PROGRAMIRANJE I.

UVOD:
- → **računalnik** hiter, natančen, neumen
- → **program**: datoteka z navodili računalniku
- → **programiranje**: pisanje programov
- → programski **jezik**: natančna nedvoumna navodila za pisanje programov npr. java

BASIC POJMI:
- → **prevajanje:** pretvorba v obliko, da jo izvede java vitrual machine: **javac**
- → **izvajanje: java**
- → rezervirana imena
- → **zamik:** { pomeni premik za 4 presledke }
- → **presledki:** okrog operatorjev, nikoli pri klicu metode!
- → **spremenljivka:** prostor v pomnilniku ki lahko hrani neko vrednost
- → **deklaracija:** tip in ime spremenljivke
- → **inicializacija:** podamo še začetno vrednost
- → **stavek:** element programa z nekim učinkom, stavek ki nekaj priredi: izraz na desni se izračuna, rezultat se zapiše v levi

PODATKOVNI TIPI:

tip	obseg
byte	$[-2^7, 2^7 - 1]$
short	$[-2^{15}, 2^{15} - 1]$
int	$[-2^{31}, 2^{31} - 1]$
long	$[-2^{63}, 2^{63} - 1]$

če je en operand realnoštevilski se izvede realnoštevlska operacija

tip	natančnost	obseg
float	7 mest	od ca. -10^{38} do ca. 10^{38}
double	15 mest	od ca. -10^{307} do ca. 10^{307}

JAVA:
- → neodvisno od platforme: **JVM**
- → **prevajanje** in **izvajanje**
- → strogo preverjanje
- → dobre **standardne knjižnice**

SCANNER:
- → branje podatkov

```
int a = sc.nextInt();
long b = sc.nextLong();
double d = sc.nextDouble();
```

- → samo next je string

OPERATORJI

+	seštevanje
-	odštevanje
*	množenje
/	deljenje
%	ostanek pri deljenju

množenje deljenje imajo višjo prioriteto kot seštevanje in odštevanje zato oklepaji

POGANJANJE:
- → prek tipkovnice: izhod na zaslon **cmd**
- → `echo podatki | java Program`
- → `java Program < vhod.txt`
- → `cat vhod.txt | java Program`
- → `java Program < vhod.txt > rezultat.txt`
- → `diff rezultat.txt izhod.txt`
- → `tj.exe Program.java testi rezultati`

PRETVORBE TIPOV:
- → realnoštevilski v celoštevilski odreže decimalke, **ne zaokroži**!
- → pišemo: `(T)spremen.`

KRAJŠANJE OPERATORJEV
- → daljša: `a = a op x`
- → krajša: `a += x`
- → normalno: `a += 1`
- → prefiksna: `++a`
- → postfiksna: `a++`
- → prefiksna najprej spremeni nato uporabi
- → postfiksna uporabi in šele nato spremeni

KRMILNI KONSTRUKTI

&& and
|| or
! not

spremenljivke so vidne samo v svojem bloku

IF STAVEK:
- → del kode se izvede če izpolnjen določen pogoj
- → **else**: koda, ki se izvede če pogoj ni izpolnjen
- → **else if**: pogojni stavek ki sledi če pogoj ni izpolnjen
- → pogoj je logični izraz **boolean**
- → samo **true** / **false**

>	večji od	<=	manjši ali enak kot
>=	večji ali enak kot	==	enak kot
<	manjši od	!=	različen od

WHILE ZANKA:
- → če pogoj true **loop**
- → ko gre ven se izvede naslednji stavek

DO ZANKA:
- → `do { stavki } while (pogoj)`
- → najprej stavki in nato v loop
- → stavki **vsaj enkrat**!

System.out.printf
- → % zamenjajo vrednosti
- → izjema prelom vrstice %n
- → izjema znak %%

%d: celo število (byte, short, int, long)
%f: realno število (float, double)
%c: znak (char)
%s: niz (String)
%b: logična vrednost (boolean)

%wx
širina izpisa *w*, desna poravnava

%-wx
širina izpisa *w*, leva poravnava

%0wd
širina izpisa *w*, polnjenje z vodilnimi ničlami

%.df
realno število, zapisano na *d* decimalk

%w.df
širina izpisa *w*, *d* decimalk

FOR ZANKA:
- → `for (inicializacija; pogoj; posodobitev) { }`
- → to je okrajšava za while, kadar vemo koliko korakov
- → izjema spremenljivka obstaja do konca **telesa zanke**

TIP CHAR:
obseg: $[0, 2^{16} - 1]$
- → spremenljivka hrani znak
- → znaki predstavljeni z celimi števili
- → **ASCII** in **Unicode**
- → pretvarjamo kot zgoraj

POGOJNI OPERATOR
- → **trojiški** operator
- → `pogoj ? izraz 1 : izraz 2`
- → če true potem 1 false pa 2

SWITCH
- → samo za tipe byte, int, shar, String, enum
- → če izraz enak **konstanti** se izvrši tisti stavek, ki pod konstanto
- → če izpustimo **break** se nadaljuje pri naslednjem **case**
- → spremenljivka do konca switch stavka razen če case zapremo v **blok**

```
switch (izraz) {
    case konstanta_1:
        stavki_1
        break;

    case konstanta_2:
        stavki_2
        break;
    ...

    case konstanta_n:
        stavki_n
        break;

    default:
        stavki_{n+1}
        break;
```

BREAK in CONTINUE:
- → **break:** prekine zanko v kateri se nahaja
- → **continue:** skoči dritto na preverjanje pogoja

METODE

- → kadar se nočemo ponavljat
- → kos kode ki ga lahko **pokličemo**
- → **funkcija**, procedura, podprogram
- → izvajanje se začne v metodi **main**
- → klicanje: `metoda();`

REKURZIJA
- → metoda kliče samo sebe
- → pazi z tabelami in ostalimi **pointerji**!

`public static void metoda(T1 f1, T2 f2, ..., Tn fn)`

PARAMETRI
- → spremenljivke samo v svojem bloku
- → podatke prenašamo z parametri
- → f so samo **formalni** parametri
- → kopirajo se v **dejanske** istoležne
- → obnašajo se kot spremenljivke
- → **ločene** od dejanskih parametrov!

VOID
- → ne vrne **ničesar**
- → če nadomestimo z **tipom** vračamo vrednost z stavkom **return**
- → return takoj **zaključi** metodo
- → v vseh možnih primerih rabimo return če metoda ni void
- → tudi v void lahko return **brez vrednosti**

TABELE

- → skupek spremenljivk istega tipa
- → dosegljive z **indeksom**
- → spremenljivke so **elementi**
- → indeksi se začnejo z **0**
- → `tabela[ indeks ]`
- → `tabela.length`
- → `ArrayndexOutOfBoundsException`

IZDELAVA TABELE
- → izdelava z **seznamom** elementov
- → izdelava z **privzetimi** vrednostmi
- → dolžine **ne moremo** spreminjati!

`T[] tabela = {element0, element1, ...};`

`T[] tabela = new T[dolžina];`

byte	(byte) 0	double	0.0
short	(short) 0	char	'\0'
int	0	boolean	false
long	0L	objektni tipi	null
float	0.0f		

METODE in STAVKI pri tabelah

```
for (int i = 0;  i < t.length;  i++) {
    // i: trenutni indeks
    // t[i]: element tabele na indeksu i
    ...
```

```
for (T element: t) {
    // v prvem obhodu: element = t[0]
    // v drugem obhodu: element = t[1]
    // ...
    // v zadnjem obhodu: element = t[t.length - 1]
    ...
```

→ indeks **največjega** elementa:

```
public static int indeksMaksimuma(int[] t) {
    int iMax = 0;
    for (int i = 0;  i < t.length;  i++) {
        if (t[i] > t[iMax]) {
            iMax = i;
        }
    }
    return iMax;
```

→ splošna metoda za **iskanje** elementa:

```
public static int poisci(int[] t, int x) {
    for (int i = 0;  i < t.length;  i++) {
        if (t[i] == x) {
            return i;
        }
    }
    return -1;
```

DVOJIŠKO ISKANJE

→ kadar tabela urejena **po velikosti**
→ pogledamo element na **sredini**
→ imamo tri opcije: enak, večji manjši
→ če **enak** smo končali
→ če **večji** izločimo **desno** polovico
→ če **manjši** izločimo levo polovico
→ pregledati moramo največ dvojiški logaritem n + 1 elementov
→ bistveno **hitrejše** od navadnega

```
public static int poisci(int[] t, int x) {
    int lm = 0;                 // leva meja
    int dm = t.length - 1;      // desna meja
    while (lm <= dm) {
        int s = (lm + dm) / 2;
        if (t[s] == x) {  // element smo našli!
            return s;
        }
        if (t[s] < x) {
            lm = s + 1;
        } else {
            dm = s - 1;
        }
    }
    return -1;   // elementa ni
```

MEMOIZACIJA

→ npr. rekurzivna rešitev
→ shit če isto vejo večkrat računamo
→ naredimo **tabelo** za n + 1
→ ko izračunamo k **vstavimo** v T [k]
→ preden računamo naslednjo vejo samo **preverimo** a velja T [k] > 0 in če velja **jo ne računamo**

UREJANJE TABELE

→ veliko različnih algoritmov
→ obstaja premade metoda:
`java.util.Arrays.sort();`
→ parametri: tabela, prvi in zadnji element po želji

```
public static void uredi(int[] t) {
    for (int i = 1;  i < t.length;  i++) {
        // vstavi element t[i] v podtabelo t[0..i-1]
        int element = t[i];
        int j = i - 1;
        while (j >= 0 && t[j] > element) {
            t[j + 1] = t[j];
            j--;
        }
        t[j + 1] = element;
```

→ `Arrays.toString(tabela);` vrne vsebino v obliki Stringa, razred je v paketu **java.util**

RAZREDI IN OBJEKTI

PRIMITIVNI IN REFERENČNI TIPI

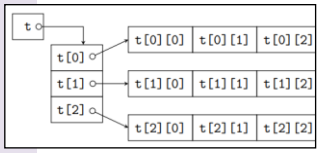
→ primitivni: byte, short, long, float, double, boolean, char
→ spremenljivka vsebuje **vrednost**
→ referenčni: String, tabela, Scanner
→ vsebuje **pomnilniški naslov objekta** oz. pointer
→ **pointer**: podatek o **lokaciji** objekta v **pomnilniku**

→ pazimo pri **kopiranju**
→ kopira se **pointer** na objekt sedaj kažeta na isti objekt, **ne spremeni se vrednost** v objektu

TABELA	OBJEKT
določen s tipom elementov in številom dimenzij	tip objekta podamo v class tam navedemo atribute in tipe
ustvarimo z new	ustvarimo z new
spremenljivka tabelaričnega tipa vsebuje pointer na tabelo	spremenljivka objektnega tipa vsebuje pointer na object
spremenljivke istega tipa	spremenljivke poljubnih tipov
praviloma imajo enak pomen	praviloma različen pomen pripadajo celoti
elementi	atributi
elementi dostopni prek indeksov	atributi dostopni prek imen `objekt.ime`

2D TABELE

→ **tabele kazalcev** na tabele
→ `T[][] = new T[m][n];`
→ če izdelujemo z določenimi vrednostmi isto kot gor samo da elementi **tabele**
→ niso nujno enako dolge
→ najprej tabela kazalcev in lahko posamezne vrstice **posebej**
→ privzeta vrednost pointerjev **null**




```
public class Ulomek {
    int stevec;
    int imenovalec;
}
```

```
Ulomek u = new Ulomek();
u.stevec = 3;
u.imenovalec = 5;
```

RAZRED

→ praviloma v **ločeni** datoteki
→ ime **datoteke** je ime **razreda**
→ samo deklariramo ne inicializiramo atribute
→ po navadi atributi niso dostopni izven razreda in so spremenljivke **private**
→ če so **public** dostopamo kot piše desno

DOSTOPNA DOLOČILA

→ dostop po navadi preko **metod** ker so atributi private
→ skrivanje načina implementacije
→ varovanje konsistentnosti objekta
→ **public:** katerikoli class
→ **protected:** razredi v istem paketu in podrazredi razreda
→ **brez:** razredi v istem paketu
→ **private:** samo razred kjer je

KONSTRUKTOR

→ special metoda ki se kliče ob izdelavi objekta
→ ime **konstruktorja** enako imenu **razreda**
→ nima izhodnega tipa
→ praviloma nastavimo atribute na **privzete vrednosti**, po defaultu java naredi svoj konstruktor če ga mi ne, operator **new**
→ kazalec na nov objekt se skopira v spremenljivko **this**
→ rezultat je **pointer** na objekt
→ med seboj se lahko kličejo z **this** vendar samo čisto **na začetku** konstruktorja!

KONSTANTA

→ določilo **final** označuje konstanto ki je ne moremo spreminjati
→ deklariramo s static ker pripada celemu razredu
→ pišemo z **caps lock**

StringBuilder: metoda append doda nek znak

v javi lahko več metod z istim imenom če imajo različen podpis enako velja za konstruktorje

metoda String.format je podobna System.out.printf ampak namesto izpisa vrne String

METODE V RAZREDIH

→ **static** pomeni da je vedno ista pri vseh objektih, potem jo moramo klicati z `Razred.metoda(objekt)`
→ če spustimo static lahko kličemo `objekt.metoda(..)`
→ kazalec objekt se skopira v **this**
→ v static lahko spustimo razred če se nahaja v istem razredu
→ **statični atributi** niso vezani na objekt pripadajo celemu razredu
→ po navadi vedno naredimo **getter** in **setter**
→ priporočljivo je da so objekti **nespremenljivi**, zato setter vrne nov objekt

DEDOVANJE

→ relacija med razredi ne objekti
→ če je B **podrazred** in A **nadrazred** je vsak objekt tipa B tudi tipa A
→ razred B **podeduje** vse atribute iz A in vse metode razen **private** metod
→ razred B lahko svoje metode
→ razred B lahko **redefinira** metode

→ po navadi želimo dodat konstruktor
→ konstruktor podrazreda se začne s stavkom **super** in **parametri**
→ če ga izpustimo prevajalnik sam doda super **brez parametrov**
→ če v nadrazredu ni konstruktorja brez parametrov **napaka**

example konstruktorja v podrazredu

```
public IzredniStudent(String ip, String vpisna,
                      int stroskiBivanja, int solnina) {
    super(ip, vpisna, stroskiBivanja);
    this.solnina = solnina;
}
```

REDEFINICIJA METODE
→ da ziher redefiniramo: `@Override`
→ more imet **enako ime** kot v nadrazredu
→ enako zaporedje in tipe parametrov
→ dostopno določilo isto ali ohlapnejše
→ lahko uporabimo `super.metoda(…)`
→ to uporabi metodo iz nadrazreda ampak z **parametri** tega podrazreda

HETEROGENA TABELA
→ če ustvarimo tabelo tipa A lahko vanjo vpisujemo objekte **tipa A** in **B**
→ ker je **object nadrazred** vseh razredov lahko v tabelo tipa object vpisujemo **vse**
→ med **izvajanjem** se pokličejo metode iz podrazreda če so redefinirane

PREVERJANJE TIPA
→ objekte preverjamo z metodo **equals**
→ pazit moramo da **pravilno nastavljena**
→ `objekt1.equals(objekt2)`
→ lahko uporabimo **instanceof**
→ `objekt instanceof Razred`
→ instanceof je true kadar kaže na objekt **razreda** ali njegovega **podrazreda**
→ včasih zaradi prevajalnika ki ne ve tipa uporabimo **pretvorbo** tipa
→ spremenljivka a v času prevajanja **tipa A** in pretvorimo z **v tip B** z `(B) a`
→ prevajalnik zaupa da je res tipa B
→ instanceof uporabljamo samo kadar je izračun **smiseln** samo za podrazred

OVOJNI TIP
→ primitivni tipi niso v hierarhiji
→ uporabljamo ovojne tipe

byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

→ **samodejno pretvarjanje** med primitivnimi in ovojnimi tipi
→ lahko v Integer `Integer.valueOf()`
→ lahko v int `intValue()`
→ ovojni tip lahko vrednost **null**

METODE COLLECTION
→ `boolean add(T element)`
→ `boolean addAll(Collection<T> collection)`
→ `boolean contains(T element)`
→ `boolean containsAll(Collection<T> collection)`
→ `boolean remove(T element)`
→ `boolean removeAll(Collection <T> collection)`
→ `boolean retainAll(Collection<T> collection)`
→ `void clear()`
→ `boolean isEmpty()`
→ `int size()`
→ `Iterator<T> iterator()`
→ `String toString()`
→ `Object[] toArray()`

`interface List<T> extends Collection<T>`

LIST
→ tudi metode ki sprejmejo **indeks**
→ nekatere metode so redefinirane

KAZALCI
→ kazalec tipa B lahko priredimo spremenljivki A
→ **obratno ne** smemo napaka pri prevajanju
→ metodi ki sprejme tip A lahko podamo tudi tip B
→ **prevajalnik** vidi samo deklaracije
→ **izvajalnik** pa pozna tipe v času izvajanja
→ ker prevajalnik **ne ve** na kaj kaže in imamo spremenljivke **tipa A** morajo biti **metode** v katerih jih uporabljamo **definirane** že v razredu **A**

razred B je podrazred razreda A

```
A a = new A(...);
B b = new B(...);
A p = b;
```

spremenljivka p je deklarirana kot spremenljivka tipa A
v času izvajanja pa p kaže na objekt tipa B
spremenljivka p ima potemtakem dva različna tipa
tip v času prevajanja: A
tip v času izvajanja: B

ABSTRAKTNA METODA
→ metoda z **praznim telesom**: `public abstract`
→ če metoda abstraktna more bit tudi **razred** `public abstract class`
→ lahko ima **konstruktor** vendar **ne moremo** ustvariti objektov iz abstraktnega razreda lahko kličemo samo iz podrazreda z **super**

RAZRED OBJECT
→ nadrazred vseh razredov
→ pomembne metode ki jih smiselno **redefiniramo**

```
public String toString()
public boolean equals(Object obj)
public int hashCode()
```

metoda toString
→ vrne niz ki predstavlja **tip** in **pomnilniški naslov** objekta v šestnajstiškem številskem sistemu
→ velja povsod kjer ni redefinirana
→ po navadi redefiniramo da poda ključne podatke o objektu
→ metode **print** imajo različico ki vsebuje klic toString zato lahko **krajšamo**
→ return String

metoda equals
→ return boolean
→ ko kazalca **this** in **object** kažeta na isti objekt
→ redefiniramo da vrne true če imata isto **vrednost atributov**

metoda hashCode
→ return int
→ vrne pomnilniški naslov
→ redefiniramo da vrne število ki čim bolj enolično določa objekt
→ **usklajena z equals**
→ zaželeno tudi obratno

rezultat metode hashCode ponavadi izračunamo kot

$$h = p_1 h(a_1) + p_2 h(a_2) + \ldots + p_n h(a_n)$$

$p_1, \ldots, p_n$ so praštevila
$h(a_1), \ldots, h(a_n)$ so rezultati metode hashCode za posamezne atribute

```
int:    h(a_i) = Integer.hashCode(a_i)
double: h(a_i) = Double.hashCode(a_i)
char:   h(a_i) = Character.hashCode(a_i)
```

$a.equals(b) \implies (a.hashCode() == b.hashCode())$
zaželeno je, da čimvečkrat velja tudi $\impliedby$

GENERIKI
→ T je nek **generični tip**
→ napišemo: `<T>`
→ **določi** kateri tip je
→ lahko več tipov

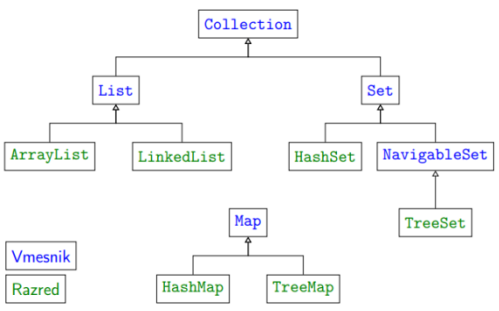
če narobe pretvorimo objekt razreda v podrazred dobimo ClassCastException

add: doda element kamorkoli v **collectionu** ampak na konec v **listu**

VSEBOVALNIKI
→ podatkovna struktura namenjena **hranjenju elementov**
→ hierarhija vmesnikov in razredov v `java.util`
→ vsi so **generični**
→ tudi tabela a ne spada k tej hierarhiji
→ tri main tipi: seznam, množica, slovar kinda vibes
→ **seznam**: po **indeksih** in elementi se **lahko ponavljajo**
→ **množica**: ne morejo se podvajati in **položaj ni določen**
→ **slovar**: **ključi** se **ne** podvajajo **vrednosti** se **lahko**

COLLECTION
→ skupni vmesnik za **zbirke** <T>
→ deklarirana množica **metod** in njihovo obnašanje
→ **podvmesniki** lahko to redefinirajo

→ **add**: doda elemente
→ **contains**: true ko equals
→ **remove**: odstrani glede na metodo equals
→ **retain**: ohrani samo tiste ki enaki glede na equals
→ **clear**: odstrani vse
→ **is empty**: true če prazna
→ **int size**: število elementov
→ **iritator**: vrne iritator po elementih zbirke this
→ **to array**: izdela vrne tabelo



METODE LIST
→ `void add(int index, T element)`
→ `T get(int index)`
→ `int indexOf(Object obj)`
→ `T remove(int index)`
→ `void sort(Comparator comp)`
→ `static List of(T... elements)`

- → **add**: na indeks ostale zamakne
- → **get**: vrne element na indeksu
- → **int index of**: vrne indeks prvega elementa equals
- → **remove**: odstrani element na indeksu in ga vrne
- → **void sort**: uredi glede na **comperator** če ni nastavljen naravno urejenost
- → **static list**: nespremenljiv seznam

ArrayList in LinkedList

- → **array list**: kot raztegljiva tabela
- → **linked list**: veriga elementov
- → **dvosmerno** povezana z **kazalci**
- → dostop po indeksih neučinkovit
- → dodajamo in odvzemamo lahko **kjerkoli** medtem ko pri array listu učinkovito samo na **koncu** in **začetku**
- → linked list uporabljamo če veliko **dodajanj** in **odvzemanj** otherwise array list.

SET

- → skupni vmesnik za **množice**
- → preverjanje podvojitev odvisno od **implementacije**
- → pri `HashSet` temelji na `equals` in `hashCode`
- → pri `TreeSet` temelji na vmesnikih `Comparable` in `Comperator`
- → isto ima **static** metodo kot list samo da set

HashSet in TreeSet

- → **hash set:** množica z **zgoščeno tabelo**
- → če `equals` more tudi `hashCode` veljat
- → **tree set**: urejena množica
- → dvojiško iskalno drevo
- → metoda equals se **ne uporablja**
- → podvojitve se preverjajo z metodo **compare**
- → `x.compareTo(y) == 0`

NavigableSet

first in **last** vrneta **najmanjši** oz **največji** element množice

floor in **celing** vrneta največji oz najmanjši element ki ni **večji** oz **manjši** od podanega oz null če ga ni

- → `T first()`
- → `T floor(T element)`
- → `NavigableSet headSet(T element, boolean inclusive)`
- → `T last()`
- → `T ceiling(T element)`
- → `NavigableSet tailSet(T element, boolean inclusive)`

headSet če **inclusive true:** množica elemntov ki so manjši ali enaki in če **false** strogo manjši oz obratno če je **tailSet**

če uporabimo konstruktor `TreeSet()`, potem mora razred, ki mu pripadajo elementi, implementirati vmesnik `Comparable`

če uporabimo konstruktor `TreeSet(Comparator<T> comp)`, potem je urejenost določena z metodo `compare` objekta `comp`

METODE MAP

- → `V put(K key, V value)`
- → `V get(Object key)`
- → `boolean containsKey(Object key)`
- → `V remove(Object key)`
- → `boolean isEmpty()`
- → `void clear()`
- → `int size()`
- → `Set keySet()`
- → `Set<Map.Entry> entrySet()`
- → `Collection values()`

- → **put:** doda par ključ vrednost oziroma zamenja vrednost če ključ obstaja
- → **get:** vrne vrednost ki pripada ključu oz null če ključa ni
- → **contains key:** true če ključ je
- → **remove:** odstrani ključ
- → int size: število parov
- → **set**: vrne množico ključev oz množico parov ključ vrednost
- → **collection**: vrne zbirko vrednosti

MAP

`interface Map<K, V>`

K: tip ključev

V: tip vrednosti

- → skupni vmesnik za **slovarje**
- → **hash map**: unikatnost ključev glede na **equals** nobena dva enaka
- → **tree map**: unikatnost ključev glede na **primerjalnik** nobena dva enaka

MEDKLIC ?

- → generična metoda
- → odvisna od generičnega tipa
- → deklaracija tipa pred izhodnim tipom

`<A extends B>`: A mora biti podtip tipa B

`<A super B>`: A mora biti nadtip tipa B

ITERABLE

- → vrne objekt tipa Iterator, ki omogoča zaporedni sprehod po vsebovalniku this

VMESNIKI

`Comparable`
`Comparator`
`Iterable`
`Iterator`

- → **abstraktne** metode
- → **statične** metode
- → **privzete** metode
- → statične **konstante**
- → vsi elementi so javno dostopni
- → relevantna dostopna določila se dodajo samodejno
- → razred **implementira** vmesnik če so v njem definirane vse abstraktne metode vmesnika
- → razred lahko podrazred največ enega razreda implementira pa lahko poljubno mnogo vmesnikov
- → podrazred: `extends`
- → vmesnik: `implements`
- → `interface`

ITERATOR

- → objekt ki omogoča zaporedni **sprehod** po elementih vsebovalnika
- → true če **obstaja** še kak element: `hasNext`
- → **naslednji** element vsebovalnika: `next`

```
Iterator<Tip> iterator = new MojIterator<>(vsebovalnik);
while (iterator.hasNext()) {
    Tip element = iterator.next();
    // obdelaj element
}
```

COMPERATOR

- → **alternativna** urejenost objektov
- → naravna je ena sama teh lahko več
- → implementiramo v ločenem razredu
- → **notranji** razred

COMPARABLE

- → metoda: `compareTo`
- → definira **naravno urejenost**
- → **negativno** število če en pred drugim
- → **nič** če sta enaka
- → **pozitivno** število če za prvim

samo bullet notes real programi so v eni mapi

PROGRAMI IZ PREDAVANJ

RANDOM NOTES:

- → `public static void main(String[] args)`
- → pazi na **robne primere**!
- → program število deliteljev
- → metoda **korena**: `Math.sqrt()`
- → metoda **zaokroži**: `Math.round()`

ŠTEVILO DELITELJEV

- → program **prebere število** n
- → za vse od 1 do n **preverimo**
- → če je deljivo **izpišemo**

```
int stevilo = sc.nextInt();
int stDeliteljev = 0;
for (int d = 1;  d <= stevilo;  d++) {
    if (stevilo % d == 0) {
        System.out.println(d);
        stDeliteljev++;
    }
}
System.out.println();
System.out.println(stDeliteljev);
```

PRAŠTEVILA

1. preverimo tako kot prej če ima **samo dva** delitelja vendar traja dolgo
2. bolj učinkovito če delimo z vsemi od 2 do n – 1 če se **izide ni** praštevilo
3. še boljše delimo samo z **lihimi**
4. lahko gremo samo do **korena** kandidata
5. fino naredit metodo
6. Eratostenovo sito

DVOJIŠKO ISKANJE

```
public static int poisci(int[] t, int x) {
    int lm = 0;                 // leva meja
    int dm = t.length - 1;      // desna meja
    while (lm <= dm) {
        int s = (lm + dm) / 2;
        if (t[s] == x) {  // element smo našli!
            return s;
        }
        if (t[s] < x) {
            lm = s + 1;
        } else {
            dm = s - 1;
        }
    }
    return -1;   // elementa ni v tabeli
}
```

```
boolean[] sestavljeno = new boolean[n + 1];
int meja = (int) Math.round(Math.sqrt(n));

int p = 2;
while (p <= meja) {
    for (int i = 2 * p;  i <= n;  i += p) {
        sestavljeno[i] = true;
    }
    do {
        p++;
    } while (p <= meja && sestavljeno[p]);
}

for (int i = 2;  i <= n;  i++) {
    if (!sestavljeno[i]) {
        System.out.println(i);
    }
}
```

eden od najučinkovitejših postopkov za iskanje praštevil

pričnemo s $p = 2$

označimo vse večkratnike števila p od $2p$ do n

p nastavimo na prvo neoznačeno število, večje od trenutnega p

označimo vse večkratnike števila p od $2p$ do n

p nastavimo na prvo neoznačeno število, večje od trenutnega p

označimo vse večkratnike števila p od $2p$ do n

...

ponavljamo do $p \leq \sqrt{n}$

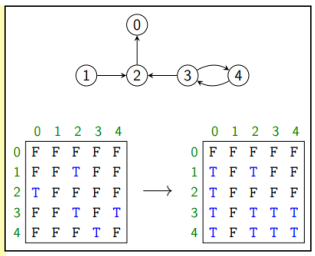
neoznačena števila med 2 in n so praštevila

FIBONACCI

```java
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    int[] memo = new int[n + 1];
    System.out.println(f(n, memo));
}
public static int f(int n, int[] memo) {
    if (n <= 1) {
        return n;
    }
    if (memo[n] > 0) {
        return memo[n];
    }
    int pp = f(n - 2, memo);
    int p = f(n - 1, memo);
    memo[n] = pp + p;
    return memo[n];
}
```

DOSEGLJIVOST VOZLIŠČ V GRAFU

→ vhod: **število vozlišč**
→ izhod: **množica parov** a in b tako da iz vozlišča a dosežemo b
→ množico povezav predstavimo z **boolean** tabelo
→ minus n obhod celoten graf
→ v **r** tem obhodu zgradimo tabelo v kateri so razdalje do **r plus 1**
→ pretvarjanje 2D tabele: `Arrays.deepToString`



RAZRED CAS

→ objekt predstavlja neko **uro**

```java
import java.util.Arrays;

public class Dosegljivost {

    Run | Debug
    public static void main(String[] args) {

        /*
              0
              ^
              |
        1 --> 2 <-- 3 <--> 4
        */

        boolean[][] graf = {
            {false, false, false, false, false},
            {false, false,  true, false, false},
            { true, false, false, false, false},
            {false, false,  true, false,  true},
            {false, false, false,  true, false},
        };

        boolean[][] dosegljivost = izracunaj(graf);
        System.out.println(Arrays.deepToString(dosegljivost));
    }

    public static boolean[][] izracunaj(boolean[][] graf) {
        int stVozlisc = graf.length;
        boolean[][] dosegljivost = new boolean[stVozlisc][stVozlisc];
        for (int i = 0;  i < stVozlisc;  i++) {
            for (int j = 0;  j < stVozlisc;  j++) {
                dosegljivost[i][j] = graf[i][j];
            }
        }

        for (int r = 1;  r < stVozlisc;  r++) {
            dosegljivost = posodobi(dosegljivost, graf);
        }
        return dosegljivost;
    }
```

to je naša default tabela

tukaj samo **prekopiramo** tabelo ki nam je dana

tukaj dejansko dobimo končno tabelo

```java
public static boolean[][] posodobi(boolean[][] dosegljivost, boolean[][] graf) {
    int stVozlisc = graf.length;
    boolean[][] novaDosegljivost = new boolean[stVozlisc][stVozlisc];
    for (int i = 0;  i < stVozlisc;  i++) {
        for (int j = 0;  j < stVozlisc;  j++) {
            novaDosegljivost[i][j] = obstaja(dosegljivost, graf, i, j);
        }
    }
    return novaDosegljivost;
}

public static boolean obstaja(boolean[][] dosegljivost, boolean[][] graf, int i, int j) {
    if (dosegljivost[i][j]) {
        return true;
    }
    int stVozlisc = graf.length;
    for (int k = 0;  k < stVozlisc;  k++) {
        if (dosegljivost[i][k] && graf[k][j]) {
            return true;
        }
    }
    return false;
}
```

graf **novega** obhoda

preverimo ali obstaja povezava ali ne vrne **true** ali **false**

dejansko preveri ali obstaja vozlišče **k povezano** z **i** in **j**

```java
/*
 * Vrne uro trenutka <this>.
 */
public int vrniUro() {
    return this.ura;
}

/*
 * Vrne minuto trenutka <this>.
 */
public int vrniMinuto() {
    return this.minuta;
}
```

tukaj imamo **getter** za **uro** in **minuto**

```java
    /*
     * Vrne razliko (v minutah) med "casovnim trenutkom <this> in "casovnim
     * trenutkom <drugi>.
     */
    public int razlikaVMin(Cas drugi) {
        return (this.ura - drugi.ura) * 60 + (this.minuta - drugi.minuta);
    }
}
```

```java
public class Cas {

    private int ura;
    private int minuta;

    // true: toString vrne zapis "casa v 12-urnem formatu;
    // false: toString vrne zapis "casa v 24-urnem formatu
    private static boolean zapis12 = false;

    /*
     * Inicializira objekt, ki predstavlja "casovni trenutek s podano uro in
     * minuto.
     */
    public Cas(int h, int min) {
        this.ura = h;
        this.minuta = min;
    }

    /*
     * Nastavi 12-urni zapis (da == true) oziroma 24-urni zapis (da == false).
     */
    public static void nastaviZapis12(boolean da) {
        zapis12 = da;
    }
```

to potem v metodi spodi dejansko naštimano pretvorba

redefinirana metoda za **izpis** ure

```java
    /*
     * Vrne predstavitev trenutka <this> v obliki niza.
     */
    public String toString() {
        if (zapis12) {
            String pripona = (this.ura < 12) ? ("AM") : ("PM");
            int h = (this.ura + 11) % 12 + 1;
            return String.format("%d:%02d %s", h, this.minuta, pripona);
        }
        return String.format("%d:%02d", this.ura, this.minuta);
    }
```

odvisna od načina zapisa **evropski** ali **ameriški**

```java
    /*
     * Nastavi uro trenutka <this> na <h>.
     */
    public void nastaviUro(int h) {
        this.ura = h;
    }

    /*
     * Nastavi minuto trenutka <this> na <min>.
     */
    public void nastaviMinuto(int min) {
        this.minuta = min;
    }
```

deljenje po **modulu 12**

setter ki **spremeni** objekt ne naredi novega

```java
    /*
     * Vrne nov objekt, ki predstavlja trenutek, ki se zgodi <h> ur in <min>
     * minut kasneje kot trenutek <this>.
     */
    public Cas plus(int h, int min) {
        int noviCas = 60 * (this.ura + h) + (this.minuta + min);
        noviCas = (noviCas % 1440 + 1440) % 1440;
        int novaUra = noviCas / 60;
        int novaMinuta = noviCas % 60;
        return new Cas(novaUra, novaMinuta);
    }
```

boljši primer setterja ki ustvari **nov objekt** in ne spremeni starega!

```java
    /*
     * Trenutku <cas> pri"steje <h> ur in <mi
     */
    public static void pristejStatic(Cas cas, int h, int min) {
        int noviCas = 60 * (cas.ura + h) + (cas.minuta + min);
        noviCas = (noviCas % 1440 + 1440) % 1440;
        cas.ura = noviCas / 60;
        cas.minuta = noviCas % 60;
    }
```

static metodo moramo izven razreda klicati kot **Razred.**metoda(…) !

spodnja metoda **ni static** zato jo kličemo drugače

```java
    /*
     * Trenutku <this> pri"steje <h> ur in <min> minut.
     */
    public void pristej(int h, int min) {
        int noviCas = 60 * (this.ura + h) + (this.minuta + min);
        noviCas = (noviCas % 1440 + 1440) % 1440;
        this.ura = noviCas / 60;
        this.minuta = noviCas % 60;
    }
```

kličemo kot objekt.metoda saj se **objekt** skopira v **this**

```java
    /*
     * Vrne true natanko tedaj, ko objekta <this> in <drugi> predstavljata
     * isti "casovni trenutek.
     */
    public boolean jeEnakKot(Cas drugi) {
        return (this.ura == drugi.ura && this.minuta == drugi.minuta);
    }

    /*
     * Vrne true natanko tedaj, ko objekt <this> predstavlja "casovni
     * trenutek, ki kronolo"sko sodi pred objekt <drugi>.
     */
    public boolean jeManjsiOd(Cas drugi) {
        return this.ura < drugi.ura || (this.ura == drugi.ura && this.minuta < drugi.minuta);
    }

    /*
     * Vrne true natanko tedaj, ko objekt <this> predstavlja "casovni
     * trenutek, ki kronolo"sko sodi pred objekt <drugi> ali pa mu je enak.
     */
    public boolean jeManjsiAliEnakOd(Cas drugi) {
        return this.jeManjsiOd(drugi) || this.jeEnakKot(drugi);
    }
}
```

podobno kot **equals** ampak ne isto

RAZRED VEKTOR

→ kot **raztegljiva tabela**
→ po potrebi **povečujemo** kapaciteto
→ atributa: **tabela** z elementi in **število elementov** v tabeli
→ tabelo ustvarimo z določeno **začetno kapaciteto**

```
public class VektorInt {

    // privzeta za"cetna kapaciteta vektorja
    private static final int ZACETNA_KAPACITETA = 10;

    // tabela elementov
    private int[] elementi;

    // dejansko "stevilo elementov (<= elementi.length)
    private int stElementov;

    /*
     * Inicializira objekt, ki predstavlja vektor s privzeto za"cetno kapaciteto.
     */
    public VektorInt() {
        this(ZACETNA_KAPACITETA);
    }

    /*
     * Inicializira objekt, ki predstavlja vektor s podano za"cetno kapaciteto.
     */
    public VektorInt(int kapaciteta) {
        this.elementi = new int[kapaciteta];
        this.stElementov = 0;
    }
```

tak klic veljaven samo v **konstruktorju**

kliče konstruktor od spodaj zato da ga ne pišemo dvakrat

to se zgodi če **ne podamo parametrov** bo prvotna tabela velika 10

prvotna tabela **poljubno velika**

```
    /*
     * Vrne "stevilo elementov vektorja <this>
     */
    public int steviloElementov() {
        return this.stElementov;
    }

    /*
     * Vrne element na podanem indeksu.
     */
    public int vrni(int indeks) {
        return this.elementi[indeks];
    }
```

ko dodajamo elemente št elementov za 1 gor

```
    /*
     * Element na podanem indeksu nastavi na podano vrednost.
     */
    public void nastavi(int indeks, int vrednost) {
        this.elementi[indeks] = vrednost;
    }

    /*
     * Doda element s podano vrednostjo na konec vektorja (na indeks
     * this.stElementov).
     */
    public void dodaj(int vrednost) {
        this.poPotrebiPovecaj();
        this.elementi[this.stElementov] = vrednost;
        this.stElementov++;
    }

    /*
     * Element s podano vrednostjo vstavi na podani indeks.
     */
    public void vstavi(int indeks, int vrednost) {
        this.poPotrebiPovecaj();
        for (int i = this.stElementov - 1;  i >= indeks;  i--) {
            this.elementi[i + 1] = this.elementi[i];
        }
        this.elementi[indeks] = vrednost;
        this.stElementov++;
    }

    /*
     * Odstrani element na podanem indeksu.
     */
    public void odstrani(int indeks) {
        for (int i = indeks;  i < this.stElementov - 1;  i++) {
            this.elementi[i] = this.elementi[i + 1];
        }
        this.stElementov--;
    }
```

ostale elemente premakne za 1 v **desno**

premakne za 1 v **levo**

```
    /*
     * "Ce je vektor <this> poln, pove"ca njegovo kapaciteto za faktor 2.
     */
    private void poPotrebiPovecaj() {
        if (this.stElementov >= this.elementi.length) {
            // ustvari novo, ve"cjo tabelo in vanjo skopiraj elemente iz stare
            // tabele
            int[] stariElementi = this.elementi;
            this.elementi = new int[2 * stariElementi.length];
            for (int i = 0;  i < this.stElementov;  i++) {
                this.elementi[i] = stariElementi[i];
            }
        }
    }
```

preveri če je potrebujemo **povečat** tabelo

```
    /*
     * Vrne vsebino vektorja v obliki niza [e_0, e_1, ..., e_{n-1}].
     * Uporablja lepljenje nizov.
     */
    public String toString0() {
        String str = "[";
        for (int i = 0;  i < this.stElementov;  i++) {
            if (i > 0) {
                str += ", ";
            }
            str += this.elementi[i];
        }
        str += "]";
        return str;
    }
```

ker so nizi nespremenljivi uporabimo **StringBuilder**

```
    /*
     * Vrne vsebino vektorja v obliki niza [e_0, e_1, ..., e_{n-1}].
     * Uporablja razred StringBuilder.
     */
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("[");
        for (int i = 0;  i < this.stElementov;  i++) {
            if (i > 0) {
                sb.append(", ");
            }
            sb.append(this.elementi[i]);
        }
        sb.append("]");
        return sb.toString();
    }
}
```

RAZRED ŠTUDENT

→ primer **dedovanja**

```
public class Student {
    private String ip;            // ime in priimek
    private String vpisna;        // vpisna "stevilka
    private int stroskiBivanja;   // stro"ski bivanja

    public Student(String ip, String vpisna, int stroskiBivanja) {
        this.ip = ip;
        this.vpisna = vpisna;
        this.stroskiBivanja = stroskiBivanja;
    }

    public String vrniIP() {
        return this.ip;
    }

    public int stroski() {
        return this.stroskiBivanja;
    }
}
```

podrazred podeduje vse **atribute** in obe **metodi**

redefinirali metodo stroški ker izredni študenti drugačne stroške

```
public class IzredniStudent extends Student {
    private int solnina;

    public IzredniStudent(String ip, String vpisna, int stroskiBivanja, int solnina) {
        super(ip, vpisna, stroskiBivanja);
        this.solnina = solnina;
    }

    @Override
    public int stroski() {
        return (super.stroski() + this.solnina);
    }
```

extends pomeni podrazred

konstruktor kliče konstruktor iz nadrazreda s predpono **super** in nato **doda** atribut

dodali smo atribut **šolnina** in obvezno **konstruktor** drugače ne bi mogli ustvariti objekta

tukaj kličemo **nespremenjeno** metodo stroški iz nadrazreda in ji dodamo nekaj zato predpona **super**

HIERARHIJA LIKOV

→ program ki obdeluje **pravokotnike kvadrate** in **kroge**
→ podatke hranimo v tabeli
→ čim bolj **poenoteno**
→ liki imajo **različne atribute**

```
public class Glavni {

    Run | Debug
    public static void main(String[] args) {…

    /** Izpi"se podatke o vseh likih. */
    public static void izpisiPodatke(Lik[] liki) {
        for (Lik lik: liki) {
            System.out.printf("%s | p = %.1f, o = %.1f%n",
                    lik.toString(),
                    lik.ploscina(),
                    lik.obseg());
        }
```

če želimo da se to prevede mora imeti razred lik metode **ploščina** in **obseg**

```
    /** Vrne kazalec na lik z najve"cjo plo"s"cino oziroma null, "ce je tabela
     * prazna. */
    public static Lik likZNajvecjoPloscino(Lik[] liki) {
        Lik najLik = null;
        double najPloscina = 0.0;
        for (Lik lik: liki) {
            double ploscina = lik.ploscina();
            if (najLik == null || ploscina > najPloscina) {
                najPloscina = ploscina;
                najLik = lik;
            }
        }
        return najLik;
```

razred **Glavni** je hierarhično **nad** razredom **Lik**

```java
/** Vrne kazalec na pravokotnik z najve"cjo "sirino oziroma null, "ce
 * tabela ne vsebuje nobenega pravokotnika. */
public static Pravokotnik pravokotnikZNajvecjoSirino(Lik[] liki) {
    Pravokotnik naj = null;
    for (Lik lik: liki) {
        if (lik instanceof Pravokotnik) {
            Pravokotnik p = (Pravokotnik) lik;
            if (naj == null || p.vrniSirino() > naj.vrniSirino()) {
                naj = p;
            }
        }
    }
    return naj;
}
```

še vedno v razredu **Glavni**

```java
public abstract class Lik {

    /* Vrne obseg lika <this>. */
    public abstract double obseg();

    /* Vrne plo"s"cino lika <this>. */
    public abstract double ploscina();

    /* Vrne predstavitev lika <this> v obliki niza. */
    public String toString() {
        // poklicale se bodo konkretne implementacije metod v podrazredih;
        // izbira metode je odvisna od tipa objekta <this> v "casu izvajanja
        return String.format("%s [%s]", this.vrsta(), this.podatki());
    }

    /* Vrne niz, ki predstavlja vrsto lika <this>. */
    public abstract String vrsta();

    /* Vrne niz, ki vsebuje podatke o liku <this>. */
    public abstract String podatki();
}
```

abstraktne metode brez telesa

zato ker bo izpis podoben metodo toString definiramo že tu

```java
public class Pravokotnik extends Lik {

    private double sirina;
    private double visina;

    public Pravokotnik(double sirina, double visina) {
        this.sirina = sirina;
        this.visina = visina;
    }

    public double vrniSirino() {
        return this.sirina;
    }

    @Override
    public double ploscina() {
        return this.sirina * this.visina;
    }

    @Override
    public double obseg() {
        return 2 * (this.sirina + this.visina);
    }

    @Override
    public String vrsta() {
        return "pravokotnik";
    }

    @Override
    public String podatki() {
        return String.format("širina = %.1f, višina = %.1f",
            this.sirina, this.visina);
    }
}
```

razred **Pravokotnik**

redefiniramo **abstraktne metode** iz razreda Lik

to so metode ki jih rabimo v metodi **toString** iz Lika

rabimo getter za širino ker ne moremo uporabit v Kvadratu

```java
public class Kvadrat extends Pravokotnik {

    // atributa <sirina> in <visina> se dedujeta

    public Kvadrat(double stranica) {
        super(stranica, stranica);
    }

    // Ploščina in obseg se izračcunata
    // na enak način kot pri pravokotniku,
    // zato nam metod <ploscina>
    // in <obseg> ni treba redefinirati.

    @Override
    public String vrsta() {
        return "kvadrat";
    }

    @Override
    public String podatki() {
        return String.format("stranica = %.1f",
            this.vrniSirino());
    }
}
```

podrazred Pravokotni ka razred **Kvadrat**

kličemo konstruktor iz pravokotnika

vedno se kliče metoda ki pripada liku **v času izvajanja !**

```java
public class Krog extends Lik {

    private double polmer;

    public Krog(double polmer) {
        this.polmer = polmer;
    }

    @Override
    public double ploscina() {
        return Math.PI * this.polmer * this.polmer;
    }

    @Override
    public double obseg() {
        return 2.0 * Math.PI * this.polmer;
    }

    @Override
    public String vrsta() {
        return "krog";
    }

    @Override
    public String podatki() {
        return String.format("polmer = %.1f", this.polmer);
    }
}
```

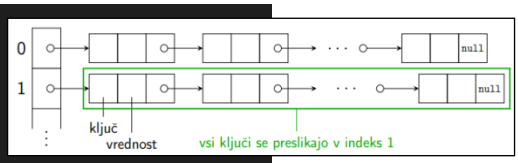
razred **Krog**

SLOVAR

- → preslikuje **ključe** v **vrednosti**
- → **posplošitev** tabele
- → tip ključa: **K**
- → tip vrednosti: **V**
- → naiven princip **na prosojnicah**!
- → uporabimo **zgoščeno tabelo**

indekse računamo z **hash code ključa** in če ima več ključev isti hash code so na istem **indeksu** lahko pa imajo **različne vrednosti**

```java
public class Slovar<K, V> {

    private static class Vozlisce<K, V> {
        K kljuc;
        V vrednost;
        Vozlisce<K, V> naslednje;

        Vozlisce(K kljuc, V vrednost, Vozlisce<K, V> naslednje) {
            this.kljuc = kljuc;
            this.vrednost = vrednost;
            this.naslednje = naslednje;
        }
    }

    private static final int VELIKOST_TABELE = 97;

    private Vozlisce<K, V>[] podatki;

    public Slovar() {
        this(VELIKOST_TABELE);
    }

    @SuppressWarnings("unchecked")
    public Slovar(int velikostTabele) {
        this.podatki = (Vozlisce<K, V>[]) new Vozlisce[velikostTabele];
```



verige so iz objektov tipa **vozlišče**

če ga ni **null**

naslednje kaže na objekt tipa vozlišče z istim **indeksom**

tukaj Slovar če konstruktorju ne podamo parametrov kliče konstruktor spodaj

računanje indeksa

```java
private int indeks(K kljuc) {
    int n = this.podatki.length;
    return ((kljuc.hashCode() % n) + n) % n;
}
```

vrne vrednost glede na ključ

```java
public V vrni(K kljuc) {
    Vozlisce<K, V> vozlisce = this.poisci(kljuc);
    if (vozlisce == null) {
        return null;
    }
    return vozlisce.vrednost;
}
```

privatna metoda kjer se sprehodimo po ključih z **istim indeksom**

```java
private Vozlisce<K, V> poisci(K kljuc) {
    int indeks = this.indeks(kljuc);
    Vozlisce<K, V> vozlisce = this.podatki[indeks];
    while (vozlisce != null && !vozlisce.kljuc.equals(kljuc)) {
        vozlisce = vozlisce.naslednje;
    }
    return vozlisce;
}
```

primerjamo z ključi v posameznih vozliščih z metodo **equals**

primer bi bil **telefonski imenik** ravno tako na prosojnicah

naredimo tudi **Iterator** po **ključih**

```java
public void shrani(K kljuc, V vrednost) {
    Vozlisce<K, V> vozlisce = this.poisci(kljuc);
    if (vozlisce != null) {
        vozlisce.vrednost = vrednost;
    } else {
        int indeks = this.indeks(kljuc);
        vozlisce = new Vozlisce<K, V>(kljuc, vrednost, this.podatki[indeks]);
        this.podatki[indeks] = vozlisce;
    }
}
```

preverimo ali **že obstaja** vozlišče z takim ključem če obstaja **posodobimo** če ne ustvarimo **novo vozlišče** na začetku verige

v razredu `IteratorPoKljucih` vzdržujemo sledeče atribute:

- `slovar`: slovar, po katerem se sprehajamo
- `indeks`: indeks trenutnega elementa tabele kazalcev
- `vozlisce`: kazalec na trenutno vozlišče v verigi
- `stevec`: globalni indeks trenutnega vozlišča

OPOMBE K MAIN PROGRAMOM

RAZRED CAS

→ metoda equals podobna vendar ne ista
→ preverit moramo ali kazalec pri objektu ki ga preverjamo kaže na **objekt čas** ali na kaj drugega z **enakimi atributi**!

```
@Override
public boolean equals(Object drugi) {
    if (this == drugi) {
        return true;
    }
    if (!(drugi instanceof Cas)) {
        return false;
    }
    Cas drugiCas = (Cas) drugi;
    return (this.ura == drugiCas.ura &&
            this.minuta == drugiCas.minuta);
}
```

→ metoda `hashCode`

```
@Override
public int hashCode() {
    return (17 * Integer.hashCode(this.ura) +
            31 * Integer.hashCode(this.minuta));
}
```

Vmesnik COMPARABLE

→ **naravna urejenost** ure
→ če sta uri enaki po minutah

```
class Cas implements Comparable<Cas> {
    ...
    @Override
    public int compareTo(Cas drugi) {
        if (this.ura == drugi.ura) {
            return this.minuta - drugi.minuta;
        }
        return this.ura - drugi.ura;
    }
}
```

RAZRED VEKTOR

→ vektor z elementi poljubnega **referenčnega** tipa
→ tabelo naredimo tipa **object**
→ uporabnik mora poznati tipe shranjene v vektorju
→ boljše z **generiki**
→ generiki sami dodajo ustrezne pretvorbe tipov

```
Vektor<String> vs = new Vektor<String>();
vs.dodaj("Dober dan!");
// vs.dodaj(42); // napaka pri prevajanju

Vektor<Integer> vi = new Vektor<Integer>();
vi.dodaj(42);
// vi.dodaj("Dober dan!"); // napaka pri prevajanju

String str = vs.vrni(0);    // prevajalnik sam doda pretvorbo
// Integer n = vs.vrni(0); // napaka pri prevajanju
```

→ pišemo povsod `<T>`
→ potem lahko delamo vektorje **katerihkoli** generičnih tipov

vmesnik Iterable v razredu Vektor

```
public class Vektor<T> implements Iterable<T> {
    ...
    @Override
    public Iterator<T> iterator() {
        return new IteratorCezVektor<T>(this);
    }
}
```

→ `IteratorCezVektor`
→ potrebuje **dostop** do vektorja da se **sprehodi** po elementih
→ ob izdelavi posredujemo **kazalec this**
→ definiramo ga kot **notranji razred**
→ indeks **hrani** indeks trenutnega elementa
→ **has next** je true ko indeks manjši od št elementov
→ **next** vrne element in poveča indeks

COMPERATOR PRIMER

```
public class Oseba {
    ...
    private static class PrimerjalnikPoStarosti
            implements Comparator<Oseba> {

        @Override
        public int compare(Oseba prva, Oseba druga) {
            return prva.letoRojstva - druga.letoRojstva;
        }
    }

    public static Comparator<Oseba> poStarosti() {
        return new PrimerjalnikPoStarosti();
    }
}
```

```
public class Slovar<K, V> implements Iterable<K> {
    ...
    @Override
    public Iterator<K> iterator() {
        return new IteratorPoKljucih<K, V>(this);
    }

    private static class IteratorPoKljucih<K, V>
            implements Iterator<K> {
        ...
    }
}
```

Iterator po slovarju

primer

```
Vektor<Integer> vektor = new Vektor<>();
...
for (Integer element: vektor) {
    System.out.println(element);
}
```

gornja koda se dejansko izvede takole:

```
Vektor<Integer> vektor = new Vektor<>();
...
Iterator<Integer> iterator = vektor.iterator();
while (iterator.hasNext()) {
    Integer element = iterator.next();
    System.out.println(element);
}
```