

**OBJEKTNO ORIJENTISANO  
PROGRAMIRANJE 1  
ODGOVORI NA PITANJA ZA  
USMENI DEO ISPITA  
J A N U A R 2 0 2 1**

Git: <https://github.com/NikolaVetnic/OOP1>



01

Objekti i klase.

**OBJEKTNO ORIJENTISAN PROGRAM:** ideja – pakovanje podataka i funkcija koje ih obrađuju u jednu celinu predstavljenu jednom promenljivom, tj. objektom sačinjenom od atributa/polja (promenljive, opisuju stanje) i metoda (funkcije, definišu ponašanje); OO program je skup objekata u međusobnim interakcijama (npr. objA poziva metode nad objB), a konkretno Java program je kolekcija **1)** klasa (apstraktne, konkretne – finalne i nefinalne, anonimne, lokalne) i **2)** interfejsa (ugovor koji kaže da ako klasa implementira interfejs tada ona implementira sve metode propisane interfejsom) koji se mogu grupisati po **3)** paketima (prostori imena, mehanizam hijerarhijske dekompozicije – klase u istom paketu imaju različita, a u različitim paketima mogu imati ista imena), koji obično sadrže pakete, srodne klase i interfejse.

**KLASA:** definicija skupa istorodnih objekata (ista specifikacija stanja i ponašanja, tj. atributi i metode), klase su referencijalni tipovi (tip određuje skup vrednosti i skup operacija nad promenljivom); pisanje OO programa podrazumeva definiciju klasa i instanciranje (kreiranje) objekata definisanih klasa (primer klase i objekata sa crteža: – klasa kao „prototip“, objekti kao konkretne instance sa različitim stanjima).

#### MODIFIKATORI:

- 1) deklaracija klase koja nije ugnježdjena može početi nekim od modifikatora: public, abstract, final, strictfp (reprezentacija i operacije sa realnim brojevima po IEEE 754 standardu);
- 2) deklaracija polja može početi modifikatorima: public, protected, private, final, static, transient (polje se ne koristi pri serijalizaciji i deserijalizaciji objekata date klase), volatile (sinhronizovano polje, atomičke operacije kod višenitnih aplikacija);
- 3) zaglavlje metoda može početi modifikatorima: public, protected, private, abstract, final, static, synchronized (sinhronizovana metoda), native (apstraktan metod implementiran u nekom drugom jeziku u nekoj od biblioteka koje se dinamički učitavaju), strictfp (striktno floating point operacije po IEEE 754).

**IMPERATIVNI PJ:** program je formalni opis procesa (imperativni PJ) ili specifikacije (deklarativni PJ) izračunavanja u PJ, ostvaren kroz skup naredbi (naredbe dodele, kontrole toka, itd.) koje menjaju vrednosti skupa promenljivih (odnosno stanje programa). Kategorije: **1)** proceduralni – dekompozicija u funkcije i procedure, lokalne i globalne promenljive; **2)** modularni – dekompozicija u module (logički srodne definicije promenljivih, funkcija i tipova, sa privatnim i javnim delom); **3)** OO – dekompozicija u klase (logički srodne def. promenljivih i funkcija, klase kao tipovi, privatni i javni deo, nasleđivanje klasa).

**KONSTRUKTORI I DESTRUKTORI:** konstruktor – specijalni metod koji se poziva pri instanciranju klase radi inicijalizacije stanja objekta, destruktor – specijalni metod koji se poziva kada se objekat uništava. U Java PJ: **1)** instanciranje operatorom new; **2)** rezultat je referenca na instancirani objekat; **3)** moguće je više referenci na isti objekat; **4)** nema destruktora, automatsko uništavanje objekata (*garbage collector*) kada više nema referenci na objekat.

02

Apstrakcija, enkapsulacija i sakrivanje informacija.

**APSTRAKCIJA:** suština PJ je da omoguće mehanizme: **1)** apstrakcije - sakrivanje (zanemarivanje, odlaganje, zaboravljanje) nebitnih (tehničkih, implementacionih) detalja, što smanjuje

kompleksnost; **2** dekompozicije – razlaganje entiteta (problema) u pod-entitete koji su manje kompleksnosti. **Paradigme:** **1** proceduralni PJ – dekompozicija u procedure, gde korisnik zna samo zaglavlje a ne i telo procedure da bi je koristio; **2** OO PJ – dekompozicija u objekte, gde korisnik zna samo zaglavlja javnih metoda objekta.

**SAKRIVANJE INFORMACIJA:** public i private delovi objekta, kroz modifikatore vidljivosti/pristupa prilikom definicije atributa i metoda klase – privatni atributi vidljivi samo u klasi i van nje su dostupni samo preko metoda (primer: get i set metode), privatne metode se pozivaju samo iz iste klase, javnim atributima i metodama se može pristupiti iz drugih klasa.

**ENKAPSULACIJA:** dizajn objekta koji skriva detalje nebitne za korišćenje od korisnika (ako su svi atributi privatni tada objekat ima potpunu kontrolu nad svojim stanjem), a takođe promena atributa klase ne utiče na druge klase kod održavanja i modifikacije programa (primer: class Tacka sa private double x, y poljima).

03

Agregacija i kompozicija klasa.

**AGREGACIJA I KOMPOZICIJA:** osnova za interakciju objekata – objekat a klase A ima referencu na objekat b klase B, odnosno u definiciji klase A postoji atribut p tipa B; ukoliko klasa A instancira p tada su A i B u **relaciji kompozicije**, budući da je b deo a koji ne može da postoji bez a i uništavanjem objekta a uništava se i objekat b (primer: A = Čovek, B = Srce); ako klasa A ne instancira p tada su A i B u **relaciji agregacije**, jer b može da postoji nezavisno od a (primer: A = FudbalskiKlub, B = Fudbaler).

04

Promenljive referencijalnog tipa, referenca this.

**PROMENLJIVE:** postoje različite vrste promenljivih: atributi klase, parametri metoda, lokalne promenljive u telu metoda; leksička pravila opsega – za neku pojavu identifikatora u kodu tražimo toj pojavi najbližu definiciju tipa, i to prvo u tekućem bloku a onda dalje redom po nadgnježenim blokovima, listi parametara metoda i na kraju atributima klase (*najpametnije je prosto izbegavati konflikte imena bilo koje vrste*).

**PROMENLJIVE REFERENCIJALNOG TIPRA:** vrednost promenljive čiji je tip neka klasa je ili referenca na objekat te klase, ili null (što znači da ona tada ne pokazuje ni na jedan objekat) – reč je o posebnoj vrednosti koja se može dodeliti bilo kojoj promenljivoj referencijalnog tipa; budući da su klase ref. tipovi one mogu biti tipovi parametara i povratne vrednosti metoda; promenljive ovog tipa se implicitno inicijalizuju na null; moguće je kreirati nizove objekata čiji je tip neka klasa.

**REFERENCA this:** svaki objekat ima implicitno definisano polje koje se zove this i to je upravo referenca na samog sebe, što je zgodna referenca kada postoji kolizija na nivou imena (primer: inicijalizacija atributa u konstruktoru).

**REFERENCA super:** još jedno polje implicitno prisutno u svim objektima, super referencira delove objekta koji su nasleđeni; koristeći super može se pristupiti nasleđenim metodama i atributima u slučaju da su redefinisani u izvedenoj klasi.

**OPERATORI:** operatorom dodele = se kopira vrednost reference, ne objekat; logičkim operatorima == i != se porede reference a ne vrednosti objekta.

05

Konstruktori i operator new.

**KONSTRUKTORI:** konstruktor (ctor) je poseban tip metoda koja se poziva prilikom instanciranja klase; klasa može definisati više ctor-a koji se tada moraju razlikovati po broju parametara ili bar jedan parametar mora biti različitog tipa; ctor takođe može pozvati drugi ctor, u kom slučaju se koristi ključna reč `this` iza koje se zadaju vrednosti parametara (na osnovu tipova kojih se određuje koji ctor se poziva) – ovo mora biti prva naredba u ctor-u; ctor takođe može koristiti `super` čime se poziva odgovarajući ctor nadklase, što se opet određuje na osnovu parametara; kod nasleđivanja ako bazna klasa definiše bar jedan ctor tada izvedena mora imati bar jedan ctor sa super pozivom, a ako to nije slučaj dodaje joj se podrazumevani ctor sa naredbom `super()`; ako klasa definiše ctor i prva naredba nije `super()` ili `this()` poziv kompajler automatski dodaje `super()` kao prvu naredbu (greška prilikom kompajliranja ako bazna klasa nema ctor bez argumenata).

Primer, prolazi kompajliranje:

```
public class Foo {
    private int x;
    public Foo(int x) {
        this.x = x; } }

public class Bar extends Foo {
    private int y;
    public Bar(int x, int y) {
        super(x);
        this.y = y; }
    public Bar(int y) {
        this(0, y); } }
```

Primer, ne prolazi kompajliranje jer **1** u `Bar(int, int)` kompajler automatski dodaje `super()` ali `Foo` nema definisan konstruktor bez parametara, a **2** nasleđeno polje `x` je privatno i nije vidljivo u klasi `Bar`:

```
public class Foo {
    private int x;
    public Foo(int x) {
        this.x = x; } }

public class Bar extends Foo {
    private int y;
    // ne prolazi kompajliranje!
    public Bar(int x, int y) {
        this.x = x;
        this.y = y; } }
```

06

Statički atributi i metodi, *singleton pattern*.

**STATIČKI ATRIBUTI I METODI:** definisani korišćenjem ključne reči `static`, oni nisu vezani za objekte nego za klasu tako da im se pristupa preko imena klase, ali moguće je i preko objekta; budući da postoje nezavisno od objekata klase, statički atributi i klase postoje čak i kada klasa nije nijednom instancirana; statički metodi ne mogu pristupati nestatičkim atributima, dok nestatičke metode mogu pristupati i jednim i drugim; svaki objekat ima svoju (nezavisnu) kopiju nestatičkih atributa, a svi objekti dele iste statičke attribute (koji se ne kopiraju):

```
public class Ucenik {
    private int id;
    private String ime;
    private static int idVal;
    pub. stat. v. initID(int start) {
        idVal = start; }
    public Ucenik(String ime) {
        id = idVal++;
        this.ime = ime; }
    ... }

class Brojac {
    private int val = 0;
    // singleton brojac
    private static Brojac singleton = null;
    // sakrivamo konstruktor
    private Brojac() {}
    public static Brojac getInstance() {
        if (singleton == null)
            singleton = new Brojac();
        return singleton;
    }
    public void inc() { val++; }
    public int getVal() { return val; }
}
```

**SINGLETON PATTERN:** šema u dizajnu softvera – ograničavanje instanciranja na jedan objekat:

Korisno je kada je tačno jedan objekat potreban da koordiniše akcije u čitavom programu.

07

Konstruktori i nasleđivanje.

**KONSTRUKTORI I NASLEĐIVANJE:** odgovor pokriven odgovorom na pitanje 05 i druga.

08

Redefinisanje nasleđenih atributa i metoda

**REDEFINISANJE NASLEĐENIH ATRIBUTA I METODA:** odgovor pokriven odgovorom na pitanje 09 i druga.

09

Nasleđivanje, Liskov princip supstitucije i dinamičko vezivanje.

**NASLEĐIVANJE:** klase se mogu međusobno nasleđivati (ključna reč *extends*) i to tako da ako *A* nasleđuje *B* tada: **1)** *A* nasleđuje sve attribute i metode klase *B* (nasleđuju se i privatni članovi ali nisu vidljivi bez modifikatora *protected*), **2)** objekat klase *A* sadrži attribute i metode definisane u klasi *B*, **3)** u klasi *A* se mogu redefinisati nasleđeni atributi i metode (*overriding*) i naravno dodati novi, **4)** *A* – izvedena klasa / podklasa / klasa dete, **5)** *B* – bazna klasa / nadklasa / super klasa / klasa roditelj; ako *A* nasleđuje *B* tada su one u relaciji „*is a*“ (primer: Mačka „*is a*“ Životinja); postoji direktno i indirektno nasleđivanje, a ukoliko klasa ne nasleđuje nijednu drugu ona implicitno nasleđuje *Object* iz *java.lang*.

Nasleđivanje klase je mehanizam ponovnog iskorišćenja koda (*code reuse*) koji omogućava: **1)** specijalizaciju – izvedena klasa dodaje nove attribute i nove metode, **2)** proširenje postojeće funkcionalnosti – izvedena klasa dodaje nove metode, **3)** modifikaciju postojeće funkcionalnosti – izvedena klasa modifikuje nasleđene metode; na ovaj način se kod iz bazne klase ne kopira, pa tako nema klonova u kodu, čije su pak posledice glomazniji programi koji se teže održavaju (primer: *class Kvadrat extends Pravougaonik*).

**LISKOV PRINCIP SUPSTITUCIJE:** ako klasa *S* nasleđuje klasu *T* tada objekat klase *S* može da se koristi gde god se očekuje objekat klase *T* (primer: *ekran.nacrtaj(pravougaonik / kvadrat)*):

```
T a = new T();
```

```
T b = new S();
```

I *a* i *b* su reference tipa *T*, što znači da referenca tipa *T* može da pokazuje na objekte klase T ali i objekte klase izvedenih iz T (tip u vremenu kompajliranja i tip u vremenu izvršavanja):

```
public class Pravugaonik {
    protected double a, b;
    public Pravugaonik(double a, double b) {
        this.a = a;
        this.b = b;
    }
    public boolean podudaranSa(P-nik p) {
        return a == p.a && b == p.b;
    }
}

p. c. Kvadrat extends Pravugaonik { ... }
=====
Pravugaonik p = new Pravugaonik(4, 4);
Kvadrat k = new Kvadrat(4);
if (p.podudaranSa(k))
    System.out.println("Podudarni");
```

„Svaki kvadrat je pravougaonik, nije svaki pravougaonik kvadrat“.

**POLIMORFIZAM I DINAMIČKO VEZIVANJE:** polimorfizam – različito ponašanje operatora ili metode (*overloaded* operatori i metodi) u zavisnosti od tipova argumenata (primer: '+' kao znak za sabiranje i konkatenciju); statičko vezivanje – određivanje se vrši u vremenu kompajliranja, dinamičko vezivanje – određivanje se vrši u vremenu izvršavanja; polimorfizam u PJ Java –

koja od preopterećenih metoda se poziva se određuje u vremenu izvršavanja na osnovu tipova referenci u vremenu izvršavanja (a ne kompajliranja):

```
Figure f1 = new Square();
double d1 = f1.calcSurface();
```

```
Figure f2 = new Circle();
double d2 = f2.calcSurface();
```

U različitim PJ postoji jednostruko i višestruko nasleđivanje (jedna klasa nasleđuje više klasa); u PJ *Java* postoji samo jednostruko nasleđivanje.

**ANOTACIJA @Override:** anotacije su dodatne informacije o klasi i delovima klase; počinju sa @ i daju se pre definicije klase, metoda, atributa; nemaju nikakav efekat na izvršavanje ali mogu biti zgodne kompajleru prilikom detekcije nekih vrsta grešaka, kao i za zanemarivanje nekih upozorenja; konkretno @Override data pre definicije metoda ukazuje da će taj metod redefinisati nasleđeni metod (ako metod takvog zaglavlja ne postoji u baznoj klasi generiše se greška u vreme kompajliranja).

**KLJUČNA REČ final:** ako je klasa definisana sa ključnom rečju final tada se ona ne može nasleđivati; ako je metod definisan sa final on se ne može redefinisati; ako je atribut definisan sa final on ne može menjati vrednost (konstanta) nakon inicijalizacije.

10

instanceof operator i eksplicitne konverzije referenci.

**instanceof OPERATOR:** u pitanju je binarni infiksni (što znači da dolazi između operanada) operator kojim proveravamo da li je objekat instanca neke klase – p instanceof C, gde je p promenljiva referencijalnog tipa i C ime klase (ime ref. tipa); rezultat je true/false; operator instanceof uzima u obzir nasleđivanje, odnosno vraća true za proveru svih klasa koje direktno ili indirektno nasleđuje klasa objekta koji se proverava.

#### EKSPPLICITNE KONVERZIJE REFERENCI:

```
class Figura {}
class Pravougaonik extends Figura {}
class Kvadrat extends Pravougaonik {}
```

```
Figura f = new Kvadrat();
Kvadrat k = (Kvadrat) f;
Pravougaonik p = (Pravougaonik) f;
```

11

Klasa Object.

**KLASA Object:** klasa na vrhu hijerarhije nasleđivanja, sadrži metode:

- 1) **boolean equals(Object o):** proverava da li su dva objekta identična po sadržaju,
- 2) **String toString():** vraća string reprezentaciju objekta, poziva se kada konkatenujemo string objekat sa objektom nekog drugog tipa, automatski se poziva od S.O.P.(),
- 3) **int hashCode():** vraća *hash* kod objekta, dva objekta identična po sadržaju moraju imati isti *hash* kod.

12

Apstraktne klase.

**APSTRAKTNE KLASKE:** apstraktne klase se deklariraju ključnom reči abstract, sadrže apstraktne metode (metodi koji nisu implementirani i kod kojih je dato samo zaglavlje) i one se ne mogu instancirati (ne može se primeniti operator new); smisao apstraktnih klasa je **1)** obezbeđenje neke opšte funkcionalnosti, kao i **2)** da se specifične funkcionalnosti implementiraju

izvedenim klasama, budući da se **apstraktne uvek nasleđuju** (primer: `abstract class Figura` koja definiše metode `obim()` i `toString()`).

Apstraktne klase dakle pored apstraktnih metoda (samo zaglavlje bez implementacije) sadrže i konkretne (sa zaglavljem i telom), ali takođe ne moraju da sadrže nijednu apstraktnu metodu (ako klasa sadrži bar jednu apstraktnu metodu ona mora biti apstraktna); hijerarhija klasa u OO programima – od apstraktnih ka specifičnim klasama (primer: niz nastavnika od kojih su neki asistenti a neki profesori); primer instanciranja izvedenih klasa:

```
abstract class Prnt {          public S(int x) {          psvm(String[] args) {
    int x;                    super.x = x;      }
    abstract void t(); }      void t() {          S s = ts.new S(5);
                                S.O.P(super.x); } }      T ss = ts.new S(5);
class C extends Prnt {        }                  ss.t(); s.t(); }
```

U apstraktnim klasama implementiraju se opšte funkcionalnosti – specifične funkcionalnosti u apstraktnim klasama su apstraktne i implementiraju se tek u izvedenim klasama; opšte funkcionalnosti mogu koristiti specifične funkcionalnosti – konkretna metoda iz apstraktne klase može pozvati apstraktnu metodu u kom slučaju se izvršava neki konkretan metod iz neke od izvedenih klasa po pravilu dinamičkog vezivanja; ako klasa nasleđuje apstraktnu klasu tada ona mora implementirati sve nasleđene apstraktne metode ili i sama mora biti apstraktna.

## 13

Modifikatori vidljivosti.

**MODIFIKATORI VIDLJIVOSTI:** `public` – klasa/interfejs vidljiv u svim paketima, član klase vidljiv za sve klase svih paketa; `protected` – član klase vidljiv za sve klase svog paketa i za nasleđene klase, ili interfejse iz bilo kog paketa; `default` (bez modifikatora) – klasa/interfejs vidljiv samo u okviru svog paketa, član klase vidljiv za sve klase u istom paketu; `private` – član klase vidljiv samo u okviru svoje klase.

modifikator	klasa / interfejs	član klase
<b>public</b>	u svim paketima	za sve klase svih paketa
<b>protected</b>	/	za sve klase i nasleđene kl. svog, za interfejse bilo kog paketa
<b>default</b>	u svom paketu	za sve klase u istom paketu
<b>private</b>	/	samo u okviru svoje klase

Tabela vidljivosti prema modifikatoru.

## 14

Inicijalizacija statičkih atributa klase.

**STATIČKI ATRIBUTI:** mogu se inicijalizovati direktno prilikom deklaracije atributa, ili u statičkom inicijalizatoru (kojih može biti više) oblika `static { .. }`:

```
public class Foo {          static {
    priv. st. double dvaPi = 2.0 * Math.PI;      ljudi = new Covek[10];
    priv. st. d. cexp = M.log10(3.4) + dvaPi;      for (int i = 0; i < ljudi.length; i++)
    private static Covek[] ljudi;                  ljudi[i] = newCovek("Covek " + i); } }
```

Prilikom direktne inicijalizacije statičkog atributa i u statičkom inicijalizatoru mogu se koristiti samo statička polja i metode.

**STATIČKI INICIJALIZATORI:** može ih biti više, izvršavaju se u redosledu u kojem su navedeni:

```
public class Bar {          static {
    private static int x, y;      System.out.println("Inicijalizujem y");
    static {                      y = 2; }
        System.out.println("Inicijalizujem x");      public static void main(String[] args) {
        x = 1; }                  System.out.println(x + " ", " + y"); } }
```



Statički inicijalizatori se izvršavaju tačno jednom (pri prvom referenciranju klase, kada se `.class` fajl učitava u memoriju *JVM*), a služe za inicijalizaciju kompleksnih statičkih atributa klase.

15

Inicijalizacija nestatičkih atributa klase.

**NESTATIČKI ATRIBUTI:** mogu se inicijalizovati direktno, u konstruktoru i u nestatičkim inicijalizatorima; nestatički inicijalizatori su blok naredbe bez zaglavlja i sa njima ne treba preterivati:

```
public class FooBaz {
    private int x, y, z;
    { System.out.println("Inicijalizator 1");
      x = 10; }
    public FooBaz() {
        System.out.println("Konstruktor");
    }
    y = 20; }
    { System.out.println("Inicijalizator 2");
      z = 30; }
    public static void main(String[] args) {
        new FooBaz(); } }
```

Nestatički inicijalizator izvršava se pri svakom instanciranju klase; pristupa i statičkim i nestatičkim elementima; svi inicijalizatori se izvršavaju u redosledu u kom su navedeni, a nestatički inicijalizatori se izvršavaju pre konstruktora klase, a nakon što se završi izvršavanje konstruktora iz bazne klase: **ctor bazne klase** → **nestatički inicijalizator** → **ctor klase**.

16

Ugnježdene statičke klase.

**UGNJEŽDENE KLASSE (UOPŠTENO):** u pitanju su klase definisane unutar neke druge klase kada imamo potrebu za pomoćnom klasom koja se ne koristi van te klase (povećana enkapsulacija i čitljivost koda), ili kada je klasi prirodno mesto unutar neke druge klase (npr. kod klasa koje definišu kontejner prirodno je tip elementa definisati unutar njega).

**UGNJEŽDENE STATIČKE KLASSE:** razlika u odnosu na unutrašnje (iliti ugnježdene **nestatičke**) se ogleda u instanciranju van spoljašnje klase i moogućnostima pristupa elementima spoljašnje klase; drugi referencijalni tipovi (interfejsi i nabrojivi) takođe se mogu definirati unutar neke klase i implicitno su statički ugnježdjeni tipovi.

Budući da su statički elementi nezavisni od instanci klase tako i statičke ugnježdene klase se mogu instancirati nezavisno od instanci spolje klase – tada statičku ugnježdenu klasu van spoljne referenciramo punim (kanoničkim) imenom:

```
public class Spoljasnja {
    public static class Ugnjezdjena {
        public Ugnjezdjena(int x) { . } }
    priv. Ugnjezdjena u = new Ugnjezdjena(4); }
    public class FooBar {
        private Spoljasnja.Ugnjezdjena ug
        = new Spoljasnja.Ugnjezdjena(2); }
```

Postoje takođe i puna i relativna imena ugnježđenih statičkih klasa (primer).

**VIDLJIVOST:** ako je ugnježdjena statička klasa `public` tada je ona vidljiva koliko je vidljiva njena spoljašnja klasa, a ako je `private` tada nije vidljiva van spoljašnje; ako je `protected` tada je vidljiva u paketu najviše spoljašnje klase ako su sve njene spoljašnje klase javne ili deklarirane bez modifikatora pristupa, i u klasama izvedenim iz spoljašnje klase; ista pravila važe i za ugnježdene nestatičke klase, ugnježdene interfejse i ugnježdene nabrojive tipove; ugnježdene statičke klase mogu pristupati samo statičkim elementima spoljašnje klase, uključujući i privatne.

**NASLEĐIVANJE:** ugnježdene statičke klase se mogu nasleđivati, iako se u praksi to uglavnom ne radi (gotovo nikad van spoljašnje klase); realizuje se kao nasleđivanje običnih klasa (koristi se puno ime ako je nasleđivanje van spoljašnje klase).

17

Unutrašnje klase.

**UGNJEŽDENE NESTATIČKE (UNUTRAŠNJE) KLASSE:** razlika u odnosu na ugnježdene statičke se ogleda u instanciranju van spoljašnje klase i mogućnostima pristupa elementima spoljašnje klase; dve specijalne vrste unutrašnjih (ugnježdenih nestatičkih) klasa su **1**) lokalna klasa – klasa deklarisanu u nekom bloku naredbi, i **2**) anonimna klasa – singleton klasa bez imena koja se definiše prilikom instanciranja singleton objekta.

Instance unutrašnje klase su vezane za instance spoljašnje klase – unutrašnje klase se van spoljašnje klase instanciraju pozivajući operator `new` nad objektom spoljašnje klase:

```
public class Outer {
    public class Inner {
        public void hello(String x) {
            S.O.P.(x); } }
    public void m() {
        Inner i = new Inner();
        i.hello("Outer.m()"); } }

public class OuterInner {
    public static void main(String[] args) {
        Outer o = new Outer();
        o.m();
        Outer.Inner i = o.new Inner();
        i.hello("OuterInner.main()"); } }
```

Još jedan primer:

```
public class SA {
    public static class SB {
        public static class SC {} } }

public class A {
    public class B {
        public class C {} } }

public class FooBar {
    public void m() {
        SA.SB sb = new SA.SB();
        A.B b = new A().new B();
        SA.SB.SC cs = new SA.SB.SC();
        A.B.C c2 = new A().new B().new C(); } }
```

Instance unutrašnje klase su vezane za instance spoljašnje klase i mogu pristupiti svim njenim elementima (uključujući i privatne); unutrašnja klasa može deklarirati atribut/metod istog imena kao atribut/metod iz spoljašnje klase (*shadowing*), pa se tada “zasenjenom” identifikatoru iz spoljašnje klase može pristupiti koristeći kvalifikovani `this` izraz oblika `ImeKlase.this`.

U sledećem primeru `C.this` je referenca na atribut/metode spoljašnje klase `C`, a kval. izraz `C.super` je referenca na atribut/metode bazne klase spoljašnje klase `C`):

```
public class A {
    private int x = 10;
    public class B {
        private int x = 20;

        public class C {
            private int x = A.this.x + B.this.x;
        } } }
```

U unutrašnjoj klasi nije moguće definisati statičke metode, a statička polja je moguće definisati samo ako je statičko i finalno (kod ugnježdenih statičkih nema ovih ograničenja).

Primer agregacije (klase `Radnik` i `RadnaOrganizacija`) i kompozicije (klase `Cvor` i `ListaBrojeva`) – u prvom slučaju unutrašnja klasa ima smisla bez spoljašnje, a u drugom ne.

**VIDLJIVOST:** pokriveno odgovorom na pitanje 16.

**NASLEĐIVANJE:** unutrašnje klase se takođe mogu nasleđivati, iako se u praksi to uglavnom ne radi (gotovo nikad van spoljašnje klase); kod nasleđivanja van spoljašnje klase mora se pozvati njen konstruktor nad vezanim objektom spoljašnje klase:

```
public class A {
    public A (int x) {}
    public class B {
        public B (char c) {} } }

class C extends A.B {
    public C (A a, char c) {
        a.super(c); } }
```

18

Lokalne i anonimne klase.

**LOKALNE KLASSE:** definisane u bilo kom bloku programskog koda, instanciraju se u onom bloku u kom su definisane:

```
public int saberi(String prostIzraz) {
    int pIdx = prostIzraz.indexOf("+");
    if (pIdx == -1)
        throw new IllegalArgumentException...
    Str sab1 = pIzraz.substr(0, pIdx).trim();
    Str sab2 = pIzraz.substr(pIdx + 1).trim();

    class Par {
        private int prvi, drugi;

        public Par(String prvi, String drugi) {
            this.prvi = Integer.parseInt...
            this.drugi = Integer.parseInt... }
        pub. int saberi() { ret prv + drg; } }

    Par p = new Par(sab1, sab2);
    return p.saberi(); }
}
```

Lokalne klase mogu biti deklarisanе sa modifikatorima `abstract`, `final`, `strictfp` ili bez; ne mogu biti `private`, `public` ili `static`, atributi/metodi ne mogu biti `static` (osim ako su i `final`); mogu pristupati atributima i vidljivim lokalnim varijablama koje su `final` (konstante) ili **efektivno final** (ne menjaju vrednost nakon inicijalizacije):

```
class Foo {
    private final int x = 2;
    public void method() {
        final int y = 5;
        class Loc 1 {
            private int z = x + y; } } }
}
```

**ANONIMNE KLASSE:** radi se o *singleton* klasama bez imena, mogu se definisati u bilo kom bloku programskog koda i prilikom direktne inicijalizacije atributa; anonimna klasa se definiše prilikom kreiranja singleton objekta (definicija klase daje se iza operatora `new`); ovakve klase ili nasleđuju neku klasu `C` (`C c = new C(parametri_ctor-a_C) telo_anon_klase`) ili implementiraju neki interfejs `I` (`I i = new I() telo_anon_klase`); anonimna klasa nema ime i zato ne može imati ni `ctor`:

```
public abstract class Akcija {
    protected String ime;
    public Akcija(String ime) {
        this.ime = ime; }
    public abstract void akcija(); }

public class AnonimnaKlasa {
    public static void main(String[] args) {
        Akcija p = new Akcija("pozdrav") {
            public void akcija() {
                S.O.P("Akcija: " + ime); } };
        p.akcija(); } }
}
```

19

Interfejsi.

**INTERFEJSI:** specifikacije funkcionalnosti klase, ugovori kojima se definišu fundamentalne interakcije između klasa: klasa koja implementira interfejs je obavezna da implementira sve funkcionalnosti propisane njime, a kada druga klasa koristi ovakvu klasu koristi je na način specifikovan interfejsom; kao i klase, interfejsi su referencijalni tipovi (`I` – interfejs, `r` – prom. tipa `I`, tada je `r` referenca na objekat bilo koje klase koja implementira `I`), ali se interfejsi ne mogu instancirati; u telu interfejsa je moguće specifikirati:

- konstante
- zaglavlja metoda (najčešće)
- default i static metode
- ugnježdene static tipove

Sve deklaracije u interfejsu su implicitno public:

- **promenljive** implicitno `static` i `final`
- **zaglavlja metoda** implicitno `abstract`
- **ugnj. klase i interfejsi** implicitno `static`
- **metode u int. impl.** `static` ili `default`

Klasa može implementirati više od jednog interfejsa; interfejsi su referencijalni tipovi promenljivih:

```
Krug k = new Krug(2, 4, 5);
k.prosiri(2);
k.pomeri(1, 1);

Prosiriv p = new Krug(1, 2, 4);
p.prosiri(3);

// p.pomeri(1, 1); --ne kompajlira se
Translabilan t = new Krug(3, 4, 1);
t.pomeri(2, 3);
// t.prosiri(3); --ne kompajlira se
```

**default METODE:** klasa koja implementira interfejs nasleđuje njegove default metode i može da ih redefiniše:

```
public interface Prosiriv {
    void prosiri (double k);
    default voids manji(double k) {
        if (k <= 0)
            throw new IllegalArg...;
        prosiri(1 / k); } }
```

**KOLIZIJE:** kod implementacije dva ili više interfejsa kolizije se na nivou statičkih elemenata trivijalno rešavaju upotrebom punih (kvalifikovanih) imena; problem je u tome što može doći do kolizije na nivou imena i to ne na nivou zaglavlja (jer njih tek treba implementirati) nego na nivou default metoda (različite implementacije za ista zaglavlja) – ako klasa A implementira interfejs X i Y koji oba sadrže default metod istog zaglavlja tada klasa A mora redefinisati tu default metodu (inače ne prolazi kompajliranje).

20

Nasleđivanje interfejsa.

**NASLEĐIVANJE INTERFEJSA:** može biti jednostruko:

```
public interface Knjiga {
    String ime();
}

public interface Udzbenik {
    String oblast();
}

class ITUdzb implements Udzbenik {
    private String ime;

    public ITUdzb(String ime) {
        this.ime = ime;
    } }
```

Nasleđivanje interfejsa može biti i višestruko:

```
public interface NaucniRdn {
    String opisPosla = "op1";
}

public interface ProsvetniRdn {
    String opisPosla = "op2";
}

public interface Profesor
    extends NaucniRdn, ProsvRdn {
    String imeFak = "imeF";
    Profesor pr = new Profesor(..);
    NaucniRdn n = pr;
    S.O.P.(n.opisPosla());
    ProsvRdn p = pr;
    S.O.P.(p.opisPosla());
}
```

**UGNJEŽDENI TIPOVI:** unutar interfejsa je moguće definisati ugnježdenu static klasu, interfejs ili enum; unutar klase je takođe moguće definisati interfejs (koji je tada implicitno static):

- 1) primer Knjiga: URL
- 2) primer RadnaJedinica: slajdovi sa predavanja 05-16.

21

Nabrojivi tipovi (*enum*-i).

**NABROJIVI TIPOVI:** promenljiva ovog tipa uzima neku vrednost iz skupa eksplicitno nabrojanih simboličkih konstanti (primer: Dan, Mesec, Planeta); definisanje tipa vrši se ključnom rečju enum i u definiciji tipa nabrajaju se vrednosti:

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

Vrednosti nabrojivog tipa se navode kvalifikovano uz navođenje naziva tipa (osim u switch naredbi): Dan.PON, Dan.UTO, i slično (primer).

*Enum*-i su specijalna vrsta klasa, svaki implicitno nasleđuje `java.lang.Enum` koja nasleđuje `java.lang.Object` – ne nasleđuju druge tipove, mogu implementirati proizvoljno mnogo interfejsa:

```
public enum Planet {
    MERCURY (3.303e+23, 2.43e6),
    ..
    NEPTUNE (1.024e+26, 2.47e7);
    private final double mass;
    private final double rad;

    priv Planet (double m, double r) {
        this.mass = m; this.rad = r;
    }

    pub st fin d G = 6.67e-11;
    public double surfaceGrav() {
        return G * mass / (rad * rad);
    } }
```

Kao što se vidi iz primera, svaka vrednost nabrojivog tipa je objekat koji može imati stanje i ponašanje, pa tako u *enum*-ima možemo, pored obaveznog navođenja liste konstanti, definisati i attribute, metode, non-public ctor-e (eksplicitno se pozivaju prilikom navođenja konstantni), unutrašnje tipove:

```
Enum Predmet {
    BP1(6, "Rackovic", true),
    OOP1(6, "Savic", true),
    SPA2(6, "Savic", true);
    private final int espb;
    private final String prof;
    private final boolean online;
    public Predmet(int e, String p, bool o) {
        this.espb = e;
        this.prof = p;
        this.online = o;
    }
    public String toString() {
        return prof + ", " + espb + ", " + online;
    }
}
```

Svaki *enum* ima implicitno definisan statički metod `values()` koji vraća niz koji sadrži sve konstante definisane *enum*-om:

```
public static void main(String[] args) {
    Planet[] p = Planet.values();
    for (int i = 0; i < p.length; i++)
        S.o.p(p[i] + ", " + p[i].surfGrv());}
```

## 22

Klasa `String`, informativne metode.

**STRING:** reč je o nizu karaktera; klasa `String` je iz paketa `java.lang` pa je zato ne moramo eksplicitno importovati, a pruža osnovne operacije u radu sa stringovima kao i naprednije metode za procesiranje tekstualnih podataka; `String` je `final` klasa – ne može se nasleđivati; ova klasa skriva iternu reprezentaciju samog stringa – nije dozvoljena direktna manipulacija nizom karaktera; objekti klase `String` su *immutable* – stanje objekta ne može da se promeni, a metodi koji „transformišu“ string zapravo proizvode nove objekte.

Klasa `String` je „privilegovana“ – objekti ove klase mogu se kreirati bez `new`, mogu se konkatenerirati korišćenjem operatora `+` kako međusobno, tako i sa objektima drugih klasa; ova klasa implementira interfejs `Comparable<String>` (leksikografsko poređenje stringove koristeći `compareTo` metod); sadrži mnoštvo ctor-a; sve metode klase `String` dele se na informativne (vraćaju neku informaciju o stringu) i transformativne (kreiraju nove `String` objekte na osnovu postojećih).

**INFORMATIVNE METODE:** spisak informativnih metoda dat je u nastavku:

```
char charAt(int pos)
boolean equals(Object o)
int compareTo(String o)
int length()
boolean endsWith(String suff)
boolean startsWith(String pref)
boolean equalsIgnoreCase(String p)
int indexOf(String s) // prva poj.
int indexOf(String s, int pos)
int lastIndexOf(String s) // posl.
int lastIndexOf(String s, int p)
```

## 23

Klasa `String`, transformativne metode.

**OSNOVNE TRANSFORMATIVNE METODE:** metode koje kreiraju nove `String` objekte na osnovu postojećih:

```
String trim()
String concat(String d)
String toLowerCase()
String toUpperCase()
String substring(int beg, int end)
String substring(int beg)
String replace(CharSequence target, CharSequence replacement)
```

**NAPREDNE TRANSFORMATIVNE METODE:** ove metode zasnovane su na regularnim izrazima – stringovima koji opisuju skup stringova sa istom strukturom i koji se sastoje od karaktera i operatora:

```
[abcd]      - izraz koji opisuje bilo koji od navedenih karaktera
[^abcd]     - izraz koji opisuje bilo koji karakter osim navedenih
```

[a-z]	- izraz koji opisuje jedan karakter iz datog opsega
[a-zA-Z]	- unija dva opsega
[a-m&&c-z]	- presek dva opsega
\d	- bilo koja cifra
\s	- belina (space, tab, new-line)
\w	- isto što i [a-zA-Z0-9]
X?	- tačno jedna ili nijedna pojava stringa koji zadovoljava X
X*	- nula, jedna ili više pojava stringa koji zadovoljava X
X+	- jedna ili više pojava stringa koji zadovoljava X
X{n}	- tačno n pojava stringa koji zadovoljava X
XY	- string koji zadovoljava X konkateniran sa stringom koji zadovoljava Y
X   Y	- string koji zadovoljava ili X ili Y
(X)	- zagradama možemo regulisati prioritet

### Napredne transformativne metode:

`boolean matches(String regex)` - proverava da li string zadovoljava regex

`String replaceAll(String regex, String replacement)` - zamenjuje svaki podstring koji zadovoljava regex sa datom zamenom

`String replaceFirst(String regex, String replacement)` - zamenjuje samo prvi podstring koji zadovoljava regex datom zamenom

`String[] split(String regex)` - deli string u niz tokena naspram delimitera u regex-u

### Primer naprednih transformativnih metoda:

```
public static void main(String[] args) {
    String ulaz = "B434 ;. Truc ... Tika;Tak";
    // separator sve sto nije niz slova/cifara
    String sepRegex = "[^a-zA-Z0-9]+";
    String[] tok = ulaz.split(sepRegex);

    for (int i = 0; i < tok.length; i++) {
        String t = tok[i].trim();
        // izbacujemo sve brojeve
        t = t.replaceAll("[0-9]+", "");
        System.out.println(t);
    }
}
```

Postoje još i statičke metode `valueOf()` za konverziju primitivnih tipova u stringove.

24

Regularni izrazi i transformativne metode klase `String`.

**REGULARNI IZRAZI:** odgovor pokriven odgovorom na pitanje 23.

25

Klasa `StringBuilder`.

**KLASA `StringBuilder`:** problem sa objektima klase `String` je što su *immutable*, što znači da se prilikom svake transformacije kreira novi `String` objekat; klasa `StringBuilder` predstavlja mutabilne stringove, odnosno reč je o klasi kojom se prave stringovi primenjujući stringovne operacije koje ne kreiraju nove objekte; koriste se operacije `insert` (dodavanje stringa u string na proizvoljnu poziciju) i `append` (dodavanje stringa na kraj stringa) koje ne postoje u klasi `String`.

Konstruktori (napomena: `CharSequence` je interfejs koji implementiraju klase koje realizuju stringove – `String`, `StringBuilder`, `StringBuffer`):

<code>StringBuilder()</code>	<code>StringBuilder(CharSequence c)</code>
<code>StringBuilder(int capacity)</code>	<code>StringBuilder(String s)</code>

`CharSequence` je interfejs koji implementiraju klase koje realizuju stringove – `String`, `StringBuilder`, `StringBuffer`:

<code>char charAt(int index);</code>	<code>int length();</code>	<code>CharSeq subSequence(int start, int end);</code>
--------------------------------------	----------------------------	---

Metode klase `StringBuilder`:

<code>StringBuilder append(int i, Obj o...)</code>	<code>int indexOf(String s)</code>
<code>StringBuilder insert(int pos, int i...)</code>	<code>int indexOf(String s, int pos)</code>
<code>StringBuilder delete(int start, int end)</code>	<code>StringBuilder reverse()</code>
<code>StringBuilder deleteCharAt(int index)</code>	<code>String substring(int start)</code>
<code>SB replace(int start, int end, String repl)</code>	<code>String substring(int start, int end)</code>

Korišćenje `StringBuilder` klase umesto „obične“ konkatencije stringova drastično smanjuje količinu iskorištene memorije kao i brzinu izvršavanja.

26

Izuzeci i vrste izuzetaka.

**IZUZECI:** objekti kojima se signalizira pojava neke greške prilikom izvršavanja programa; moguće ih je: **1)** generisati (naredba `throw`), čime se signalizira pojava greške (kod proveravanja se tipovi specificiraju u zaglavlju metode iza ključne reči `throws`); **2)** obraditi (`try-catch-finally`, `try-with-resources`), čime se obrađuje nastala greška tako što, ako se izuzetak generiše, tok se prebacuje na odgovarajuću `catch` granu, dok se `finally` grana izvršava u svakom slučaju; **3)** proslediti (`throws` sa svim tipovima koji se prosleđuju), čime se izuzetak prosleđuje metodi koja ga je pozvala i koja će ga zatim i obraditi.

Postoje dve vrste izuzetaka:

**1)** proveravani (*checked*), izuzetak koji se proverava u toku izvršavanja i koji se MORA obraditi ili proslediti (primer: `FileNotFoundException`, `IOException`), i

**2)** neproveravani (*unchecked*) – `Runtime` ili `NullPointerException` izuzeci, izuzetak koji se NE MORA ni obraditi ni proslediti (najčešće je u pitanju *bug* koji se popravlj a ne obrađuje), primeri ovakvog tipa: `NullPointerException`, `ArithmeticException` (deljenje nulom), `ClassCastException` (*cast*-ovanje u pogrešan tip), `IllegalArgumentException`, `NegativeArraySizeException`, `NoSuchElementException`, `ArrayIndexOutOfBoundsException`, `StringIndexOutOfBoundsException`, `NumberFormatException` (tekstualni string kao ulaz kod parsiranja celobrojnog tipa) – primer:

```
public void dodajSaGreskom(String orig, String prevod) {
    LinkedList<String> prevodi = r.get(orig);
    prevodi.add(prevod); // null pointer! }
public void dodaj(Str orig, Str prevod) {
    LinkedList<String> prevodi = r.get(orig);
    if (prevodi == null) {
        prevodi = new LinkedList<String>();
        r.put(orig, prevodi); }
    prevodi.add(prevod); }
```

Svi izuzeci su objekti klase `Exception` ili klase koja nasleđuje `Exception` (paket `java.lang`); neproveravani su objekti `RuntimeException`; nove izuzetke definisemo nasleđivanjem klase `Exception` (direktno ili indirektno).

27

Obrada i prosleđivanje izuzetaka.

Program prekida sa radom i ispisuje se *stack trace* ako se:

**1)** proveravani izuzetak prosleđuje sve do *JVM*-a (main metod prosledi neki proveravan izuzetak, npr. `IOException`),

**2)** neproveravani izuzetak se generiše i ne obradi (a ne treba obrađivati ako su to faktički bagovi, npr. `NumberFormatException`).

**OBRADA IZUZETAKA:** izuzeci koji se mogu generisati u nekom bloku koda obrađuju se `try-catch-finally` ili `try-with-resources` naredbom.

**try-catch-finally:** ako se izuzetak generiše kontrola toka se prebacuje na odgovarajuću `catch` granu – na ovaj način kod za obradu grešaka nije izmešan sa kodom koji realizuje normalan tok izvršavanja što daje čitljiviji i razumljiviji kod; kod u `finally` grani se izvršava

neovisno od toga da li je neki izuzetak bio generisan ili ne (primer sa čitanjem iz fajla pomoću `BufferedReader`-a); `finally` grana služi za oslobađanje resursa bilo da se greška desila ili ne, obavezna je čak i kada nema `catch` grana (tada se izuzetak prosleđuje uz oslobađanje resursa), a izvršava se čak i ako se metod prekine `return` naredbom; takođe, može postojati više `catch` grana (primer sa `FileNotFoundException` i `IOException`), a jedna `catch` grana može obrađivati više tipova izuzetaka – tzv. *multi-catch* grana (`catch (SQLException | IOException e)`).

**try-with-resources:** ovde je `try` naredba parametrizovana jednim resursom ili sa više resursa koji su međusobno razdvojeni znakom „;“, a svaki resurs je objekat klase koja implementira `Closeable` ili `AutoCloseable` interfejs (što su npr. sve klase koje realizuju tokove podataka); na kraju `try` naredbe resursi bivaju automatski zatvoreni (poziva se `close()` metod); `try-with-resources` ne mora imati ni `catch` ni `finally` grane.

**PROSLEĐIVANJE IZUZETAKA:** metoda može da prosledi izuzetak onoj metodi koja ju je pozvala; za proveravanje izuzetke je obavezno u zaglavlju metoda navesti da metod prosleđuje izuzetke tako što se iza ključne reči `throws` nabrajaju tipovi izuzetaka koji se prosleđuju razdvojeni zarezima:

```
public class ZdravoSvete {
    private static void pisi
        (String outFileName)
        throws IOException {
        PrintWriter pw = new PrintWriter(
            new BufferedWriter(
                new FileWriter(outFileName)));
        pw.println("Zdravo svete");
        pw.close(); }

    public static void main(String[] args) {
        try {
            pisi("out.txt");
        } catch(IOException e) {
            System.out.println(
                "Greska prilikom pisanja u fajl");
        } } }
```

## 28

Generisanje i definisanje izuzetaka.

**GENERISANJE IZUZETAKA:** izuzeci se generišu koristeći `throw` naredbu – ako metoda generiše proveravane izuzetke (drugi metod u primeru dole) tada takve tipove izuzetaka treba specifirati u zaglavlju metode iza ključne reči `throws`:

```
public String prvaDvaSlova(String rec) {
    if (rec.length() < 2)
        throw new IllegalArgumentException...
    return rec.substring(0, 2); }

public String prvaLinija(String ulazniFajl)
    throws IOException {
    File f = new File(ulazniFajl);
    if (!f.exists()) throw new IOException...
    if (!f.canRead()) throw new IOException...
    try (BufferedReader br...) {
        return br.readLine(); } }
```

**DEFINISANJE IZUZETAKA:** nove tipove izuzetaka definišemo nasleđujući klasu `Exception` direktno ili indirektno:

```
public class KratakString
    extends Exception {
    public KratakString() {
        super("Str ima manje od dva slova"); } }

public String prvaDvaSlova(String rec)
    throws KratakString {
    if (rec.length() < 2)
        throw new KratakString();
    return rec.substring(0, 2); }

public void stampajPrvaDva(String str) {
    try {
        S.O.P.(prvaDvaSlova(str));
    } catch(KratakString ks) {
        S.O.P.(ks.getMessage()); } }
```

Klasa `Throwable` direktno nasleđuje `Object`, a nju nasleđuju `Exception` i `Error` (čine je izuzeci od kojih je oporavak uglavnom nemoguć, reč je o greškama *JVM*-a).

## 29

Brojač linija tekstualnog fajla.

URL ZADATKA: [https://github.com/NikolaVetnic/OOP1/tree/master/src/usmeni\\_ispit/qB29\\_brojac\\_linija](https://github.com/NikolaVetnic/OOP1/tree/master/src/usmeni_ispit/qB29_brojac_linija)



30

Kopiranje tekstualnog fajla.

URL ZADATKA: [https://github.com/NikolaVetnic/OOP1/tree/master/src/usmeni\\_ispit/qB30\\_kopiranje\\_txt\\_fajla](https://github.com/NikolaVetnic/OOP1/tree/master/src/usmeni_ispit/qB30_kopiranje_txt_fajla)

31

Balansirane zagrade.

URL ZADATKA: [https://github.com/NikolaVetnic/OOP1/tree/master/src/usmeni\\_ispit/qB31\\_balansirane\\_zagrade](https://github.com/NikolaVetnic/OOP1/tree/master/src/usmeni_ispit/qB31_balansirane_zagrade)

32

Paketi i jedinice prevođenja.

**PAKETI:** definicije referencijalnih tipova (klase, interfejse, nabrojive) mogu se grupisati po paketima; svaka klasa/interfejs/enum pripada nekom paketu; u isti paket se smeštaju oni tipovi koji zajedno čine neku logičku celinu; paket je mehanizam za hijerarhijsko uređivanje komponenti softverskog sistema (paket može sadržati druge pakete), a takođe je i mehanizam za kontrolu konflikta imena (različiti paketi mogu sadržati definicije istog imena, dok u jednom paketu imena moraju biti različita); struktura paketa *Java* softverskog sistema korespondira strukturi direktorijuma u kojima se čuvaju izvorni i prevedeni fajlovi, odnosno paket  $\equiv$  direktorijum.

Paketi funkcionišu kao mehanizam sakrivajna informacija: tipovi iz paketa nisu vidljivi van paketa ukoliko nisu `public`; paket može da sadrži proizvoljan broj paketa (direktorijuma) i jedinica prevođenja (fajlova); paket ne mora da sadrži nijednu jedinicu prevođenja – tada sadrži druge podpakete ili je prazan; po konvenciji imena paketa počinju malim slovima; puno (kvalifikovano) ime tipa je ime koje sadrži ime tipa, ime paketa u kojem je tip definisan i imena svih nadpaketa (primer #1: puno ime klase *A* koja je definisana u paketu *z*, koji se nalazi u paketu *y*, koji se nalazi u paketu *x* je `x.y.z.A`; primer #2: puno ime klase *B* koja je ugnježđena u klasu *A* je `x.y.z.A.B`).

Neka je javna klasa *A* definisana u paketu *x*, i neka je klasa *B* definisana u paketu *y*; tada da bi koristili klasu *A* prilikom definisanja klase *B* neophodno je (ili-ili-ili): **1)** klasu *A* navoditi njenim punim imenom u telu klase *B*, **2)** u jedinici prevođenja (fajlu) u kojoj je definisana *B* uvesti (importovati) klasu *A* (koristi se „skraćeno“ ime – samo *A*), **3)** u jedinici prevođenja (fajlu) u kojoj je definisana *B* uvesti (importovati) sve klase iz paketa *x* (takođe se koristi „skraćeno“ ime).

Paketi su nezavisni od podpaketa koji su sadržani u njima – klase iz podpaketa se takođe moraju uvesti (importovati) ili navoditi punim imenima; uvozom (importom) svih klasa iz paketa se ne uvoze klase iz podpaketa.

33

Definicija jedinice prevođenja.

**JEDINICE PREVOĐENJA:** u pitanju je jedan tekstualni fajl sa ekstenzijom `.java`<sup>1</sup> koji sadrži: **1)** deklaraciju paketa – navodi se kom paketu jedinica prevođenja pripada, **2)** uvozne (`import`)

<sup>1</sup> Praktično, u pitanju je *Java* source code koji se piše prilikom kodiranja

deklaracije – deklaracije kojim se uvoze imena definisana u drugim paketima, **3)** definicija tipova – definicija klase, interfejsa i *enum*-a u jedinici prevođenja; ukoliko se u jedinici prevođenja ne navede deklaracija paketa tada sve definicije tipova iz te jedinice prevođenja pripadaju neimenovanom (anonimnom, podrazumevanom, *default*) paketu; ako je A javni tip (klasa/interfejs/*enum* imena A deklarisan sa modifikatorom *public*) tada A mora biti definisan u jedinici prevođenja koja se zove A.java – posledica: u jednoj jedinici prevođenja može postojati najviše jedna definicija javnog tipa.

**Primeri imena paketa**

```
package aritmetika;
package fizika.atomska;
package pera.diplomski.prviDeo;
```

**Primeri uvoza**

```
import java.util.ArrayList;
import java.util.*;
```

**Korišćenje klase bez uvoza**

```
java.util.ArrayList<Integer> v =
    new java.util.ArrayList<>();
```

**Uvoz statičkih članova klase**

```
import static java.lang.System.out;
```

**Višestruki uvoz statičkih članova klase**

```
import static java.lang.System.*;
```

Ako postoje dve klase iz različitih paketa koje se isto zovu tada se ili obe klase navode punim imenima, ili se jedna importuje a druga navodi punim imenom.

**34**

Imenovanje i dizajn paketa.

**IMENOVANJE PAKETA:** ime paketa se izvodi od obrnutog imena domena firme (rs.uns.pmf.dmi) + ime projekta (oop1) + ime komponente unutar projekta (lekcijs5); dizajn paketa prati hijerarhijsku strukturu komponenti softverskog sistema (sistem se sastoji od komponenti, komponente od podkomponenti, itd.); važi „*high cohesion, low coupling*“ princip (koji takođe važi i na nivou klasa): **1)** visoka kohezivnost unutar paketa – stepen zavisnosti klasa unutar paketa je veći nego njihov stepen zavisnosti od klasa van paketa, **2)** slaba povezanost sa drugim paketima – lokalizovana propagacija promena (promena klase unutar paketa ne zahteva promenu klasa van paketa).

**JAR FAJLOVI:** arhiva (kompresovani fajl) koji sadrži .class fajlove; Java biblioteke se distribuiraju kao .jar fajlovi.

**35**

Struktura JavaFX aplikacije, tipovi okana.

**GUI APLIKACIJE:** *graphical user interface* aplikacija sadrži interaktivni grafički korisnički interfejs: **1)** sadrži grafičke komponente (polja za unos teksta, dugmad, menije, itd.), **2)** korisnik u interakciji sa aplikacijom koristeći tastaturu, miš, touch screen, **3)** svaka grafička komponenta je jedan objekat; reč je o programiranju vođenom događajima (*event-driven programming*): **1)** program izvršava pozadinsku nit koja čeka na neki događaj koji korisnik generiše preko grafičkih komponenti interfejsa, **2)** za svaki događaj je vezan *event handler* – kod koji se izvršava kada je događaj generisan (kod za obradu događaja), **3)** kod konzolnih aplikacija interakciju program-korisnik kontroliše program a kod GUI aplikacija tu interakciju kontroliše korisnik.

Programiranje GUI aplikacija se sastoji od:

- 1) programiranja GUI-ja (kreiranje objekata koji predstavljaju komponente, raspoređivanje komponenti unutar okna – auto-raspoređivači u zavisnosti od tipa okna u koje se smeštaju komponente, ručno programiranje interfejsa VS grafički *drag-and-drop* editori sa auto-generisanjem koda / konfiguracionim fajlovima koji opisuju interfejs),

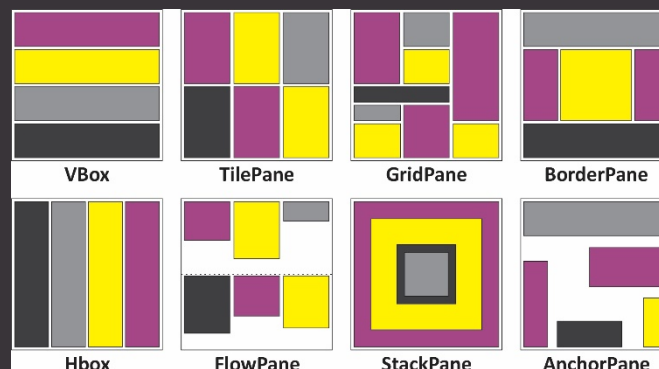
2) programiranje *event handler*-a,

3) vezivanje *event handler*-a za grafičke komponente (jedna komponenta interfejsa može generisati više tipova događaja – za svaki tip događaja imamo zaseban *event handler*, a takođe jedan *event handler* se može izvršavati na događaje koje generišu različite komponente).

**JAVA FX:** skup klasa za programiranje Java GUI aplikacija; GUI JavaFX aplikacije ima strukturu stabla u čijem korenu se nalazi **pozornica** (*stage*) – kod desktop GUI aplikacija pozornica je prozor sa *title bar*-om; **scena** (*scene*) – deo prozora u kojem možemo smeštati grafičke komponente (pozornica ima tačno jednu scenu); **čvorovi** (*nodes*) su grafičke komponente ili okna: okna su kontejneri koji mogu da sadrže grafičke komponente ili druga okna, na scenu se može postaviti tačno jedan čvor, a čvorovi se automatski raspoređuju u oknu u zavisnosti od tipa okna.

JavaFX aplikacija se pravi tako što glavna klasa nasledi `Application` iz `javafx.application`, nakon čega se implementira apstraktni metod `void start(Stage stage) throws Exception` kojim se kreira scena aplikacije i postavlja na pozornicu; sama aplikacija pokreće se metodom `launch()` iz klase `Application` – ovaj metod napravi instancu klase koja nasleđuje `Application` i nad tom instancom nakon neophodnih inicijalizacija poziva metod `void start(Stage stage) throws Exception`.

**JAVA FX OKNA:** okna se prilagođavaju dimenzijama scene – raspored i/ili veličina čvorova može da se promeni promenom veličine pozornice; za svaki čvor moguće je zadati minimalnu, poželjnu i maksimalnu veličinu (širinu i visinu): sve tri veličine se automatski inicijalizuju u zavisnosti od tipa grafičke komponente, čvor ima željenu veličinu ukoliko u oknu ima mesta za takvu veličinu, veličina se može smanjivati do minimalne, a povećavati do maksimalne veličine, u slučaju smanjivanja pozornice grafičke komponente mogu da se preklope, moguće je zadati eksplicitno poziciju i veličinu za svaki čvor ali se to ne preporučuje.



BorderPane okno: sadrži ctor-e koji primaju nula, jedan (*center*) ili pet čvorova; metode `setCenter()`, `setLeft()`, .., `setBottom()`; metode za postavljanje poravnanja za komponente unutar dela okna, za postavljanje margine oko okna, za postavljanje margine oko komponenti unutar okna.

FlowPane okno: ređa komponente jednu za drugom horizontalno/vertikalno sa prelaskom u novu „vrstu“/„kolonu“; ctor-i: `FlowPane()`, `FlowPane(double hgap, double vgap)`, `FlowPane(Orientation orient)` (*enum* koji definiše `HORIZONTAL` ili `VERTICAL`), kombinacija prethodna dva; metoda koja vraća listu čvorova koji se nalaze u oknu: `ObservableList<Node> getChildren()` – pozivajući metod `add(Node n)` nad tom listom dodaje se čvor u okno (`ObservableList` – lista uz koju možemo vezati funkcijski objekat koji se izvrši automatski kada se lista promeni).

Hbox i VBox okna: komponente se ređaju jedna za drugom horizontalno/vertikalno bez prelaska u novu „vrstu“/„kolonu“; ako je pozornica mala postoje nevidljive komponente koje se pojavljuju proširenjem pozornice.

GridPane okno: predstavlja tabelu (matricu) gde se čvor može postaviti u proizvoljnu ćeliju u tabeli; tabela raste automatski kako se dodaju čvorovi – ne navodi se veličina tabele prilikom kreiranja okna; jedna komponenta se može prostirati kroz više ćelija u tabeli; metode za dodavanje čvorova: `void add(Node child, int columnIndex, int rowIndex)`, `void add(Node child, int column, int row, int columnSpan, int rowSpan)` – prvo idx kolone, a onda vrste!

36

Tipovi i obrada događaja.

**TIPOVI DOGAĐAJA:** svaki tip događaja je realizovan odgovarajućom klasom koja direktno ili indirektno nasleđuje `javafx.event.Event`; neki od tipova događaja su: `KeyEvent`, `MouseEvent`, `MouseEvent`, `MouseDragEvent`, `WindowEvent`, `TouchEvent`, `SwipeEvent`, `ZoomEvent`, `ActionEvent` – događaj visokog nivoa apstrakcija uključujući i logički (a ne fizički) klik na dugme.

**OBRADA DOGAĐAJA:** *Event handler* je objekat klase koja implementira interfejs `EventHandler<T extends Event>` koji propisuje implementaciju samo jednog metoda – `void handle(T event)`; *Event handler* je potrebno vezati za grafičku komponentu:

```
Button btn = new Button("Click me");
class ClickHandler implements
EventHandler<ActionEvent> {
    @Override
    public void handle(ActionEvent event) {
        .. }
}
btn.setOnAction(new ClickHandler());
```

**ANONIMNE METODE:** anonimna metoda je metoda koja nema ime i koju zadajemo lambda izrazima: (lista parametara) -> telo\_metoda, gde je telo metoda izraz ili blok naredba, tipovi parametara se mogu izostaviti ukoliko kompajler može sam da ih zaključi (*type inference*), i zagrade u listi parametara se mogu izostaviti ukoliko imamo jedan parametar naveden bez tipa:

```
(String s, char c) -> s.charAt(0) == c
(s, c) -> s.charAt(0) == c
```

Funkcijski interfejs je interfejs koji propisuje implementaciju tačno jednog metoda; ako je *x* promenljiva tipa *x* pri čemu je *x* funkcijski interfejs tada promenljivoj *x* možemo dodeliti anonimni metod gde je *x* – referenca na funkcijski objekat; interfejs `EventHandler<T extends Event>` je primer f-jskog interfejsa:

```
EvtHnd<AcEvent> h = (ActionEvent e) -> {...}
EventHandler<ActionEvent> h = e -> {...}
btn.setOnAction(h)
btn.setOnAction(e -> {...})
```

Konačno, referencijalni lambda izraz predstavlja anonimni metod koji samo pozove neki konkretan metod za sopstvene argumente:  $(x_1, x_2, \dots, x_k) \rightarrow a.m(x_1, x_2, \dots, x_k)$ :

```
pub class JavaFxApp
extends Application {
    priv Button btn = newButton("Click me");
    public void start(Stage stage)
    throws Exception {
        btn.setOnAction(this::handleClick);
        ... }
    private void handleClick(ActionEvent e) {
        ... } }

```

Ovakvi lambda izrazi se skraćeno mogu navesti kao *a::m*, pri čemu je *a* objekat (*m* – nestatički metod) ili klasa (*m* – statički metod).

37

Interfejsi Collection i Map.

**KOLEKCIJE:** u pitanju je objekat koji sadrži druge objekte: liste, skupovi, stekovi, redovi opsluživanja; mapa je kolekcija parova (ključ, vrednost) u kojoj nema duplikata ključeva;

interfejsi i klase koje realizuju standardne kolekcije su u paketu `java.util`; centralni interfejsi iz paketa `java.util`: `Collection`, `Set`, `List`, `Queue`, `SortedSet`, `Deque`, `Map`, `SortedMap`.

**INTERFEJS `Collection<E>`:** generički interfejs koji definiše sledeće operacije:

<code>boolean add(E e)</code>	<code>boolean contains</code>	<code>int size()</code>
<code>void clear()</code>	<code>(Object o)</code>	<code>Iterator&lt;E&gt; iterator()</code>
<code>boolean remove(Object o)</code>	<code>boolean isEmpty()</code>	

**`Iterator<E>` je generički interfejs koji definiše sledeće operacije:**

<code>boolean hasNext()</code>	<code>E next()</code>	<code>void remove()</code>
--------------------------------	-----------------------	----------------------------

`Iterator<E>` briše iz kolekcije poslednji element isporučen kroz iterator; ponašanje iteratora nije specifikirano ako se iz kolekcije uklanjaju elementi dok se kroz kolekciju iterira osim koristeći `remove()` metod.

**KLASA `Collections` iz `java.util`:** metode:

- 1) `Collections.sort(List<E> l)` – sortiranje po prirodnom uređenju, elementi liste su objekti klase koja implementira `Comparable`
- 2) `Collections.sort(List<E> l, Comparator<E> cmp)` – sortiranje po proizvoljnom komparatoru
- 3) `Collections.binarySearch(List<E> l, E key)` – binarno pretraživanje, elementi liste su objekti klase koja implementira `Comparable`
- 4) `Collections.copy(List<E> dst, List<E> src)` - kopiranje liste `dst` u listu `src`
- 5) `Collections.shuffle(List<E> src)` - izmeša elemente liste

38

Skupovi - interfejsi i klase.

**INTERFEJS `Set`:** reč je o generičkim interfejsima koji opisuju operacije u radu sa skupovima – operacije iz interfejsa `Collection`, `add(E e)` uspeva samo ukoliko u skupu ne postoji `x` takvo da je `x.equals(e)` tačno.

**KLASE `HashSet` i `LinkedHashSet`:** implementiraju interfejs `Set`: **1)** `HashSet` – skup realizovan *hash* tabelom (otvoreno *hash*-ovanje), **2)** `LinkedHashSet` – skup realizovan istovremeno i *hash* tabelom i listom (može se iterirati u redosledu umetanja).

**INTERFEJS `SortedSet`:** skup kod kog su elementi sortirani po prirodnom uređenju ili po proizvoljnom komparatoru (iteriranje po poretku); postoje metode koje vraćaju *min* i *max* element skupa, podskup koji sadrži sve elemente veće/manje od zadatog elementa, podskup koji sadrži sve elemente u nekom intervalu.

**KLASA `TreeSet`:** implementira interfejs `SortedSet`.

39

Liste - interfejsi i klase.

**INTERFEJS `List<E>`:** lista je (uređena kolekcija) sekvenca elemenata kojima se pristupa preko indeksa; metode definisane interfejsom:

<code>boolean add(int index, Object o)</code>	<code>void set(int index, E element)</code>
<code>E get(int index)</code>	<code>ListIterator&lt;E&gt; listIterator()</code>
<code>int indexOf(Object o)</code>	<code>hasNext(), next(),</code>
<code>E remove(int index)</code>	<code>hasPrevious(), previous()</code>

**KLASE ArrayList i LinkedList:** implementiraju List: **1)** ArrayList – sekvenca realizovana dinamički proširivim nizom, **2)** LinkedList – sekvenca realizovana dvostruko-povezanom listom (kod obe klase metod add() dodaje na kraj liste); ArrayList je efikasnija kada se elementima pristupa preko indeksa i kada se elementi dodaju/brišu sa kraja, u svim ostalim slučajevima je efikasnija LinkedList; klasa LinkedList takođe implementira interfejs Deque (double ended queue):

```
addFirst      removeFirst      addLast      removeLast
```

Stoga, LinkedList može se koristiti i kao stek (*last in first out*) i queue (*first in first out*).

40

Mape - interfejsi i klase.

**INTERFEJS Map<K, V>:** opisuje operacije u radu sa mapama:

```
V put(K key, V value)
V get(Object key)
V remove(Object key)
Set<Map.Entry<K, V>> entrySet();
interface Entry { K getKey(); V getVal(); }
```

Skup pogled na mapu, operacije nad skupom utiču na mapu i obratno.

**KLASE HashMap i LinkedHashMap:** implementiraju interfejs Map: **1)** HashMap – mapa realizovana hash tabelom (otvoreno hash-ovanje), **2)** LinkedHashMap – mapa realizovana istovremeno i hash tabelom i listom (može se iterirati u redosledu umetanja).

**INTERFEJS SortedMap:** nasleđuje Map i dodaje sledeće operacije: vraćanje *min* i *max* ključa, kreiranje podmape koja sadrži sve ključeve veće/manje od nekog ključa, kreiranje podmape koja sadrži sve ključeve u nekom intervalu.

**KLASA TreeMap:** implementira interfejs SortedMap.

## T

## Teze.

- 1) Polje bez modifikatora vidljivosti (default) vidljivo je u svim klasama istog paketa.
- 2) Apstraktna klasa ne mora da sadrži nijedan apstraktan metod; ako klasa sadrži bar jedan apstraktan metod mora i sama biti apstraktna.
- 3) `String x = 1 + 2 + "Pera" + 3 + 4 → x == "3Pera34"`
- 4) Nabrojivi tipovi mogu implementirati interfejs i ne mogu nasleđivati klase.
- 5) Svaka klasa unutar interfejsa je implicitno `public static`.
- 6) Klasa ne može istovremeno biti `public` i `protected`.
- 7) Ako klasa nije `abstract` mora imati telo.
- 8) Klasa `String` se ne može nasleđivati.
- 9) Anonimna klasa koja implementira interfejs `X` se instancira: `X x = new X() { .. }`; - obavezno se implementira sve što interfejs propisuje.
- 10) Ako metod baca `Exception` on može da baca i `RuntimeException` budući da je `Exception` nadklasa; obrnuto ne važi – ako metod baca `RuntimeException` on može da baca samo *unchecked* izuzetke.
- 11) Ako u klasi metod `m1` baca *checked* izuzetak (`m1() throws Exception`), metod `m2` poziva metod `m1` i niti obradi, niti prosledi izuzetak, takva klasa se ne kompajlira.
- 12) Korišćenje interfejsa `Iterator` sa listom: `Iterator<T> it = lista.iterator();` – nakon toga se iterira sa `while(it.hasNext()) { .. }`
- 13) Kod naredbi oblika `a = b + c` prvo se evaluira vrednost desno od operatora dodele i generiše bilo kakav izuzetak koji tu može da se generiše, a zatim se prelazi na operator dodele koji takođe može da generiše neki drugi izuzetak.
- 14) Kada klasa `B` nasleđuje `A` i `A` ima polje `x`, tada se polju `x` klase `B` pristupa ili direktno sa `x` ili sa `super.x` – oba načina funkcionišu.
- 15) `HBox`, `VBox`, `FlowPane` – smeštaju proizvoljno mnogo čvorova, `BorderPane` – smešta maksimalno pet čvorova.
- 16) Konstruktor *enum*-a je uvek `private`.
- 17) Kada podklasa ima metod istog imena kao i nadklasa, tada ona mora imati ili različit broj parametara ili bar jedan parametar mora biti različitog tipa, a ako to nije slučaj tada metod u podklasi mora propisno redefinisati metod i to tako što će zadržati isti tip povratne vrednosti.
- 18) Podklasa ne može da pristupa `private` poljima nadklase.
- 19) Dinamičko vezivanje podrazumeva da čak i metoda koja je definisana samo u nadklasi poziva metod iz podklase ukoliko je u vremenu izvršavanja promenljiva tipa podklase.
- 20) Kod naredbe `A a = new B();` gde `B` nasleđuje `A` deo `new B()` je instanciranje i zato klasa `B` ne može biti apstraktna; takođe, smer nasleđivanja mora ići `A → B`.
- 21) Ako se jedan interfejs nasleđuje / implementira “na dve strane” (npr. `Interface I1 extends I0` i `Interface I2 extends I0`) i ako se definiše promenljiva koja je tipa `I2` ona je takođe i tipa `I0`, ali nikako ne i tipa `I1` – drugim rečima, „preuzimanje tipova“ ide samo do „korena“ nasleđivanja.
- 22) Operator `instanceof` slično kao operatori `<`, `>`, `==` i slični nema asocijativnost (`a instanceof b instanceof c` ne prolazi kompajliranje).
- 23) Apstraktna klasa može sadržati definiciju neapstraktne metode, pa samim tim i `final` metode.

24) Ako `class A implements I` i zatim `class B extends A`, tada klasa B nasleđuje sve metode iz `I` koje implementira A pa ne mora eksplicitno da stoji `class B extends A implements I`.

25) Budući da nema ime, anonimnu klasu nijedna klasa ne nasleđuje niti ona može da ima konstruktore.

26) Klasa može biti apstraktna bez obzira koju i kakvu klasu nasleđuje, uključujući `Exception`.

27) Bez obzira koju klasu nasleđuje svaka klasa može implementirati proizvoljno mnogo interfejsa.

28) Neproveravani izuzeci ne moraju se navoditi u zaglavlju metoda.

29) `BufferedReader` i `FileReader` generišu *checked* izuzetke `FileNotFoundException` i `IOException` – ovi izuzeci se uvek moraju obraditi (`try-catch-finally`, `try-with-resources`) ili proslediti dalje (navesti u zaglavlju metode).