Andmehaldusvahendid

Massiiv

Andmeid hoitakse Javas lihttüüpidena, vajadusel grupeeritakse neid selguse huvides struktuurtüüpidesse. Ühetüübiliste või ühest ülemklassist alanevat tüüpi andmeid saab hoida massiivis. Massiivi andmete töötlemiseks on loodud paketti `java.util` klass `Arrays`. Seal olevate meetodite abil on võimalik massiivi sortida, massiivist sobivat elementi leida, massiivi määratud väärtusega täita ning kaht massiivi võrrelda. Järgnev näide kirjutab sorteeritutena välja käsurea parameetritena antud sõnad. Lisaks teatatakse sõna "kass" asukoht massiivis või selle puudumine.

```
import java.util.*;
public class Sort2 {
    public static void main(String args[]) {
        Arrays.sort(args);
        for(int i=0; i<args.length; i++)
          System.out.println(args[i]);

        String otsitav="kass";
        int koht=Arrays.binarySearch(args, otsitav);
        if(koht>=0){
          System.out.println("Massiivis asub "+otsitav+
            " kohal nr. "+koht);
        } else{
          System.out.println(otsitav+" puudub massiivist");
        }
    }
}
```

```
D:\arhiiv\naited\io\muu>java Sort2 koer ahv kass lammas kits
ahv
kass
kits
koer
lammas
Massiiviis asub kass kohal nr. 1

D:\arhiiv\naited\io\muu>java Sort2 leevike tihane kurg
kurg
leevike
tihane
kass puudub massivist
```

Sorteerimiseks on käsk `sort`, otsimiseks on meetod `binarySearch`, mis eeldab massiivi eelnevat sorditust. Massiivi andmetega täitmiseks on meetod `fill` ning kaht massiivi aitab võrrelda `equals`. Samanimelise meetodiga võib sorteerida nii sõnesid, täisarve, reaalarve, kui igasugu muid objekte, st. nii liht- kui struktuurtüüpe. Sorteerides tõstetakse massiivi sees elemendid ringi. Sorteerimisalgoritmiks on kiirsortimise ning shell-sortimise vahepealne algoritm, kus sorteerimata algandmete korral ei kulu sortimiseks aega enam kui n*log(n), samas aga eelnevalt sorditud andmete korral töötab tunduvalt kiiremini. Sama väärtusega elementide järjekord jääb ka pärast sorteerimist samaks. Lihttüüpide korral võrreldakse andmete väärtusi, liittüüpide korral kasutatakse `compareTo` meetodit, eeldades, et massiivis asuvaid elemente on võimalik omavahel võrrelda. Probleemi tekkimisel heidetakse meetodist välja erind. Kui andmeid ei saa omavahel `compareTo` abil võrrelda või annab see kasutaja jaoks soovimatu tulemuse, siis saab kirjutada ise liidest `Comparator` realiseeriva klassi mille abil massiivis olevaid elemente võrrelda. Nii tuleks näiteks teha, kui soovime, et nimede tähestiku järjekorda seadmisel arvestataks täpitähti õigesti.

Järgnevas näites on loodud oma lihtne `Comparator`, mis võrdleb sõnade järjestust tähestikus, jättes esimese tähe arvestamata. Nii on Lammas eespool kui Koer, sest esimesel juhul on teiseks täheks a, teisel juhul o. Võrdleja kirjutamisel on pääsetud lihtsalt, sest on kasutatud kahe sõne võrdlemiseks tarvitatavat meetodit compareTo. Üle kaetud meetodile compare antakse ette kaks sorteeritava massiivi elementi, millede vahelist omavahelist järjestust on vaja sorteerimisel teada. Ehkki meetodi päises on mõlema elemendi tüübiks Object, võime praegu päris kindlad olla, et tegelikult saabuvad andmed on tüübist String. Seda lihtsalt sellepärast, et me ise anname sorteerida String-tüüpi massiivi ning mujalt pole pakutavaid elemente kusagilt võtta. Et aga kõiki struktuurtüüpe võib Javas `Object`iks ning tagasi muuta, sel juhul on mugav kirjutada liideses `Comparator` meetodi compare parameetriteks kaks

`Object`i ning lasta juba edaspidi programmeerijal sealt omale vajalikku tüüpi andmed tagasi välja võtta. Enamasti tuleb andmeid omale sobivaks muundada tüübimuunduse ehk `cast`'iga ehk `String s1=(String)o1;` , kuid kuna klassis `Object` (ning ka kõigis tema alamklassides ehk kokkuvõttes üldse kõigis Javas ette tulevates klassides) on meetod `toString` ning `String`i puhul väljastab see lihtsalt tema väärtuse, siis saab sõne väärtuse ka lihtsalt kätte, kirjutades `o1.toString()`.

```
import java.util.*;
public class Sort3{
  public static void main(String[] argumendid){
    String[] loomad={"Koer", "Kass", "Lammas", "Lehm", "Elevant", "Kits"};
    Arrays.sort(loomad, new TeisestTahestVordleja());
    for(int nr=0; nr<loomad.length; nr++){
      System.out.println(loomad[nr]);
    }
  }
}

class TeisestTahestVordleja implements Comparator{
  public int compare(Object o1, Object o2){
    return o1.toString().substring(1).compareTo(o2.toString().substring(1));
  }
}
```

```
C:\kodu\jaagup\0204\k1>java Sort3
Lammas
Kass
Lehm
Kits
Elevant
Koer
```

Java Collections Framework

Lisaks massiivile on Java keeles ka muid vahendeid andmete hoidmiseks ja töötlemiseks. Mitmetes masinalähedasemates programmeerimiskeeltes aitavad andmetega kiiremini ja paindlikumalt ümber käia viidad, siin asendavad neid osutid. Et programmeerijad saaksid paremini ja paremaid programme kirjutada, selleks on loodud andmehulkade hoidmist ning nendega manipuleerimist kirjeldav liideste kogum, Java Collections Framework. Liidesed on kaetud klassidega, kuid neid saab vajadusel ise luua ning tõenäoliselt luuakse ka Java arendajate poolt valmis liidestele konkreetsetesse oludesse klasse juurde. Java loojad pakuvad välja Collections Framework'i kasutamisel välja järgmised eelised.

Programmeerimise kiirus kasvab, kuna ei tule pidevalt luua uusi klasse eri tüüpi andmega tegelemiseks ega andmete muundamiseks. Ka programmi enese kiirus peaks kasvama, kuna optimeeritud valmisklasside kasutamisel ei pea nende kallal enam vaeva nägema, samas aga on kindlasti võimalik vabaneva ajaga ülejäänud programmi paremaks teha. Selge süsteem võimaldab kergemini andmeid vahetada teiste programmidega. Õppimine ning ajaga kaasas käimine peaks minema lihtsamaks, kuna põhilised osad jäävad samade liideste kasutamisel ka erinevatel andmetel samaks. Uue andmetöötlussüsteemi välja töötamisel saab olemasolevad liidesed aluseks võtta, kuna neis on juba läbikaalutud meetodid olemas. Suuremas plaanis suurendab ühtne liideste komplekt programmi taaskasutust, sest siis on kergem ennustada, mida ühelt või teiselt programmijupilt oodata võib.

Loomulikult on aga ühtsel süsteemil omad puudused. Mõnes olukorras on ta kohmakam, sest harva on võimalik optimiseerida sama hästi erinevate ootuste jaoks kui seda saaks ühe kindla eesmärgi jaoks teha. Samas aga on ehk korralik kombain parem kui ülejala tehtud ning läbi kontrollimata vahend. Tavanäiteks võiks olla, et kärbsepiits-klopper-pudeliavajaga on raskem kärbest kinni püüda kui lihtsa

viimistletud kärbsepiitsaga, kuid tõenäoliselt hulga tulemuslikum kui tavalise tolmulapiga, mis meil enamasti juhtub käepärast olema.

Kollektsioon

Andmeliideste juureks on `Collection`, igasugune andmekogum. Kirjeldatud on mõniteist meetodit, mida nende andmetega teha saab. Meedodi `add` abiga saab elemendi lisada, `remove` võtab vastava elemendi kollektsioonist ära; `addAll` lisab terve teise kollektsiooni, `clear` teeb platsi puhtaks; saab kontrollida, kas kollektsioon on tühi, kui palju temas on elemente, kas temas leidub määratud objekt, kas temas sisaldub teine kollektsioon või on ta hoopistükkis teisega samaväärne. Vastavad meetodid oleksid `isEmpty`, `size`, `contains`, `containsAll` ja `equals`. Kollektsioonist saab andmeid kätte, muundades ta massiiviks või küsides temalt objektijada tüübist `Iterator`. Selle abil saab ükshaaval läbi vaadata kõik kollektsiooni elemendid, vajadusel küsides osuti vajalikule elemendile või eemaldades ta kollektsioonist.

Nimistu

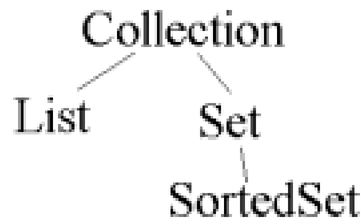
`List` on järjestatud kollektsioon, ehk selline andmekogum, kus igal elemendil on järjekorranumber. Ta on kollektsiooni alamliideseks. Juurde tulevad meetodid, mis on seotud järjekorranumbriga. Näiteks saab lisada elementi määratud kohale, samuti eemaldada või küsida teda sealt. Elemendi lisamisel temast paremale jääjate järjekorranumber suureneb ühe võrra, eemaldamisel väheneb. Kui sooviksime massiivi keskele lisada, peaksime vastava algoritmi ise kirjutama, `List`i ehk nimekirja puhul on see juba olemas. Massiivi saab nimekirjaks muuta klassi `Arrays` käsuga `asList`, vastupidi saab aga nimekirjale öeldes `toArray()`. Kui massiivis asuvad elemendid enamjaolt üksteise järel ning nende asetsemist mälus saab muuta üksnes intepretaator või operatsioonisüsteem, siis `List` on vaid liides ning tema realisatsioonis saab vabalt määrata, kas elemente hoitakse osutite abil teineteisele viidates, massiivis või hoopis mõnel muul moel (kas või näiteks faili abi kasutades). Liides `List` on vaid kirjeldus, kuidas andmetele ligi pääseb.

Hulk

`Set` on samuti kollektsiooni alamliides. Temas võib iga väärtusega elementi olla vaid üks nagu matemaatilises hulgas kohane. Liides `Set` ei ütle aga midagi elementide järjekorra kohta. Temasse lisades jäävad alles vaid erinevad elemendid. Nii saab tema abil kergesti suurest komplektist erinevad välja sõeluda.

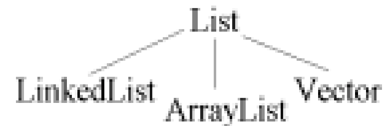
`SortedSet` on `Set`'i alamliides. Nagu nimigi ütleb, on temas elemendid järjestatud. Lisaks muule `Set`'i võimalustele saab temalt küsida esimest ja viimast elementi, samuti alamhulki alates või kuni mingi väärtuseni ning hulka kahe väärtuse vahelt.

Nagu klass `Arrays` sisaldab meetodeid massivi töötlemiseks, nii klassi `Collections` meetodid aitavad ümber käia kollektsiooni ning tema alanejatega. Saab leida kollektsioonist vähimat ning suurimat elementi, muuta kollektsioon sünkroniseerituks (s.t. et tema poole saab korraga pöörduda vaid üks lõim) või keelata temasse kirjutamine. Klass aitab sorteerida `list`i, samuti sorteeritud `list`is elemente segi paisata (meetod `shuffle`). Võib keerata järjekorra vastupidiseks, täita nimekirja määratud elemendiga või otsida elementi.



Tegelikud realisatsioonid

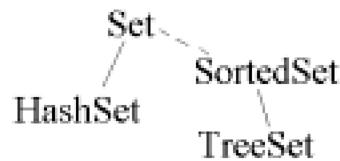
Et programmeerija ei peaks vaid ilusaid kirjeldusi vaatama, vaid saaks ka välja pakutud hüvesid ise neid loomata maitsta, selleks on loodud liidesed ka realiseeritud. Kollektsiooni realiseerijaks on küll vaid `AbstractCollection`, mis mõeldud aitamaks ise kollektsiooni luua, kuid muudele liidestele on kasutatavad katted olemas. Listi realiseerijateks on määratud `LinkedList`, `Vector` ning `ArrayList`.



Kasutaja jaoks käituvad nad ühtmoodi nii, nagu liideses kirjeldatud, sisimas aga on erinevad ning suuremate andmehulkade korral on kasulik nende eripära arvestada. `LinkedList` on jadas paiknevate elementide kogum, igalt elemendil (v.a. viimane) on osuti järgmisele. Sellise lahenduse juures on suhteliselt kerge elemente keskele vahele panna, sest piisab vaid kahe (pandava ning temale eelneva) elemendi ümber tõstmisest. Samas on pika jada keskele jõudmine suhteliselt vaevarikas, sest tuleb kõik vahepeal asuvad elemendid läbi käia, et otsitu juurde jõuda.

`Vector` ning `ArrayList` hoiavad oma andmeid massiivis. Andmetele on lihtne ligi pääseda, kuid eemaldamiseks või lisamiseks tuleb kõiki muudetavast taga pool paiknejaid ühe võrra liigutada, mis jällegi suuremate andmete puhul päris palju operatsioone nõuab. Samuti saab järjest andmeid lisades ükskord massiiv lihtsalt täis. Sellisel puhul luuakse uus suurem massiiv ning kopeeritakse andmed sinna. Et andmete lisamisel ümber kopeerimist liialt tihti ette ei tuleks, selleks luuakse igal korral vähemalt `Vectori` puhul eelmisest kaks korda suurem massiiv.

Set-liidese realiseerijaks on `HashSet` ning `SortedSet` võimalusi aitab kasutada `TreeSet`. Viimane vähemasti standardrealisatsioonis kasutab kahendpuud ning selle algoritmi järgi on soovitud väärtusi küllalt kiire leida ka suuremate andmehulkade puhul.



Järgnevas näites luuakse alustuseks loomanimedest sõnemassiiv. Massiiv muudetakse `Vector`'iks, mis realiseerib liidest `List`. Nagu nimistule kohane, saab temasse andmeid soovitud kohta lisada ning sealt eemaldada. Siin paigutatakse kohale nr. 2 (ehk kolmandaks) rebane. Väljatrükil on näha, et parempoolsed elemendid on ühe võrra edasi nihkunud. Nimistu muudetakse `HashSet`i abil hulgaks. Väljatrükil on näha, et korduv koer on kadunud. Ka muud elemendid on oma kohta vahetanud, kuid see on lubatud, sest hulgas pole elementide järjekord määratud. Lisades kitse ja lamba, tuleb juurde vaid kits, sest lammas oli hulgas juba olemas. Hulga sorteerimiseks paigutame andmed isendisse tüübist `TreeSet`. Selles klassis jäävad andmed sordituks ka pärast uute elementide lisamist.

```
import java.util.*;
public class Andmed{
  public static void main(String argumendid[]){
    String loomamassiiv[]={"kass", "koer", "lammas", "koer", "ahv"};
    List loomalist=new Vector(Arrays.asList(loomamassiiv));
    loomalist.add(2, "rebane");
    System.out.println(loomalist);
    Set loomahulk=new HashSet(loomalist);
    System.out.println(loomahulk);
    loomahulk.add("kits");
    loomahulk.add("lammas");
    System.out.println(loomahulk);
    SortedSet sorteeritudLoomahulk=new TreeSet(loomahulk);
    System.out.println(sorteeritudLoomahulk);
    sorteeritudLoomahulk.add("antiloop");
    System.out.println(sorteeritudLoomahulk);
  }
}
```

Ning programmi töö tulemus:

```
[kass, koer, rebane, lammas, koer, ahv]
[ahv, lammas, kass, rebane, koer]
[ahv, kits, lammas, kass, rebane, koer]
[ahv, kass, kits, koer, lammas, rebane]
[ahv, antiloop, kass, kits, koer, lammas, rebane]
```

Paisktabel

Objektipaaride jaoks on loodud liidesed `Map` ning `SortedMap`. Neid realiseerivate klasside `HashMap`, `Hashtable` ning `TreeMap` abil saab võtmetele panna vastama väärtused, samuti küsida võtmetele vastavaid väärtusi. Neid saab kasutada nagu sõnaraamatut, kuid seletuste asemel ei pruugi olla vaid sõnad, vaid sobivad igasugu objektid. All näites on võtmeteks valvamiskohad ning väärtusteks inimeste nimed, kes vastaval kohal olema peavad.

```
import java.util.*;
public class Paisktabel{
  public static void main(String[] argumendid){
    Hashtable korrapidajad=new Hashtable();
    korrapidajad.put("koridor", "Juku");
    korrapidajad.put("klass", "Kati");
    System.out.println(korrapidajad);
    korrapidajad.put("klass", "Mari");  //asendab Kati
    System.out.println("Koridoris valvab:"+korrapidajad.get("koridor"));
    System.out.println("Valvekohti kokku: "+korrapidajad.size());
    Enumeration kohaloend=korrapidajad.keys();
    while(kohaloend.hasMoreElements()){
      String koht=(String)kohaloend.nextElement();
      System.out.println("Valvekoht: "+koht+", valvaja: "+korrapidajad.get(koht));
    }
    Collection valvajad=korrapidajad.values();
    Iterator valvajaloend=valvajad.iterator();
    System.out.println("Valvajad:");
    while(valvajaloend.hasNext()){
      System.out.println(valvajaloend.next());
    }
  }
}
```

```
D:\kodu\0309\oma>java Paisktabel
{klass=Kati, koridor=Juku}
Koridoris valvab:Juku
Valvekohti kokku: 2
Valvekoht: klass, valvaja: Mari
Valvekoht: koridor, valvaja: Juku
Valvajad:
Mari
Juku
```

Ülesanded

Sünniaastad

Koosta tekstifail, milles on hulk sünniaastaid.

- Leia, mitu korda esineb nende seas aasta 1959
- Loe tekstifaili aastaarvud LinkedList-i
- Väljasta need aastaarvud sorteerituna teise tekstifaili
- Otsi, kas loetelus leidub kasutajalt küsitud aastaarv
- Väljasta teise tekstifaili vaid erinevad aastaarvud
- Sega aastaarvude järjekord

Dokumenteerimine

Paarist reast pikemate programmide mõistmiseks ei pruugi põgusast pealevaatamisest piisata. Meetodite ja klasside nimed veidi selgitavad olukorda, kuid peetakse sobivaks ka pikemaid seletusi programmilõikude töö ning kasutusvõimaluste kohta anda. Enesel on nii lihtsam koodi meelde tuletada ning annab grupitöö korral võimaluse teistel programmeerijatel tehtut kasutada, ilma et peaks autorilt selgitusi pärima või iga programmirea eesmärki eraldi uurima. Java lähtekoodi saab kommentaare lisada kahel moel. Tekst kahest järjestikusest kaldkriipsust rea lõpuni loetakse kommentaariks näiteks:

```
int arv=3 // kassi poegade arv
// poegade arvu järgi tuleb arvestada toiduvajadust
double toidukulu=arv*koefitsient
```

Pikemat programmi osa saab välja kommenteerida märkides selle osa alguse ja lõpu. Algust tähistab `/*` ning lõppu `*/`. Valikuvõimaluse puhul soovitakse kasutada ridade kaupa väljakommenteerimist, pidada vähem vigu tekkima.

Programmifaile iseloomustavate HTML-lehtede loomiseks kuulub JDK programmide koosseisu `javadoc`, mis koondab programmidest meetodite kirjeldused koos vastavalt märgistatud kommentaaridega. Vastavalt sihtgrupile võib lasta javadocil välja tuua mitmesuguse tähtsusastmega programmiosi. Näiteks kaasprogrammeerijatele tuleks näidata ka vaid paketi või klassi piires kasutatavaid muutujaid, klassi väljapoolt kasutajale pole neid tarvis. Ka loob `javadoc` võimaluse korral viited muude klasside kirjeldustele. Lisaks sellele on siiski sageli viisakas lisada "kirjeldav" ülevaade ning kasutamisõpetus. `Javadoc`i abil on koostatud ka JDK API dokumentatsioon, mida abiinfona kasutame. Selliselt automaatsena loodud dokumentatsiooni puhul on eelis, et andmed ei saa inimlike eksituste tõttu valeks minna. Näiteks C-keele puhul võis kergemini ette tulla olukordi, kus käsud on küll valmis kirjutatud ja olemas, kuid neid ei tea kasutada, kuna on unustatud dokumentatsiooni lisada või on selle kirjutamise juures viga tekkinud.

Klassi ja meetodite kirjelduste lisamiseks javadoci abil loodud HTML-faili tuleb need kirjeldused paigutada `javadoc`ile mõistetavalt kujundatuna klassi või meetdi ette. Äratundmiseks peab kommentaar algama reaga, millel kaldkriips ja kaks tärni. Iga rea algul on üks tärn ning kommentaari lõpus tärn ja kaldkriips. Teksti esimene rida peab olema kokkuvõttev - see tõstetakse meetodite omaduste kokkuvõtete tabelisse. Edasine pikem kirjeldus on näha vaid tagapool. Sellisena on hea kiiresti ülevaade saada ning suurest meetodite hulgast omale sobivad välja leida. Kirjelduse sees võib kasutada HTML kujundust. Erikirjelduste abil saab määrata tunnuste väärtusi. `@author` tähendab programmifaili autorit, `@since` versiooni, millest alates seda programmi kompileerida ning kaivitada saab. `@see` loob viite olemasolevale meetodile, mis kommentaari kirjutaja selles kohas tähsaks viidata peab.

Järgnevalt on Javadoci kommenteerimise näitena toodud klass punkti koordinaatide hoidmiseks ning sealt väärtuste pärimiseks.

```
/**
* Klass abistab <b>kasutajat</b>
* punkti andmete hoidja ning analüüsijana.
* Koostatud <a href="http://www.tpu.ee">Tallinna Pedagoogikaülikoolis</a>.
* @see #kaugusNullist
* @author Jaagup Kippar
* @since JDK1.0
*/
```

```
public class DokumenteeritudPunkt{
  /**
  * x-koordinaat
  */
  public int x;

  /**
  * y-koodinaat
  */
  public int y;

  /**
  * Klassimuutuja loodud punktieksemplaride arvu loendamiseks.
  * @see #teataPunktideArv
  */
  static int punktideArv;

  /**
  * Parameetriteta konstruktor, jätab muutujate väärtused nulliks.
  */
  public DokumenteeritudPunkt(){
    x=y=0;
  }

  /**
  * Konstruktor lähteandmete sisestamiseks.
  * <p>
  * Konstruktoris määratud parameetrid jäävad
  * vastavate muutjate väätusteks
  * @param x loodava punkti x-koordinaat
  * @param y loodava punkti y-koordinaat
  */
  public DokumenteeritudPunkt(int x, int y){
    this.x=x;
    this.y=y;
  }

  /**
  * Leitakse vahemaa tasandil oleva punkti asukoha ning
  * koordinaatide alguspunkti vahel.
  * Vahemaa leitakse ruutjuure abil koordinaatide ruutude summast.
  * @return kaugus koordinaatide alguspunktist reaalarvuna.
  */

  public double kaugusNullist(){
    return Math.sqrt(x*x+y*y);
  }

  /**
  * Väljastatakse loodud punktide arv.
  * @return programmi töö keskel loodud vastavat tüüpi isendite arv.
  */

  public static int teataPunktideArv(){
    return punktideArv;
  }
}
```

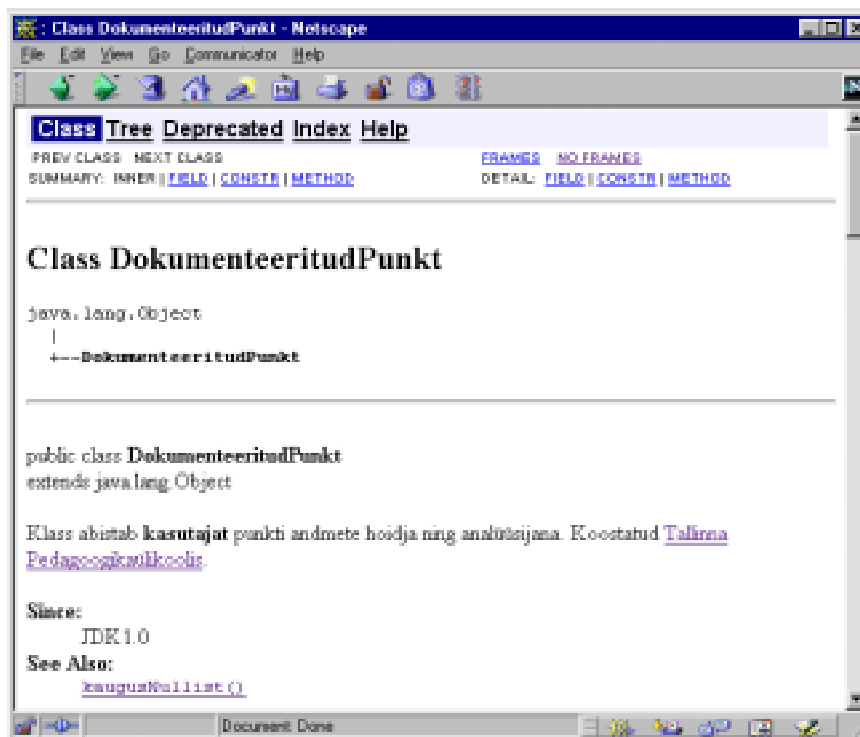
Dokumentatsiooni loomine käivitatakse, kirjutades javadoc ning dokumenteeritava faili nimi. Soovi korral saab mitmesuguste võtmetega päris palju dokumentatsiooni loomise juures määrata. Aeglasema masina puhul võib javadoci töö päris hulga sekundeid aega võtta. Samuti venib aeg pikemaks, kui korraga koostatakse dokumentatsiooni mitte ühele lähtekoodifailile vaid rohkematele. Samuti luuakse töö käigus päris hulk HTML-faile, mis võtab ka oma aja.

```
C:\kodu\jaagup\0204\k1>javadoc DokumenteeritudPunkt.java
Loading source file DokumenteeritudPunkt.java...
Constructing Javadoc information...
Building tree for all the packages and classes...
Building index for all the packages and classes...
Generating overview-tree.html...
Generating index-all.html...
Generating deprecated-list.html...
Building index for all classes...
Generating allclasses-frame.html...
Generating index.html...
```

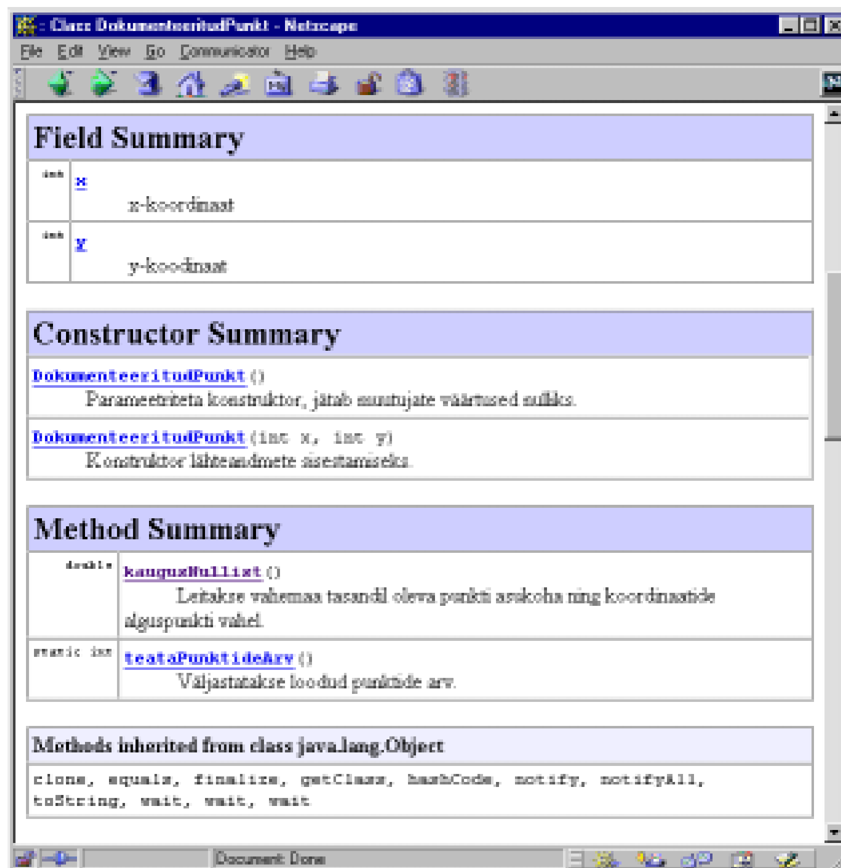
```
Generating packages.html...
Generating DokumenteeritudPunkt.html...
Generating serialized-form.html...
Generating package-list...
Generating help-doc.html...
Generating stylesheet.css...
```

Javadoci töö tulemuseks on loodetavasti juba tuttava väljanägemisega abiinfo leht. Ülaservas olevate viidetega on rohekem peale hakata, kui on tegemist rohkemate klasside ja meetoditega (nagu näiteks standarpakettide puhul). Tree näitab puuna välja kõik dokumentatsioonis leiduvad klassid. Deprecated-loetelus on kirjas vananenud ning ebasoovitavad meetodid. Index-ist võib tähestiku järjekorras leida soovitud klassi või meetodi - hea abiline juhul, kui mäletad küll käsu ligikaudset nime, aga ei tule meelde, kust kandist seda otsida võiks.

Edasi allpool tulevad konkreetse klassi andmed. Nimi ning joonis, kuidas vastav klass asub objektide hierarhias. Et loodud DokumenteeritudPunkt'il eraldi ülemklassi pole, on tema ülemklassiks kõige juur Javas ehk java.lang.Object. Edasi tulevad juba lähtekoodi seest välja korjatud kommentaarid.



Siis nii väljade, konstruktorite, meetodite lühikirjeldused ning päritud meetodite nimed. Viimaste kirjelduste vaatamiseks tuleks juba üles otsida vastava klassi (praegusel juhul java.lang.Object) dokumentatsioon.



Allapoole minnes juba leiab iga käsu detailsema kirjelduse, samuti seletused, mis on määratud parameetrite ning väljastatavate väärtuste juurde.