurzivnih sinhronizovanih metoda).

NAREDBE ZA SINHRONIZACIJU: umesto čitavih metoda mogu se zaštiti samo pojedine naredbe korišćenjem ključne reči `synchronized`; ovakve naredbe se koriste za optimizaciju ili *fine-tuning* sinhronizacionog procesa kako bi se izbeglo suvišno blokiranje; za razliku od sinhronizovanih metoda naredbe moraju specifirati objekat za koji se pruža *intrinsic lock*.

14 Osobina *liveness* i tri posledice loše sinhronizacije. Opis problema filozofa i diskusija o rešenjima.

***LIVENESS PROPERTY*:** sposobnost konkurentne aplikacije da se izvrši blagovremeno se naziva *liveness* (životnost); loš konkurentni dizajn može dovesti do jednog od sledećih problema: *deadlock* – dve ili više niti su blokirane zauvek (tj. svaka čeka na onu drugu da oslobodi *lock*); *starvation* (gladovanje) – nit ne može da dobije regularan pristup deljenim resursima i usled toga ne može da napreduje (što je uzrokovano drugom, „pohlepnom“ niti); *livelock* – dve ili više niti ne mogu da napreduju zato što akcije svake niti zavise od akcija neke druge niti (slično pešacima koji idu jedni drugima u susret trudeći se da se međusobno izbegnu); osiguranje *liveness*-a konkurentne aplikacije je često najteži zadatak sinhronizacije niti.

STUDIJA SLUČAJA – PROBLEM FILOZOFA: problem filozofa na večeri se koristi da ilustruje česte probleme u razvoju konkurentnih aplikacija: **1)** pet filozofa sedi za okruglim stolom i ispred svakog je tanjir špageta; **2)** po jedna viljuška se nalazi između susednih filozofa i na taj način svaki ima po jednu sa leve i sa desne strane; **3)** filozofu su potrebne dve viljuske za jelo, a može da koristi samo one koje su neposredno levo i desno od njega; **5)** svaki filozof ili razmišlja ili jede – ako razmišlja ne jede, i obrnuto; **6)** filozofi ne razgovaraju međusobno.

REŠENJE #1: najjednostavnije rešenje bi bilo:

```
while (true) {                  takeRightFork();                putRightFork(); }
  think();                      eat();
  takeLeftFork();               putLeftFork();
```

Iako pravolinijsko, rešenje može dovesti do *deadlock*-a – ako npr. svaki filozof uzme samo desnu ili samo levu viljušku svaki od njih će čekati da se oslobodi druga kako bi je uzeo a to se neće nikad desiti.

REŠENJE #2: poboljšanje – ako je uzeo levu viljušku a desna je zauzeta, spusti levu:

```
while (true) {                  think();                        while (true) {
```

```
    takeLeftFork();                putLeftFork();              putLeftFork();
    if (tryTakeRgtFork())          wait(delta); }              putRightFork(); }
      break;                     eat();
```

I dalje je moguće da „nesrećan“ filozof ostane gladan – ukoliko svi filozofi sednu za sto u isto vreme i čekaju jednako vremena pre nego što ponovo pokušaju da uzmu viljuške, svi će biti savršeno sinhronizovani i radiće to u isto vreme pa će se tako doći do *livelock*-a.

REŠENJE #3: uvođenje „konobara“ – objekta koji vodi računa o tome da mušterije ne gladuju tako što broji koliko je puta svaki filozof želeo da jede ali nije mogao da dođe do viljuške; nakon što broj pokušaja pređe određeni prag filozofu se dodeljuje veći prioritet – uvodi se novo pravilo koje kaže da filozof ne može da uzme viljuške ako njegov sused gladuje; pregled klase filozofa:

```
public enum State {
  THINKING, HUNGRY, EATING }

public class Phlsph impl Runnable {
  priv State state;
  priv int id;
  priv Waiter waiter;
  priv stat fin Rnd rnd = new Rnd();

  public Philosopher(int id, Waiter w) {
    this.id = id;
    this.waiter = w;
    state = State.THINKING;
    new Thread(this).start();          }

  @Override public void run() {
    try { while (true) {
      think();
      while (!waiter.takeForks(id))
        Thread.sleep(rndTime(1k, 2k));
      eat();
      waiter.putForks(id); }
    catch (InterruptExc e) { } } ... }
```

Pregled klase konobara:

```
public class Waiter {

  priv stat fin int NUM_PH = 5;
  priv stat fin int MAX_WAIT = 10;
  priv Phil[] phil;
  priv int[] waitCount;
  priv Set<Integer> starving;

  public Waiter() {
    waitCount = new int[NUM_PH];
    starving = new HashSet<Integer>();
    phil = new Philosopher[NUM_PH];
    for (int i = 0; i < NUM_PH; i++)
      phil[i] = new Phil(i, this); }

  public synch bool takeForks(int id) {
    phil[id].setState(State.HUNGRY);
    int left = (id + NUM_PH – 1) % NUM_PH;
    int right = (id + 1) % NUM_PH;

    // id can eat only if neighbors are not
    b canEat = lf != St.EAT && rg != St.EAT;

    // forks are free, id can eat if it is
    // starving or no neighbor is starving
    if (canEat) {
      phil[id].setState(St.EAT);
      waitCount[id] = 0;
      starving.remove(id);
      return true; }

    // id has to try later
    if (++waitCount[id] > MAX_WAIT)
      starving.add(id);

    return false;  } ... }
```

15 Osnove *JFC, AWT, Swing* biblioteka, odnosno *JavaFX* biblioteke. Osnovni kontejneri, struktura aplikacije i okviri (*panes*). [Bira se jedna od lekcija: *Swing* ili *JavaFX*.]

***JAVA FOUNDATION CLASSES*:** reč je o programerskom *framework*-u koji se koristi za razvoj aplikacija sa GUI-jem u Javi koji obuhvata grupu *feature*-a za kreiranje grafičkih korisničkih interfejsa: **1)** *Abstract Window Toolkit* (*AWT*): originalni, zavistan od platforme komplet alata za razvoj GUI-ja; **2)** *Swing*: primarni, nezavistan od platforme komplet GUI alata; **3)** Java 2D API: omogućava laku inkorporaciju 2D grafike, teksta i slika u aplikacije i aplete.

KOMPONENTE GUI-JA: reč je o objektu koji ima grafičku reprezentaciju koja se može iscrtati na ekranu i sa kojom korisnik može da vrši interakciju (dugmad, labele, *checkbox*-ovi, i slično); Java GUI komponenta može biti teška (*heavyweight*, vezana je za sopstveni *native* prozor) i laka (*lightweight*, „pozajmljuje“ resurs na ekranu od „pretka“); sve *AWT* komponente su teške, a sve *Swing* komponente su lake (izuzev kontejnera na najvišem nivou).

***ABSTRACT WINDOW TOOLKIT* I *SWING*:** *AWT* je zavistan od platforme – njegove teške komponente najčešće su *wrapper*-i oko *native* komponenti, što znači da je *AWT* dosta brz ali: **1)** komponente koriste više resursa od *Swing* parnjaka, **2)** stvaran broj ponuđenih komponenti je relativno mali budući da uključuje samo podskup onih koje su dostupne na svim ciljanim

platformama; *Swing* je nezavistan od platforme, napisan od nule u Javi: **1)** samo kontejneri na najvišem nivou alociraju *native* resurse, **2)** osigurava da aplikacije na raznim platformama imaju isti izgled, **3)** može se koristiti sa *AWT*-om, **4)** kasnije su postignuta značajna poboljšanja u opštoj brzini i *native* izgledu i osećaju.

***SWING* I *JAVAFX*:** *JavaFX* je započeta kao samostalan API, kasnije je integrisan u JDK 8-10 ali je od JDK 11 ponovo samostalan i *open-source*; obe biblioteke se baziraju na sličnim konceptima (pod različitim imenima): **1)** komponente – čvorovi, **2)** kontejneri/paneli – okna, **3)** *layout*-i (drugačije rukovanje: sadržavanje naprotiv nasleđivanja), **4)** programiranje zasnovano na događajima; *JavaFX* cilja na širi spektar uređaja i podržava specijalne efekte, animacije; *MVC pattern* je lakše postići u *JavaFX* i uopšte *JavaFX* ima čistiji dizajn; *Swing* ima podršku ogromnog broja resursa, takođe mnoge postojeće aplikacije nemaju razlog da napuste *Swing*.

KONTEJNERI NAJVIŠEG NIVOA: da bi se pojavila na ekranu GUI komponenta mora pripadati kontejneru i biti deo hijerarhije sadržavanja – stabla komponenti čiji je koren *top-level* kontejner; *Swing* pruža tri korisne klase *top-level* kontejnera: **1)** *JFrame* – predstavlja prozore sa naslovom i granicama, **2)** *JApplet* – predstavlja aplete, poseban tip Java programa koji se skidaju sa interneta i izvršavaju u *browser*-ima koji ih podržavaju, **3)** *JDialog* – predstavlja dijaloge, npr. za prikaz grešaka ili upozorenja korisniku; program koji koristi *Swing* komponente poseduje bar jedan *top-level* kontejner kao koren hijerarhije sadržavanja (npr. *standalone* aplikacija koja prikazuje dijalog ima dve hijerarhije sadržavanja – jedna sa *JFrame*-om u korenu i druga sa *JDialog*-om u korenu); GUI komponenta se može sadržavati samo jednom – ako je komponenta već u kontejneru i pokuša se dodavanje u drugi, ona će se ukloniti iz prvog i dodati u drugi.

OKNA (*PANES*): svaki *top-level* kontejner sadrži hijerarhijski organizovane slojeve (okna) sa različitim namenama: **1)** *root pane* – koren hijerarhije, sadrži i kontroliše ostala okna; **2)** *layered pane* – kontejner sa dubinom tako da se preklapajuće komponente mogu postaviti jedne povrh drugih, a takođe sadrži i *menu bar* ako je ima; **3)** *content pane* – okno koje bi po pravilu trebalo da sadrži komponente nevezane za meni; **4)** *glass pane* – nalazi se na vrhu i po *default*-u je transparentno, koristi se za hvatanje ulaznih poruka (poput klikova), za crtanje po komponentama, i slično.

16 Programiranje zasnovano na događajima. *Action* i *mouse listener*-i. Načini definisanja *listener*-a. [Bira se jedna od lekcija: *Swing* ili *JavaFX*.]

PROGRAMIRANJE ZASNOVANO NA DOGAĐAJIMA: reč je o paradigmi u kojoj je tok programa određen događajima poput akcija korisnika (klikovi, pritisak tastera), poruka drugih programa, senzornih signala i slično; u Javi se prepoznaju tri važne komponente događaja: **1)** izvor – komponenta koja generiše događaj (dugme, miš, tastatura, itd.), **2)** *listener* (*handler*) – objekat koji prima vesti o događaju i procesira ih, **3)** objekat – sadrži sve neophodne informacije o događaju (tip, izvor, i slično). Neodređeno mnogo *listener* objekata može da čeka na događaje od neodređeno mnogo izvora; uopšteno, model događaja u Javi prati sledeći algoritam: **1)** *listener* se registruje sa izvorom – nakon registracije on čeka na događaj, **2)** nakon što se desi događaj *event* objekat se kreira i „ispaljuje“ ka registrovanim *listener*-ima, **3)** nakon što *listener* primi *event* objekat od izvora on vadi iz njega potrebne informacije i procesira događaj. *Listener*-i se registruju kod izvora pozivom metode `void addTypeListener(TypeListener listenerObject)`, gde `Type` zavisi od izvora (*Key*, *Mouse*, i slično); *listener* se može od-registrovati pozivom `void removeTypeListener(TypeListener`

`listenerObject`); u pozivima metoda `listenerObject` predstavlja stvaran *listener*, odnosno objekat koji implementira odgovarajući interfejs (`ActionListener`, `MouseListener`, i slično), gde interfejsi sadrže metode koji se pozivaju kada se desi događaj.

***EVENT* OBJEKTI:** svaki *event-listener* metod prima jedan argument – objekat koji nasleđuje `EventObject` klasu; iako argument uvek potiče od `EventObject` klase, njegov tip je uopšte preciznije specifiran (npr. argument za metode koje rukuju *event*-ima miša su instance `MouseEvent`); `EventObject` klasa definiše jedan koristan metod – `Object getSource()`, koji vraća objekat koji je „ispalio“ *event*.

***ACTION LISTENER*:** *action event* se dešava svaki put kada korisnik izvrši akciju (npr. kada korisnik klikne na dugme, odabere stavku iz menija, pritisne *Enter* na tekstualnom polju); `ActionListener` interfejs:

- `void actionPerformed(ActionEvent e);`

`ActionEvent` klasa:

- `String getActionCommand()` – string vezan za akciju o kojoj je reč, većina objekata koji „ispaljuju“ *action event*-ove podržavaju metod `setActionCommand()` koji dozvoljava programeru da postavi ovaj string;
- `int getModifiers()` – vraća *integer* koji predstavlja modifikujuće tipke koje je korisnik držao kada se *action event* desio (sledeće konstante koje definiše `ActionEvent` se mogu koristiti da se odredi koje tipke su pritisnute: `SHIFT_`, `CTRL_`, `META_` i `ALT_MASK`).

***MOUSE LISTENER*:** *mouse event*-ovi notifikuju kada korisnik koristi miša (ili sličan ulazni uređaj) kako bi vršio interakciju sa komponentom, i događaju se kada kurzor uđe u ili izađe iz ekranske oblasti komponente, i kada korisnik pritisne ili pusti jedno od dugmadi miša; `MouseListener` interfejs:

- `mouseClicked(MouseEvent)` – korisnik je kliknuo komponentu;
- `mouseEntered/Exited(MouseEvent)` – korisnik je ušao u / izašao iz komponente;
- `mousePressed/Released(MouseEvent)` – pritisnuto / pušteno dugme nad komponentnom.

`MouseEvent` klasa (parametar svih *mouse listener* metoda), najčešće metode:

- `int getClickCount()` – broj uzastopnih klikova (vraća `2` za dupli klik);
- `int getButton()` – vraća koje dugme miša je, ako je ijedno, promenilo stanje (`NOBUTTON`, `BUTTON`, 2, 3);
- `int getX / Y(), Point getPoint()` – vraća poziciju kurzora.

***ADAPTER* KLASE:** neki *listener* interfejsi (poput `MouseListener`) sadrže više od jednog metoda, što je problem ako je aplikacija zainteresovana samo za klikove budući da ona svejedno mora da implementira svih pet metoda interfejsa; *Swing* nudi *adapter* klase za svaki *listener* interfejs koje sadrže prazne verzije svih metoda interfejsa, tako da se mogu implementirati samo neophodne (*default* metode u interfejsima su uvedene tek u Javi 8 a *Swing* je stariji).

UNUTRAŠNJE KLASE: ipak, nasleđivanje *adapter* klase nije uvek moguće ni poželjno, budući da prozor koji treba da reaguje na klikove ne može da nasledi i `JFrame` i `MouseAdapter` – unutrašnje klase su rešenje ovog problema:

```
public class MyClass extends JFrame {
  class MyAdapter ext MouseAdapter {
    public void mouseClicked(MouseEv e) {
      // event listener impl... } }
  public MyClass() {
    obj.addMouseLsnr(new MyAdapter()); } }
```

ANONIMNE UNUTRAŠNJE KLASE: unutrašnja klasa definisana bez imena, može učiniti kôd čitljivijim budući da se klaša definiše tamo gde se i referencira; unutrašnja klasa može da referencira polja i metode instance klase koja je sadrži kao da je reč o njenom kôdu (sve dok metode nisu statičke) – za direktnu referencu koristi se `EnclosingClass.this`.

OSTALI *MOUSE LISTENER*-I: praćenje pokreta kurzora zahteva više resursa nego praćenje ostalih događaja miša pa se ova vrsta izdvaja u `MouseMotionListener` interfejs, a `MouseInputListener` objedinjuje `MouseListener` i `MouseMotionListener`; pored toga, `MouseWheelListener` se koristi za događaje sa točkićem miša; na kraju, postoje i `MouseAdapter` (sva tri osnovna *mouse listener*-a) i `MouseInputAdapter` (implementacija metoda `MouseInputListener`-a).

OSTALI *LISTENER*-I: *Swing* nudi i brojne druge, poput: **1)** *key listener*-i – rukovanje ključnim događajima koje „ispaljuju" komponente sa fokusom tastature kada korisnik pritisne ili pusti tipku; **2)** *component listener*-i – pozivaju se kada se promeni veličina, položaj ili vidljivost komponente; **3)** *window listener*-i – bave se aktivnostima ili stanjima prozora (otvaranje, zatvaranje, itd.); **4)** *item listener*-i – dostupni kod komponenti koje održavaju *on/off* stanja (*check box*-ovi, *radio button*-i, i slično).

17 Raspoređivanje komponenti. Apsolutno pozicioniranje. Opisi standardnih raspoređivača (*layouts*). [Bira se jedna od lekcija: *Swing* ili *JavaFX*.]

RASPOREĐIVANJE KOMPONENTI: komponente se u kontejner dodaju preko njegovog `add()` metoda – veličina, pozicija, poravnanje komponente kontroliše *layout manager*; komponente mogu da daju „nagoveštaj" veličine i poravnanja, ali *manager* ima poslednju reč kada se radi o veličini i poziciji komponente u kontejneru; dostupni su: *Border-*, *Flow-*, *Grid-*, *Box-*, *Card-*, *GridBag-* i *SpringLayout*. Kontejnerov *layout manager* se postavlja kao argument `setLayout()` metoda, gde `null` forsira apsolutno pozicioniranje pri kojem programer mora ručno da specifira poziciju i veličinu svake komponente; kad god je moguće treba koristiti *layout manager* jer bez njega komponente ne reaguju dobro na različite veličine fontova, promenljivu veličinu kontejnera i različita lokalna podešavanja.

***BORDER LAYOUT*:** *default layout manager* za *JFrame* i *JApplet*, deli prostor na pet regiona: `N`, `S`, `W`, `E`, `C`; pri dodavanju komponente drugi argument `add()` metoda kontejnera je ciljni region (*default* je `CENTER`), a sama komponenta se razvlači na čitav region (region sadrži ne više od jedne komponente).

***FLOW LAYOUT*:** *default manager* za *JPanel*, slaže komponente u red pri njihovoj preferiranoj veličini; ako je horizontalni prostor kontejnera premali ovaj *layout* prelazi u nov red; po *default*-u red komponenata je vertikalno centriran unutar kontejnera, što se može promeniti `void setAlignment(int align)` metodom ili konstruktorom `FlowLayout(int align)` (parametar za poravnanje može biti `FlowLayout.LEFT`, `.CENTER` i `.RIGHT`); po *default*-u horizontalni i vertikalni prostor između komponenti je 5px.

***GRID LAYOUT*:** postavlja komponente u ćelije mreže, gde svaka komponenta zauzima sav dostupan prostor unutar ćelije i svaka ćelija je iste veličine (ćelije su maksimalne veličine uzevši u obzir dostupan prostor u kontejneru); broj redova i kolona se određuje konstruktorom – jedan (ali ne oba) broj može biti nula, što znači da bilo koji broj objekata može biti postavljen u red ili kolonu.

***BOX LAYOUT*:** dozvoljava postavljanje više komponenti ili u vertikali (odozgo naniže), ili horizontali (sleva nadesno), što se specifira u konstruktoru (`X_AXIS` ili `Y_AXIS`, takođe i `LINE_AXIS` – horizontalno s orijentacijom određenom `ComponentOrientation` svojstvom kontejnera, i `PAGE_AXIS` – vertikalno s orijentacijom određenom istim svojstvom); po *default*-u komponente se slažu jedna do druge bez razmaka koji se mogu napraviti korišćenjem nevidljivih komponenti: **1)** *rigid area* – fiksiran razmak između dve komponente, **2)** *glue* –

kontroliše kuda ide višak prostora (npr. horizontalni *glue* između dve komponente u horizontalnom *box*-u će ubaciti dodatni prostor između ovih komponenti, umesto s desne strane svih komponenti).

***CARD LAYOUT*:** rukuje dvema ili više komponentama koje dele isti ekranski prostor; konceptualno, svaka komponenta kojom *layout* rukovodi je poput karte u špilu, pri čemu je vidljiva samo najviša karta u bilo kom trenutku; raspored *card*-ova je određen sopstvenim unutrašnjim uređenjem objekata komponenata kontejnera – obično je prva dodata komponenta na vrhu, a poslednja dodata na dnu. *Card*-ovima se može upravljati metodama za prebacivanje na određeni *card*: `first(Container parent)`, `next(Container parent)`, `previous(Container parent)`, `last(Container parent)` i `show(Container parent, String name)`; pri korišćenju ovog *layout*-a komponente se dodaju u kontejner metodom `void add(Component c, Object constraints)`, gde `constraints` predstavlja jedinstveno stringovno ime komponente.

***GRID BAG LAYOUT*:** moćnija varijanta *grid layout*-a, dozvoljava ćelije različite veličine, jedna komponenta može da se proteže kroz više ćelija.

***SPRING LAYOUT*:** veoma fleksibilan *manager* (dodat u JDK 1.4) koji emulira mnoge *feature*-e drugih *manager*-a; ipak, mogu biti veoma komplikovani za ručno kodiranje i bolje je koristiti ih sa GUI *builder*-om.

01 Osnovni pojmovi računarskih mreža, IP adrese, portovi, *DNS*. Slojevi mrežnih protokola.

OSNOVE: mreža je kolekcija kompjutera i drugih uređaja koji mogu da primaju/šalju podatke jedni drugima, manje-više u realnom vremenu; čvor je bilo koja mašina na mreži, a čvor koji je potpuno funkcionalni kompjuter najčešće se zove *host*; sve savremene mreže funkcionišu po sistemu prebacivanja paketa (*packet-switching*): podaci koji putuju kroz mrežu se razbijaju na komade zvane paketi i svakim od njih se rukuje zasebno; svaki čvor u mreži ima dodeljenu adresu i koristi portove za razmenu podataka sa drugim čvorovima.

INTERNET ADRESE: svaki uređaj koji se povezuje na mrežu (uključujući i internet) se može jedinstveno identifikovati preko mrežne adrese (odnosno internet adrese – IP adrese); trenutno postoji dva IP standarda: **1)** IPv4 (koristi 4 bajta, npr. `192.168.0.1`) i **2)** IPv6 (koristi 16 bajtova), a JDK pruža transparentnu podršku za oba standarda; zarad bolje čitljivosti mrežnih adresa često im se pridružuju tzv. *host* imena (npr. `dmi.rs`); *Domain Name System* (*DNS*) – softverski sistem namenjen prevođenju *host* imena u numeričke IP adrese; svaki računar povezan na internet ima pristup *domain name server*-u koji izvršava DNS softver: **1)** kada se zatraži konekcija (npr. na `dmi.rs`) računar kontaktira DNS server, **2)** server potraži u bazi prosleđeno ime i vrati numeričku IP adresu – ova procedura se naziva rezolucija imena.

PORTOVI: reč je o virtuelnoj *data* konekciji koju koriste programi za direktnu razmenu podataka umesto da koriste fajlove ili slično privremeno skladište podataka; omogućavaju *host*-u da izvrši više različitih stvari odjednom – preuzimanje *web* stranica, slanje *e-mail*-ova ili *upload* fajlova na FTP server; svaki računar u mreži ima hiljade logičkih portova, od kojih svaki može biti namenjen posebnom vidu mrežne komunikacije; portovi su uopšte čista apstrakcija u računarskoj memoriji i ne predstavljaju nijedan fizički deo računara, a predstavljeni su 16-bitnom numeričkom vrednošću: brojevi 1 do 1023 su rezervisani za

poznate servise poput HTTP – 80, FTP – 21, SMTP – 25, IMAP – 143, Telnet – 23, i slično, dok su brojevi 1024 do 65535 slobodni za korišćenje od strane korisničkih aplikacija.

PROTOKOLI: reč je o preciznom skupu pravila koji određuju kako računari komuniciraju, i to na način definisanja formata i redosleda poruka koje se razmenjuju između dva ili više entiteta u komunikaciji, kao i akcija koje se moraju preduzeti po slanju i/ili primanju poruke; kako bi se redukovala složenost dizajna dizajneri mreža organizuju protokole u slojeve tako da svaki sloj obuhvati određen broj logički grupisanih protokola; *Internet Protocol Suite* ima pet slojeva: **1)** aplikacioni (*application*) – zadužen je za podršku mrežnih aplikacija, uključuje protokole poput HTTP, SMTP za poštu, FTP za transfer fajlova; **2)** transportni (*transport*) – zadužen je za transport poruka od aplikativnog sloja do ciljnog mrežnog čvora, glavni protokoli koje obuhvata su TCP i UDP; **3)** mrežni (*network*) – zadužen za usmeravanje paketa informacija od startnog do ciljnog čvora, ponekad kroz više posrednika, glavni protokol je IP (TCP/IP kombinacija transportnog i mrežnog protokola je osnova interneta); **4)** vezni (*link*) – mrežni sloj zavisi od sloja veze kako bi prebacio pakete informacija između dva susedna (direktno povezana) čvora, u LAN mrežama *Ethernet* je najrasprostranjeniji protokol ovog sloja; **5)** fizički (*physical*) – dok je posao veznog sloja da prebacuje čitave pakete posao fizičkog sloja je da pomera individualne bitove podataka unutar paketa od jednog čvora do drugog (poređani od najvišeg ka najnižem prema nivou apstrakcije).

TCP: *Transmission Control Protocol* je fokusiran na uspostavljanje pouzdanih mrežnih konekcija, prati svaki otpravljeni paket i ukoliko se isti izgubi TCP će ga ponovo poslati, a takođe se brine o tome da se paketi prime u istom redosledu u kom su i poslati; TCP rukovodi svim implementacionim detaljima koji se tiču ponovnog slanja i ređanja paketa, i obaveštava programera samo u ozbiljnim slučajevima (npr. kada je konekcija prekinuta); ipak, TCP je uopšte sporiji i zahteva više resursa od UDP-a, pa se koristi u slučajevima kada je potrebna pouzdana transmisija podataka.

UDP: *User Datagram Protocol* je bezkonekcioni protokol za transport podataka – bajtovi podataka se grupišu zajedno u diskretne pakete koji se zatim šalju preko mreže nezavisno jedni od drugih; UDP ne garantuje dostavu svakog paketa ili da će paketi doći u nekom određenom redosledu – izvorni ne može da utvrdi da li je otpravljen paket primljen od strane ciljnog čvora; ipak, budući da ne postoji potreba za uspostavljanjem i održavanjem konekcije i ne postoji provera grešaka i oporavak, UDP je uopšteno govoreći brži pa se zato koristi kod aplikacija u realnom vremenu koje zahtevaju brzu dostavu (poput *audio/video streaming* aplikacija).

02 Predstavljanje IP adresa u Javi. *URI* – opšta sintaksa i kategorije. Klasa `URLConnection`.

KLASA `InetAddress`: reprezentacija IP adrese ili *host* imena visokog nivoa, od Jave 4 radi i sa IPv4 i v6 (`Inet4Address` i `Inet6Address` klase); kao i sve mrežne klase `InetAddress` je u `java.net` paketu; ne postoje `public` konstruktori za ove klase – objekti se kreiraju pozivom statičkih metoda:

- `InetAddress getByName(String host)` / `InetAddress[] getAllByName(String host)`
- `InetAddress getByAddress(byte[] address)` – prima sirovu IP adresu od 4 (IPv4) ili 16 bajtova (IPv6)
- `InetAddress getByAddress(String host, byte[] address)` – prima i ime *host*-a i IP adresu

Ako se prosledi ime *hosta* metodima `getByName()` ili `getAllByName()` IP adresa će biti ustanovljena kontaktiranjem DNS servera – posledice ovog pristupa su da može potrajati dok se objekat ne kreira (iako će Java izvršiti proveru imena); ako je DNS server nedostupan bacиće se izuzetak, a ako je tekstualna IP adresa prosleđena nekom od ovih metoda proverava se samo validnost formata adrese; kôd u nastavku meri vreme za koje se dobijaju informacije od *DNS*-a:

```
public class Host2IP {
 pub st void main(Str[] args) {
  long start = Sys.currentTimeMmilis();
  try {
   out.println("IP: " +
    InetAddress.getLocalHost().getHostAdd();
   InAdd[] ad = InetAdd.getAllByNm(args[0]);
   for (InetAdd a : ad) out.println("" + a);
  } catch { (UnknHstExc e) { }
  } finally { S.O.P.("Total time.."); }
```

URI-JI: *Uniform Resource Identifier* je string karaktera koji identifikuje resurs (fajl na serveru, email adresu, poruku, knjigu) čija je sintaksa oblika `scheme:scheme_part`, gde je primer *scheme* `http`, `file`, itd.; dve kategorije: *Universal Resource Names* (*URN*), *Universal Resource Locators* (*URL*); u Javi je predstavljen `java.net.URI` klasom.

URN-OVI: ime određenog resursa, ne pruža referencu na određenu lokaciju ili metod za pribavljanje istog; cilj URN-a je rukovanje resursima koji su *mirror*-ovani na više različitih lokacija ili koji su premešteni sa jednog mesta na drugo; opšti oblik – `urn:namespace:resource_name` (primer knjige: `urn:ISBN:1565924851`); u Javi ne postoji specijalizovana klasa koja predstavlja URN-ove.

URL-OVI: mehanizam za specifiranje lokacije gde je identifikovan resurs dostupan i mehanizma za pribavljanje istog; sintaksa – `scheme://user:pass@domain:port/path?params#fragment`; Java koristi java.net.URL klasu za reprezentaciju URL-ova i nudi šest konstruktora: **1)** `URL(String specification)`, **2)** `URL(String protocol, int port, String file)`, **3)** `URL(URL base, String path)`, i drugi; svi konstruktori bacaju izuzetak ako se ne može izgraditi validan URL (nedostajuća ili nepodržana specifikacija protokola, takođe pri proveri konstrukcije URL-a Java samo proverava prepoznavanje sheme ali ne i da li je URL pravilno formiran); klasa je *immutable*.

KLASA `URLConnection`: apstraktna klasa koja predstavlja aktivnu konekciju sa resursom specificiranim URL-om, pruža detaljnu kontrolu nad interakcijom sa serverom (naročito HTTP serverom) – npr. može da proveri *header* koji šalje server i prikladno odgovori, preuzme binarne fajlove i vrati podatke web serveru sa `POST` ili `PUT` i da koristi druge HTTP *request* metode; podklase `URLConnection` se koriste za implementaciju rukovanja određenim protokolom (npr. `ftp`). Instanca `URLConnection` klase se može dobiti pozivom openConnection() metode nad postojećom instancom URL-a; nakon otvaranja `URLConnection` objekat je nekonektovan – aplikacija može da zatraži konekciju pozivom `connect()` metoda, a ako se pozove metod koji zahteva aktivnu konekciju `connect()` će biti pozvan automatski; opciono, aplikacija može konfigurisati konekciju, čitati *header* polja, pribaviti ulazni tok i čitati podatke ili izlazni tok i pisati podatke. Klasa `URLConnection` otkriva i sledeće metode za pribavljanje ulaznih i izlaznih tokova konekcije: `getInput/OuputStream()` – podaci se čitaju sa i pišu se na server korišćenjem uobičajenog APIja za tokove; budući da HTTP serveri često pružaju značajnu količinu informacija u *header*-u koji prethodi svakom odgovoru, `URLConnection` klasa uključuje i `get` metode za čitanje uobičajenih *header* polja, poput `content-type`, `content-length`, `date`, `last-modified`, itd.

03

TCP komunikacija, prednosti i mane u odnosu na UDP. Klasa `Socket` – kreiranje, upotreba, komunikacija (čitanje i pisanje podataka). Klasa `ServerSocket` – kreiranje, upotreba, prihvatanje više dolaznih konekcija.

TCP SOCKET-I: reč je o konekciji između dva *host*-a, komunikacionom kanalu koji omogućava transfer podataka preko određenog porta; *socket*-i čine osnovu većeg dela Java mrežnog programiranja – Java izvršava svu svoju mrežnu komunikaciju niskog nivoa kroz *socket*-e; osnovne operacije uključuju **1)** konekciju na udaljenu mašinu, **2)** slanje i primanje podataka, **3)** zatvaranje konekcije, **4)** vezivanje za port i prihvatanje ulaznih konekcija, **5)** osluškivanje dolaznih podataka.

PREDNOSTI I MANE: TCP rukovodi svim implementacionim detaljima koji se tiču ponovnog slanja i ređanja paketa, i obaveštava programera samo u ozbiljnim slučajevima (npr. kada je konekcija prekinuta); ipak, TCP je uopšte sporiji i zahteva više resursa od UDP-a, pa se koristi u slučajevima kada je potrebna pouzdana transmisija podataka.

KLASA `Socket`: fundamentalna Java klasa za klijentsku stranu TCP mrežnog programiranja; sama klasa koristi *native* kôd za komunikaciju sa lokalnim TCP stekom operativnog sistema *host*-a; druge klase koje kreiraju mrežne konekcije (poput `URLConnection`) sve na kraju završe na pozivima metoda ove klase; nakon uspostavljanja konekcije kroz instancu klase `Socket`, stvarno čitanje i pisanje podataka preko *socket*-a se vrši preko klasa tokova. Konstruktori `Socket` klase dopuštaju specifiranje *host*-a (u vidu `InetAdress` objekta ili stringa) i porta (u vidu broja od `0` do `65535`) za koji se želi konekcija; neki konstruktori takođe specifiraju lokalnu adresu i lokalni port sa kog će se slati podaci, što je opcija koja se koristi kada se želi jedan određeni mrežni interfejs od kog se šalju podaci; postoji i `ctor` bez parametara koji kreira nekonektovan *socket*, što je korisno kada aplikacija mora da podesi opcije *socket*-a pre konekcije (primeri: skeniranje otvorenih portova određenog *host*-a, Echo klijent).

KLASA `SocketAddress`: primarna svrha ove klase je pružanje zgodnog skladišta informacija za *socket* konekcije (IP adresa i portova) koje se mogu ponovo koristiti za kreiranje novih *socket*-a; reč je o praznoj apstraktnoj klasi bez metoda osim *default* konstruktora – teoretski, predviđena je za nasleđivanje sa specifičnom, zavisnom od protokola implementacijom, ali u praksi samo TCP/IP je trenutno podržan kroz `InetSocketAddress` klasu.

KLASA `ServerSocket`: nasleđuje klasu `Socket` i pruža funkcionalnost neophodnu za serversku stranu aplikacije; u opštem slučaju, osnovni životni ciklus serverske aplikacije je: **1)** novi `ServerSocket` se kreira na određenom portu, **2)** server osluškuje dolazeće pokušaje konekcija na tom portu koristeći `accept()` metod, **3)** u zavisnosti od servera ulazni i/ili izlazni tokovi se pribavljaju, **4)** server i klijent vrše interakciju, **5)** server/klijent/oba zatvaraju konekciju, **6)** server se vraća na korak #2 i čeka sledeću konekciju. Četiri konstruktora za kreiranje *server socket*-a:

- `ServerSocket(int port)` – kreira *server socket* vezan za određeni lokalni port;
- `ServerSocket(int port, int queue)` – isto kao prethodni, sa *queue*-om određene dužine
- `ServerSocket(int port, int queue, InetAddress addr)` – vezan za lokalnu IP adresu
- `ServerSocket()` – nevezan *server socket*, `bind()` metod se poziva pre prihvatanja konekcija

Server obično radi u petlji koja konstantno prihvata konekcije – pri svakom prolasku kroz petlju poziva se `accept()` metod koji vraća `Socket` objekat koji predstavlja konekciju između udaljenog klijenta i lokalnog servera (interakcija sa klijentom se odvija kroz taj `Socket` objekat); kad se transakcija završi server poziva `close()` metod `Socket` objekta; ako klijent zatvori konekciju dok server još radi ulazno-izlazni tokovi između servera i klijenta bacaju

izuzetak pri sledećem pisanju ili čitanju; budući da poziv `accept()` metoda obično blokira pozivaoca dok se klijent ne konektuje, serveri se obično izvršavaju na odvojenim nitima.

***MULTIPLEXING*:** pristup kreiranja nove niti za svaku novu konekciju lepo radi na jednostavnim serverima i klijentima koji nemaju ekstremne potrebe za performansama; ipak, *overhead* u slučaju postojanja posebne niti za svaku konekciju postaje netrivijalan na velikim serverima koji mogu procesirati i hiljade zahteva u sekundi – mnogo brži pristup je korišćenje jedne niti koja upravlja višestrukim konekcijama tako što uzima onu koja je spremna, rukuje njome najbrže što može, a onda prelazi na sledeću; reč je o *multiplexing*-u koji je podržan od strane većine savremenih OS-ova, a zasnovano je na NIO konceptima poput kanala i *buffer*-a.

04 UDP komunikacija, prednosti i mane u odnosu na TCP. Pojam UDP *datagram*-a i diskusija o veličini UDP paketa. Komunikacija preko UDP-a, klase `DatagramPacket` i `DatagramSocket`.

PREDNOSTI I MANE: UDP ne garantuje dostavu svakog paketa ili da će paketi doći u nekom određenom redosledu – izvorni ne može da utvrdi da li je otpravljen paket primljen od strane ciljnog čvora; ipak, budući da ne postoji potreba za uspostavljanjem i održavanjem konekcije i ne postoji provera grešaka i oporavak, UDP je uopšteno govoreći brži pa se zato koristi kod aplikacija u realnom vremenu koje zahtevaju brzu dostavu (poput *audio/video streaming* aplikacija).

UDP DATAGRAMI: nezavisne i samostalne poruke (paketi) koji se šalju preko mreže i čiji se dolazak, vreme dolaska i sadržaj ne garantuju; Java nudi tri klase koje se bave datagramima.

KLASA `DatagramPacket`: UDP datagram predstavljen je instancom ove klase koja nudi funkcionalnost za pakovanje sirovih bajtova podataka u izlazni datagram, kao i ekstrakciju podataka iz primljenih datagrama; pored toga, za izlazne datagrame adresa je uključena u sam paket; datagram se kreira za primanje ili za slanje – izlazni datagram se razlikuje uključivanjem ciljne adrese i porta; u oba slučaja, instance `DatagramPacket` klase predstavljaju *wrapper*-e oko nizova bajtova koji sadrže stvarne podatke; dostupni konstruktori:

- `DatagramPacket(byte[] buf, int length)`: konstruiše datagram za primanje paketa dužine `length`, počevši od `buf[0]`;
- `DatagramPacket(byte[] buf, int offset, int length)`: početak od `buf[offset]`.

Konstruktori izlaznih datagrama su slični, s dodatkom parametra za ciljnu adresu.

VELIČINA DATAGRAMA: ako je primljeni datagram veći od za njega određenog objekta, mreža odseca višak ili odbacuje ulaz (Java program se ne obaveštava o problemu); iako je teorijski maksimum podataka u datagramu `65,507B`, u praksi je uvek reč o manjoj količini; na mnogim platformama stvarni limit je bliži `8kB` iako implementacije nisu obavezne da primaju datagrame veće od `512B` – za maksimalnu sigurnost, deo sa podacima UDP paketa bi trebao da bude `512B` ili manji, iako ovaj limit može negativno da utiče na performanse.

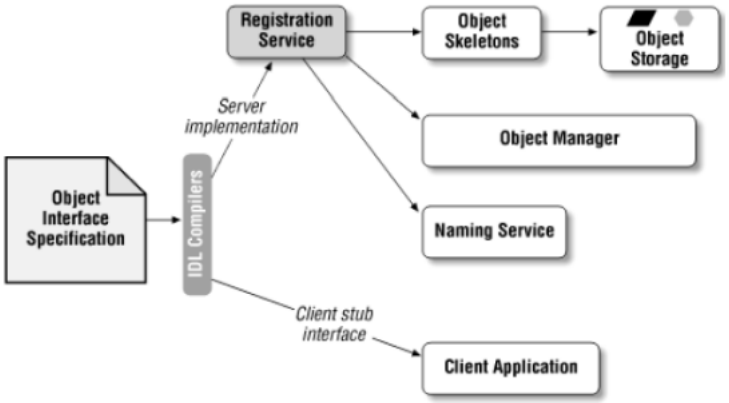
KLASA `DatagramSocket`: kako bi se zaista slali ili primali datagrami aplikacija mora otvoriti *socket* – ovo se postiže preko `DatagramSocket` klase koja nudi nekoliko konstruktora za vezivanje datagram *socket*-a za lokalni port i/ili adresu (ciljna adresa i port su sadržani u samom datagramu). Nakon konstrukcije `DatagramPacket` i `-Socket` objekata aplikacija može da šalje pakete prosleđivanjem istog *socket*-ovom `send()` metodu; slično, metod `receive()` prima jedan UDP datagram sa mreže i čuva ga u prethodno napravljen `DatagramPacket` objekat (poput `accept()` metoda kod `ServerSocket` klase ovaj metod blokira pozivajuću nit

dok datagram ne stigne); ukoliko dođe do problema u primanju ili slanju podataka može se baciti `IOException` (u praksi je ovo retko jer je UDP po prirodi nepouzdan).

05 Osobine i primene distribuiranog programiranja. Osnovni elementi distribuiranih aplikacija. Koncept serijalizacije objekata. Serijalizacija atributa i ključna reč `transient`. Kontrola verzija klase u procesu serijalizacije.

OSOBINE I PRIMENE DISTRIBUIRANOG PROGRAMIRANJA: reč je o procesu razbijanja aplikacije na pojedinačne objekte izračunavanja koji mogu biti raštrkani širom mreže računara ali da i dalje rade zajedno na kooperativnim zadacima; glavna prednost distribuiranog izračunavanja obuhvata: **1)** paralelizam – korišćenje manjih, jeftinijih računara za rešavanje velikih problema umesto upošljavanja velikih mašina; **2)** snižen *bandwith* mreže – veliki skupovi podataka se tipično teško relociraju i lakše ih je kontrolisati i administrirati tamo gde se nalaze, tako da udaljeni dana serveri mogu da se koriste za pribavljanje potrebnih informacija; **3)** redundansa – objekti izračunavanja na višestrukim računarima u mreži se mogu koristiti od strane sistema koji zahtevaju toleranciju grešaka, odnosno ako mašina padne posao se može nastaviti.

SISTEMI DISTRIBUIRANIH OBJEKATA: predstavljaju skup API-ja koji skrivaju kompleksnost distribuiranog programiranja i koji dozvoljavaju programerima da pozivaju objekte na udaljenim *host*-ovima i vrše interakciju sa njima kao da je reč o objektima na lokalnom *host*-u; opšta arhitektura sistema distribuiranih objekata:



Osobine DOS-a: **1)** specifikacija interfejsa objekta – pruža način (jezik) za specifiranje interfejsa objekata nezavisno od implementacionih detalja; **2)** *stub*-ovi – odgovorni za otpravljanje udaljenih zahteva, *stub*-ovi se koriste za usmeravanje lokalnih poziva metoda ka objektima na serveru; **3)** kosturi – odgovorni za procesiranje udaljenih zahteva, koriste ih serveri za kreiranje novih instanci udaljenih objekata i usmeravanje udaljenih poziva metoda ka implementaciji objekata; **4)** *naming service* – asocira udaljen objekat sa imenom koji klijent može da iskoristi kako bi dobio referencu na objekat; **5)** *object manager* – srce svakog DOS-a, rukovodi kosturima i referencama objekata na serveru objekata (obično takođe podržava naprednije *feature*-e, poput perzistencije objekata); **6)** registracioni servis – registruje novo-implementirane klase kod *naming service*-a i *object manager*-a, i zatim skladišti klasu na serveru; **7)** *object communication protocol* – podržava načine za slanje i primanje referenci na objekte, metode i samih podataka (u idealnom slučaju klijentska aplikacija ne zna detalje ovog protokola).

SERIJALIZACIJA: reč je o procesu pretvaranja skupa instanci objekata koji sadrže referencu jednih na druge u linearni tok bajtova, što je mehanizam koji koristi RMI kako bi prosleđivao objekte između JVM-ova; serijalizacioni proces objekta **1)** čuva ime klase i signaturu klase, **2)**

vrednosti svih ne-statičkih polja objekta, **3)** zatvaranje bilo kog objekta koji se referencira od strane inicijalnih objekata; obrnut proces – kreiranje skupa objekata od toka podataka – se naziva deserijalizacija. Najčešći primeri upotrebe: **1)** perzistencija – korišćenjem `FileOutputStream` tok objekta se može automatski zapisati u fajl; **2)** mehanizam kopiranja – korišćenjem `ByteArrayOutputStream` tok objekta se zapisuje u niz bajtova u memoriji, koji se zatim može koristiti za duplikaciju originalnog objekta; **3)** komunikacija – korišćenjem toka koji dolazi iz *socket*-a objekti se mogu automatski slati „preko žice“ ka primajućem *socket*-u. Kako bi objekat mogao da se serijalizuje klasa mora implementirati `Serializable` interfejs – reč je o *marker* interfejsu koji ne sadrži ništa i prosto označava klasu kao serijalizabilnu; `NonSerializableException` se baca kada se pokuša serijalizacija nad objektima nad kojima to nije moguće; serijalizabilnost se nasleđuje – potrebno je implementirati je samo jednom u hijerarhiji klasa (većina Java klasa su serijalizabilne).

KLJUČNA REČ `transient`: da bi se objekat serijalizovao sva njegova polja moraju biti serijalizabilna – serijalizacija ne brine o modifikatorima pristupa poput `private`; `transient` se koristi za obeležavanje polja klase koja se ne mogu serijalizovati, pa se time zapravo sprečava serijalizacija podataka; tokom deserijalizacije ovakva polja se inicijalizuju na *default* vrednosti (`0`, `null`, itd.); `transient` se može koristiti i sa serijalizabilnim poljima radi poboljšanja performansi.

KONTROLA VERZIJE KLASE: svim serijalizabilnim klasama se automatski dodeljuje identifikator verzije koji se čuva zajedno sa objektom i automatski se ažurira svaki put kada se klasa objekta promeni (npr. kada se doda novo polje); identifikatori verzija se porede tokom deserijalizacije – ako se verzija klase razlikuje od verzije objekta u toku baca se izuzetak; da bi se kontrolisao sistem verzioniranja potrebno je prosto dodati statičko `serialVersionUID` polje ručno i obezbediti da je uvek isto, osim ako dođe do promena nad klasom koje poništavaju prethodno serijalizovane objekte.

06 Koncept Reflection API-ja i primene. Implementacija mobilnih agenata.

REFLEKSIJA: reč je o procesu tokom kog softver može da posmatra i modifikuje sopstvenu programsku strukturu i ponašanje u vremenu izvršavanja; dozvoljava inspekciju klasa i njihovih elemenata, kao i instanciranje novih objekata i pozive metoda u vremenu izvršavajna bez znanja o stvarnim imenima u vremenu kompajliranja; u Javi pristup ovim metapodacima o objektima dostupni su kroz *immutable* instancu klase `java.lang.Class`; Javin *Reflection* API dostupan u `java.lang.reflect` paketu se najčešće koristi za: **1)** serijalizaciju i RMI, **2)** *browser*-e klasa i okruženja za vizuelni razvoj, **3)** *debugger*-e i alate za testiranje koji, npr. trebaju da pristupaju `private` svojstvima klasa. Postoji nekoliko načina dobavljanja `Class` instance: **1)** ako je instanca objekta dostupna najjednostavniji način je pozivanje njene `getClass()` metode; **2)** ako je tip dostupan ali nema instance, onda je moguće dobaviti klasu dodavanjem `.class` na ime tipa (funkcioniše i sa primitivnim tipovima); **3)** ako je dostupno puno kvalifikovano ime klase, moguće je dobaviti klasu korišćenjem statičkog metoda `Class.forName()`; za primitivne tipove (poput `int`) koristi se `int.class` ili `Integer.TYPE`. Postoje dve kategorije metoda dostupnih u `Class` za pristup poljima, metodima i konstruktorima: metodi koji nabrajaju sve članove i metodi koji traže određenog člana; pored toga, postoje odvojene metode za pristup članovima deklarisanim direktno u klasi i onim

nasleđenim od nadinterfejsa i nadklasa; konačno, neke metode `Class` mogu da traže samo `public` članove, dok druge mogu da pristupe i `private` i `protected` članovima.

STUDIJA SLUČAJA – MOBILNI AGENTI: tehnologija agenata predstavlja jedan od najkonzistentnijih pristupa distribuiranom razvoju softvera; ona upošljava distribuiranu mrežu autonomnih, izvršivih softverskih entiteta koji se nazivaju agenti; agent se smatra mobilnim ako se može kretati od jednog računara od mreži do drugog; uobičajena upotreba mobilnih agenata je u procesiranju velikih skupova podataka: dobavljanje velikog skupa podataka iz udaljene baze može biti preskupo ili čak nemoguće, dok je mnogo bolji pristup slanja (obično malog) agenta na ciljni računar kako bi izvršio procesiranje na mestu. Svaki računar koji želi da prihvati mobilne agente *host*-uje agentski server – server prima serijalizovanu formu mobilnog agenta, a zatim ga deserijalizuje i pokreće; kako bi se server koliko je god moguće uopštio, *Reflection* API se koristi za izvršavanje agenta (server će potražiti i pozvati metod `void onArrival(ServerSocket host)`); dodati kôd implementacije agenata…

07 RMI – registri i servisi. Definisanje funkcionalnosti servisa, implementacija, registracija i pronalaženje. RMI klijenti.

RMI: *Remote Method Invocation* je Java *core* API i biblioteka klasa koja dozvoljava Java programima koji se izvršavaju na jednoj JVM pozivanje metoda objekata koji se izvršavaju na drugoj, moguće fizički udaljenoj, JVM – RMI kreira privid da se konkretan distribuirani program izvršava na jednom sistemu sa jednim memorijskim prostorom; reč je o čistom Java pristupu distribuiranom razvoju softvera; RMI se oslanja na napredne Java *feature*-e poput serijalizacije objekata, refleksije i dinamičkog učitavanja klasa; dostupan u paketu `java.rmi`.

RMI SERVISI: reč je o udaljenom objektu sa metodima koji se mogu pozvati iz druge JVM, odnosno ne iz one u kojoj se sam objekat nalazi; svaki RMI servis implementira `Remote` interfejs i specifira koje metode se mogu pozivati od strane klijenata; iz perspektive programera, RMI servisi funkcionišu u suštini kao lokalni objekti – odnosno, klijenti pozivaju metode udaljenih objekata na gotovo identičan način kao i u slučaju lokalnih metoda; ipak, RMI je mnogo sporiji i manje pouzdan od uobičajenih lokalnih poziva metoda – sa RMI-jem može doći i dolazi do problema koji ne utiču na lokalne pozive metoda.

RMI REGISTAR: kako bi ih klijent pronašao RMI servisi moraju biti registrovani kod *lookup* servisa zvanog RMI registar – on se izvršava kao zaseban proces i dozvoljava aplikaciji da registruje RMI servise ili da pribavi referencu na imenovan servis; svakom registrovanom servisu se dodeljuje ime koje klijenti koriste da pronađu servis – klijent nije svestan stvarne fizičke lokacije servisa; Java platforma uključuje RMI registarsku aplikaciju `rmiregistry` koja se može podesiti da osluškuje dolazeće konekcije.

DEFINISANJE FUNKCIONALNOSTI SERVISA: prvi korak pri kreiranju RMI servisa je definisanje funkcionalnosti u interfejsu koji implementira `Remote` interfejs - `Remote` je *marker* interfejs koji nema sopstvene metode i njegova jedina svrha je da „označi“ objekte tako da se mogu identifikovati od strane RMI servisa; ovaj pod-interfejs od `Remote` određuje koje metode klijenti RMI servisa mogu da pozivaju – RMI servis može imati mnogo javnih metoda, ali samo one koje su deklarisane u `Remote` interfejsu se mogu pozivati udaljeno (druge `public` metode se mogu pozivati samo u okviru JVM u kojoj objekat živi).

IMPLEMENTACIJA: pored implementacije metoda RMI servisnog interfejsa, klasa koja funkcioniše kao implementacija RMI servisa bi trebala da nasledi `UnicastRemoteObject` klasu

koja pruža osnovne RMI funkcionalnosti poput eksporta RMI servisa i pribavljanja klijenta koji komunicira sa servisom; nasleđivanje `UnicastRemoteObject` klase je jedini kôd specifičan za RMI koji se mora napisati u okviru implementacije servisa – van toga nikakav dodatni mrežni kôd nije potreban; nakon što implementacija servisa postoji alat `rmic` koji je deo JDK-a se koristi za kreiranje *stub*-ova i kostura (korišćenje `rmic` sa RMI-jem je *deprecated* u novijim verzijama Jave budući da se *stub*-ovi i kosturi generišu dinamički).

REGISTRACIJA RMI SERVISA: proces asociranja RMI servisa sa imenom u registru se zove registracija (*binding*); registracija se može izvršiti kroz statičke metode `Naming` klase:
- `void bind(String name, Remote obj)` – registruje objekat pod određenim imenom;
- `void rebind(String name, Remote obj)` – ponovo registruje objekat pod određenim imenom, postojeća registracija pod datim imenom biva pregažena;
- `void unbind(String name)` – uništava registraciju pod zadatim imenom;
- `String[] list(String name)` – niz imena registrovanih u datom registru.

Ime svakog RMI servisa se specifira u URL formatu `rmi://host:port/name`, gde su `host` i `port` parametri opcioni – *default* host je `localhost` dok je *default* port `1099` (`rmi` odrednica sheme je takođe opciona); objekat koji zaista registruje i instancira RMI servis se naziva RMI server.

POZIVANJE RMI SERVISA: kako bi se pozvao RMI servis klijentska aplikacija mora samo da pribavi referencu na udaljeni interfejs – ne mora se brinuti o tome kako se poruke šalju ili primaju, niti gde se servis nalazi; kako bi se servis pronašao izvršava se *lookup* operacija nad RMI registrom – klijentska aplikacija poziva sledeći statički metod `Naming` klase:
- `Remote lookup(String name)` – vraća *stub* za RMI servis koji je vezan za prosleđeno ime.

Svaki metod u `Remote` interfejsu deklariše `RemoteException` u listi mogućih izuzetaka.

08 Pregled Java tehnologije. Višeslojne aplikacije u Javi EE, motivacija i osobine.

PREGLED JAVA TEHNOLOGIJE: Java tehnologija obuhvata programski jezik – objektno orijentisani jezik visokog nivoa sa određenom sintaksom i stilom, i platformu – određeno okruženje u kom se izvršavaju Java aplikacije, koje se sastoji od: JVM – aplikacije napisane za određen hardver i softverska platforma za pokretanje Java aplikacija, *Application Programming Interface* – kolekcija softverskih komponenti koje se koriste za stvaranje aplikacija ili drugih softverskih komponenti. Postoje tri Java platforme: **1)** *Standard Edition* – *core* funkcionalnost jezika koja definiše sve od osnovnih tipova do klasa visokog nivoa, **2)** *Enterprise Edition* – izgrađena na osnovi Jave SE, koristi se za razvoj i pokretanje velikih višeslojnih mrežnih aplikacija, **3)** *Micro Edition* – JVM malog otiska za pokretanje Java aplikacija na uređajima sa ograničenim resursima (poput smart telefona).

***ENTERPRISE* APLIKACIJE:** Java EE platforma je dizajnirana da pomogne u razvijanju velikih, višeslojnih, skalabilnih, pouzdanih i bezbednih mrežnih aplikacija koje se često zovu i *enterprise* aplikacije budući da često služe podršci funkcionisanja velikih preduzeća (ipak, benefiti *enterprise* aplikacija mogu biti od pomoći čak i malim organizacijama i pojedinačnim *developer*-ima); Java EE platforma je dizajnirana da redukuje kompleksnost razvoja *enterprise* aplikacija kroz pružanje modela razvoja, API-ja i *runtime* okruženja. Funkcionalnost ovakvih aplikacija je razdeljena u više izolovanih funkcionalnih oblasti zvanih slojevi (*tiers*) – višeslojna aplikacija na bazi Java EE sastoji se od sledećih slojeva: **1)** klijentskog, **2)** web, **3)** poslovnog i **4)** *data* ili *enterprise information system* (EIS) sloja (ipak, aplikacije na bazi Java EE se smatraju

troslojnim, budući da su distribuirane na tri lokacije – **1)** klijentske računare, **2)** Java EE serverski računar i **3)** računar sa bazom podataka u *back end*-u); troslojne aplikacije proširuju standardni dvoslojni klijent-server model smeštanjem višenitnog aplikacionog servera između klijenta i skladišta podataka u *back end*-u. Java EE server je aplikacija koja implementira Java EE platforme i pruža standardne Java EE servise (Java EE serveri se često zovu i aplikacioni serveri budući da dozvoljavaju *developer*-ima da serviraju aplikaciju klijentu poput web servera koji nude web stranice); tradicionalno Java EE serveri pružaju podršku za rukovanje transakcijama, kontrolu konkurentnosti, bezbednosnu autorizaciju i balansiranje učitavanja; trenutno postoji pet Java 8 EE sertifikovanih servera: *Eclipse GlassFish*, *Wildfly*, *WebSphere*, *JBoss* i *InforSuite*.

SLOJEVI JAVA EE APLIKACIJA: klijentski sloj se izvršava na klijentskim računarima u vidu *browser*-a, apleta ili samostalnih aplikacija (web-bazirani klijenti komuniciraju kroz web sloj, dok samostalne aplikacije mogu da pristupe direktno poslovnom sloju); web sloj je međusloj kod web-baziranih klijenata i služi kao veza sa *core* funkcionalnošću aplikacije (*host*-uje web aplikacije napravljene od *servlet*-a, *JavaServer Pages* i *JavaServer Faces*); poslovni sloj skladišti *core* funkcionalnost *enterprise* aplikacije (izvršava *Enterprise Java Bean*-ove – EJB, i *Web Services* – WS); *EIS* sloj se sastoji od servera baze podataka i pristupa mu se kroz Java EE server korišćenjem *Java Database Connectivity* API-ja, *Java Persistence* API-ja, i slično.

09 Koncept *object-relational mapping*-a. JPA entiteti, osnovne JPA anotacije i koncept *programming-by-exception*.

JAVA PERSISTENCE API: perzistencija je svojstvo objekta koje mu dozvoljava da sačuva svoje stanje u vremenu izvršavanja i učita sledeći put kada se program pokrene; Java nudi dva glavna načina za čuvanje stanja objekta: **1)** serijalizaciju, i **2)** objektno-relaciono mapiranje koje dozvoljava programerima da na objektno-orijentisan način razmišljaju o klasama, interfejsima objektima i sličnom, a da pristup relacionim bazama delegiraju eksternim alatima *framework*-a; *Java Persistence API* (JPA) je standardni Java EE ORM *framework*.

JPA ENTITETI: kada je reč o perzistentnim objektima JPA specifikacija koristi termin „entitet“, dok u JPA terminologiji „objekat“ označava POJO; entitet je zapravo objekat za dodatnim svojstvima: **1)** može se trajno sačuvati u bazi, **2)** može se dobaviti iz baze, **3)** prati definisan životni ciklus.

OBJEKTNO RELACIONO MAPIRANJE: u radu JPA se oslanja na metapodatke asocirane sa svakim entitetom, obično obezbeđenih putem anotacija (za JPA, kao i za celu Java EE platformu, anotacije predstavljaju jedan od najvažnijih programerskih konstrukta); paket `javax.persistence` definiše velik broj anotacija od kojih su neke: `@Entity` – markira anotiranu klasu kao entitet, `@Id` – markira polje entiteta kao primarni ključ (može biti bilo koji primitivan tip i *wrapper* primitivnih tipova, `String`, `Date`, `BigDecimal`, `BigInteger`), `@GeneratedValue` – ukazuje na to da se vrednost polja generiše automatski (npr. kod primarnih ključeva); primer entitet klase `Person`. Pravila entiteta: **1)** svakom entitetu odgovara tabela u bazi koja se kreira sa istim imenom; **2)** svako polje entiteta postaje kolona tabele, pri čemu JPA vodi računa o konverziji između Java i SQL tipova podataka (`@Transient` anotacija se koristi za markiranje polja koja su neperzistentna); **3)** entitet mora imati *default* konstruktor bez argumenata.

***PROGRAMMING BY EXCEPTION*:** kao i mnoge druge Java EE specifikacije, JPA koristi koncept programiranja po izuzetku – osnovna ideja koncepta je u tome da JPA prati određena *default* pravila mapiranja (npr. ime tabele u bazi odgovara nazivu entiteta, ime kolone odgovara polju

entiteta); ukoliko se želi drugačije ponašanje moraju se obezbediti dodatni metapodaci (primer entiteta `Person` za koji se želi naziv tabele `Human` i slično).

10 Svrha *persistence unit*-a. Upravljanje entitetima i EntityManager. Upravljanje transakcijama.

***PERSISTENCE UNIT*:** svi entiteti aplikacije bazirane na JPA bi trebali biti nabrojani u *persistence unit*-u, koja pored toga sadrži određen broj atributa koji opisuju **1)** koja konkretno implementacija JPA se koristi (*EclipseLink*, *TopLink*, *Hibernate*), **2)** ime i drajver ciljne databaze kao i njenu fizičku lokaciju, *username*, *password* i ostalo, i **3)** kako bi trebalo rukovoditi transakcijama baze podataka; *persistence unit* je opisan u eksternom fajlu persistence.xml koji obično nije potrebno ručno editovati.

KONTEKST PERZISTENCIJE: reč je o skupu rukovođenih instanci entiteta u datom trenutku, za datu transakciju i može se posmatrati kao vid *cache*-a: kada aplikacija zatraži entitet *persistence* API će prvo proveriti trenutni kontekst perzistencije, a ako se entitet ne pronađe API će pokušati da ga učita iz baze; entiteti žive u kontekstu perzistencije tokom trajanja transakcije – nakon što se transakcija izvrši, sve promene u kontekstu perzistencije se oslikavaju u bazi.

***ENTITY MANAGER*:** u pitanju je objekat koji se koristi za interakciju sa kontekstom perzistencije – upravlja stanjima i životnim ciklusima entiteta i koristi se za dobavljanje entiteta iz konteksta perzistencije; kada *entity manager* dobavi referencu na entitet za njega se kaže da je *managed* – do tog moment entitet je *detached* POJO; snaga JPA je u tome da se entiteti mogu koristiti kao obični objekti od strane različitih slojeva aplikacije i da ulaze u *managed* stanje od strane *entity managera* kada je potrebno učitati ili ubaciti podatke u bazu. Interfejs `EntityManager` predstavlja JPA entity manager i uključuje sledeće metode:

- `void persist(Object entity)`: entitet ulazi u *managed* i *persistent* stanje;
- `void detach(Object entity)`: izbacuje se iz konteksta perzistencije i postaje *detached*;
- `void refresh(Object entity)`: osvežavanje statusa instance iz baze, u kom slučaju se prebrišu sve promene napravljene nad entitetom, ako ih je i bilo;
- `void remove(Object entity)`: uklanjanje entiteta, koji se tako briše iz baze i postaje *detached* u odnosu na *entity manager*;
- `<T> T find(Class<T> entityClass, Object primaryKey)`: traženje entiteta specifirane klase i primarnog ključa.

UPRAVLJANJE TRANSAKCIJAMA: za razliku od aplikacija na bazi Java SE koje moraju ručno da upravljaju transakcijama, one na bazi Java EE to prepuštaju Java EE serveru; transakcija se kreira pozivanjem `getTransaction()` metoda nad instancom `EntityManager` interfejsa; same transakcije su predstavljene `EntityTransaction` interfejsom:

- `void begin()`: započinje transakciju resursa;
- `void commit()`: izvršava trenutnu transakciju resursa, zapisuje sve promene iz konteksta perzistencije u bazu;
- `void rollback()`: vraćanje resursa trenutne transakcije na početno stanje.

Primer perzistencije osobe (gradivo BP2).

11 JPQL i rad sa upitima.

JPQL: *Java Persistence Query Language* se koristi za definisanje pretraga usmerenih na perzistentne entitete bez obzira na bazu koja se nalazi u osnovi; sintaksa je bazirana na SQLu, s tom razlikom da se rezultati u SQLu dobijaju u vidu tabela dok JPQL vraća entitete ili kolekcije istih; JPQL podržava upite: `SELECT`, `UPDATE`, `DELETE`. Glavni tipovi upita podržanih od strane JPA: **1)** dinamički – string upita se gradi dinamički u vremenu izvršavanja, **2)** imenovani – specifiraju se u vremenu kompajliranja, **3)** *native* – izvršavanje nativne SQL naredbe; imenovani upiti su efikasniji od dinamičkih budući da *persistence provider* može da prevede JPQL u SQL po pokretanju aplikacije umesto da to radi svaki put kada se upit izvršava; *native* upiti bi se trebali koristiti samo ako aplikacija cilja određenu bazu.

UPRAVLJANJE UPITIMA: nezavisno od tipa JPA upiti su predstavljeni `Query` interfejsom koji nudi obilje metoda za upravljanje upitima:

- za izvršavanje `SELECT` upita koriste se metodi `getResultList()` i `getSingleResult()`;
- za izvršavanje `UPDATE` i `DELETE` upita koristi se metod `executeUpdate()` koji vraća broj entiteta koji su ažurirani ili obrisani.

Dodatno, JPA nudi i `TypedQuery` što je generička verzija `Query` interfejsa. Instance `Query` interfejsa se dobavljaju od `EntityManager` instance:

- `Query createQuery(String jpqlQuery)`: dinamički upit zasnovan na JPQL stringu;
- `Query createNativeQuery(String sqlQuery)`: nativni upit zasnovan na SQL stringu;
- `Query createNamedQuery(String name)`: imenovan upit od prosleđenog imena.

Svaki metod ima i generičkog parnjaka koji vraća instancu `TypedQuery` tipa. Primeri (BP2).

12 Osnove EJB tehnologije, EJB kontejner i usluge koje pruža. Aplikacioni serveri. Dve osnovne kategorije (i pod-kategorije) EJB-ova.

***ENTERPRISE JAVABEANS*:** *Enterprise JavaBeans* (EJB) su upravljane komponente sa serverske strane koje enkapsuliraju poslovnu logiku *enterprise* aplikacija – koriste se za izgradnju poslovnog sloja višeslojnih aplikacija; EJB tehnologija je uvedena kako bi se upravljalo uobičajenim problemima tokom *development*-a poput perzistencije, transakcionog integriteta i bezbednosti na standardan način, što omogućava programerima da se skoncentrišu na konkretan problem.

EJB KONTEJNERI: EJB-ovi se izvršavaju unutar kontejnera – okruženja u vremenu izvršavanja koje obezbeđuje uobičajene *feature*-e za većinu *enterprise* aplikacija, a koji obuhvataju: **1)** *Remote client communication* – bez pisanja kompleksnog koda EJB klijent može da pozove metode udaljeno preko standardnih protokola; **2)** *Dependency injection* – kontejner može da injektuje više resursa direktno u EJB; **3)** *State Management* – kontejner može da upravlja stanjima nekih EJB-ova umesto određenog klijenta na transparentan način; **4)** *Pooling* – kontejner kreira *pool* instanci koje dele više klijenata, nakon poziva EJB se vraća u *pool* kako bi se ponovo koristio umesto da se uništi; **5)** *Component life-cycle*; **6)** *Asynchronous calls* – neki EJB-ovi mogu da komuniciraju sa drugim komponentama (i obrnuto) korišćenjem asinhronih poziva metoda ili slanje poruka; **7)** *Transaction management* – kontejner se brine o *commit* i *rollback* operacijama; **8)** *Security* – kontrola pristupa na nivou klasa ili metoda se može specifirati nad EJB-om kako bi se sprovela autentikacija korisnika i uloga; **9)** *Concurrency support* – većina EJB-ova su *thread-safe* po prirodi. Kontejneri se pokreću unutar aplikacionih servera koji dodatno mogu da uvedu podršku za skalabilnost, balansiranje učitavanja, transparentni *failover*, administraciju, monitoring, keširanje, itd.

KATEGORIJE EJB-OVA: kako bi se izborili sa složenošću *enterprise* aplikacija uvedena su dva osnovna tipa EJB-ova: *session beans* – predstavljaju najvažniji deo EJB tehnologije budući da enkapsuliraju poslovnu logiku visokog nivoa, i *message-driven beans* (MDB), koji predstavljaju poslovne objekte čije se izvršavanje okida porukama umesto pozivima metoda (dodati su kako bi se podržalo programiranje zasnovano na događajima); MDB-ovi obično samo osluškuju dolazzeće poruke i pozivaju *session bean*-ove u skladu sa tim.

KATEGORIJE *SESSION BEAN*-OVA: u zavisnosti od tražene funkcionalnosti, aplikacija može da bira jedan od tipova *session bean*-ova: **1)** *stateless* – ne održavaju nikakvo konverzaciono stanje sa klijentskom aplikacijom (koriste se za upravljanje zadacima koji se mogu završiti u jednom pozivu metode), **2)** *stateful* – održavaju stanje i vezani su za određenog klijenta (korisni su kod zadataka koji se izvršavaju u nekoliko koraka), **3)** *singleton* – postoji samo jedna instanca *singleton session bean*-a za čitavu aplikaciju.

13 *Stateless* EJB i koncept *pooling*-a. *Stateful* EJB i poređenje sa *stateless* EJB-ovima. Aktivacija i pasivacija, uništavanje *stateful* EJB-ova.

***STATELESS BEAN*-OVI:** najpopularnija *session bean* komponenta, idealni su za implementaciju zadataka koji se završavaju u jednom pozivu metode; mogu se *pool*-ovati i deli ih nekoliko klijenata: **1)** za svaki *stateless* EJB kontejner drži određeni broj instanci u memoriji (to je *pool*), **2)** kada klijent pozove metodu nad *stateless bean*-om kontejner kupi instancu iz *pool*-a i dodeljuje je klijentu, **3)** a kada se završi klijentski zahtev instanca se vraća u *pool* kako bi se ponovo koristila. *Stateless bean*-ovi su efikasni zato što samo je potreban svega mali broj *bean*-ova za opsluživanje mnoštva klijenata – ista instanca se može iznova i iznova reciklirati, smanjujući na taj način memorijske zahteve i broj memorijskih (de)alokacija; ipak, uzastopne metode koje poziva isti klijent mogu završiti u različitim instancama *stateless bean*-ova; anotacija kojom se POJO pretvara u *stateless bean*: `@Stateless`.

***STATEFUL BEAN*-OVI:** održavaju konverzaciono stanje sa klijentom – korisni su za zadatke koji se moraju izvršiti u nekoliko koraka; kada klijent pozove *stateful bean* na serveru EJB kontejner mora da dostavi istu instancu za svaki naredni poziv metoda; *stateful bean*-ovi se ne mogu koristiti ponovo od strane drugih klijenata, što znači da mora biti onoliko instanci *bean*-ova koliko ima i klijenata; kako bi se izbegao veliki otisak u memoriji serveri koriste tehniku aktivacije i pasivacije: **1)** pasivacija je proces uklanjanja instance iz memorije i njeno čuvanje na perzistentnoj lokaciji pre nego što stigne sledeći zahtev od klijenta, **2)** aktivacija je obrnut proces učitavanja stanja i primenjivanje istog na instancu. Anotacija je `@Stateful` – kontejner automatski uklanja instancu *stateful bean*-a kada se klijentska sesija završi ili istekne; ipak, programeri mogu da kontrolišu kada bi instanca trebala da se ukloni koristeći anotacije `@Remove` (anotira metod, izaziva da instance *bean*-a bude trajno uklonjena iz memorije nakon poziva metoda) i `@StatefulTimeout` (dodeljuje `timeout` vrednost koja predstavlja trajanje tokom kog *bean* može da ostane *idle* pre nego što ga kontejner ukloni).

14 Udaljeni i lokalni pristup EJB-u. *Dependency injection. Singleton* EJB, inicijalizacija i kontrola konkurentnog pristupa.

UDALJENI I LOKALNI PRISTUP EJB-U: *session bean*-ovi mogu biti dizajnirani za lokalnu (u okviru iste JVM) ili udaljenu upotrebu, ili za obe; udaljena funkcionalnost *session bean*-a se mora specifirati u interfejsu anotiranom sa `@Remote` dok lokalna koristi `@Local` (alternativno se

anotira *bean* sa `@LocalBean`); interfejsi za udaljene i lokalne *session bean*-ove se često nazivaju poslovni interfejsi budući da sadrže deklaraciju poslovnih metoda koje su vidljive klijentu i implementirane od strane *bean* klase.

***DEPENDENCY INJECTION*:** mehanizam koji se koristi za injekciju referenci resursa u polja – osnovna ideja DI je da komponente prosto izlistaju sve neophodne servise koje će zatim DI *framework* da dobav, bez da se zavisnosti *hard-code*-iraju (zavisnosti uključuju reference na druge komponente, EJB-ove, odgovarajuće drajevere baze podataka, i slično); Java EE koristi anotacije za DI:
- `@EJB`: injektuje reference na EJB u anotiranu promenljivu;
- `@PersistenceContext`: injektuje instancu *entity manager*-a u anotiranu varijablu.

***SINGLETON BEAN*-OVI:** *session bean* koji je instanciran jednom po aplikaciji – kod kolica za kupovinu *singleton bean* bi mogao predstavljati globalni cenovnik, što bi sprečilo aplikaciju da pretražuje bazu za svakog novog klijenta; kako bi se anotirao *singleton bean* koristi se `@Singleton`. Po *default*-u *singleton bean* se kreira prvi put kada ga klijent koristi, ali se može forsisrati inicijalizacija odmah po pokretanju aplikacije kroz anotiranje klase sa `@Startup`; ukoliko se više *singleton bean*-ova koriste za inicijalizaciju podataka za aplikaciju i ako se moraju inicijalizovati u određenom redosledu koristi se `@DependsOn` anotacija (prihvata jedan ili više stringova koji indikuju imena *singleton bean*-ova koji se moraju inicijalizovati pre anotiranog). Za razliku od drugih *session bean*-ova *singleton bean*-ovima se može i pristupa se konkurentno – postoje dva glavna načina kontrole konkurentnosti pristupa: **1)** *Container-managed concurrency* (CMC), gde kontejner kontroliše konkurentni pristup instanci *bean*-a na osnovu metapodataka; **2)** *Bean-managed Concurrency* (BMC), gde kontejner dozvoljava puni konkurentni pristup i prebacuje odgovornost za sinhronizaciju na *bean* (CMC je *default*-no i poželjno ponašanje).

***CONTAINER-MANAGED CONCURRENCY*:** kod CMC-a programeri koriste `@Lock` anotaciju kako bi specifirali kako kontejner mora da upravlja konkurentnošću kada klijent pozove metod; anotacija ima vrednosti:
- `@Lock(LockType.WRITE)`: ekskluzivni *lock* za pisanje, što je *default* atribut za zaključavanje;
- `@Lock(LockType.READ)`: deljeni *lock* za čitanje.

`@Lock` može biti specifiran na klasi ili metodima – ako je specifiran na klasi primenjuje se na sve metode; kako bi se izbeglo neodređeno dugo blokiranje klijenta aplikacija koristi anotaciju `@AccessTimeout` koja indikuje odbacivanje zahteva ako nije pribavljen *lock* u određenom vremenskom intervalu.