PROGRAMSKI JEZIK
JAVA

⇒ komentiranje koda:

a) //... ⇒ jedan red komentara

b) /* .... */ ⇒ komentar kraj - početak

c) /** .... */ ⇒ „javadoc komentari" - pišu se na engleskom i služe za generiranje dokumentacije (vrijedi za pojedine metode)

⇒ mehanizmi obrade ulaznih podataka:

↳ razred: java.util.Scanner

↳ korištenje:

Scanner sc = new Scanner(System.in);

↳ najava ulaznog toka (System.in je standardni ulazni tok)

⇒ mehanizmi obrade izlaznih podataka:

System.out.printf("...", ...);

↳ vrlo slično „printf"-u u C-u

System.out.format("...", ...);

↳ vidi dokumentaciju

⇒ NAPOMENA: standardne tokove možemo zatvoriti samo jednom jer ih ne možemo ponovno otvoriti (to je jer javin virtualni stroj vrši otvaranje standardnih tokova, a ne mi samo - mi radimo

samo najavu tokova pod željenim imenom)

⇒ grupiranje podataka („strukture"):

↳ u C-u moramo najavit strukturu i moramo se bavit s alociranjem memorije (i pointerima)

↳ u javi, koristimo klase:

```
class X
    {....}
X var = new X();
```

↳ u javi nema objekata niti struktura već je sve sadržano u tipu **class**

⇒ u javi nema pokazivača, al imamo „reference" koje su iste kao pointeri samo bez mogućnosti vršenja aritmetičkih operacija s njima

KATEGORIZACIJA

⇒ bitna stavka objektno orijentiranog programiranja jer čovjeku bitno olakšava rad s podacima

⇒ program koji razvijamo ima neku „domenu" (ono na što se program odnosi) i sadrži svoje kategorije (klase):

FAKULTET (domena) → nastavnici, → studenti, → predmet } klase

⇒ **klasa** - „nacrt" stvaranja nekog objekta
⇒ objekt - jedan primjerak (instanca) klase

⇒ kako bi baratali sa strukturama podataka u C-u, trebamo raditi funkcije koje su specijalizirane (npr. student_new), al još nismo raščistili da netko preko pointera pristupi podatku i onda ga izmjeni (namjerno ili nenamjerno)

↳ u javi, takve promjene nisu moguće, već se podaci (objekt) mijenjaju samo preko danih funkcija - to je ENKAPSULACIJA

↳ ovo je bio povod stvaranju objektno orijentiranih programskih jezika

↳ također, razmišljanje u objektno orijentiranom programskom jeziku vrši se u delnama

koda, a ne u pojedinačnim naredbama - to je APSTRAKCIJA

⇒ NAPOMENA: enkapsulaciju je moguće izvest i u C-u pomoću „void" pokazivača, al je to prilično mukotrpno i nespretno

⇒ klase su zapravo poopćenje C-ove strukture podataka („struct")

↳ stoga, <u>**klasa (class)**</u> je struktura podataka koja osim članskih varijabl (**atributi**) ima i vlastite pripadne funkcije (**metode**) te nudi i kontrolu pristupa (tko smije mijenjat što)

⇒ NAPOMENA: objektni jezici C-ovske sintakse (C++, C#) imaju i klase i funkcije (razlika je u smještaju podataka po stogu i heap-u)

⇒ u javi ne moramo navesti na početku kolko je polje veliko (slično dinamički alociranom polju)

⇒ **static** - statične metode koje nisu povezane s nekim određenim objektom te njih moramo koristit na način da ispred navedemo naziv klase

⇒ **final** - ključna riječ koja kaže da se vrijednost varijable do kraja programa neće promijenit (npr. final int i = 5;)

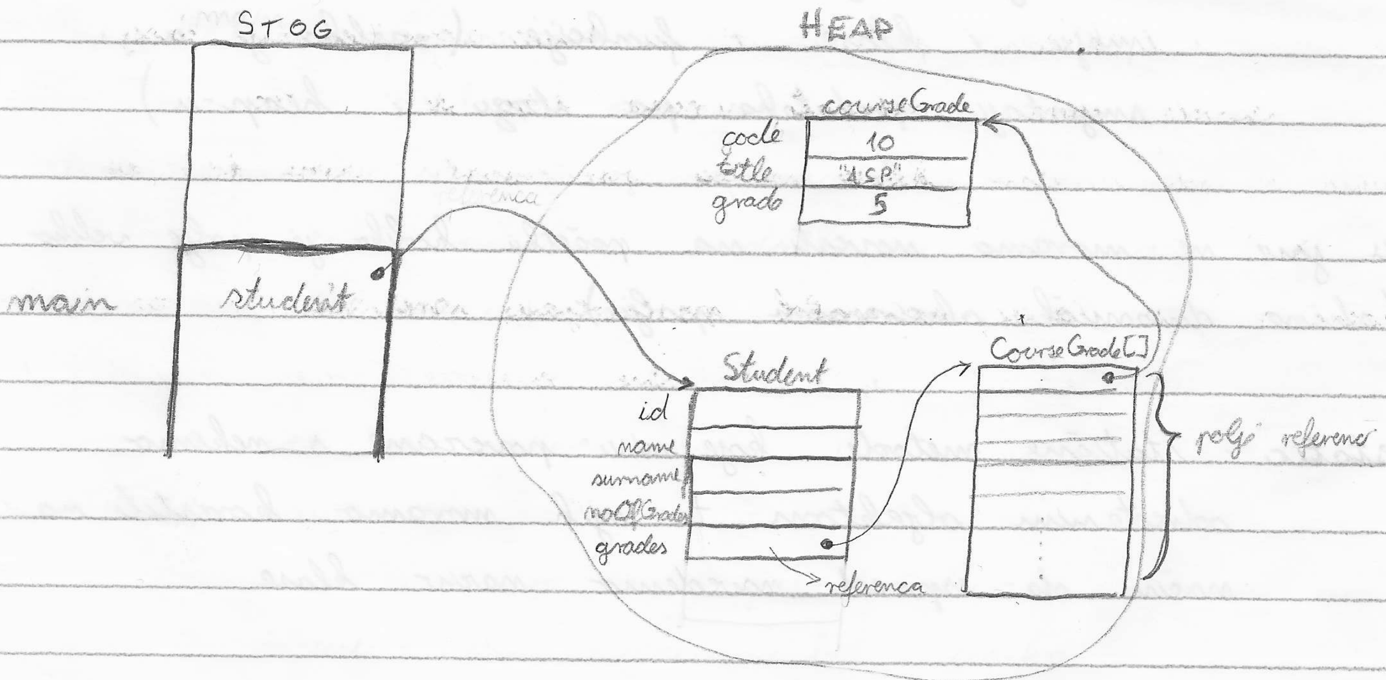
⇒ učahurivanjem se postiže slaba povezanost objekata što je dobro jer interna promjena jednog objekta (klase) ne utječe na rad drugog

↳ kod toga je bitno da se „interface" ne mijenja (atribut i metode)

⇒ kada radimo novi razred, mi ne možemo samo alocirat memoriju već pišemo ovako:

```
CourseGrade courseGrade = new CourseGrade();
```

referenca koja se stvara na stogu, a na heap-u se rezervira mjesto za sve atribute objekta courseGrade



⇒ „new" uvijek vraća inicijalizirane vrijednosti na nula ili null

VIDLJIVOST

⇒ ograničavanje vidljivost (pristupa) vrši se zbog sigurnost i konzistentnost koda

⇒ klasa koja sadrži „main" i sam „main" mora biti public

⇒ MODIFIKATORI:

a) package private (prazno / blank)
b) public
c) private
d) protected

⇒ sve metode klase mogu pristupat svemu definiranome u svojem paketu (ako nemaju ništa ispred imena - blank)
⇒ pomoću modifikatora postižemo učahurivanje / enkapsulaciju
⇒ sve metode mogu pristupati svemu što je public
⇒ samo metode unutar klase mogu pristupat svemu što je private
⇒ uobičajeno je da se varijable postavljaju private, a za njihovo čitanje i pisanje se koriste pripadne metode (getteri i setteri)

KORISNO O JAVI

⇒ sve što nisu primitivni tipovi (int, double, char, ...) java tretira kao reference
↳ slanjem objekta u funkciju mi zapravo šaljemo samo referencu (tako se na stog ne kopira masivni komad memorije)

⇒ sve što ima malo početno slovo je po konvenciji referenca

⇒ „new" vraća referencu na dio memorije koji je alociran za željeni objekt
↳ taj dio memorije je po default-u inicijaliziran na nulu ili „null" - ovisno o tipu

⇒ modifikator „vršnog" (vanjskog) razreda može imati samo „public" ili „blank" modifikator

⇒ javno sučelje razreda - „ono što korisnik vidi i može koristiti"
- javne metode i atributi

⇒ sve vidi sve u svojem razredu - modifikatori se odnose na ono što nije u pripadnom razredu

⇒ PRIVATE ⇒ samo ista klasa vidi

⇒ BLANK ⇒ package - private

↳ isti razred (naravno) i svi razredi u istom paketu

⇒ PROTECTED ⇒ vide svi u istom paketu i klase koje nasljeđuju tu klasu (razred)

⇒ PUBLIC ⇒ svi imaju pristup ovome

⇒ po dogovoru, SVE članske varijable (atributi) ćemo stavljati PRIVATE kako bi izbjegli neželjenu promjenu od „lošeg" korisnika

↳ stoga, njih ćemo mijenjati preko posebnih metoda za to: **getteri** i **setteri**

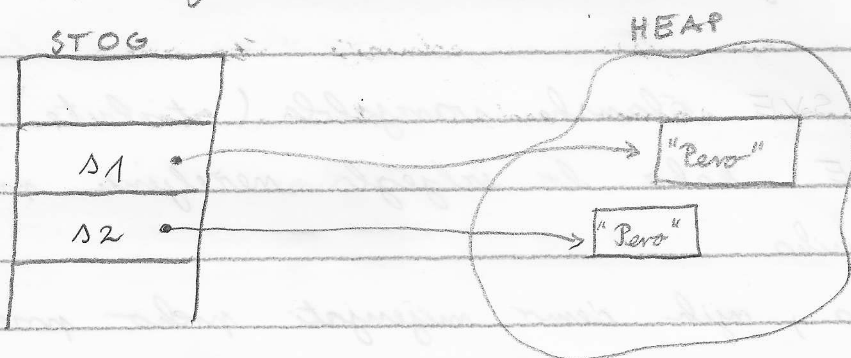
↳ getteri i setteri su uvijek tipa **void**

↳ getteri i setteri imaju i ulogu sprječavanja neželjenog unosa

DODATNO O UČAHURIVANJU

⇒ java definira razred Object koji je nadređen svima ostalima

⇒ NAPOMENA: operator ekvivalencije uspoređuje primitivne vrijednosti




```
String s1 = new String("Pero");
String s2 = new String("Pero");
```

↳ sada ako pišemo:

```
if(s1 == s2) { ... };
```

↳ neće ući u „if" jer ove reference nisu jednake, pa ako želimo ispitat jesu li stringovi jednaki pišemo:

```
if(s1.equals(s2) == true)
```

⇒ metoda „equals" radi nad svim objektima i ona je definirana u razredu Object

⇒ ako imamo neku metodu iz nekog „nadređenog" razreda te ju želimo preraditi, prije naše definicije nove metode u željenom razredu pišemo @Override

⇒ nama su najbitnije metode iz klase Object za rad:

a) equals()

b) toString()

⇒ ako napišemo dva ista konstantna stringa na dva mjesta u kodu te imamo pokazivače na njih - te dvije reference imaju iste vrijednosti što znači da je u memoriji stvoren samo jedan konstantan string

KONSTRUKTORI I APSTRAKCIJA

⇒ ako nakon stvaranja objekta automatski želimo stavit neku defaultnu vrijednost moramo napisat konstruktor

↳ ako nismo napisali konstruktor, kompiler će napisat defaultni konstruktor za nas

⇒ konstruktor je metoda koja nema povratni tip (nije void!) i ona prije rezervacije memorije („new") postavi vrijednost na željenu vrijednost

⇒ kada napišemo konstruktor, više nemamo defaultni konstruktor

⇒ ako želimo delegirat izvođenje nekog konstruktora nekom drugom konstruktoru, pišemo odmah u prvom retku novog konstruktora this(„model napisanog konstruktora")

⇒ konstruktori će bit korisni kod kasnijih objekata koji su „immutable" (nepromijenjivi) – takvi objekti su odlični kod višedretvenosti jer nije potrebno sinkronizirat takve objekte

⇒ u javi nema manualnog oslobađanja memorije – kada zauzmemo mjesto za objekt, mi ga ne možemo oslobodit

↳ java ima sustav GARBAGE COLLECTOR – ako je dokazano da program ne može pristupit nekom objektu (nema daljnje reference na objekt) taj se dio memorije oslobađa

```
class Point {
    double x;
    double y;
}
```
} varijable koje su specifične za svaki objekt posebno

⇒ STATIC VARIJABLE ⇒ varijable koje pripadaju svim objektima, zajedno, a ne svakom objektu posebno

⇒ možemo reći da static varijable pripadaju klasi, a ne objektu

⇒ takve varijable se ne spremaju u svaki objekt već se spremaju nekamo posebno - to mjesto je na zajedničkom mjestu i ono se alocira samo jednom (također na heapu)

⇒ STATIC METODE ⇒ metode koje ne pripadaju objektima, već pripadaju razredu

⇒ unutar statičke metode vidimo samo statičke varijable razreda, a ne ostale članske varijable

↳ samim time, nema smisla pisati „this" unutar statičke metode

PR:
```
class Matematika{
    public static double suma(double x, double y){
        return x + y;
    }

    public static double suma(double x, double ...ostali){
        // argument ostali je u kodu vidljiv kao polje
        double trenutnaSuma = x;
        for(int i = 0; i < ostali.length, i++){
            double trenutniPribrojnik = ostali[i]
            trenutnaSuma += trenutniPribrojnik;
        }
        return trenutnaSuma;
    }
}
```

⇒ NAPOMENA: varijabilni broj argumenata metode možemo pisati samo jednom i to na kraju liste argumenata

⇒ NAPOMENA: ako imamo dvije metode s istim imenom, ali različitim brojem argumenata, a jedna je s varijabilnim argumentima, uvijek se poziva prvo ona koja nije s varijabilnim argumentima (naravno, ako broj argumenata odgovara onoj metodi s fiksnim brojem argumenata

⇒ „for" petlja u drugoj metodi na prošloj stranici se može napisati na bolji način:

```
for (double trenutnaVrijednost : ostali) {
    trenutnaSuma += trenutnaVrijednost;
}
```

↳ ovo je puno manji problem za čitanje nečijeg koda

OGRANIČAVANJE

⇒ kako ograničit korisnika da može stvorit samo jedan primjerak razreda?

↳ to nam je korisno ako npr. imamo konfiguraciju našeg programa i to trebamo jednom

⇒ ovaj problem se naziva Singleton - problem ograničavanja na jedan primjerak

```
public class Singleton {
    private Singleton() {.....}  // privatni konstruktor
    private static Singleton primjerak;

    public static Singleton getInstance() {
        if( primjerak == null)
            primjerak = new Singleton();
        return primjerak;
    }
    ...... // ostale metode za primjerak
    ......
}
```

↳ ovo je najčešći način rješavanja ograničavanja na jedan (više) primjeraka i naziva se:

OBLIKOVNI OBRAZAC SINGLETON

PR: class Y {

```
public int K = 7;
}
```

↳ ovo baš nije korisno da svaki objekt ima svoju konstantu koja je jednaka za cijeli razred

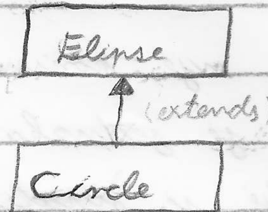
↳ stoga, bolje je pisati:

```
public static final int K = 7;
```

↳ na taj način, zauzeli smo samo jedno mjesto, a ne onoliko mjesta za konstantu koliko objekata smo napravili

NASLJEĐIVANJE

⇒ ključnja riječ extends




```
public Circle {
    super(radius, radius)
}
```

⇒ razred kružnica izveden je iz elipsa

⇒ razred kružnica je naslijedio razred elipsa te samim time nasljeđuje sve njegove metode

⇒ **super** - delegirao je konstrukciju na „nadređeni" razred

⇒ razred kružnica neće imati članske varijable već će imati konstruktor koji će delegirati nastajanje

⇒ krug nasljeđuje elipsu znači da je krug elipsa

⇒ možemo pisati:

```
Elipse e = new Circle(10);
```

↳ sada vidimo samo metode razreda Elipse

```
Circle c = new Circle(10);
```

↳ sada vidimo sve metode razreda Circle i razreda Elipse

⇒ ne možemo pisati:

```
Circle c2 = e;
```

↳ razlog je što kompiler ne može pridružiti referenci kruga referencu elipse

↳ stoga, moramo napraviti „castanje" reference na razred - naravno, razredi se moraju pravilno naslijeđivati:

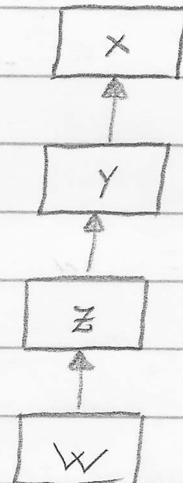
```
Circle c2 = (Circle) e;
```

↳ ako se ovdje desi pogreška dobijemo grešku: class_cast_exception

↳ najbolji način da se osiguramo od takve greške je ispitivanje pomoću „if":

```
if (e instanceof Circle) {
    Circle c3 = (Circle) e;
}
```

⇒ hijerarhija ovo izgleda ovako:




```
X x1 = new W();

if (x1 instanceof W) {...}
if (x1 instanceof Z) {...}
if (x1 instanceof Y) {...}
```

⇒ NAPOMENA: u gornjem primjeru, objekt x1 će imati sve članske varijable razreda W (i biti će stoga nešto veći)

⇒ ako želimo saznati je li nešto baš primjerak nekog razreda (različito od nasljeđivanja), pišemo:

```
if(x1.getClass().equals(Z.class)) {
    ...
}
```

⇒ compiler prilikom pokretanja u memoriji stvara primjerak (objekt) nekog konkretnog razreda u vršnom razredu Class, pa smo mogli pisati i ovako (taj objekt opisuje taj razred):

```
if(x1.getClass() == Z.class) {
    ...
}
```

PR: Način rada nasljeđivanja:

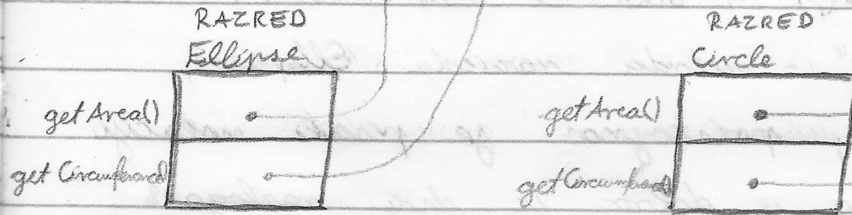
↳ konstruktor elipse: Ellipse(a, b){...}

↳ metode Elipse: getArea(){...}

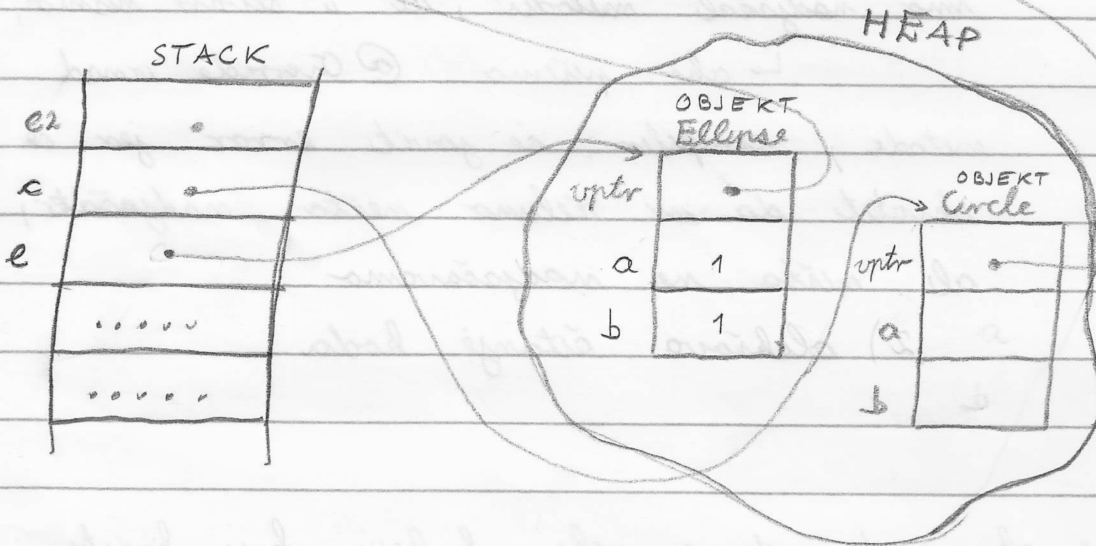
getCircumference(){...}

↳ konstruktor kruga: Circle(a){...}

ovo se kompelira u strojni kod negdje u memoriji



primjerci koji samo opisuju razrede - sadrže reference na memorijske lokacije gdje se nalaze pripadne funkcije razreda



↳ primjerci izvedenog razreda sadrže varijable (iste varijable) izvedenog razreda na istom odmaku od pokazivača na tablicu metoda (tablicu referenci metoda)

⇒ NAPOMENA: ako u razredu Circle definiramo novi metodu getCircumference() na drugi način za krugove, svaki put kada zovemo tu metodu, zvat će se ta funkcija, a ne funkcija definirana za razred Ellipse

↳ to se radi na način da se u tablici metoda promijeni pokazivač za metodu getCircumference(), i tako se NADJAČAVA metoda „odozgo" - metoda razreda Ellipse

↳ također, pristojno je pisati notaciju @Override - to je dobro iz dva razloga:

1) ako smo načinili grešku u pisanju i pogriješili u nazivu metode, mislit ćemo da smo nadjačali metodu, ali u stvari nismo,

↳ ako pišemo @Override iznad metode, compiler će javit error jer će shvatit da mi želimo nešto nadjačati, ali ništa ne nadjačavamo

2) olakšava čitanje koda

⇒ NAPOMENA: ako sada damo neku funkciju koja koristi elipsu nekom klijentu, on može mijenjat pomoću settera veličine poluosi, doveli smo objekt krug u nekonzistentno stanje jer pomoću settera početni krug može imati različite poluosi

↳ pogledaj LISCOVNO NAČELO SUPSTITUCIJE

PRIMJER: Imamo razred GeomLik koji sadrži elipsu, krug i pravokutnik i želimo u razredu GeomLik ponuditi neke metode za izračun površine i opsega.
↳ taj nadrazred nam služi kako bi mogli napisati „polje" likova:

```
public abstract class GeomLik {
    public abstract double getArea();
    public abstract double getCircumference();
}

public static void main (void) {

    GeomLik[] likovi = new GeomLik() {
        new Circle();
        new Rectangle();
    }
}
```

⇒ NAPOMENA: ne možemo raditi primjerke abstraktnih razreda jer compiler ne može napraviti tablicu metoda

⇒ NAPOMENA: apstraktne metode su samo prototip za sve razrede koji nasljeđuju razred s našom apstraktnom metodom

⇒ NAPOMENA: svaki razred koji sadrži apstraktnu metodu ne može popuniti tablicu funkcija svojih objekata, pa ne možemo kreirati objekte tog razreda te je samim time, razred mora biti apstraktan

⇒ NAPOMENA: ako razred nema ponuđene sve metode definirane kao „prototipi" (abstract) u nadređenom razredu, tada je i taj razred apstraktan jer se očekuje da svaki objekt mora imati sve te „predefinirane" apstraktne metode, pa se ne može ispuniti tablica funkcija koja je očekivana

SUČELJE

⇒ ključna riječ „interface"
  ↳ ne nalazi se u razredu

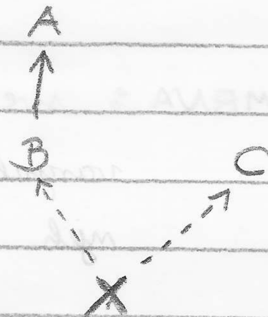
⇒ u sučelju definiramo popis funkcija koje se mogu koristiti (klasa mora ponuditi implementacije metoda)

⇒ sučelja možemo shvatiti kao POTPUNO ABSTRAKTNE KLASE

⇒ svaki razred koji implementira sučelje je apstraktan, osim ako ponudi implementacije svih metoda navedenih u sučelju

⇒ sučelja se mogu nadopunjavati / naslijeđivati:

```
interface A {
    void m1();
}
interface B extends A {
    void m2();
}
interface C {
    void m3();
}
```




```
class X implements B, C {
    m1() {....} , m2() { } , m3() { }
}
```

⇒ ako sada napišemo:

```
A a = new X();
B b = new X();
C c = new X();
X x = new X();
```

↳ sada u objektu „a" vidmo samo metode navedene u sučelju A

⇒ O KLJUČNOJ RIJEČI „final":

a) na razini metoda

↳ ta se metoda ne može nadjačat

b) na razini razreda

↳ taj se razred ne može naslijedivat

⇒ NAPOMENA: sve metode u sučeljima su javne i sve varijable u sučeljima su konstante iako ispred njih ne mora pisat „public final"

AUTO - BOXING

⇒ WRAPPERI - služe za korištenje metoda s primitivnim tipovima te služe za konverziju primitivnog tipa u objekt što je za neke metode potrebno

PR: Izgled razreda Integer:

```
class Integer {
    privat int value;
    public Integer(int value){
        this.value = value;
    }
    public int intValue() {
        return value;
    }
}
```

⇒ auto - boxing: compiler vidi primitivni tip s desne strane jednakosti, a objekt s lijeve strane te on automatski „radi" objekt i vraća referencu

⇒ auto - unboxing: suprotan postupak auto - boxingu

JAVA GENERICS

⇒ tehnologija koju koristimo kada želimo načinit dva gotovo ista razreda koji samo imaju razliku u tipu podataka

⇒ tada se radi parametrizacija
↳ npr. ako želimo stvoriti vlastite wrappere za već definirane primitivne tipove - ti wrapperi bi imali vrlo sličan kod, pa nam tu pomaže java generics

PR: Primjer Integer wrappera!

```
Wrapper<Integer> iNumber = new Wrapper<>(5);
```

PR: Parametrizacija i nasljeđivanje!

```
A<T>        C
 ↑          ↑
B<T>        D
```

B<C> je podtip od A<C> ⇒ VRIJEDI!
↳ ovo je jedini slučaj dobrog parametriziranog nasljeđivanja

B<D> je podtip od B<C> ⇒ NE VRIJEDI!

PR: Parametrizirani razred koji može „rasti"!

```
class GrowableArray<T> {

    private T[] elements;
    private int count;


    /* konstruktor */
    public GrowableArray(int initialCapacity) {
        elements = (T[]) new Object[initialCapacity];
        count = 0;
    }
    /* defaultni konstruktor */
    public GrowableArray(); {
        this(4);
    }

    /* neke metode → npr. size, getElementAt, add */
}
```

↳ na ovaj način smo stvorili dinamičko polje referenci koje možemo koristi s parametrizacijom

OKRENI

```
public class Main() {

    public static void main(void){

        GrowableArray <String> col = new GrowableArray<>();
        col.add("Ivana");
        col.add("Maja");
        cal.add("Marko");
        col.add("Juraj");
```

↳ kako sada iterirati po ovom polju?

↳ trebamo koristiti sučelje Iterable<T>

⇒ prije smo trebali ručno napraviti iterator i trčati po polju while petljom sve dok „iterator". has Next();

↳ od jave 5 nadalje možemo preskočiti izradu iteratora i koristiti:

for (..... : ...)

{ ... }

↳ da bi ove stvari funkcionirale, naš razred mora implementirati sučelje Iterable<T>

UGNJEŽĐENI RAZREDI

⇒ NAPOMENA: iterator radi sve dok netko nije modificirao polje po kojem se iterira

↳ to je dogovoreno ponašanje iteratora te je obavezno da naš iterator implementira takvo ponašanje

↳ to se najčešće implementira pomoću jednog „modificationCounter"-a koji

⇒ ključna riječ „**enum**"

↳ definira razred koji definira konstante

Pr:
```
public enum CarType {
    DIESEL, PETROL;
}
```

↳ objekt CarType razreda može poprimiti samo dvije vrijednost koje su preddefinirane

ANONIMNI RAZREDI

⇒ oblikovni obrazac STRATEGIJA:

↳ npr. imamo listu automobila i što ako želimo isprogramirat filter funkciju koja ispisuje automobile koji ispunjavaju automobile pod zadanim uvjetima:

↳ rješenje:

```
interface Tester {
    boolean testiraj(Car c);
}

class SkupAuto implements Tester {
    public boolean testiraj(Car c) {
        return c.getPrice() > 100 000;
    }
}

void printOut(ArrayList<Car> cars, Tester t) {
    for (Car c : cars) {
        if (t.testiraj(c)) {
            Syso(c);    // System.out ...
        }
    }
}
```

⇒ ovakva implementacija je „stacionarna" i ispituje po samo jednom parametru - kako bi ovo izbjegli koristimo parametrizaciju

⇒ ANONIMNI RAZREDI:

↳ koristi se samo u jednom mjestu u kodu

↳ po definiciji, to je razred čije ime ne znamo

↳ na mjestu tog razreda odmah pišemo i deklaraciju i definiciju tog razreda

↳ anonimni razredi obično imaju jednostavan kod

⇒ mi nećemo raditi obrazac STRATEGIJA na naš način jer java već ima implementirano sučelje (parametrizirano):

```
interface Predicate<T> {
    boolean test(T object);
}
```

↳ sada bi naš razred pisao ovako:

```
class SkupAuto implements Predicate<Car> {
    public boolean test(Car car) {
        return car.getPrice >= 100 000;
    }
}
```

⇒ kako mi nebi stvorili poseban razred za svaku uvjet (npr. CarDiesel, CarPink, CarExpensive, ...), mi koristimo anonimni razred na mjestu gdje nam je to potrebno:

```
printCars( cars, new Predicate<Car>() {
                     public boolean test(Car c) {
                         return c.getPrice() >= 100 000;
                     }
                 }
);
```

↳ da smo gore umjesto sučelja „Predicate" naveli neki razred, mi bi stvorili anonimni razred koji nasljeđuje razred koji smo naveli

↳ na način na koji je gore pisano, mi stvaramo anonimni razred koji implementira sučelje Predicate

PR: Imamo dvije identične metode koje vraćaju polje doublova. Jedina razlika u metodama je izračun svakog člana.

↳ kako bi izbjegli dvije metode koje se razlikuju u samo jednoj liniji koda, ovo modeliramo sučeljem:

```
interface Transformator {
    double transformiraj(double vrijednost);
}
```

↳ sada svaki razred ima svoju implementaciju metode transformiraj, ali početna metoda koja vraća double treba primati i referencu na razred koji implementira sučelje Transformator

↳ ovo je tipičan oblik **oblikovnog obrasca strategija**

LAMBDA IZRAZI

⇒ sintaktna poboljšanja koja služe poboljšanju čitljivost koda

⇒ uvedeni su u Javi 8

⇒ npr. skraćuju anonimne razrede koji implementiraju jednostavne funkcije

⇒ NAPOMENA: lambde ne uvode novi doseg što znači da se „this" odnosi na vanjski razred, a ne na anonimni razred na tome mjestu

PR: Lambda izraz za ispisivanje:

```
public void accept(Car c) {
    Syso("Ako je" + c);
}
```

```
c → Syso(c)
```

LOKALNI RAZRED

⇒ koristi se kada nam treba samo na samo jednom mjestu u nekoj metodi često iz razloga što nam to odgovara semantički

PR: Lokalni razred u metodi za izračun udaljenost točaka

```
static double distance (int x1, int y1, int x2, int y2){

        class Point {
            int x;
            int y;
        }

        Point aPoint = new Point();
        aPoint.x = x1;
        aPoint.y = y1;

        /* isto napravimo za drugu točku i izračunamo
        udaljenost */

}
```

IZNIMKE

⇒ NAPOMENA: ako npr. svaki put umjesto Math. sqrt(9) želimo pisati samo sqrt(9), moramo načiniti statički import:

```
import static java.lang.Math.*;
```
↳ uključuje sve iz razreda „Math"

```
import static java.lang.Math.sqrt;
```
↳ uključuje samo metodu „sqrt"

↳ na ovaj način compiler zna na što mislimo kada napišemo samo „sqrt"

⇒ IZNIMKE:

↳ iznimke su također razred i rad sa njima se svodi na slično kao i kod rada s razredima
↳ iznimke služe za prekidanje i opisivanje nekog problema – prekida se cijela dretva (ako se iznimka ne obradi)
↳ iznimke se obično u modernim aplikacijama zapisuju u log datoteku

```
RuntimeException opisIznimke = new IllegalArgumentException(
        "Funkciji je nedozvoljeno predati takav argument"
    );
throw opisIznimke;
```

⇒ obrada iznimke:

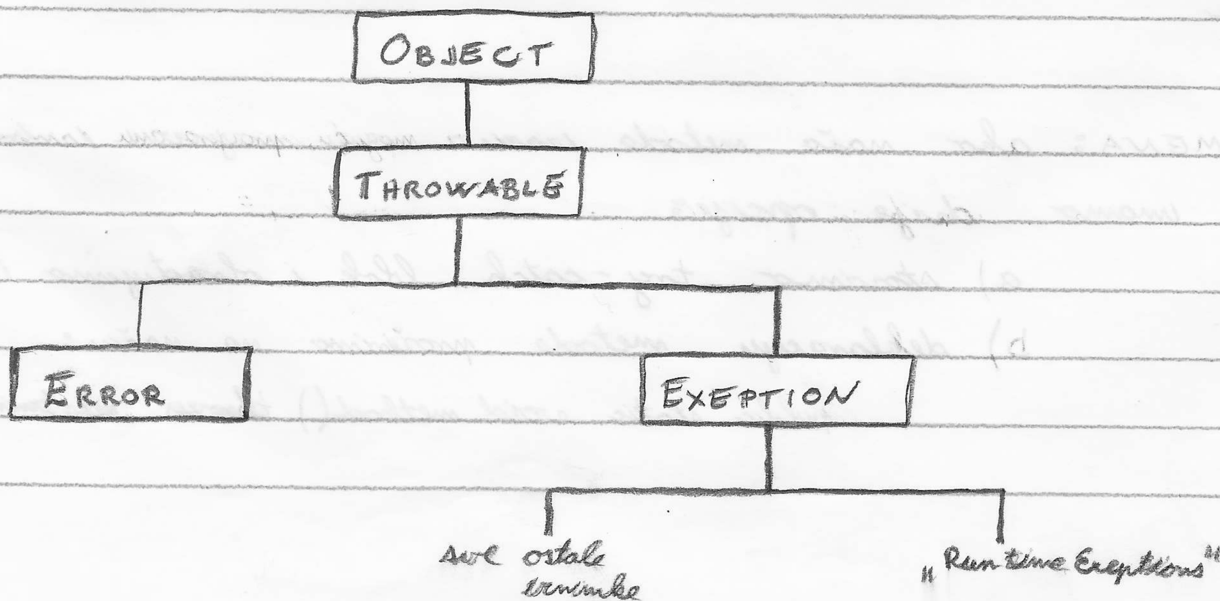
a) može ju obradit javin virtualni stroj te on veli – „imamo iznimku, „štampaj" kod – ruši trenutnu dretvu"

b) obraduje ju try {
⋮
} catch (....) {
⋮
}

↳ gleda se postoji li „catch" blok u trenutnoj metodi, ako ne, izlazimo iz trenutne metode i gledamo da li u pozivateljima postoji „try-catch" blok sa traženim tipom iznimke

⇒ iznimke su veliko poboljšanje u odnosu na klasično upravljanje pogreškama – izbjegli smo mnoštvo „if" blokova (i to ugnježđenih!)

⇒ iznimna situacija je opisana objektom – IZNIMKOM



⇒ pri poretku catch blokova, bitno je da pazimo na nasljeđivanje
↳ općenitiji „catch" blok uvijek mora doći poslije onog manje općenitog

⇒ finally blok - kako god završi try-catch blok, finally blok će se izvršiti

⇒ napomena: prije garbage collectora, iznimke su bile opasne za koristit jer ako bi izgubil referencu na nešto, imali bi memory leak

⇒ PODJELA IZNIMNIH SITUACIJA:
a) grube pogreške (errors)
↳ npr. „ostao sam bez memorije"
b) neprovjeravane iznimke (runtime exeptions)
c) provjeravane iznimke (checked exeptions)

⇒ neprovjeravane iznimke može izazvat gotovo svaka linija koda

⇒ NAPOMENA: ako naša metoda izaziva moguću provjeravanu iznimku, imamo dvije opcije:
a) stavimo try-catch blok i obradimo iznimku
b) deklaraciju metode proširimo na način:
public static void method() throws „neka Iznimka" {

⇒ ako imamo neke resurse u try bloku, java automatski u finally bloku generira kod koji suprotnim redoslijedom zatvara otvorene resurse
↳ ako nema finally bloka, on se implicitno generira

OOP - 2015 POKAZNI MEĐUISPIT

① 1) prvo razmišljamo kakav treba biti prototip:

```
public static <T> void task1(Iterable<T> iter,
        Predicate<T> predicate, Consumer<T> action)
```

2) sada pogledajmo kako će izgledat poziv:

```
Exam.task1(list, new Predicate<String>() {
        public boolean test(String str) {
            return str.length() == 1;
        }
    }, new Consumer<String>() {
        public void accept(String str) {
            dest.add(str);
        }
    });
```

↳ pisano preko lambda izraza:

```
Exam.task1(list, str -> str.length() == 1,
        str -> dest.add(str))
```

3) implementacja metode task1:

```java
public static <T> void task1(Iterable<T> iter,
        Predicate<T> predicate, Consumer<T> action){

    Set<T> set = new HashSet<>();

    for(T elem : iter){
        boolean nowy = set.add(elem);

        if(!nowy)
            continue;

        if(!predicate.test(elem))
            continue;

        action.accept(elem);
    }
}
```

②

```
public class PrimeNumber implements Iterable<Integer> {

    private int a, b;

    public PrimeNumber(int a, int b) {
        super();
        this.a = a;
        this.b = b;
    }

    // propisano sučeljem Iterable:
    public Iterator<Integer> iterator() {
        return new PrimeIter();
    }
```

/* za iterator koristimo nestatički ugniježđen razred (mogli smo i statički, ali ovako imamo pristup članskim varijablama i konstruktoru) */

```
private class PrimeIter implements Iterator<Integer>{

    private boolean imamBroj = false;
    private int kandidat = a;

    // propisano sučeljem Iterator:
    public boolean hasNext() {
        if(imamBroj)
            return true;
        for( ; kandidat <= b ; kandidat++) {
            if( jeProstBroj(kandidat)) {
                imamBroj = true;
                break;
            }
        }
        return imamBroj;
    }

    public Integer next() {
        if(!hasNext)
            throw new NoSuchElementException("Nema!");
        int broj = kandidat; imamBroj = false;
        kandidat++;
        return broj;
    }
```

```
//metoda jeProstBroj koji smo trebali ranije:
private boolean jeProstBroj( int broj) {
    for ( int i = 2, n = (int)(Math.sqrt(broj)+0.5); i<=n; i++)
        if ( broj % 2 == 0)
            return false;
        }
    }
}
}
```

③ public class CardDealer {

```
public static <T> Map<String, Set<T>> dealCards(
                Set<String> players,
                Collection<T> cards) {

    /*prvo treba ovo permutirat, a da bi to mogli
    prvo kolekciju treba pretvorit u listu kako bi
    mogli znati tko je na kojem mjestu */

    List<T> spilKarata = new ArrayList<T>(cards);
    Collections.shuffle(spilKarata);

    /*mapa nam je uređen par igrača i karata te
    to trebamo i vratit */

    Map<String, Set<T>> mapa = new HashMap<>();

    /*trebamo i špil (podijeljene karte) za svakog
    igrača posebno, pa radimo listu setova */

    List<Set<T>> spiloviIgraca = new ArrayList<>();

    /* stvorimo špil za svakog igrača */
```

OKRENI →

```
for ( String  igrac :  players) {
    Set <T>   spil = new  HashSet<>();
    mapa. put( igrac,  spil);
    spiloviIgraca . add ( spil);
}
/* brojač igrača - služi za kretanje po igračima u krug */
int  brojIgraca = 0;
/* krećemo se po kartama i dijelimo kartu po kartu */
for ( T  karta :  spilKarata)
    spiloviIgraca.get ( brojIgraca % players. size()).
            add ( karta);
    brojIgraca ++;
}
}
}
```

⑤ Napomena o zatvaranju i otvaranju resursa prilikom iznimaka:

PR:

```
try ( A a1 = new A(1) ) {
    ...
} catch (...) {
    ...
} finally {
    ...
}
```

„a1" će se zatvoriti po izlasku iz bloka „try" jer je navoljen u oblim zagradama

finally → uvijek se izvodi

⇒ rješenje zadatka:

<u>ISPIS:</u>

ctor 1
ctor 2
ctor 3
inner exc
finally 1
ctor 4
close 4
finally 2
close 1
exception
outer finally

④ Zbog razumijevanja ovakvih zadataka, poželjno je skicirati ono što je zadano:

Ⓘ Game Equipment Ⓘ Card Rank Ⓘ Card Suite

Ⓐ Card

Card Rank rank;

Card Suite suite;

get Suite();

get Rank();

Ⓒ Italian Card Hungarian Card French Card

⇒ nadalje iz zadatka vidimo da:

Ⓐ Game <T extends Game Equipment>

protected broj Rekvizita : int

protected players : Set <String>

" **konstruktor** " { iznimke }

public Set <String> set Winners();

Card Game < T extends Card >
„konstruktor"(brojkorata, Set < String > igraci){
super (brojkorata, igraci);
}
deal Cards (...);

Tressete < Italian Card >
static list < Italian Card > spil;
static method {
spil = new ArrayList < > ();
for (Italian Card . Suites boja :
Italian Card . Suites . values ()) {
for (Italian Card . Ranks oznaka :
Italian Card . Ranks . values ()) {
spil . add (new Italian Card (boja, oznaka));
}
}

UPORABA VLASTITIH RAZREDA S JAVINIM OKVIROM KOLEKCIJA

⇒ implementacije kolekcija obično se oslanjaju na dvije metode:

a) boolean equals(Object o);
b) int hashCode();

⇒ u razredu Object, metoda equals je definirana ovako:

```
class Object {
    ⋮
    public boolean equals(Object o) {
        return this == o;
    }
    ⋮
}
```

⇒ u razredu String, metoda equals je definirana ovako:

```
class String {
    private char[] elems;
    public boolean equals(Object o) {
        if(!(o instanceof String))
            return false;
        String s = (String) o;
        if(elems.length() != s.elems.length())
            return false;
```

```
        for (int i = 0; i < elems.length(); i++) {
            if ( elems[i] != s.elems[i])
                return false
        }
        return true;
    }
    ⋮
}
```

⇒ postoje stoga dvije jednakost:

a) semantička - upravo takva je ostvarena u razredu String

b) memorijska - dva objekta su jednaka ako se nalaze na istoj memorijskoj lokaciji

⇒ na isti način trebamo razmišljat kada implementiramo metodu **hash Code ()** → ako smo definiral semantičku jednakost, dva jednaka objekta (semantičk jednaka) trebaju imati isti „hash" - sažetak

↳ npr. u razredu String se koristi svaki pojedni element u izračunavanju sažetka

⇒ sučelja Comparable i Comparator se oslanjaju na ove implementacije metoda **equals** i **hash Code**

⇒ potrebno je proučiti lambda izraz za tako zvani „Method Handle":

Student :: menu

⇒ kada god radimo sa Javinim okvirom kolekcija, moramo nadjačat metodu **equals** osim ako nam odgovara memorijska jednakost

⇒ uoči da se pomoću metode hashCode() može ispitat jesu li objekti različiti - ako su im sažeta različiti, onda su sigurno i sami objekti različiti

↳ obrat ne vrijedi

⇒ BITNA KONVENCIJA:

↳ ako su dva objekta jednaka, oni nužno moraju imati isti sažetak

↳ stoga, skup varijabli koje se uzimaju za izračun jednakosti mora biti nadskup varijablama koje se uzimaju za izračun sažetka (ili jednak skup)

KOMPARATORI

⇒ npr. kako bi TreeSet znao graditi stablo objekata, svaki se objekt treba znati usporediti sa objektom istog razreda

↳ Java nudi da takav objekt implementira sučelje Comparable:

```
interface Comparable <T> {
    int compareTo( T other);
}
```

⇒ od metode compareTo(...) očekuje se:

„< 0" ⇒ this < other

„> 0" ⇒ this > other

„= 0" ⇒ this = other

⇒ naša implementacija komparatora kod razreda Student:

```
public class Student implements Comparable <Student> {
    ⋮
    public int compareTo (Student other) {
        return this.studentID.compareTo(other.studentID);
    }
    ⋮
}
```

⇒ kada implementiramo sučelje Comparable, kažemo da smo obavezni definirati PRIRODAN POREDAK OBJEKATA tog razreda

⇒ u Javi postoji i sučelje Comparator koje je vrlo slično sučelju Comparable:

```
interface Comparator <T> {
    int compare (T first, T second);
}
```

⇒ od metode compare(...) očekuje se:

"< 0" ⇒ first < second

"> 0" ⇒ first > second

"= 0" ⇒ first = second

⇒ DRUGA MOGUĆNOST:

↳ umjesto implementacije sučelja u objektu u kojemu želimo ostvariti usporedbu, napravimo novi razred (npr. StudentComparator) koji implementira sučelje Comparator

↳ tada, prilikom izgradnje nekog npr. TreeSet-a, moramo predati referencu na neki objekt koji zna uspoređivati objekte koji se nalaze u TreeSet-u (u našem slučaju, StudentComparator)

↳ bolje je preko anonimnih razreda ili lambda

```
int main(void){
    StudentComparator usporednik = new StudentComparator();
    TreeSet drvo = new TreeSet(usporednik);
    return 0;
}
```

↳ ovo je bolje pisat pomoću lambda izraza:

```
int main(void){
    TreeSet drvo = new TreeSet(
            (s1, s2) → „nek naš uvjet usporedbe")
    return 0;
}
```

⇒ UZLAZNI I SILAZNI KOMPARATORI:

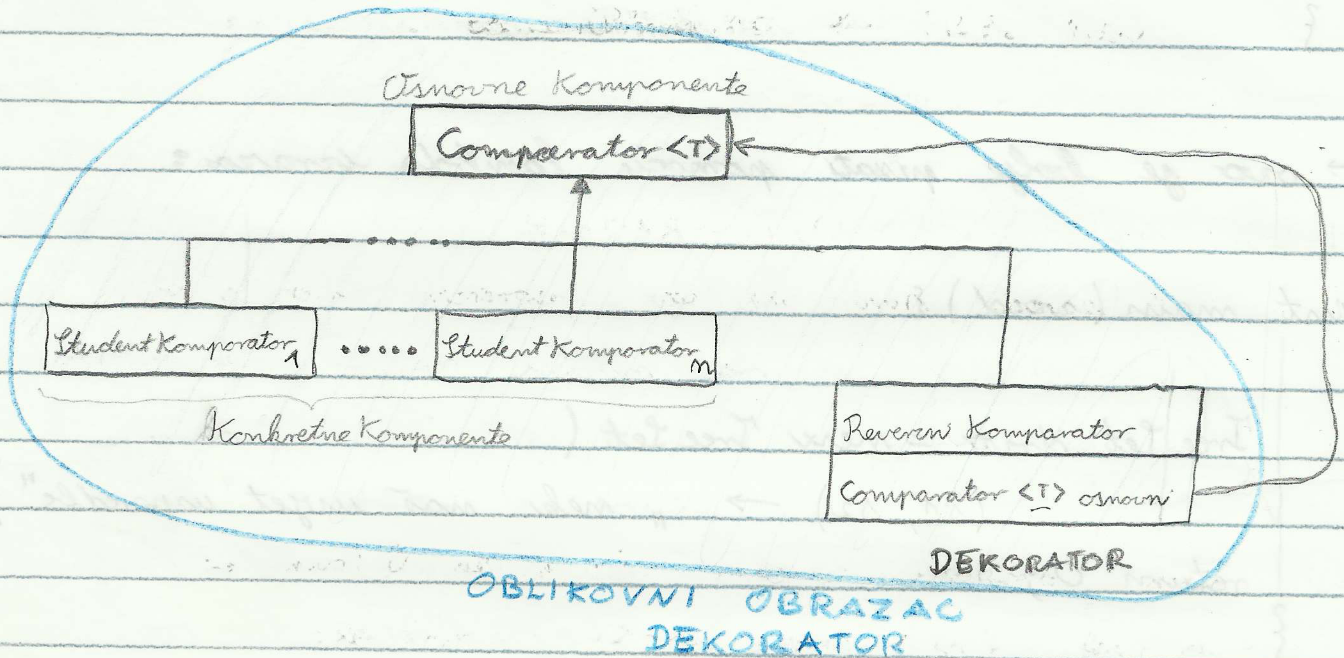
↳ ako želimo „suprotni" komparator, trebamo stavit samo minus ispred povratne vrijednosti metoda za usporedbu

↳ tako umjesto uzlaznog ispisa, dobijemo silazni ispis

↳ no, trebamo onoliko komparatora koliko imamo različitih uvjeta, pa vidimo da ih odmah trebamo duplo više ako želimo suprotan uvjet, tj. suprotan redoslijed

↳ rješenje je stvoriti novi razred koji

se naziva npr. Reverzni Komparator i on implementira sučelje Comparator, al on koristi prethodni komparator kako bi samo obrnuo dobivenu vrijednost



↳ NAPOMENA: Reverzni Komparator ne trebamo pisati jer i Javinom okviru kolekcija postoji metoda koja automatski iz komparatora radi reverzni: Collections.reverseOrder(new StudentComparator())

↳ NAPOMENA: sučelje Comparator ima default metodu reversed koja vraća obrnuti komparator u odnosu na onoga nad kojim je pozvana

⇒ ponekad si je korisno omogućiti da imamo konstante kojima uspoređujemo po određenom kriteriju:

```
public static final Comparator<Student> PO_IMENU =
        (s1, s2) -> s1.firstName.compareTo(s2.firstName);
```

⇒ KOMPOZITNI KOMPARATOR:

↳ sada npr. želimo usporedbu po više varijabli i to na način da su neki reverzni, a neki nisu

```
public class KompozitniKomparator<T> implements Comparator<T> {
    private List<Comparator<T>> kriteriji;

    public KompozitniKomparator(Comparator<T>... kriteriji) {
        this.kriteriji = new ArrayList<>();
        Collections.addAll(this.kriteriji, kriteriji);
    }
    :
}
```

↳ NAPOMENA: kompozitni komparator ne trebamo pisati jer se on također nalazi u

Javnom okviru kolekcija pa možemo pisati:

```
Set<Student> students = new TreeSet<>(
        Student.PO_IMENU.reversed()
            .thenCompare(PO_PREZIMENU)
            .thenCompare(PO_IDU)
);
```

↳ ovakve pokrate koda su jako potrebne

⇒ OBLIKOVNI OBRAZAC STRATEGIJA:

↳ koristi se kada imamo blokove koda koji rade vrlo slične stvari samo se razlikuju u nekoj akciji

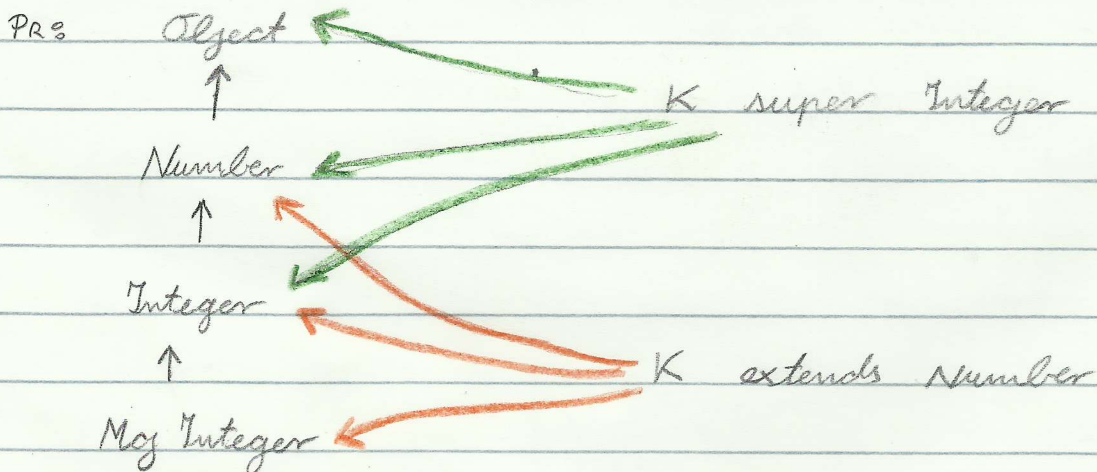
↳ tada stvaramo sučelje s generičkom metodom za koji svaki razred koji implementira to sučelje nudi svoju vlastitu akciju

KOLEKCIJSKI TOKOVI

⇒ npr. u sučelju Map postoj defaultna metoda forEach(...) koja prima argument BiConsumer
↳ smatra se da će nam jako često kod rada s mapama trebati funkcionalnost da obiđemo sve članove mape

⇒ također, postoj metoda compute(...) koja prima argument BiFunction< ? super K, ? super V, ? extends V> jer jako često trebamo činiti nešto sa članovima mape
↳ to činimo onda pomoću metode apply(...) korištene najčešće preko lambda izraza

⇒ OGRANIČENJA NA BiFunction:



⇒ još je bitna metoda merge(...), također iz sučelja map

↳ njome se postiže slična funkcionalnost kao i metodom apply(...)

⇒ pogledaj „Method Handle" oblika:

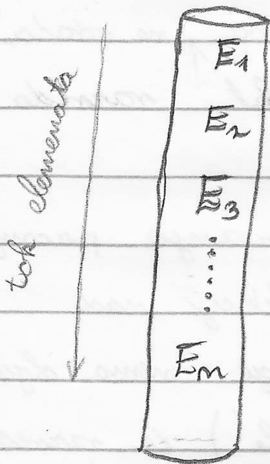
grades.merge("Ante", 1, Integer::sum);

↳ dodaje 1 elementu Ante iz mape

KOLEKCIJSKI TOKOVI

⇒ pomažu kod višedretvenost kako ne bi moral stalno razmišljati o sinkronizaciji i stvaranju dretvi

⇒ zamislimo kolekciju kao cjevovod:



↳ mi samo navedemo želimo li ovakav slijed operacija vršiti slijedno ili paralelno, a Java se za nas brine o sinkronizaciji na temelju terminalne operacije

filter
npr.
e1.grades > 3

ISTI TIP
↳ filter ne mijenja tip podatka

još filtera

transformacija tipa
npr.
e.setGrades

... još transformacija ...

sortiranje
....

terminalna operacija

rad se neka obrada (npr. average())

⇒ ovakve tokove moramo pisati pomoću razreda stream()

```
students.stream().filter(s -> s.grade == 5).
                  forEach(......);
```

⇒ kod rada sa primitivnim tokovima, možemo koristit neke česte metode

↳ npr. average() → primjeti da ona ne vraća prosjek već vraća objekt razreda OptionalDouble

↳ to je razlog sprječavanja rušenja programa ako u kolekciji nad kojom zovemo average nema objekata

↳ metoda getAsDouble() iz razreda OptionalDouble vraća primitivnu double vrijednost

RAD SA DATOTEKAMA

⇒ imamo razred File iz paketa java.io

↳ taj razred sadrži skoro sve što je potrebno za rad s datotekama

⇒ postoji i paket java.nio koji je noviji i dosta efikasniji, ali se zbog velike kompleksnosti rijetko koristi (nudi i nove funkcionalnosti)

⇒ RAZRED FILE:

↳ NAPOMENA: prilikom pokretanja java programa, trenutni direktorij se postavlja na direktorij Java projekta

```
File dir = new File(".XY");
```

↳ ovo bi značilo da imamo razred koji se odnosi na datoteku XY u trenutnom direktoriju (direktoriju projekta)

↳ stoga, objekt razreda File predstavlja putanju u datotečnom sustavu - to ne znači da ta datoteka ili direktorij postoji

↳ nad tim objektom možemo pozvati npr. mkdir(), pa će se stvoriti taj direktorij na disku

⇒ PAKET **java.nio**:

↳ postoje dva temeljna razreda:

- **Files** – ispituje i obrađuje određene stvari
- **Path** – rad samo sa stazama

↳ primijeti da je Path apstraktni razred, pa ga ne možemo stvoriti sa „new"

↳ stoga, stvaramo ga sa:

```
Path path = Paths.get("d:/ja/ti/on.txt");
```

⇒ NAPOMENA: Java je neovisna o platformi, pa se slash ('/'), na Windows-u interpretira kao separator direktorija

PR: Zamislimo da pišemo rekurzivnu metodu koja ispisuje sve o direktoriju i datotekama od nekog zadanog direktorija!

↳ koristimo oblikovni obrazac strategija te pišemo sučelje:

```
interface Obrada {
    void obradiDatoteku(Path p);
    void ulazimUDirektorij(Path p);
    void izlazimIzDirektorija(Path p);
}
```

↳ naravno, ovo sučelje se često koristi i stoga je napisano u standardnoj Java knjižnici:

File Visitor <T>

↳ ovo sučelje ima metode koje komuniciraju sa korisnikom (i međusobno) pomoću enumeracije:

File Visitor Result

⇒ I/O TOKOVI:

VRSTA	IZVOR PODATAKA	PONOR PODATAKA
oktet	Input Stream	Output Stream
znakovi	Reader	Writer

↳ oktet u Javi: byte

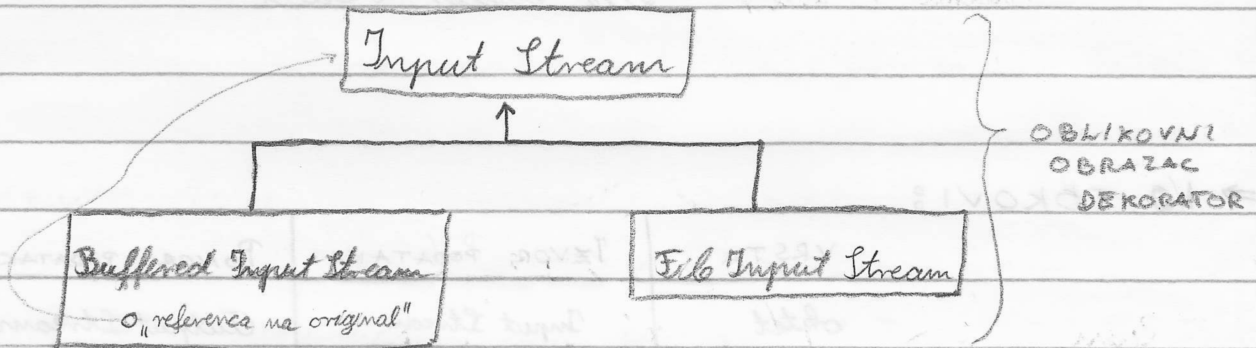
↳ znak u Javi: char

⇒ tokovi su jednosmjerni – nakon što smo pročitali podatak, ne možemo se vraćati natrag

↳ u posebnim slučajevima to možemo ako tok to podržava – to ispitujemo pomoću metode markSupported(), ali ovo nije osobito bitno

⇒ kako povećati efikasnost čitanja?

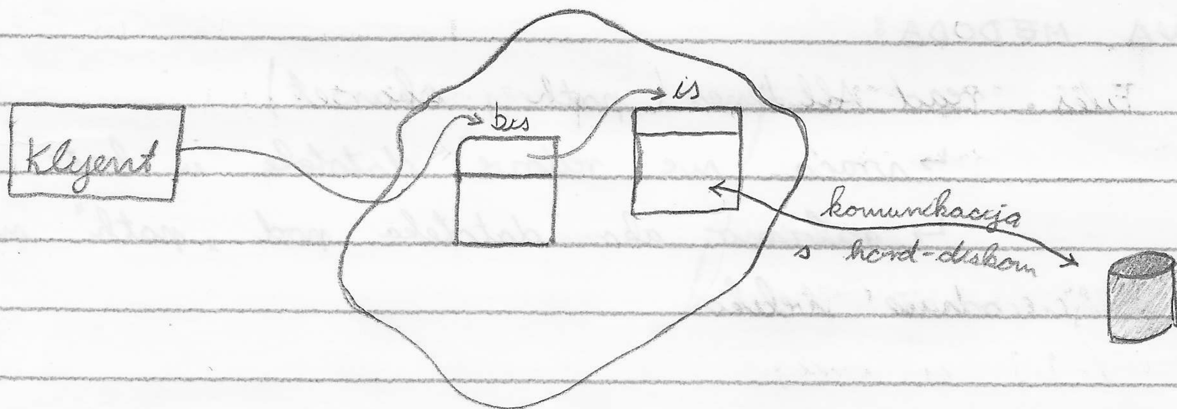
↳ ako klijent uvijek čita pomoću read(), to može biti užasno neefikasno jer on svaki put ponovno podešava glavu čitača diska, stoga, mi ćemo zapakirati metodu read() na način:




```
class BufferedInputStream {
    byte[] spremnik;
    InputStream original;
    int stvarnoProcitano;
    int iducaVracenaPozicija;

    read();
}
```

↳ klijent će sada komunicirati sa diskom preko BufferedInputStreama koji je zapravo dekorator početnog InputStreama

```
public static void main(void) {

    InputStream is = new InputStream('.');
    BufferedInputStream bis = new BufferedInputStream(is);

    /*klijentu sada šaljemo „bis", a ne „is"*/
    Klijent( bis);
    ⋮
}
```



⇒ pomoću metode Files. write(Path path, „standard")
možemo zapisati na „path" pomoću standarda

↳ „standard" je kodna stranica koja govori kako koj znak zapisati u datoteku (broj okteta, kodiranje, Little ili Big Endian)

⇒ u Javi, Char ≠ Byte ⇒ baš zbog toga što char može biti višeoktetni

⇒ File Reader i File Writer nisu najbolji za rješavanje problema sa datotekama jer sve rade sa kodiranjem znakova po default-noj kodnoj stranici operacijskog sustava

↳ gubi se prenosivost

⇒ preporuka: koristi UTF-8 jer je ta kodna stranica najraširenija po današnjim operacijskim sustavima

⇒ POMOĆNA METODA:

Files. read All Lines (path , charset)

↳ vraća sve redove datoteke u list

↳ koristimo ako datoteka pod „path" nije odviše velika

⇒ postoje načini preko Buffered Readera da obradimo datoteku na način da nikad ne učitavamo cijelu datoteku u memoriju (PROUČI!)

⇒ RAD SA .zip DATOTEKOM:

↳ postoje razredi Zip File , Zip Content Writer i Zip Entry

↳ prilikom stvaranja toka prema .zip datoteci dobro je pisati:

Input Stream is = ZipFile. get Input Stream (zip Entry);

GRAFIČKO SUČELJE
SWING

⇒ grafičko korisničko sučelje (GUI) se sastoji od:
- Abstract Windows Toolkit (AWT)
  ↳ zastarijela tehnologija
- **Swing**
  ↳ nadogradnja AWT-a
- 2D API
  ↳ klase za crtanje u Swing komponentama
- Accessibility API
- Java FX
  ↳ najnovija tehnologija, ali još nije dio standardne Java platforme

⇒ jedna od osnovnih komponenti GUI-a je sučelje **Layout Manager**, **Layout Manager 2**
↳ npr. jedna od zadaća im je da se komponente prozora prilikom promjene veličine „lijepo" ponašaju

⇒ sve što je prikazano u Swingu mora biti smješteno u jedan od 3 vrsta „container"-a:
- **J Frame** ⇒ standardni prozor
- **J Dialog** ⇒ dijaloški prozor (sa kontrolom korisnika)
- **J Applet** ⇒ koristi se za web aplikacije i stranice

↳ samo se vršni „container"-i prikazuju u OS-u

⇒ Swingove metode nisu višedretveno sigurne, pa postoji jedna Swingova dretva (dok ima Swingovih komponenti)
⇒ prilikom pozivanja Swingove metode, moramo signalizirati Swingovoj dretvi da mi sada želimo raditi za GUI-om:

```
SwingUtilities.invokeAndWait(new Runnable() {
                                @Override
                                public void run() {
                                    :
                                }
                             });
```

↳ ova metoda baca hrpu iznimaka, pa moramo koristiti try-blok

↳ kako bi ovo izbjegli, a za naše programe nisu ni bitne te iznimke, koristimo metodu:

```
SwingUtilities.invokeLater(new .... );
```

↳ <u>ovdje koristimo lambda izraz:</u>

```
SwingUtilities.invokeLater(() -> { window.size(...)
                                   :
                                 });
```

⇒ FLOW LAYOUT:

↳ najjednostavniji razred čija je zadaća razmještaj komponenata

↳ razmješta sve u jedan red, a ako nema mjesta, onda u novi red

↳ napomena: svaka od komponent vraća „layout manageru" veličinu potrebnu za pravilan prikaz komponente

↳ umjesto nagađanja veličine prozora za „ljepi" prikaz svih komponent, svaki vršni container ima metodu **pack()** koja pita „flow layout manager"-a za tu veličinu koju on zna na temelju vrijednosti vraćenih od svih komponenti

↳ možemo podesivati vertikalni i horizontalni razmak između komponent:

- setVGap(...)
- setHGap(...)

⇒ BORDER LAYOUT:

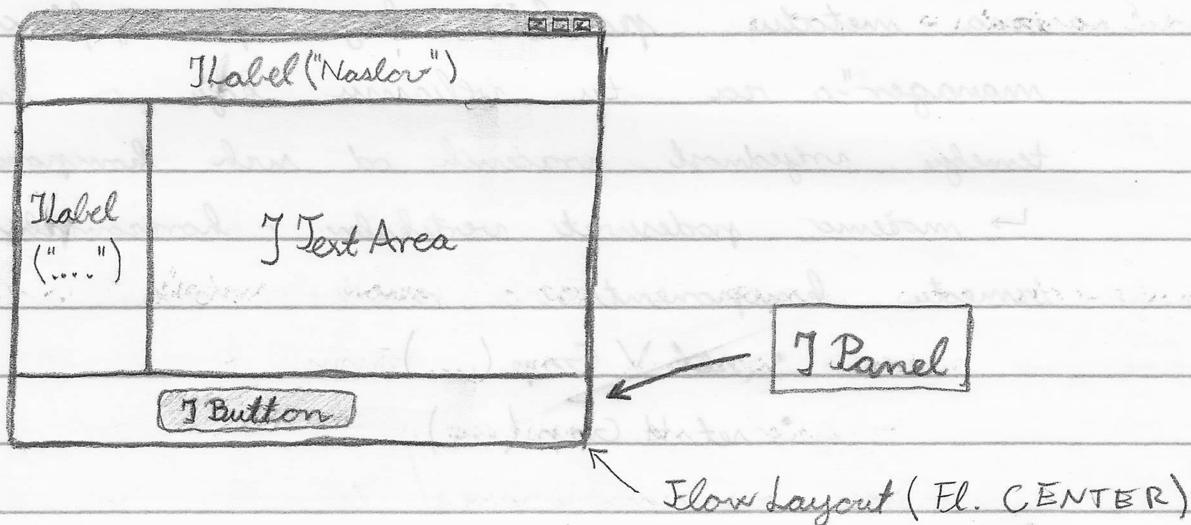
↳ dijeli područje prozora na 5 cjelina:

- SJEVER: poštuje visinu, rasteže širinu na kolko god ima mjesta
- JUG: analogno sjeveru
- ZAPAD: poštuje širinu, rasteže visinu na preostalo mjesto
- ISTOK: analogno zapadu
- CENTAR: komponente preuzimaju preostalo mjesto

⇒ NAPOMENA: umjesto geografskih konstanti, u novije vrijeme je bolje koristiti konstante (također definirane u razredu Border Layout): LINE_START, LINE_END, PAGE_START, ...

↳ time se npr. u zemljama gdje se čita sa lijeva nadesno cijeli prozor mijenja na očekivan način u toj zemlji

⇒ pogledajmo primjer ovakvog prozora:



⇒ NAPOMENA: radi urednosti prozora često koristimo granice (Border) koje uzimaju npr. prazan prostor od prozora prema unutra

⇒ NAPOMENA: J Panel po defaultu koristi Flow Layout.

⇒ GRID LAYOUT:

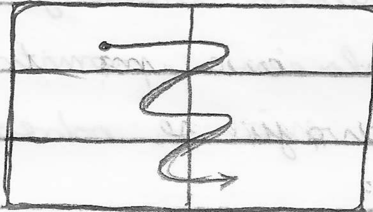
↳ tablični raspored

↳ konstruktor: GridLayout(int rows, int cols)

↳ ima metode:

- setColumns(int cols)
- setRows(int rows)

↳ ako je jedan od argumenata prazan, uzima se ona dimenzija da se prikažu sve komponente

GridLayout(3, 2) ⇒ 

⇒ NAPOMENA: uvijek je bolje pisati metodu „add" nad nekim containerom (vršnom komponentom)

↳ ako ne pišemo, to će automatski, implicitno („ispod haube) biti delegirano vršnoj komponenti

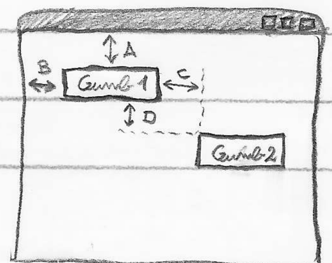
⇒ SPRING LAYOUT:

↳ definira odnose između rubova komponenti

↳ dodavanje odnosa:

- layout.putConstraints()

↳ ima puno pisanja al je jako definiran (što je nekad potrebno)

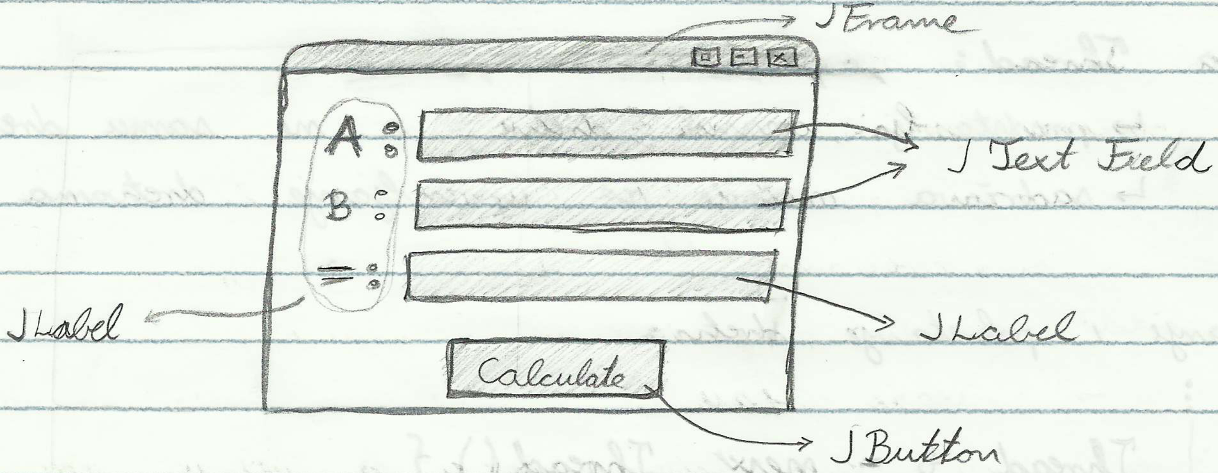


MODEL DOGAĐAJA

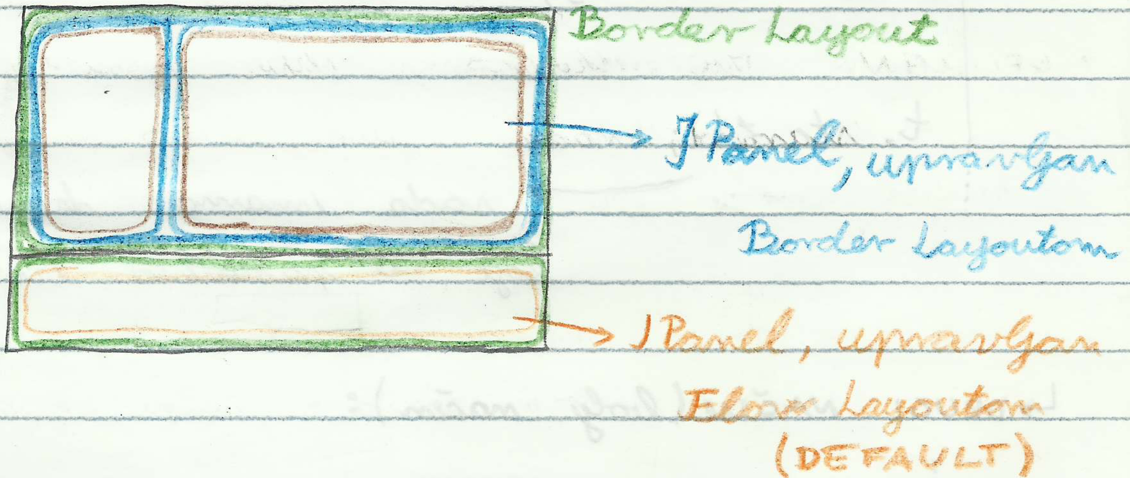
⇒ „Event-driven programming"
⇒ komponente okidaju događaje
↳ postoje promatrači („observer" ili „listener") koji su zainteresirani za pojedini događaj
↳ promatrač je pretplaćen na određene događaje
↳ nakon okidanja događaja, svi pretplaćeni promatrači reagiraju (pozivaju se određene metode)

⇒ NAPOMENA: Za prikazivanje slika je najbolje koristiti „Buffered Image" – ona koristi razred Graphics

PR: Izrada željenog prozora u Swingu:



↳ ovo nije moguće načinit jednim od Javinih defaultnih layout managera, već ih treba kombinirati



↳ u ovaj plavi JPanel treba dodat još dva JPanela kako bi rasporedili komponente (to su ova dva smeđa JPanela)

↳ sada u JPanele možemo počet dodavat željene komponente

VIŠEDRETVENOST

⇒ klasa Thread:

↳ predstavlja opisnik dretve, a ne samu dretvu
↳ sadržava metode za upravljanje dretvama

⇒ stvaranje i pokretanje dretvi:

```
Thread t = new Thread() {
        public void run() {
            „kod koji dretva izvršava"
        }
};

t.start();
```

→ sada imamo dvije dretve koje se paralelno izvode

↳ DRUGI NAČIN (bolji način):

```
Runnable posao = new Runnable() {
        public void run() {
            „posao koji treba izvesti"
        }
};
Thread t = new Thread(posao);
t.start();
```

⇒ preko lambda izraza:

```
Runnable posao = () → {
        „posao koji treba izvest"
};
```

⇒ modeliranje preko poslova je bolje jer tako više dretvi možemo predat isti posao (također, posao je uvijek bolje modelirat kao posao, a ne kao dretvu)

⇒ postoje dvije vrste dretvi:

- **korisničke**
- **daemon** - pomoćne dretve

↳ ako su sve korisničke dretve nekog procesa ugasle, gasi se i cijeli proces

⇒ primjer pomoćne (daemon) dretve je **garbage collector**

⇒ dobra praksa je smisleno imenovat dretve jer generička imena nisu osobito korisna

⇒ PRIORITETI DRETVI:

↳ prioriteti se rade sa metodom:

setPriority(int newPriority)

↳ veći broj je veći prioritet

↳ Java ne može garantirat da će namještanje prioriteta radit idealno jer je svaki OS poseban, a Java je neovisna o platformi

↳ prioriteti dretvi su cijeli brojevi od 1 do 10 i to na način da je:

- 1 → MIN_PRIORITY
- 5 → NORM_PRIORITY (default)
- 10 → MAX_PRIORITY

⇒ metoda **yield()**

↳ naputak OS-u da dretva nema ništa korisno dalje za raditi i da bi željela ući u red pripravnih dretvi

↳ OS može ignorirati ovaj naputak

⇒ metode za „nasilno" zaustavljanje ili suspendiranje dretvi iz neke druge dretve maknute su još u ranim verzijama Jave

↳ te metode mogu izazvati potpuni zastoj

– npr. dretva koristi zajednički spremnik i zaurela je „mutex", a onda prije izlaska ju neka druga dretva suspendira

↳ time je „mutex" vječno zauzet i imamo potpuni zastoj

⇒ „Event Dispatching Thread" ne smije raditi neki dugi posao (npr. 5 sekundi), (mi taj posao modeliramo funkcijom sleep(...)) jer će ta glavna dretva biti zablokirana na 5 sekundi što znači da ćemo imati mrtvo korisničko sučelje

↳ rješenje je da taj posao radi neka druga dretva - radna dretva

⇒ SWING i AWT nisu višedretveno sigurna sučelja što znači da bi samo swing-ova dretva trebala upravljati swing-ovim sučelje

⇒ pogledaj metodu za upravljanje komponentama swing-ova sučelja iz „ne-swingovih" dretvi:

- SwingUtilities.invokeLater(...)

SWING I DRETVE

⇒ ako postoji neki dugi, zahtjevni posao, mi želimo da se on vrti u pozadini:

↳ ovo nam daje SwingWorker (apstraktan razred s metodom doInBackground koju trebamo nadjačati)

↳ metoda publish - daje podatak na obradu

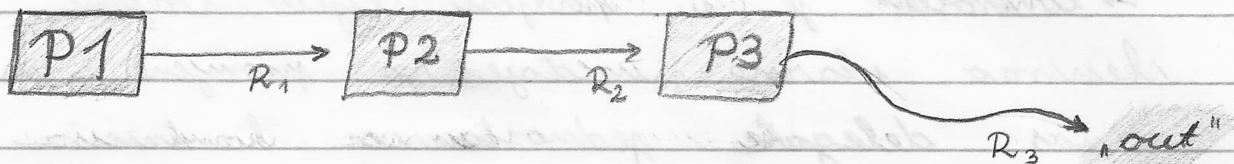
↳ metoda process - obrađuje sve podatke koje do sad nije obradila (prima listu podataka)

⇒ ukratko, SwingWorker nudi jednostavnu apstrakciju kojom se rješavaju problemi swing-ove višedretvene nesigurnosti

↳ npr. metoda doInBackground garantira da će se dugi posao izvoditi u novoj dretvi, a ne u Event Dispatching dretvi kako se korisničko sučelje ne bi smrznulo

⇒ pogledajmo sljedeći problem zavisnih poslova:

↳ rezultat posla 1 treba za posao 2, a rezultat posla 2 treba za posao 3



↳ rješenje je da dodamo novu dretvu koja se bavi samom sinkronizacijom ovih poslova - ovdje je svaki od poslova modeliran zasebnim SwingWorker-om

⇒ PERIODIČKO ODGAĐANJE DOGAĐAJA:

↳ java.util.Timer ⇒ radi pozadinsku dretvu koja radi brojanje vremena

↳ **java.swing.Timer** ⇒ koristi se kod aplikacija sa swing-om jer se on automatski brine za sinkronizaciju

⇒ OBLIKOVNI OBRAZAC „GENERAL MVC":

↳ **model, view, controller:**

↳ model - unutarnje stanje komponente

↳ view - određuje vizualni izgled

↳ controller - mijenja model u odnosu na ulazne događaje

⇒ danas, view i controller su povezani u jednu komponentu **delegate** - ona komunicira s operacijskim sustavom

↳ **controller** je u prošlosti cijelo vrijeme skenirao ulazne uređaje za promjene, a danas **delegate** jednostavno komunicira s OS-om i pazi na njegove dojave

⇒ **delegate** je onaj tko se stvarno bavi crtanjem, pa sadrži metodu **paint(...)**

↳ komponente pozivom svoje metode **paint()** dojavljuju delegatu da žele biti nacrtane, pa jednostavnom promjenom delegata mijenjamo temu (vanjski izgled) prozora

↳ ako želimo promijeniti izgled nekog prozora, pozivamo metodu:

- SwingUtilities.updateComponentTreeUI(...)

↳ ovo mijenja delegate sve djece dane komponente

DODATNO O DELEGATIMA

⇒ <u>komponenta JList:</u>

↳ prikazuje podatke u listi

↳ JList dozvoljava nasumičnu selekciju više elemenata u listi

↳ komponenta JList treba model podataka:

data 1 ~~
data 2 ~~
data 3 ~~
data 4 ~~
⋮

ovo se može promijeniti

```
interface ListModel <E> {
    int getSize();
    E getElementAt(int index);
    void addListDataListener(ListDataListener l);
    void removeListDataListener(ListDataListener l);
}
```

} INFORMACIJE O PODACIMA

↳ prve dvije metode trebaju za informacije o podacima u listi kako bi ona znala kako veliku površinu treba za iscrtavanje (JList uz te podatke pogleda kojim fontom ono ispisuje, prazne prostore i onda tek može vizualno složiti samu komponentu)

↳ problem može biti u sljedećem:

- ako imamo promjenjiv model (npr. dodamo grad u listu), model treba dojaviti promjenu svim pogledima da se model promijenio te tako izazvati ponovno iscrtavanje

, "ono", , razlicit pogled (npr. JList, JTextBox, JButton, ...) mogu gledat na isti model, pa model ne smije znat ništa o samom pogledu na njega već ćemo poglede modelirati sučeljima:

```
interface ListDataListener extends EventListener {
        void intervalAdded ( ListDataEvent e);
        void intervalRemoved (ListDataEvent e);
        void contentsChanged(ListDataEvent e);
}
```

↳ prve dvije metode omogućuju pogledu da radi manje promjene, a ne da svaki put crta sve opet od nule (također, omogućuje da selekcije ostaju na ekranu ako selektirani element nije izbrisan)

↳ treća metoda contentsChanged() radi veliku promjenu i crta cijeli pogled iz početka (ovo se radi za npr. neko sortiranje)

⇒ svi pogledi se prijavljuju nekom modelu preko metode addListDataListener (...)

↳ time se pogled „pretplati" na promjene u modelu, pa je model dužan izvijestit njegovu promjenu prilikom svake promjene samog modela

↳ vidimo da model stoga treba imat neku strukturu (npr. Set) u kojem drži popis svih pogleda koji su pretplaćeni

⇒ ovo je bitno uređenje grafičkog sučelja:

- MODEL – ne zna tko ga gleda, samo dojavljuje promjene svima koji gledaju
- POGLED – ne zna od kud stižu podaci za iscrtavanje (svejedno je je li to disk, TCP server, neki drugi program, ...)

↳ ovakvo modeliranje GUI-a se primjenjuje u gotovo svim objektnim programskim jezicima

⇒ neke komponente imaju više vrste modela:

↳ npr. JList ima:

- DataModel – prati podatke
- Selection Model – prati selekciju kursora

⇒ pomoćni razred AbstractListModel ima implementirane (neke) metode iz ListModel sučelja (ili pomoćne metode za implementiranje tih metoda – npr. fireIntervalAdded(...))

TESTIRANJE

⇒ izvođenje programskog koda skupom ulaznih podataka

⇒ TESTNI UZORAK: uređen par testnog ulaza i očekivanog izlaza

⇒ xUNIT → zajedničko ime za programske okvire za testiranje jedinica

↳ za Javu je to JUNIT