OAPS I – predavanja:

1. Ponovitev jave:

```
public class Point
{
    protected double x, y;

    public Point (double x, double y)
    {
            this.x = x;
            this.y = y;
    }

    public double getx()
    {
            return x;
    }

    public double gety()
    {
            return y;
    }

    public Point getLocation()
    {
            return new Point(x,y);
    }

    public void setLocation(double x, double y)
    {
            this.x = x;
            this.y = y;
    }

    public void translate(double dx, double dy)
    {
            x += dx;
            y += dy;
    }

    public String toString()
    {
            return new String("("+(float)x+","+(float)y+")");
    }
}
```

- **deklaracija razreda:**
  - glava razreda
  - deklaracija atributov: private ali protected
  - deklaracija metod:
    - praviloma public:
      - setty metode (mutator methods): vpis oz. spremembe vrednosti atributov. Set x
      - getty metode (accessor methods): vračanje vrednosti atributov. Get x
    - lahko tudi private: pomožne metode, ki jih uporabljajo druge metode v tem razredu

- **kreiranje objektov:**
  - Dva koraka:
    - deklaracija objekta - Point p;

- generiranje objekta - p = new Point(2,3);
- oba koraka lahko združimo: Point p = new Point(2,3);

Primer:

```
Point p=new Point(2,3);
Point q=p.getLocation();
q.translate(5,-1);
q=p;
```

- dedovanje:

Nek razred lahko podeduje atribute in metode nekega drugega razreda:

- osnovni razred: razred, ki služi kot osnova za dedovanje - nadrazred
- izpeljan razred: razred ki deduje - podrazred

Deklaracija izpeljanega razreda:

- osnovni razred mora že obstajati
- v glavi uporabimo rezervirano besedo extends

Deklariramo lahko dodatne atribute:

- deklariramo lahko dodatne metode
- namesto podedovanih metod lahko deklariramo drugačne metode (redefinicija)
- konstruktor podrazreda mora najprej poklicati konstruktor nadrazreda (super)

Primer:

```
public class ColoredPoint extends Point
{
    private String color = "black";

    public ColoredPoint(double x, double y, String color)
    {
            super(x,y);
            this.color = color;
    }

    public String getColor()
    {
            return color;
    }

    public void setColor(String color)
    {
            this.color = color;
    }

    public String toString()
    {
            return new String("("+(float)x+","+(float)y+","+color+")");
    }
}
```

Abstraktni razred:

Splošni nadrazred, ki služi kot osnova za izpeljavo različnih podrazredov.
Po zgradbi je podoben ostalim razredom (vsebuje deklaracije atributov in metod), le da je ena ali več metod abstraktnih (abstraktna metoda vsebuje samo glavo, nima pa telesa; abstract).

Ne moremo generirati objektov, ampak moramo najprej izpeljati ustrezne podrazrede.
V podrazredih moramo redefinirati abstraktne metode.

Primer:

```
abstract class Element
{
    public abstract boolean manjsi(Element b);
}
```

2. Algoritmi za sortiranje tabel

Razširitev razreda Element:

```
public class Student extends Element
{
    String priimek, ime;
    double povpOcena;
    public Student(String p, String i, double po)
    {
            priimek = p;
            ime = i;
            povpOcena = po;
    }

    public boolean manjsi(Element b)
    {
            Student s=(Student)b; //konverzija tipa Element v tip Student
            return this.povpOcena < s.povpOcena;
    }

    public String toString()
    {
            return priimek +" "+ime+" "+povpOcena;
    }
}
```

- Maksimalno izkoriščanje prednosti OOP (objektno usmerjenega programiranja)
- Vse algoritme bomo sprogramirali ob predpostavki, da sortiramo tabelo objektov tipa Element. Element[] a
- Podatke bomo uredili v narascajocem vrstnem redu

Razred za vse sortirne metode:

```
Public class SortiranjeObjektov
{
    Public static void straightInsertion(Element[] a)
    {
            …
    }
    //se vse ostale metode za sortiranje bodo tu naprej
}
```

- **Navadno vstavljanje:**

  - Grob opis algoritma:

```
For(int i=1; i<a.length; i++)
{
    X=a[i];
    Vstavi x na pravo mesto upostevajoc elemente a[0] do a[i];
}
```

  - Primer:

```
44 55 12 42 94 18  6 67
44 55 12 42 94 18  6 67   i=1
```

```
12 44 55 42 94 18  6 67   i=2
12 42 44 55 94 18  6 67   i=3
12 42 44 55 94 18  6 67   i=4
12 18 42 44 55 94  6 67   i=5
 6 12 18 42 44 55 94 67   i=6
 6 12 18 42 44 55 67 94   i=7
```

- Realizacija v javi:

```
public static void straightInsertion(Element[] a)
{
    int i, j;
    Element x;

    for(i=1; i<a.length; i++)
    {
            x=a[i];
            j=i-1;
            while(j>=0 && x.manjsi(a[j]))
            {
                    a[j+1]=a[j];
                    --j;
            }
            a[j+1]=x;
    }
}
```

- Testni program:

```
public class GlavniProgram
{
    public static void main(String[] args)
    {
            Student[] s = new Student[5];

            s[0]=new Student("Novak", "Janez", 8.12);
            s[1]=new Student("Nova", "Janez", 7.72);
            s[2]=new Student("Novak2", "Janez2", 9.04);
            s[3]=new Student("Novak", "Janez1", 7.52);
            s[4]=new Student("Novak1", "Janez", 8.04);

            SortiranjeObjektov.straightinsertion(s);
            for(int i=0; i<s.length; ++i)
                    System.out.println(s[i].toString());
    }
}
```

- Analiza navadnega vstavljanja:
  - Stevilo primerjanj (kolikokrat se klice metoda manjsi)
  - Stevilo premikov (kolikokrat se nek objekt premakne z ene lokacije na drugo)

<u>Stevilo primerjanj:</u>

- i-ti korak:

$C_{i\,min} = 1$ v najboljsem primeru, ko so ze posortirani
$C_{i\,max} = i$ v najslabsem primeru, ko so podatki v nasprotnem vrstnem redu
$C_{i\,avg} = i/2$ v povprecju

- v celoti:

$C_{min} = Sum[i=1, n-1]\ 1 = n-1$ n – st. elementov v tabeli
$C_{max} = Sum[i=1, n-1]\ i = 1/2\ (n2-n)$
$C_{avg} = Sum[i=1, n-1]\ i = 1/4\ (n2-n)$

<u>Stevilo premikov:</u>

- i-ti korak:

$M_{i\,avg} = C_{i\,avg} + 1 = i/2 + 1$

- v celoti:

M_{avg} = Sum[i=1, n-1] (i/2 + 1) = 1/4 (n^2 - n) + n - 1

T=o(n^2)

- **Navadno izbiranje:**
    - Grob opis algoritma:

```
for(int i=0; i<=a.length-2; ++i) //za vse elemente od prvega do predzadnjega
{
    a[k]=min a[j]; //poišci najmanjši element izmed a[i] do a[a.length-1]
    zamenjaj a[k] in a[i];
}
```

    - Primer:

```
    44 55 12 42 94 18  6 67
i=0
     6 55 12 42 94 18 44 67
i=1
     6 12 55 42 94 18 44 67
i=2
     6 12 18 42 94 55 44 67
i=3
     6 12 18 42 94 55 44 67
i=4
     6 12 18 42 44 55 94 67
i=5
     6 12 18 42 44 55 94 67
i=6
     6 12 18 42 44 55 67 94
```

    - Realizacija v javi:

```
public static void straightSelection(Element [] a)
{
    int i,j,iMin;
    Element vMin;

    for(i=0; i<=a.length-2; ++i)
    {
            iMin=i;
            vMin=a[i];

            for(j=i+1; j<=a.length-1; ++j)
                    if(a[j].manjsi(vMin))
                    {
                            iMin=j;
                            vMin=a[j];
                    }
            a[iMin]=a[i];
            a[i]=vMin;
    }
}
```

    - Analiza navadnega izbiranja:

<u>Število primerjanj:</u>

Neodvisno od zacetne permutacije vhodnih podatkov

- i-ti korak:

  C_i = n-1-i
- v celoti:

seštevamo po vseh korakih i=1,2,3....,n-2

C = sum[i=0, n-2] (n-1-i) = (n-1)+(n-2)+...+1 = 1/2n(n-1)

$T=o(n^2)$

Število premikov:
Analiza je bolj zapletena. M_{avg} je približno n(ln n + omega)

- **BubbleSort (navadno sortiranje s premenami):**
    - Grob opis algoritma:

```
for(int i=1; i<a.length; ++i)
    od desne proti levi primerjaj dva sosednja elementa in ju po potrebi zamenjaj.
```

    - Primer:

```
    44 55 12 42 94 18  6 67
i=1
     6 44 55 12 42 94 18 67
i=2
     6 12 44 55 18 42 94 67
i=3
     6 12 18 44 55 42 67 94
i=4
     6 12 18 42 44 55 67 94
i=5
     6 12 18 42 44 55 67 94
i=6
     6 12 18 42 44 55 67 94
i=7
     6 12 18 42 44 55 67 94
```

    - Realizacija v javi:

```
public static void bubbleSort(Element[] a)
{
    int i,j;
    Element x;

    for(i=1; i<a.length; ++i)
            for(j=a.length-1; j>=i; --j)
                    if(a[j].manjsi(a[j-1]))
                    {
                            x=a[j];
                            a[j]=a[j-1];
                            a[j-1]=x;
                    }
}
```

    - Analiza:

Število primerjanj:
neodvisno od zacetne permutacije podatkov v tabeli

- i-ti korak:

$C_i = n-1-(i-1) = n-i$

- v celoti:

$C = sum[i=1, n-1] (n-i) = 1/2 (n^2 - n)$

$T=o(n^2)$

Število premikov:

- Če je tabela že urejena: $M_{min}=0$
- Če je tabela nasprotno urejena, vsako primerjanje povzroči zamenjavo (vsaka zamenjava zahteva 3 pomike) $M_{max} = 3 * C = 3/2(n^2 - n)$
- V povprečju: polovica primerjav zahteva zamenjavo. $M=3/4(n^2 - n)$

$T=o(n^2)$

- **Možne izboljšave:**
  - postopek prekinemo, ce v nekem prehodu ni nobene zamenjave - bubblesort1
  - primerjanje dveh sosednjih elementov izvajamo samo do zadnje zamenjave - bubblesort2
  - menjavamo smeri prehodov - shakersort

- **Bubblesort 1**

Spremenljivka k tipa boolean pove, ali je v nekem prehodu prišlo do zamenjave

- na zacetku vsakega prehoda jo postavimo na false
- ob vsaki zamenjavi jo postavimo na true

Zunanjo zanko realiziramo s stavkom while (ne vemo kolikokrat se bo izvršila)

```
public static void bubbleSort1(Element[] a)
{
    int i,j;
    Element x;
    boolean k=true;
    i=1;

    while (i<a.length && k)
    {
           k=false;
           for(j=a.length-1; j>=i; --j)
                  if (a[j].manjsi(a[j-1]))
                  {
                         x=a[j];
                         a[j]=a[j-1];
                         a[j-1]=x;
                         k=true;
                  }
           ++i;
    }
}
```

- **Bubblesort 2**

Zapomnimo si mesto zadnje zamenjave. Naslednji prehod ustavimo na tem mestu (namesto da bi nadaljevali do fiksne določene spodnje meje).

Spremenljivka m tipa int vsebuje indeks tistega elementa v tabeli, pri katerem je prišlo do zadnje zamenjave:

1. na zacetku vsakega prehoda: m=a.length-1
2. ob vsaki zamenjavi postane m enaka j
3. na koncu vsakega prehoda dobi vrednost m

```
public static void bubbleSort2(Element[] a)
{
    int i,j,m;
    Element x;
    i=1;

    while (i<a.length)
    {
```

```
            m=a.length-1;
            for(j=a.length-1; j>=i; --j)
                    if(a[j].manjsi(a[j-1]))
                    {
                            x=a[j];
                            a[j]=a[j-1];
                            a[j-1]=x;
                            m=j;
                    }
            i=m+1;
    }
}
```

- **Shakesort:**

Menjavamo smer prehodov:

- pregledovanje od desne proti levi pripelje na pravo mesto najmanjši element
- pregledovanje od leve proti desni pripelje na pravo mesto najvecji element

Neurejeni del tabele na sredini se postopoma oži, postopek zakljucimo, ko se leva in desna meja srečata na sredini tabele.

```
public static void shakerSort(Element[] a)
{
    int j,m,l,r;
    Element x;
    l=1;
    r=a.length-1;
    m=r;

    do
    {
            for(j=r; j>=l; --j)
                    if(a[j].manjsi(a[j-1]))
                    {
                            x=a[j];
                            a[j]=a[j-1];
                            a[j-1]=x;
                            m=j;
                    }
            l=m+1;

            for(j=l; j<=r; ++j)
                    if(a[j].manjsi(a[j-1]))
                    {
                            x=a[j];
                            a[j]=a[j-1];
                            a[j-1]=x;
                            m=j;
                    }
            r=m-1;
    } while(l<r);
}
```

- Primer:

```
            44 55 12 42 94 18  6 67
l=1,r=7
             6 44 55 12 42 94 18 67
l=2,r=7
             6 44 12 42 55 18 67 94
l=2,r=6
             6 12 44 18 42 55 67 94
l=3,r=6
             6 12 18 42 44 55 67 94
l=3,r=3
```

Izboljšali smo število primerjanj.
Število pomikov ostane istega velikostnega reda.

Izboljsane metode sortiranja:

- Shellsort Wirth T=o($n^{1.2}$), Hubbard T=o($n^{1,5}$)
- Heapsort (sortiranje s kopico) T_w = o(n*log_2 n)
- Quicksort (sortiranje s porazdelitvijo) T_{avg} = o(n*$\log_2$ n), T_w = o(n^2)

- **ShellSort:**

Izboljšava navadnega izbiranja:

- uporabimo postopek navadnega vstavljanja z različnimi koraki
- koraki se zmanjšujejo; zadnji korak mora biti enak 1

  - Grob opis algoritma:

```
for(int m=0; m<T; ++m)
{
    določi korak k za to etapo;
    for(int i=k; i<a.length; ++i)
    {
            x:=a[i];
            vstavi x na pravo mesto, upoštevajoč korak
    }
}
```

  - Primer: 4 etape s koraki 9,5,3,1

```
    35 12 46  3 85 19 11 33 67 15 41 62
m=1, k=9
    15 12 46  3 85 19 11 33 67 35 41 62
m=2, k=5
    15 11 33  3 35 19 12 46 67 85 41 62
m=3, k=3
     3 11 19 12 35 33 15 41 62 85 46 67
m=4, k=1
     3 11 12 15 19 33 35 41 46 62 67 85
```

  - Realizacija v javi:

```
public static void shellSort (Element [] a)
{
    final int T=4;
    int i,j,k,m;
    int[] h={9,5,3,1};
    Element x;

    for(m=0; m<T; ++m)
    {
            k=h[m];
            for(i=k; i<a.length; ++i)
            {
                    x=a[i];
                    j=i-k;
                    while (j>=0 && x.manjsi(a[j]))
                    {
                            a[j+k]=a[j];
                            j-=k;
                    }
                    a[j+k]=x;
            }
    }
}
```

  - Analiza časovne zahtevnosti:

Wirth: T=o($n^{1.2}$)
Hubband: T=o($n^{1.5}$)

- Največ pomikov opravimo v prvi etapi (na začetku).
- Kasneje so verige večinoma urejene oz. je potrebno le manjše število pomikov.

Izbira korakov:

- pomembno za ustrezno obnašanje algoritma
- koraki naj bodo izbrani tako, da omogočajo čim boljše prepletanje verig
- primer slabe izbire korakov: 16, 8, 4, 2, 1

Priporočene formule za izbiro korakov:

koraki: $h_1,h_2,...,h_T$ $h_T = 1$; $h_m+1 < h_m$

1. $h_t=1$; $h_{m-1} = 3*h_m +1$
2. $h_t=1$; $h_{m-1} = 2*h_m +1$

```
public static void shellSort1(Element [] a)
{
    final int T=4;
    int i,j,k=1;
    Element x;

    // izračun korak za prvo etapo
    while (9*k < a.length)
           k=3*k+1;

    // zanka za etape
    while (k>0)
    {
           for(i=k; i<a.length; ++i)
           {
                  x=a[i];
                  j=i-k;
                  while (j>=0 && x.manjsi(a[j]))
                  {
                         a[j+k]=a[j];
                         j-=k;
                  }
                  a[j+k]=x;
           }
           k/=3;   // k=k/3
    }
}
```

- **HeapSort (sortiranje s kopico):**

  - Lastnosti kopice:

1. ponazoritev v obliki binarnega drevesa
2. vsako vozlišče ustreza enemu izmed elementov, ki jih urejamo
3. za vsako vozlišče velja, da je element v tem vozlišču večji ali enak od vseh elementov v pripadajočem popdrevesu
4. dolžini dveh pripadajočih vej se lahko razlikujeta kvečjemu za 1; daljše veje so skrajno levo

  - Ponazoritev kopice v računalniku:

- elementi kopice so shranjeni v tabeli: Element[] a
- koren kopice se nahaja v a[0]
- levi sin vozlišča a[i] je shranjen v a[2*i+1]
- desni sin vozlišča a[i] je shranjen v a[2*i+2]

Primer:

indeks	0	1	2	3	4	5	6	7	8	9
vsebina	75	34	62	28	18	45	61	21	25	7

- Opis algoritma

4 koraki

1. vhodno zaporedje uredimo v kopico
2. zamenjamo prvi in zadnji element v kopici
3. popravimo kopico (upoštevamo, da zadnji element ni več v kopici)
4. ponavljamo koraka 2. in 3., dokler v kopici ne zmanjka podatkov

- Popravljanje kopice (pogrezanje)

Postopek pogrezanja uporabimo tako v 1. koraku kot v 3. koraku.
Kopico, ki ima neustrezen koren, popravimo tako, da koren pogrezamo po tisti veji, v kateri se nahaja večji naslednjik. Ta postopek ponavljamo, dokler ne pridemo do lista ali dokler ne dosežemo lastnosti kopice.

- Metoda za popravljanje kopice:

predpostavka:

- kopico tvorijo elementi z indeksi od l do r
- koren se nahaja na poziciji l
- r je desna meja kopice

```
public static void heapify(Element[] a, int l, int r)
{
    int i=l, j=2*i+1;
    Element x=a[i];

    while (j<=r) //dokler je naslednik znotraj kopice
    {
            if ((j<r) && (a[j].manjsi(a[j+1]))) ++j;            // a[j] je večji naslednik
            if (! x.manjsi(a[j])) break;
            a[i]=a[j];
            i=j;
            j=2*i+1;
    }
    a[i]=x;
}
```

- Gradnja kopice:

Uporabimo postopek pogrezanja.

- Elementi v desni polovici tabele nimajo naslednikov, zato pogrezanje ni potrebno
- Preostale elemente pogrezamo od sredine proti zacetku tabele. Zanka:

```
For(int l=a.length/2-1; l>=0; l--)
    Heapify(a,l,a.length-1);
```

- Gradnji kopice sledi zanka, v kateri izvajamo koraka 2 in 3:

```
For(int r=a.length-1; r>=1; r--)
{
    //zamenjava prvega in zadnjega elementa:
    X=a[0];
    a[0]=a[r];
    a[r]=x;

    heapify(a,0,r-1); //pogreznemo koren
}
```

- Koncna resitev: zdruzimo vse dosedanje korake:

```
Public static void heapSort(Element[] a)
{
    Element x;
```

```
    //uredimo tabelo v kopico (1. korak)
    For(int l=a.length/2-1; l>=0; l--)
            Heapify(a,l,a.length-1);

    //koraka 2 in 3:
    For(int r=a.length-1; r>0; r--)
    {
            X=a[0];
            a[0]=a[r];
            a[r]=x;
            heapify(a,0,r-1);
    }
}
```

- Analiza:

Premiki:

M_w – najslabsa mozna vrednost (worst)

$M_w < n/2 * \log_2 n + (n-1) \log_2 n + 3 (n-1)$

$M_w = (3n/2 - 1) \log_2 n + 3 (n-1) \Rightarrow T_w = o(n*\log_2 n)$

- **QuickSort:**

Osnova za algoritem quicksort je postopek porazdelitve, v katerem razdelimo tabelo na 2 dela:

- Izberemo nek poljuben element tabele (npr. srednji element), recimo da bo to element x
- Pregledujemo tabelo z leve dokler ne naletimo na element a[i], ki je vecji ali enak x
- Pregledujemo tabelo z desne dokler ne naletimo na element a[j], ki je manjsi ali enak x
- Zamenjamo a[i] in a[j]
- Ponavljamo korake 2, 3 in 4 dokler se obe pregledovanji ne srecata nekje na sredini tabele

- Primer:

```
i        x        j<-j
44 55 12 42 94  6 18 67
   i     x      j
18 55 12 42 94  6 44 67
      i->ixj<-j
18  6 12 42 94 55 44 67
       j  x  i
18  6 12 42 94 55 44 67
```

Levi del tabele:

a[k]<x, k=0, 1, ..., j

desni del:

a[k]>x, k=i, i+1, ..., a.length-1

postopek porazdeljevanja rekurzivno nadaljujemo na levem in desnem delu tabele toliko casa, dokler v nekem delu tabele ne ostane samo en element

- Realizacija v javi:

```
public static void quickSort(Element[] a)
{
    sort(a,0,a.length-1);
}

private static void sort(Element[] a, int l, int r)
{
    int i=l, j=r;
    Element x=a[(l+r)/2], w;

    do
    {
```

```
            while(a[i].manjsi(x))
                    i++;

            while(x.manjsi(a[j]))
                    j--;
            if(i<=j)
            {
                    w=a[i];
                    a[i]=a[j];
                    a[j]=w;
                    i++;
                    j--;
            }
    } while(i<=j);

    if(l<j)
            sort(a,l,j);
    if(i<r)
            sort(a,i,r);
}
```

- Analiza:

St. primerjanj:
vsaka porazdelitev zahteva n primerjanj.
ce x razdeli tabelo na 2 dela, potem je potrebno $\log_2 n$ porazdelitev (nivojev):
$C=n*\log_2 n \Rightarrow T=o(n*\log_2 n)$

ce imamo smolo in x izberem tako, da je najvecji ali pa najmanjsi element:
$C=2+3+4+...+(n-1)+n= (n (n+1))/2 – 1$
$T=o(n^2)$

- **LeksiSort (leksikografsko sortiranje):**

Postopek sortiranja temelji na razvrscanju podatkov v razrede glede na doloceno mesto v kljucu. »Kljuc« razsekamo na posamezna mesta in postopek ponavljamo v zanki po vseh mestih od najmanj pomembnega do najbolj pomembnega.
V vsaki iteraciji podatke razporedimo po razredih in jih (delno urejene) spet prepisemo v tabelo a.
Uporabno je za sortiranje stevil in nizov.

- Primer: sortiranje 3 mestnih celih stevil:

stevilo mest: 3
stevilo razredov: 10

Najprej razporejamo stevila po enicah, nato po deseticah, nazadnje po stoticah.

- zacetno stanje v tabeli a:

257, 915, 655, 315, 411, 65, 721, 852, 101, 477, 67, 222

- razporejanje po enicah:

razred 0:
razred 1: 411, 721, 101
razred 2: 852, 222
razred 3:
razred 4:

razred 5: 915, 655, 315, 65
razred 6:
razred 7: 257, 477, 67
razred 8:
razred 9:

- tabela a po prvi iteraciji:

411, 721, 101, 852, 222, 915, 655, 315, 65, 257, 477, 67

- razporejanje po deseticah:

razred 0: 101
razred 1: 411, 915, 315
razred 2: 721, 222
razred 3:
razred 4:
razred 5: 852, 655, 257
razred 6: 65, 67
razred 7: 477
razred 8:
razred 9:

- tabela a po drugi iteraciji:

101, 411, 915, 315, 721, 222, 852, 655, 257, 65, 67, 477

- razporejanje po stoticah:

razred 0: 65, 67
razred 1: 101
razred 2: 222, 257
razred 3: 315
razred 4: 411
razred 5:
razred 6: 655
razred 7: 721
razred 8: 852
razred 9: 915

- tabela po tretji iteraciji:

65, 67, 101, 222, 257, 315, 411, 655, 721, 852, 915

- realizacija v javi:

```
public class LeksiSort
{
    public static void sortiraj(int[] a, int raz, int mest) //raz-st. razredov, mest-st.
mest
    {
            for(int m=0; m<mest; m++)
                    razvrsti(a,raz,mest,m); //razvrsti podatke v razrede
    }

    private static void razvrsti(int[] a, int raz, int mest, int m)
    {
```

```
            int[] c=new int[raz];

            for(int i=0; i<a.length; i++)
                    //doloci stevilo elementov v vsakem razredu
                    ++c[dolociRazred(m,a[i],raz)];
            for(int i=1; i<raz; i++)
                    //c[i] vsebuje st. elementov, ki so <= i
                    c[i]+=c[i+1];

            int[] tmp=new int[a.length];
            for(int i=a.length-1; i>=0; i--)
                    tmp[--c[dolociRazred(m,a[i],raz)]]=a[i];
            for(int i=0; i<a.length; i++)
                    a[i]=tmp[i];
    }

    private static int dolociRazred(int m, int stevilo, int raz)
    {
            return stevilo/(int)Math.pow(10,m)%raz;
    }
}
```

Spremembe za sortiranje besed:

1. seznam parametrov v metodi sortiraj
2. zanka v metodi sortiraj
3. seznam parametrov v razvrsti
4. tabela tmp
5. DolociRazred - parametri
6. izvedba dolociRazred

```
public static void sortiraj(String[] a, int raz, int mest) // rez-št. razredov, mest-št.
mest

{
    for(int m=mest-1; m>=0; m--)
    {
            // razvrsti podatke v razrede glede na mesto m
            razvrsti(a, raz, mest, m);
    }
}

private static void razvrsti(String[] a, int raz, int mest, int m)
{
    int c[] = new int[raz];
    for(int i=0; i<a.length; i++)
    {
            // določi število elementov v vsakem razredu
            ++c[dolociRazred(m, a[i], raz)];
    }
    for (int i=1; i<raz; i++)
    {
            c[i]+=c[i-1];
    }
    String tmp[] = new String[a.length];
    for (int i=a.length-1; i>=0; i--)
    {
            tmp[--c[dolociRazred(m, a[i], raz)]] = a[i];
    }
    for (int i=0; i<a.length; i++)
            a[i]=tmp[i];
}

private static int dolociRazred(int m, String niz, int raz)
{
    if (m>niz.length()-1 || niz.charAt(m)=' ')
            return 0;
    else
            return Character.toUpperCase(niz.charAt(m))-'A'+1;
}
```

3. Zbirke (collections)

Mnozica objektov tipa Object ali kateregakoli iz Object izpeljanega tipa.

Primer: sklad (LIFO), seznam, vrsta (FIFO), ...

V javi obstaja razred Collection, ki definira seznam operacij nad zbirko; vsak razred, ki implementira neko zbirko mora zagotavljati te operacije.

- **Razred Vector:**

Definira zbirko (ang. Collection) elementov tipa Object, ki se obnasa podobno kot tabela.

Ta razred je definiran v paketu java.util:
import java.util.Vector; ALI
import java.util.*;

Razlike v primerjavi s tabelo:
- velikost vektorja se po potrebi avtomatsko povecuje
- vektor lahko hrani objekte razlicnih tipov (ker so vsi izpeljani iz razreda Object)

Podobno kot pri tabelah in objektih:
- velja da spremenljivka tipa Vector vsebuje naslov lokacije v pomnilniku, kjer se dejansko nahajajo elementi vektorja

Kreiranje vektorjev:
4 konstruktorji:
- konstruktor brez argumentov kreira vektor standardne dolzine 10 elementov. Vector v=new Vector();
- kot argument lahko navedemo zahtevano kapaciteto. Vector v=new Vector(100) //vektor s kapaciteto 100
- poleg kapacitete lahko navedemo se prirastek ob vsakem povecanju velikosti. Vector v=new Vector(100,10) //ob vsakem povecanju se doda prostor se za 10 objektov
- kreiramo vektor, ki vsebuje objekte iz neke druge ze obstojece zbirke. Vector v=new Vector(c) //c je zbirka objektov tipa Collection ali katerega iz Collection izpeljanega tipa

Kapaciteta in velikost vektorja:
capacity: st. objektov, ki jih lahko spravimo v vektor
size: st. dejansko shranjenih objektov

V razredu Vector obstajata dve metodi:
- capacity() – vrne kapaciteto
  int kapaciteta=v.capacity();
- size() – vrne velikost
  int velikost=v.size();

Razpolozljiv prostor v vektorju:
int prostor=v.capacity() – v.size();

Ostale metode povezane s kapaciteto in velikostjo:

- ensureCapacity(150) //ce je kap. manjsa, jo postavi na 150, ce je vecja, ostane nespremenjena
- setSize(50) //ce je vel. manjsa od 50, se v elemente do vkljucno 50 vpise vrednost null, ce je vecja od 50, potem se elementi nad 50 izgubijo
- trimToSize() //nastavi kapaciteto na dejansko velikost vektorja

Shranjevanje objektov v vektor:

- add(novObjekt) //na prvo prosto mesto v vektorju se vpise referenca (naslov) na ta objekt; velikost vektorja se poveca za eno, ce je dodajanje uspesno vrne true, sicer false
- add(2, novObjekt) //ta klic doda novObjekt na pozicijo 2; elementi na pozicijah od vkljucno 2 dalje se pomaknejo za eno mesto naprej. Pozicije so ostevilcene od 0 naprej. Prvi argument ne sme biti vecji od velikosti vektorja.
- addElement(novObjekt) //deluje enako kot add(novObjekt), le da ne vrne rezultata
- addAll(c) //c je tipa Collection in predstavlja neko ze obstojeco zbirko; metoda deluje tako, da na konec vektorja doda vse elemente iz zbirke c
- addAll(2, c) //na pozicije od vkljucno 2 naprej vrine elemente zbirke c

Branje podatkov iz vektorja:

Za prikaz teh metod bomo predpostavili da vektor hrani objekte, ki pripadajo razredu Delavec.

- Delavec d=(Delavec) v.get(4); //vrne element na poziciji 4; potrebna je eksplicitna konverzija tipa, ker get vraca tip Object
- Delavec d=(Delavec) v.elementAt(4); //isto kot zgoraj
- Delavec d=(Delavec) v.firstElement(); //vrne prvi element
- Delavec d=(Delavec) v.lastElement(); //vrne zadnji element

Odstranjevanje elementov iz vektorja:

Pri odstranjevanju elementov iz vektorja se kapaciteta ne zmanjsuje, zmanjsuje se samo velikost.

- remove(3) //odstrani element na poziciji 3, preostali elementi se pomaknejo za eno mesto proti zacetku
- removeElementAt(3) //isto kot zgoraj
    - razlika med tema dvema: metoda remove vrne kazalec (naslov) odstranjenega elementa
      Delavec d=(Delavec) v.remove(3); //ce hocemo se kaj delat z odstranjenim elementom, ga mamo zdej shranjenega v novi spremenljivki d
- boolean brisan=v.remove(d); //odstrani prvo nastopanje elementa d iz vektorja v. Ce elementa, ki hocemo brisat (d) v vektorju ni, potem vrne false, drugace ga izbrise in vrne true
- clear() //odstrani vse elemente
- removeAll(c) //odstrani vse elemente, ki se nahajajo v zbirki c (vsa nastopanja)

Iskanje elementov v vektorju:

- int poz=v.indexOf(d); //vrne pozicijo objekta d v vektorju v
- int poz=v.indexOf(d,5); //isce od pozicije 5 naprej (vkljucno s 5)
- int poz=v.lastIndexOf(d); //isce od konca vektorja proti zacetku

- int poz=v.lastIndexOf(d,5); //isce od pozicije 5 proti zacetku

Prepis elementov iz vektorja v tabelo:

- toArray -> poglej v knjige za razlago

Obdelava vseh elementov:

```
Delavec d;

for(int poz=0; poz<v.size(); poz++)
{
    d=(Delavec)v.get(poz);
    //se ostali stavki, za obdelavo tega objekta potem
}
```

Obhod s pomocjo iteratorja:

V vseh razredih, ki implementirajo vmesnik Collection obstaja objekt imenovan Iterator, ki omogoca sprehajanje po elementih zbirke.

Iterator ima 3 metode:

- next() //vrne naslednji objekt (zacne s prvim objektom in vsakokrat pripravi vse potrebno za naslednjega)
- hasNext() //vrne true, ce obstaja naslednji element, false ce naslednjega elementa ni
- remove() //odstrani element, ki smo ga dobili ob zadnjem klicu next()

```
Delavec d;
Iterator it=v.iterator();

while(it.hasNext())
{
    d=(Delavec)it.next();
    //se naprej stavki za obdelavo tega elementa
}
```

- Primer: program DemoVector

```
import java.util.*;

public class DemoVector
{
    public static void main(String[] args)
    {
            Vector osebe = new Vector(10);
            String ime;
            //polnjenje vektorja
            for ( ; ; )
            {
                    System.out.print("Vpisi ime: ");
                    ime = BranjePodatkov.preberiString();
                    if (ime.length() == 0) break;
                    osebe.add(ime);
                    System.out.println("Kapaciteta: "+osebe.capacity() +"  Velikost: "+
osebe.size());
            }

            // izpis vektorja
            for (int poz=0; poz < oseba.size(); poz++)
                    System.out.println(oseba.get(poz));

            // izpis vektorja s pomocjo iteratorja
            Iterator it = osebe.iterator();

            // ne izpise prvega vektorja
            while (it.hasNext())
                    System.out.println((String) it.next());

            // iskanje
            for ( ; ; )
```

```
            {
                    System.out.print("Vpisi ime: ");
                    ime = BranjePodatkov.preberiString();
                    if (ime.length() == 0) break;
                    int poz = osebe.indexOf(ime);
                    if (poz == -1) System.out.println(ime+ " ni v vektorju");
                    else System.out.println(ime+ " se nahaja na poziciji "+poz);
            }

            // odstranitev vseh razen prvih treh
            int v = osebe.size();
            for (int poz = v-1; poz > 2; poz--)
            {
                    osebe.remove(poz);
                    System.out.println("Kapaciteta: "+osebe.capacity() +"  Velikost: "+
osebe.size());
            }

            // odstranitev vseh na enkrat
            osebe.clear();
            System.out.println("Kapaciteta: "+osebe.capacity() +"  Velikost: "+
osebe.size());
    }
}
```

4. Sklad (stack)

Podatkovna struktura, ki deluje na principu last-in-first-out (LIFO).
Za predstavitev sklada razred Stack, ki je deklariran kot razširitev razreda Vector in se nahaja v paketu java.util.

Operacije za delo s skladom:

empty() - funkcija, ki vrne true, če je sklrd prazen
peek() - funkcija, ki vrne objekt, ki je na vrhu sklada (sklad se ne spremeni)
pop() - funkcija, ki s sklada vzame objekt na vrhu
posh(objekt) - doda objekt na vrh sklada
search(objekt) - funkcija, ki vrne zaporedno številko objekta (Podobno kot indexOf vendar je prvi element označen z 1 in ta element je na vrhu sklada.)

Deklaracija Stacka - kot v javi

```
public class Stack extends Vector {
    public boolean empty() {
            return (size() == 0);
    }

    public Object peek() {
            if (size() == 0) throw new EmptyStackException();
            return elementAt(size()-1);
    }

    public Object pop() {
            Object object = peek();
            removeElementAt(size()-1);
            return object;
    }

    public Object push (Object object) {
            addElement(object);
            return object;
    }

    public int search(Object object) {
            int i = lastIndexOf(object);
            if (i < 0) return -1;         // objekt ni v skladu
            return size() - i;
    }

    public Stack() {
```

```
    }
}
```

Uporaba sklada:

- računanje aritmetičnih izrazov v postfiksni obliki
- odprava rekurzije

Primer 1: računanje v postfiksni obliki (Reverse Polish Notatign - RPN)
Najprej zapišemo oba operanda, nato operator.

3 + 4 = 7 infiksna oblika
3 4 +
Najprej postavimo na sklad vrednost 3.
Nato postavimo na sklad vrednost 4.
Obe vrednosti se odvzamemo s sklada, in seštejeta, rezultat se zopet postavi na sklad.

infiksni izraz: 3*(4+5)/10 - 1
v postfiksni obliki: 3 4 5 + * 10 / 1 -

Operand na vrhu sklada se upošteva kot drugi operand, operand pod njim pa se upošteva kot prvi operand.

```
import java.util.Stack;
public class kalkulator {
    public static void main (String[] args) {
            boolean konec = false;
            String input;
            double x, y, z;
            Stack sklad = new Stack();
            while (! konec) {
                    System.out.print("Vnesi operand ali operator: ");
                    input = BranjePodatkov.preberiString();
                    switch (inpot.charAt(0)) {
                            case 'k' :
                                    System.out.println("Rezultat je: " + sklad.peek());
                                    konec = true;
                                    break;
                            case '+' :
                                    y = Double.parseDouble((String)sklad.pop());
                                    x = Double.parseDouble((String)sklad.pop());
                                    z=x+y;
                                    sklad.push(new Double(z).toString());
                                    break;
                            case '-' : // v podobnem načinu naprej po /
                            default: sklad.push(input)
                    }
            }
    }
}
```

Primer 2: Odprava rekurzije s pomočjo sklada

Hanojski stolpiči:

Problem je po svoji naravi rekurziven; rešitev je sestmvljena iz 3 delov:

- najprej prestavimo n-1 obročev s palice A na pomožno palico B
- največji obroč prestavimo z A na C
- n-1 obročev prestavimo s palice B na C

```
public class HanoiRek {
    public static void main (String[] args) {
            hanoi(3, 'A', 'B', 'C');
    }

    private static void hanoi (int n, char a, char b, char c) {
            if (n==1)
                    System.out.println("Prestavi obroc s palice "+a+" na palico "+c);
            else {
```

```
                hanoi(n-1, a, c, b);
                hanoi(1, a, b, c);
                hanoi(n-1, b, a, c);
        }
    }
}
```

Iterativna rešitev s pomočjo sklada:
Posamezne zahtevke za rekurzijo odlagamo na sklad.

- najprej postavimo na sklad začetni zahtevek
- nato v zanki vsakokrat odvzamemo z vrha sklada en zahtevek in ga obdelamo (med obdelavo se lahko na sklad postavljajo novi zahtevki)
- postopek ponavljamo dokler sklad ni prazen.

Predstavitev posameznih zahtevkov ponazorimo z razredom.

```
import java.util.Stack;
class Zahtevek {
    public int n;
    public char a, b, c;

    public Zahtevek(int n, char a, char b, char c) {
            this.a=a;
            this.b=b;
            this.c=c;
            this.n=n;
    }
}
public class HanoiIte {
    public static void main (String[] args) {
            hanoi(3, 'A', 'B', 'C');
    }

    private static void hanoi (int n, char a, char b, char c) {
            stack s = new Stack();
            s.push(new Zahtevek(n, a, b, c));
            while (!s.empty()) {
                    Zahtevek z = (Zahtevek)s.pop();
                    n = z.n;
                    a = z.a;
                    b = z.b;
                    c = z.c;
                    if(n==1)
                            System.out.println("Prestavi obroc s palice "+a+" na palico "+c);
                    else {
                            s.push(new Zahtevek(n-1, b, a, c);
                            s.push(new Zahtevek(1, a, b, c);
                            s.push(new Zahtevek(n-1, a, c, b);
                    }
            }
    }
}
```

5. Vmesnik (Interface)

Nadomestilo za večkratno dedovanje.

Večkratno dedovanje:

- podrazred podeduje atribute in metode več kot enega nadrazreda
- realizacija večkratnega dedovanja je kompliciranа, java ne omogoča večkratnega dedovanja, ampak namesto tega ponuja koncept vmesnika

Vmesnik je podoben razredu s to razliko, da:

- vse metode v vmesniku morajo biti abstraktne
- vsi atributi (če jih ima) morajo biti static final

Z deklaracijo vmesnika predpišemo metode, ki jih mora implementirati nek podrazred (predpišemo obnašanje podrazreda).

Nek podrazred lahko deduje samo od enega nadrazreda, implementira pa lahko več različnih vmesnikov.

```
public class Podrazred extends Nadrazred implements Vmesnik1, Vmesnik2
```

S stališča dedovanja Podrazred

- podeduje atribute in metode razreda Nadrazred
- dodatno deklarira nove atribute in metode
- če mu katera od podedovanih metod ne ustreza jo lahko redefinira

S stališča implementacije vmesnik

- v podrazredu moramo deklarirati vse metode, ki so specificirane v vmesnikih Vmesnik1, Vmesnik2
- podrazred lahko uporblja atribute (statične spremenljivke), ki so definirani v vmesnikih

Primerjava: abstraktni razred - vmesnik

- v abstraktnih razredih imamo poleg abstraktnih metod todi sprogramirane metode, ki jih lahko dedujejo podrazredi, vmesnik ima samo abstraktne metode.
- atribute abstraktnega razreda se lahko obnašajo kot spremenljivke objektov (niso statični) in lahko spreminjamo vrednost, v vmesniku so atributi konstante.

Za predstavitev različnih zbirk objektov (collections) obstaja v javi več vnaprej predpisanih vmesnikov:

- Collection — osnovni vmesnik za neko splošno zbirko objektov
- List — zaporedje elementov
- Set — zbirka elementov brez duplikatov
- SortedSet — urejena zbirka elementov brez duplikatov
- Map — zbirka parov (key, value); ključi morajo biti enolični
- SortedMap — urejena zbirka parov (key, value); ključi morajo biti enolični
- Iterator — objekt, s katerim se sprehajamo po zbirki
- ListIterator — objekt, s katerim se sprehajamo po zaporedno urejeni zbirki

Obstajajo razredi, ki implementirajo te vmesnike:

AbstractCollection
AbstractList
AbstractSet
TreeSet
AbstractMap
TreeMap

Object
AbstractCollection
AbstractList
Vector
Stack

- **Vmesnik Collection**

definiran v paketu java.util

zahteva implementacijo naslednjih metod:

```
public boolean add(Object object);
public boolean addAll(Collection collection);
public void clean();
public boolean contains(Object object);
```

public boolean containsAll(Collection collection);
public boolean equals(Object object);
public int hashCode();
public boolean isEmpty();
public Iterator iterator();
public boolean remove(Object object);
public boolean removeAll(Collection collection);
public boolean retainAll(Collection collection);
public int size();
public Object[] toArray();
public Object[] toArray(Object[] objects);

Razred AbstractCollection

- delna implementacija vmesnika Collection
- implementira vse, kar se da implementirati, ne da bi poznali pomnilniško strukturo za predstavitev zbirke
- manjka implementacija metod equals in hashCode
- metodi iterator in size sta še vedno abstraktni
- zanašamo se da te metode deklarirajo podrazredi
- nedefinirana je metoda toString()

```
public String toString() {
    if (isEmpty()) return "[]";
    Iterator it = iterator();
    String str = "["+it.next();
    while (it.hasNext())
           str = ", "+ it.next();
    return str+"]";
}
```

Primer deklaracije isEmpty()

```
public boolean isEmpty() {
    return size() == 0;
}
```

Bag ali multiset: poljubna zbirka objektov, ki lahko vsebuje duplikate. Realizirali jo bomo kot razširitev razreda AbstractCollection
Za predstavitev objektov bomo uporabili tabelo.

```
import java.util.*;
public class Bag extends AbstractCollection {
    private Object[] objects;
    private int size = 0; // stevilo objektov v zbirki
    private static final int CAPACITY = 16;

    private void resize() {
           Object[] temp=objects;
           objects = new Object[2*temp.length];
           for(int i=0; i<size; i++)
                  objects[i]=temp[i];
    }

    public Bag() {
           objects = new Object[CAPACITY];
    }

    public Bag(Object[] objects) {
           this.objects = new Object[2*objects.length];
           for(int i=0; i<objects.length; i++)
                  this.objects[size++] = objects[i];
    }

    public boolean add(Object object) {
           if (size == objects.length)
                  resize();
           objects[size++] = object;
           return true;
```

```
    }

    public boolean addAll(Collection collection) {
            while (size + collection.size() > objects.length)
                    resize();
            for (Iteration it=collection.iterator(); it.hasNext(); )
                    objects[size++]=it.next;
            return true;
    }

    public void clean() {
            for(int i=0; i<size; i++)
                    Objects[i] = null;
            size=0;
    }

    public boolean contains(Object object) {
            for(int i=0; i<size; i++)
                    if(object.equals(objects[i]))
                            return true;
            return false;
    }

    public boolean containsAll (Collections collection) {
            for (Iterator it=collection.iterator(); it.hasNext(); )
                    if (!this.contains(it.next())) return false;
    }

    public boolean isEmpty() {
            return size==0;
    }
/*
Realizacija iteratorja za razred Bag
potrebujemo metodo public Iterator iterator();
vmesnik (interface), ki predpisuje 3 metode: hasNext, next, remove
najprej je potrebno deklarirati razred, ki implementira vmesnik Iterator
na podlagi tega razreda šele lahko generiramo objekt tipa Iterator.
*/
    private class BagIterator implements Iterator {
            private int cursor=0;

            public boolean hasNext() {
                    return cursor<size;
            }

            public Object next() {
                    if (corsor >= size) return null;
                    return objects[cursor++];
            }

            public void remove() {
                    objects[--cursor] = objects[--size];
                    objects[size]=null;
            }
    }

    public Iterator iterator() {
            return new BagIterator();
    }

/*
Obstaja še možnost krajšega zapisa v obliki ti. anonimnega notranjega razreda (anonymous
inner class)
*/

    public Iterator iterator() {
            return new Iterator() {
                    public boolean hasNext() {
                            return cursor<size;
                    }

            public Object next() {
                    if (corsor >= size) return null;
                    return objects[cursor++];
            }

            public void remove() {
                    objects[--cursor] = objects[--size];
                    objects[size]=null;
```

```
                }
        };
    }

    public boolean remove(Object object) {
        for(int i=0; i<size; i++)
            if (object.equals(objects[i])) {
                objects[i] = object[--size];
                return true;
            }
        return false;
    }

    public boolean removeAll(Collection collection) {
        boolean modified = false;
        for(Iterator it = collection.iterator(); it.hasNext(); )
            if (this.remove(it.next())) modified=true;
        return modified;
    }

    publc boolean retainAll(Collection collection) {
        // v torbi obdrži samo tiste elemente, ki so v zbirki collection
        boolean modified = false;
        for (int i=0; i<size; i++)
            if (!collection.contains(objects[i])) {
                remove(object[i]);
                modified = true;
            }
        return modified;
    }

    public int size() {
        return size;
    }

    public Object[] toArray() {
        // prepiše objekte iz torbe v tabelo objektov
        Object[] objects = new Object[size];
        for (int i=0; i<size; i++)
            objects[i] = this.objects[i];
        return objects;
    }
```

6. List (seznam)

```
public interface List extends Collection
{
    A€ public boolean add(Object objekt);
    S,A€ public void add (int index, Object objekt);
    S,A public boolean addAll(Collection collection);
    S,A public boolean addAll(int index, Collection collection);
    A public void clear();
    public boolean contains(Object objekt);
    public boolean containsAll(Collection collection);
    A public boolean equals(Object objekt);
    S,A€ public Object get(int index);
    A public int hashCode();
    A public int indexOf(Object objekt);
    public boolean isEmpty();
    S,A public Iterator iterator();
    S,A public int lastIndexOf(Object objekt);
    S,A public ListIterator listIterator();
    S,A public ListIterator listIterator(int index);
    A public boolean remove(Object objekt);
    S,A public Object remove(int index);
    public boolean removeAll(Collection collection);
    public boolean retainAll(Collection collection);
    S,A€ public Object set(int index, Object objekt);
    public int size();
    A public List subList(int start, int stop);
    public Object[] toArray();
    public Object[] toArray(Object[] objekt);
}
```

Object

- AbstractCollection ..... Collection
  - AbstractList ..... List
    - AbstractSequentialList
      - LinkedList
    - ArrayList
    - Vector

Iterator

- ListIterator

			Get(), set()	Add(), remove()
AbstractList	ArrayList	Tabela objektov	0(1)	0(n)
AbstractSequentialList	LinkedList	Povezan seznam objektov	0(n)	0(1)

A -> AbstractList

- € -> ne more podedovati

S -> AbstractSequentialList (realizirano)

```
public interface ListIterator extends Iterator
{
    public boolean hasNext();
    public Object next();
    public void remove();
    public boolean hasPrevious();
    public Object previous();
    public int nextIndex();
    public int previousIndex();
    public void add(Object object);
    public void set(Object object);
}
```

7. Vrsta (FIFO first in first out)

Hierarhija Collection Framework ne vsebuje posebnega vmesnika ali razreda za vrsto.
Kot izhodisce izberemo vmesnik Collection.

Collection

- Queue: vmesnik za vrsto, ki dodatno specificira 4 metode, ki so specificne za vrsto

```
import java.util.*;

public interface Queue extends Collection
{
    public Object dequeue(); //odvzemanje elementa na zacetku
    public Object enqueue(Object object); //doda element na koncu
    public Object getBack(); //dobimo vrednost zadnjega elementa vrste, vsebina se ne
spremeni
```

```
    public Object getFront(); //dobimo vrednost prvega elementa vrste, -||-
}
```

Primer delovanja vrste:

Recimo da je q objekt tipa Queue.

```
                                vrsta:
q.enqueue('Novak');             Novak
q.enqueue('Kranjc');            Novak, Kranjc
q.enqueue('Petelin');           Novak, Kranjc, Petelin
q.enqueue('Vidmar');            Novak, Kranjc, Petelin, Vidmar
q.dequeue();                    Kranjc, Petelin, Vidmar
q.dequeue();                    Petelin, Vidmar
q.enqueue('Kajzer');            Petelin, Vidmar, Kajzer
```

Na koncu metoda getFront vrne 'Petelin', metoda getBack pa 'Kajzer'.

Vmesniki:	Implementacije teh vmesnikov:
Collection • Queue	Object • AbstractCollection o AbstractQueue ▪ ArrayQueue ▪ LinkedQueue

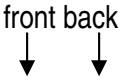
Najprej sprogramiramo razred AbstractQueue:

- je razsiritev razreda AbstractCollection
- implementira vmesnik Queue

```
import java.util.*;

public abstract AbstractQueue extends AbstractCollection implements Queue
{
    protected AbstractQueue()
    {
    }

    public abstract Object dequeue();
    public abstract Object enqueue(Object object);
    public boolean equals(Object object);
    {
            if(object==this) return true; //gre za isti objekt
            if(!(object instanceOf AbstractQueue)) return false; //objekta nista istega
tipa
            AbstractQueue aq=(AbstractQueue) object; //pretvorimo object v vrsto
            if(aq.size()!=this.size()) return false; //vrsti se ujemata v st. elementov
            return containsAll(aq); //vrsti sta enaki, ce vsebujeta iste elemente
    }
    public abstract Object getBack();
    public abstract Object getFront();
    public int hashCode()
    {
            int n=0;
            for(Iterator it=iterator(); it.hasNext(); )
            {
                    Object object=it.next();
                    if(object!=null)
                            n+=object.hashCode();
            }
            return n;
    }
    public abstract Iterator iterator();
    public abstract int size();
}
```

tabela z objekti:

front back

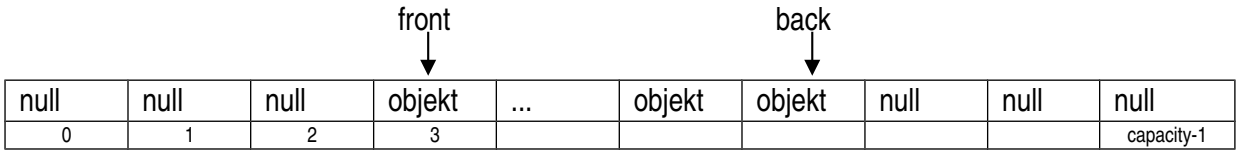


front in back sta indexa

null	null	null	null	...	null	null	null	null	null
0	1	2	3						capacity-1

capacity: st. elementov v tabeli
Zgoraj je prikazana prazna vrsta

Sredi delovanja vrste:



null	null	null	objekt	...	objekt	objekt	null	null	null
0	1	2	3						capacity-1

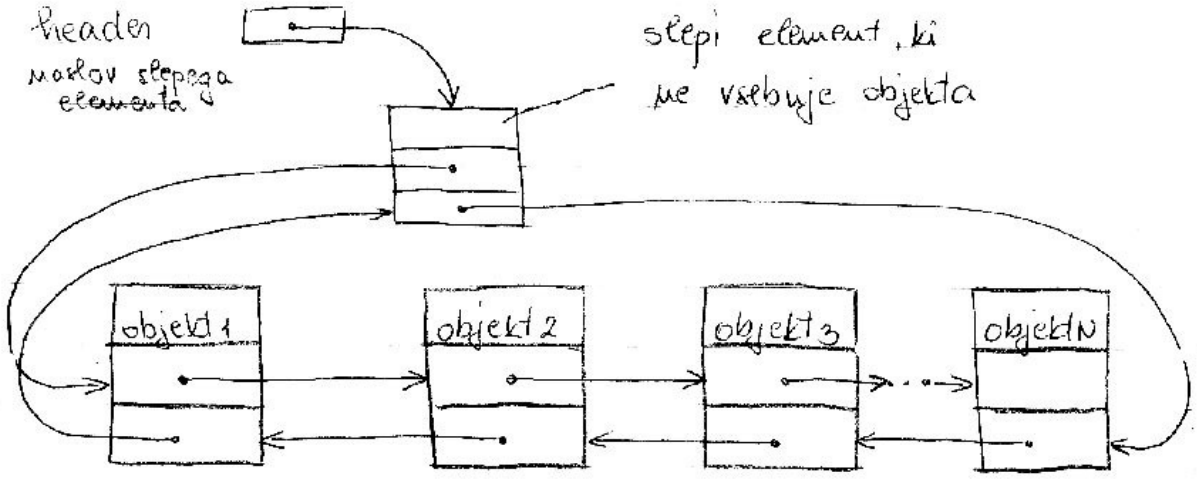
Pravila (in variante):

- objects[i]==null za 0<=i<front
- objects[i]!=null za front<=i<back
- objects[i]==null za back<=i<capacity

```
import java.util.*;

public class ArrayQueue extends AbstractQueue
{
    protected Object[] objects;
    protected int front=0;
    protected int back=0;
    protected int capacity=16;

    public ArrayQueue()
    {
            objects=new Object[capacity];
    }

    public ArrayQueue(int capacity)
    {
            this.capacity=capacity;
            objects=new Object[capacity];
    }
    .
    .
    .
}
```

Razred LinkedQueue:
zgradba enega elementa (vozlisca):

Objekt (naslov objekta)
Naslov naslednika
Naslov predhodnika


header
naslov slepega elementa
slepi element, ki ne vsebuje objekta
objekt 1
objekt 2
objekt 3
objektN

```
import java.util.*;

public class LinkedQueue extends AbstractQueue
{ private static class Node
  { Object object;
    Node next, previous;
    Node()
    { this.next = this.previous = this;
    }
    Node(Object object, Node next, Node previous)
    { this.object = object;
      this.nextt = next;
      this.previous = previous;
    }
  }
  private Node header = new Node();
  private int size = 0;

  public Object dequeue()
  { if (isEmpty()) throw new NoSuchElementException("queue is empty");
    Object object = header.next.object;
    header.next = header.next.next;
    header.next.previous = header;
    --size;
    return object;
  }

  public Object enqueue(Object object)
  { Node p = header.previous;  // last element in qusue
    header.previous = p.next = new Node(object,header,p);
    ++size;
    return object;
  }

  public Object getBack()
  { if (isEmpty()) throw new NoSuchElementException("queue is empty");
    return header.previous.object;
  }

  public Object getFront()
  { if (isEmpty()) throw new NoSuchElementException("queue is empty");
    return header.next.object;
  }

  public Iterator iterator()
  { return new Iterator()  // anonymous inner class
    { private Node cursor=header;
      public boolean hasNext()
      { return cursor.next != header;
      }
      public Object next()
      { if (cursor.next==header) throw new NoSuchElementException();
        cursor = cursor.next;
        return cursor.object;
      }
      public void remove()
      { throw new UnsupportedOperationException();
      }
    };
  }

  public LinkedQueue()
  {
  }

  public int size()
  { return size;
  }
}
```

```
import java.util.*;

public class ArrayQueue extends AbstractQueue
{ protected Object[] objects;
  protected int front=0;
  protected int back=0;
  protected int capacity=16;
  // INVARIANTS: objects[i] == null for 0 <= i < front;
  //             objects[i] != null for front <= i < back;
  //             objects[i] == null for back <= i < capacity;

  public ArrayQueue()
  { objects = new Object[capacity];
  }

  public ArrayQueue(int capacity)
  { this.capacity = capacity;
    objects = new Object[capacity];
  }

  public Object dequeue()
  { if (isEmpty()) throw new NoSuchElementException("Queue is empty");
    Object object = objects[front++];
    if (2*front>=capacity)  // shift left
    { for (int i=0; i<size(); i++)
        objects[i] = objects[i+front];
      back -= front;
      front = 0;
    }
    return object;
  }

  public Object enqueue(Object object)
  { if (back>=capacity)
    { Object[] temp = objects;
      capacity *= 2;  // double the capacity
      objects = new Object[capacity];
      for (int i=0; i<back-front; i++)
        objects[i] = temp[i+front];
      back -= front;
      front = 0;
    }
    objects[back++] = object;
    return object;
  }
  public Object getBack()
  { if (isEmpty()) throw new NoSuchElementException("Queue is empty");
    return objects[back-1];
  }

  public Object getFront()
  { if (isEmpty()) throw new NoSuchElementException("Queue is empty");
    return objects[front];
  }

  public Iterator iterator()
  { return new Iterator()  // anonymous inner class
    { private int cursor=front;
      public boolean hasNext()
      { return cursor<back;
      }
      public Object next()
      { if (cursor>=back) throw new NoSuchElementException();
        return objects[cursor++];
      }
      public void remove()
      { throw new UnsupportedOperationException();
      }
    };
  }

  public int size()
  { return back-front;
  }
}
```

stanje na začetku bo je vrsta prazna

header