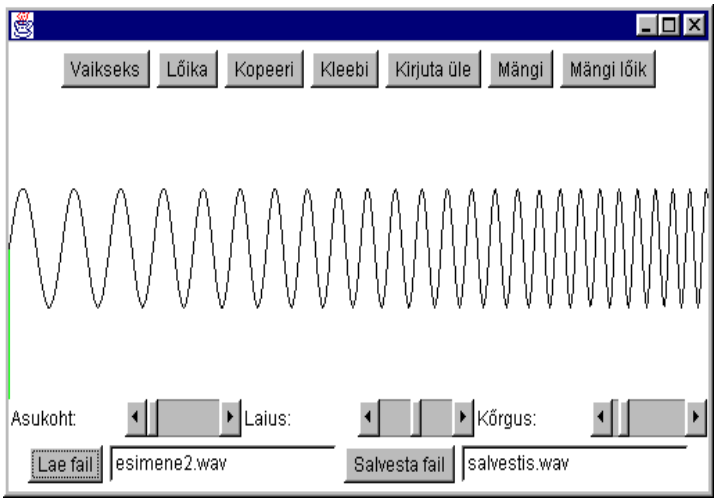
Tallinna Pedagoogikaülikool

Informaatika osakond

Graafika ja muusika programmeerimine



Jaagup Kippar

Tallinn 2003

Eessõna

Käesolev konspekt sisaldab näiteid ja seletusi mitmete graafika ning muusika programmeerimisega seotud valdkondade kohta. Lugejalt eeldatakse varasemat programmeerimiskogemust vähemasti Java põhikursuse konspektis kirjutatu tasemel. Koostamisel kasutati enamjaolt TPÜ programmeerimispraktikumides läbi lahendatud näiteid ning kursustel läbitud teemasid, kuid piisava süvenemise korral peaksid teemad olema mõistetavad keskkoolirahvalegi. Juhendajad võiks siit leida materjale ka põhikooli programmeerimisringide tarbeks. Sugugi ei pea rakendust enese tarbeks kohandamisel kohe lõpuni mõistma. Ükshaaval parameetreid muutes ning lisandusi tehes tekib paratamatult ka üldpildist mõningane ettekujutus. Samas on püütud näited vähemalt teemade algul koostada võimalikult lühidad, et soovijad suudaksid kõiges näpuga järge ajada ning iga koodirea eesmärki mõista. Tahtes omaloodud rakendust lõpuni usaldada ning suuta igale ilmnevale veale põhjus leida ning see parandada, peab kogu toimuvast ülevaade olema.

Teemade lõpus paiknevad ülesanded on mõeldud enam enesekontrolliks, kuid on samas praktikumides või eksamitöödena läbi lahendatud ning peaksid sobima ka arvutitundides ettevõtmiseks. Ülesande punktid on püütud teha vähemalt kolme raskustaseme jaoks, kusjuures esimene võiks olla pea igale teemaga kokkupuutunule jõukohane. Mitmedki ülesanded on mõeldud ideede ärgitamiseks ning on tänuväärne, kui õpetaja või lahendaja ise suudab selle põhjal hetkel huvitava ning vajaliku ülesande sõnastada.

Konspekti valmimisele kaasa aitamise eest tänan TPÜ üliõpilasi, kes on näiteid kasutades ning ülesandeid lahendades andnud põhjust täiendusi ja parandusi sisse viia. Tänud Kaur Männikole, kelle loodud MIDI redaktor sobis parajasti vastavat teemat illustreerima, nii et konspekti koostajal polnud põhjust paremat looma hakata. Ülejäänu osas on pea täielikult tegu autori originaalkoodiga.

Vastuseks varemgi esitatud küsimusele, et kas siin kirjutatud materjali tohib igaüks oma tarbeks või koolitundides või muul viisil kasutada vastan julgesti jah. Leian, et kõik need, kes tahavad, suudavad ja viitsivad eestikeelset õppematerjali tarvitada, väärivad piisavalt austust, et mitte hakata oma rahva seas piirangute pärast kemplema. Samas olen tänulik teadete kohta, kus siinse konspekti materjalid kasutust on leidnud.

Head koodikirjutamist!

Jaagup Kippar

Sisukord

Graafiku koostamine.

Joonistuskäsud, ekraanikoordinaadid, maailmakoordinaadid, skaala, ümardamine

Mõnikord tavatsetakse öelda, et üks pilt on rohkem väärt kui tuhat sõna. Ei pruugi see alati kehtida, kuid joonistest võib vahel kasu küll olla. Kui andmed ei muutu, siis kannatab mõne vastavaotstarbelise programmiga pildid valmis teha ning tekstile juurde liita. Kui aga andmed muutuvad või ei olda olemasoleva programmi poolt pakutava väljundiga rahul, siis on põhjust ise koodilõik kirjutada, mis andmed jooniseks muundab.

Nagu programmide ja ka muude toimingute puhul, alustame lihtsamast ja liigume keerulisema suunas. Näited tehakse pideva ruutfunktsiooni tarvis, kuid sarnased arvutused tuleb ka muul puhul ette võtta, kui oma andmetele vastavalt püüame miskit ekraanile paigutada.

Üksikud ekraanipunktid.

Joonistamisel samastatakse graafiku matemaatilised maailmakoordinaadid ning arvuti ekraanikoordinaadid. Funktsiooni väärtus arvutatakse täisarvuliste x-ide juures ning vastavale x ja y kombinatsiooniga määratud kohale joonistatakse täpike.


```
import java.awt.*;
import java.applet.*;
public class Graafik1 extends Applet{
  public void paint(Graphics g){
    for(int x=0; x<100; x++){
      g.drawOval(x, x*x, 1, 1);
    }
  }
}
```

Nihutus ja keeramine

Üksühene arvutuskoordinaatide ja ekraanipunktide vastavus võib küll mugav koodiks kirjutada olla ning esmase pildi selle abil ka saab, kuid selline telgede asetus võib arvutigraafikast kaugemal seisvale vaatajale harjumatu ning võõrastav tunduda. Samuti tekivad probleemid x-i ja y-i negatiivsete väärtuste juures, sest neid pole lihtsalt näha. Veidi mugavam peaks olema järgmine lähend:

Üks ühik graafikul vastab endiselt ekraanipunktile. Joonist aga nihutati x-teljel 100 punkti võrra paremale ja y-teljel 200 punkti võrra allapoole ning y-telje kasvamise suund keerati vastupidiseks.

Ekraani y-punkti arvutamisel lahutatakse kahesajast maha graafikul oleva igreki väärtus. Kui y=0, siis joonistatakse täpp rakendi y-koordinaadile 200. Kui y>0, siis tuleb 200-y kahesajast väiksem ning täpp ekraanil järelikult kõrgemal. Negatiivse y puhul ületab 200-y kahtsadat ning tulemuseks on täpp nullpunktist (kahesajast) allpool.


```
import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;
import java.applet.*;
public class Graafik2 extends Applet{
  public void paint(Graphics g){
    for(int x=-100; x<100; x++){
      g.drawOval(x+100, 200-x*x, 1, 1);
    }
  }
  public static void main(String argumendid[]){
    Frame f=new Frame("Ruutfunktsiooni graafik");
    f.add(new Graafik2());
    f.setSize(250, 250);
    f.setVisible(true);
  }
}
```

Mõõtkava.

Koodi sisse trükitud konstandid on asendatud muutujatega. Nii on vähe suurema programmi puhul kergem näha, millega tegu, sest muutujal on nimi. Samuti, kui sama suurust on kasutatud mitmes kohas, siis muutmisvajaduse korral saab väärtuse määrata ühes kohas ning jääb ära oht, et kusagilt midagi paika muutmata jääb. Põhjalikud programmeerijad soovitavad koguni kõik väärtused (v.a. 0 ja 1) asendada tähenduslike nimedega muutujatega. Siis teistel koodi lugejatel rohkem lootust aru saada. Ning mõnikord on mõistlik needki numbrid sõnadega asendada.

Arvutustehe on toodud välja eraldi funktsioonina. Nii on teda kergem mitmest kohast vajadusel välja kutsuda, samuti muuta ja parandada kui vajadust peaks olema.

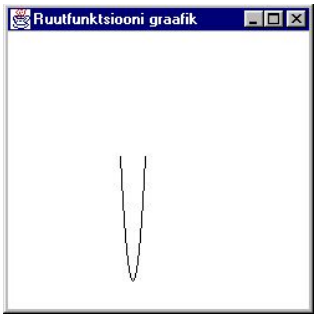
Koefitsent näitab, mitu ekraanipunkti vastab ühele ühikule graafikul. Kui koefitsent on võrdne ühega, siis graafiku- ning ekraanipunktide arv kattub. Ühest suurema kordaja korral venitatakse ekraanil graafik vastava telje suunas välja. Kui kordaja on 0 ja 1 vahel, siis ekraanijoonist vähendatakse. Ning kui kordaja on alla nulli, siis joonistatakse ekraanil maailmakoordinaatidele vastupidises suunas.

Siin näites y koefitsent -0.5 tähendab, et ekraani- ning graafikuühikud on eri suundades (miinusmärk ees) ning ekraanipunkte on poole vähem kui punkte graafikul. Nii on võimalik kiirelt kasvav ruutfunktsioon ekraanile paremini ära mahutada.


```
import java.awt.*;
import java.applet.*;
public class Graafik3 extends Applet{
  double minx=-10, maxx=10, samm=1;
  double koefitsientx=3, koefitsienty=-0.5;
   //mitu ekraanipunkti vastab ühele ühikule joonisel
   //miinus vahetab suuna
  int xnihe=100, ynihe=200;
  public void paint(Graphics g){
    for(double x=minx; x<=maxx; x=x+samm){
      g.drawOval(xnihe+(int)(  x *koefitsientx),
                 ynihe+(int)(f(x)*koefitsienty), 1, 1);
    }
  }
  double f(double x){
    //funktsioon eraldi välja, siis kergem
    //kasutada ja muuta
    return x*x;
  }
  public static void main(String argumendid[]){
    Frame f=new Frame("Ruutfunktsiooni graafik");
    f.add(new Graafik3());
    f.setSize(250, 250);
    f.setVisible(true);
  }
}
```

Pideva joonega graafik

Endiste täppide asemel ühendatakse nüüd punktid omavahel joontega. Kui jooned piisavalt lühikesed on, siis jääb vaatajale mulje, nagu oleks tegemist kõveraga. Kõigepealt tuleb välja arvutada esimese punkti asukoht. Seejärel leida teise punkti asukoht ning kahe esimese punkti vahele tõmmata joon. Siis jäetakse viimatiarvutatud punkt meelde, arvutatakse järgmine, tõmmatakse
taas joon ning jäetakse punkt meelde. Kuni ollakse jõudnud arvutatava lõigu lõpuni.




```
import java.awt.*;
import java.applet.*;
public class Graafik4 extends Applet{
  public void paint(Graphics g){
    int vanax=90;
    int vanay=100;
    for(int x=-10; x<=10; x++){
      int uusx=x+100;
      int uusy=200-x*x;
      g.drawLine(vanax, vanay, uusx, uusy);
      vanax=uusx;
      vanay=uusy;
    }
  }
  public static void main(String argumendid[]){
    Frame f=new Frame("Ruutfunktsiooni graafik");
    f.add(new Graafik4());
    f.setSize(250, 250);
    f.setVisible(true);
  }
}
```

Komponendi suuruse arvestamine

Head programmid pidavat suutma arvestada ressurssidega, mis neile kättesaadavad on. Et kui mäluga arvutis kitsas, siis hoitakse enam andmeid kettal ja lihtsalt arvutatakse mõnevõrra kauem. Või kui ekraanipinda vähem kasutada, siis jäetakse nähtavale vaid tähtsaimad lõigud. Püüame ka siin niimoodi teha.

Raami suuruse muutmisel arvutatakse pilt uuesti-käivitatakse paint. Nagu näha, seal on töö mitmeks alalõiguks jagatud ning vaid otsesed joonistuskäsud paint'i sisse jäänud. Suurenduseks tarvilikud konstandid leitakse eraldi alamprogrammis, samuti paigutatakse joonise servadesse koordinaadid. Ka maailmakoordinaatide ja ekraanikoordinaatide teisendamiseks on omaette funktsioon loodud. Joonise tarvis arvutatakse välja kõigepealt vasakpoolseim x ja y. Edasi igal järgmisel korral tõmmatakse joon eelmisest punktist uude punkti ning jäetakse uus punkt eelmisena meelde.

```
public void paint(Graphics g){
  leiaKonstandid();
  joonistaKoordinaadid(g);
  int vanax=ekraaniX(minx);
  int vanay=ekraaniY(f(minx));
  for(double x=minx; x<=maxx; x=x+samm){
    int uusx=ekraaniX(x);
    int uusy=ekraaniY(f(x));
    g.drawLine(vanax, vanay, uusx, uusy);
    vanax=uusx;
    vanay=uusy;
  }
}
```

Joonistamisel arvestatakse ekraanipinna suurust ning nii x- kui y-teljel funktsiooni

suurimaid ja vähimaid väärtusi. Selle järele arvutatakse koefitsendid. Siin näites antakse ette x-i vähim ja suurim väärtus ning funktsiooni kõvera punktid arvutatakse iga kahekümnendiku arvutuspiirkonna pikkuse tagant. Et funktsioon ega arvutusvahemik praegu programmi töö ajal ei muutu, saab suurimad ja vähimad väärtused leida juba rakenduse töö algul.

```
double minx=-10, maxx=10, samm=(maxx-minx)/20;
double miny=minY(), maxy=maxY();
```

Suurima y-i leidmiseks käiakse lihtsalt kogu funktsioon arvutussammu pikkuste lõikude tagant läbi ning jäetakse meelde suurim väärtus. Tegu pole küll matemaatiliselt päris korrektse lähenemisega, sest võib juhtuda, et kuhugi arvutuskoha vahele satub piik, kus funktsiooni väärtus erineb tunduvalt kõrval asuvatest väärtustest, kuid harilike funktsioonide puhul peaks sellisest lähenemisest piisama.

```
double maxY(){
  double max=f(minx);
  for(double x=minx; x<=maxx; x+=samm){
    if(f(x)>max)max=f(x);
  }
  return max;
}
```

Kui vähimad ja suurimad väärtused olemas, siis edasi tasub leida muud joonistamisel tarvilikud kordajad. Ning põhiliseks määrajaks on sel korral kasutaja ette antud komponendi suurus, mille annab pärida getSize abil. Suurenduskordaja leidmiseks arvutatakse nii x- kui y-suunal väärtuste ulatus maailmakoordinaatides. Edasi leitakse, mitu ekraanikoordinaati selle ala peale jagub. Et ei tekiks jagamist nulliga, on ära määratud vähim ulatus, millest väiksemaks ei või maailmakoordinaatide vahemik minna. Samuti jäetakse ekraanil mõningane servaruum, et joonte otsad päris äärteni välja ei läheks. Ka leitakse funktsiooni nähtavale osale keskpunkt, et oleks võimalik joonis ekraanil suhteliselt ühtlaselt paigutada.

```
void leiaKonstandid(){
  korgus=getSize().height;
  laius=getSize().width;
  ulatusx=maxx-minx;
  ulatusy=maxy-miny;
  if(ulatusx<ulatusmin)ulatusx=ulatusmin;
  if(ulatusy<ulatusmin)ulatusy=ulatusmin;
  koefitsientx=(laius-2*servaruum)/ulatusx;
  koefitsienty=-(korgus-2*servaruum)/ulatusy;
  keskx=(maxx+minx)/2;
  kesky=(maxy+miny)/2;
}
```

Ka servadesse koordinaatide joonistamiseks on loodud eraldi meetod. Laiust pidi on jäetud iga väärtuse tarvis 40 ja kõrgust pidi 30 punkti. Selle järgi leitakse, mitu väärtust kummassegi jadasse paigutatakse. Edasi leitakse piirkonnas ühtlaste vahede järgi väärtused ning joonistatakse ekraanile.

Arvutus minx+i/(double)numbreidx*(maxx-minx) lahti seletatult: maxx-minx on x-i väärtuste vahemik. i on joonistatava väärtuse järjekorranumber. i/numbreidx näitab suhte, kui kaugel väärtuste joonistamisega ollakse. Tüübimuundur (double) väärtuse numbreidx ees hoolitseb, et jagataks reaalarvuliselt. Kuna nii i kui numbreidx on täisarvulised, siis Java antaks vastuseks ka täisarv ehk nende arvude suhte täisosa. i/(double)numbreidx*(maxx-minx) näitab seega x-ide vahemiku läbitud vahemaad ning minx sinna otsa liidetult annab i-nda joonisel oleva arvu väärtuse.

```
void joonistaKoordinaadid(Graphics g){
  int numbreidy=(korgus-2*servaruum)/30;
  int numbreidx=(laius-2*servaruum)/40;
  for(int i=0; i<=numbreidx; i++){
    double x=umarda(minx+i/(double)numbreidx*(maxx-minx), 2);
    g.drawString(x+"", ekraaniX(x), korgus-5 );
  }
  for(int i=0; i<=numbreidy; i++){
    double y=umarda(miny+i/(double)numbreidy*(maxy-miny), 2);
    g.drawString(y+"", 3,  ekraaniY(y));
  }
}
```

Käsklus määratud arvu kümnendkohtadega ümardamiseks, mida Javas vaikimisi kujul sisse ehitatud pole korrutab arvu kõigepealt kümne astmega, mitu kohta tahetakse koma taha jätta. Seejärel ümardatakse round-käsuga täisosani ning edasi jagatakse tulemus arvu 10 vastava astmega.

```
  double umarda(double arv, int kohti){
    return(Math.round(arv*Math.pow(10, kohti))/Math.pow(10, kohti));
  }
```

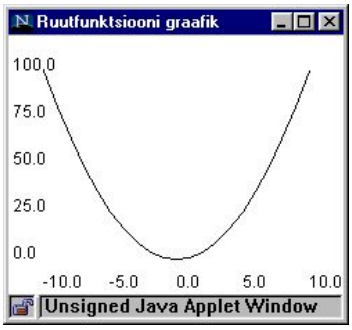
Et veebilehel oleval rakendil kannataks joonistatava ala suurust muuta, on üheks võimaluseks paigutada joonis eraldi aknaraami. Et iseseisva programmi puhul käivitus juba ilusti main-meetodi abil töötab, siis kannatab rakendist see valmisolev programm kogu täiega välja kutsuda. Käsurealt võetavaid ja mainile ette antavaid parameetreid siin kusagilt võtta pole. Et aga main on tavaline alamprogramm ning üldjuhul kannatab muutujale ette anda ka lihtsalt tühiväärtuse, siis kutsutakse põhiprogramm välja käsuga Graafik5.main(null).

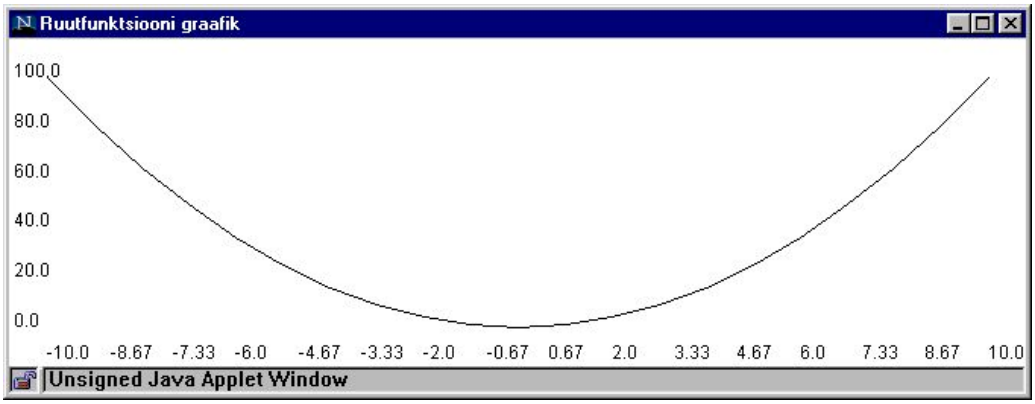
```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class Graafik5Rakend extends Applet{
  Button nupp=new Button("Ava graafik");
  public Graafik5Rakend(){
    setLayout(new BorderLayout());
    add(nupp, BorderLayout.CENTER);
    nupp.addActionListener(new ActionListener(){
     public void actionPerformed(ActionEvent e){
      Graafik5.main(null);
    }});
  }
}
```

Ning lõpuks joonistuslõuendi kood tervikuna.

```
import java.awt.*;
import java.awt.event.*;
public class Graafik5 extends Canvas{
  double minx=-10, maxx=10, samm=(maxx-minx)/20;
  double miny=minY(), maxy=maxY();
  double ulatusmin=0.01, ulatusx, ulatusy;
  int korgus, laius;
  int servaruum=25;
  double koefitsientx, koefitsienty, keskx, kesky;

  public void paint(Graphics g){
    leiaKonstandid();
    joonistaKoordinaadid(g);
    int vanax=ekraaniX(minx);
    int vanay=ekraaniY(f(minx));
    for(double x=minx; x<=maxx; x=x+samm){
      int uusx=ekraaniX(x);
      int uusy=ekraaniY(f(x));
      g.drawLine(vanax, vanay, uusx, uusy);
      vanax=uusx;
      vanay=uusy;
    }
  }

  int ekraaniX(double matemx){
    return laius/2+(int)((matemx-keskx)*koefitsientx);
  }

  int ekraaniY(double matemy){
    return korgus/2+(int)((matemy-kesky)*koefitsienty);
  }

  double minY(){
    double min=f(minx);
```

```
      for(double x=minx; x<=maxx; x+=samm){
        if(f(x)<min)min=f(x);
      }
      return min;
    }

    double maxY(){
      double max=f(minx);
      for(double x=minx; x<=maxx; x+=samm){
        if(f(x)>max)max=f(x);
      }
      return max;
    }

    void leiaKonstandid(){
      korgus=getSize().height;
      laius=getSize().width;
      ulatusx=maxx-minx;
      ulatusy=maxy-miny;
      if(ulatusx<ulatusmin)ulatusx=ulatusmin;
      if(ulatusy<ulatusmin)ulatusy=ulatusmin;
      koefitsientx=(laius-2*servaruum)/ulatusx;
      koefitsienty=-(korgus-2*servaruum)/ulatusy;
      keskx=(maxx+minx)/2;
      kesky=(maxy+miny)/2;
    }

    void joonistaKoordinaadid(Graphics g){
      int numbreidy=(korgus-2*servaruum)/30;
      int numbreidx=(laius-2*servaruum)/40;
      for(int i=0; i<=numbreidx; i++){
        double x=umarda(minx+i/(double)numbreidx*(maxx-minx), 2);
        g.drawString(x+"", ekraaniX(x), korgus-5 );
      }
      for(int i=0; i<=numbreidy; i++){
        double y=umarda(miny+i/(double)numbreidy*(maxy-miny), 2);
        g.drawString(y+"", 3,  ekraaniY(y));
      }
    }

    double umarda(double arv, int kohti){
      return(Math.round(arv*Math.pow(10, kohti))/Math.pow(10, kohti));
    }

    double f(double x){
      return x*x;
    }

public static void main(String argumendid[]){
    final Frame f=new Frame("Ruutfunktsiooni graafik");
    f.add(new Graafik5());
    f.setSize(250, 250);
    f.setVisible(true);
    f.addWindowListener(new WindowAdapter(){
      public void windowClosing(WindowEvent e){
        f.setVisible(false);
        System.exit(0);
      }
    });
  }
}
```





Sama graafik laiemaks venitatuna

Ülesandeid

- Joonista kuupfunktsiooni graafik
- Võimalda kasutajal määrata kordajaid
- Viiruta graafiku joone ja x-telje vaheline ala.

Graafikakomponendid

Graafikakomponendid aitavad programmeerijal hõlbustada programmi ja kasutaja suhtlemist. Samad võimalused saab luua ka joonistamisvahendite abil, kuid varem loodud komponentide puhul peab kasutaja nende käsitsemist õppima vaid korra, kasutada mõistab aga igal pool kus nad ette tulevad. Samuti piisab programmeerijal vaid paigutada komponent sobivale kohale, muuta tema omadusi vastavalt programmi vajadustele ning paluda programmil reageerida kasutaja tegevusele vastavalt. Komponendi sisemise ehituse üle on vaja pead murda vaid siis, kui soovida tema võimalustele midagi lisada, ise uut komponenti luua või olemasoleva vigu parandada.

Valmiskomponendid

Java keeles saab kasutada ligikaudu kümmet operatsioonisüsteemi juurde kuuluvat ning ligi seitsetkümmet java oma vahenditega loodud valmiskomponenti. Vajaduse korral aga saab neid täiendada ning ise juurde kombineerida ja luua. Operatsioonisüsteemi poolt pakutavad komponendid asuvad paketis `java.awt`, nad töötavad enamasti kiiremini ehk nõuavad arvutilt vähem ressursse. Nad näevad välja vastavalt operatsioonisüsteemile ning nende järgi vaadates ei pruugi väljagi paista, et rakendus on Java abil kirjutatud. Kui aga soovitakse (või on vajalik) et programmid erinevates keskkondades sarnaselt välja näeksid, siis tuleb kasutada "puhtaid" java komponente, mille värvus ning kuju operatsioonisüsteemist ei sõltu. Sellised valmiskomponendid asuvad enamjaolt klassis `javax.swing`.

Komponendid (nagu muudki objektid) saavad vastu võtta ning välja saata teateid. Saatja käivitab vastuvõtja meetodi. Komponendile teate saatmisel käivitatakse komponendi meetod (näiteks värvi muutmiseks). Teate saatmisel aga käivitab komponent (kui saatja) vastuvõtja meetodi. Näiteks kui nupule vajutamisel muutub tekst tekstikastis suuremaks, siis öeldakse, et nupp saatis tekstikastile teate. Sisuliselt tähendab see, et nupule vajutamise tulemusena käivitatakse tekstikasti meetod, millega saab muuta fondi suurust. Et nupul oleks võimalik tekstikasti meetodit käivitada, peab nupule olema antud osuti tekstikastile ning öeldud, et juhul kui nupule vajutatakse, tuleb tekstivälja vastav meetod käima panna.

Aken

Iseseisvalt saab ekraanil avada akent (Window) või raami (`Frame`, raamaken). Java keeles tähistab aken lihtsalt ekraanipiirkonda mille kasutamist saab programm juhtida, raami puhul on sel piirkonnal ümber raam ning üleval nupud suurusemuutmiseks ja sulgemiseks ning pealkirjariba. Enamasti kasutatakse kasutaja mugavuse huvides programmides raami, kuid näiteks täisekraaniefektide loomiseks on akna kasutamine paratamatu. Nende suuruse määramisel saab arvestada ekraani suurust. Dialog on eriline raam, mille avamisel võib jätta programmi töö seniks pooleli, kuni kasutaja on vastuse

sisestanud. Sulgemisnupule vajutamine ei sule raami automaatselt. Vajutuse juurde on võimalik siduda programmilõik, kus kirjeldatakse programmi vajutusjärgset käitumist. Seal saab näiteks küsida, et "kas soovite salvestada" või muud taolist.

Tekstiväli

Tekstiväli (TextField) ning tekstiala (TextArea) võimaldavad ekraanile näidata ning kasutajal sisestada teksti. Esimesel neist võib olla vaid üks rida, tekstiala puhul aga saab määrata, mitut rida talle tahetakse. Värvi ning šrifti on võimalik muuta vaid kogu teksti juures korraga, keerulisemat kujundust nõudvate tekstide puhul tuleks kasutada `swing`i paketti kuuluvat `JEditorPane`. Redigeerimise jaoks aga lihttekstikomponentidest piisab. Lisaks komponendis oleva teksti küsimisele ja määramisele saab eraldi küsida ja määrata märgistatud teksti, samuti kursori asukohta.

Valik

Nimekiri (`List`) ning valik (`Choice`) lubavad kasutajal valida etteantud võimaluste seast. Nimekirjas on samaaegselt näha mitu rida. Kasutajal võib lasta märgistada samaaegselt ka mitu rida. Valiku puhul tuleb rippmenüü lahti vaid valimise ajaks, ülejäänud ajal on näha vaid üks, parasjagu valitud rida. Andmed ridadel on sõnedena, muul otstarbel kasutades tuleb neid vastavalt intepreteerida. Et saaks valida näiteks pilte või värve, tuleb kasutada swingi komponente või siis vastav komponent ise kokku panna. Meetodite abil saab nendel komponentidel lisada ja eemaldada ridu, küsida, millise numbri või millise sisuga rea on kasutaja märgistanud. Saab lasta ka automaatselt rida märgistada. Veidi sarnane on ka `swing`i komponent `JTree`, kus kasutaja saab samuti ridu märgistada, seal aga on andmed paigutatud hierarhiliselt, puuna.


```
import java.applet.Applet;
import java.awt.List;
public class AwtList1 extends Applet{
  List list1=new List();
  public void init(){
    list1.add("Sinine");
    list1.add("Punane");
    list1.add("Kollane");
    list1.add("Valge");
    list1.add("Roheline");
    add(list1);
  }
}
```

```
import java.applet.Applet;
import java.awt.Choice;
public class AwtChoice1 extends Applet{
  Choice valik=new Choice();
  public void init(){
    valik.add("Sinine");
    valik.add("Punane");
    valik.add("Kollane");
    add(valik);
  }
}
```

Silt (`Label`) võimaldab endasse paigutada ühe rea teksti. Swingi analoogile saab panna ridu mitu ning soovi korral ka pilte sisse. Nupp (`Button`) reageerib hiirevajutusele. Ka temale on võimalik `awt`-variandis panna üks rida teksti, `swing`i juures aga enam kujundada.

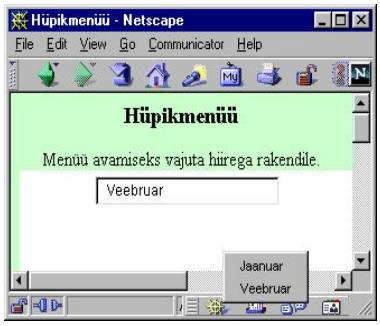
Märkeruudud

Märkeruut (`Checkbox`) laseb kasutajal valida kahe võimaluse vahel (märgitud või mitte). Raadionupp saab samuti olla kas sees või väljas, kuid neid kasutatakse enamasti juhul, kui saab valida vaid ühte mitme võimaluse seast. Selle eest hoolitsemiseks on vaja luua märkeruudugrupp (`CheckBoxGroup`), kes hoolitseb, et sinna gruppi lisatud nuppudest oleks vaid üks sisse lülitatud.


```
import java.awt.*;
import java.applet.Applet;
public class Raadionupud extends Applet{
  public Raadionupud(){
    CheckboxGroup grupp1=new CheckboxGroup();
    add(new Checkbox("Esimene", grupp1, false));
    add(new Checkbox("Teine",   grupp1, true));
    add(new Checkbox("Kolmas",  grupp1, false));
  }
  public static void main(String argumendid[]){
    Frame f=new Frame("Raadionupud");
    f.add(new Raadionupud());
    f.setSize(200, 200);
    f.setVisible(true);
  }
}
```

Menüü

Menüüriba (`MenuBar`) annab kinnitada raami külge. Temast saab panna hargnema menüüd ning neist omakorda alammenüüd. Hüpikmenüü (`Popup`) võib panna välja hüppama mis tahes koha pealt.




```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class Hypikmenyy extends Applet implements
ActionListener{
  TextField ta=new TextField(20);
  PopupMenu pm=new PopupMenu("Kuud");
  public Hypikmenyy(){
    MenuItem mi=new MenuItem("Jaanuar");
    mi.addActionListener(this);
    pm.add(mi);
    mi=new MenuItem("Veebruar");
    mi.addActionListener(this);
    pm.add(mi);
    add(pm);
    addMouseListener(
      new MouseAdapter(){
       public void mousePressed(MouseEvent e){
         pm.show(Hypikmenyy.this, e.getX(), e.getY());
       }
      }
    );
    add(ta);
  }
  public void actionPerformed(ActionEvent e){
    ta.setText(((MenuItem)e.getSource()).
                                    getLabel()+"");
  }
}
```

Joonistamise jaoks on loodud lõuend (`Canvas`), kuid vajaduse korral saab ka teistele komponentidele (näiteks rakendile) joonistada. Lõuendi abil saame suhteliselt kergesti omale soovitud komponendi luua. Näiteks kui soovime pildiga nuppu, mille pilt vajutamise ajal muutuks, siis tuleks

luua lõuendi alamklass. Sinna saab kirjeldada, millist pilti näidata nupu üleval oleku ajal ning millist siis, kui hiirega tema peale vajutatakse.

Kerimispaneel

Kui komponent on suurem kui tema jaoks eraldatud ekraanipind, siis saab ta paigutada `ScrollPane` sisse, mille tulemusena saab kerimisribade abil komponenti liigutada, vaadates teda läbi tema jaoks loodud "akna".


```
import java.awt.*;
import java.applet.Applet;
public class Paigutus10 extends Applet{
  public Paigutus10(){
    setLayout(new BorderLayout());
    Panel nupupaneel=new Panel(new GridLayout(10, 10));
    for(int i=0; i<10; i++){
      for(int j=0; j<10; j++){
        nupupaneel.add(new Button("Nupp "+i+""+j));
      }
    }
    ScrollPane sp=new ScrollPane();
    sp.add(nupupaneel);
    add(sp, BorderLayout.NORTH);
  }
  public static void main(String argumendid[]){
    Frame f=new Frame("Paigutus");
    f.add(new Paigutus10());
    f.setSize(400, 200);
    f.setVisible(true);
  }
}
```

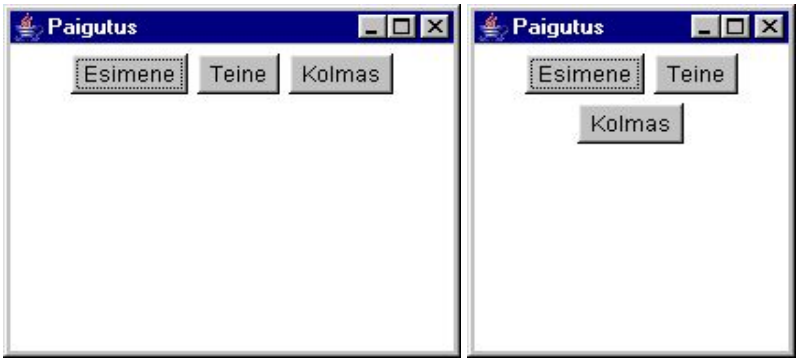
Paigutushaldurid

Graafikakomponent tuleb ekraanil näitamiseks paigutada konteinerisse (nt. raam, paneel, rakend). Konteineri sees komponente paigutada aitavad paigutushaldurid. Nad hoolitsevad, et näiteks raami suuruse muutmisel komponendid ekraanil mõistlikus suuruses näha jääksid. Kuna iga konteiner on samaaegselt ka komponent (ehk tema alamklass), siis saab ka konteinereid endid paigutushaldurite abil paigutada. Niimoodi paneele (või muid konteinereid) sobivalt üksteise sisse paigutades on võimalik saavutada peaaegu igasugune soovitud tulemus. Võimalik on ka täpselt ekraanipunktide järgi komponentide paigutus määrata, kuid see pole soovitav, sest näiteks raami suuruse või ekraani resolutsiooni muutmisel või uue komponendi lisamisel tuleks kogu kujundus uuesti kirjutada.

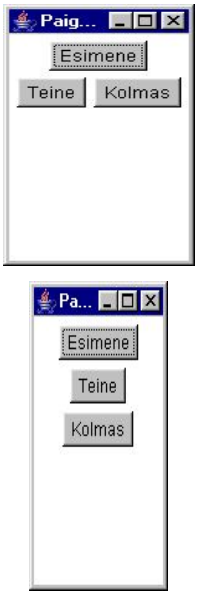
FlowLayout

Katsetamise ajal on lihtsaimaks paigustushalduriks `FlowLayout`. Seal pannakse komponendid üksteise järgi ritta ning kui rida täis saab, siis minnakse ekraanil järgmisesse ritta. Klassidel`Applet` ja `Panel` näiteks ongi `FlowLayout` vaikimisi paigutushalduriks. Sedasi ei pea paigutades arvestama komponetide suureustega. Haldur arvestab ise, et iga element nähtavale jääks, kui ekraanil vähegi ruumi

on. Samas – vähegi keerukama kujunduse puhul ei saa siiski ainuüksi FlowLayouti oskustele lootma jääda.



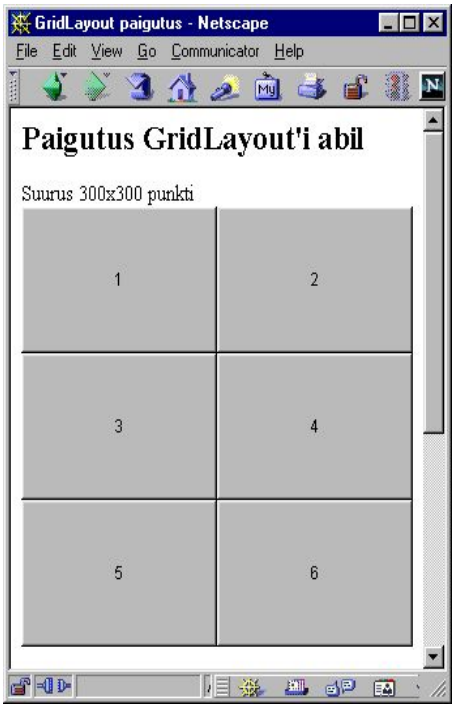

```
import java.awt.*;
import java.applet.Applet;
public class Lihtpaigutus extends Applet{
  Button nupp1=new Button("Esimene");
  Button nupp2=new Button("Teine");
  Button nupp3=new Button("Kolmas");
  public Lihtpaigutus(){
    add(nupp1);
    add(nupp2);
    add(nupp3);
  }
  public static void main(String argumendid[]){
    Frame f=new Frame("Paigutus");
    f.add(new Lihtpaigutus());
    f.setSize(250, 200);
    f.setVisible(true);
  }
}
```



FlowLayout paigutuse näited.

GridLayout

`GridLayout` paigutuse puhul jagatakse konteineri juurde kuuluv piirkond ridadeks ja veergudeks, andes igale komponendile ühe lahtri. `GridBagLayout` on sarnane, kuid seal võib üks komponent katta ka mitu lahtrit. Swingi `BoxLayout` lubab komponendid panna kas ridadena või tulpadena, jättes nad loomulikku suurusse. GridLayouti on hea kasutada, kui on vajalik joondada komponendid tabelisse või muul puhul ühesuguse laiuse ja kõrgusega osadeks.

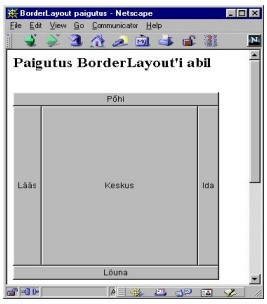



```
import java.awt.*;
import java.applet.Applet;

public class Paigutus2 extends Applet{
  public void init(){
    setLayout(new GridLayout(3, 2));
    add(new Button(" 1 "));
    add(new Button(" 2 "));
    add(new Button(" 3 "));
    add(new Button(" 4 "));
    add(new Button(" 5 "));
    add(new Button(" 6 "));
  }
}
```

BorderLayout

`BorderLayout` lubab komponendid paigutada nelja serva ning keskele. Servas olevad komponendid jäetakse servaga risti olevat mõõdet pidi nende loomulikku suurusse, keskele pandud komponent venitatakse kogu ülejäänud pinna ulatuses välja. Sugugi ei pea alati kõiki viit võimalust kasutama – piisab kui ühte serva on vaja panna nii, et komponent peaks terve servaala enda alla võtma ning muidu loomuliku suurusega välja paistma.

```
import java.awt.*;
import java.applet.Applet;

public class Paigutus1 extends Applet{
  public void init(){
    setLayout(new BorderLayout());
    add(new Button("Põhi"), BorderLayout.NORTH);
    add(new Button("Lõuna"), BorderLayout.SOUTH);
    add(new Button("Ida"), BorderLayout.EAST);
    add(new Button("Lääs"), BorderLayout.WEST);
    add(new Button("Keskus"),BorderLayout.CENTER);
  }
}
```



Paneel paigutamisel.

Paneel aitab kasutada oleva nelinurkse ala osadeks jaotada. Samuti nagu võib rakendile või aknale määrata paigutushalduri ning selle abil vastava konteineri sisse komponendid paigutada, saab nii olemasoleva ala jaotada paneeli abil komponetide vahel.

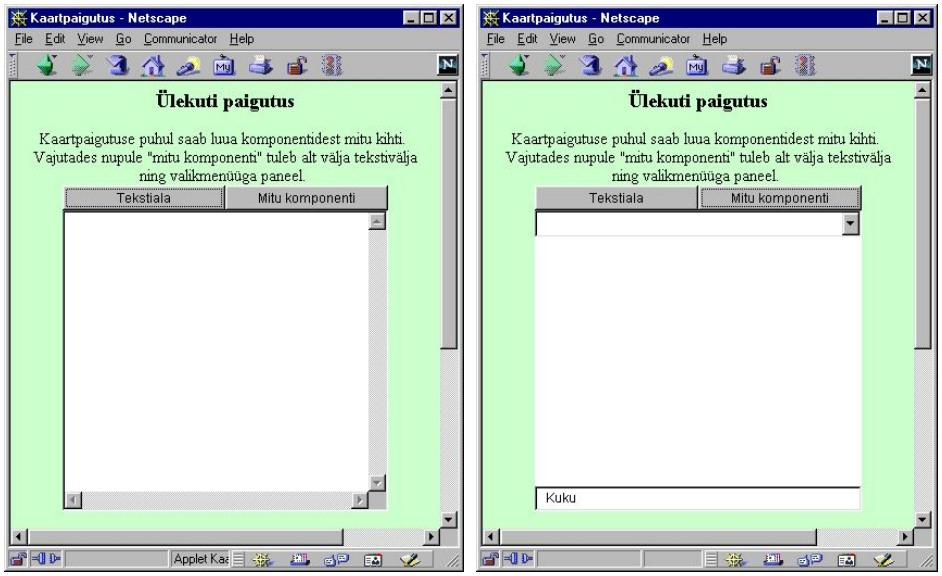
Järgnevas näites määratakse rakendi paigutushalduriks BorderLayout, mis lubab nagu ikka paigutada nii servadesse kui keskele. Kasutatakse vaid ülaserva, mis jagatakse GrigLayout-paigutusega paneeli abil üheks reaks ja kaheks veeruks ning siis lisatakse paneeli mõlemasse pessa nupp. Lõpuks pannakse paneel rakendi ülaserva.


```
import java.applet.Applet;
import java.awt.*;

public class Paneelpaigutus extends Applet{
  Button nupp1=new Button("Esimene");
  Button nupp2=new Button("Teine");
  public Paneelpaigutus(){
    setLayout(new BorderLayout());
    Panel p=new Panel(new GridLayout(1, 2));
    p.add(nupp1);
    p.add(nupp2);
    add(p, BorderLayout.NORTH);
  }
  public static void main(String[] argumendid){
    Frame f=new Frame();
    f.add(new Paneelpaigutus());
    f.setSize(200, 200);
    f.setVisible(true);
  }
}
```

Üldjuhul õnnestub paneele, BorderLayout'i ja GridLayout'i kombineerides kokku panna pea kõik võimalikud paigutusolukorrad, mis traditsioonilise „viisaka“ kujundusega rakenduse puhul ette tulevad.

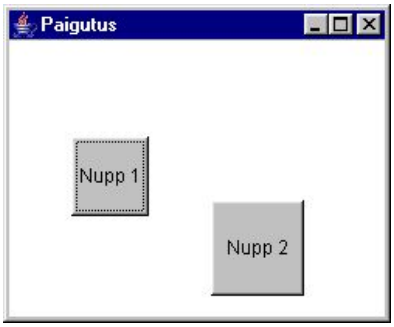
CardLayout

`CardLayout` võimaldab panna mitu kihti komponente üksteise peale, näidates välja pealmise kihi ning lubades kihte vahetada. Nõnda võib samal kohal vaadata kordamööda selliseid elemente nagu programmi looja parajasti tarvilikuks on pidanud. Swingi JTabbedPane eesmärk on sarnane, kuid seal on kohe automaatselt juurde lisatud võimalus kasutajal soovitud kihti välja kutsuda.



Absoluutsete koordinaatidega paigutus.

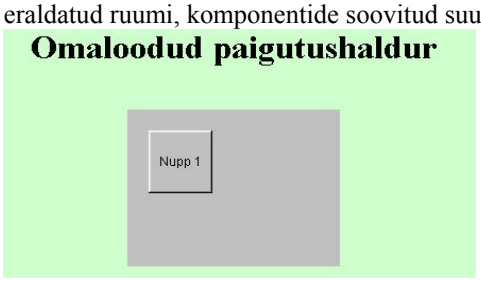
Pea alati tuleb esimese graafikakomponentidega tegelemise tunni jooksul kelleltki küsimus, et „kuidas ma saan täpselt määrata tekstiala/nupu koordinaadid“. On ju nii Visual Basicus, Multimedia Toolbook'is kui mõnes muuski keeles võimalik ekraanipunktide või muude numbrite abil määrata, kus miski komponent asub. Ning seletus, et „Java programmide juures peetakse sellist paigutust ebaviisakaks“ ei tundu kuigi usutavana. Võimalus on täiesti olemas, nii nagu järgmisest näitest paista võib. Koordinaatide järgi paigutatakse siis, kui paigutushaldur puudub, ehk selleks on seatud tühi osuti null. Enne konteineri sisse paigutamist määratakse komponentidele suurused ning siis lisamisel nad satuvadki määratud kohtadesse.




```
import java.awt.*;
import java.applet.Applet;
public class Paigutus9 extends Applet{
  public Paigutus9(){
    setLayout(null);
    Button nupp1=new Button("Nupp 1");
    Button nupp2=new Button("Nupp 2");
    nupp1.setBounds(40, 60, 50, 50);
    nupp2.setBounds(130, 100, 60, 60);
    add(nupp1);
    add(nupp2);
  }
  public static void main(String argumendid[]){
    Frame f=new Frame("Paigutus");
    f.add(new Paigutus9());
    f.setSize(250, 200);
    f.setVisible(true);
  }
}
```

Omaloodud paigutushaldur

Kasutatud haldurid pole muud kui tavalised vastavate oskustega Java klassid. Kui mingil põhjusel selgub, et soovitakse täiesti erilist paigutust, siis võib sellise omale kirjutada. Tuleb lihtsalt koostada klassile java.awt.LayoutManager oma alamklass ning seal mõned meetodid üle katta. Tähtsam neist layoutContainer, mille sees igale konteineris asuvale komponendile määrata tema asukoht. Siin on piirdutud lihtsaima näitega, kus eeldatakse et tegemist on vaid ühe komponendiga ning sellelegi määratakse alati samad koordinaadid. Põhjalikuma paigutamise puhul aga tuleb arvestada konteinerile eraldatud ruumi, komponentide soovitud suurus ning muudki võimaluste järgi.




```
import java.awt.*;
import java.applet.Applet;
public class Paigutus11 extends Applet{
  public Paigutus11(){
    setLayout(new Ruutpaigutus());
    add(new Button("Nupp 1"));
  }
  public static void main(String argumendid[]){
    Frame f=new Frame("Paigutus");
    f.add(new Paigutus11());
    f.setSize(200, 200);
    f.setVisible(true);
  }
}
```

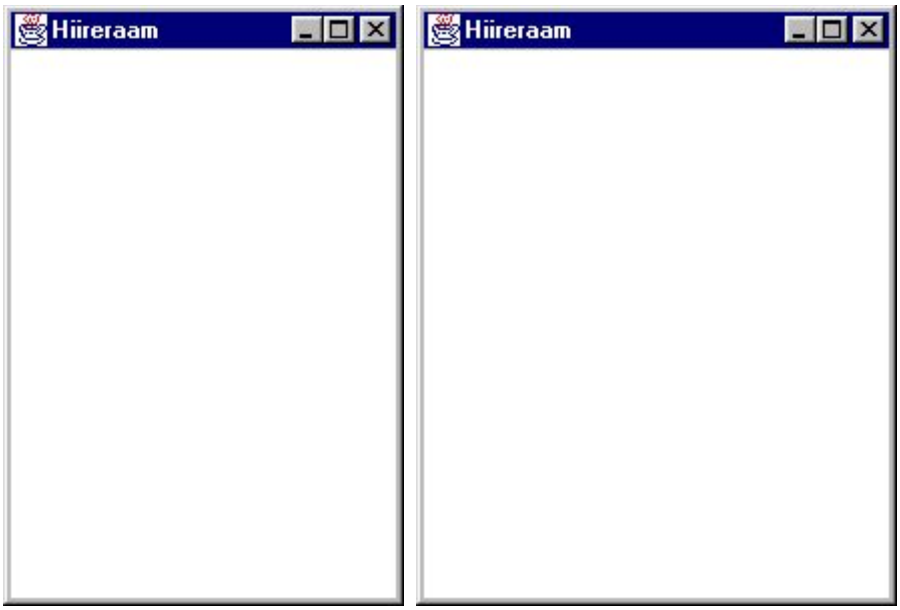
```
import java.awt.*;
class Ruutpaigutus implements LayoutManager{
  public void layoutContainer(Container kest){
    if(kest.getComponentCount()>0){
      Component c=kest.getComponent(0);
      c.setBounds(20, 20, 60, 60);
    }
  }
  public void addLayoutComponent(String nimi, Component c){}
  public void removeLayoutComponent(Component c){}
  public Dimension preferredLayoutSize(Container kest){
    return new Dimension(100, 100);
  }
  public Dimension minimumLayoutSize(Container kest){
    return preferredLayoutSize(kest);
  }
}
```

Kuularid

Sündmuste tulemusena millegi juhtumiseks tuleb sündmuse registreerijalt (ehk allikalt) saata teade sündmuse vastuvõtjale. Ühe sündmuse (näiteks nupuvajutuse) teadet saab saata mitmele vastuvõtjale, muuhulgas ka iseendale. Et allikas vastuvõtjale teate saadaks, selleks peab olema vastuvõtja end allika juures registreerinud vastavat tüüpi sündmuse kuulajaks. Kuulajaks saab end registreerida isend, kelle klass realiseerib vastava sündmusetüübi kuulamiseks loodud liidest.

Näiteks hiirevajutuse kuulamiseks peab klass realiseerima liidest MouseListener. Liidese realiseerimine aga tähendab seda, et klassis oleks kirjeldatud kõigi liideses deklareeritud meetodite käivitamisel tehtavad tegevused. Näiteks MouseListeneri realiseerijal peavad olema kõik liideses kirjeldatud meetodid: nii hiire vajutamise, ülestõstmise, komponendi piirkonda sisenemise kui komponendist väljumise kohta. Kui soovitakse, et mõnel juhul ei reageeritaks, siis tuleb seegi arvutile selgeks teha, s.t. vastava meetodi sisuks kirjutada tühjad sulud.

```
import java.awt.*;
import java.awt.event.*;
public class Hiir1{
  public static void main(String argumendid[]){
    Frame f=new Frame("Hiireraam");
    f.setSize(200, 300);
    f.setVisible(true);
    f.addMouseListener(new HiireKuular1(f));
  }
}

class HiireKuular1 implements MouseListener{
  Frame raam;
  public HiireKuular1(Frame uusraam){
    raam=uusraam;
  }
  public void mousePressed(MouseEvent e){
    //suurendab raami laiust 10 ühiku võrra
    Dimension suurus=raam.getSize();
    raam.setSize(suurus.width+10, suurus.height);
  }
  public void mouseReleased(MouseEvent e){}
  public void mouseClicked(MouseEvent e){}
  public void mouseEntered(MouseEvent e){}
  public void mouseExited(MouseEvent e){}
}
```



Et tühje sulge peaks vähem kirjutama, selleks on loodud adapterklassid liideste jaoks, mis defineerivad enam kui ühe meetodi. Vastava liidese adapterklass realiseerib liidest, jättes meetodi kehaks tühjad sulud, s.t. tehes meetodi käivitamisel mitte midagi. Kui me nüüd oma kuulari loome vastava adapteri alamklassina, siis piisab meil vaid üle katta meile vajalik meetod. Näiteks oma hiirekuulari loomisel katame üle vaid hiire vajutamisel käivitatava meetodi. Muude hiiresündmuste puhul kasutatakse adapterklassi meetodeid ning kuna seal midagi teha ei paluta, siis meile näib, nagu ei reageeritakski nendele sündmustele.

```
import java.awt.*;
import java.awt.event.*;
public class Hiir2{
  public static void main(String argumendid[]){
    Frame f=new Frame("Hiireraam");
    f.setSize(200, 300);
```

```
    f.setVisible(true);
    f.addMouseListener(new HiireKuular2(f));
  }
}

class HiireKuular2 extends MouseAdapter{
  Frame raam;
  public HiireKuular2(Frame uusraam){
     raam=uusraam;
  }
  public void mousePressed(MouseEvent e){
    Dimension suurus=raam.getSize();
    raam.setSize(suurus.width+10, suurus.height);
  }
}
```

Komponent võib ka ise saata teateid enese juures juhtunud sündmustest ning neid siis töödelda. Sel juhul pole eraldi klassi (ega isendit) kuulari jaoks vaja luua. Piisab vaid, kui komponendi alamklass ise realiseerib vastava sündmuse kuulamiseks vajalikku liidest. Liidese realiseerimiseks peab aga meetodi keha olema kõikidel liideses kirjeldatud meetoditel. Sellest siin need tühjad meetodid, et programm teaks, et näiteks hiire sisenemise korral ei tule tal midagi teha.

```
import java.awt.*;
import java.awt.event.*;
public class Hiir3 extends Frame implements MouseListener{
   public Hiir3(){
     addMouseListener(this);
   }
   public void mousePressed(MouseEvent e){
     Dimension suurus=getSize();
     setSize(suurus.width+10, suurus.height);
   }
   public void mouseReleased(MouseEvent e){}
   public void mouseClicked(MouseEvent e){}
   public void mouseEntered(MouseEvent e){}
   public void mouseExited(MouseEvent e){}

   public static void main(String argumendid[]){
     Frame f=new Hiir3();
     f.setSize(200, 300);
     f.setVisible(true);
   }
}
```

Kui aga ka komponendi alamklassis tahetakse kasutada adapteri võimalusi ning leitakse, et tühjade meetodite kirjutamine on aja ja ruumi raiskamine, siis tuleb veidi keerulisemalt hakkama saada. Kuna java keeles on võimalik pärida ainult ühelt eellaselt, siis üheaegselt nii Frame kui adapteri klassis paiknevat koodi pole võimalik pärida. Selgem ning kindlam on kirjutada eraldi adapteri alamklass ning ta konstruktorile anda parameetriks osuti komponendile nagu esimestes näidetes. Siis saab selle osuti kaudu komponendi tegevust juhtida. Versioonist 1.1 alates aga loodi võimalus sama probleemi ka lühemalt lahendada. Alljärgnevas näites luuakse adaptrile nimetu alamklass, kus kaetakse üle tema meetod, siis luuakse isend ning pannakse ta teateid kuulama. Meetod windowClosing kutsutakse välja siis, kui kasutaja vajutab raami sulgemise nupule. Selle tulemusena lõpetatakse programmi töö (System.exit).

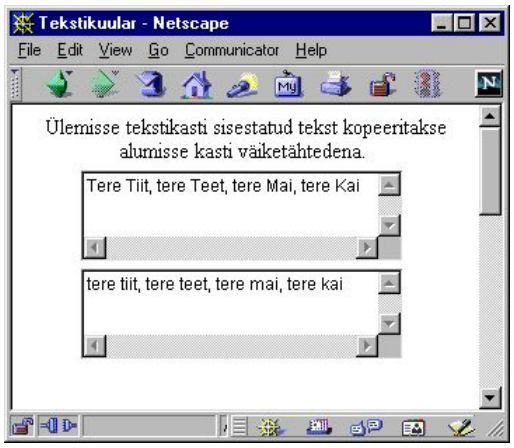
```
import java.awt.*;
import java.awt.event.*;
import java.awt.*;
import java.awt.event.*;
public class Raam4 extends Frame {
   public Raam4(){
     addWindowListener(
       new WindowAdapter(){
        public void windowClosing(WindowEvent e){
          System.exit(0);
        }
       }
     );
   }

   public static void main(String argumendid[]){
     Frame f=new Raam4();
     f.setSize(200, 300);
```

```
      f.setVisible(true);
    }
}
```

Tekstikuular

Tekstikuulari liideses on kirjeldatud vaid üks meetod: textValueChanged. See meetod käivitatakse kuulajatel siis, kui allika (ehk tekstivälja või tekstiala) sees olev tekst on muutunud. Siin näites on rakend ühtlasi esimese tekstiala kuulariks. Kui esimeses tekstialas ehk allikas teksti muudetakse, siis selle tulemusena pannakse teise tekstiala sisuks esimese sisu koopia, kusjuures kõik tähed muudetakse väiketähtedeks.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class Tekstikuular extends Applet
       implements TextListener{
  TextArea tekstiala1=new TextArea(3, 30);
  TextArea tekstiala2=new TextArea(3, 30);
  public Tekstikuular(){
    add(tekstiala1);
    add(tekstiala2);
    tekstiala1.addTextListener(this);
  }
  public void textValueChanged(TextEvent e){
    tekstiala2.setText(tekstiala1.getText().toLowerCase());
  }
}
```

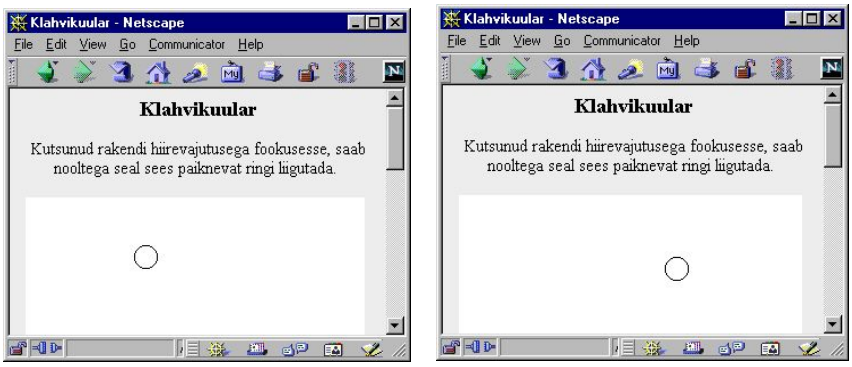


Klahvikuular

Klahvikuular peab realiseerima liidest KeyListener ning temale saadetakse teated keyPressed, keyReleased ning keyTyped. Siin näites pööratakse tähelepanu vaid allavajutamise juhule. KeyEvent isendi meetod getKeyCode annab tulemuseks täisarvu, mis vastab klahvi koodile. Siis kontrollitakse, millise klahviga on tegemist ning toimitakse vastavalt sellele. Konstant KeyEvent.VK_LEFT tähendab näiteks noolt vasakule. Käsklus repaint() keyPressed meetodi lõpus käsib ekraani üle joonistada, s.t. kustutab vaja joonise ning käivitab siis eraldi lõimena meetodi paint. Sellise toimimise korral on ring õiges kohas ekraanil ka näiteks pärast rakendi nihutamist või teiste programmide alla sattumist.

```
import java.applet.Applet;
import java.awt.event.*;
import java.awt.*;

public class Klahvikuular2 extends Applet implements KeyListener{
  int x=100, y=100;
  public Klahvikuular2(){
    addKeyListener(this);
  }
  public void paint(Graphics g){
    g.drawOval(x-10, y-10, 20, 20);
```

```
  }
  public void keyPressed(KeyEvent e){
    int kood=e.getKeyCode();
    if(kood==KeyEvent.VK_LEFT)x--;
    if(kood==KeyEvent.VK_RIGHT)x++;
    if(kood==KeyEvent.VK_UP)y--;
    if(kood==KeyEvent.VK_DOWN)y++;
    repaint();
  }

  public void keyReleased(KeyEvent e){}

  public void keyTyped(KeyEvent e){}


  public static void main(String argumendid[]){
    Frame f=new Frame("Klahvikuular");
    f.add(new Klahvikuular2());
    f.setSize(300, 300);
    f.setVisible(true);
  }

}
```

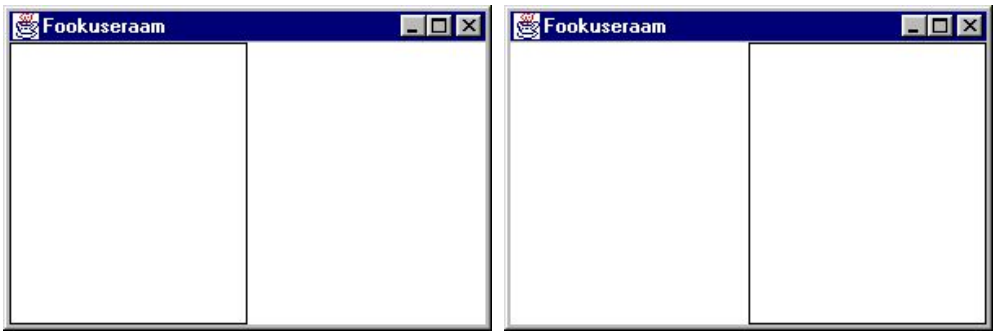


Fookusekuular

Fookusekuular registreerib fookuse saabumise ning eemaldumise teated, s.t. nende sündmuste toimumisel käivitatakse vastavad meetodid. Siin näites lihtsalt omistatakse tõeväärtusmuutujale väärtus tõene või väär vastavalt sellele, kas rakend on fookuses või mitte. Seejärel palutakse ekraan üle joonistada. Ülejoonistamine sõltub muutuja väärtusest. Kui komponent on fookuses, siis joonistatakse esiplaanivärviga rakend üle. Kui aga komponent fookuses pole, siis jäetakse pind värvimata.

```
import java.applet.Applet;
import java.awt.event.*;
import java.awt.Graphics;
public class Fookusekuular extends Applet implements FocusListener{
  boolean fookuses=true;
  public Fookusekuular(){
    addFocusListener(this);
  }

  public void paint(Graphics g){
    if(fookuses){
      g.drawRect(0, 0, getSize().width-1, getSize().height-1);
    }
  }
  public void focusGained(FocusEvent e){
    fookuses=true;
    repaint();
  }
  public void focusLost(FocusEvent e){
    fookuses=false;
    repaint();
  }

  public static void main(String[] argumendid){
    Frame f=new Frame("Fookuseraam");
    f.add(new Fookusekuular());
```

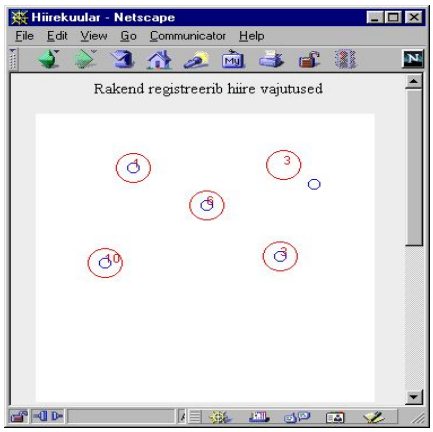
```
    f.setSize(200, 200);
    f.setVisible(true);
  }
}
```



Hiirekuular

Hiire vajutuste ning hiire liikumise registreerimiseks on kummagi jaoks omaette kuular. Nii on kasulik seetõttu, et hiire liigutamisel tuleb teateid tunduvalt rohkem kui vajutamisel. Liigutamisel tuleb registreerida piisavalt palju punkte, et nende abil oleks võimalik kõverjoonelist liikumisteed ette kujutada. Järgnevas näites joonistatakse hiire vajutamise kohale punane ring, ülestõstmise kohale sinine ning kuid hiir väljub komponendi piirkonnast, siis joonistatakse viimase pind valge värviga üle. Vajutamise juures kirjutatakse ka, mitu korda selles punktis on hiireklahvi lühikeste vahedega vajutatud. Andmed selle kohta saab hiirevajutussündmuse puhul väljakutsutava meetodi MouseEvent tüüpi parameetrist.

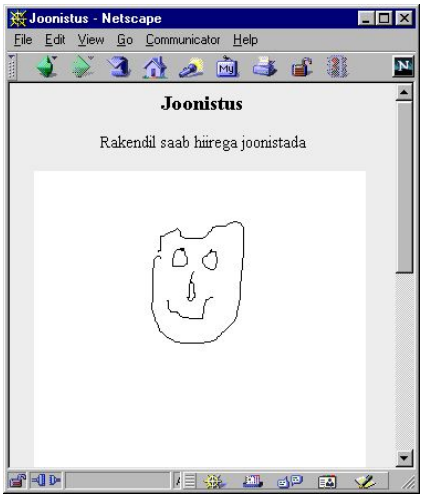
```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class Hiirekuular extends Applet implements MouseListener{
  public Hiirekuular(){
    addMouseListener(this);
  }
  public void mousePressed(MouseEvent e){
    Graphics g=getGraphics();
    g.setColor(Color.white);
    g.fillRect(e.getX()-15, e.getY()-15, 30, 30);
    g.setColor(Color.red);
    g.drawOval(e.getX()-15, e.getY()-15, 30, 30);
    g.drawString(e.getClickCount()+"",e.getX(), e.getY());
  }
  public void mouseReleased(MouseEvent e){
    Graphics g=getGraphics();
    g.setColor(Color.blue);
    g.drawOval(e.getX()-5, e.getY()-5, 10, 10);
  }
  public void mouseExited(MouseEvent e){
    Graphics g=getGraphics();
    g.setColor(Color.white);
    g.fillRect(0, 0, getSize().width, getSize().height);
  }
  public void mouseEntered(MouseEvent e){}
  public void mouseClicked(MouseEvent e){}
}
```



Hiire liikumise kuular

Pildi joonistamiseks on vaja registreerida nii hiire vajutusi kui liikumist. Et pääseda hiirekuulari praeguses programmis mitte vaja minevate sündmuste töötlemisest, selleks loon oma kuulari klassi MouseAdapter alamklassina, mis samaaegselt realiseerib MouseMotionListener liidest (meeldetuletuseks: korraga võib laiendada vaid ühte klassi, kuid liideseid realiseerida kuitahes mitu). Hiire allavajutamisel jäetakse meelde vajutuse asukoht. Liigutamisel aga tõmmatakse joon eelmise punkti ning hetkel hiire asukohaks oleva punkti vahele.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class Joonistus extends Applet{
  public Joonistus(){
    JoonistuseKuular kuular=new JoonistuseKuular(this);
    addMouseListener(kuular);
    addMouseMotionListener(kuular);
  }
}
class JoonistuseKuular extends MouseAdapter
      implements MouseMotionListener{
  Joonistus peremees;
  int vanax, vanay, uusx, uusy;
  public JoonistuseKuular(Joonistus j){
    peremees=j;
  }
  public void mousePressed(MouseEvent e){
    vanax=e.getX();
    vanay=e.getY();
  }
  public void mouseDragged(MouseEvent e){
    uusx=e.getX();
    uusy=e.getY();
    Graphics g=peremees.getGraphics();
    g.drawLine(vanax, vanay, uusx, uusy);
    vanax=uusx;
    vanay=uusy;
  }
  public void mouseMoved(MouseEvent e){}
}
```



Järgnev näide erineb eelmisest selle poolest, et pilt joonistatakse enne mällu ning alles sealt ekraanile. Sellisel juhul saab pildi ka pärast selle ekraanilt kadumist uuesti sinna tekitada. Pilt luuakse mällu paint-meetodi esmakordsel väljakutsel. Meetod update on üle kaetud, et joonistamine ilusamini välja näeks. Vaikimisi update enne joonistab komponendi taustavärviga üle ning alles siis joonistab sinna peale pildi. Kui aga paluda update'l kohe pilt joonistada, siis aitab see vältida vilkumist.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class Joonistus2 extends Applet{
  Image pilt;
  public Joonistus2(){
    Joonistuse2Kuular kuular=new Joonistuse2Kuular(this);
    addMouseListener(kuular);
    addMouseMotionListener(kuular);
  }
  public void paint(Graphics g){
    if(pilt==null)pilt=createImage(getSize().height, getSize().width);
    g.drawImage(pilt, 0, 0, this);
  }
  public void update(Graphics g){
    paint(g);
  }
  public static void main(String[] argumendid){
     Frame f=new Frame("Joonistus");
     f.add(new Joonistus2());
     f.setSize(300, 300);
     f.setVisible(true);
  }
}

class Joonistuse2Kuular extends MouseAdapter
      implements MouseMotionListener{
  Joonistus2 peremees;
  int vanax, vanay, uusx, uusy;
  public Joonistuse2Kuular(Joonistus2 j){
    peremees=j;
  }
  public void mousePressed(MouseEvent e){
    vanax=e.getX();
    vanay=e.getY();
  }
  public void mouseDragged(MouseEvent e){
    uusx=e.getX();
    uusy=e.getY();
    Graphics g=peremees.pilt.getGraphics();
    g.drawLine(vanax, vanay, uusx, uusy);
    vanax=uusx;
    vanay=uusy;
    peremees.repaint();
  }
  public void mouseMoved(MouseEvent e){}
}
```

Graafikakomponendi loomine

Omatehtud jooniste ning komponentide aluseks sobib lõuend (Canvas). Ta sobib lihtsaks joonistamiseks, kuid samas saab ta panna ka teateid vastu võtma, s.t. näiteks hiirevajutusele reageerima. Kui oled kord komponendi loonud, siis saad seda tervikuna kasutada seal kus parajasti vaja on. Kui oled komponendi tööga rahul, siis võid tarvitada teda ilma sisemisse ehitusse süvenemata. Alati ei pea komponendi loomisel kõike otsast tegema, vaid võib kasutada juba varem olemas olevaid tükke. Samuti võib luua varemvalmistatud komponendile alamklassi ning seal soovitud meetodid muuta. Nii võib kerge vaevaga lisada tekstialale võimaluse, et ta väljastaks oma sees oleva ridade arvu või paneks lisatavatele ridadele tühikud ette. Kui aga tahetakse sündmused ja kujundused täiesti ise määrata, siis tuleb aluseks võtta tühi pind ehk lõuend.

Hulknurk

Siin näites joonistatakse lõuendile soovitud nurkade arvuga hulknurk. Nurkade arvu saavad väljapoolsed muuta vaid meetodi abil. Iga muutmisega kaasneb uus joonistamine.

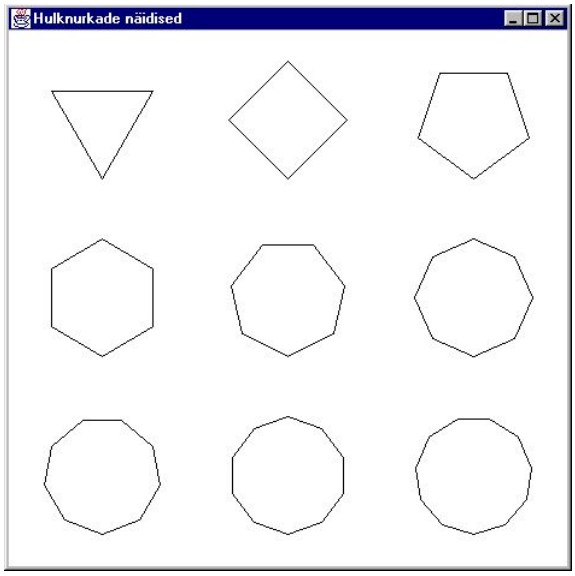
```
import java.awt.*;
public class Nurgad extends Canvas{
  protected int nurkadearv;
  public Nurgad(){
    nurkadearv=3;
  }
  public Nurgad(int uusarv){
```

```
    nurkadearv=uusarv;
  }

  public void muudaNurkadeArv(int uusarv){
    nurkadearv=uusarv;
    repaint();
  }

  public void paint(Graphics g){
    int korgus=getSize().height;
    int laius=getSize().width;
    double nurgavahe=2*Math.PI/(double)nurkadearv;
    int raadius=Math.min(korgus, laius)/3;
    int keskx=laius/2;
    int kesky=korgus/2;
    int vanax=keskx;
    int vanay=kesky+raadius;
    int uusx, uusy;
    for(int i=1; i<=nurkadearv; i++){
      uusx=keskx+(int)(raadius*Math.sin(i*nurgavahe));
      uusy=kesky+(int)(raadius*Math.cos(i*nurgavahe));
      g.drawLine(vanax, vanay, uusx, uusy);
      vanax=uusx;
      vanay=uusy;
    }
  }
}
```

Komponendi kasutamine

Loodud komponenti saab kasutada seal, kus vaja hulknurki joonistada. Siin näites pannakse rakendi ekraanile üheksa hulknurka, nurkade arvuga kolmest üheteistkümneni.

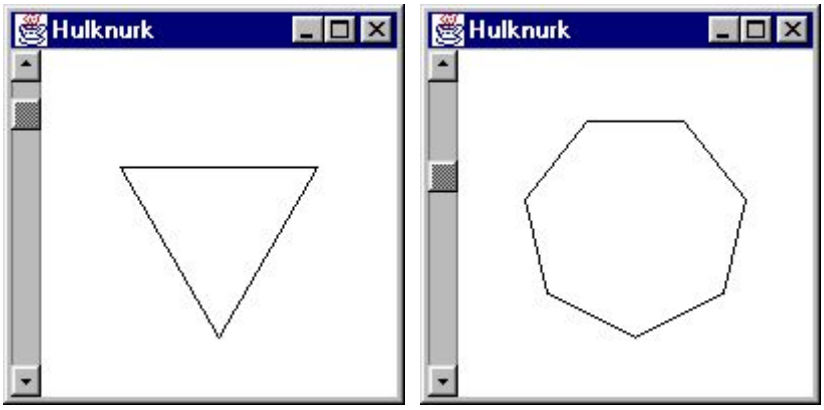
```
import java.applet.Applet;
import java.awt.Frame;
public class Nurgarakend extends Applet{
  public Nurgarakend(){
    setLayout(new java.awt.GridLayout(3, 3));
    for(int nr=3; nr<12; nr++){
      add(new Nurgad(nr));
    }
  }
  public static void main(String[] argumendid){
    Frame f=new Frame("Hulknurkade näidised");
    f.add(new Nurgarakend());
    f.setSize(200, 200);
    f.setVisible(true);
  }
}
```



Samuti saab loodud komponendi abil lasta kasutajal valida, mitme nurgaga hulknurka soovib. Selleks panin rakendile kerimisriba ning loodud komponendi. Rakendi panin kerimisriba kuulajaks

(AdjustmentListener). Kui kerimisriba määratud koha väärtust muudetakse, siis saadetakse uus väärtus rakendile, kes selle omakorda saadab hulknurka joonistavale komponendile.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class Nurgarakend2 extends Applet implements AdjustmentListener{
  Nurgad ng=new Nurgad();
  Scrollbar sb=new Scrollbar(
    Scrollbar.VERTICAL, 3, 2, 2, 20
  );
  public Nurgarakend2(){
    setLayout(new BorderLayout());
    add(sb, BorderLayout.WEST);
    add(ng, BorderLayout.CENTER);
    sb.addAdjustmentListener(this);
  }

  public void adjustmentValueChanged(AdjustmentEvent e){
    ng.muudaNurkadeArv(e.getValue());
  }

  public static void main(String[] argumendid){
    Frame f=new Frame("Hulknurk");
    f.add(new Nurgarakend2());
    f.setSize(200, 200);
    f.setVisible(true);
  }
}
```



Andmete ülekanne

Kopeerimine

Programmide sees ning ka programmide vahel kasutatakse andmete vahetamiseks mälupuhvrit (`clipboard`). Sinna saab andmeid paigutada ning sealt vajadusel kopeerida. Kui puhver on operatsioonisüsteemi juures ning sinna pääsevad ligi mitmed programmid, siis saab selle abil nende vahel andmeid vahetada. Et andmeid sobiksid, peavad osapooled aru saama andmete formaadist. Kõige lihtsamaks formaadiks on lihtne tekst, kuid selle kaudu saab kõike vahetada, mida on võimalik baitideks muundada.

Järgnevast näitest suurem osa kulub kujundusele, kus luuakse raam, pannakse sinna sisse tekstiväli, tekstiala ning menüü. Kopeerimine ja kleepimine asub meetodis actionPerformed, mis käivitub menüüst valiku tegemisel. Vastavalt menüürea nimele käivitatakse tegevus. Meetodi algul küsitakse juurdepääs operatsioonisüsteemi mälupuhvrile.

```
Clipboard malu = getToolkit().getSystemClipboard();
```

Kui antakse korraldus Kopeeri, siis võetakse tekstiväljast tekst ning muudetakse `StringSelection`'iks. Viimatinimetatud klassis on tekst kujul, mida saab mälupuhvrisse panna ning mida teised programmid lugeda mõistavad.

```
StringSelection ss = new StringSelection(tf.getText());
```

Seejärel öeldakse, et mingu see tekst mälupuhvrisse. Meetodi teiseks parameetriks on `ClipboardOwner`, kellele saadetakse teade puhvri sisu vahetumisest. Siin näites tegelikult nende teadetega midagi ette ei võeta.

```
clipboard.setContents(ss, ss);
```

Kleepimise puhul võetakse teade mälupuhvrist välja ning pannakse ta tekstialasse. Puhvrist saadakse andmed kätte esialgu tüübina `Transferable`, mis tuleb seejärel sobivaks tüübiks muundada. Transferable käest on võimalik küsida millisel kujul ta andmeid kannab. Siin aga eeldame, et tegemist on sõnega ning palume tal sellisena need andmed ka välja anda.

```
Transferable andmed = clipboard.getContents(this);
String s = (String)(andmed.getTransferData(DataFlavor.stringFlavor));
```

Tulemuseks on programm, mille abil saab andmeid tekstina programmide vahel vahetada. Selgitust vajab ka ehk menüü loomine. Algul luuakse menüüriba (`MenuBar`), sinna külge pannakse menüü(d) (`Menu`) ning viimasesse menüüread (`MenuItem`). Menüüridadele öeldakse (`addActionListener`), kellele nende peale vajutamisel teateid saata.

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.datatransfer.*;

public class Tekstikopeering extends Frame implements ActionListener {
  TextField tf=new TextField();
  TextArea ta=new TextArea();
  public Tekstikopeering() {
    super("Kopeerimine");
    add(tf, BorderLayout.NORTH);
    add(ta, BorderLayout.CENTER);
    MenuBar mb = new MenuBar();
    mb.add(looMenyy());
    setMenuBar(mb);
  }

Menu looMenyy() {
    Menu m = new Menu("Parandused");
    MenuItem mi = new MenuItem("Lõika");
    mi.addActionListener(this);
    m.add(mi);
    mi = new MenuItem("Kopeeri");
    mi.addActionListener(this);
    m.add(mi);
    mi = new MenuItem("Kleebi");
    mi.addActionListener(this);
    m.add(mi);
    m.addSeparator();
    mi = new MenuItem("Puhasta");
    mi.addActionListener(this);
    m.add(mi);
    return m;
 }
```


```
  public void actionPerformed (ActionEvent e) {
    Clipboard malu = getToolkit().getSystemClipboard();
    String kask = e.getActionCommand();
    if (kask.equals("Kopeeri")) {
       StringSelection data = new StringSelection(tf.getText());
       malu.setContents(data, data);
    } else if (kask.equals("Puhasta")) {
       tf.setText("");
    } else if (kask.equals("Kleebi")) {
       Transferable andmed = malu.getContents(this);
       String s;
       try {
        s = (String)(andmed.getTransferData(DataFlavor.stringFlavor));
       } catch (Exception viga) {
        s = viga.getMessage();
       }
       ta.setText(s);
    } else if (kask.equals("Lõika")) {
       StringSelection ss = new StringSelection(tf.getText());
       malu.setContents(ss, ss);
       tf.setText("");
    }
  }
  public static void main (String argumendid[]) {
    Frame f=new Tekstikopeering();
```

```
    f.setSize(300, 300);
    f.setVisible(true);
  }
}
```

Andmete vedamine (Drag and Drop)

Lisaks mälupuhvri abil kopeerimisele püütakse andmete ülekannet ka hiirega vedamise abil kasutajale intuitiivsemaks muuta. Enamik meist on tõenäoliselt hiirega Word'i redaktoris sõnu lauses ringi tõstnud või Windows Exploreri aknas faile ühest kataloogist teise lohistanud. Andmete allikaks või suudmeks saab määrata ükskõik millise komponendi, kes on võimeline hiire teateid vastu võtma. Komponendile tuleb määrata sündmus, mille peale ta end andmete allikaks loeb. Sageli on selleks näiteks hiire vajutus ning lohisemine tema peal vähemalt viie punkti ulatuses. Kui andmed on kord liikuma pandud, saab nende "käekäigu" üle teateid andmeveo kuulari abil, kes teatab hiire sattumisest võimaliku vastuvõtja alasse.

Andmete vastuvõtjalgi on kuular. Tema saab teateid enesele sattunud andmetega varustatud hiirest ning on võimeline vastavalt nendele teadetele käituma. Ta saab võrrelda pakutavat andmete tüüpi enese poolt vastu võtta suudetavate andmetüüpidega ning sellest kasutajale teadma andma. Kui hiire klahv lastakse vastuvõtja kohal lahti, siis saabub teade drop ning andmed võib vastu võtta.

Siin näites luuakse raam kolme sildiga. Ülemised kaks on andmete allikaks ning nende siltide pealt vedama hakkamisel kaasneb andmetena sildi peal olev kiri. Kolmas silt on vastuvõtja. Kui selle peal vabastatakse andmeid kandva kursoriga hiire klahv, siis jääb saabunud tekst sildi sisse.

Nii allika kui suudme olen loonud sildi alamklassina. AndmeveoAlguseKuularis on kirjas, mida tuleb teha, kui `DragSource` poolt loodud `DefaultDragGestureRecognizer` on märganud, et sildi pealt hakatakse andmeid vedama. Sel puhul võetakse sildi tekst, muutetakse ta `Transferable` tingimustele vastavaks `StringSelection`'iks, et teda saaks üle kanda ning siis käivitatakse vedu käsuga `startDrag`. Parameetriteks on vedamise ajal näidatav kursor, kantavad andmed ning kuular, kellele saadetakse teated andmetega teel toimuva kohta.

Suudmel on isend `DropTarget`, kelle poolt loodud `AndmeteSaabumiseKuular` saabuvate andmetega tegeleb. Kui suudme kohal lastakse lahti andmehulk, saab selle tüüpide sobivuse korral vastu võtta. Esialgu küsitakse meetodi parameetrilt `Transferable`-tüüpi andmed, sealt oodatud kujul `Object`ina ning lõpuks tuleb nad kasutatavale kujule muudada. Siis võib nendega edasi toimida, siin näites andmete sees paiknev tekst oma sildile paigutada.

```
import java.awt.*;
import java.awt.dnd.*;
import java.awt.datatransfer.*;

public class Allikas extends Label {
  private DragSource dragSource;
  public Allikas(String s) {
    setText(s);
    dragSource = DragSource.getDefaultDragSource();
    dragSource.createDefaultDragGestureRecognizer(
      this, DnDConstants.ACTION_COPY,  new AndmeveoAlguseKuular());
    }

  class AndmeveoAlguseKuular implements DragGestureListener {
    public void dragGestureRecognized(DragGestureEvent e) {
      Transferable andmed = new StringSelection( getText() );
        e.startDrag(DragSource.DefaultCopyDrop,
          andmed, new AndmeveoKuular());
    }
  }


  class AndmeveoKuular implements DragSourceListener {
    public void dragDropEnd(DragSourceDropEvent e) { }
    public void dragEnter(DragSourceDragEvent e) {  }
    public void dragOver(DragSourceDragEvent e) {  }
    public void dragExit(DragSourceEvent e) { }
    public void dropActionChanged (DragSourceDragEvent e) { }
  }
}
```

```
import java.awt.*;
import java.awt.dnd.*;
import java.awt.datatransfer.*;
import java.io.*;
public class Suue extends Label {
  private DropTarget dropTarget;
  public Suue(String s) {
    setText(s);
    dropTarget = new DropTarget(this,
      DnDConstants.ACTION_COPY,
      new AndmeteSaabumiseKuular(), true);
  }

  class AndmeteSaabumiseKuular implements DropTargetListener {
    public void dragOver(DropTargetDragEvent e) { }
    public void dropActionChanged(DropTargetDragEvent e) { }
    public void dragExit(DropTargetEvent e) { }
    public void dragEnter(DropTargetDragEvent e) { }
    public void drop(DropTargetDropEvent e) {
     try{
        e.acceptDrop(DnDConstants.ACTION_COPY);
        Object data = e.getTransferable().
                      getTransferData(DataFlavor.stringFlavor);
        setText(data.toString());
     }catch(Exception ex){ ex.printStackTrace(); }
    }
  }
}


import java.awt.*;
public class Vedamine{
  public static void main(String argumendid[]){
    Frame f=new Frame("Andmeveo raam");
    f.setLayout(new GridLayout(3, 1));
    f.add(new Allikas("Karu"));
    f.add(new Allikas("Rebane"));
    f.add(new Suue("Vea siia!"));
    f.setSize(200, 100);
    f.setVisible(true);
  }
}
```

Trükkimine

Lühike näide

Lihtsaks trükkimiseks tuleb luua liidest `Printable` realiseeriv klass, kus meetodis `print` öeldakse, kuidas tuleb trükkida. Printerisse joonistamine käib samuti graafilise konteksti `Graphics` abil nagu ekraanile või mällugi joonistamine. Meetodi `print` parameetrina tuleva `PageFormat`'i abil saab teada, kui suure trükitava lehega tegemist on ning millisele osale lehest on võimalik joonistada. Enamasti on lehe serva määratud vaikimisi selline osa, kuhu joonistada ei saa. Joonistuskõlbuliku osa alguse x-koordinaadi annab `getImageableX`(). Samuti saab `PageFormat`'i käest küsida y-koordinaati ning joonistatava ala kõrgust ja laiust. Siin näites transleeritakse graafilise konteksti nullkoht joonistatava ala algusesse. Kolmanda parameetrina tulev leheküljenumber näitab, millist lehekülge soovitakse trükkida. Isendil on täiesti võimalik mitmele leheküljele trükkida. Lihtsalt tuleb meetodis print igale leheküljenumbrile vastavalt reageerida. Kui vastava numbriga lehekülge ei soovita trükkida, peab meetod tagastama väärtuse `Printable.NO_SUCH_PAGE`, muul juhul `Printable.PAGE_EXISTS`.

Trükkimise käivitamiseks luuakse isend tüübist `PrinterJob`, määratakse, milline `Printable` oskusega isend trükitöö ära teeb ning siis palutakse trükkima hakata.

```
import java.awt.print.*;
import java.awt.*;

public class Trykk1{
  public static void main(String argumendid[])
                      throws PrinterException{
    PrinterJob pj=PrinterJob.getPrinterJob();
    pj.setPrintable(new Trykitoo1());
    pj.print();
  }
}

class Trykitoo1 implements Printable{
  public int print(Graphics g, PageFormat pf, int lk)
                               throws PrinterException{
    if(lk>0) return Printable.NO_SUCH_PAGE;
    g.translate((int)pf.getImageableX(), (int)pf.getImageableY());
    g.drawOval(10, 10, 200, 200);
    return Printable.PAGE_EXISTS;
  }
}
```

Komponendi trükkimine

Kuna nii ekraanile kui printerisse joonistab klassi `Graphics` järglane, siis saab joonistamisel kasutada sama meetodit. Siin näites joonistatakse mõlemasse meetodi `paint` abil. Programm loob ekraanile nupu ning omaloodud komponendi `Kiri2`. Nupule vajutades trükitakse `Kiri2` printerisse. Trükkimine on korraldatud nii, et vajutamise peale trükitakse `Kiri2` tüüpi isendit 2 lehekülge, kummalegi lehele joonistatakse tema kujutis ning lehe alla kirjutatakse lehekülje number. Klassi `PrinterJob` meetod `printDialog` kutsub välja dialoogiakna, kust kasutaja saab määrata printerit ning väljastatavate lehekülgede numbreid ja koopiate arvu.

```
import java.awt.*;
import java.awt.event.*;
import java.awt.print.*;

class Kiri2 extends Canvas implements Printable {

  public void paint(Graphics g) {
    g.setColor(Color.black);
    int W = (int)getSize().getWidth();
    int H = (int)getSize().getHeight();
    g.drawRect(1, 1, W-3, H-3);
    g.drawString("Tere!", W/2, H/2);
  }

  public int print(Graphics g, PageFormat pf, int lk)
                              throws PrinterException {
    if (lk >= 2) {
```

```
      return Printable.NO_SUCH_PAGE;
    }
    g.translate((int)pf.getImageableX(), (int)pf.getImageableY());
    g.setColor(Color.black);
    paint(g);
    g.drawString("lk nr. "+(lk+1), 100,
                              (int)pf.getImageableHeight()-50);
    return Printable.PAGE_EXISTS;
  }
}

public class Trykk2 extends Panel implements ActionListener {
        Kiri2 kiri=new Kiri2();
        Button b = new Button("Tryki");

    public Trykk2() {
        b.addActionListener(this);
        add(b);
        kiri.setSize(100, 50);
        add(kiri);
    }

    public void actionPerformed(ActionEvent e) {
        PrinterJob pj = PrinterJob.getPrinterJob();
        pj.setPrintable(kiri);
        try {
          if(pj.printDialog())
                pj.print();
        } catch (Exception PrintException) { }
    }

    public static void main(String s[]) {
        Frame f = new Frame("Trykkimisraam");
        f.add("Center", new Trykk2());
        f.pack();
        f.setSize(400,300);
        f.show();
    }
}
```

Trükitava ala suuruse muutmine

Trükitava ala suurust lehel saab ka ise muuta. Sel juhul tuleb `PageFormat`'i käest küsida `Paper` tüüpi isend, ja siis sinna määrata soovitud suurusega joonistusala. Edasi määrata PageFormat'ile vastav paber ning paluda `PrinterJob`'il vastava `PageFormat`'i järgi trükkida. Siin näites lubatakse trükkida lehel servast serva.

```
import java.awt.print.*;
import java.awt.*;

public class Trykk3{
  public static void main(String argumendid[])
                        throws PrinterException{
    PrinterJob pj=PrinterJob.getPrinterJob();
    PageFormat pf=pj.defaultPage();
    Paper p=pf.getPaper();
    p.setImageableArea(0, 0, p.getWidth(), p.getHeight());
    pf.setPaper(p);
    pj.setPrintable(new Trykitoo3(), pf);
    pj.print();
  }
}

class Trykitoo3 implements Printable{
  public int print(Graphics g, PageFormat pf, int lk)throws PrinterException{
    if(lk>0) return Printable.NO_SUCH_PAGE;
    g.drawOval(0, 0, 300, 200);
    return Printable.PAGE_EXISTS;
  }
}
```

Trükitava ala suurust saab lasta ka kasutajal dialoogiakna abil määrata. Sellise akna manab ekraanile `pageDialog`.

```
public class Trykk3a{
  public static void main(String argumendid[])
   throws PrinterException{
```

```
    PrinterJob pj=PrinterJob.getPrinterJob();
    pj.setPrintable(new Trykitoo3a(),
            pj.pageDialog(pj.defaultPage()));
    pj.print();
  }
}
```

Lisavõimalused

`Printable` liidese abil saab trükkida ühesuguse suurusega lehekülgi. Kui peaks aga vaja olema ühte trükitavasse dokumenti kokku panna mitmesuguseid (näiteks püsti- ning põikiformaadis) lehti, siis tuleb algul panna kirjutatavatest lehtedest kokku `Book` ning seda trükkima hakata.

Kokkuvõte

Graafikakomponendid aitavad lihtsustada kasutajaga suhtlemist. Kümmekonda awt-paketis olevat komponenti juhitakse operatsioonisüsteemi poolt, nad näevad välja nii nagu vastavas operatsioonisüsteemis tavaks. Mõnikümmend swing-komponenti lisavad võimalusi. Need näevad välja igal pool ühtemoodi, töötavad suhteliselt aeglasemalt, kuid neid on kergem pilkupüüdvaks kujundada.

Komponentidega juhtunud sündmuste töötlemiseks tuleb luua vastava sündmuse kuular ning kuularil lasta end sündmuse allika juures registreerida. Komponent võib ka ise olla enesega juhtunud sündmuste kuulariks. Lisaks kuularitele saab vajadusel ka uusi sündmusi ja komponente ise luua.

Ülesandeid

Paigutamine

- Paiguta BorderLayout-i abiga nupp ülaserva ning tekstiala keskele.
- Jaga ülaserv võrdselt kolme nupu vahel.
- Paiguta allserva ühte ritta valik (choice) ja kerimisriba nii, et valik võtab tema jaoks hädavajaliku ruumi, riba aga ülejäänu.
- Allserva teise ritta lisa märkeruut ning kolm üheskoos töötavat raadionuppu.

Püüdmine

- Hiirega vajutamise kohale joonistatakse ristkülik
- Hiirevajutuse tulemusena hüppab ristkülik suvalisse kohta
- Hiirega ristküliku tabamisel hüppab viimane suvalisse kohta.
- Tekstiväljades loetakse, mitu tabamust on pihta, mitu mööda läinud.
- Kasutajal on võimalik valida, kas tal tuleb püüda ruutu või ringi.

Diagrammikomponent

- Loodavale komponendile joonistatakse etteantud arvu kõrgune tulp.
- Tulpade kõrgused antakse komponendile ette massiiviga.
- Joonistamisel leitakse koefitsiendid nii, et suurima tulba pikkus oleks 80% komponendi kõrgusest.

Graafilise liidesega võrgurakendused

Võrguprotokoll, lõimed, klient, server, kuularid, dokumenteerimine, vektorgraafika

Trips-traps-trull

Kirjeldus

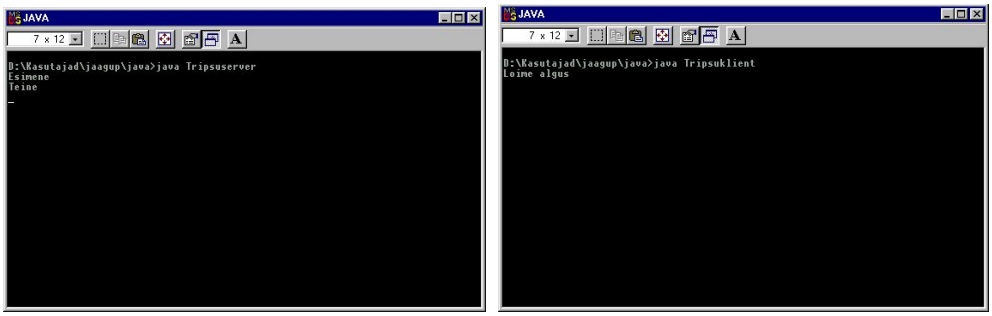
Võrguprogrammi näiteks on siin kokku pandud Trips-traps-trulli mäng. Serveripoole ülesanne on kaks mängijat kohale oodata, siis kordamööda teineteisele käiguõigus anda ning saadetud käsud edasi saata. Samuti on serveripoole otsustada, kumba mängijat tähistatakse nulli ja kumba ristiga. Otsus on tehtud lihtne: esimene saabunud mängija saab märgiks X, teine 0. Server ise on koostatud nii, et kui üks paar on omaette lõime abil mängima saadetud, siis asub serveri peaprogramm ise järgmist kasutajat ootama ning paari kokku saamisel paneb need taas omavahel mängima. Edasine klientide vaheline vestlus on juba nende otsustada. Server seda otseselt ei määra ega kontrolli. Vaid kirjutamise järg antakse mängijatele kordamööda. Kui keegi mängijatest saadab lõputunnuse, siis lõpetatakse neid mängijaid teenindava lõime töö.

Kliendi pool ühendab end serveriga ning asub sealt tulevaid teateid ootama. Kui öeldakse, millise sümboliga klient mängib, siis jäetakse see meelde. Kui teatatakse toimunud käigust, siis joonistatakse selle tulemus nuppudest moodustatud mängulauale. Kui oli tegemist mängu alguse teatega või vastase käiguga, siis muudetakse lubatud käigunupud aktiivseks, et kasutaja saaks sobiva valida. Vajutuse peale muudetakse nupud taas külmunuks, saadetakse nupule vastav käik serverisse ning edasi tegutsetakse vastavalt saabuvatele teadetele. Lõpetusnupule vajutades saadetakse lõpetusteade, mille peale server teab selle edasi saata mängupartnerile ning samas ka nende mängijatega suhtleva lõime sulgeda. Saadetavad käsud näevad välja järgnevad:

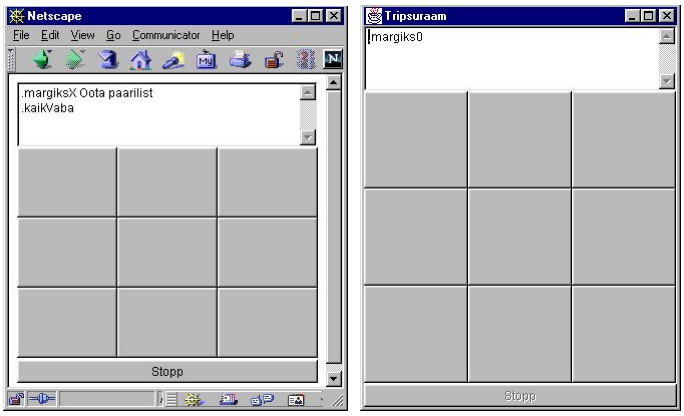
Kuju	Tähendus
.margiksX	Teate saanud klient teab edaspidi, et tema mängib ristidega. Kui olnuks .margiks0, tuleks nullidega mängida
.kaikX5	Mängija X käis nupu 5. .kaik03 tähendanuks, et mängija 0 vajutas nuppu 3.
.kaikVaba	Teate saanud klient võib oma käiguga alustada.
.ots	Mäng on lõppenud

Tutvustavad pildid

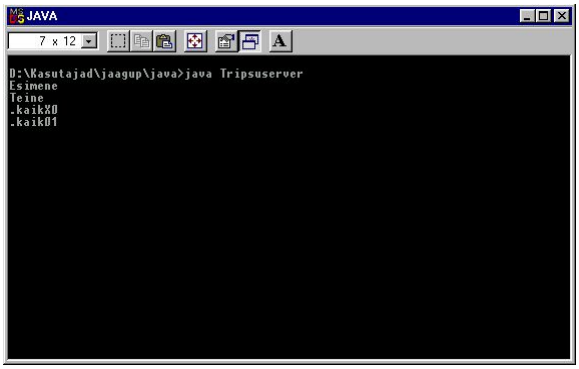
Järgnevalt on mõned pildid ühest konkreetsest mängust. Masinas on käivitatud serverprogramm, üks klient käsurealt ning teine veebiseilurist. Serverisse on jõudnud mõlemad kliendid ühenduda (sellest teatavad serveri ekraanil sõnad Esimene ja Teine). Klient on serverist teateid kuulava lõime tööle saanud.



Seiluriaknas olev klient jõudis ühenduda esimesena. Temale tuli kõigepealt teade .margiksX Oota paarilist, mille peale inimene teab, et tal tuleb partnerit oodata. Partneri saabumisel antakse sellele teada .margiks0 ning seejärel esimesele ühendunule .kaikVaba. Selle peale vabastatakse esimese kliendi nupud lukust ning tal on vaba voli käia. Mängijal on vaba voli valida üheksa mängunupu ning Stopp-nupu vahel.

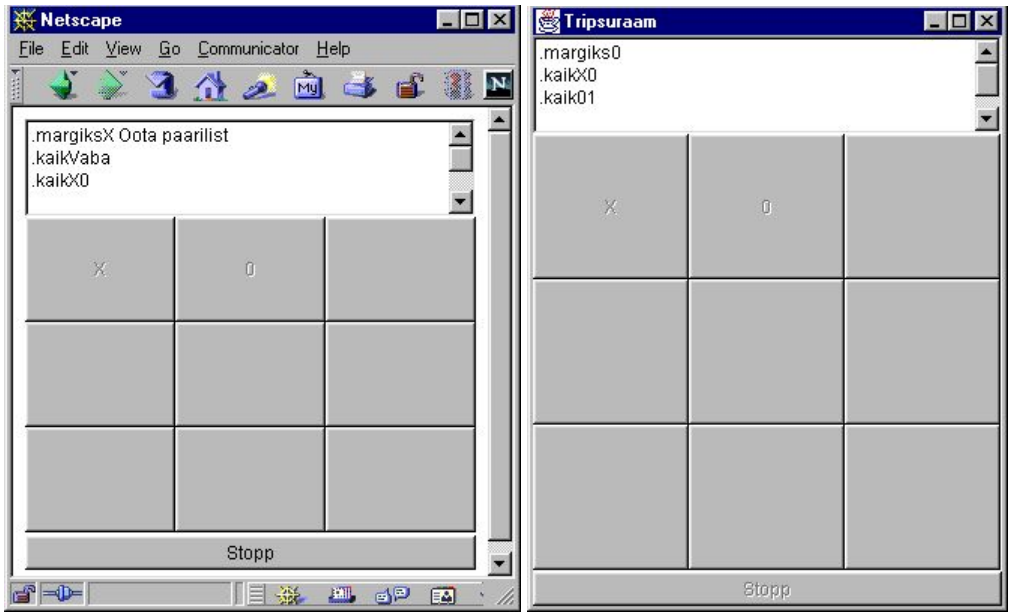


Nagu serveri konsoolilt näha, käis X-iga mängija kõigepealt nupule nr. 0 ning seejärel 0-ga mängija nupule 1.




```
D:\Kasutajad\jaagup\java>java Tripsuserver
Esimene
Teine
.kaikX0
.kaik01
```

Sama tulemus paistab ka mängulaudu vaadates. Saabuvad teated on lihtsalt kontrolli mõttes ülal asuvasse tekstivälja paigutatud. Nupud on aga samuti õigesti märgitud.



Nii võib mäng jätkuda kuniks mängijad seda soovivad. Praeguse näite puhul veel võite ei loeta ning tulemusi ei kontrollita. Kes leiab, et ta enam midagi arukat käia ei taha ega oska, võib vajutada stopp. Selle peale katkestatakse serveris selle mängu lõim ning ka kaaslasele saadetakse teade lõpetamisest. Nupule ilmub kiri Start ning kui sellele vajutada, algab mängijate kohtumine otsast peale.

Serveripoole lähtekood seletustega

```
import java.io.*;
import java.net.*;
```

Võrguprogrammide puhul alati vajalikud paketid

```
public class Tripsuserver{
```

Programmi töö käigus vajalikud konstandid. Kui nende väärtused ühte kohta kirjutada, siis on vajadusel nende väärtusi kergesti võimalik muuta. Näiteks kui juhtub vastava värati peal juba mõni programm jooksma, siis tuleb siia mõni vaba number määrata. Kui tegemist on käsurealt käivitatava kliendiga ning server jookseb kohalikust masinast väljaspool, siis tuleb vastava serveri nimi siia kirjutada, et klient teaks sinna ühenduda. Rakendi puhul pole siia märgitud masina nimi tähtis, sest rakend saab alati ühenduda vaid serverisse kust ta ise pärit ning selle aadress küsitakse töö käigus.

```
  static final int pordinr=3001;
  static final String masin="localhost";
  static final String lopp=".ots";
```

Serveri main-meetod ei tegele lihtsuse mõttes eriolukordade töötlemisega vaid laseb veateated lihtsalt konsoolile trükkida. Selleks on kiri `throws Exception` meetodi päises.

```
  public static void main(String argumendid[]) throws Exception{
```

Asutakse väratit kuulama

```
    ServerSocket ss=new ServerSocket(Tripsuserver.pordinr);
```

Jätkatakse igaveses tsüklis, st. senikaua kuni serverirakendus CTRL+C-ga kinni pannakse. Viisakam, kuid keerulisem võimalus oleks luua eraldi protokoll serveri juhtimiseks ning selle abil saata serverile teateid sulgemise või muude toimingute kohta.

```
    while(true){
```

Oodatakse esimese kliendi saabumist ning teatatakse talle, et tema märgiks on X. Serveri administraatorile antakse teada, et paarist esimene klient on saabunud.

```
      Socket sc1=ss.accept();
       new PrintWriter(sc1.getOutputStream(), true).println(".margiksX "+
                                                    "Oota paarilist");
       System.out.println("Esimene");
```

Sama lugu teise kliendiga.

```
      Socket sc2=ss.accept();
      new PrintWriter(sc2.getOutputStream(), true).println(".margiks0");
      System.out.println("Teine");
```

Luuakse `Tripsuloim`'e nimelisest klassist uus eksemplar ning antakse sellele kaasa juurdepääsuvõimalus mõlema kliendi ühendusele, et loodud isendil oleks võimalus hakata nende omavahelise suhtluse üle hoolt kandma.

```
      new Tripsuloim(sc1, sc2);
    }
  }
```

Ning serveri peaklassil rohkem tööd polegi.

```
}
```

Tripsuloim'e ülesandeks on siis talle etteantud ühendustega suhtlema hakata ning hoolitseda, et nad omavahel saaksid mängu ilusti peetud.

```
class Tripsuloim extends Thread{
```

Klassi sisse luuakse koht ühenduste andmete hoidmiseks. Lühiduse mõttes on tehtud Socket tüüpi massiiv. Sellisel juhul pole vaja kahte eraldi muutujat ning ühised operatsioonid (näiteks sulgemine lõpus) on võimalik tsükli abil läbi viia nii, et pole tarvilik kummagi pistiku jaoks eraldi käske välja kirjutada. Eeldatakse, et esimese mängija ühendus paigutatakse elemendiks sc[0] ning teise oma sc[1].

```
  Socket[] sc=new Socket[2];
```

Konstruktor saab kahe pistiku andmed, talletab need kohalikku pistikühenduste massiivi ning `start`-meetodi käivitamise abil palub virtuaalmasinal käivitada `run`-nimeline meetod eraldi lõimena, mis peaks hoolt kahe saabunud ühenduse vahelise suhtlemise eest. Kuni siiani ootab veel klassis `Tripsuserver` olev peaprogramm iga täidetava käsu taga enne kui oma järjega edasi läheb. Kui aga konstruktor on läbitud, siis võib peaprogramm käsu `new Tripsuloim(sc1, sc2);` edukalt lõppenuks lugeda ning järgmise juurde asuda. Järgmiseks tuleb aga peaprogrammis hüpe tsükli algusse ning siis asutakse juba uut klienti ootama. Loodud `Tripsuloim`'e isend aga toimetab tasapisi `run`-meetodis edasi.

```
  public Tripsuloim(Socket usc1, Socket usc2){
    sc[0]=usc1;
    sc[1]=usc2;
    start();
  }
```

Omaette lõimes töötav mängijapaarivahelist suhtlust korraldav meetod.

```
  public void run(){
```

Veapüünis, kuna üle kaetud run-meetodist pole võimalik erindeid throws käsuga välja lasta, samas aga mitmed võrguga seotud käsud nõuavad erindite töötlemist. Samuti on viisakas tekkivatele probleemidele reageerida.

```
    try{
```

Nii sisend- kui väljundvoogude tarbeks luuakse massiivid, et pärastpoole oleks nende voogude poole mugavam pöörduda.

```
      BufferedReader sisse[]=new BufferedReader[2];
      PrintWriter valja[]=new PrintWriter[2];
      for(int i=0; i<2; i++){
        sisse[i]=new BufferedReader(
          new InputStreamReader(sc[i].getInputStream())
        );
        valja[i]=new PrintWriter(sc[i].getOutputStream(), true);
      }
```

Esimesele kliendile teatatakse, et too võib oma käiguga alustada.

```
      valja[0].println(".kaikVaba");
```

Muutuja veel näitab, kas vastav lõim peab oma tööd jätkama. Kui klient saadab lõpetamisteate, siis muutuja väärtus läheb vääraks ning enam uut ringi teateid kuulama ei minda.

```
      boolean veel=true;
      while(veel){
```

Esimeselt kliendilt saabunud käik või lõpetamisteade saadetakse mõlemale edasi. Klient on koostatud nii, et korrektseks toimimiseks peab ta ka ise serverilt oma saadetud teated kätte saama. See on justkui kontroll, et ühendus serveriga töötab.

```
        String vastus=sisse[0].readLine();
        kirjuta(valja, vastus);
```

Kui esimeselt kliendilt saadi lõpetamisteade, siis teise kliendi teadet enam ei oodata.

```
        if(!vastus.equals(Tripsuserver.lopp)){
```

Muul juhul kirjutatakse ka teiselt saabunud teade mõlemale.

```
            vastus=sisse[1].readLine();
            kirjuta(valja, vastus);
          }
          if(vastus.equals(Tripsuserver.lopp))veel=false;
        }
```

Jõudnud tsüklist välja, suletakse ühendused mõlema kliendiga.

```
        for(int i=0; i<2; i++){
          sc[i].close();
        }
    }catch(Exception e){
      System.out.println("Probleem:"+e);
    }
  }
```

Abimeetod teate välja saatmiseks. Teade jõuab nii mõlemale kliendile kui serveri konsoolile. Piiritleja private meetodi sees teatab, et seda võib kasutada ainult sama klassi (`Tripsuloim`) isendi seest.

```
  private void kirjuta(PrintWriter[] pw, String teade){
    for(int i=0; i<pw.length; i++){
      pw[i].println(teade);
    }
    System.out.println(teade);
  }
}
```

Kliendipoole lähtekood seletustega

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
```

Appleti (rakendi) alamlass, et oleks võimalik seda veebilehel tööle panna.

```
public class Tripsuklient extends Applet{
```

Serveri nimi küsitakse klassi Tripsuserver staatilisest muutujast. Andmeid käiakse sellepärast teisest klassist uurimas, et nii on võimalik väärtused ühte kohta kokku kirjutada ning konfigureerimine on lihtsam. Samas aga peab kliendi käivitamiseks sellisel juhul ka serveriklass kaasas olema.

```
  static String serverinimi=Tripsuserver.masin;
  public Tripsuklient(){
    try{serverinimi=getCodeBase().getHost();}catch(Exception e){}
      //rakendi puhul võetakse serveri nimi brauserilt
```

Rakendi vaikimisi paigutushalduriks on FlowLayout (elemendid vabas suuruses üksteise järel). BorderLayouti abil keskele paigutatuna on võimalik sisemine paneel üle kogu pinna välja venitada.

```
    setLayout(new BorderLayout());
    add(new Tripsupaneel(), BorderLayout.CENTER);
  }

  public static void main(String argumendid[]){
```

Käsurealt käivitades võimaldatakse serveri nimi käsurea parameetrina anda. Kui aga pole parameetreid antud, sel juhul jäetake vaikimisi nimi. Seilurilt klienti vaadates main meetod ei käivitu.

```
    if(argumendid.length>0)serverinimi=argumendid[0];
    Frame f=new Frame("Tripsuraam");
    f.add(new Tripsupaneel(), BorderLayout.CENTER);
    f.setSize(300, 300);
    f.setVisible(true);
  }
}
```

Kujundus on Tripsupaneeli nimelisse klassi kokku pandud. Klass nimega Tripsuklient on vaid käivitamiseks.

```
class Tripsupaneel extends Panel{
```

Kõik lehel nähtavad nupud on paigutatud ühisesse massiivi. Sellisena on neid kergem tsükli abil külmutada, sulatada või puhastada.

```
  Button nupp[]=new Button[10];
  String omanimi;  //X või 0, mille alt klient mängib
  TextArea suhtlus=new TextArea("",3, 60,TextArea.SCROLLBARS_VERTICAL_ONLY);
  public Tripsupaneel(){
    setLayout(new BorderLayout());
    add(suhtlus, BorderLayout.NORTH);
```

Nupud omaette paneeli, kus 3*3 elementi.

```
    Panel nupupaneel=new Panel();
    nupupaneel.setLayout(new GridLayout(3, 3));
```

Tsükli abil luuakse üheksa mängunuppu, igaüks neist lisatakse nii paneeli näitamiseks kui massiivi pärastiseks kergemaks ligipääsuks.

```
    for(int i=0; i<9; i++){
      nupp[i]=new Button("  ");
      nupupaneel.add(nupp[i]);
    }
    add(nupupaneel, BorderLayout.CENTER);
```

Massiivi viimane nupp mängu peatamiseks. Kui nupupaneel paigutati `Tripsupaneel`i keskele (kogu vabale alale), siis seiskamis/käivitusnupp pannakse `Tripsupaneel`i allserva, nupupaneeli alla.

```
    nupp[9]=new Button("Stopp");
    add(nupp[9], BorderLayout.SOUTH);
```

Kõik nupud külmutatakse.

```
    seaNupud(false);
```

Käivitatakse serveri teadete kuulamiseks ning nende töötlemiseks omaette lõim. Lõimele antakse ette osuti loodud Tripsupaneelile, mille kaudu on omakorda võimalik ligi pääseda seal paiknevatele nuppudele ning tekstialale.

```
    TripsukliendiLoim vahendaja=new TripsukliendiLoim(this);
  }
```

Alamprogramm kliendi nuppude külmutamiseks ning lahti sulatamiseks. Külmutamise korral külmutatakse kõik nupud. See toimub juhul, kui käigujärg läheb vastasele ning sel ajal pole server võimeline ühtki teadet vastu võtma. Lahti sulatamisel aga tehakse toimivaks vaid nupud, millel pole mingit kirja ehk mängunuppude puhul need, mis on veel vabad.

```
  void seaNupud(boolean kasutatav){
    if(!kasutatav)for(int i=0; i<nupp.length; i++)nupp[i].setEnabled(kasutatav);
    else{
      for(int i=0; i<9; i++)if(nupp[i].getLabel().trim().equals(""))
         nupp[i].setEnabled(kasutatav);
    }
```

Seiskamis/käivitusnupp muudetakse igal juhul vastavalt etteantud parameetrile, st. tehakse kasutatavaks ka siis kui sinna on midagi kirjutatud.

```
    nupp[9].setEnabled(kasutatav);
  }
}
```

Lõim, mille ülesandeks on sidepidamine kliendi ja serveri vahel: serverist tulevate teadete järgi nupu pealkirjade muutmine, samuti kasutajapoolsetest nupuvajutustest käikude koostamine ning serveri poole saatmine.

```
class TripsukliendiLoim implements ActionListener, Runnable{
  Tripsupaneel tp;
```

```
PrintWriter valja;
BufferedReader sisse;
boolean veel=true;
Socket sc;

public TripsukliendiLoim(Tripsupaneel utp){
```

Jäetakse meelde osuti etteantud Tripsupaneelile

```
  tp=utp;
```

Kõik Tripsupaneeli nupud pannakse siinsele lõimeklassi isendile teateid saatma. St, et ükskõik millist nuppu paneelil vajutatakse, ikka käivitub siinse isendi actionPerformed.

```
  for(int i=0; i<10; i++)tp.nupp[i].addActionListener(this);
  alusta();
}
```

Uue mängu alustamisel sooritatavad toimingud. Väljastatav tõeväärtus teatab, kas töö õnnestus.

```
public boolean alusta(){
  boolean korras=true;
  try{
```

Mängunupud pealkirjatuks

```
   for(int i=0; i<9; i++)tp.nupp[i].setLabel(" ");
```

Käivitusnupule kiri Stopp

```
   tp.nupp[9].setLabel("Stopp");
```

Teatekastina kasutatav tekstiväli tühjaks

```
   tp.suhtlus.setText("");
```

Uus ühendus serverisse, sellest omakorda küsitakse sisend- ja väljundvoog.

```
   sc=new Socket(Tripsuklient.serverinimi, Tripsuserver.pordinr);
   sisse=new BufferedReader(
      new InputStreamReader(sc.getInputStream())
   );
   valja=new PrintWriter(sc.getOutputStream(), true);
   veel=true;
```

Luuakse uus lõim, mis run-meetodi (taas) käima paneb. Kui tähele panna, siis ülal klassi kirjelduses polnud mitte kirjas extends `Thread`, vaid oli implements `Runnable`. Seetõttu tuleb omaette `Thread` klassi isend luua ning sellele anda siinne isend käivitamiseks (ei saa lihtsalt siin samale isendile `start` ütelda). Samas on nii võimalik kergesti iga uue mängu algul `run` taas uue lõimena käima panna.

```
   new Thread(this).start();
  }catch(IOException e){
```

Veateade väljastatakse nii konsoolile kui tekstialasse.

```
    e.printStackTrace();
    tp.suhtlus.setText("Ühendus serveriga "+Tripsuklient.serverinimi+" puudub.");
    korras=false;
  }
  return korras;
}

public void run(){
  System.out.println("Loime algus");
  try{
     while(veel){
```

Serverist saabuvad read saadetakse tootle –nimelisele meetodile töötlemiseks.

```
       tootle(sisse.readLine());
     }
  } catch (Exception e){
```

```
      System.out.println("Probleem: "+e);
      e.printStackTrace();
    }
    System.out.println("Loime ots");
  }
```

Iga saabuva teatega käitutakse vastavalt selle sees olevale sisule.

```
  private void tootle(String teade) throws IOException{
```

Igal juhul lisatakse teade tekstialasse kontrolliks

```
    tp.suhtlus.setText(tp.suhtlus.getText()+teade+"\n");
```

Kui saabub teade kliendi kasutatava märgi kohta, siis jäetakse see meelde. Et vastav muutuja asub siinsele lõimeklassile etteantud Tripsupaneeli isendis, tuleb sellele Tripsupaneelile kõigepealt tp-nimelise muutuja kaudu ligi minna ning siis sealtkaudu märk muutujale omanimi omistada.

```
    if(teade.startsWith(".margiks"))tp.omanimi=teade.substring(8, 9);
```

Käigu puhul eeldatakse, et täht number 5 näitab käija nime ning täht nr. 6 nuppu, millele vajutati. Kui pakutud number on vigane, siis jäetakse käsk täitmata.

```
    if(teade.startsWith(".kaik")){
      try{
        int nr=Integer.parseInt(teade.substring(6, 7));
        tp.nupp[nr].setLabel(
          " "+teade.substring(5, 6)+" "
        );
      } catch(NumberFormatException e){} //sobimatu ruut
```

Vastase käigu puhul tehakse laual olevad nupud taas tundlikuks. Kontrollitakse, et käija nimi ei ühtiks kliendi enese märgiga.

```
      if(!teade.substring(5, 6).equals(tp.omanimi))tp.seaNupud(true);
    }
```

Lõputeate saabumise puhul antakse muutujale veel väärtuseks false, et enam uusi teateid ei kuulataks. Alumine suur nupp tehakse vajutatavaks ning sinna kirjutatakse Start. Suletakse ühendus serveriga.

```
    if(teade.equals(Tripsuserver.lopp)){
      veel=false;
      tp.nupp[9].setLabel("Start");
      tp.nupp[9].setEnabled(true);
      sc.close();
    }
  }
```

Käivitatakse, kui kliendiprogrammis on vajutatut ükskõik millist nuppu.

```
  public void actionPerformed(ActionEvent e){
```

Testiks teatatakse kliendi konsoolile, et nupuvajutus on kinni püütud

```
    System.out.println("Vajutus");
```

Nupud külmutatakse, et sama hooga poleks võimalik rohkem vajutada.

```
    tp.seaNupud(false);
```

Tehakse kindlaks, millisele nupule vajutati. Selleks vaatatakse läbi kõik olemasolevad nupud ning millise osuti on sama väärtusega kui vajutuse allika oma, sellele järelikult vajutati.

```
    int nr=0;
    Button allikas=(Button)e.getSource();
    for(int i=0; i<10; i++)if(tp.nupp[i]==allikas)nr=i;
```

Kui tegemist oli ühega ruudustiku nuppudest, siis saadetakse serverisse teade, millist nuppu kasutaja vajutas.

```
    if(nr<9)
      valja.println(".kaik"+tp.omanimi+nr);
```

Muul juhul peab olema tegemist seiskamis/käivitusnupuga

```
    else if(nr==9){
```

Kui sellel oli kiri stopp, siis saadetakse serverisse selle mängu lõpetusteade.

```
      if(tp.nupp[9].getLabel().equals("Stopp"))
         valja.println(Tripsuserver.lopp);
```

Muul juhul alustatakse uut ühendust ja mängu.

```
      else {
        tp.nupp[9].setLabel("Stopp");
        alusta();
      }
    }
  }
}
```

Edasiarendusvajadused

Näiteprogramm on püütud koostada võimalikult lihtsalt. Seetõttu on mitmed tarvilikud kohad välja jäetud. Võrguprogrammide koostamisel on üheks nõudeks, et kunagi ei tohi usaldada kliendi poolt saadetavaid andmeid, sest avalikus võrgus töötava serveri külge võib ühineda ükskõik kes ning miski ei takista pahatahtlikul sisenejal temale sobivaid baite teele saata. Praegu aga on näiteks mängija nimi meeles kliendipoolses programmis ning kui kliendi simuleerija otsustaks saata enese asemel kellegi teise nime, siis teda ei takistataks ning tal õnnestuks vastase eest käia. Kuna aga server siiski kindlalt otsustab, et käia saab kordamööda, siis õnnestub vaid teise käike rohkem teha, ülemääraseid oma märke kuhugile paigutada ei õnnestu.

Võrguprogrammide puhul on kombeks toimunud tegevused faili üles märkida ehk logida. Siis on selle järgi võimalik leida seletusi nii programmi töö käigus tekkinud probleemidele kui tagantjärele reageerida mängijatevahelistele vaidlustele.

Ka kliendiakna sulgemiseks peaks lihtsast töö katkestamisest viisakam võimalus olema, kus saadetakse serverile lõpetusteade. Rakendi puhul peaks see käivituma veebilehelt lahkumisel, rakenduse puhul aga akna sulgemisristile vajutamisel.

Programmitekst

Järgnevalt programmitekst tervikuna ilma vahele piktud kommentaarideta.

```
import java.io.*;
import java.net.*;
public class Tripsuserver{
  static final int pordinr=3001;
  static final String masin="localhost";
  static final String lopp=".ots";
  public static void main(String argumendid[]) throws Exception{
    ServerSocket ss=new ServerSocket(Tripsuserver.pordinr);
    while(true){
      Socket sc1=ss.accept();
       new PrintWriter(sc1.getOutputStream(), true).println(".margiksX "+
                                                    "Oota paarilist");
       System.out.println("Esimene");
      Socket sc2=ss.accept();
       new PrintWriter(sc2.getOutputStream(), true).println(".margiks0");
       System.out.println("Teine");
      new Tripsuloim(sc1, sc2);

    }
  }
}

class Tripsuloim extends Thread{
  Socket[] sc=new Socket[2];
  public Tripsuloim(Socket usc1, Socket usc2){
    sc[0]=usc1;
    sc[1]=usc2;
    start();
```

```
  }
  public void run(){
    try{
       BufferedReader sisse[]=new BufferedReader[2];
       PrintWriter valja[]=new PrintWriter[2];
       for(int i=0; i<2; i++){
         sisse[i]=new BufferedReader(
           new InputStreamReader(sc[i].getInputStream())
         );
         valja[i]=new PrintWriter(sc[i].getOutputStream(), true);
       }
       valja[0].println(".kaikVaba");
       boolean veel=true;
       while(veel){
         String vastus=sisse[0].readLine();
         kirjuta(valja, vastus);
         if(!vastus.equals(Tripsuserver.lopp)){
            vastus=sisse[1].readLine();
            kirjuta(valja, vastus);
         }
         if(vastus.equals(Tripsuserver.lopp))veel=false;
       }
       for(int i=0; i<2; i++){
         sc[i].close();
       }
    }catch(Exception e){
      System.out.println("Probleem:"+e);
    }
  }
  private void kirjuta(PrintWriter[] pw, String teade){
    for(int i=0; i<pw.length; i++){
      pw[i].println(teade);
    }
    System.out.println(teade);
  }
}
```

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;

public class Tripsuklient extends Applet{
  static String serverinimi=Tripsuserver.masin;
  public Tripsuklient(){
    try{serverinimi=getCodeBase().getHost();}catch(Exception e){}
      //rakendi puhul võetakse serveri nimi brauserilt
    setLayout(new BorderLayout());
    add(new Tripsupaneel(), BorderLayout.CENTER);
  }

  public static void main(String argumendid[]){
    if(argumendid.length>0)serverinimi=argumendid[0];
    Frame f=new Frame("Tripsuraam");
    f.add(new Tripsupaneel(), BorderLayout.CENTER);
    f.setSize(300, 300);
    f.setVisible(true);
  }
}
```

```
class Tripsupaneel extends Panel{
  Button nupp[]=new Button[10];
  String omanimi;
  TextArea suhtlus=new TextArea("",3, 60,TextArea.SCROLLBARS_VERTICAL_ONLY);
  public Tripsupaneel(){
    setLayout(new BorderLayout());
    add(suhtlus, BorderLayout.NORTH);
    Panel nupupaneel=new Panel();
    nupupaneel.setLayout(new GridLayout(3, 3));
    for(int i=0; i<9; i++){
      nupp[i]=new Button("  ");
      nupupaneel.add(nupp[i]);
    }
    add(nupupaneel, BorderLayout.CENTER);
    nupp[9]=new Button("Stopp");
```

```
    add(nupp[9], BorderLayout.SOUTH);
    seaNupud(false);
    TripsukliendiLoim vahendaja=new TripsukliendiLoim(this);
  }

  void seaNupud(boolean kasutatav){
    if(!kasutatav)for(int i=0; i<nupp.length; i++)nupp[i].setEnabled(kasutatav);
    else{
      for(int i=0; i<9; i++)if(nupp[i].getLabel().trim().equals(""))
         nupp[i].setEnabled(kasutatav);
    }
    nupp[9].setEnabled(kasutatav);
  }
}

class TripsukliendiLoim implements ActionListener, Runnable{
  Tripsupaneel tp;
  PrintWriter valja;
  BufferedReader sisse;
  boolean veel=true;
  Socket sc;

  public TripsukliendiLoim(Tripsupaneel utp){
    tp=utp;
    for(int i=0; i<10; i++)tp.nupp[i].addActionListener(this);
    alusta();
  }

  public boolean alusta(){
    boolean korras=true;
    try{
     for(int i=0; i<9; i++)tp.nupp[i].setLabel(" ");
     tp.nupp[9].setLabel("Stopp");
     tp.suhtlus.setText("");
     sc=new Socket(Tripsuklient.serverinimi, Tripsuserver.pordinr);
     sisse=new BufferedReader(
        new InputStreamReader(sc.getInputStream())
     );
     valja=new PrintWriter(sc.getOutputStream(), true);
     veel=true;
     new Thread(this).start();
    }catch(IOException e){
      e.printStackTrace();
      tp.suhtlus.setText("Ühendus serveriga "+Tripsuklient.serverinimi+" puudub.");
      korras=false;
    }
    return korras;
  }

  public void run(){
    System.out.println("Loime algus");
    try{
       while(veel){
         tootle(sisse.readLine());
       }
    } catch (Exception e){
      System.out.println("Probleem: "+e);
      e.printStackTrace();
    }
    System.out.println("Loime ots");
  }

  private void tootle(String teade) throws IOException{
    tp.suhtlus.setText(tp.suhtlus.getText()+teade+"\n");
    if(teade.startsWith(".margiks"))tp.omanimi=teade.substring(8, 9);
    if(teade.startsWith(".kaik")){
      try{
        int nr=Integer.parseInt(teade.substring(6, 7));
        tp.nupp[nr].setLabel(
          " "+teade.substring(5, 6)+" "
        );
      } catch(NumberFormatException e){} //sobimatu ruut
      if(!teade.substring(5, 6).equals(tp.omanimi))tp.seaNupud(true);
    }
    if(teade.equals(Tripsuserver.lopp)){
      veel=false;
      tp.nupp[9].setLabel("Start");
      tp.nupp[9].setEnabled(true);
      sc.close();
    }
  }
```

```
  public void actionPerformed(ActionEvent e){
    System.out.println("Vajutus");
    tp.seaNupud(false);
    int nr=0;
    Button allikas=(Button)e.getSource();
    for(int i=0; i<10; i++)if(tp.nupp[i]==allikas)nr=i;
    if(nr<9)
      valja.println(".kaik"+tp.omanimi+nr);
    else if(nr==9){
      if(tp.nupp[9].getLabel().equals("Stopp"))
         valja.println(Tripsuserver.lopp);
      else {
        tp.nupp[9].setLabel("Stopp");
        alusta();
      }
    }
  }
}
```

Jututoa graafiline klient.

Märgatava osa graafiliste võrguprogrammide puhul tuleb rakenduse töölesaamiseks lahendada hulk sarnaseid probleeme. Jututoa näite põhjal püüame need lõigud läbi käia.

Lihtsaim komplekt.

Alustame lihtsast Java põhikursuse konspektis kirjeldatud serverist ning kirjutame sinna juurde graafilise kliendi andmete saatmiseks ja lugemiseks. Serveri pea ainsaks oskuseks on kõik kasutajate poolt tulnud andmed kõigile sisse meldinud klientidele laiali saata.

Klient seevastu võtab ühenduse serveriga. Iga kasutaja tipitud rea sisestuse järel saadetakse see serverisse. Kõik serverist saabunud read aga paigutatakse üksteise järel tekstialasse. Ühendus serveriga luuakse kliendi käivitamise ajal. Serveri nimi ja värat on koodi sees kirjas. Paigutuseks piisab kolmest reast.

```
    setLayout(new BorderLayout());
    add(tf1, BorderLayout.SOUTH);
    add(ta1, BorderLayout.CENTER);
```

BorderLayout lubab paigutada servadesse ja keskele. Alla serva paigutatakse tekstiväli teadete sisestamiseks. Keskele tekstiala andmete lugemiseks. Tekstivälja venitatakse laiust pidi vastavalt akna suurusele. Tekstiala katab akna sees ülejäänud keskele jääva vaba pinna.

```
       Socket sc=new Socket(serverinimi, 3001);
```

küsib pistikühenduse soovitud serveri ja väratiga. Kui ühenduse loomine õnnestus, siis küsitakse sinna pistikusse sisnd- ning väljundvoog. Parameeter true PrintWriteri konstruktoris tähendas, et andmed saadetakse iga println-käskusega teele. Vastasel juhul võiks juhtuda, et ennem kogutakse mitme kilobaidi jagu teateid, kui programm leiab vajaliku neid võrku mööda teele saata.

```
       pw1=new PrintWriter(sc.getOutputStream(), true);
       br1=new BufferedReader(new
                   InputStreamReader(sc.getInputStream()));
```

Kui ühendused loodud, lükatakse tööle eraldi lõim teadete püüdmiseks. Kui jääda lihtsalt miskis alamprogrammis ootama võrgust saabuvaid teateid, siis võib jääda kogu rakendus seniks hangunuks, kuni sealt rida saabub. On aga teadete vastuvõtt omaette lõimes, siis see ootamine muid toiminguid ei sega.

```
       new Thread(this).start();
```

Meetodis run paiknev teadete vastuvõtt on kirja pandud küllalt lühidalt:

```
    while(true){
      ta1.append(br1.readLine()+"\n");
    }
```

Ehk siis igavene tsükkel, kus readLine muudkui ootab võrgu pealt saabuvat teadet. Saabumise järel lisatakse see tekstialasse koos järgneva reavahetusega ning asutakse taas uut rida ootama. Juhtub aga lugemisega probleeme olema, satutakse tsüklist välja katsendiplokki. Ning programmi sulgemiseks pole esiotsa viisakamat moodust kui protsessi töö kas siis Ctrl+C või mõne muu vahendi abil lõpetada. Omad puudused sel kliendil on, kuid lihtsaks ühenduse testimiseks peaks sobima küll.

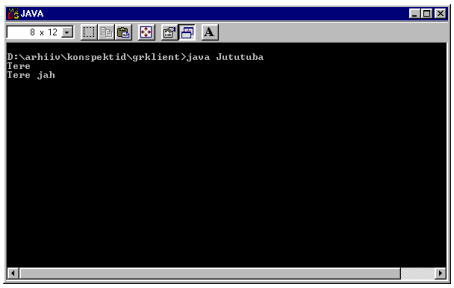

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;

public class GrKlient1 extends Applet
                implements ActionListener, Runnable{
  TextField tf1=new TextField(15); //suurus
  TextArea ta1=new TextArea(5, 20);
  PrintWriter pw1;
  BufferedReader br1;
  final String serverinimi="localhost";
  public GrKlient1(){
    setLayout(new BorderLayout());
    add(tf1, BorderLayout.SOUTH);
    add(ta1, BorderLayout.CENTER);
    tf1.addActionListener(this);
    try{
       Socket sc=new Socket(serverinimi, 3001);
       pw1=new PrintWriter(
                  sc.getOutputStream(), true);
       br1=new BufferedReader(new
         InputStreamReader(sc.getInputStream()));
       new Thread(this).start();
    }catch(Exception viga){
      ta1.setText(viga.getMessage());
    }
  }

  public void actionPerformed(ActionEvent e){
     pw1.println(tf1.getText());
     tf1.setText("");
  }

  public void run(){
   try{
    while(true){
      ta1.append(br1.readLine()+"\n");
    }
   }catch(Exception viga){
     viga.printStackTrace();
     ta1.append("Oled väljas");
   }
  }

  public static void main(String[] argumendid){
     Frame f=new Frame();
     f.add(new GrKlient1());
     f.setSize(300, 300);
     f.setVisible(true);
  }
}
```

Lihtne server

Lühidalt taas toodud koodilõik, mille abil võimalik käivitada serverprogramm, kuhu huvilised liituma pääsevad ning mis igalt kasutajalt saabuvad teated kõigile edasi annab. Kuna siinne programm ei tea koodi sisust midagi, siis võib sama "mootori" külge ehitada rakendusi vastavalt kirjutaja fantaasiale.

```
import java.io.*;
import java.net.*;
import java.util.Vector;
public class Jututuba{
  public static void main(String argumendid[]) throws IOException{
```

```
    ServerSocket ss=new ServerSocket(3001);
    Vector uhendused=new Vector();
    while(true){
      Socket sc=ss.accept();
      uhendused.add(sc);
      new JututoaLoim(sc, uhendused);
    }
  }
}

class JututoaLoim extends Thread{
  Vector v;
  Socket sc;
  public JututoaLoim(Socket uus_sc, Vector uus_v){
    v=uus_v;
    sc=uus_sc;
    start();
  }
  public void run(){
    try{
      BufferedReader sisse=new BufferedReader(
        new InputStreamReader(sc.getInputStream())
      );
      boolean veel=true;
      while(veel){
         String rida=sisse.readLine();
         System.out.println(rida);
         if(rida.startsWith(".ots"))veel=false;
         for(int i=0; i<v.size(); i++){
           Socket skt=(Socket)v.elementAt(i);
           PrintWriter valja=new PrintWriter(skt.getOutputStream(), true);
           valja.println(rida);
         }
      }
     sc.close();
    } catch(Exception e){
      System.out.println("Probleem: "+e);
    }
    v.remove(sc);
  }
}
```

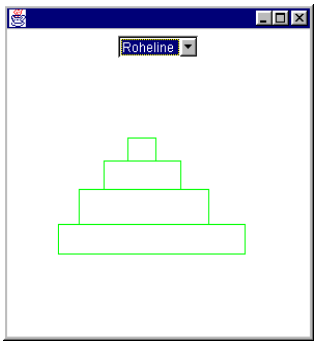


Tahvel

Soovides jututoale külge ehitada tahvlit, on enne hea järele proovida, kuidas lihtne joonistusvahend eraldi elama panna. Esiotsa joonistatakse kujund mouseReleased käskluse sees, ehk kohe, kui soovitud andmed teada on. paint-meetodi puudumise tõttu akna suurendamisel või korraks teise alla peitmisel lähevad andmed kaduma - see lihtsustusest tulenev viga parandatakse järgmises näites. Värv valitakse vastavalt rippmenüüle. Kuna värv määratakse elemendi järjekorranumbri ja mitte teksti abil, siis oleks vajadusel võimalik rakendus tõlkida mõnda muusse keelde ilma, et sõnade muutmine toimimist takistaks.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Tahvel1 extends Applet
  implements MouseListener{
  int hax, hay, hyx, hyy;
  String[] varvid={"Sinine", "Punane", "Kollane", "Roheline"};
  Color[] c={Color.blue, Color.red, Color.yellow, Color.green};
  Choice varvivalik=new Choice();
  public Tahvel1(){
    addMouseListener(this);
    for(int i=0; i<varvid.length; i++){
      varvivalik.addItem(varvid[i]);
    }
    add(varvivalik);
```

```
  }

 public void mousePressed(MouseEvent e){
   hax=e.getX();
   hay=e.getY();
 }
 public void mouseReleased(MouseEvent e){
   hyx=e.getX();
   hyy=e.getY();
   int laius=hyx-hax;
   int korgus=hyy-hay;
   Graphics g=getGraphics();
   g.setColor(c[varvivalik.getSelectedIndex()]);
   g.drawRect(hax, hay, laius, korgus);
 }
 public void mouseClicked(MouseEvent e){}
 public void mouseEntered(MouseEvent e){}
 public void mouseExited(MouseEvent e){}
 public static void main(String[] argumendid){
   Frame f=new Frame();
   f.add(new Tahvel1());
   f.setSize(300, 300);
   f.setVisible(true);
 }
}
```



Täiendatud tahvel

Eelnevalt sai meelde tuletada, kuidas midagi pinnale joonistada õnnestub. Selline tahvel võib olla muu rakenduse juures küll niisama joonistusharjutuseks, kuid kuigi lihtsat tahvli pilti mõne muu programmi osaga ühendada ei saa.

Järgnevalt on kirjeldatud liides JooniseKuular ning seal sees käsklus, mis mõeldud kujundi andmete ühest komponendist teise saatmiseks.

```
interface JooniseKuular{
  public void kujund(String rida);
}
```

Kompilaator kontrollib vaid vastava käsu olemasolu ning lubab liidesetüüpi muutuja kaudu vastavaid käsklusi välja kutsuda. Rea formaat tuleb aga ise välja mõelda ja pärast sellest kinni pidada, et liidest kasutavad objektid suudaksid teineteisele käsklusi ühemõtteliselt edasi anda. Määran siis oma mõttes (ja vajadusel kirjutan ka liidese juurde kommentaarina üles), et rea abil saab edasi anda joont, ristkülikut või ovaali. Esimesel juhul algab rida tähekombinatsiooniga .pj, teisel .pr ning kolmandal .po. Edasi järgnevad juba neli koordinaati, mis Javas vastavate kujundite joonistamisel tarvilikud on. Joon ekraanipunktidest 10, 50 punktideni 100, 50 ehk horisontaaljoon peaks siis käsuna välja nägema

```
.pj 10 50 100 50
```

Hiire sündmuste vastuvõtt ning joonistamine ekraanil on teineteisest nõnda lahku viidud, et ainsaks ühenduslüliks nende vahel on loodud kuularliides. Hiire ülesliikumisel käivitatakse käsklus saada, mis siis meelde jäetud koordinaatide abil paneb kokku kujundi määrava rea ning kuulaja olemasolu korral edastab rea kuulajale liidesest võetud käsu kujund abil. Kas kuulajaks on teine tahvel, jututuba või kohalik tahvel ise, sellest ei tea joonistusrida kokkupanev kood midagi. Tahvel2 konstruktoris on rida

```
    kuulaja=this; //Vaikimisi joonistatakse iseenesele.
```

mis nagu juurdelisatud kommentaarigi tähendab, teatab, et kui hiljem pole midagi ümber määratud, siis tahvli sündmuste kuulajaks on tahvel ise, s.t. tahvlile joonistatud kujundid ilmuvad tahvlile enesele.

Väljapoolt tahvli kuulari määramiseks on loodud käsklus

```
  public void seaJooniseKuular(JooniseKuular j){
    kuulaja=j;
  }
```

Nagu näha, saab käsule ette anda JooniseKuular-tüüpi liidest realiseeriva objekti, kuhu siis edaspidi tahvlil hiirega toimetamistest tekkinud sündmuste põhjal kokku pandud kujundeid kirjeldavad

read edasi saadetakse. Erinevalt järgmisest näitest saab siin korraga vaid üks objekt olla tahvlil joonistamise kuulariks. Selline "ühe kuulaja tava" on rohkem kasutusel Java mobiilirakenduste juures, kus programmid väiksemad ning ressursse usinamini kokku hoitakse.

Saatmisel kontrollitakse kõigepealt, kas üldse keegi joonistatavate andmete vastu huvi tunneb. Kui kuulaja on null, siis pole keegi saabuvatest andmetest huvitatud ning teksti pole vaja ka kokku panna. Sarnaseid kokkuhoidmiskohti õnnestub mõnigikord koodi sisse paigutada ning keerulisemate arvutuste ja joonistuste korral võib nii märgatavalt hoida kokku arvuti tööaega.

Edasi leitakse kujundi laius ja kõrgus ehk hiire alla- ja ülesliikumise koordinaatide vahe. Siis saadetakse vastavalt valitud kujundile andmed kuulaja poole teele.

```
void saada(){
  if(kuulaja==null){return;}
  int laius=hyx-hax;
  int korgus=hyy-hay;
  if(kujundivalik.getSelectedItem().equals("Joon")||
     kujundivalik.getSelectedItem().equals("Vabakäejoon")){
    kuulaja.kujund(".pj "+hax+" "+hay+" "+hyx+" "+hyy);
  }
  if(kujundivalik.getSelectedItem().equals("Ristkülik")){
    kuulaja.kujund(".pr "+hax+" "+hay+" "+laius+" "+korgus);
  }
  if(kujundivalik.getSelectedItem().equals("Ovaal")){
    kuulaja.kujund(".po "+hax+" "+hay+" "+laius+" "+korgus);
  }
}
```

Kuna ka tahvel ise realiseerib liidest JooniseKuular, et ta saaks jututoa kliendilt, teiselt tahvlilt või iseeneselt sama liidese kaudu jooniste teateid vastu võtta, siis peab ka tahvli sees olema meetod kujund sõnelise parameetriga. Nagu näha, palutakse vastuvõetud kujundi puhul kõigepealt see ekraanile joonistada ning siis lisatakse saabunud kujundi andmed hoidlasse, et oleks paint-meetodis võimalik ekraaniseis taastada.

```
public void kujund(String andmed){
  joonista(andmed);
  hoidla.add(andmed);
}
```

Joonistuskäskluses lõigatakse andmeid hoidev rida StringTokenizeri abil lõikudeks, leitakse andmereast joonistamiseks vastavad koordinaadid ning esimene jupp reast näitab, millise kujundiga on tegemist. Katsendiplokk on käskudele ümber pandud selleks, et üksik hulka sattunud vigane käsklus ei takistaks kogu pildi joonistamist. Praegusel juhul jääb lihtsalt konkreetne joonistuskäsk täitmata, väljakutsuvale funktsioonile aga sellekohast teadet edasi ei anta.

```
void joonista(String andmed){
 try{
  Graphics g=getGraphics();
  StringTokenizer stk=new StringTokenizer(andmed);
  String tyyp=stk.nextToken();
  int a[]=new int[4];
  for(int i=0; i<4; i++){
     a[i]=Integer.parseInt(stk.nextToken());
  }
  if(tyyp.equals(".pj")){
    g.drawLine(a[0], a[1], a[2], a[3]);
  }
  if(tyyp.equals(".pr")){
    g.drawRect(a[0], a[1], a[2], a[3]);
  }
  if(tyyp.equals(".po")){
    g.drawOval(a[0], a[1], a[2], a[3]);
  }
 }catch(Exception viga){viga.printStackTrace();}
}
```

Meetodis paint on küllalt lühidalt hakkama saadud. Käiakse läbi kõik hoidlas olevad elemendid ning iga rea puhul palutakse sellele vastavad andmed komponendi pinnale joonistada. Listi käsklus iterator väljastab objekti, mille abil võimalik mööda ahelat üha järgmisi elemente küsida. Käsklus hasNext kontrollib, kas veel midagi tulemas on ning next võtab siis järgmise elemendi. Et hoidlas püsivad kõik read ülemklassi Object isendina, siis et saaks andmeid omistada String-tüüpi muutujale, selleks tuleb soovitud tüüp sulgudes ette kirjutada. Edasi juba käsklus joonista andmereale vastava kujundi ekraanile kuvamiseks.

```
public void paint(Graphics g){
```

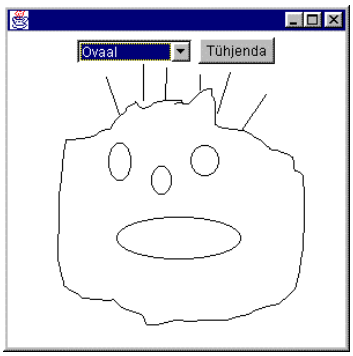
```
    for(Iterator it=hoidla.iterator(); it.hasNext();){
      String s=(String)it.next();
      joonista(s);
    }
  }
```

Kood

Ning edasi rakenduse kood tervikuna.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Tahvel2 extends Applet
  implements MouseListener, MouseMotionListener, JooniseKuular, ActionListener{
  int hax, hay, hyx, hyy;
  String[] kujundid={"Joon", "Ristkülik", "Ovaal", "Vabakäejoon"};
  Choice kujundivalik=new Choice();
  JooniseKuular kuulaja=null;
  LinkedList hoidla=new LinkedList();
  Button tyhjenda=new Button("Tühjenda");

  //objekt, kellele saadetakse tahvlil toimunud sündmused.
  public Tahvel2(){
    addMouseListener(this);
    addMouseMotionListener(this);
    for(int i=0; i<kujundid.length; i++){
      kujundivalik.addItem(kujundid[i]);
    }
    add(kujundivalik);
    add(tyhjenda);
    tyhjenda.addActionListener(this);
    kuulaja=this; //Vaikimisi joonistatakse iseenesele.
  }
  public void seaJooniseKuular(JooniseKuular j){
    kuulaja=j;
  }
  public void paint(Graphics g){
     for(Iterator it=hoidla.iterator(); it.hasNext();){
       String s=(String)it.next();
       joonista(s);
     }
  }

  void joonista(String andmed){
   try{
    Graphics g=getGraphics();
    StringTokenizer stk=new StringTokenizer(andmed);
    String tyyp=stk.nextToken();
    int a[]=new int[4];
    for(int i=0; i<4; i++){
       a[i]=Integer.parseInt(stk.nextToken());
    }
    if(tyyp.equals(".pj")){
      g.drawLine(a[0], a[1], a[2], a[3]);
    }
    if(tyyp.equals(".pr")){
      g.drawRect(a[0], a[1], a[2], a[3]);
    }
    if(tyyp.equals(".po")){
      g.drawOval(a[0], a[1], a[2], a[3]);
    }
   }catch(Exception viga){viga.printStackTrace();}
  }

  public void kujund(String andmed){
    joonista(andmed);
    hoidla.add(andmed);
  }

  public void mousePressed(MouseEvent e){
    hax=e.getX();
    hay=e.getY();
  }
  public void mouseReleased(MouseEvent e){
    hyx=e.getX();
    hyy=e.getY();
    saada();
  }
```

```
  public void actionPerformed(ActionEvent e){
    if(e.getSource()==tyhjenda){
      hoidla.clear();
      repaint();
    }
  }
  public void mouseDragged(MouseEvent e)
  {
    if(kujundivalik.getSelectedItem().equals("Vabakäejoon")){
        hyx = e.getX();
        hyy = e.getY();
        saada();
        hax = hyx;
        hay = hyy;
    }
  }
  public void mouseMoved(MouseEvent e){}

  void saada(){
    if(kuulaja==null){return;}
    int laius=hyx-hax;
    int korgus=hyy-hay;
    if(kujundivalik.getSelectedItem().equals("Joon")||
       kujundivalik.getSelectedItem().equals("Vabakäejoon")){
      kuulaja.kujund(".pj "+hax+" "+hay+" "+hyx+" "+hyy);
    }
    if(kujundivalik.getSelectedItem().equals("Ristkülik")){
      kuulaja.kujund(".pr "+hax+" "+hay+" "+laius+" "+korgus);
    }
    if(kujundivalik.getSelectedItem().equals("Ovaal")){
      kuulaja.kujund(".po "+hax+" "+hay+" "+laius+" "+korgus);
    }
  }

 public void mouseClicked(MouseEvent e){}
 public void mouseEntered(MouseEvent e){}
 public void mouseExited(MouseEvent e){}
 public static void main(String[] argumendid){
   Frame f=new Frame();
   f.add(new Tahvel2());
   f.setSize(300, 300);
   f.setVisible(true);
 }
}
```



Jututoa tahvliga klient.

Kui andmete sisselugemise ja väljasaatmise võimega tahvel olemas, siis tuleb see vaid jututoa kliendile sobivalt külge ühendada ning inimesed saavadki üksteisele pilte ja jooniseid saatma hakata. Kujundite andmeid on vaja saata mõlemas suunas. Võrku ühendunud klient edastab võrgust saabunud kujundikäsklused tahvlile joonistamiseks. Samas tahvlil toimunud hiiretoimingute põhjal saadetakse teated võrku ühendunud kliendile. Et see omakorda saadab teated serverisse, server aga kõigile klientidele laiali, siis jõuab nõnda ringiga ka kohalikus masinas hiirega määratud kujund ekraanile. Kuna tahvlile jõuavad kõik teated ühtviisi võrgu kaudu sõltumata sellest, kas teade pandi teele kohalikust või mõnest muust masinast, siis on kujundi ekraanile ilmumine ka tõend selle kohta, et andmed on serverisse läbi läinud ning sealt tagasi jõudnud.

Et klient ning tahvel on pandud vastastikku teineteisele kujundite teateid saatma, on näha kliendi konstruktori viimastest ridadest.

```
    tahvel.seaJooniseKuular(this);
    seaJooniseKuular(tahvel);
```

Väike mõtlemisülesanne: mis juhtuks siis, kui klient oleks enesele ning tahvel enesele jooniskuulariks pandud. Ning ülesande lahendus:

Tahvliga ei juhtuks midagi erilist, sest kui tahvel iseenesele teateid saadaks, siis tahvli peal hiirega joonistatud kujundid tekiksid tahvlile enesele, just nii nagu kasutaja seda ootabki. Kui aga klient asuks võrgust saabunud teateid enesele saatma, siis oleks muresid rohkem. Algul ei pruugiks midagi

hullu juhtuda - seni, kuni keegi pole veel ühtki kujundit teele saatnud. Samuti ei satu kohalikul tahvlil tehtud kujundid võrku juhul, kui tahvel teateid vaid iseenesele saadab. Kui nüüd aga mõne teise kliendi kaudu või lihtsalt kasutaja tippimise peale satuks kasvõi üks kujundi joonistamist nõudev käsk võrku, siis edasi tekiks tsükkel. Serveri ülesandeks on klientidelt saabuvad teated kõikidele klientidele edasi saata. Kui nüüd juhtuks selline klient tekkima, kes võrgust tulnud pilditeated enesele saadaks nii nagu tuleksid need tahvlilt, siis jääks pildikäsk võrku tiirlema. Ikka kliendilt serverile ja tagasi. Ning kui sarnase omadusega kliendi eksemplare oleks serveri küljes mitu, tekiks ahelreaktsioon: iga serveri poolt saadetud pildisoovi peale tuleks igalt kujundit tagasisaatvalt kliendilt teade kujundi kohta. Need saadaks server jälle kõikidele laiali ning mõne aja pärast oleks võrgus korralik kaos.

Et aga tahvel ja klient vastastikku andmeid vahetavad, siis kirjeldatud probleem oli vaid uitmõte ning siinne näide peaks korralikult töötama.

Võrreldes eelmise klientprogrammiga on näha muutujat seisund, näiteks:

```
String seisund="algus";
```

Selle abil määratakse, millises staadiumis rakendus parajasti on. Seisunditeks on veel nimesisestus, paroolisisestus ja tavatekst. Siinne rakendus eeldab, et kasutajalt küsitaks serveri pool nime ja parooli ning alles nende sobivuse korral lastaks võrku suhtlema nagu allpool kirjeldatud serverprogramm teeb. Samas aga on rakendus võimeline ühendust pidama ka eelpool kirjeldatud lihtsama serverprogrammiga, kes kõik ühendujad kohe jutule võtab ning neilt tulnud andmed kogu kuulajaskonnale laiali paiskab. Nagu koodi piiluda ja toimingute järjekordadele mõelda, siis esiotsa eeldatakse, et kasutaja vajutab nupule ühenda, edasi sisestab nime, siis parooli ning ühenduse õnnestumise korral asub teateid saatma ja vastu võtma. Parooli sisestamise ajaks määratakse tekstiväljas nähtavad tähed tärnideks käsu setEchoChar abil. Kui tahta taas tekstiväljas näha kirjutatavaid tähti endid, siis aitab, kui näidatavaks määrata täht koodina 0.

Meetodis run määratakse, et kõik .p-ga algavad read saadetakse tahvlile joonistamiseks, ülejäänud read paigutatakse tekstialasse.

```
String rida=br1.readLine();
if(rida.startsWith(".p")){
  saada(rida);
}else{
  ta1.append(rida+"\n");
}
```

Saatmine iseenesest lihtne. Igaks juhuks kontroll, et tahvel ikka ühendatud on ning edasi lihtsalt kuulaja vastava meetodi väljakutse.

```
void saada(String andmed){
  if(kuulaja==null){return;}
  kuulaja.kujund(andmed);
}
```

Samuti käib lühidalt tahvlilt saabunud teadete edasisaatmine. Ilma mingi täiendava kontrollita saadetakse need lihtsalt PrintWriteri abil serveri poole teele.

```
public void kujund(String andmed){
  pw1.println(andmed);
}
```

main-meetodis veel eelmise kliendiga võrreldes juures käsklus, mis akna sündmustele kuulari lisab.

```
f.addWindowListener(new Raamikuular());
```

Kuular ise paar rida allpool, staatilise sisemise klassina. Ning ainsaks toiminguks tal kogu virtuaalmasina julm sulgemine akna sulgemisristile vajutamise puhul. Tahtes akna sulgemisristi kaudu võrgukliendil paluda serveri küljest viisakamalt väljuda, tuleks teha mõningane ring. Üheks võimaluseks oleks aknakuulamisoskused siduda otse kliendiklassi külge. Sel puhul võiks juba kliendis eneses otsustada, mis sulgemisteate peale teha - küsida kasutajalt täiendavat nõusolekut, saata serverile lõpetusteade või lihtsalt ühendus sulgeda. Selline lähenemine aga eeldaks kliendiklassis kõikide aknaga seotud käskluste realiseerimist olgu siis või tühjade meetoditena.

```
static class Raamikuular extends WindowAdapter{
  public void windowClosing(WindowEvent e){
    System.out.println("Programmi ots");
    System.exit(0);
  }
}
```

Ning kood tervikuna.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;

public class GrKlient2 extends Applet
     implements ActionListener, Runnable, JooniseKuular{
  TextField tf1=new TextField(15); //suurus
  Button nupp1=new Button("Ühenda");
  TextArea ta1=new TextArea(5, 20);
  Panel p1=new Panel(new GridLayout(2, 1)); //alumine
   //2 rida ja 1 veerg tabelis
  Panel p2=new Panel(new GridLayout(2, 1)); //tekst, tahvel
  Tahvel2 tahvel=new Tahvel2();
  Label silt1=new Label();
  PrintWriter pw1;
  BufferedReader br1;
  String seisund="algus";
  String serverinimi="localhost";
  JooniseKuular kuulaja=null;
  public GrKlient2(){
    setLayout(new BorderLayout());
    p1.add(tf1);
    p1.add(silt1);
    add(p1, BorderLayout.SOUTH);
    add(nupp1, BorderLayout.NORTH);
    p2.add(ta1);
    p2.add(tahvel);
    add(p2, BorderLayout.CENTER);
    tf1.addActionListener(this);
    nupp1.addActionListener(this);
    tahvel.seaJooniseKuular(this);
    seaJooniseKuular(tahvel);
  }
  public void seaJooniseKuular(JooniseKuular j){
    kuulaja=j;
  }
  public void actionPerformed(ActionEvent e){
    if(e.getSource()==tf1){
       pw1.println(tf1.getText());
       tf1.setText("");
       if(seisund.equals("nimesisestus")){
         silt1.setText("Palun parool");
         tf1.setEchoChar('*');
         seisund="paroolisisestus";
       } else if(seisund.equals("paroolisisestus")){
         tf1.setEchoChar((char)0);
         silt1.setText("Kirjuta rahus");
         seisund="tavatekst";
       }

    }
    if(e.getSource()==nupp1){
      try{
         Socket sc=new Socket(serverinimi, 3001);
         pw1=new PrintWriter(sc.getOutputStream(), true);
         br1=new BufferedReader(new
           InputStreamReader(sc.getInputStream()));
         new Thread(this).start();
         silt1.setText("Palun nimi: ");
         seisund="nimesisestus";
      }catch(Exception viga){
        ta1.setText(viga.getMessage());
      }
    }
  }

  public void run(){
   try{
    while(true){
      String rida=br1.readLine();
      if(rida.startsWith(".p")){
        saada(rida);
      }else{
        ta1.append(rida+"\n");
      }
    }
   }catch(Exception viga){
     viga.printStackTrace();
     ta1.append("Oled väljas");
```

```
   }
  }
  void saada(String andmed){
    if(kuulaja==null){return;}
    kuulaja.kujund(andmed);
  }
  public void kujund(String andmed){
    pw1.println(andmed);
  }

  public static void main(String[] argumendid){
     Frame f=new Frame();
     f.add(new GrKlient2());
     f.setSize(300, 300);
     f.setVisible(true);
     f.addWindowListener(new Raamikuular());
  }
  static class Raamikuular extends WindowAdapter{
    public void windowClosing(WindowEvent e){
      System.out.println("Programmi ots");
      System.exit(0);
    }
  }
}
```

Registreeritud kasutajatega server.

Järgnevalt algsest näitest mõnevõrra arukam serverprogramm. Sisenejatelt küsitakse kasutajanime ja parooli. Uue nime puhul teatatakse uuest kasutajast, tuttava kombinatsiooni puhul rõõmustatakse jällenägemise puhul ning olemasoleva kasutajanime kuid vigase parooliga meldimise korral sisse ei lasta. Andmed püsivad mälus küll vaid serveri tööajal, kuid faili kirjutamine ja sealt lugemine on siit puudu vaid näite lühiduse ettekäändel. Soovides andmeid püsivalt kettal talletada, oleks mõistlik need programmi käivitumisel mällu (HashMap-i) lugeda ning iga uue kasutaja lisandumisel vastav rida faili juurde kirjutada. Failioperatsioonid võiksid olla sünkroniseeritud nii nagu kasutajate loetelu puhul näha on. Sel juhul pole vaja peljata, et mitme lõime üheaegne failikirjutussoov andmeid rikkuda võiks.

Andmete hoidmiseks siis kaks kogu programmi piires kasutatavat andmestruktuuri. Üks kasutajate lõimede jaoks, teine nimede ja paroolide tarbeks. Mõlema operatsioonid on määratud sünkroniseerituks, et lõimed korraga samu andmeid muutma ei asuks ning et ei asutaks küsima kasutaja andmeid, keda enam loetelus pole.

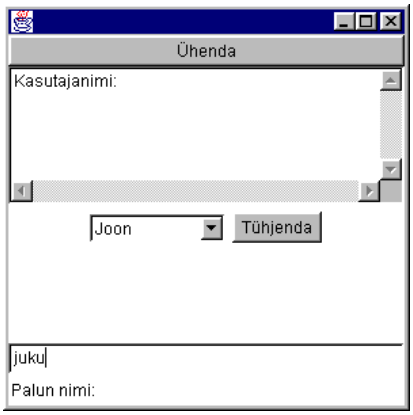
```
  static List kasutajad=Collections.synchronizedList(new LinkedList());
  static Map paroolid=Collections.synchronizedMap(new HashMap());
```

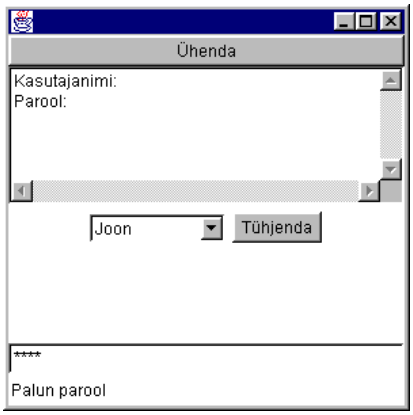
Võrreldes eelpool oleva serverinäitega ei hoita siin loetelus mitte kasutajate pistikuid, vaid eraldi objekte, kuhu on koondatud mitmed andmed kasutaja kohta. Et igal kasutajal on PrintWriter valja temale andmete saatmiseks, siis pole enam kasutajale andmete saatmiseks vaja igal korral pistikust väljundvoogu küsida, vaid sellele võib vastava muutuja kaudu kohe juurde pääseda.

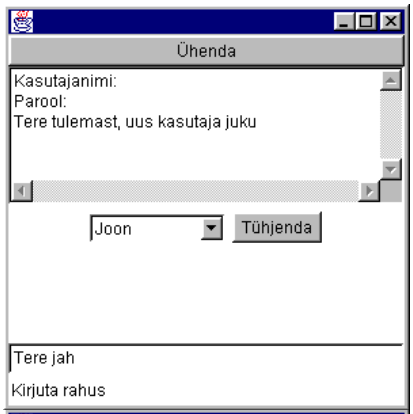
```
       synchronized(kasutajad){
           Iterator loend=kasutajad.iterator();
           while(loend.hasNext()){
             Kasutaja k=(Kasutaja)loend.next();
             k.valja.println(rida);
           }
         }
      }
```

Kuna kasutajate loend on sünkroniseeritud ning kasutajatele andmete trükkimise tsükkel vastava loendi järgi sünkroniseeritud plokis, siis ei peaks saama kasutajaid trükkimise ajal lisada ning eemaldada.

Et kasutajale loodud klassi eksemplar lisatakse vektorisse alles pärast nime ja parooli küsimist, siis ei hakka ta ka enne muudelt inimestelt saabuvaid teateid saama, kui nime ja parooli sobivus kontrollitud.








```
import java.io.*;
import java.net.*;
import java.util.*;
public class ParooligaJututuba{
  static List kasutajad=
     Collections.synchronizedList(new LinkedList());
  static Map paroolid=
     Collections.synchronizedMap(new HashMap());
  public static void main(String argumendid[])
                              throws IOException{
    ServerSocket ss=new ServerSocket(3001);
    while(true){
      new Kasutaja(ss.accept());
    }
  }


  static class Kasutaja extends Thread{
    Socket sc;
    PrintWriter valja;
    String kasutajanimi="";
    public Kasutaja(Socket uus_sc){
      sc=uus_sc;
      start();
    }
    public void run(){
     try{
      BufferedReader sisse=new BufferedReader(
        new InputStreamReader(sc.getInputStream())
      );
      valja=new PrintWriter(
                    sc.getOutputStream(), true);
      valja.println("Kasutajanimi: ");
      kasutajanimi=sisse.readLine();
      valja.println("Parool: ");
      String parool=sisse.readLine();
        //tavaline readLine ei suuda märke peita.
      if(paroolid.get(kasutajanimi)==null){
        valja.println(
           "Tere tulemast, uus kasutaja "+
            kasutajanimi);
        paroolid.put(kasutajanimi, parool);
      } else if(paroolid.get(kasutajanimi).
                                  equals(parool)){
        valja.println("Tere taas, "+kasutajanimi);
      } else {
        valja.println("Vigane meldimine.");
        sc.close();
        return;
      }
      kasutajad.add(this);
      boolean veel=true;
      while(veel){
         String rida=sisse.readLine();
         System.out.println(rida);
             //administraatori tarbeks
         if(rida.startsWith(".ots")){veel=false;}
         synchronized(kasutajad){
           Iterator loend=kasutajad.iterator();
           while(loend.hasNext()){
             Kasutaja k=(Kasutaja)loend.next();
             k.valja.println(rida);
           }
         }
      }
      sc.close();
    } catch(Exception e){
      System.out.println("Probleem: "+e);
    }
    kasutajad.remove(this);
  }
 }
}
```

```
D:\arhiiv\konspektid\grklient>java ParooligaJututuba
.pr 94 62 33 21
.pr 113 48 8 14
.pr 112 70 12 13
.pr 99 70 7 7
```

Joonistasin sulle maja pildi

Väljund serveriaknas

Kolmas arendusring.

Järgnevalt on tahvli koodi täiendatud javadoc-ile sobivate kommentaaridega. Nii õnnestub ülevaatlikkuse tarbeks genereerida koodi kohta mõningane dokumentatsioon ilma selle jaoks eraldiseisvat juttu kirjutamata. Joonistussündmustega ümber käimiseks on lihtne JooniseKuular-liidese tüüpi muutuja asemele paigutatud nimistu jooniseKuularid, mis võib eneses hoida kuulajaid nõnda palju kui parajasti tarvilik on.

```
LinkedList jooniseKuularid=new LinkedList();
```

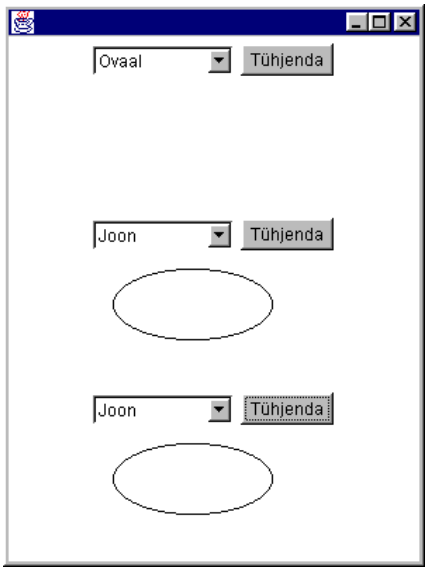
Et kannataks tahvlile väljapoolt kuulajaid külge panna, selleks alljärgnev meetod. Töötab teine samal põhimõttel, kui näiteks nupu või tekstivälja puhul addActionListener. Ehk sündmuse allikaks olev objekt jätab enese juurde meelde, kuhu tuleb sündmuse toimumise puhul teated välja saata.

```
public void lisaJooniseKuular(JooniseKuular j){
  jooniseKuularid.add(j);
}
```

Saatmise puhul tuleb hoolitseda, et teade kõikide sooviavaldanuteni jõuaks. Kui ennist oli valida kas nulli või ühe kuulaja vahel, siis nüüd alampiiriks endiselt null, ülempiiri pole aga seatud. Tõsi küll, mõni klass võib liiga suure kuulajaks registreerunute arvu puhul anda TooManyListenersException'i. Nagu ikka, on korduvate tegevuste puhul abiks tsükkel. Nimistust soovitud järjekorranumbriga elemendi aitab kätte saada get. Et oleme nimistusse paigutanud vaid JooniseKuulareid, siis võime ka kindlad olla, et sama tüüpi elemente sealt kätte saame.

```
void saada(){
 if(jooniseKuularid.size()==0){return;}
 int laius=hyx-hax;
 int korgus=hyy-hay;
 for(int i=0; i<jooniseKuularid.size(); i++){
  JooniseKuular kuulaja=(JooniseKuular)jooniseKuularid.get(i);
....
  if(kujundivalik.getSelectedItem().equals("Ovaal")){
    kuulaja.kujund(".po "+hax+" "+hay+" "+laius+" "+korgus);
  }
 }
}
```

Ehkki jututoa kliendi rakenduses piisab siinsele tahvlile vaid ühest kuulajast - kliendist, mille kaudu andmed võrku saadetakse kannatab tahvlile külge panna näiteks teisi tahvleid, nagu allpool olevas näites näha on. Tahvli t1 sündmustele reageerivad nii t2 kui t3.




```
import java.awt.*;
public class Tahvel3KuulariDemo{
  public static void main(String[] argumendid){
    Tahvel3 t1=new Tahvel3();
    Tahvel3 t2=new Tahvel3();
    Tahvel3 t3=new Tahvel3();
    Frame f=new Frame();
    f.setLayout(new GridLayout(3, 1));
    f.add(t1);
    f.add(t2);
    f.add(t3);
    t1.lisaJooniseKuular(t2);
    t1.lisaJooniseKuular(t3);
    f.setSize(300, 400);
    f.setVisible(true);
  }
}
```

Tahvlile lisati nupp.

```
Button tyhjenda=new Button("Tühjenda");
```

Nagu nimigi näitab, on see loodud ala puhastuseks, et õnnestuks soovi korral taas valgelt lehelt alustada. Tühjenduseks piisab vaid hoidlas olevate teadete kaotamisest - siis järgnev repaint vaid kustutab platsi taustaga ühte värvi, kuid midagi vaadatavat ei lisa.

```
public void actionPerformed(ActionEvent e){
  if(e.getSource()==tyhjenda){
    hoidla.clear();
    repaint();
  }
}
```

Et õnnestuks tervet pildi sisu kas kopeerida või arhiveerida, selleks juures vastav meetod. Hoidla enese osutit ei väljastata seetõttu, et kui osuti kätte saanud klass asuks hoidla sisu muutma, siis võiks see kergesti siinse pildi segamini keerata. Kui aga väljastatakse koopia, siis vastavat ohtu pole. Ehkki ka koopiahoidlasse jäävad osutid algsetele sõnedele ja mitte nende koopiatele, siis kuna klassi String eksemplari väärtust pole võimalik pärast selle loomist muuta, pole karta andmete muutumist algses hoidas.

```
public LinkedList hoidlaKoopia(){
  return new LinkedList(hoidla);
}
```

Eelmisega võrreldes vastandlik käsk: pildile saab ette anda uue sisu kollektsioonina. Nagu käskudest näha, tehakse uue sisu määramisel tahvel vanadest andmetest puhtaks ning lisatakse kõik, mis etteantust võtta on.

```
/**
* Uus hoidla sisu ja ekraanipilt.
*/
public void hoidlaUusSisu(Collection c){
  hoidla.clear();
  hoidla.addAll(c);
  repaint();
}
```

Eelnevad kaks käsku üheskoos võimaldavad ühe pildi sisu teisele kopeerida. Ühe tahvlilt küsitud andmed antakse ilusti teisele ette nagu järgnevas kahe tahvliga jututoa kliendi näites.

```
  if(e.getSource()==kopeeriPilt){
    tahvelOma.hoidlaUusSisu(tahvel.hoidlaKoopia());
  }
```

Ning laiendatud tahvli kood tervikuna.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Tahvel3 extends Applet
  implements MouseListener, MouseMotionListener, JooniseKuular, ActionListener{
  /**
  *  Kujundi alguse x (hiir alla)
  */
  int hax;
  /**
  *  Kujundi alguse y
  */
  int hay;
  /**
  * Kujundi lõpu x
  */
  int hyx;
  /**
  * Kujundi lõpu y
  */
  int hyy;
  /**
  * Joonistatavate kujundite loetelu.
  */
  String[] kujundid={"Joon", "Ristkülik", "Ovaal", "Vabakäejoon"};
  /**
  * Valikukombo.
  */
  Choice kujundivalik=new Choice();
  /**
  * Kujunditeadete püüdjate loend. Võivad olla nii tahvel ise, teine
  * tahvel, GrKlient kui mõni muu JooniseKuularit realiseeriva
  * klassi eksemplar.
  */
  LinkedList jooniseKuularid=new LinkedList();
  /**
  * Joonistamiseks meeles peetavad kujundid
  */
  LinkedList hoidla=new LinkedList();
  /**
  * Hoidla tühjendus, pildi puhastus.
  */
  Button tyhjenda=new Button("Tühjenda");

  /**
  * Initsialiseerimine
  */
  public Tahvel3(){
    addMouseListener(this);
    addMouseMotionListener(this);
    for(int i=0; i<kujundid.length; i++){
      kujundivalik.addItem(kujundid[i]);
    }
    add(kujundivalik);
    add(tyhjenda);
    tyhjenda.addActionListener(this);
  }

  /**
  * Kujundite kuulaja lisamine. Korraga suudab teateid
  * saata vaid ühele kuulajale.
  */
  public void lisaJooniseKuular(JooniseKuular j){
    jooniseKuularid.add(j);
  }

  /**
  * Joonis hoidla põhjal.
  */
  public void paint(Graphics g){
     for(Iterator it=hoidla.iterator(); it.hasNext();){
       String s=(String)it.next();
       joonista(s);
     }
  }

  /**
  * Ühe saabunud kujundi lahkamine sõnest ja joonistus.
```

```
*/
void joonista(String andmed){
 try{
  Graphics g=getGraphics();
  StringTokenizer stk=new StringTokenizer(andmed);
  String tyyp=stk.nextToken();
  int a[]=new int[4];
  for(int i=0; i<4; i++){
     a[i]=Integer.parseInt(stk.nextToken());
  }
  if(tyyp.equals(".pj")){
    g.drawLine(a[0], a[1], a[2], a[3]);
  }
  if(tyyp.equals(".pr")){
    g.drawRect(a[0], a[1], a[2], a[3]);
  }
  if(tyyp.equals(".po")){
    g.drawOval(a[0], a[1], a[2], a[3]);
  }
 }catch(Exception viga){viga.printStackTrace();}
}

/**
* Saabuv kujund
*/
public void kujund(String andmed){
  joonista(andmed);
  hoidla.add(andmed);
}

/**
* Salvestatakse hiire allavajutuse koordinaadid.
*/
public void mousePressed(MouseEvent e){
  hax=e.getX();
  hay=e.getY();
}

/**
* Salvestatakse hiire üleslaskmise koordinaadid. Vajadusel
* luuakse kujund.
*/
public void mouseReleased(MouseEvent e){
  hyx=e.getX();
  hyy=e.getY();
  saada();
}

/**
* Tühjendusnupule reageerimine.
*/
public void actionPerformed(ActionEvent e){
  if(e.getSource()==tyhjenda){
    hoidla.clear();
    repaint();
  }
}

/**
* Lohistamine. Vähemasti vabakäejoone tarbeks.
*/
public void mouseDragged(MouseEvent e)
{
  if(kujundivalik.getSelectedItem().equals("Vabakäejoon")){
      hyx = e.getX();
      hyy = e.getY();
      saada();
      hax = hyx;
      hay = hyy;
  }
}

/**
* Tühi meetod, vajalik liikumisliidese realiseerimiseks.
*/
public void mouseMoved(MouseEvent e){}


/**
* Andmete väljastus jooniseKuulajale. Kuulaja puudumise
* korral ei väljastata midagi.
*/
void saada(){
```

```
   if(jooniseKuularid.size()==0){return;}
   int laius=hyx-hax;
   int korgus=hyy-hay;
   for(int i=0; i<jooniseKuularid.size(); i++){
    JooniseKuular kuulaja=(JooniseKuular)jooniseKuularid.get(i);
    if(kujundivalik.getSelectedItem().equals("Joon")||
       kujundivalik.getSelectedItem().equals("Vabakäejoon")){
      kuulaja.kujund(".pj "+hax+" "+hay+" "+hyx+" "+hyy);
    }
    if(kujundivalik.getSelectedItem().equals("Ristkülik")){
      kuulaja.kujund(".pr "+hax+" "+hay+" "+laius+" "+korgus);
    }
    if(kujundivalik.getSelectedItem().equals("Ovaal")){
      kuulaja.kujund(".po "+hax+" "+hay+" "+laius+" "+korgus);
    }
   }
  }

  /**
  * Olemasolevate kujundite koopia. Muutumatu sisuga loetelu
  * joonistusaja tarbeks, samuti kogu pildi teele saatmiseks.
  */
  public LinkedList hoidlaKoopia(){
    return new LinkedList(hoidla);
  }

  /**
  * Uus hoidla sisu ja ekraanipilt.
  */
  public void hoidlaUusSisu(Collection c){
    hoidla.clear();
    hoidla.addAll(c);
    repaint();
  }

  /**
  * Tühi meetod, vajalik hiireliidese realiseerimiseks.
  */
  public void mouseClicked(MouseEvent e){}
  /**
  * Tühi meetod, vajalik hiireliidese realiseerimiseks.
  */
  public void mouseEntered(MouseEvent e){}
  /**
  * Tühi meetod, vajalik hiireliidese realiseerimiseks.
  */
  public void mouseExited(MouseEvent e){}
  /**
  * Võimaldab tahvlit iseseisvalt joonistuslõuendina käivitada.
  * Joonistab iseendale.
  */
  public static void main(String[] argumendid){
    Frame f=new Frame();
    Tahvel3 t=new Tahvel3();
    t.lisaJooniseKuular(t);
    f.add(t);
    f.setSize(300, 300);
    f.setVisible(true);
  }
}
```

Lühidokumentatsioon

Javadoci abil vormistatult näeb tahvli väljade ja meetodite dokumentatsioon välja järgmine. Esmasel vaatamisel peaks siit olema kergem klassi käsklused üles leida.

Väljad	
(package private) int	**hax** Kujundi alguse x (hiir alla)
(package private) int	**hay** Kujundi alguse y
(package private) java.util.LinkedList	**hoidla** Joonistamiseks meeles peetavad kujundid
(package private) int	**hyx** Kujundi lõpu x

(package private) int	**hyy** Kujundi lõpu y
(package private) java.util.LinkedList	**jooniseKuularid** Kujunditeadete püüdjate loend.
(package private) java.lang.String[]	**kujundid** Joonistatavate kujundite loetelu.
(package private) java.awt.Choice	**kujundivalik** Valikukombo.
(package private) java.awt.Button	**tyhjenda** Hoidla tühjendus, pildi puhastus.

Konstruktor

Tahvel3()	
Initsialiseerimine	

Meetodid

void	actionPerformed(java.awt.event.ActionEvent e) Tühjendusnupule reageerimine.
java.util.LinkedList	hoidlaKoopia() Olemasolevate kujundite koopia.
void	hoidlaUusSisu(java.util.Collection c) Uus hoidla sisu ja ekraanipilt.
(package private) void	joonista(java.lang.String andmed) Ühe saabunud kujundi lahkamine sõnest ja joonistus.
void	kujund(java.lang.String andmed) Saabuv kujund
void	lisaJooniseKuular(JooniseKuular j) Kujundite kuulaja lisamine.
static void	main(java.lang.String[] argumendid) Võimaldab tahvlit iseseisvalt joonistuslõuendina käivitada.
void	mouseClicked(java.awt.event.MouseEvent e) Tühi meetod, vajalik hiireliidese realiseerimiseks.
void	mouseDragged(java.awt.event.MouseEvent e) Lohistamine.
void	mouseEntered(java.awt.event.MouseEvent e) Tühi meetod, vajalik hiireliidese realiseerimiseks.
void	mouseExited(java.awt.event.MouseEvent e) Tühi meetod, vajalik hiireliidese realiseerimiseks.
void	mouseMoved(java.awt.event.MouseEvent e) Tühi meetod, vajalik liikumisliidese realiseerimiseks.
void	mousePressed(java.awt.event.MouseEvent e) Salvestatakse hiire allavajutuse koordinaadid.
void	mouseReleased(java.awt.event.MouseEvent e) Salvestatakse hiire üleslaskmise koordinaadid.
void	paint(java.awt.Graphics g) Joonis hoidla põhjal.
(package private) void	saada() Andmete väljastus jooniseKuulajale.

Kahe tahvliga klient.

Võrdlusseeria lõpetuseks juba rohkem ametliku väljanägemisega klient. Kasutajal tuleb määrata ühendumiseks vajalik server ja värat, samuti kasutajanimi ja parool. Edasi ekraan tühjendatakse ning sinna paigutatakse juba suhtlemiseks vajalikud vahendid: tekstiväli kirjutamiseks, tekstiala teadete vastuvõtmiseks ning kaks tahvlit: üks joonistamiseks ja andmete teele saatmiseks, teine võrgu pealt saabuva vaatamiseks.

Nagu koodist näha, näitab nupule nupp1 vajutus, et tuleb asuda ühendust võtma ning seejärel ekraan ümber kujundada. Serverist saabuvaid andmeid kontrollitakse ning kui esimesena ei küsita kasutajanime, siis pole satutud oodatud serveri otsa, on tegemist vigase protokolliga ning väljastatakse erind.

```
new Thread(this).start();
```

Lükkab tööle teadete püüdmiseks mõeldud lõime.

Elementide vahetamiseks eemaldatakse kõigepealt kõik ekraanile paigutatu käsuga

```
removeAll();
```

Edasi säetakse nii tahvlid, nupud kui tekstikastid paika ning lõpetuseks öeldakse

```
validate();
```

mis peaks hoolitsema, et ekraanile paigutatu ka kõik ilusti välja näidataks.

```
if(e.getSource()==nupp1){
  try{
     Socket sc=new Socket(tf3.getText(), Integer.parseInt(tf4.getText()));
     pw1=new PrintWriter(sc.getOutputStream(), true);
     br1=new BufferedReader(new
       InputStreamReader(sc.getInputStream()));
     if(!br1.readLine().equals("Kasutajanimi: ")){
       throw new IOException("Vigane protokoll");
     }
     pw1.println(nimesisestus.getText());
     br1.readLine(); //eeldatavasti küsitakse parooli
     pw1.println(tf2.getText());
     tf2.setText("");
     new Thread(this).start();
     removeAll();
     //Aken puhtaks ning uus paigutus
     setLayout(new BorderLayout());
...
     validate();
  }catch(Exception viga){
    nimesisestus.setText(viga.getMessage());
  }
}
```

Andmete võrku saatmise nupule on antud programmis kaks ülesannet. Kui ühendus olemas, siis nupule vajutades küsitakse tahvli peal asuv joonis ning sealsed teated saadetakse ükshaaval võrku, et ka ülejäänud vestlusel osalejad saaksid pilti näha.

Kui aga ühendus juhtub katkema, siis määratakse nupu peal olevaks tekstiks "Alusta" ning nupule vajutusel manatakse kasutaja ette algpaigutus, kus kasutajal on võimalik määrata, millise serveriga ja millisesse väratisse ühendust võtta. Nõnda ühe nupuga piirdudes pole vaja kujundust muutma asuda. Mõnel sarnasel klientprogrammil kipub kombeks olema kohe ühenduse katkemisel sissemeldimisaken ette visata, kuid see ei tundu meeldiva lahendusena, sest nõnda pole võimalik enam eelnenud teksti üle vaadata. Nupu pealkirja järgi reageerimise miinuseks oleks aga võimatus kergesti silti muuta või tõlkida. Eraldi muutuja kasutamisel sellist probleemi pole.

```
if(e.getSource()==piltVorku){
 if(yhendusOlemas){
  Iterator it=tahvelOma.hoidlaKoopia().iterator();
  while(it.hasNext()){
    pw1.println(it.next());
  }
 } else {
   removeAll();
   algpaigutus();
   validate();
   ta1.setText("");
   nimesisestus.setText("");
   piltVorku.setLabel("Võrku");
 }
}
```

Kui võrgust lugemisel tekib viga või kui ühendus katkestatakse, sel juhul satub kood while-tsükli seest katsendiploki veatöötlusossa, kus jäetakse meelde ühenduse katmine ning vahetakse nupul pealkiri et kasutaja teaks sellel vajutades otsast alustada.

```
public void run(){
 try{
```

```
    while(true){
      yhendusOlemas=true;
      String rida=br1.readLine();
      if(rida==null){
        throw new IOException("Ühenduse lõpp");
      }
      if(rida.startsWith(".p")){
        saada(rida);
      }else{
        ta1.append(rida+"\n");
      }
    }
   }catch(Exception viga){
     yhendusOlemas=false;
     piltVorku.setLabel("Alusta");
     ta1.append("Oled väljas");
   }
  }
```

Muud vahendid sarnased kui eelneval kliendil.

Vahepalaks Javadociga välja eraldatud käsklused koos kommentaaridega.

Meetodid	
void	actionPerformed(java.awt.event.ActionEvent e) Sündmustele reageerimise keskus.
void	algpaigutus() Paigutus, kus on näha sisenemiseks tarvilikud tekstiväljad.
void	kujund(java.lang.String andmed) Parameetrina saadavad andmed saadetakse võrku edasi.
static void	main(java.lang.String[] argumendid) Käivitus.
void	run() Teateid püüdva lõime käsud.
(package private) void	saada(java.lang.String andmed) Käsklus pildikujundi (edasi) saatmiseks (eeldatavalt tahvlile) juhul kui vastuvõtja on olemas.
void	seaJooniseKuular(JooniseKuular j) Määratakse, millisele objektile saadetakse edasi kliendile saabunud teated pildile paigutatavate kujundite kohta.

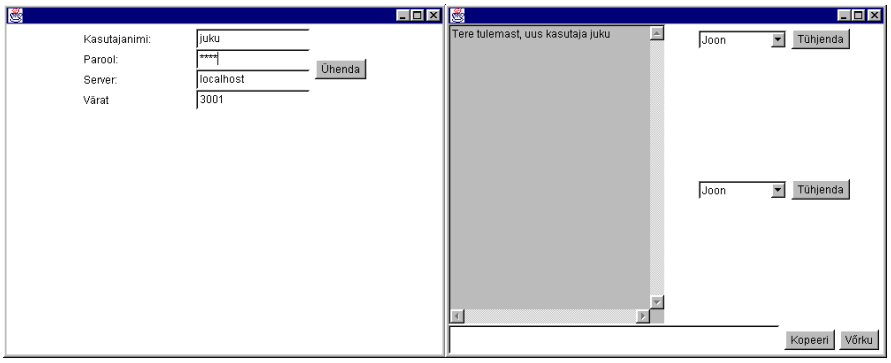
Ning edasi kahe tahvliga kliendi kood.

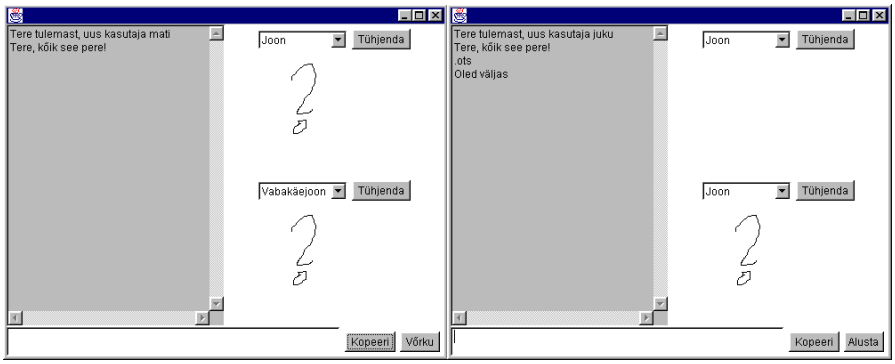
```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import java.util.*;

public class GrKlient3 extends Applet
     implements ActionListener, Runnable, JooniseKuular{
  /**
  * Tekstiväli andmete võrku saatmiseks.
  */
  TextField tf1=new TextField(15);
  /**
  * Nimesisestuskoht.
  */
  TextField nimesisestus=new TextField(15);
  /**
  * Paroolisisestusväli.
  */
  TextField tf2=new TextField(15);
  /**
  * Serveri nime tarvis.
  */
  TextField tf3=new TextField("localhost", 15);
  /**
  * Serveri värati number.
  */
  TextField tf4=new TextField("3001", 15);
  /**
```

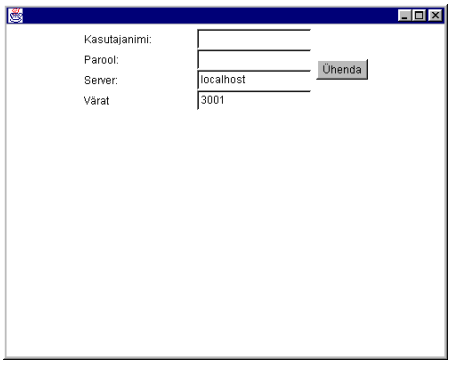
```
* Vajutuse peale ühendatakse klient serveriga.
*/
Button nupp1=new Button("Ühenda");
/**
* Serverist saabuvad andmed.
*/
TextArea ta1=new TextArea(5, 20);
/**
* Võrgust saabuv joonis, võimalus otse võrku joonistada.
*/
Tahvel3 tahvel=new Tahvel3();
/**
* Rahus omaette joonistamiseks. Hiljem võimalik kopeerida.
*/
Tahvel3 tahvelOma=new Tahvel3();
/**
* Pilt võrgutahvlilt oma tahvlile.
*/
Button kopeeriPilt=new Button("Kopeeri");
/**
* Pilt oma tahvlilt võrku.
*/
Button piltVorku=new Button("Võrku");
/**
* Voog võrku saatmiseks.
*/
PrintWriter pw1;
/**
* Voog võrgust lugemiseks.
*/
BufferedReader br1;
/**
* Kuular kujundite saatmiseks. Siin juhul liiguvad
* võrgust saabuvad andmed avalikule tahvlile.
*/
JooniseKuular kuulaja=null;
/**
* Konstruktor kujunduse sättimiseks ning teadete saatmise seadmiseks.
*/
boolean yhendusOlemas=false;
/**
* Kuularite paika sättimine.
*/
public GrKlient3(){
  algpaigutus();
  nupp1.addActionListener(this);
  tf2.setEchoChar('*');
  tf1.addActionListener(this);
  kopeeriPilt.addActionListener(this);
  piltVorku.addActionListener(this);
  tahvel.lisaJooniseKuular(this);
  tahvelOma.lisaJooniseKuular(tahvelOma);
  seaJooniseKuular(tahvel);
  ta1.setEditable(false);
}

/**
* Paigutus, kus on näha sisenemiseks tarvilikud tekstiväljad.
*/
public void algpaigutus(){
  setLayout(new FlowLayout());
  Panel p1=new Panel(new GridLayout(4, 2));
  p1.add(new Label("Kasutajanimi:"));
  p1.add(nimesisestus);
  p1.add(new Label("Parool:"));
  p1.add(tf2);
  p1.add(new Label("Server:"));
  p1.add(tf3);
  p1.add(new Label("Värat"));
  p1.add(tf4);
  add(p1);
  add(nupp1);
}
/**
*  Määratakse, millisele objektile saadetakse edasi kliendile saabunud
*  teated pildile paigutatavate kujundite kohta.
*/
public void seaJooniseKuular(JooniseKuular j){
  kuulaja=j;
}
/**
* Sündmustele reageerimise keskus.
*/
```

```
public void actionPerformed(ActionEvent e){
  if(e.getSource()==tf1){
     pw1.println(tf1.getText());
     tf1.setText("");
  }
  if(e.getSource()==nupp1){
    try{
       Socket sc=new Socket(tf3.getText(), Integer.parseInt(tf4.getText()));
       pw1=new PrintWriter(sc.getOutputStream(), true);
       br1=new BufferedReader(new
         InputStreamReader(sc.getInputStream()));
       if(!br1.readLine().equals("Kasutajanimi: ")){
         throw new IOException("Vigane protokoll");
       }
       pw1.println(nimesisestus.getText());
       br1.readLine(); //eeldatavasti küsitakse parooli
       pw1.println(tf2.getText());
       tf2.setText("");
       new Thread(this).start();
       removeAll();
       //Aken puhtaks ning uus paigutus
       setLayout(new BorderLayout());
       Panel kesk=new Panel(new GridLayout(1, 2));
       kesk.add(ta1);
       Panel tahvlid=new Panel(new GridLayout(2, 1));
       tahvlid.add(tahvelOma);
       tahvlid.add(tahvel);
       kesk.add(tahvlid);
       add(kesk, BorderLayout.CENTER);
       Panel nupud=new Panel();
       nupud.add(kopeeriPilt);
       nupud.add(piltVorku);
       Panel alumine=new Panel(new BorderLayout());
       alumine.add(tf1, BorderLayout.CENTER);
       alumine.add(nupud, BorderLayout.EAST);
       add(alumine, BorderLayout.SOUTH);
       tf1.setText("");
       validate();
    }catch(Exception viga){
      nimesisestus.setText(viga.getMessage());
    }
  }
  if(e.getSource()==kopeeriPilt){
    tahvelOma.hoidlaUusSisu(tahvel.hoidlaKoopia());
  }
  if(e.getSource()==piltVorku){
   if(yhendusOlemas){
    Iterator it=tahvelOma.hoidlaKoopia().iterator();
    while(it.hasNext()){
      pw1.println(it.next());
    }
   } else {
     removeAll();
     algpaigutus();
     validate();
     ta1.setText("");
     nimesisestus.setText("");
     piltVorku.setLabel("Võrku");
   }
  }
}

/**
*  Teateid püüdva lõime käsud. Võrgust saabuvad andmed paigutatakse
*  vastavalt algusele kas teatena tekstialasse või saadetakse
*  kujundina pildile.
*/
public void run(){
 try{
  while(true){
    yhendusOlemas=true;
    String rida=br1.readLine();
    if(rida==null){
      throw new IOException("Ühenduse lõpp");
    }
    if(rida.startsWith(".p")){
      saada(rida);
    }else{
      ta1.append(rida+"\n");
    }
  }
 }catch(Exception viga){
   yhendusOlemas=false;
   piltVorku.setLabel("Alusta");
```

```
      ta1.append("Oled väljas");
    }
   }

   /**
   * Käsklus pildikujundi (edasi) saatmiseks (eeldatavalt tahvlile) juhul kui
vastuvõtja on olemas.
   */
   void saada(String andmed){
     if(kuulaja==null){return;}
     kuulaja.kujund(andmed);
   }

   /**
   *  Parameetrina saadavad andmed saadetakse võrku edasi. Vajalik
   *  ka JooniseKuulari liidese realiseerimiseks.
   */
   public void kujund(String andmed){
     pw1.println(andmed);
   }

   /**
   * Käivitus. Luuakse raamaken ning paigutatakse kliendi eksemplar sellesse.
   */
   public static void main(String[] argumendid){
      Frame f=new Frame();
      f.add(new GrKlient3());
      f.setSize(500, 400);
      f.setVisible(true);
      f.addWindowListener(new Raamikuular());
   }

   /**
   * Akna sulgemine ristile vajutusel.
   */
   static class Raamikuular extends WindowAdapter{
     public void windowClosing(WindowEvent e){
       System.out.println("Programmi ots");
       System.exit(0);
     }
   }
}
```







Loetud kolme keerukustaseme juures pole läbi proovitud sootukski kõik võrgurakendustega seotud võimalused, kuid siinse aluse peale peaks andma ehitada enamiku lahendustest, mis tellijal või programmeerija mõttesse saaksid tulla. Ikka tuleb välja mõelda mõningane protokoll andmete saatmiseks ning kanali kummassegi otsa rakendus saabuvate teadete tekitamiseks ja töötlemiseks. Täiesti lubatud on andmeid saata binaarformaadis, ainult et sel juhul on liikluse kontroll veidi keerulisem. Ka liikumise või muusika kannatab võrgurakendusele külge ehitada. Võrk on lihtsalt üks täiendav vahend programmile andmete saatmiseks ja sealt vastuvõtuks.

Ülesandeid

Paigutatud kasutajatega jututuba

- Hiirevajutuse koordinaadid saadetakse jututoa serverisse kujul kasutajanimi x y
- Programmis paigutatakse ekraanile saabunud kasutajanimi sellega kaasas olevatele koordinaatidele.
- Iga kasutaja viimase hiirevajutuse asukohal on klientprogrammis näha tema nimi.
- Lisaks oma asukohale saab üle kanda ka tavalist teksti. Võrguliikluses on sellise rea ees hüüumärk.

Laevade pommitamine

- Serverprogramm mõtleb numbri ühest kümneni. Kasutaja võtab ühendust ning pakub numbri. Programm teatab, kas number pakuti õigesti või valesti.
- Serveriga saavad ühendust võtta kaks kasutajat. Nad võivad hakata teineteisele kordamööda andmeid saatma (nagu käike males või kuule laevade pommitamisel)
- Kaks kasutajat saavad graafilise liidese abil laevade pommitamist mängida. Kumbki kasutaja saab algul hiire abil oma laevade asukohad märkida. Siis saavad kasutajad näidata, kuhu hiirega lasta, ülejäänud töö teeb arvuti mängijate eest ära.

Rändurid võrgumaastikul

- Kaks kasutajat saavad võrgu kaudu saata teineteisele oma koordinaate.
- Kummagi kasutaja asukoht on näha ekraanil. Kumbki saab klahvide abil oma asukohta muuta.
- Mängu alustamisel on mõlemad kasutajad vasakul alumises nurgas. Võidab see, kes jõuab rutem ümber akna keskel paiknevate tõkete paremasse ülemisse nurka.

Liiklus võrgus

- Iga kasutaja saab oma sõiduvahendit liigutada.
- Lisaks eelmisele peavad sõidukid liikuma mööda teid ning ei tohi sattuda üksteise peale.
- Teedel on eesõigusmärgid ning valgusfoorid eesõigusi seadmas, eeskirju rikkuda ei saa.

Juhitav animatsioon

Joonistuskäsud, lõimed, mälupilt, heli

Taustateave

Joonistamine

Javas on püütud joonistusvahendid teha sihtpinnast sõltumatuks, nii et sama programmilõigu abil võib joonistada nii ekraanile, mällu kui printerisse ning sobiva draiveri abil ka muude vahendite abil. Programmi poolt saadetakse joonistatavale pinnale teateid graafilise konteksti abil, milleks on enamasti klassi Graphics või Graphics2D oskustega isend. Kontekstile antakse üldisi käsklusi joone, ringi, teksti jm. joonistamise kohta, viimase ülesandeks on tulemus kanda üle sihtpinnale. Kas tegemist on mälupildi joonistamisel massiivi väärtuste arvutamisega või põletuspulga liigutamisel plotteris, sellele ei pea programm tähelepanu pöörama. Programmi kirjutaja isegi ei pea teadma, millise sihtseadmeni tema käskude mõju lõpuks ulatub. Et iga lõigu või punkti joonistamisel ei peaks eraldi kaasa lisama joonistamisel vajalikke parameetreid, hoitakse neid graafilise konteksti juures ning koos joonistuskäsklustega edastatakse sihtseadmele. Graphics hoiab eneses joonistusvärvi, -piirkonda ning nihet, Graphics2D juures aga tulevad juurde joone laius, pildi keere, kalle, suurendus, joonte ühendus. Samuti võib joonistusvärviks valida mustri või värviülemineku. Sellisel juhul arvutades mälus asuvale pildile joonistatavat ringi iga punkti puhul vaadatakse, milline koht mustrist vastab joone alla jäävale kohale sihtpinnal.

Välja paistva graafikakomponendi joonistuskäsud koondatakse üldiselt meetodisse nimega paint. Meetodile antakse parameetrina graafiline kontekst, mis oma väljundi suunab pinnale, kus kasutaja arvab komponenti asuvat, üldjuhul ekraanile. Nii õnnestub lihtsustada komponenti näitava kesta tööd. Igal korral kui leitakse, et komponendi sisu on kaduma läinud või vajab lihtsalt uuendamist, kutsutakse taas välja paint, mis komponendi arvates sobiva oleku taastab. Kui komponendi omanik vaatab, et uuendamist vajab vaid osa komponendi pinnast, võidakse ressursside kokkuhoiu huvides saata paint-meetodisse pügatud (vähendatud, clip) joonistusalaga Graphics, mis kannab sihtpinnale edasi vaid nende joonistuskäskude tulemused, mis jäävad konteineri (komponendi omaniku) arvates uuendamist vajavasse piirkonda. Nõnda võib mõnikord veebilehe kerimisel avastada, et konteiner on uuendamist vajavat osa valesti hinnanud ning valgeks jäetakse ka osa komponendi pinnast, kus peaks midagi muud leiduma. Soovides kogu komponendi pinda uuendada, tuleb anda käsklus repaint(). See käsk käivitab pinnauuenduse eraldi lõimes. Ta ootab kõigepealt, et virtuaalmasinal jaguks piisavalt ressursse joonistamiseks. Edasi kutsutakse välja komponendi meetod update parameetrina graafiline kontekst komponendi asukohaseadme pinnale joonistamiseks, mis klassist java.awt.Component päritud vaikimisi oskuste alusel kõigepealt katab komponendi pinna taustavärviga ning seejärel kutsub välja meetodi paint, kus kirjeldatud käskude abil peaks kompomendi arvatav pinnavälimus kasutajani jõudma. Nõndamoodi saab hoolitseda, et vanast välimusest uude juhuslike jupikeste näol rudimente ei jääks, sest kui vana pind ühtlaselt taustavärvi plaadiga katta, siis ei tohi ju põhja midagi alla silma häirima jääda. Samas võib suurema pinna uuendamisel tekkida ekraanile vilgatus, kui platsi tühjendamise ning uue sisu joonistamise vahele piisavalt suur vahe jääb, et inimsilm seda märkab. Lahenduseks leitakse, et tuleb komponendi kohal näidatav pilt eelnevalt valmis arvutada ning siis ühekorraga või rida-realt ekraanile joonistada. Nõnda saab teha, kui pilt koostatakse mällu ning paint-meetodi sisuks oleks vaid selle pildi ekraanile manamine, update aga loobuks taustavärviga katmisest (taustavärv oleks ekraanile joonistatava pildi põhjaks) ning kutsuks kohe välja paint-i. update't annab muuta ülekatte abil: kui kirjutatada

```
public void update(Graphics g){
  paint(g)
}
```

siis jääbki taustaga katmine ära ning kogu töö usaldatakse paint'ile. Sellist joonistusviisi nimetatakse topeltpuhverduseks ning Swingi juures võib sama skeemi joonistamise juures ühe lisaparameetriga sisse lülitada.

Ekraanile joonistada saab ka mujal kui paint-meetodis, kuid sellisel juhul konteineri poolt kutsutud pinnauuenduse korral läheb kõik muu kaduma, mis paint'is kirjas pole. Sellegipoolest on mõnikord mugav kohe hiirevajutuse peale küsida pinna graafiline kontekst ning sealtkaudu soovitud ringike ekraanile manada, mitte asuda andmeid muutujates säilitama ning paint'i kaudu ekraanile tooma. Tõsisemate programmide juures tuleks aga valida viimane tee ning pea alati on siiski vaja ka omal teada, mida ja kus peaks ekraanile ilmuma. Ka liikumise puhul on mõistlik jätta pinna uuendamine paint-meetodi hooleks, kuigi vahel on mugavam otse leitud uute andmete järgi ka pilt ekraanile joonistada.

Taust liikumise ajal mälus.

Järgnevas näites koos kasutaja hiire lohistamisega liigub ekraanil ring. Liikumise taustaks on eelnevalt hiire lahtilaskmise kohas taustapildile joonistatud ringid ning iga uue lahtilaskmisega tekib taustapildile üks ring juurde. Joonistusmeetod paint viib ekraanile vaid taustapildi, liikumismulje jätab mouseDragged, kus alla joonistatakse taust ning peale hiire parajatist asukohta tähistav ring. Et taustapilt oleks kõikjalt Liigu2 isendist kätte saadav, selleks on deklareeritud

```
Image pilt;
```

klassi algusse. Pilt saab enesele väärtuse ning võimaluse andmeid sisestada aga alles joonistusmeetodis

```
public void paint(Graphics g){
    if(pilt==null)looPilt();
    g.drawImage(pilt, 0, 0, this);
  }
```

, sest varem pole kindlustatud, et createImage suudaks Liigu2 komponendi omadustele vastava pildi luua. Koos pildi loomisega küsitakse ka selle graafiline kontekst, et edaspidi oleks hea lihtne mälupildi peale andmeid saata.

```
void looPilt(){
    pilt=createImage(getSize().width, getSize().height);
    piltg=pilt.getGraphics();
  }
```

Java uuemates versioonides on võimalik pilt ka varem luua BufferedImage abil. Hiire sündmustele reageerimiseks on rakendile külge pandud kuular nii hiirevajutuste (MouseListener) kui hiire liikumise (MouseMotionListener) tarvis.

```
  public Liigu2(){
    addMouseMotionListener(this);
    addMouseListener(new Liigu2kuular(this));
  }
```

Liikumise ja vedamise tarvis kaks meetodit on realiseeritud Liigu2 enese sees, vajutuste püüdmiseks on aga loodud eraldi adapterklass, kust hiirenupu lahtilaskmise peale mälupildile ring joonistatakse.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
public class Liigu2 extends Applet
           implements MouseMotionListener{
  Image pilt;
  Graphics piltg;
  public Liigu2(){
    addMouseMotionListener(this);
    addMouseListener(new Liigu2kuular(this));
  }
  public void paint(Graphics g){
    if(pilt==null)looPilt();
    g.drawImage(pilt, 0, 0, this);
  }
  void looPilt(){
    pilt=createImage(getSize().width, getSize().height);
    piltg=pilt.getGraphics();
  }
  public void mouseMoved(MouseEvent e){}
  public void mouseDragged(MouseEvent e){
    Graphics g=this.getGraphics();
    g.drawImage(pilt, 0, 0, this);
    g.drawOval(e.getX()-5, e.getY()-5, 10, 10);
  }
  public static void main(String argumendid[]){
    Frame f=new Frame();
    f.add(new Liigu2());
    f.setSize(200, 200);
    f.setVisible(true);
  }
}
```

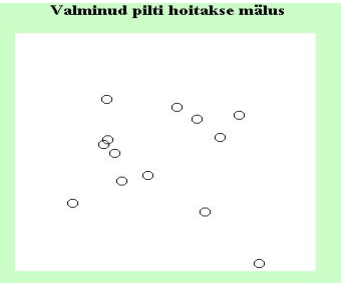
Et hiire vajutamise teateid püüdev suudaks Liigu2 juurde kuuluvale pildile midagi joonistada, selleks peab siin olema ligipääs ekraanile joonistatava pildi graafilisele kontekstile. Üheks võimaluseks on lisada kuularile

osuti, mille kaudu sihtisendile ligi pääseda. Selle muutuja nimeks on siin peremees. Objektorienteeritud programmeerimises on viisakas isendimuutujatele anda väärtusi meetodite kaudu, siin saadakse andmed kohale konstruktori kaudu. Kui üleval kirjutati kuulari lisamiseks

```
addMouseListener(new Liigu2kuular(this));
```

, siis see this Liigu2kuulari loomisel tähendabki osutit Liigu2 eksemplarile.

```
class Liigu2kuular extends MouseAdapter{
  Liigu2 peremees;
  public Liigu2kuular(Liigu2 l2){
    peremees=l2;
  }
  public void mouseReleased(MouseEvent e){
    peremees.piltg.drawOval(
                e.getX()-5, e.getY()-5, 10, 10);
  }
}
```



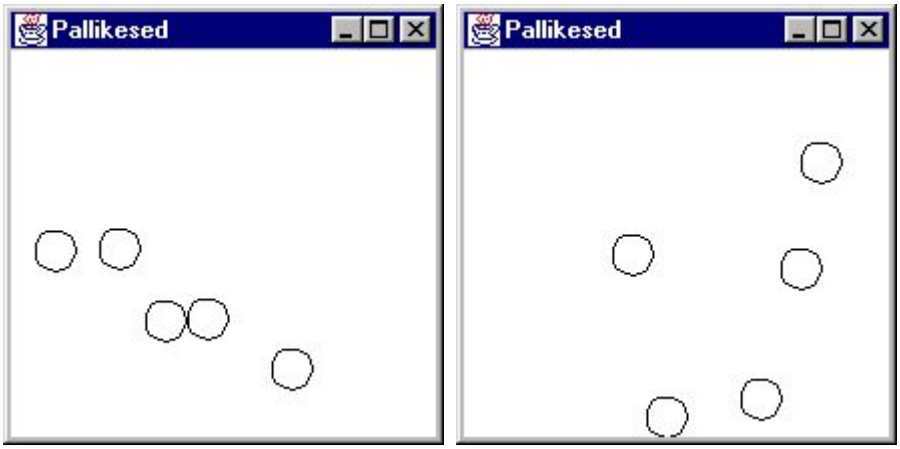
Lõim liikumisel

Järgnevas näites on lõime alamklass Pall2. Igal pallil on koordinaadid x ja y ning iga sammuga muutuvad need dx ja dy võrra. Iga pall liigub omaette lõimes, s.t. programmi muudest osadest sõltumatult. Pall alandab enese prioriteeti, s.t. et protsessori tööaja jagamisel loeb ta end keskmisest vähem tähtsamaks. Prioriteedi alandamine on vajalik selleks, et pallide liigutamine ei hakkaks märkimisväärselt aeglustama teisi lõimi, näiteks ekraanile joonistamist. Muidu võib juhtuda, et suure hulga pallide puhul jäävad ülejäänud tegevused suhteliselt unarusse. Käsk yield tähendab, et lõim annab oma tööjärje üle järgmisele ning asub ise järjekorda uuesti protsessori aega ootama.

Raami alamklass Loim2 loob enesele viis palli isendit ning laseb nad liikuma. Loim2 ise realiseerib liidest Runnable, s.t., et tema run-meetodit on samuti võimalik panna tööle eraldi lõimes. Selle lõime ülesandeks on joonistada iga natukese aja tagant uus ekraanipilt vastavalt pallide uutele koordinaatidele.

```
import java.awt.*;
public class Loim2 extends Frame implements Runnable{
 Pall2[] pallid;
 public Loim2(){
   pallid=new Pall2[5];
   for(int i=0; i<pallid.length; i++)
     pallid[i]=new Pall2();
   setSize(200, 200);
   setVisible(true);
   new Thread(this).start();
 }
 public void joonista(){
   Image pilt=createImage(200, 200);
   Graphics piltg=pilt.getGraphics();
   for(int i=0; i<pallid.length; i++){
     piltg.drawOval(pallid[i].x()-10, pallid[i].y()-10, 20, 20);
   }
   this.getGraphics().drawImage(pilt, 0, 0, this);
 }
 public void run(){
   while(true){
     joonista();
     Thread.yield();
     try{Thread.sleep(100);}catch(Exception e){}
   }
 }
 public static void main(String argumendid[]){
   new Loim2();
 }
}

class Pall2 extends Thread{
  double x=200, y=200, dx, dy;
  int vasak=20, ulal=50, parem=200, all=200;
  public Pall2(){
    dx=5*Math.random();
    dy=5*Math.random();
```

```
    start();
  }
  public int x(){
    return (int)x;
  }
  public int y(){
    return (int)y;
  }
  public void run(){
    setPriority(Thread.NORM_PRIORITY-2);
    while(true){
      if(x>parem)dx=-Math.abs(dx);
      if(y>all)dy=-Math.abs(dy);
      if(x<vasak)dx=Math.abs(dx);
      if(y<ulal)dy=Math.abs(dy);
      x+=dx;
      y+=dy;
      yield();
      try{Thread.sleep(100);}catch(Exception e){}
    }
  }
}
```



Sirelasemäng

Järgnevalt vaadeldalse näidete abil teekond, mida läheb tarvis enamiku liikumisega seotud rakenduste puhul. Alustatakse tausta liigutamisest ning ükshaaval lisatakse elemente. Kas tulemuseks on mäng või juhitav simulaator sõltub rohkem vaatenurgast.

Liikuv taust

Küllalt mugav on tausta tekitada korduva pildiga, kuid see pole sugugi ainuke võimalus. Mustri võib ka pidevalt mõne kavala valemi abil välja arvutada. Mugavaks ümberkäimiseks peaks taustapilt olema nähtavast alast suurem, kuigi mälu kokkuhoiu huvides võib vajadusel proovida ka sama väiksemat pilti mitu korda üksteise kõrvale joonistada. Samuti on nõnda võimalik panna rakendus tausta joonistamisel ka nähtava ala suurust arvestama. Siin näites on taustapildi suuruseks 320x480 punkti, eeldatav vaatamissuurus aga 300x300 punkti. Tausta muster kordub iga 160 punkti tagant. Taustapilti joonistatakse iga kaadri haaval niikaua allapoole, kuni on läbitud kogu mustripikkus (siin 160 punkti). Siis alustatakse järgmisel korral jälle mustri joonistamist ülevalt. Nii tekibki pilt liikumisest, sest hüpe mustri kõrguse jagu üles ei ole kasutajale nähtav.

Muudetavad parameetrid on koondatud programmi algusse, et neid saaks ka siis oma vajaduste järgi paika sättida, kui programmi sisust suuremat ülevaadet pole. Lähem seletus:
Taustapildimuutuja on toodud meetoditest välja, et selle abil oleks võimalik pidevalt pildi poole pöörduda ning ei peaks iga joonistamisvajaduse eel pilti asuma failist uuesti välja lugema.
Mustripikkus on eraldi muutujas, sest taustapildi vahetamisel teistsuguse vastu on tarvis teada, kui pika hüppe peab üles tegema, et pilt kasutaja silmadele taas samasugune välja näeks.
Samm hoiab meeles, mitme ekraanipunkti jagu iga korraga pilti allapoole nihutatakse, paus teatab joonistamise vahel oodatavate millisekundite arvu. Sammu suurendamisel või ooteaja lühenemisel

liikumiskiirus kasvab. Liialt suur samm muudab aga liikumise hüplikuks, liialt väike ooteaeg aga ei pruugi lasta masinal tulemust korralikult välja joonistada.

```
Image taust;
 int mustripikkus=160;
 int samm=3;
 int paus=50;
 int nihe=0;
 boolean veel=false;
```

Plinkimise vältimiseks on üle kaetud meetod update, paludes sel kohe paint välja kutsuda. Muul juhul tahetaks igal korral ekraan enne pildi joonistamist taustavärviga katta ning selle tulemusena hakkaks silmeesine virvendama. Kuna aga taustapilt nagunii kogu nähtava ala katab, siis ei jää vana pilt nagunii näha ning taustavärviga katmine oleks mõttetu.

Pildi laadimiseks on loodud eraldi alamprogramm, kuna rakend ja rakendus saavad pildi kätte erinevalt. See meetod proovib kõigepealt pilti saada rakendi moel. Kui nii ei õnnestu, siis üritatakse Toolkiti abil failist lugeda. Kui fail leidub ja formaadid sobivad, siis üks võimalus nendest võiks õnnestuda.

Pilt laetakse sisse, kui käivitusjärg jõuab esimest korda paint-alamprogammini. Loogiline võiks tunduda pildi laadimine otse muutujate deklareerimise ajal, aga rakendi getImage pole selleks ajaks veel töövõimeline. Nii ongi tehtud paint'i algusesse valik, kus muutuja taust null-väärtuse korral (mis tal algselt on) palutakse pilt sisse laadida.

Iga paint-meetodi väljakutse korral liigutatakse taustapilti sammu jagu allapoole. Nihe näitab, palju ollakse algsest asendist edasi liikunud. Kui pärast arvutust ületab nihe mustripikkust, siis hüpatakse mustri pikkuse jagu ülespoole, et oleks taas võimalik rahumeeles alla liikuda.

Joonistamiskäskluses on x-koordinaadiks 0, sest tausta joonistatakse alates vasakust servast. Y-koordinaadi väärtus nihe-mustripikkus näitab, et igal korral määrab pildi asukoha nihke suurenemine. Samas konstantne mustripikkuse muutuja väärtus hoiab pilti pidevalt niipalju kõrgel, et pildi ülaserv ei hakkaks kasutajale paistma.

Kui kood vaid sellega piirduks, siis püsiks pilt enamiku ajast paigal. Ning vaid juhul, kui mõnd teist akent me rakenduse peale ja sealt ära liigutada või natukese aja tagant meie akna suurust muuta võiksime märgata mingit liikumist. Et aga saaksime silme ees näha pidevalt liikuvat pilti, peaksid üksikud hüpped toimuma piisavalt sageli.

Sarnase ülejoonistuse nagu toimub akna suuruse muutuse juures, saab esile kutsuda ka käsu repaint() käivitamisega. Püsivat pilti joonistavates programmides paigutati see enamasti tekstivälja või hiiresündmuse töötlusossa. Siin aga soovime, et meetodit kutsutaks välja pidevalt iga natukese aja tagant. Leidub ka klass, mille abil peaks võimalik olema täpselt soovitud ajavahemike tagant etteantud meetodit käivitada, kuid lihtsamal juhul piisab lahendusest, kus pärast iga joonistust peetakse soovitud aja pikkune paus. Selline vaikselt põksuv süda on ka siia rakendusse sisse ehitatud ning võlusõnaks on lõim.

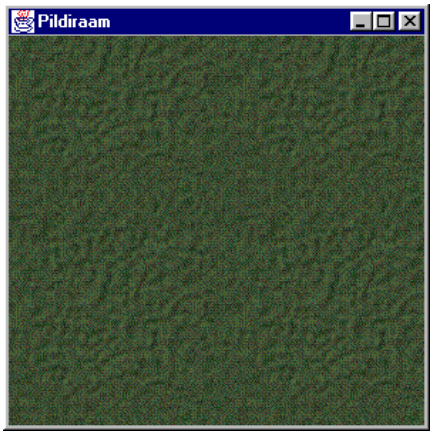
Kuigi traditsioonilise programmeerimise juures on programmi tööjärg kogu aeg kindlas kohas ning ilma eelmist sammu lõpetamata edasi ei liiguta, siis pole see sugugi ainus tava rakendusi töös hoida. Näiteks liigutamise, ehk praegusel juhul repaint'i väljakutsumise saab jätta suhteliselt iseseisvalt töötava lõime hooleks. Selle käivitamise, tööshoidmise ja seiskamise tarvis on näha järgnevas koodis hulk sõnu.

Lõime abil omaette käivitatav kood paigutatakse meetodi run sisse. Selline nimi on määratud liideses Runnable ning selle järgi teab Thread-tüüpi objekt, mida käivitada. Meetodi run ülesandeks selles programmis on repaint-käsklust korduvalt välja kutsuda. Selleks on loodud tsükkel milles käsklus repaint() ning ootamiseks Thread.sleep(paus). Viimane on paigutatud try-catch katsendiplokki, kuna võib mõnes olukorras (lõimede omavahelise suhtlemise korral) heita erindeid. Siin koodis taolist võimalust pole, kuid sellest hoolimata nõuab Java kompilaator potentsiaalselt erindiohtliku käsu ümbritsemist vastava plokiga. Muutuja veel while-tsükli päises võimaldab lõimel oma töö lõpetada, kui too tõeväärtusmuutuja väärtuseks antakse false. See juhtub näiteks käsus stop, mis kutsutakse välja seilur lahkub rakendit sisaldavalt lehelt.

Tööle lükatakse lõim meetodist start. Näites võib märgata kaht käsklust start, mis aga omavahel kuigivõrd seotud pole. Esimene kuulub klassi Pildike1 külge ning katab üle klassi Applet vastavat meetodit. Rakendikäitur (seilur) teatab selle kaudu Pildike1 eksemplarile, et isend on lehel avanenud. Siis on paras aeg liikumislõime käivitamiseks. Käsk new Thread(this).start() teatab, et loodagu uus klassi Thread eksemplar, millele antakse ette this ehk klassi Pildike1 eksemplar. Ning et etteantud eksemplaris pandagu iseseisvana käima run-meetod. Et run just käivitada tuleb, seda teab juba Thread. Iseseisva programmina käsurealt käivitades on näha main-meetodis käsku ap.start() – see käivitab Pildike1 start-käsu samuti nagu rakendikäiturgi veebis. Ning tulemusena õnnestub liikuvat tausta vaadata.

```
import java.applet.Applet;
import java.awt.*;
```

```
public class Pildike1 extends Applet implements Runnable{
  Image taust;
  int mustripikkus=160;
  int samm=3;
  int paus=50;
  int nihe=0;
  boolean veel=false;

  public void paint(Graphics g){
    if(taust==null)taust=laePilt("rohetaust320x480.gif");
    nihe=nihe+samm;
    if(nihe>mustripikkus)nihe=nihe-mustripikkus;
    g.drawImage(taust, 0, nihe-mustripikkus, this);
  }
  public void update(Graphics g){
    paint(g);
  }

  public void start(){
    veel=true;
    new Thread(this).start();
  }
  public void run(){
    while(veel){
        repaint();
      try{Thread.sleep(paus); }catch(Exception e){}
    }
  }
  public void stop(){
    veel=false;
  }


 Image laePilt(String failinimi){
    try{
     return getImage(getCodeBase(), failinimi);
    }catch(Exception e){}
     return Toolkit.getDefaultToolkit().
                           getImage(failinimi);
 }
 public static void main(String argumendid[]){
   Frame f=new Frame("Pildiraam");
   Pildike1 ap=new Pildike1();
   f.add(ap);
   f.setSize(300, 300);
   f.setVisible(true);
   ap.start();
 }
}
```



Taust koos lilledega

Nagu iga keerukama ülesande kallale tuleb asuda üksikute osade kaupa, nii ka siin on järgmiseks ette võetud lillede paigutamine taustale. Ning hoolitsetud, et lilled suhteliselt ühtlase tihedusega, kuid samas juhuslikult pinnale jaotuksid ning pinna suhtes paigal püsides inimese eest läbi liiguksid. Juurde on tulnud muutujad lilledega seotud andmete hoidmiseks. Lille kõrguse järgi saab arvestada millal ta ekraanile paistma hakkab, millal kaob ning hiljem ka leida, kas lill mõne muu alaga kattuma juhtub. Reaalarvuline lillelisamistõenäosus näitab, mitu lille luuakse keskmiselt iga sammu korral. Et väärtuseks on praegu 0.03*samm, tähendab, et iga kõrguse ekraanipunkti kohta tuleb keskmiselt 0.03 lille, ehk siis ligikaudu üks lill iga 33 punkti kohta. Lilli endeid hoitakse Vector'is; kollektsioonis, mille elemente võib vabalt lisada ja eemaldada ilma, et peaks ise muret tundma mälu eraldamise või vabastamise pärast. Aega arvutatakse ekraanipunktides. Et liikumine on ühtlase kiirusega, siis leidub ka kulunud ajaga võrdeline seos. Samahästi võiks muutujat nimetada kogunihkeks, ehk maapinna liikumise kogu punktide arvuks.

Lillede haldamise tarbeks on loodud mitu alamprogrammi: lisamise, eemaldamise ja joonistamise tarbeks.

Lillelisamistõenäosus näitas, mitu lille tuleb ühe joonistustsükli eel lisada. Kui see väärtus on näiteks 2,7, siis kaks lille lisatakse kindlasti, kolmas aga tõenäosusega 0,7 ehk 70%. Muutuja lt liigub tõenäosuse algsest väärtusest kuni nullini. Kuni lt väärtus on suurem kui 1, lisatakse lill kindlasti, sest Math.random()-i väljastatu on vahemikus 0<=x<1. Jääb aga lt väärtus alla ühe, siis sõltub Math.random'i tegevusest, kas tuleb veel lill või mitte. Lill luuakse uue objektina, millele jäetakse meelde tema x-koordinaat (muutumatu) ning aeg ehk maapinna liikumise teepikkus ekraanipunktides lille lisamise ajaks. Selle järgi on võimalik pärast arvutada, kus lill ekraanil peaks asuma, sest kõik lilled luuakse vaikimisi ülaserva. X-koordinaat leitakse juhuslikult pea kogu nähtava laiuse ulatuses. Kümme punkti võetakse laiusest x-i leidmisel maha, et ei tekiks lille, mis ekraanil sugugi näha poleks. Meeldejäetud x tähistab lille vasakut serva.

```
  void lisaUusiLilli(){
   for(double lt=lillelisamistoenaosus; lt>0; lt=lt-1){
    if(Math.random()<lt){
      lillekesed.addElement(new Lilleke2((int)((laius-10)*Math.random()), aeg));
    }
   }
  }
```

Lille klass näeb välja suhteliselt lihtne. Vaid kaks muutujat ning konstruktor algandmete sisestamiseks. Siin näites on ta paigutatud eraldi klassina samasse faili, kuid selle võiks paigutada ka sootuks omaette faili samasse kataloogi või siis hoopis sisemise klassina Pildike2 sisse. Eraldi failis saaks Lilleke2-te kasutada soovi korral otse ka teised klassid. Sisemise klassina paigutamise puhul aga pole muret et mõni muu samanimeline klass kusagil segadust tekitama hakkaks.

```
class Lilleke2{
  int x, algaeg;
  public Lilleke2(int x1, int algaeg1){
    x=x1; algaeg=algaeg1;
  }
}
```

Iga sammu juures kontrollitakse kõik lilled läbi ning eemaldatakse alt üle ääre sattunud. Käsuga elementAt küsitakse lillekeste vektorist välja järjekorranumbrile vastav lill. Tüübimuundus on tarvilik, kuna Vector hoiab kõiki andmeid ülemklassina Object, meil on aga tarvis lillelt küsida vaid temale omase välja algaeg väärtust.

```
  void eemaldaVanadLilled(){
    for(int i=0; i<lillekesed.size(); i++){
      Lilleke2 l=(Lilleke2)lillekesed.elementAt(i);
      if(aeg-l.algaeg>korgus+lillekorgus){
        lillekesed.removeElementAt(i);
        i--;
      }
    }
  }
```

Joonistamine läks pikemaks. Ehkki paint näeb välja ilus lühike, on tema ülesannet täitma tulnud hulk abilisi.

```
  public void paint(Graphics g){
    koostaPilt();
    g.drawImage(pilt, 0, 0, this);
  }
```

Lisaks taustale tuleb sisse lugeda ka lille pilt. Samuti ei joonistata tausta enam otse ekraanile, vaid pannakse mälus enne taust ja lilled uue pildi peale kokku ning alles seejärel näidatakse tulemus ekraanile. Mälupildi koostamiseks on createImage, mis jällegi rakendi puhul pole enne võimeline käivituma kui esimese paint-i ajal. Loodud pildi käest küsitud piltg jääb globaalsena üle kogu isendi kättesaadavaks, et ei pea muudele joonistavatele alamprogrammidele seda eraldi kätte jagama. Nagu näha, pärast iga sammu eemaldatakse vanad lilled, lisatakse uusi, joonistatakse taust mälupildi põhjaks ning lilled ükshaaval sellele.

```
  void koostaPilt(){
    if(taust==null)taust=laePilt("rohetaust320x480.gif");
    if(lill==null)lill=laePilt("lill1.gif");
    if(pilt==null){
      pilt=createImage(laius, korgus);
      piltg=pilt.getGraphics();
    }
    nihe=nihe+samm;
```

```
    aeg=aeg+samm;
    if(nihe>mustripikkus)nihe=nihe-mustripikkus;
    eemaldaVanadLilled();
    lisaUusiLilli();
    piltg.drawImage(taust, 0, nihe-mustripikkus, this);
    joonistaLilled();
  }
```

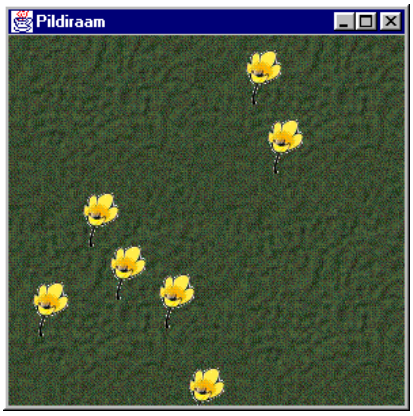
Lillede joonistamisel käiakse lihtsalt läbi kõik vektoris olevad lilled ning paigutatakse nende andmete järgi lillepilt taustapildile. Arvutus aeg-lille algaeg näitab lille asukoha ekraanil. Lisaks võetakse maha veel lillekõrgus, et lill asuks nähtavasse alasse sisenema oma alumise poolega ja mitte ei tekiks järsku tervikuna inimese vaatevälja.

```
  void joonistaLilled(){
    for(int i=0; i<lillekesed.size(); i++){
      Lilleke2 l=(Lilleke2)lillekesed.elementAt(i);
      piltg.drawImage(lill, l.x, aeg-l.algaeg-lillekorgus, this);
    }
  }
```

Ja tervikuna liikuva tausta ja juhuslike lilledega rakenduse kood.

```
import java.applet.Applet;
import java.awt.*;
import java.util.Vector;

public class Pildike2 extends Applet implements Runnable{
  Image taust;
  Image pilt;
  Graphics piltg;
  Image lill;
  int lillekorgus=100;
  int mustripikkus=160;
  int samm=3;
  int paus=50;
  int nihe=0;
  int aeg=0;
  int laius=300, korgus=300;
  Vector lillekesed=new Vector();
  double lillelisamistoenaosus=0.03*samm;
  boolean veel=false;

  public void paint(Graphics g){
    koostaPilt();
    g.drawImage(pilt, 0, 0, this);
  }
  public void update(Graphics g){
    paint(g);
  }

  void koostaPilt(){
    if(taust==null)taust=laePilt("rohetaust320x480.gif");
    if(lill==null)lill=laePilt("lill1.gif");
    if(pilt==null){
      pilt=createImage(laius, korgus);
      piltg=pilt.getGraphics();
    }
    nihe=nihe+samm;
    aeg=aeg+samm;
    if(nihe>mustripikkus)nihe=nihe-mustripikkus;
    eemaldaVanadLilled();
    lisaUusiLilli();
    piltg.drawImage(taust, 0, nihe-mustripikkus, this);
    joonistaLilled();
  }


  void lisaUusiLilli(){
   for(double lt=lillelisamistoenaosus; lt>0; lt=lt-1){
    if(Math.random()<lt){
      lillekesed.addElement(new Lilleke2((int)((laius-10)*Math.random()), aeg));
    }
   }
  }

  void eemaldaVanadLilled(){
    for(int i=0; i<lillekesed.size(); i++){
      Lilleke2 l=(Lilleke2)lillekesed.elementAt(i);
      if(aeg-l.algaeg>korgus+lillekorgus){
        lillekesed.removeElementAt(i);
```

```
          i--;
        }
      }
    }

    void joonistaLilled(){
      for(int i=0; i<lillekesed.size(); i++){
        Lilleke2 l=(Lilleke2)lillekesed.elementAt(i);
        piltg.drawImage(lill, l.x, aeg-l.algaeg-lillekorgus, this);
      }
    }

    public void start(){
      veel=true;
      new Thread(this).start();
    }

    public void run(){
      while(veel){
          repaint();
        try{Thread.sleep(paus); }catch(Exception e){}
      }
    }
    public void stop(){
      veel=false;
    }

    Image laePilt(String failinimi){
       try{
        return getImage(getCodeBase(), failinimi);
       }catch(Exception e){}
        return Toolkit.getDefaultToolkit().getImage(failinimi);
    }

public static void main(String argumendid[]){
     Frame f=new Frame("Pildiraam");
     Pildike2 ap=new Pildike2();
     f.add(ap);
     f.setSize(300, 300);
     f.setVisible(true);
     ap.start();
   }
}

class Lilleke2{
  int x, algaeg;
  public Lilleke2(int x1, int algaeg1){
    x=x1; algaeg=algaeg1;
  }
}
```



Liigutatav putukas

Ehkki lõpupoole soovin näha nii lilli kui putukaid koos liiklemas, on hea väiksemaid osi alustuseks eraldi poovida. Järgnevalt saab putukat noolte abil liigutada omasoodu liikuva tausta kohal. Putuka tarvis on juurde tulnud muutujad tema asukoha kohta, samuti sammu pikkus, mõõtmed ja pilt. Joonistatakse sarnaselt lilledega, st., et kõigepealt koostatakse mälus pilt taustast ja putukast ning siis kantakse tulemus tervikuna ekraanile.

Põhiline lisandus on putuka liigutamine klaviatuuri abil. Klahvisündmuste kuulamiseks on realiseeritavaks liideseks lisandunud KeyListener paketist java.awt.event. Liidesega koos kolm kohustuslikult realiseeritavat meetodit: keyPressed, keyReleased ja keyTyped. Et reageerida soovime vaid esimesele, on ülejäänute koodiosa tühi.

Klahvivajutusele reageerimisel küsitakse kõigepealt klahvi kood, et oleks võimalik asuda võrdlema, millisele klahvile vajutati. Vasaku noole puhul kontrollitakse kõigepealt, et putukas oleks vasakust äärest vähemalt oma sammupikkuse kaugusel ning sel puhul vähendatakse putuka x-koordinaadi väärtust sammu jagu. Nõnda võin vasakut noolt vajutades liikuda putukaga vasaku serva

lähedale, kuid mitte kaugemale. Pole karta, et putukas ekraani pealt lahkuks. Sarnane kontroll on ka teiste külgede juures, kuid paremal ja all arvestatakse lisaks veel putuka mõõtmetega, et ka parem ega alumine külg üle piiri ei läheks.

```
  public void keyPressed(KeyEvent e){
    int kood=e.getKeyCode();
    if((kood==KeyEvent.VK_LEFT) && (putukax>putukasamm))putukax-=putukasamm;
    if((kood==KeyEvent.VK_RIGHT) && (putukax<laius-putukasamm-putukalaius))
        putukax+=putukasamm;
    if(kood==KeyEvent.VK_UP && putukay>putukasamm)putukay-=putukasamm;
    if(kood==KeyEvent.VK_DOWN && putukay<korgus-putukasamm-putukakorgus)
        putukay+=putukasamm;
  }
```

Muu kood on küllalt sarnane tausta liikumise näitega. Lõim hoolitseb omasoodu sagedase pildiuuenduse eest, lisaks taustapildile on aga tarvis lisada ka putukas vastavalt oma koordinaatidele.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class Pildike3 extends Applet implements Runnable, KeyListener{
  Image taust;
  Image pilt;
  Graphics piltg;
  Image putukas;
  int putukax=100, putukay=150;
  int putukalaius=30, putukakorgus=15;
  int putukasamm=4;
  int mustripikkus=160;
  int samm=3;
  int paus=50;
  int nihe=0;
  int aeg=0;
  int laius=300, korgus=300;
  boolean veel=false;

  public Pildike3(){
    addKeyListener(this);
    requestFocus();
  }
  public void paint(Graphics g){
    koostaPilt();
    g.drawImage(pilt, 0, 0, this);
  }
  public void update(Graphics g){
    paint(g);
  }

  void koostaPilt(){
    if(taust==null)taust=laePilt("rohetaust320x480.gif");
    if(putukas==null)putukas=laePilt("sirelane.gif");
    if(pilt==null){
      pilt=createImage(laius, korgus);
      piltg=pilt.getGraphics();
    }
    nihe=nihe+samm;
    aeg=aeg+samm;
    if(nihe>mustripikkus)nihe=nihe-mustripikkus;
    piltg.drawImage(taust, 0, nihe-mustripikkus, this);
    piltg.drawImage(putukas, putukax, putukay, this);
  }


  public void start(){
    veel=true;
    new Thread(this).start();
  }

  public void run(){
    while(veel){
        repaint();
      try{Thread.sleep(paus); }catch(Exception e){}
    }
  }
  public void stop(){
    veel=false;
  }
  public void keyPressed(KeyEvent e){
```

```
    int kood=e.getKeyCode();
    if((kood==KeyEvent.VK_LEFT) && (putukax>putukasamm))putukax-=putukasamm;
    if((kood==KeyEvent.VK_RIGHT) && (putukax<laius-putukasamm-putukalaius))
        putukax+=putukasamm;
    if(kood==KeyEvent.VK_UP && putukay>putukasamm)putukay-=putukasamm;
    if(kood==KeyEvent.VK_DOWN && putukay<korgus-putukasamm-putukakorgus)
        putukay+=putukasamm;
  }

 public void keyReleased(KeyEvent e){}
 public void keyTyped(KeyEvent e){}
 Image laePilt(String failinimi){
    try{
     return getImage(getCodeBase(), failinimi);
    }catch(Exception e){}
     return Toolkit.getDefaultToolkit().
                         getImage(failinimi);
 }

 public static void main(String argumendid[]){
   Frame f=new Frame("Pildiraam");
   Pildike3 ap=new Pildike3();
   f.add(ap);
   f.setSize(300, 300);
   f.setVisible(true);
   ap.start();
 }
}
```

Nektarit imev putukas

Liikuvad lilled ja liigutatav putukas omaette läbi proovitud, nüüd kannatab nad ühte kesta kokku panna. Juures on omadus, kus putukas lilleni jõudmisel selle nektarist tühjaks imeb, nii et viimane pärast imemist seest tühjemana paistab. Ka lilleklassi sai veidi täiendatud: nüüd on sel lisaks oma x-koordinaadile ja algusajale meeles, kas on ta juba nektarist tühjaks imetud või veel mitte. Ning nii nagu objektorienteeritud programmile kohane, sai tühjuse määramiseks ja küsimiseks koostatud eraldi meetodid. Nii on näiteks võimalik oleku muutumisel kontrollida, kas uus väärtus sobib või soovi korral toimingud kuhugile testiks logida. Samuti võiks praeguse programmi korral olla lubatud tühi vaid tõeseks muuta, sest iseenesest juba tühjaks imetud õied siin enam mesimahla ei tekita.

```
class Lilleke4{
  int x, algaeg;
  boolean tyhi=false;
  public Lilleke4(int x1, int algaeg1){
    x=x1; algaeg=algaeg1;
  }
  void paneTyhi(boolean kasTyhi){
    tyhi=kasTyhi;
  }
  boolean kasTyhi(){
    return tyhi;
  }
}
```

Teadmaks, kas putukas mõne lille pihta satub, kontrollitakse läbi kõikide loetelus olevate lillede kaugused putukast. Kontrollimisel abiks põhikoolist tuttav Pythagorase teoreem, et täisnurkse

kolmnurga külgede ruutude summa on võrdne pikima külje ruuduga. Imemiskaugus on paigutatud eraldi muutujasse, et oleks võimalik määrata, kui kaugelt suudab putukas nektari kätte saada. Esimene mõte imemisvõimaluse leidmisel tõenäoliselt tuleb, et peaks leidma putuka ja lille vahelise kauguse ning siis võrdlema, kas leitud suurus on imemiskaugusest väiksem. Et aga ruutjuure arvutamine nõuab arvutilt küllalt palju protsessoritehteid ning iga sammu ajal on tarvilik leida hulga lillede ja putuka vaheline kaugus, siis annab programmi sujuvamaks teha, kui vaid võrrelda külgede ruutude summat imemiskauguse ruuduga. Viimane ei muutu ning selle saab vähemasti enne sammu algust valmis arvutada.

Kui leitakse, et lill putukale piisavalt lähedale sattus, siis antakse lillele teada, et mingu ta tühjaks. Samuti palutakse Toolkit'il tekitada väikene kõll.

```
  void kontrolliImemisi(){
    int kauguseruut=imemiskaugus*imemiskaugus;
    for(int i=0; i<lillekesed.size(); i++){
      Lilleke4 l=(Lilleke4)lillekesed.elementAt(i);
      int lilley=aeg-l.algaeg-lillekorgus/2;
      int xkaugus=putukax-l.x;
      int ykaugus=putukay-(lilley-lillekorgus/2);
      if(!l.kasTyhi() && (xkaugus*xkaugus+ykaugus*ykaugus)<kauguseruut){
        l.paneTyhi(true);
        Toolkit.getDefaultToolkit().beep();
      }
    }
  }
```

Joonistamise puhul tuleb siis iga lille puhul otsustada, milline pilt lillest välja näidata. Nektarit täis lille pilt on nime all lill, tühjaks imetu oma lill2. Avaldis (l.kasTyhi())?lill2:lill väljastab sulgudes oleva tõese tingimuse puhul küsimärgile kohe järgneva väärtuse, ehk pildi lill2. Kui aga lill pole veel tühjaks imetud, võetakse tulemus pärast koolonit ehk pilt muutujast lill.

```
  void joonistaLilled(){
    for(int i=0; i<lillekesed.size(); i++){
      Lilleke4 l=(Lilleke4)lillekesed.elementAt(i);
      Image lillepilt=(l.kasTyhi())?lill2:lill;
      piltg.drawImage(lillepilt, l.x, aeg-l.algaeg-lillekorgus, this);
    }
  }
```

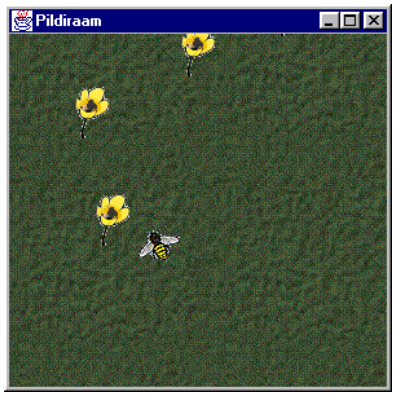
Ja taas kogu peaklassi kood tervikuna.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.util.Vector;

public class Pildike4 extends Applet implements Runnable, KeyListener{
  Image taust;
  Image pilt;
  Graphics piltg;
  Image lill, lill2;
  int lillekorgus=100;
  Image putukas;
  int putukax=100, putukay=150;
  int putukalaius=30, putukakorgus=15;
  int putukasamm=4;
  int imemiskaugus=10;
  int mustripikkus=160;
  int samm=1;
  int paus=50;
  int nihe=0;
  int aeg=0;
  int laius=300, korgus=300;
  Vector lillekesed=new Vector();
  double lillelisamistoenaosus=0.03*samm;

  boolean veel=false;

  public Pildike4(){
    addKeyListener(this);
  }
  public void paint(Graphics g){
    koostaPilt();
    g.drawImage(pilt, 0, 0, this);
  }
  public void update(Graphics g){
```

```
  paint(g);
}

void koostaPilt(){
  if(taust==null)taust=laePilt("rohetaust320x480.gif");
  if(putukas==null)putukas=laePilt("sirelane.gif");
  if(lill==null)lill=laePilt("lill1a.gif");
  if(lill2==null)lill2=laePilt("lill1.gif");
  if(pilt==null){
    pilt=createImage(laius, korgus);
    piltg=pilt.getGraphics();
  }
  nihe=nihe+samm;
  aeg=aeg+samm;
  if(nihe>mustripikkus)nihe=nihe-mustripikkus;
  eemaldaVanadLilled();
  lisaUusiLilli();
  kontrolliImemisi();
  piltg.drawImage(taust, 0, nihe-mustripikkus, this);
  joonistaLilled();
  piltg.drawImage(putukas, putukax, putukay, this);
}

void lisaUusiLilli(){
 for(double lt=lillelisamistoenaosus; lt>0; lt=lt-1){
  if(Math.random()<lt){
    lillekesed.addElement(new Lilleke4((int)((laius-10)*Math.random()), aeg));
  }
 }
}

void eemaldaVanadLilled(){
  for(int i=0; i<lillekesed.size(); i++){
    Lilleke4 l=(Lilleke4)lillekesed.elementAt(i);
    if(aeg-l.algaeg>korgus+lillekorgus){
      lillekesed.removeElementAt(i);
      i--;
    }
  }
}

void kontrolliImemisi(){
  int kauguseruut=imemiskaugus*imemiskaugus;
  for(int i=0; i<lillekesed.size(); i++){
    Lilleke4 l=(Lilleke4)lillekesed.elementAt(i);
    int lilley=aeg-l.algaeg-lillekorgus/2;
    int xkaugus=putukax-l.x;
    int ykaugus=putukay-(lilley-lillekorgus/2);
    if(!l.kasTyhi() && (xkaugus*xkaugus+ykaugus*ykaugus)<kauguseruut){
      l.paneTyhi(true);
      Toolkit.getDefaultToolkit().beep();
    }
  }
}

void joonistaLilled(){
  for(int i=0; i<lillekesed.size(); i++){
    Lilleke4 l=(Lilleke4)lillekesed.elementAt(i);
    Image lillepilt=(l.kasTyhi())?lill2:lill;
    piltg.drawImage(lillepilt, l.x, aeg-l.algaeg-lillekorgus, this);
  }
}



public void start(){
  veel=true;
  new Thread(this).start();
}

public void run(){
  while(veel){
      repaint();
    try{Thread.sleep(paus); }catch(Exception e){}
  }
}
public void stop(){
  veel=false;
}
public void keyPressed(KeyEvent e){
  int kood=e.getKeyCode();
  if((kood==KeyEvent.VK_LEFT) && (putukax>putukasamm))putukax-=putukasamm;
  if((kood==KeyEvent.VK_RIGHT) && (putukax<laius-putukasamm-putukalaius))
```

```
        putukax+=putukasamm;
    if(kood==KeyEvent.VK_UP && putukay>putukasamm)putukay-=putukasamm;
    if(kood==KeyEvent.VK_DOWN && putukay<korgus-putukasamm-putukakorgus)
        putukay+=putukasamm;
  }
  public void keyReleased(KeyEvent e){}
  public void keyTyped(KeyEvent e){}
  Image laePilt(String failinimi){
     try{
      return getImage(getCodeBase(), failinimi);
     }catch(Exception e){}
      return Toolkit.getDefaultToolkit().getImage(failinimi);
  }

  public static void main(String argumendid[]){
    Frame f=new Frame("Pildiraam");
    Pildike4 ap=new Pildike4();
    f.add(ap);
    f.setSize(300, 300);
    f.setVisible(true);
    ap.start();
  }
}
```

Sirelane mängu alguses



Esimesed lilled

Kaks tühjaks imetud lille

Püüdlus terviku poole

Nähtavalt on juurde tulnud kerimisribad putuka liikumistundlikkuse, maapinna liikumiskiiruse ja lillede tiheduse määramiseks. Näha veel väike statistika ning klahve vajutades tunda, et putukas mitte ei hüppa klahvivajutuse peale, vaid klahvidega saab pigem putuka suunda ja kiirust määrata. Ühes sellega on muutunud või lisandunud hulga koodi.

Kõigepealt tuleb kerimisribad mälus valmis luua. Sulgude sees näha neil hulga parameetreid. Scrollbar.HORIZONTAL näitab, et riba on pikali. Sooviks püstist riba, oleks vastavaks konstandiks Scrollbar.VERTICAL. Järgnevad arvud: algne väärtus, nupu laius, vähim ja suurim võimalik väärtus. Nagu siit aimata võib, määratakse ja küsitakse kerimisriba väärtusi arvude abil.

```
Scrollbar sbtaustasamm=new Scrollbar(Scrollbar.HORIZONTAL, 10, 5, 0, 100);
```

Et kerimisriba sündmustele annaks reageerida, selleks on klassi realiseeritavate liideste loendisse lisandunud AdjustmentListener

```
public class Pildike5 extends Applet implements Runnable, KeyListener,
AdjustmentListener{
```

Liidesega käib koos meetod adjustmentValueChanged, mis on võimalik kerimisriba liigutamise peale käivitada. Iga käivituse puhul küsitakse kõigist kolmest kerimisribast väärtused ja paigutatakse vastavatesse muutujatesse. Iseenesest oleks võimalik küsida meetodi parameetriks tulnud AdjustmentEvent'i käest käsuga e.getSource(), et millist riba just liigutati. Kuna aga kerimisribast väärtuse küsimine ei nõua kuigivõrd ressurssse, siis pole eristust tehtud. Samuti oleks võimalik läbi ajada sootuks ilma sammupikkuse või mõne muu muutujata, küsides igal vajaminemiskorral väärtuse otse kerimisribast.

Kerimisribast saab väärtuse küsida vaid täisarvuna. Et aga liikumisel saaks asukohti sujuvamalt arvutada, on liikumisega seotud väärtused salvestatud reaalarvudena. Putuka ja maapinna liikumise puhul on kerimisribalt väärtus jagatud kümnega. Lillelisamistõenäosuse puhul aga, kus väärtused väiksemad ning kümnendkohtadel suurem tähtsus - tuhandega. Lõpuks on palutud lõuend fookusesse kutsuda, et kasutaja klahvivajutused taas lõuendilt kinni püütavad oleksid. Lõuendil asuvad taust, lilled ja putukas.

```
public void adjustmentValueChanged(AdjustmentEvent e){
  samm=sbtaustasamm.getValue()/10.0;
  kiirusesamm=sbputukatundlikkus.getValue()/10.0;
  lillelisamistoenaosus=sblilletoenaosus.getValue()/1000.0*samm;
  louend.requestFocus();
}
```

Graafikakomponendid paigutatakse konstruktoris. Kui eelmises näites joonistati rakendi enese pinnale, siis nüüd kasutatakse selleks eraldi lõuendit. Nõnda on kergem hoolitseda, et kerimisribad sisestusfookust enesele ei haarakse. Veel taseme jagu viisakam oleks kogu liikumine ja joonistamine jätta eraldi komponendi sisse ning komponendi külge ehitada meetodid, mille abil saab parameetreid muuta. Siis luua kestprogramm, mis paigutaks enese peale nii tolle joonistava komponendi kui kerimisribad ning kas vahendaks kerimisribade teated loodud komponendile või paluks ribadel otse oma teated sinna saata.

Pildike5 paigutushalduriks on valitud BorderLayout. Nii saab paigutada (alla) serva juhtribad ning joonistuskomponendi venitada üle muu pinna, nii et rakend oleks ühtlaselt kaetud.

Alumised kerimisribad koos nende sisu selgitavate siltidega paigutati omaette paneeli. Nii saab valmis ehitatud ploki pärast tervikuna rakendi allserva paigutada. Paneeli paigutushalduriks määrati GridLayout kolme rea ja kahe veeruga. Ning edasi paigutati sinna järgemööda sisse nii kirjeldavad sildid kui ribad ise.

```
public Pildike5(){
  setLayout(new BorderLayout());
  Panel p1=new Panel(new GridLayout(3, 2));
  p1.add(new Label("Maapinna kiirus")); p1.add(sbtaustasamm);
  p1.add(new Label("Putuka liikumistundlikkus")); p1.add(sbputukatundlikkus);
  p1.add(new Label("Lillede sagedus")); p1.add(sblilletoenaosus);
  add(p1, BorderLayout.SOUTH);
  add(louend, BorderLayout.CENTER);
  louend.addKeyListener(this);
  sbtaustasamm.addAdjustmentListener(this);
  sbputukatundlikkus.addAdjustmentListener(this);
  sblilletoenaosus.addAdjustmentListener(this);
}
```

Asukohtade arvutamisel on juurde tulnud putuka liikumise arvutus. Nii nagu maapind, nii ka putukas liigub üldjuhul ühtlase kiirusega. Enne putuka sammu võrra liikumist kontrollitakse, kas uus soovitav asukoht asub liikumiseks sobiliku pinna sees. Meetodi esimesed kaks koordinaati tähistavad uuritavat kohta, viimased kaks lubatud ala laiust ja kõrgust. Eeldatakse, et ala vasak ja ülemine serv hakkavad nullist.

```
boolean sees(double x, double y, double x2, double y2){
  if(x>=0 && x<x2 && y>=0 && y<y2) return true;
  return false;
}
```

Kui aga juhtub, et putukas on sattunud lubatud ala serva lähedale ning edasi pole võimalik liikuda, siis seatakse ta liikumiskiirus (sammu pikkus ühe joonistuse jooksul) nulliks.

```
void arvutaAsukohad(){
  nihe=nihe+samm;
  aeg=aeg+samm;
  if(nihe>mustripikkus)nihe=nihe-mustripikkus;
  if(sees(putukax+putukaxkiirus, putukay+putukaykiirus,
              laius-putukalaius, korgus-putukakorgus)){
    putukax+=putukaxkiirus;
    putukay+=putukaykiirus;
  } else {
    putukaxkiirus=putukaykiirus=0;
  }
}
```

Reageering klahvivajutusele on mõnevõrra muutunud. Kui ennist muutus vajutusel putuka asukoht, siis nüüd püütakse muuta ta liikumise kiirust. Näitena näha vasak nooleklahv. Kui putukas liikus ennist vasakule (st. putukaxkiirus oli nullist väiksem) sel juhul klahvile vajutades vasakule liikumise kiirus kasvab kiirusesammu võrra, ehk kiirusest lahutatakse kiirusesammu väärtus. Kui aga putukas juhtus enne paigal seisma või paremale liikuma, siis määratakse uueks kiiruseks -kiirusesamm, ehk sama palju, kui muidu iga vajutamisega kiirust juurde tuleb. Nõnda ei pea kiiruse suuna muutmisel liialt aega kulutama pidurdamisele, vaid võib küllalt järsku teises suunas liikuma hakata, nagu putuka lennu puhul ikka näha võib.

```
public void keyPressed(KeyEvent e){
  int kood=e.getKeyCode();
  if(kood==KeyEvent.VK_LEFT){
    if(putukaxkiirus<0)putukaxkiirus-=kiirusesamm;
    else putukaxkiirus=-kiirusesamm;
  }
  //...
}
```

Juurde tuli mõningane statistika, et kasutajal oleks näha, kaugele ta mänguga jõudnud on. Kulunud aja arvutamiseks küsitakse käivitamise ajal eraldi muutujasse alghetke väärtus. Tegemist küllalt suure ja väheütleva arvuga: millisekundeid alates aastast 1970.

```
long alghetk=new Date().getTime();
```

Kui nüüd aga millalgi uuesti süsteemi käest aeg küsida, siis nende kahe väärtuse vahe ütleb, palju mängu algusest aega kulunud on.

```
void joonistaStatistika(){
  String st="Lilli: "+lilli+" Imetud: "+imetud+
       " Kadunud: "+kadunud+" Aeg: "+(new Date().getTime()-alghetk)/1000;
  piltg.setColor(Color.white);
  piltg.drawString(st, 30, korgus-10);
}
```

Rakendusele pandi kaasa heli. Pidevalt korduv linnutaust, et veidigi tekiks mulje niidul lendavast putukast ning märku andev sahin, kui putukas lillest nektari kätte sai. Taustaga on lihtsam. Kui esimest korda joonistama asutakse, siis palutakse muutujasse laadida linnutaust ning käsu loop abil pannakse too korduvalt end ketrama.

```
  if(linnutaust==null){
    linnutaust=laeKlipp("linnutaust.au");
    linnutaust.loop();
  }
```

Lille tabamise sahina saaks ka lihtsamal juhul laadida ning käsuga play mängima panna. Kui aga lilli palju ning sahinaheli vähegi pikem, siis kipuvad järjestikustel tabamustel helid kattuma ning üksikud

tabamused ei eristu kõrvale ilusti ja loendatavalt. Sahinahelide eristamiseks loodi omaette klass. Lähem kirjeldus näha juba kommentaarides.

```
/**
* Lõimeklass, mille abil saab meloodiajuppe mängida. Sünkroniseerimise abil
hoolitsetakse,
* et mängualguste vahel oleks vähemalt poolesekundiline paus. Luku
* (loa)ga sünkroniseeritud plokki pääseb korraga vaid üks lõim. Ülejäänud
* peavad selle ees ootama, kuni eelmine on plokist väljunud.
*/
class Piiksuja extends Thread{
  static Object lukk=new Object();
  static java.applet.AudioClip sahin;
  public void run(){
   synchronized(lukk){
    sahin.play();
    try{Thread.sleep(500);}catch(Exception e){}
   }
  }
}
```

Nagu näha, on Piiksuja nii lukk kui sahin staatilised muutujad. See tähendab, et neile on võimalik ligi pääseda sõltumata klassi eksemplaride arvust. Lukuobjekt luuakse programmi käivitamisel ning sama isend on kättesaadav kõigile klassi isenditele - nõndamoodi saab selle järgi sünkroniseerimisel otsustada, et korraga ei juhtuks mitu sahinat mängima. Piiksuja sahin laetakse koos muude klippidega ja paigutatakse sinna staatilisse muutujasse.

```
    if(Piiksuja.sahin==null) Piiksuja.sahin=laeKlipp("sahin.au");
```

Kui nüüd käivitamise juures jõutakse niikaugele, et on paras aeg sahistada, siis luuakse lõimeklassist Piiksuja uus eksemplar ning palutakse start abil eraldi lõimes ta run käivitada.

```
      if(!l.kasTyhi() && (xkaugus*xkaugus+ykaugus*ykaugus)<kauguseruut){
        l.paneTyhi(true);
        imetud++;
        new Piiksuja().start();
      }
```

Kui eelmist mängijat pooleli pole, siis asutakse pea kohe mängima. Kui aga eelmine lõime eksemplar peatub luku abil sünkroniseeritud ploki sees, siis jääb uus mängija järjekorda, kuni eelmine on oma töö lõpetanud ning plokist väljunud. Kui nüüd liikuda putukaga läbi tiheda lillesalu, võib järgemööda eristatult kuulda mitut sahinat - iga tabatud lille kohta üht.

```
   synchronized(lukk){
    sahin.play();
    try{Thread.sleep(500);}catch(Exception e){}
   }
```

Nagu koodist näha, pole rakendiklassil Pildike5 enam main-meetodit. Lähem seletus klassi kommentaarides.

```
/**
* Alamklass liikuvate lilledega rakendile Pildike5. Meetod laeKlipp on siin üle
kaetud,
* sest rakendil saadakse heliklipi andmed käsust getAudioClip, rakendusel aga
samaotstarbeliseks
* käsuks Applet.newAudioClip. Viimane kuulub aga JDK koosseisu alates versioonist 1.2
ning
* varasema versiooni seilurid annavad neile tundmatu meetodi sisse lugemisel klassi
kohta veateate
* ning keelduvad vastava klassiga edaspidi tegelemast kartes turvamuresid. Kui aga
vastav meetod
* siin üle katta, siis ülemklass on sellest meetodist prii ja võib rahus rakendis
töötada,
* käsurealt käivitades aga võetakse siinse alamklassi üle kaetud käsklus ning pannakse
* tööle.
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.io.*;
class Pildike5Raam extends Pildike5{
  AudioClip laeKlipp(String failinimi){
```

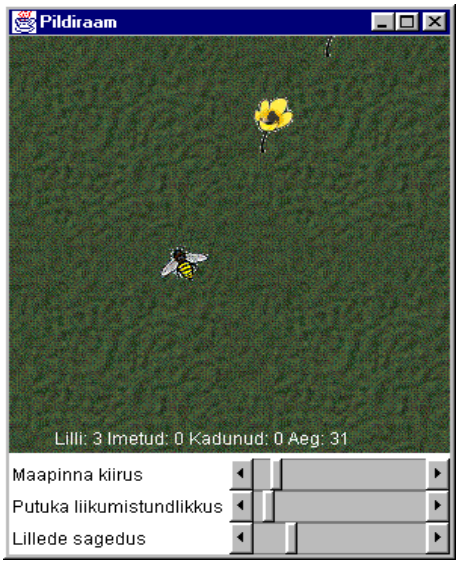
```
    try{return Applet.newAudioClip(new File(failinimi).toURL());}catch(Exception e){}
    return null;
  }

  public static void main(String argumendid[]){
    Frame f=new Frame("Pildiraam");
    Pildike5Raam ap=new Pildike5Raam();
    f.add(ap);
    f.setSize(300, 400);
    f.setVisible(true);
    ap.start();
    f.addWindowListener(
      new WindowAdapter(){
        public void windowClosing(WindowEvent e){
          System.exit(0);
        }
      }
    );
  }
}
```

Nõnda on tehtud muudatused/täiendused üle vaadatud ning rakenduse terviku kood loodetavasti arusaadav.

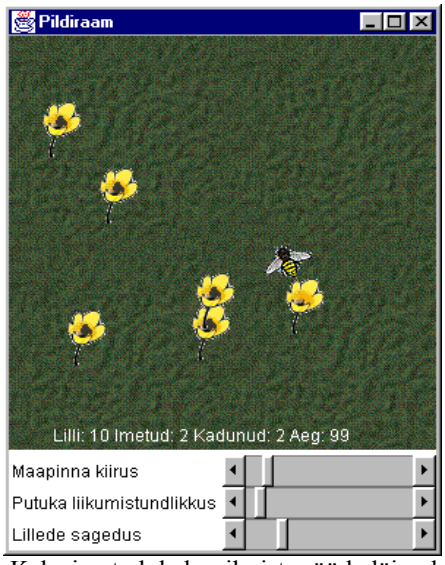
```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Pildike5 extends Applet implements Runnable, KeyListener,
AdjustmentListener{
  Image taust;
  Image pilt;
  AudioClip linnutaust;
  Graphics piltg;
  Image lill, lill2;
  int lillekorgus=100, lillelaius=50;
  Image putukas;
  double putukax=100, putukay=150;
  double putukaxkiirus=0, putukaykiirus=0;
  double kiirusesamm=0.2;
  int putukalaius=35, putukakorgus=35;
  int imemiskaugus=10;
  int mustripikkus=160;
  double samm=1;
  int paus=50;
  double nihe=0;
  double aeg=0;
  int laius=300, korgus=300;
  int lilli=0, imetud=0, kadunud=0;
  long alghetk=new Date().getTime();
  Vector lillekesed=new Vector();
  double lillelisamistoenaosus=0.02*samm;
  Canvas louend=new Canvas(){
    public boolean isFocusTraversable(){
      return true;
    }
  };
  Scrollbar sbtaustasamm=new Scrollbar(Scrollbar.HORIZONTAL, 10, 5, 0, 100);
  Scrollbar sbputukatundlikkus=new Scrollbar(Scrollbar.HORIZONTAL, 2, 1, 1, 20);
  Scrollbar sblilletoenaosus=new Scrollbar(Scrollbar.HORIZONTAL, 20, 1, 0, 100);
  boolean veel=false;

  public Pildike5(){
    setLayout(new BorderLayout());
    Panel p1=new Panel(new GridLayout(3, 2));
    p1.add(new Label("Maapinna kiirus")); p1.add(sbtaustasamm);
    p1.add(new Label("Putuka liikumistundlikkus")); p1.add(sbputukatundlikkus);
    p1.add(new Label("Lillede sagedus")); p1.add(sblilletoenaosus);
    add(p1, BorderLayout.SOUTH);
    add(louend, BorderLayout.CENTER);
    louend.addKeyListener(this);
    sbtaustasamm.addAdjustmentListener(this);
    sbputukatundlikkus.addAdjustmentListener(this);
    sblilletoenaosus.addAdjustmentListener(this);
  }
  public void adjustmentValueChanged(AdjustmentEvent e){
    samm=sbtaustasamm.getValue()/10.0;
    kiirusesamm=sbputukatundlikkus.getValue()/10.0;
```

```
    lillelisamistoenaosus=sblilletoenaosus.getValue()/1000.0*samm;
    louend.requestFocus();
  }

  public void joonista(){
    koostaPilt();
    louend.getGraphics().drawImage(pilt, 0, 0, this);
  }

  void laeKlipid(){
    if(taust==null)taust=laePilt("rohetaust320x480.gif");
    if(putukas==null)putukas=laePilt("sirelane.gif");
    if(lill==null)lill=laePilt("lill1a.gif");
    if(lill2==null)lill2=laePilt("lill1.gif");
    if(pilt==null){
      pilt=createImage(laius, korgus);
      piltg=pilt.getGraphics();
    }
    if(Piiksuja.sahin==null) Piiksuja.sahin=laeKlipp("sahin.au");
    if(linnutaust==null){
      linnutaust=laeKlipp("linnutaust.au");
      linnutaust.loop();
    }
    korgus=louend.getSize().height;
    laius=louend.getSize().width;
    louend.requestFocus();
  }

  void arvutaAsukohad(){
    nihe=nihe+samm;
    aeg=aeg+samm;
    if(nihe>mustripikkus)nihe=nihe-mustripikkus;
    if(sees(putukax+putukaxkiirus, putukay+putukaykiirus,
                laius-putukalaius, korgus-putukakorgus)){
      putukax+=putukaxkiirus;
      putukay+=putukaykiirus;
    } else {
      putukaxkiirus=putukaykiirus=0;
    }
  }

  void koostaPilt(){
    if(taust==null)laeKlipid();
    arvutaAsukohad();
    eemaldaVanadLilled();
    lisaUusiLilli();
    kontrolliImemisi();
    piltg.drawImage(taust, 0, (int)nihe-mustripikkus, this);
    joonistaLilled();
    joonistaStatistika();
    piltg.drawImage(putukas, (int)putukax, (int)putukay, this);
  }



  void lisaUusiLilli(){
   for(double lt=lillelisamistoenaosus; lt>0; lt=lt-1){
    if(Math.random()<lt){
      lillekesed.addElement(new Lilleke4((int)((laius-lillelaius)*Math.random()),
(int)aeg));
      lilli++;
    }
   }
  }

  void eemaldaVanadLilled(){
    for(int i=0; i<lillekesed.size(); i++){
      Lilleke4 l=(Lilleke4)lillekesed.elementAt(i);
      if(aeg-l.algaeg>korgus+lillekorgus){
        if(!l.tyhi)kadunud++;
        lillekesed.removeElementAt(i);
        i--;
      }
    }
  }

  void kontrolliImemisi(){
    int kauguseruut=imemiskaugus*imemiskaugus;
    for(int i=0; i<lillekesed.size(); i++){
      Lilleke4 l=(Lilleke4)lillekesed.elementAt(i);
      int lilley=(int)aeg-l.algaeg-lillekorgus/2;
      int xkaugus=(int)putukax-l.x;
      int ykaugus=(int)putukay-(lilley-lillekorgus/2);
```

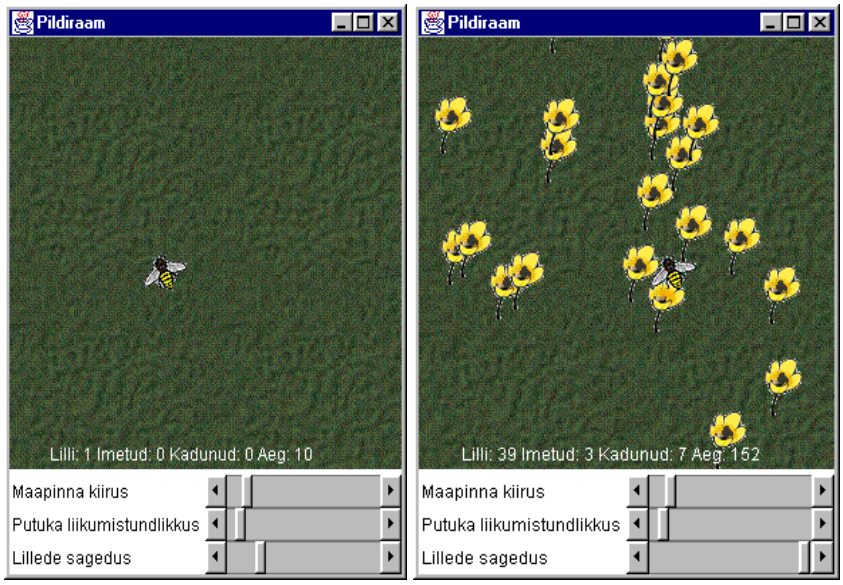
```
      if(!l.kasTyhi() && (xkaugus*xkaugus+ykaugus*ykaugus)<kauguseruut){
        l.paneTyhi(true);
        imetud++;
        new Piiksuja().start();
      }
    }
  }

  void joonistaLilled(){
    for(int i=0; i<lillekesed.size(); i++){
      Lilleke4 l=(Lilleke4)lillekesed.elementAt(i);
      Image lillepilt=(l.kasTyhi())?lill2:lill;
      piltg.drawImage(lillepilt, l.x, (int)aeg-l.algaeg-lillekorgus, this);
    }
  }

  void joonistaStatistika(){
    String st="Lilli: "+lilli+" Imetud: "+imetud+ " Kadunud: "+kadunud+" Aeg: "+(new
Date().getTime()-alghetk)/1000;
    piltg.setColor(Color.white);
    piltg.drawString(st, 30, korgus-10);
  }

  public void start(){
    veel=true;
    new Thread(this).start();
    if(linnutaust!=null)linnutaust.loop();
  }

  public void run(){
    while(veel){
        joonista();
      try{Thread.sleep(paus); }catch(Exception e){}
    }
  }
  public void stop(){
    veel=false;
    linnutaust.stop();
  }

  boolean sees(double x, double y, double x2, double y2){
    if(x>=0 && x<x2 && y>=0 && y<y2) return true;
    return false;
  }

  public void keyPressed(KeyEvent e){
    int kood=e.getKeyCode();
    if(kood==KeyEvent.VK_LEFT){
      if(putukaxkiirus<0)putukaxkiirus-=kiirusesamm;
      else putukaxkiirus=-kiirusesamm;
    }
    if(kood==KeyEvent.VK_RIGHT){
      if(putukaxkiirus>0)putukaxkiirus+=kiirusesamm;
      else putukaxkiirus=kiirusesamm;
    }
    if(kood==KeyEvent.VK_UP){
      if(putukaykiirus<0)putukaykiirus-=kiirusesamm;
      else putukaykiirus=-kiirusesamm;
    }
    if(kood==KeyEvent.VK_DOWN){
      if(putukaykiirus>0)putukaykiirus+=kiirusesamm;
      else putukaykiirus=kiirusesamm;
    }
  }

  public void keyReleased(KeyEvent e){}
  public void keyTyped(KeyEvent e){}

  Image laePilt(String failinimi){
     try{
      return getImage(getCodeBase(), failinimi);
     }catch(Exception e){}
      return Toolkit.getDefaultToolkit().getImage(failinimi);
  }

  AudioClip laeKlipp(String failinimi){
      return getAudioClip(getDocumentBase(), failinimi);
  }
}
```



Esimesed lilled



Kaks imetud, kaks silmist mööda läinud



Tihedaks määratud lilleväli

Edasiarendusvõimalused

Loodud rakendus küll töötab ning on mõningase näpuharjutusena mängitav, kuid pole ta veel ei terviklik mäng ega saa seda kuigivõrd asjalikult ka mõneks muuks otstarbeks kasutada. Olemasolevat koodi vähemalt osalt mõistes on võimalik küllalt vähese vaeva abil koodi toimimist ja rakenduse väljanägemist täiesti märkimisväärselt muuta. Lihtsaimaks asenduseks ehk pildid ja heliklipid. Kui joonistada või leida muru ja lillede abil miskit muud, siis annab kergesti kokku panna koristaja maanteel või elevandi portselanikaupluses. Üks üliõpilane aga kujundas piltide muutmise abil sellest samast näitest aga näiteks küllaltki verise sõjamängu.

Kui näidet mänguks kujundada, siis tõenäoliselt tuleb üsna pea lisada punktidearvutus ja mängu lõpp, ehk ka tasemed. Punkte kannatab kuvada statistikafunktsiooni veidi muutes. Iseseisva rakenduse puhul annab vahetulemusi faili salvestada. Rakendi puhul on turvapiirangute tõttu see veidi keerulisem, kuid kel PHP või muu veebiserveripoolse programmeerimisvahendiga kokkupuuteid ja kasutusvõimalusi, siis on ka see täiesti tehtav.

Ilus võistlusmoment tekib, kui ühe putuka asemel lendleb kaks, kummagi juhtimiseks omaette klahvid. Ning samuti võib saabuvaid pilte olla mitut tüüpi. Nii magusaid mesikaid, õiteta okaspuid kui muude vahele ära eksinud kärbsepabereid.

Näide ei pea aga mitte ainult mängu aluseks olema. Mõningase muutmise teel saab selle põhjal koostada katseseadme pidurdusteekonna pikkuse või molekulide kaootilise liikumise näitamiseks. Kõige rohkem läheb vaja pealehakkamist.

Ülesandeid

Sirelasemäng
Alustuseks on kasutada mängu põhi

- Tutvu mänguga
- Lae lähtekood, pildid ja helilõigud oma arvutisse, kompileeri ja käivita.
- Otsi Internetist pilte ning kujunda nende abil mäng oma silma järgi.
- Lisa punktidearvutus.
- Loo mängule lõpp.
- Luba samaaegselt liikuda kahel mängijal

Klahvidega liigutamine.

- Nooleklahvidega saab liigutada ekraanil olevat ringi.
- Ekraanil juhuslikus kohas paikneb rist. Jõudes ringiga selleni, hüppab rist uude juhuslikku kohta.
- Liigutatavaid ringe on kaks, kummalgi oma klahvid.
- Risti asemel on pisike pilt. All servas on kirjas, mitu korda kumbki mängija on ristini jõudnud.

Tennis

- Võrguga tenniseväljakul liigub pall (ring) vasakult paremale.
- Samaaegselt ringi liikumisega saab ekraanil liigutada reketit.
- Kummalgi serval on liigutatav reket, mille tabamisel pall tagasi põrkab. Loetakse punkte.

Ussimäng

- 10*10 ruudustikul asub vasakul ülaservas neljalüliline uss, keda saab nooleklahviga paremale liigutada.
- Uss liigub pidevalt, nooleklahvidega saab tema liikumise suunda määrata. Seinani jõudmisel uss seiskub.
- Ussiga püütakse juhuslikus kohas asuvat kuldmuna. Muna kättesaamisel uss pikeneb kahe lüli võrra ning muna tekib uude kohta. Seina või enese hammustamisel uss lüheneb nelja lüli võrra.

Tilgapüüdja

- All servas saab liigutada kaussi
- Ekraanil kukub tilk. Selle kaussi püüdmisel saab punkti.
- Tilku võib korraga kukkuda mitu. Mäng on meeldivalt kujundatud ning töötab märgatavate vigadeta.

Nooltega ralli.

- Noolte abil saab muuta ekraanil liikuva ringi kiirust.
- Külgmiste nooltega saab muuta liikumise suunda, teiste nooltega kiirust.
- Korraga võib üle võrgu sõita kaks kasutajat. Aetakse taga juhuslikus kohas paiknevat aaret, mis selleni jõudmisel hüppab uude kohta.

Graphics2D

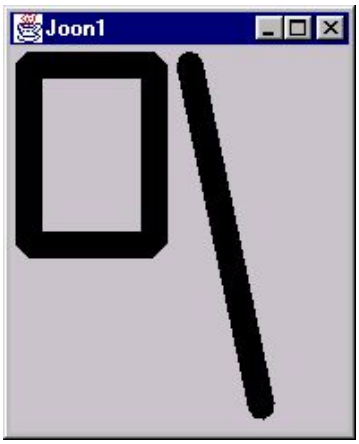
Graafiline kontekst, joone ja tausta omadused, kujundid.

Põhilised joonistamisfunktsioonid on klassis java.awt.Graphics. Alates versioonist 1.2 pakub laiendatud joonistamisvõimalusi eelmise alamklass Graphics2D. Kui esimesel juhul tuli alati arvutada ekraanikoordinaatides ning joone laiuseks oli üks punkt, siis siin võib valida omale sobiva taustsüsteemi ning ka joonistamisel saab enam parameetreid määrata. Kuna Graphics2D on klassi Graphics alamklass, siis ta oskab kõike mida eellanegi ning vajadusel saab temaga joonistada samade meetoditega, mis klassist Graphics omale sisse harjunud. Ühilduvuse huvides on komponendi meetodi paint parameetriks endiselt Graphics, kuid tegelikult antakse sellesse meetodisse joonistamiseks isend, kes suudab ka Graphics2D klassis kirjeldatud operatsioone täita. Juhul, kui spetsiifilisi omadusi vajatakse, tuleb muutuja tüüp enne teisendada.

Joone omadused

Joone tõmbamisel saab Graphics2D juures klassi BasicStroke abil määrata joone laiust, otsa kuju ning kahe joone ühendust (võimalused leiad API dokumentatsioonist).

```
import java.awt.*;
public class Joon1 extends Canvas{
  public void paint(Graphics alggr){
    Graphics2D g=(Graphics2D)alggr;
    float laius=15;
    int jooneots=BasicStroke.CAP_ROUND;
    int uhendus=BasicStroke.JOIN_BEVEL;
    g.setStroke(new BasicStroke(
                    laius, jooneots, uhendus));
    g.drawRect(10, 10, 70, 100);
    g.drawLine(100, 10, 140, 200);
  }
}
```



Punktiirjoon

Punktiiri puhul tuleb määrata, millise pikkusega on punktiiri kriipsud, lisaks sellele hulk joontega seotud andmeid. Kuid kui korra on sulepea (Stroke) valmis tehtud, võib sellega rahumeeli jooni tõmmata nii palju kui vaid soovid – nii nagu eelmises näites. Kui vaadata sulepea koostamise käsklust

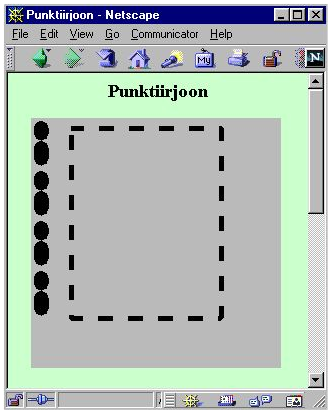
```
BasicStroke bs1=new BasicStroke(
                  15, ots, yhendus, yhendusemaxpikkus, punktiir, punktiirinihe);
```

siis 15 tähendab tõmmatava joone laiust. Ots näitab, millised (ümarad, kandilised) tuleb joone otsad teha. Muutuja yhendusemaxpikkus on enamasti tarbetu, vaja võib teda minna vaid siis, kui miskis kujundis ühinevad jooned väga väikese nurga all ning tekib oht pika väljaulatuva nurga moodustumiseks. Siis selle muutuja järgi vaadatakse, mitmest punktist vastav nurk pikemaks ei tohi minna. Eelnevalt loodud massiiv punktiir näitab, kuidas punktiirjoones vahelduvad kriipsud ja vahed. Nagu siin näites eespoolt võib piiluda, on esimese kriipsu pikkuseks määratud 5 punkti, sellele peaks järgnema viieteistpunktiline vahe. Edasi kümnepunktiline kriips ning selle järele kahekümnepunktine vahe. Siis taas algusest peale, et kahekümnepunktise vahe järele jälle viiene kriips, siis viieteistkümnene vahe ning nõnda edasi. Muutujast punktiirinihe vaadatakse, kus kohalt mustriga peale hakata. Kui nihe on 0, siis algab esimene kriips joone otsast. Kui mõni suurem arv, siis on esimese kriipsu algus vastavalt nihutatud.

Teine BasicStroke on loodud ühe käsuga, enamjaolt on väärtused otse konstruktorisse kirjutatud. `new float[]{15}` loob üheelemendilise massiivi otse kohapeal.

```
import java.awt.*;
import java.awt.geom.*;
import java.applet.*;
public class Punktiir1 extends Applet{
  float laius=5;
  int ots=BasicStroke.CAP_ROUND, yhendus=BasicStroke.JOIN_MITER;
  float yhendusemaxpikkus=10;
  float[] punktiir={5, 15, 10, 20}; //kriips, vahe, kriips, vahe, ...
```

```
  float punktiirinihe=0;
  BasicStroke bs1=new BasicStroke(
                15, ots, yhendus, yhendusemaxpikkus, punktiir, punktiirinihe);
  BasicStroke bs2=new BasicStroke(laius, BasicStroke.CAP_BUTT, BasicStroke.JOIN_ROUND,
                   yhendusemaxpikkus, new float[]{15}, 2);
  public void paint(Graphics g){
    Graphics2D g2=(Graphics2D)g;
    Line2D l1=new Line2D.Float(10, 10, 10, 200);
    Rectangle2D r1=new Rectangle2D.Float(40, 10, 150, 190);
    g2.setStroke(bs1);
    g2.draw(l1);
    g2.setStroke(bs2);
    g2.draw(r1);
  }
  public static void main(String argumendid[]){
    Frame f=new Frame("Punktiir");
    f.add(new Punktiir1());
    f.setSize(250, 250);
    f.setVisible(true);
  }
}
```



Joonistusala piiramine

Joonistamisala on võimalik piirata käsuga clip, andes ette kujundi, mille piires tohib joonistada. Siin näites määratakse joonistamise alaks ellipsi pind ning seejärel joonestatakse seest täidetud ristkülik. Tulemusena tekib ekraanile vaid ellipsi ning ristküliku ühisosa.

```
import java.awt.*;
import java.awt.geom.*;
public class Kujund1 extends Canvas{
  public void paint(Graphics alggr){
    Graphics2D g=(Graphics2D)alggr;
    Shape kujund=
        new Ellipse2D.Float(10, 10, 300, 100);
    g.clip(kujund);
    g.fillRect(20, 20, 400, 60);
  }
}
```

Venitamine, keeramine

Joonistuspinda saab liigutada, venitada ja keerata. AffineTransform'i abil saab määrata, kuidas ja kui palju. Esimese näite puhul lükatakse koordinaatide alguspunkti saja ühiku võrra paremale ning alla.

```
import java.awt.*;
import java.awt.geom.AffineTransform;
public class Transform2 extends Canvas{
  public void paint(Graphics alggr){
    Graphics2D g=(Graphics2D)alggr;
    g.setTransform(AffineTransform.getTranslateInstance(100, 100));
    g.fillRect(20, 20, 40, 20);
  }
}
```

rida

```
g.setTransform(AffineTransform.getScaleInstance(3, 1.5));
```

suurendab joonist x-telje suunas 3 ning y-suunas 1,5 korda.

```
g.setTransform(AffineTransform.getRotateInstance(Math.PI/4, 100, 75));
```

keerab joonist Pi/4 ehk 45 kraadi võrra ümber punkti 100, 75

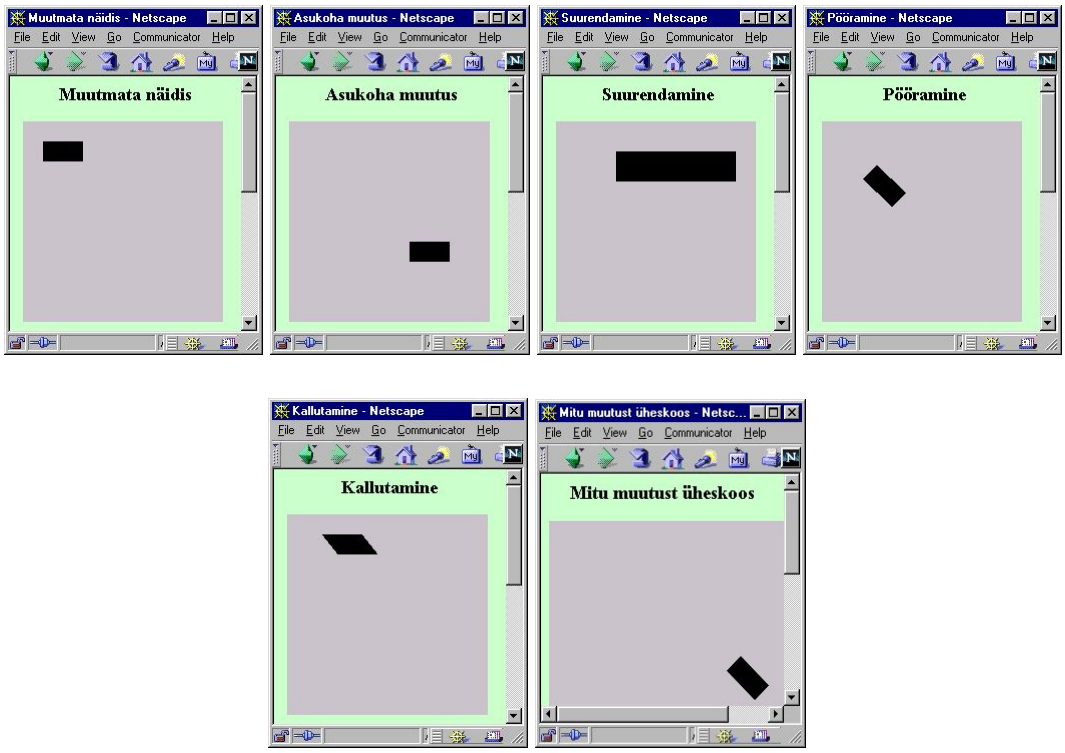
```
g.setTransform(AffineTransform.getShearInstance(Math.PI/4, 0));
```

keerab püsttelge Pi/4 võrra.

Kui soovida mitut muutust üheskoos, siis võib luua AffineTransform tüüpi isendi ning talle soovitud muutusi järjekorras rakendada.

```
import java.awt.*;
import java.awt.geom.AffineTransform;
public class Transform6 extends Canvas{
  public void paint(Graphics alggr){
```

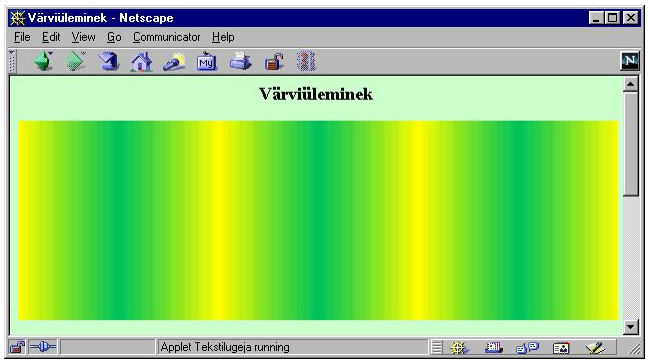
```
    Graphics2D g=(Graphics2D)alggr;
    AffineTransform tr=new AffineTransform();
    tr.rotate(Math.PI/4, 50, 100);
    tr.translate(150, 0);
    g.setTransform(tr);
    g.fillRect(20, 20, 40, 20);
  }
}
```



Värviüleminek

Joonistamisel saab lisaks ühele värvile soovi korral pinda katta ka nii värviülemineku kui piltidest koostatud mustriga. Soovitud katmisstiil tuleb määrata Graphics2D meetodiga setPaint. Värviülemineku andmeid kannab GradientPaint. Konstruktoris tuleb määrata kaks punkti ning kummagi punkti juurde kuuluv värv. Nendes punktides vastab joonise värv sinna määratud värvile, punkte ühendaval sirget mööda muutub värv sujuvalt ühest värvist teiseks. Tavajuhul jääb kummagi punkti "selja taha" punktile vastav värv, kuid kui lisada konstruktorisse tõeväärtusmuutuja, siis saab panna värvi tsükliliselt lainetama nii, et laine pikkuseks jääb kahe punkti vahe.

```
import java.awt.*;
import java.awt.geom.*;
public class Kujund2a extends Canvas{
  public void paint(Graphics alggr){
    Graphics2D g=(Graphics2D)alggr;
    g.setPaint(new GradientPaint(0, 100, Color.yellow,
               getSize().width, 100, new Color(0, 200, 100)
    ));
    g.fillRect(0, 0, getSize().width, getSize().height);
  }
}
```

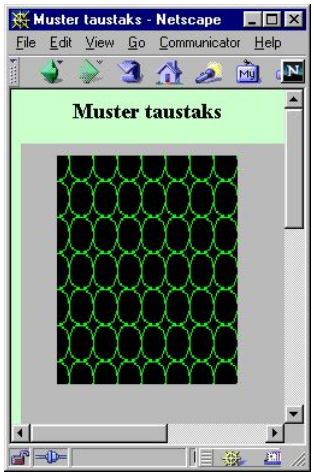


Muster

Et taustaks saaks mustrit panna, selleks tuleb kõigepealt mustripilt koostada või välja otsida ning alles seejärel saab määrata, et joonistatava kujundi värviks on pidevalt korduv loodud muster. Järgnevas näites on mustritükiks lihtsalt roheline ring mustal taustal. `BufferedImage bi` hoiab eneses loodavat pilti, TexturePaint tp aga juba hoolitseb, kuidas pilt korralikult õigesse kohta paigutada. Konstruktoris luuakse pildile graafiline kontekst, mille abil joonistatakse pildile 20*20 punkti laiune roheline ring. Rida `tp=new TexturePaint(bi, new Rectangle(0, 0, 20, 30));` tähendab, et olemasolev pilt (algse suurusega 20*20 punkti) asub loodud `TexturePaint`'is olema 20 punkti lai ning 30 kõrge, seega pikkust pidi välja venitatud.

Joonistuskäsu `paint` sees võetakse pakutud `Graphics` vastu `Graphics2D`-na, et õnnestuks vajalikke käske (`setPaint`) kasutada. Edasi joonistatakse pinnale ristkülik. Kuna aga joonistusmustriks oli määratud pilt, siis loodud ristkülik näebki välja mustrilisena.

```
import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;
import java.applet.*;
public class Mustritaust extends Applet{
  BufferedImage bi=new BufferedImage(
                  20, 20, BufferedImage.TYPE_INT_RGB);
  TexturePaint tp;
  public Mustritaust(){
    Graphics2D big=bi.createGraphics();
    big.setColor(Color.green);
    big.drawOval(0, 0, 20, 20);
    tp=new TexturePaint(bi, new Rectangle(0, 0, 20, 30));
  }
  public void paint(Graphics g){
    Graphics2D g2=(Graphics2D)g;
    Rectangle2D r1=new Rectangle2D.Float(30, 10, 150, 190);
    g2.setPaint(tp);
    g2.fill(r1);
  }
  public static void main(String argumendid[]){
    Frame f=new Frame("Muster taustaks");
    f.add(new Mustritaust());
    f.setSize(250, 250);
    f.setVisible(true);
  }
}
```



Värviüleminekuga tekst

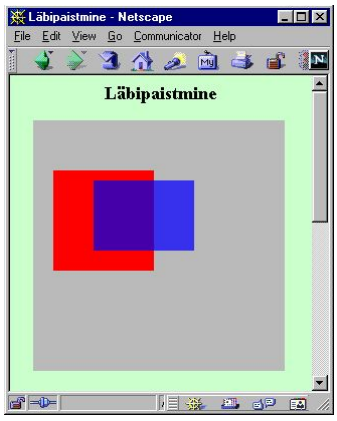
Joonistusala piiramist ning värviüleminekut kombineerides saab päris keeruka ja ilusa kujundusega pildi kokku panna.

```
import java.awt.*;
import java.awt.geom.*;
import java.awt.font.*;
public class Kujund2 extends Canvas{
  public void paint(Graphics alggr){
    Graphics2D g=(Graphics2D)alggr;
```

```
      TextLayout tl=new TextLayout("Kassikene",
        new Font("Arial", Font.PLAIN, 100),
        new FontRenderContext(null, false, false)
      );

      g.setPaint(new GradientPaint(200, 0, Color.yellow,
                 200, getSize().height, new Color(255, 200, 0)
      ));
      g.fillRect(0, 0, getSize().width, getSize().height);

      g.clip(tl.getOutline(AffineTransform.getTranslateInstance(20, 120)));
      g.setPaint(new GradientPaint(0, 100, new Color(66, 66, 0),
                 getSize().width, 100, new Color(0, 200, 100)
      ));
      g.fillRect(0, 0, getSize().width, getSize().height);
    }
  }
```

Soovides jätta all olevat pilti joonistatava kujundi alt läbi paistma, tuleb enne peale joonistamist määrata `Composite`, mis paluks peale joonistataval alumine vaid osaliselt ära katta ning jätta tulemuseks vanaga segatud värvid.

```
import java.awt.*;
import java.applet.*;
public class Labipaistmine extends Applet{
  public void paint(Graphics g){
    Graphics2D g2=(Graphics2D)g;
    g2.setColor(Color.red);
    g2.fillRect(20, 50, 100, 100);
    g2.setComposite(AlphaComposite.getInstance(
            AlphaComposite.SRC_OVER, 0.7f));
          //70% joonistab, 30% paistab alt läbi
    g2.setColor(Color.blue);
    g2.fillRect(60, 60, 100, 70);
  }
  public static void main(String argumendid[]){
    Frame f=new Frame("Läbipaistmine");
    f.add(new Labipaistmine());
    f.setSize(250, 250);
    f.setVisible(true);
  }
}
```



Kujundi äärejooned

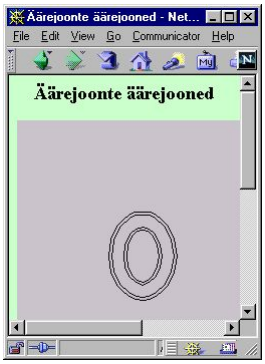
Kujundi (näiteks ellipsi) äärejooned annab BasicStroke meetod createStrokedShape. Nii näiteks on võimalik kujundist vaid äärejooned välja joonistada või siis teise värviga esile tuua.

```
public void paint(Graphics alggr){
    Graphics2D g=(Graphics2D)alggr;
    BasicStroke b1=new BasicStroke(15);
    Shape kujund=b1.createStrokedShape(
      new Ellipse2D.Float(100, 100, 50, 70)
    );
    g.fill(kujund);
    g.setColor(Color.green);
    g.draw(kujund);
  }
```

Äärejoonte äärejooned

Kuna nii ellips on kujund ning ka jämedajoonelise ellipsi äärejooned on samuti kujundid, siis juhul, kui annan äärejoontele jämeduse, võin ka nendelt omakorda äärejooned küsida. Nii saan tulemuseks juba neli joont: kaks üksteisele lähedal asuvat ovaali sees ning teised kaks ovaali välisringis.

```
public void paint(Graphics alggr){
    Graphics2D g=(Graphics2D)alggr;
    BasicStroke b1=new BasicStroke(15);
    BasicStroke b2=new BasicStroke(3);
    Shape kujund=b1.createStrokedShape(
      new Ellipse2D.Float(100, 100, 50, 70)
    );
    Shape kujund2=b2.createStrokedShape(kujund);
    g.draw(kujund2);
 }
```

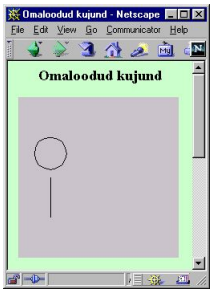


Kujundi koostamine

Kui sooviksin nende joontega midagi eraldi teha, näiteks neid igaüht isevärvi värvida, siis on mul võimalik kujund joonteks jagada klassi PathIterator abil. Selle abil saan kätte iga joone andmed. Sirgjoonel piisab kahest punktist. Kolme punkti abil määratakse kõverjoon, kus kaks punkti on otspunktideks ning kolmas näitab, milline peab kaar tulema. Nelja punktiga määratud joone puhul on samuti kaks otspunktideks, ülejäänud kahe punkti abil aga määratakse joone suunda otspunktist väljumisel.

Uusi kujundeid aitab kombineerida klass GeneralPath. Talle tuleb lihtsalt öelda millise koha peale joon või ring või muu olemasolev kujund paigutada. Kriipsujuku saab tema abil küllalt kergesti valmis ning siis võib seda kasutada nagu iga muud kujundit.

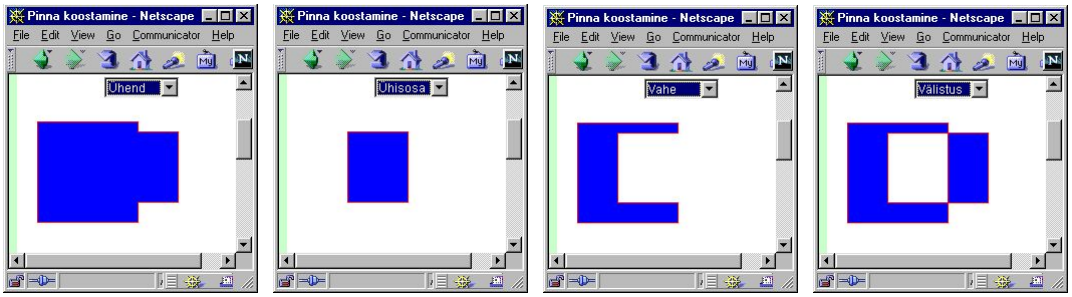
```
public void paint(Graphics alggr){
    Graphics2D g=(Graphics2D)alggr;
    GeneralPath gp=new GeneralPath();
    gp.append(new Ellipse2D.Float(20, 50, 40, 40), false);
    gp.append(new Line2D.Float(40, 100, 40, 150), false);
    g.draw(gp);
}
```



Tehted kujunditega

Kujundit luues on võimalik olemasolevatega ka keerukamaid tehteid teha. Näiteks kui soovida koostada kuuveerandikku, siis võib kõigepealt teha ühe väiksema ringi ning siis sellest lahutada maha suurem ring, nii et vaid väike osa algsest jääb alles. All olevas näites on võetud kas osaliselt kattuvat ristkülikut ning näidatud, mis tehteid nendega läbi viia saab. Liites esimesele juurde teise saame kujundite ühendi, kus mõlema kujundi pinnad on liidetud. Ühisosa tekitab käsk `intersect` – alles jääb vaid osa pinnast, mis kuulub mõlema kujundi koosseisu. Lahutamise (`subtract`) korral jääb esialgsest pinnast alles vaid osa, mis teise alla ei kuulu. Välistuse (exclusiveOr) puhul jääb mõlemast pinnast alles vaid osa, kus üks pind teisega ei kattu.

```
import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;
import java.awt.event.*;
import java.applet.*;
public class Pind extends Applet implements ItemListener{
  Area r1=new Area(new Rectangle2D.Float(20, 50, 100, 100));
  Area r2=new Area(new Rectangle2D.Float(60, 60, 100, 70));
  Choice valik=new Choice();
  public Pind(){
    valik.add("Ühend");
    valik.add("Ühisosa");
    valik.add("Vahe");
    valik.add("Välistus");
    add(valik);
    valik.addItemListener(this);
  }
  public void paint(Graphics g){
    Area pind=new Area();
    pind.add(r1);
    String s=valik.getSelectedItem();
    if(s.equals("Ühend"))pind.add(r2);
    if(s.equals("Ühisosa"))pind.intersect(r2);
    if(s.equals("Vahe"))pind.subtract(r2);
    if(s.equals("Välistus"))pind.exclusiveOr(r2);
    Graphics2D g2=(Graphics2D)g;
    g2.setColor(Color.blue);
    g2.fill(pind);
    g2.setColor(Color.red);
    g2.draw(pind);
  }
  public void itemStateChanged(ItemEvent e){
    repaint();
  }
  public static void main(String argumendid[]){
    Frame f=new Frame("Muster taustaks");
    f.add(new Pind());
    f.setSize(250, 250);
    f.setVisible(true);
  }
}
```



Ülesandeid

- Joonista ring joone laiusega 25 punkti.
- Määra joonistusvärviks üleminek punaselt kollasele
- Määra taustamustriks väikesed kriipsujukud.
- Muuda joon punktiiriks.
- Määra joonistusala nii, et sinna jääb vaid osa ringjoont
- Jäta taust ringi alt veidi läbi paistma.

Swing

Operatsioonisüsteemist sõltumatud komponendid, kujundus

Lisaks kümnekonnale paketis `java.awt` asuvale komponendile saab kasutajaga suhtlemiseks tarvitada ka paketi `javax.swing` graafilisi komponente. Nagu kirjeldatud, palutakse `awt`-komponendid joonistada operatsioonisüsteemil, `swing`-komponente joonistatakse Java vahenditega. Sellest tulenevalt näevad esimesed välja nii nagu vastavas operatsioonsiüsteemis tavaks, `swing`i nupp või silt aga on igal pool tavajuhul peaaegu ühesugune. `UIManager`i abil aga on võimalik panna ka `swing`-komponente vastavalt operatsioonisüsteemile välja nägema.

Swingi graafikakomponendid algavad tähega J. Tõenäoliselt seetõttu, et neid oleks kerge eristada analoogilistest awt komponentidest. Järgnevas näiteks on raamiks `JFrame`, selle sees on silt `JLabel` ning nupp `JButton`. Nii nupule kui sildile (ja ka mitmetele muudele komponentidele) saab tema ilmestamiseks määrata ikooni. `ImageIcon` loob ikooni kasutades aluseks pildifaili, kuid vajadusel saab ikooni ka käskude abil joonistada. Kõikidele `swing`i komponentidele saab määrata `ToolTipText`'i. Seda näidatakse ekraanile juhul, kui kasutaja on hiirega vastava komponendi peale liikunud. Enamasti vastav tekst seletab komponendi otstarvet või annab kasutajale tegutsemissoovitusi. Korraldus `setMnemonic` lubab klahvikombinatsiooni (Alt + täht) võrdsustada hiirega nupule vajutamisele.

Komponentide raami lisamisel tuleb swingi puhul määrata, millisesse kihti ta paigutada. Harilikult kasutatava alumise kihi saab kätte `getContentPane()` abil. Pealmist kihti nimetatakse `GlassPane` ning vahepealsetesse kihtidesse paigutamiseks saab kasutada `LayeredPane` vahendeid. Kihtidega mängides saab komponente mitmesse kihti paigutada.

```
import java.awt.*;
import javax.swing.*;
public class Pildid{
  public static void main(String argumendid[]){
    JLabel silt=new JLabel("Maja silt");
    Font suurkiri=new Font("Serif", Font.BOLD+Font.ITALIC, 30);
    Icon majapilt=new ImageIcon("maja.gif");
    silt.setFont(suurkiri);
    silt.setIcon(majapilt);

    JButton nupp=new JButton("Maja nupp", majapilt);
    nupp.setToolTipText("Head vajutamist!");
    nupp.setMnemonic(java.awt.event.KeyEvent.VK_M);
   JFrame f=new JFrame("Sildiraam");
   Container p=f.getContentPane();
   p.setLayout(new GridLayout(2, 1));
   p.add(silt);
   p.add(nupp);
   f.pack();
   f.setVisible(true);
  }
}
```

HTML-kujundus

Swingi komponentidel näidatavat teksti saab HTMLi abil kujundada. Täpne väljanägemine võib sõltuda intepretaatorist, kuid selliselt on programmi väljanägemist lihtsam pilkupüüdvaks muuta kui awt vahenditega kujundades. Nagu lihtsat teksti kandva sildi puhul, nii ka siin on võimalik programmi töö käigus sildi sisu muuta.

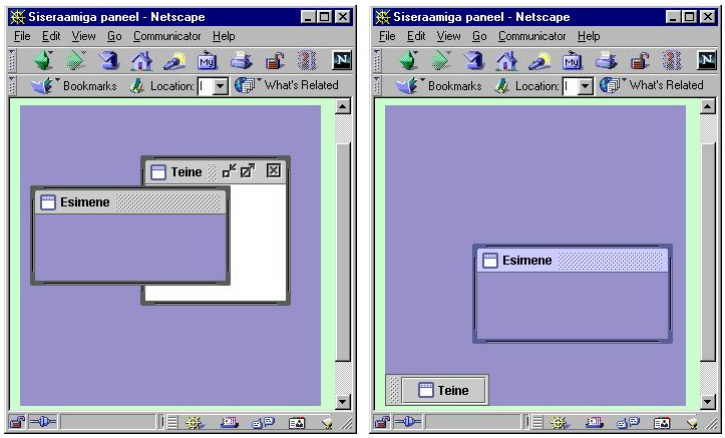
```
import javax.swing.*;
public class HtmlLabel{
  public static void main(String argumendid[]){
    JFrame f=new JFrame("Kujundatud silt");
    JLabel silt=new JLabel(
      "<html><h2>Pealkirja</h2>\n"+
      "ja <font color=red>punase tekstiga</font> silt</html>"
    );
```

```
    f.getContentPane().add(silt);
    f.pack();
    f.setVisible(true);
  }
}
```

Sisemised raamid

Mõnes programmis on näha, et pearaami sees on omaette väiksemad raamid. Nii näiteks Wordi puhul võib iga tekst olla lahti omaette raamis, need aga omakorda suure Wordi raami sees. Sellist olukorda Java keskkonnas saab tekitada `JInternalFrame` abil. Nemad käituvad `JDesktopPane` sees samuti nagu harilikud raamid suure ekraani sees. Tema sees paiknevasse paneeli saab lisada komponente nagu ikka. Ka sisemisi raame saab muuta ikooniks alla serva, suurendada ja vähendada. Kui suure raami teateid püüab `WindowListener`, siis siseraamiga toimuva teada saamiseks aitab `InternalFrameListener`. Vorm on küll erinev, kuid võimalused samad. Näiteks koostab JDesktopPane tüüpi komponendi eraldi staatiline meetod. Rakendis paigutatakse komponent ekraanile`init` meetodi sees, käsurealt käivitades luuakse raam ning siis paigutatakse komponent raami sisse.

```
import javax.swing.*;
public class SiseraamigaRaam extends JApplet{
   static JDesktopPane looRaamiPaneel(){
    JInternalFrame siseraam1=new JInternalFrame("Esimene");
    JInternalFrame siseraam2=new JInternalFrame("Teine",
                                 true, true, true, true);
    siseraam2.getContentPane().add(new JTextArea());
    JDesktopPane paneel=new JDesktopPane();
    siseraam1.setSize(200, 100);
    siseraam1.setLocation(10, 80);
    siseraam1.setVisible(true);
    paneel.add(siseraam1);
    try{siseraam1.setSelected(true);}catch (Exception e){}
    siseraam2.setVisible(true);
    paneel.add(siseraam2);
    siseraam2.setSize(150, 150);
    siseraam2.setLocation(120, 50);
    return paneel;
  }
  public void init(){
    getContentPane().add(looRaamiPaneel());
  }
  public static void main(String argumendid[]) throws Exception{
    JFrame f=new JFrame("Kest");
    f.setContentPane(looRaamiPaneel());
    f.setSize(300, 300);
    f.setVisible(true);
  }
}
```

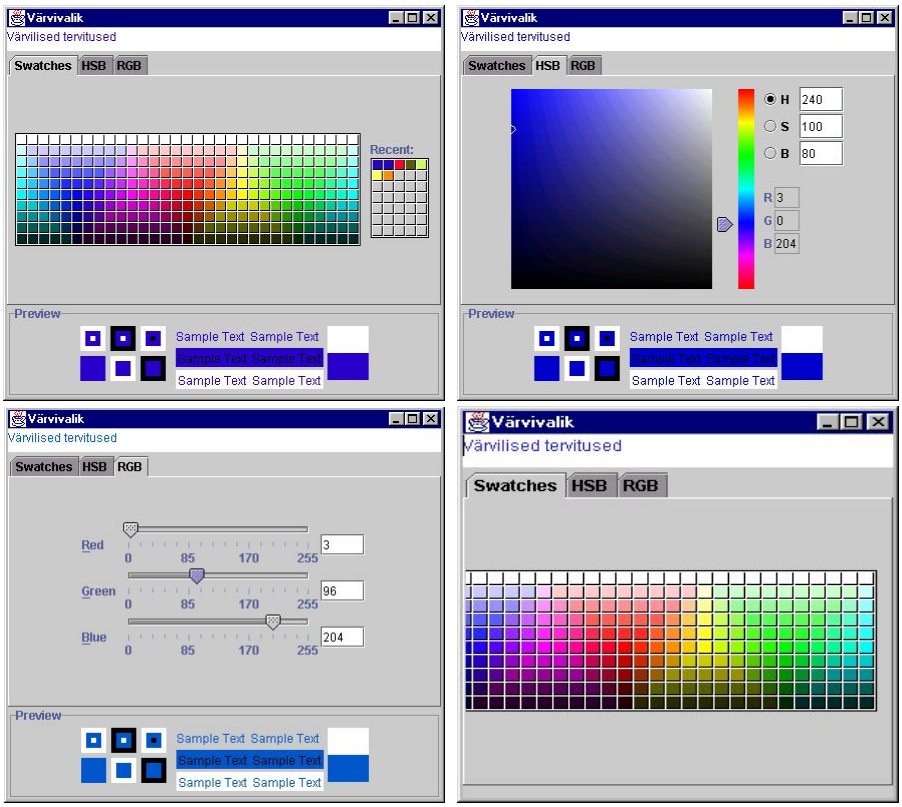


Värvivalija

Kasutajapoolseks värvi valimiseks saab vajadusel kirjutada ise dialoogiakna, kust hiirega omale sobiv värv leida võimalik on. `Swing`i all aga on programmeerimisvaeva vähendamiseks loodud

komponent `JColorChooser`, mille abil kasutaja võib pakutavate hulgast sobiva värvi välja valida. Värvivalikuks pakub komponent kolme võimalust: esimesel juhul saab hiirega vajutada sobivat värvi ruudule. Teisel puhul saab valida värvi ning teiselt skaalalt värvile vastava tumeduse. Kolmandas valikuaknas lastakse kasutajal määrata punase, rohelise ning sinise vahekord loodavas värvis. Vaikimisi kujul näitab komponent otsitavat värvi mitmesuguste kujundite ning ingliskeelse teksti peal. Selle kujundi saab aga vajadusel programmi ilmele sobivama või hoopis tühja pisikese paneeli vastu välja vahetada (järgnevas näites vastav rida välja kommenteeritud). Värvivaliku registreerimiseks ning temale reageerimiseks saab kasutada kuularit. Värvivaliku klassi juurde on loodud meetodid ka dialoogiakna kaudu värvi valimiseks.

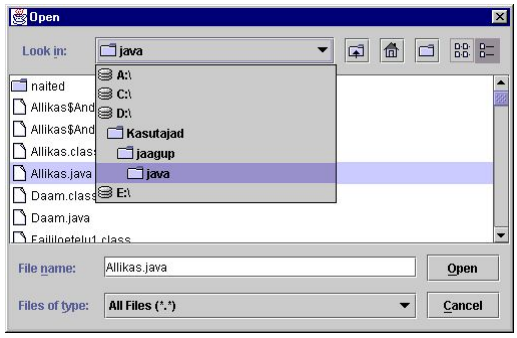
```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.BorderLayout;
public class Varvivalik{
  public static void main(String argumendid[]){
    final JTextArea tekstiala=new JTextArea("Värvilised tervitused");
    final JColorChooser valija=new JColorChooser();
//    valija.setPreviewPanel(new JPanel());
    valija.getSelectionModel().addChangeListener(
      new ChangeListener(){
        public void stateChanged(ChangeEvent e){
          tekstiala.setForeground(valija.getColor());
        }
      }
    );
    JFrame f=new JFrame("Värvivalik");
    java.awt.Container p=f.getContentPane();
    p.add(tekstiala, BorderLayout.CENTER);
    p.add(valija, BorderLayout.SOUTH);
    f.setSize(300, 500);
    f.setVisible(true);
  }
}
```



Failinime valija

Ka failinime valimise dialoogi saab ise suurema vaevata kirjutada, kuid swingi klass JFileChooser on juba vastavaks otstarbeks loodud ilusasti kujundatud komponent. Saab määrata, millisest kataloogist alates faili otsima saab hakata. Meetod `showOpenDialog` avab dialoogi ning programm jääb kasutajapoolset valikut ootama. Kui fail on valitud või valimine tühistatud, läheb programm edasi ning komponendi meetodite kaudu saab teada kasutaja vastuse. Kui soovitud fail on käes, saab temaga samuti toimida nagu failiga ikka, s.t. tema kohta infot küsida, sealt lugeda või sinna kirjutada.

```
import javax.swing.*;
import java.io.*;
public class Failivalik{
  public static void main(String argumendid[]){
    JFileChooser valija=new JFileChooser(new File("."));
    valija.showOpenDialog(new JFrame());
    System.out.println("Valiti "+valija.getSelectedFile());
  }
}
```




```
D:\Kasutajad\jaagup\java\naited\gr\swing>java
Failivalik
Valiti D:\Kasutajad\jaagup\java\Allikas.java
```

Failivalikut saab veidi programmiga kohandada. Näiteks võib määrata avamisnupu peal olevat kirja. Samuti võib lubada korraga mitut faili valida. Järgnevas näites lisatakse filtri abil võimalus eraldi vaid pildifailide hulgast kasutajal sobivat otsida. Filtri loomisel tuleb üle katta meetodid accept ning getDescription. Esimesele antakse järjekorras ette kõik failid, mida parasjagu oleks võimalik valida. See meetod peab igaühe kohta neist ütlema, kas vastavat faili kasutajale näidata või mitte. Meetod getDescription väljastab filtrile sobivate failide ühisnimetaja, siin näites "Pildifailid".

```
import javax.swing.*;
import javax.swing.filechooser.FileFilter;
import java.io.*;
public class Failivalik2{
  public static void main(String argumendid[]){
    JFileChooser valija=new JFileChooser(new File("."));
    valija.addChoosableFileFilter(new Pildifilter());
    valija.showDialog(new JFrame(), "Vali fail");
    System.out.println("Valiti "+valija.getSelectedFile());
  }
}

class Pildifilter extends FileFilter{
  public boolean accept(File f){
    String failinimi=f.getName();
    if(failinimi.endsWith(".gif")|failinimi.endsWith(".jpg"))
      return true;
    else return false;
  }
  public String getDescription(){
    return "Pildifailid ";
  }
}
```

Puud

Infohulgas orienteerumiseks saab andmeid esitada puuna. Siis on kasutajal võimalik ekraanilt kasutu kõrvaldada ning hierarhia abil enesele sobiv üles leida. Puu abil on näiteks esitatud failid ja kataloogid WindowsExploreris. Puusse saab lisada kõiki objekte. Vaikimisi juhul määrab `JTree` ise okstele ja lehtedele sobivad ikoonid ning objekti kirjelduseks kasutab sõnet mille väljastab selle `toString` meetod. Vajadusel aga võib nii ikooni kui kirjeldust muuta.

Puu hierarhia saab kokku panna `DefaultMutableTreeNode` abil. Kui element on reas viimane, on ta leht, keskel oks ning algul juur. Puu ekraanile kuvamiseks tuleb luua `JTree`, kellele määrata juur, millest alates elemente näidata tuleb. Siin näites on pandud juureks ülikool, tema alla paar teaduskonda ning teaduskonna alla mõni õppetool. Võib jääda mulje, nagu tuleks puu loomiseks hirmus palju kirjutada, kuid suuremate andmehulkade korral saab luua või kasutada olemasolevaid alamprogramme ning puu tegemine polegi kuigi keeruline.

```
import javax.swing.*;
import javax.swing.tree.*;
public class Puu{
  public static void main(String argumendid[]){
    DefaultMutableTreeNode juur=new DefaultMutableTreeNode("Ülikool");
    DefaultMutableTreeNode teaduskond=new DefaultMutableTreeNode(
       "Matemaatika-Loodusteaduskond");
    teaduskond.add(new DefaultMutableTreeNode("Informaatika õppetool"));
    teaduskond.add(new DefaultMutableTreeNode("Bioloogia  õppetool"));
    juur.add(teaduskond);
    juur.add(new DefaultMutableTreeNode("Kultuuriteaduskond"));
    JTree puu=new JTree(juur);
    JFrame f=new JFrame("Puu");
    f.getContentPane().add(puu);
    f.setSize(200, 200);
    f.setVisible(true);
  }
}
```

Kasutaja tegevuse registreerimiseks saab tarvitada mitmesuguseid kuulareid. Saab teada, millal ta puu nähtavat osa suurendas või vähendas, mis osa puust nähtav on, millal mõni element valiti. Ka pärast puu loomist saab temasse elemente lisada ja sealt eemaldada. Käsklus puu.putClientProperty ("JTree.lineStyle", "Angled"); palub puu osad omavahel joontega ühendada. Siin näites trükitakse igal valikul välja valitud komponent. TreeSelectionEvent'i meetod getPath() annab tulemuseks kogu rea alates juurest kuni märgitud leheni. Selle kaudu oleks võimalik ükskõik millise tee peale jääva komponendi poole pöörduda. Meetod getLastPathComponent() annab tulemuseks tee viimase elemendi, ehk selle, millele kasutaja vajutas.

```
import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;
import java.awt.BorderLayout;
public class Puu2a extends JApplet{
   static JTextField tekstikast=new JTextField();
   static JPanel looPuu(){
    DefaultMutableTreeNode juur=new DefaultMutableTreeNode("Ülikool");
    DefaultMutableTreeNode teaduskond=new DefaultMutableTreeNode(
       "Matemaatika-Loodusteaduskond");
    teaduskond.add(new DefaultMutableTreeNode("Informaatika õppetool"));
    teaduskond.add(new DefaultMutableTreeNode("Bioloogia  õppetool"));
    juur.add(teaduskond);
    juur.add(new DefaultMutableTreeNode("Kultuuriteaduskond"));
    JTree puu=new JTree(juur);
    puu.addTreeSelectionListener(
      new PuuKuular()
    );
```

```
     puu.putClientProperty("JTree.lineStyle", "Angled");
     JPanel paneel=new JPanel(new BorderLayout());
     paneel.add(puu);
     paneel.add(tekstikast, java.awt.BorderLayout.SOUTH);
     return paneel;
    }
    public void init(){
      getContentPane().add(looPuu());
    }
    public static void main(String argumendid[]){
     JFrame f=new JFrame("Puu");
     f.getContentPane().add(looPuu());
     f.setSize(200, 200);
     f.setVisible(true);
    }
 }

 class PuuKuular implements TreeSelectionListener{
   public void valueChanged(TreeSelectionEvent e){
     Puu2a.tekstikast.setText(e.getPath().getLastPathComponent()+"");
   }
 }
```

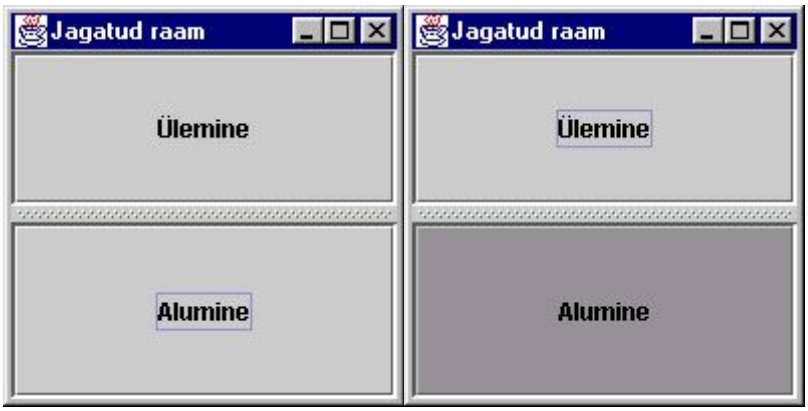
Paigutus

Jaotuspaneel

`JSplitPane` võimaldab temale eraldatud pinna jaotada kahe komponendi vahel. Tuleb vaid määrata, kas pind jaotatakse kaheks horisontaalselt või vertikaalselt ning komponendid, mis kummasegi ossa panna. Lisameetoditega saab määrata ja muuta jagamise kohta, samuti kasutajapoolset vahepiiri nihutamise võimalusi. Jagatud paneelidena näevad välja mitut lehte sisaldavad brauseri aknad, kus vasakul näiteks sisukord ja paremal sisu.

Kui koodi vaadata, siis üles on pandud JButton, alla JBoggleButton. Viimase omapäraks on, et esimest korda vajutades jääb nupp sisse (tumedaks), alles teisel korral tuleb välja tagasi.

```
import javax.swing.*;
public class Jaotuspaneel{
  public static void main(String argumendid[]){
    JSplitPane paneel=new JSplitPane(
      JSplitPane.VERTICAL_SPLIT,
      new JButton("Ülemine"),
      new JToggleButton("Alumine")
    );
    JFrame f=new JFrame("Jagatud raam");
    f.getContentPane().add(paneel);
    f.setSize(200, 200);
    f.setVisible(true);
  }
}
```



Valikupaneel

Kui määranguaknas on võimalusi rohkem kui kasutajale korraga mõistlik näidata on, siis `JTabbedPane` abil saab muud paneelid ülekuti paigutada. Sarnaselt on loodud näiteks Exceli lahtrimäärangute dialoogiaken, kus ühel paneelil saab määrata andmete tüüpi, teisel kujundust jne. Soovi korral saab paneelivaliku nuppudele lisada ikoonid, eemaldada, vahetada ja lisada paneele, mõne valiku tegemist keelata, automaatselt soovitud paneel esile tuua ning mitmel moel kasutaja tegevuse kohta teateid saada.

```
import javax.swing.*;
public class Valikupaneel{
  public static void main(String argumendid[]){
    JPanel p1=new JPanel(new java.awt.GridLayout(2,1));
    p1.add(new JButton("Ülemine"));
    p1.add(new JButton("Alumine"));

    JTabbedPane paneel=new JTabbedPane();
    paneel.add("Esimene", p1);
    paneel.add("Teine", new JLabel("Suur silt"));

    JFrame f=new JFrame("Valikupaneeli näide");
    f.getContentPane().add(paneel);
    f.setSize(200, 200);
    f.setVisible(true);
  }
}
```

Tööriistariba

Ka sellenimeline abivahend on `Swing`i all täiesti olemas. Kui on soovi nuppe ja muid tööriistu oma silma järgi ümber paigutada, siis tööriistariba peaks selleks parim valik olema. Kokku saab selle panna nagu tavalise paneeli, kuid edaspidi on loodud paneeli võimalik pea vabalt paigutada. Riba saab liigutada nii omaette raamaknana kui paigutada vabalt iga BorderLayout'i vaba serva peale. Piisab vaid riba lohistamisest õige koha lähedusse, kui see juba haakub. Paigalt eemale saab ka täiesti rahumeeli lohistada. Et JAppleti vaikimisi paigutushalduriks ongi BorderLayout, siis neli külge on tööriistaribade jaoks kohe kasutatavad.

Riba ennast annab koostada ning valmis komponente sinna peale panna paari käsuga. Pea pool näiteprogrammist on kulunud ovaali kujutava pildiga ikoone loova klassi valmistamiseks, mille abil on hõlbus äratuntava pildiga nuppe toota. Ikooni loomiseks tuleb teha liidest Icon realiseeriv klass. Siin on see paigutatud Tooriistariba sisemiseks klassiks, et oleks viimasele alati kättesaadav ning kopeerides kaduma ei läheks. Iseenesest aga võib loodav ikoone tootev klass olla rahumeeli eraldi klassina, sel juhul on võimalik ovaalseid ikoone ka teiste programmide kasuks luua. Meetodis `paintIcon` tuleb kirja panna, milline ikoon välja näeb, `getIconWidth()` ning `getIconHeight()` annavad ikooni soovitud suuruse.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class Tooriistariba extends JApplet {
   public Tooriistariba() {
    JToolBar riba = new JToolBar();
    JButton nupp = new JButton("Nupp");
    riba.add(nupp);
    riba.addSeparator();
    riba.add (new Checkbox ("Märkeruut"));
    getContentPane().add (riba, BorderLayout.NORTH);

    riba = new JToolBar();
    Icon icon = new OvaalneIkoon(Color.red);
    nupp = new JButton(icon);
    riba.add(nupp);
    icon = new OvaalneIkoon(Color.blue);
    nupp = new JButton(icon);
    riba.add(nupp);
    icon = new OvaalneIkoon(Color.green);
    nupp = new JButton(icon);
    riba.add(nupp);
    riba.addSeparator();
    icon = new OvaalneIkoon(Color.magenta);
    nupp = new JButton(icon);
    riba.add(nupp);
    getContentPane().add (riba, BorderLayout.SOUTH);
  }
  class OvaalneIkoon implements Icon {
    Color varv;
    public OvaalneIkoon (Color c) {
      varv = c;
    }
    public void paintIcon (Component c, Graphics g,
                           int x, int y) {
      g.setColor(varv);
      g.fillOval (
        x, y, getIconWidth(), getIconHeight());
    }
    public int getIconWidth() {
      return 20;
    }
    public int getIconHeight() {
      return 10;
    }
  }
  public static void main(String[] argumendid){
    JFrame f=new JFrame("Tööriistaribad");
    f.getContentPane().add(new Tooriistariba());
    f.setSize(200, 200);
    f.setVisible(true);
  }
}
```

Ennistamine

Kui üheksakümnendate aastate algul Word 2-te käsu tagasi võtmise võimalus sisse pandi, siis tundus olevat tegemist uue tähelepanuväärse ja omapärase lahendusega. Nüüd on kasutajad programmide juures pea alatise tagasivõtmise võimalusega nii harjunud, et panevad imeks, kui mõne käsu tagajärgi pole võimalik olematuks teha ning peab enne näpuliigutust hoolikalt läbi mõtlema, mis tehtu tulemuseks võib olla. Isegi terveid kataloogitäisi andmed saab Windows Exploreris paigast nihutada ning tagasi panna ilma, et selle peale suuremaid raskusi tekiks. Samuti kannatab mitmeski kohas mitte ainult üht või paari käsku tagasi võtta, vaid annab pea sammhaaval kogu töö algusesse minna.

Sellised võimalused ei teki töösse iseenesest, selle loomiseks tuleb programmi kirjutamisel kõvasti hoolt kanda. Samuti peab arvestama, et igat asja pole siiski võimalik tagasi võtta. Kui kord on soovimatu sisuga kirjad teistele inimestele laiali saadetud, siis on nad teel ja kohal ning meie loodud programmil pole nende kaotamiseks võimalik enam midagi ette võtta. Parimal juhul annab tagasivõtmise käsu juures sihtkohta uus kiri saata, et ärgu vastuvõtja eelmise saabunud kirja sisu liialt südamesse võtku.

Tagasivõtmist annab koodi sisse mitut moodi ehitada, kuid `Swing`i vahendite juures on eraldi selle tarbeks loodud `UndoManager`, mis peaks aitama suuremate tööde puhul toimingule süsteemsemalt läheneda. Et tööd saaks pärast ilusti tagasi võtta, tuleb iga tehtud samm lisada ennistushaldurisse ning

iga sammu juures peab olema kirjas, kuidas samm teha ning kuidas tagasi võtta. Niimoodi tekib sammude ahel, mida mööda on pärastpoole mugav edasi ja tagasi käia.

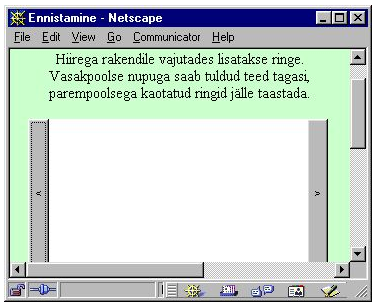
Allpool olevas näites on tagasivõetava programmi koostamine läbi mängitud lihtsa pildiredaktori peal, millele saab joonistada vaid ringe. Lisatud on nupud käskude tagasi võtmiseks ning sama teed pidi edasi liikumiseks.

Iga hiirevajutusega lisatakse näidatavate ringide nimistusse (`Vector`) (muutuva pikkusega massiiv) punkti andmed, mis tähistavad hiirevajutuse asukohta. Nii on mälus kirjas, kuhu joonistusvajaduse korral ringid tekitada ning vähemasti iga operatsiooni järel palutakse ekraanipilt uuendada.

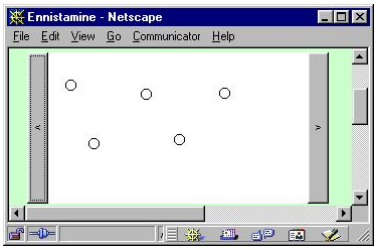
Ennistuse huvides ei lisata punkti andmeid nimistusse otseselt, vaid tehakse veidi pikem ring. Ringi lisamise kirjeldamiseks on loodud eraldi klass `LisatavRing`, mis laiendab klassi `AbstractUndoableEdit`. Klassis on käsud `undo` ja `redo`, kuhu tuleb kirja panna tegevused, mis tuleb sooritada vastavalt tagasi või edasi liikumiseks. Klassis on isendimuutujaks `Point` (paketist `java.awt`) parasjagu lisatava punkti andmete hoidmiseks. Töö lihtsustamiseks on kohe klassi LisatavRing isendi loomisel palutud tal kohe teha läbi edasi liikumisega seotud töö ehk lisada punkti andmed. Sel juhul piisab peaprogrammis sammu tegemisel vaid soovitud koordinaatidega LisatavaRingi loomisest. Edaspidise tagasivõtu sujumiseks tuleb vastav isend UndoableEditEvent'i koosseisus ennistushaldurisse lisada. Nuppude abil tööjärjes edasi-tagasi liikumiseks tuleb vaid ennistushaldurile anda käske undo või redo vastavalt soovitud suunale. Samuti on viisakas enne kontrollida, kas vastavas suunas üldse võimalik liikuda on.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.undo.*;
import javax.swing.event.*;
import java.util.Vector;

public class Ennistus extends JApplet
                      implements ActionListener{
  static Vector ringid=new Vector();
  UndoManager ennistushaldur=new UndoManager();
  Button edasi=new Button(">");
  Button tagasi=new Button("<");

  public Ennistus(){
    getContentPane().add(tagasi, BorderLayout.WEST);
    getContentPane().add(edasi, BorderLayout.EAST);
    tagasi.addActionListener(this);
    edasi.addActionListener(this);
    addMouseListener(
      new MouseAdapter(){
        public void mousePressed(MouseEvent e){
          ennistushaldur.undoableEditHappened(
              new UndoableEditEvent(
                Ennistus.this,
                new LisatavRing(e.getX(), e.getY())
              )
          );
          repaint();
        }
      }
    );
  }
```

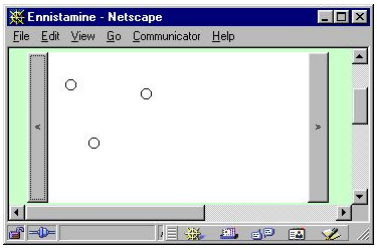
```
  public void paint(Graphics g){
    g.setColor(Color.white);
    g.fillRect(0, 0, getWidth(), getHeight());
    g.setColor(Color.black);
    for(int i=0; i<ringid.size(); i++){
      Point p=(Point)ringid.elementAt(i);
      g.drawOval(p.x-5, p.y-5, 10, 10);
    }
  }

  public void actionPerformed(ActionEvent e){
   try{
    if(e.getSource()==tagasi&&ennistushaldur.canUndo()){
      ennistushaldur.undo();
    }
    if(e.getSource()==edasi&&ennistushaldur.canRedo()){
      ennistushaldur.redo();
    }
   }catch(Exception ex){ex.printStackTrace();}
   repaint();
  }

  public static void main(String argumendid[]){
    JFrame f=new JFrame("Ennistamine");
    f.setSize(250, 200);
    f.setLocation(200, 100);
    f.getContentPane().add(new Ennistus());
    f.setVisible(true);
  }
}

class LisatavRing extends AbstractUndoableEdit{
  Point p;
  public LisatavRing(int x, int y){
    p=new Point(x, y);
    edasi();
  }

  void edasi(){
    Ennistus.ringid.add(p);
  }

  public void redo(){
    super.redo();
    edasi();
  }

  public void undo(){
    super.undo();
    Ennistus.ringid.remove(p);
  }
}
```



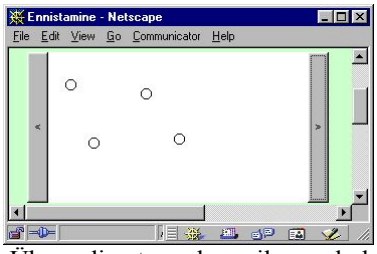
Tühi väli



Lisatud ringid



Kaks ringi tagasi võetud



Üks endine taas ekraanile pandud

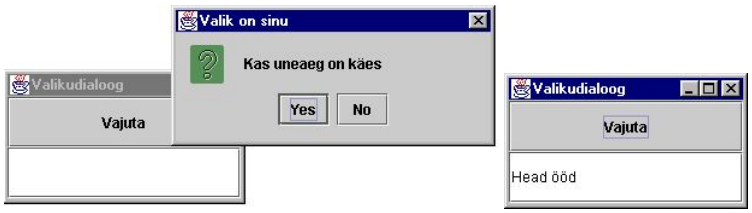
Dialoogiaknad

Enamike graafiliste programmeerimisvahendite juures on esimeseks näiteks lihtne teateaken. Java keeles ei kipu see lihtne käsk kohe silma alla jääma, kuid olemas on sellegipoolest. `Swing`i vahendite hulgas on `JOptionPane`, mille abil suhtlemise tarvis dialoogiaknaid luua annab. Kaks lihtsamat näidet kohe allpool. `System.exit(0)` on koodi viimaseks käsuks pandud, et virtuaalmasin oma töö rahus lõpetaks. Nii nagu muude graafika- ning muusikavahenditega, jääb ka teateakna avamisel programmi sees miski sisemine lõim töösse ning peaprogrammi lõppemisega ei lülitata virtuaalmasinat välja. Kui aga viimatimainitud käsklus lõppu panna, siis suleb see masina ning programm lõpetab rahulikult oma töö. Number 0 meetodi parameetrina näitab, et kõik lõppes õnnelikult ning mingeid lahendamata probleeme ei jäänud.

```
import javax.swing.*;
public class SwingiTeateaknad{
  public static void main(String[] argumendid){
     JOptionPane.showMessageDialog(new JFrame(), "Tervitus");
     JOptionPane.showMessageDialog(new JFrame(), "Tervitus", "Sõbralik teade",
        JOptionPane.PLAIN_MESSAGE);
     System.exit(0);
  }
}
```

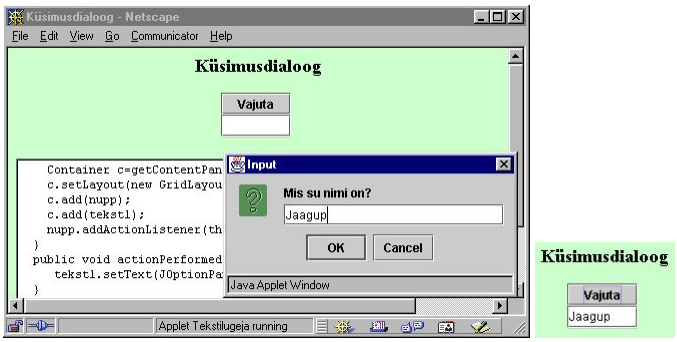
Kui soovida pakutud teatele kinnitust või ümber lükkamist, siis aitab selleks käsklus showConfirmDialog. Edasi tuleb lihtsalt käituda vastavalt kinni püütud vastusele.

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class SwingiDialoog3 extends JApplet implements ActionListener{
  JButton nupp=new JButton("Vajuta");
  JTextField tekst1=new JTextField();
  public SwingiDialoog3(){
    Container c=getContentPane();
    c.setLayout(new GridLayout(2, 1));
    c.add(nupp);
    c.add(tekst1);
    nupp.addActionListener(this);
  }
  public void actionPerformed(ActionEvent e){
     if(JOptionPane.showConfirmDialog(
         this, "Kas uneaeg on käes", "Valik on sinu",
         JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE
        )==JOptionPane.OK_OPTION){
       tekst1.setText("Head ööd");
     } else {
       tekst1.setText(" *** ");
     }
  }
  public static void main(String[] argumendid){
    JFrame f=new JFrame("Valikudialoog");
    f.getContentPane().add(new SwingiDialoog3());
    f.setSize(200, 200);
    f.setVisible(true);
  }
}
```



Teate sisestamist lubava akna loomiseks on käsklus JOptionPane.showInputDialog. Programmi töös jäädakse rahumeeli kasutaja sisestust ootama ning pärast nupulevajutust liigutakse taas rahumeeli edasi.

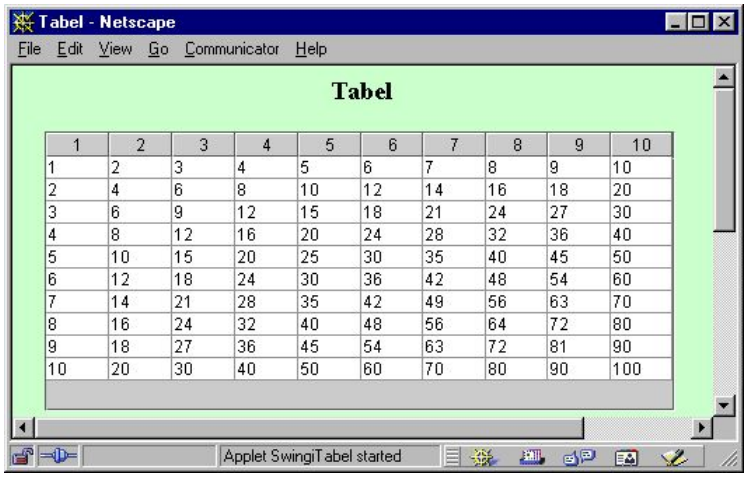
```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class SwingiDialoog4 extends JApplet implements ActionListener{
  JButton nupp=new JButton("Vajuta");
  JTextField tekst1=new JTextField();
  public SwingiDialoog4(){
    Container c=getContentPane();
    c.setLayout(new GridLayout(2, 1));
    c.add(nupp);
    c.add(tekst1);
    nupp.addActionListener(this);
  }
  public void actionPerformed(ActionEvent e){
     tekst1.setText(JOptionPane.showInputDialog("Mis su nimi on?"));
  }
}
```



Tabel

Andmete tabelina esitamiseks on Swingi paketti eraldi komponent loodud. Lihtsamal juhul tuleb luua näidatavatest elementidest kahemõõtmeline Object tüüpi massiiv, massiivi põhjal JTable tüüpi komponent ning paluda saadud andmed ekraanile näidata. JTable loomise konstruktoris JTable tabel=new JTable(andmed, andmed[0]); soovitakse ette saada kaks parameetrit: kõigepealt kahemõõtmeline massiiv lehel asuvate andmete kohta ning teise parameetrina ühemõõtmeline massiiv tulpade nimedega. Siin näites on antud tulpade nimedeks suure massiivi esimene rida ning nagu jooniselt näha, on sealt saadud numbrid ka ilusti tulpade pealkirjadeks vastu võetud.

```
import java.awt.*;
 import javax.swing.*;
 public class SwingiTabel extends JApplet {
   static JTable korrutustabel(){
   Object[][] andmed=new Object[10][10];
        for (int i=1;i<=10;i++)
        for (int j=1;j<=10;j++)
         andmed[i-1][j-1]=i*j+"";
        JTable tabel=new JTable(andmed, andmed[0]);
          //andmed ja pealkiri, milleks on ühega korrutamise rida.
        return tabel;
 }

 public void init(){
   getContentPane().add(new JScrollPane(korrutustabel()));
 }

 public static void main(String args[]) {
  JFrame f=new JFrame("Swingitabel");
  f.setSize(250,250);
  f.getContentPane().add(new JScrollPane(korrutustabel()));
  f.setVisible(true);
 }
}
```

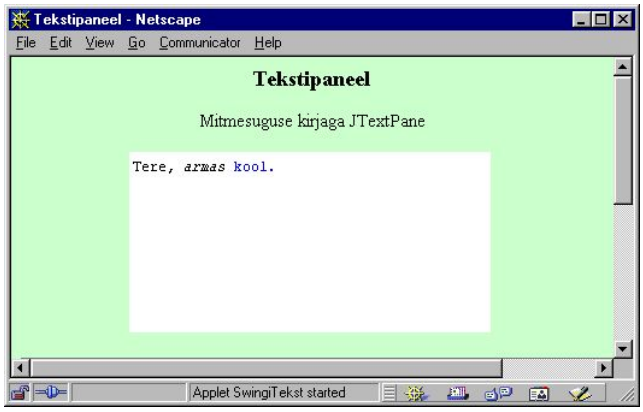


1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Kujundatava tekstiga paneel

Harilikul tekstiväljal tuleb kogu tekstile määrata ühesugune kuju ning värv. Aastaid on pidanud veebikaudsete suhtlussüsteemide kirjutajad välja mõtlema imenippe kasutajate eristamiseks ning muul puhul värvide ja kirjatüüpidega mängimiseks. JTextPane aga võimaldab igale lisatavale tekstilõigule määrata omapoolsed atribuudid ning kujundamine muutub märksa paindlikumaks. Kui kord on atribuutide kogum omistatud ühele objektile, siis võib seda kogumit edaspidi mitmes kohas tarvitada – kõikjal, kus on soovi samade parameetritega teksti järele.

```
import java.awt.*;
import javax.swing.*;
import javax.swing.text.*;
public class SwingiTekst extends JApplet{
  SimpleAttributeSet sinine=new SimpleAttributeSet();
  SimpleAttributeSet kursiiv=new SimpleAttributeSet();
  public SwingiTekst(){
   try{
    StyleConstants.setForeground(sinine, Color.blue);
    StyleConstants.setItalic(kursiiv, true);
    JTextPane tekstipaneel=new JTextPane();
    getContentPane().add(tekstipaneel, BorderLayout.CENTER);
    tekstipaneel.getDocument().insertString(0, "Tere, ", null);
    tekstipaneel.getDocument().insertString(tekstipaneel.getDocument().getLength(),
       "armas ", kursiiv);
    tekstipaneel.getDocument().insertString(tekstipaneel.getDocument().getLength(),
       "kool.", sinine);
   } catch (Exception e){
     e.printStackTrace();
   }
  }
  public static void main(String argumendid[]){
    JFrame f=new JFrame("Swingi tekst");
    f.setSize(200, 100);
    f.setLocation(100, 100);
    f.getContentPane().add(new SwingiTekst());
    f.setVisible(true);
  }
}
```



Veebiseilur

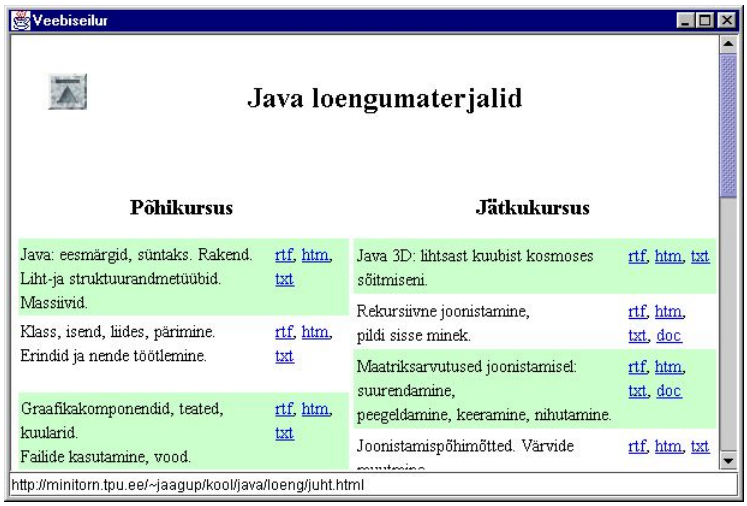
JEditorPane peal on võimalik edukalt näidata nii HTML (3.2) kui RTF-vormingus dokumente. Samuti tunneb komponent ära vajutused veebiviidetel ning nendele on võimalik kuular külge panna. Nõnda annab küllalt lihtsa vaevaga kokku panna lihtne veebiseilur, mille abil tavalised leheküljed täiesti vaadatud saab. Loodud paneelile piisab lihtsalt anda käsklus `setPage`, millele antakse ette vastav URL ning muu eest hoolitseb juba komponent ise. Programmi struktuur võib välja näha veidi harjumatu, sest suhteliselt lahkesti on kasutatud sisemisi klasse. Nii HyperlinkListener kui ActionListener on loodud otse meetodi sulgude sees, mis on aga täiesti lubatud tegevus. HyperlinkListeneri sees on lehekülje uuendamise käsud pandud Runnable liidest realiseeriva sisemise klassi run meetodi sisse ning tõmmatakse SwingUtilities.invokeLater käsu abil eraldi lõimes käima alles pärast seda, kui viitevajutuse peale tööle hakanud lõim on jõudnud oma töö lõpuni. Nii püütakse vältida programmi hangumist veebist andmete kohale tirimise ajaks. Eelnevalt nähtav dokument võetakse igaks

juhuks hoiule, et kui lehe avamisega peaks probleeme tekkima, siis pannakse vana sisu tagasi ning kasutajale näib, nagu poleks midagi kahtlast juhtunud.

Hüperlinkide puhul reageeritakse vaid sündmusetüübile `HyperlinkEvent.EventType.ACTIVATED`, ehk kui kasutaja on hiirega viitele vajutanud ning kavatseb viimase sisu vaatama hakata. Kui soovida staatusreal hiire alla jäävate viidete aadresse näidata, nagu ametlikes seilurites tavaks, siis tuleks reageerida ka sündmustele `HyperlinkEvent.EventType.ENTERED` ning `HyperlinkEvent.EventType.EXITED`.

Veateate näitamiseks on loodud eraldi meetod. Sellisel juhul on kergem veateate kuju soovi korral muuta. Praegugi kutsutakse vastavat alamprogrammi mitmest kohast välja.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.net.*;
import java.io.*;
public class Seilur extends JPanel {
    public Seilur() {
     setLayout (new BorderLayout ());
     final JEditorPane jt = new JEditorPane();
     final JTextField input =
      new JTextField("http://www.tpu.ee");
     jt.setEditable(false);
     // reageeri viidetele
     jt.addHyperlinkListener(new HyperlinkListener () {
       public void hyperlinkUpdate(final HyperlinkEvent e) {
           if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATED) {
             SwingUtilities.invokeLater(new Runnable() {
              public void run() {
               Document doc = jt.getDocument();
               try {
                URL url = e.getURL();
                jt.setPage(url);
                input.setText (url.toString());
               } catch (IOException io) {
                 //vahetus ebaõnnestus
                 veateade();
                 jt.setDocument (doc);
               }
              }
             });
           }
        }
     });
     JScrollPane pane = new JScrollPane();
     pane.setBorder (
      BorderFactory.createLoweredBevelBorder());
     pane.getViewport().add(jt);
     add(pane, BorderLayout.CENTER);

     input.addActionListener (new ActionListener() {
       public void actionPerformed (ActionEvent e) {
        try {
         jt.setPage (input.getText());
        } catch (IOException ex) {
         veateade();
        }
       }
     });
     add (input, BorderLayout.SOUTH);
     try{jt.setPage(input.getText());}
     catch(IOException e){veateade();}
   }

   void veateade(){
    JOptionPane.showMessageDialog (
    Seilur.this, "Vigane URL",
    "Tõenäoliselt vigane URL",
    JOptionPane.ERROR_MESSAGE);
   }

   public static void main(String argumendid[]){
    JFrame f=new JFrame("Veebiseilur");
    f.getContentPane().add(new Seilur());
    f.setSize(300, 300);
    f.setVisible(true);
   }
}
```



Kokkuvõte

Swingi paketis on hulk klasse, mis peaksid aitama kasutajal programmiga meeldivalt suhelda. Samuti on püütud nende loomisel silmas pidada erivahendite kasutajaid (ekraanilt lugejad, lihtsustatud klaviatuurid). Action-liidese abil saab samastada menüüelemendi ning tööriistariba nupu. Tabelisse paigutada aitab JTable ning teksti näidata ja redigeerida saab JEditorPane ning mitme muu klassi abil.

Ülesandeid

Swingi tabel

- Loo nulle ja x-e sisaldav 3x3 JTable.
- Luba kasutajal andmeid muuta ning loe kokku, mitu risti on märgitud.
- Lase kasutajal andmed valida valikmenüüst.

Swingi puu

- Loo puu, mille juure nimeks on "1" ning lehtedeks "2" ja "3".
- Loo puu, mille lehtedeks on arvud ühest tuhandeni. Oksteks algarvud nii, et mööda okstest koosnevat teed pidi korrutades jõutaks lehe väärtuseni. Mitme teguri korral panna väiksem juure poole.
- Lisaks eelmisele peab saama määrata sisestatavate arvude vahemikku. Arvu küsimise peale avatakse puu tee soovitud leheni.

Progressiriba

- Liiguta progressiriba väärtust algusest lõpuni.
- Ring liigub ühest servast teise. Ühes sellega kasvab progressiriba väärtus.
- Ring liigub ekraanil kümme korda ühest servast teise ja tagasi. Ühe progressiriba väärtus kasvab kogu ulatuses paremale liikumise ajal, teise väärtus vasakule liikumise ajal ning kolmas kasvab tasapisi, jõudes maksimumi liikumise lõpuks.

Slaider

- Loo ekraanile Swingi slaider.
- Lisa slaiderile suured kriipsud 5 ning väikesed 3 ühiku järel. Lisa tekstiväli. Slaideri väärtust näidatakse tekstiväljas. Slaideri all on skaala.
- Lisaks eelmisele muudetakse tekstiväljas oleva numbri muutmisel slaideri väärtust. Skaala väärtuse osa on tumedam (ClientProperty JSlider.isFilled), skaalaks on sõnad "külm", "leige", "soe", "tuline".

Valikupaneel

- Loo valikupaneel Eesti suuremate linnade nimedega.
- Lisa igale linnale ikoon ning vihjeks (ToolTip) ligikaudne rahvaarv. Iga paneeli sisuks kirjuta suure kirjaga seda linna läbiva jõe nimi. Pane linnad automaatselt iga viie sekundi tagant vahetuma.
- Iga linna puhul saab kasutaja anda viiepallisüsteemis hinnangu vaatamisväärsuste, toitlustuse ning teedevõrgu kohta. Kasutaja saab soovi korral linnu lisada. Valikute vastused salvestatakse faili.

Tekstiredaktor valikupaneelil

- Loo kahe valikuga valikupaneel, kus kummalgi paneelil paikneb tekstiala.
- Lisa kummalegi paneelile nupud failist lugemise ning sinna salvestamise kohta. Esimene paneel on seotud failiga katse1.txt ning teine failiga katse2.txt.
- Kasutaja saab avada ja salvestada soovitud tekstifaile. Kõik avatud failid on näha valikupaneeli paneelidena.

Menüüd

- Loo raam menüüga "Tervitused" elementidega "Sünnipäev" ning "Kooli lõpp"
- Lisa alammenüü, valitav menüüelement ((J)CheckboxMenuItem) ning toimiv menüükäsk programmi töö lõpetamiseks.
- Lisa menüü "numbrid" alammenüüdega "0".."9", igaühes asuvad vastava kümne numbrid. Numbrile vajutamisel kirjutatakse vastav arv tekstivälja. Loo hüpikmenüü (PopupMenu) kolmega jaguvate menüüelementide lubamiseks/keelamiseks ning algarvuliste menüüelementide peitmiseks/näitamiseks.

Sisemised raamid.

- Loo aken kolme sisemise raamiga.
- Kasuta sisemisi raame pildifailide näitamiseks. Kasutaja valitud pildifail avatakse uues raamis. Raami suuruse muutudes muutub ka pilt. Sulgemisnupule vajutades raam kaob.
- Lisaks eelmisele saab avatud pilti redigeerida ning jpeg-failina salvestada.

Swingi komponendid

- Loo JFrame.
- Paiguta selle ülaserva pildiga nupp.
- Nupule vajutades valitakse JColorChooseri abil nupu värv.
- Raami vasakusse serva paigutatakse puu (JTree). Puu elementideks on arvud ühest kümneni
- Igal esimese taseme elemendil on samuti alamelemendid ühest kümneni.
- Valitud elemendi väärtus kirjutatakse lehe allservas olevasse tekstivälja.
- Paremal servas olevale nupule vajutades avatakse JFileChooser, kust valitud pildifail joonistatakse raami keskele paigutatud paneelile.

Kolmemõõtmeline graafika

Java3D, koordinaateljestik, kujundid, vaateplatvorm, liikumine

Kolmes mõõtmes kujundite ekraanile paigutamiseks, liigutamiseks ning keeramiseks on Java keelde loodud abivahendite komplekt nimega Java 3D. See tuleb lisaks kompilaatorile/interpretaatorile masinasse installeerida ning seejärel saab temasse kuuluvaid vahendeid kasutada. Kui soovida, et seilur suudaks oma ekraanil selle paketi abil loodud rakendeid näidata, siis tuleb 3D ka brauseri Java virtuaalmasina juurde installeerida. Pea kõike siinse paketi abil loodut õnnestub ka tavavahendite ning keskkooli matemaatika abil lahendada, sest sisuliselt on ju ikka tegemist mingil hetkel ekraanipunktide värvimise ning hiire- ja klaviatuurisündmustele reageerimisega, kuid siin õnnestub veidi arvutamisvaeva kokku hoida ning samuti aitab operatsioonisüsteemi kaudu graafikakaardi/kiirendiga suhtlemine pildi arvutamist ja liigutamist sujuvamaks muuta.

Nagu iga uus asi, on seegi API arenemisjärgus (2001), kuid juba on siinsete vahenditega täiesti võimalik midagi ette võtta. Programmeerija tarvis lisatakse mõniteist paketti hulga klasside ning meetoditega. "Valmis" pakettideks on javax.media.j3d üldisemate klassidega kolmemõõtmeliste kujundite koostamiseks, liigutamiseks ja valgustamiseks. javax.vecmath aitab hoida ja töödelda kolme mõõtme kohta käivaid andmeid. Abipaketid algavad com.sun.j3d-ga ning sealt leiab valmis kujundeid ning muid realisatsioone. Nende pakettide nimed võivad tulevikus tõenäolisemalt muutuda, kuid võimalused ja põhimõtted peaksid ka edaspidi samasuguseks jääma.

Kuna Java3D kasutab operatsioonisüsteemist sõltuvaid vahendeid, siis kuulub kolmemõõtmeline lõuend Canvas3D AWT (mitte Swingi) komponentide hulka. Selle lõuendi sisse saabki oma kolmemõõtmelist programmi ehitama hakata. Lõuendi sees paiknevad objektid tuleb asetada puukujulisse hierarhiasse. Puu juureks on BranchGroup. Sinna külge saab lisada nii kujundeid kui edaspidi ka sõlmi kujundite nihutamiseks ja keeramiseks. SimpleUniverse loob kesta, mille abil ühendab kujundipuu 3D lõuendiga ning hoolitseb kasutaja asukoha eest. Järgnevas näites ilmub ekraanile värviline kuup (õigemini üks külg temast).

```
import java.applet.Applet;
import java.awt.*;
import com.sun.j3d.utils.geometry.ColorCube;
import com.sun.j3d.utils.universe.SimpleUniverse;
import javax.media.j3d.*;

public class Kuup1 extends Applet {
   public Kuup1() {
      setLayout(new BorderLayout());
      Canvas3D c = new Canvas3D(SimpleUniverse.getPreferredConfiguration());
      add(c, BorderLayout.CENTER);

      BranchGroup juur = new BranchGroup();
      juur.addChild(new ColorCube(0.5));
      juur.compile();

      SimpleUniverse u = new SimpleUniverse(c);
      u.getViewingPlatform().setNominalViewingTransform();
      u.addBranchGraph(juur);
   }

   public static void main(String[] args) {
      Frame f=new Frame("Kuup");
      f.add(new Kuup1());
      f.setSize(300, 300);
      f.setVisible(true);
   }
}
```

Nüüd veidi lähem seletus.

```
Canvas3D c = new Canvas3D(SimpleUniverse.getPreferredConfiguration());
```

loob 3D lõuendi vaikeparameetritega, et saaks järgnevalt loodud universumi panna oma andmeid sellele lõuendile saatma.

```
setLayout(new BorderLayout());
add(c, BorderLayout.CENTER);
```

määravad paigutushalduriks BorderLayout'i mille tulemusena õnnestub loodud lõuend paigutada rakendi (graafikakomponendi) keskele ning kuna sellesse rakendisse muid komponente paigutatud pole, siis venitatakse kolmemõõtmelise graafikakomponendi tarvis loodud lõuend üle kogu rakendile eraldatud pinna laiali.

```
BranchGroup juur = new BranchGroup();
```

loob hierarhia alguse, mille külge saan edasi kujundeid paigutada. Isendile annan nimeks juur, sest tegemist olekski nagu puu juurega.

```
juur.addChild(new ColorCube(0.5));
```

loob värvilise kuubi küljepikkusega 0,5 suhtelist pikkusühikut ning paigutab selle juure järglaseks (leheks).

```
juur.compile();
```

optimeerib loodud süsteemi. Ühe lihtsa komponendi puhul ei pruugi olulist kiirusevõitu olla, kuid vähegi rohkemate või keerulisemate kujundite puhul luuakse selle käsu tulemusena seosed, mille tulemusena saab pilti märgatavalt kiiremini ekraanile manada ning liigutada või valguse peegeldusi arvutada.

```
   SimpleUniverse u = new SimpleUniverse(c);
```

tulemusena luuakse ruum (universum), mis oma sees paikneva vaataja asukohast paistva pildi saadab eelnevalt loodud lõuendile nimega c.

```
   u.getViewingPlatform().setNominalViewingTransform();
```

nihutab vaataja veidi nullpunktist eemale, nii et keskele (vaikimisi asukoht) paigutatud oleksid nähtavad.

```
   u.addBranchGraph(juur);
```

määrab kujunditepuu, millist loodud ruumis näitama hakatakse.

Edasised käsud on juba traditsioonilised, et loodud süsteem ka silmale nähtavaks teha. Kui rakend käivitada veebilehel, siis võib main-meetodi kõige täiega ära jätta. Sel juhul tuleks aga hoolitseda, et 3D oleks ka brauseri JRE juurde installeeritud (Netscape puhul Windows 95 all näiteks kataloogi C:\Program Files\Netscape\Communicator\Program\java).

Kuubi keeramine

Keeramise korral tuleb juure ning kuubi vahele paigutada keerav sõlm. Transform3D suudab nii nihutada kui keerata.

```
Transform3D keerd1=new Transform3D();
keerd1.rotX(Math.PI/4);
TransformGroup keere1=new TransformGroup(keerd1);
```

tulemusena loodakse sõlm (tüübist TransformGroup), mis panduna juure ja kuubi vahele keerab viimast nii palju kui sõlmes ette nähtud.

```
  BranchGroup juur = new BranchGroup();
  juur.addChild(keere1);
  keere1.addChild(new ColorCube(0.5));
  juur.compile();
```

Nagu analoogia põhjal oletada võib, on klassis Transform3D lisaks rotX-le kasutada ka meetodid rotY ning rotZ keeramiseks, lisaks transform() nihutamiseks ning matemaatikahuvilistele veel hulga vahendeid asukoha määramiseks. Kogu arvutamine käib – nagu pea mujalgi kolmemõõtmelises graafikas – maatriksite ja vektorite korrutamise abil.

Nihutamine

Keeramine ja nihutamine käivad sarnaselt. Mõlemal juhul tuleb muutuse andmed esialgu paigutada Transform3D tüüpi objekti ning seejärel sealtkaudu TransformGroup'ile üle kanda. Nihutamiseks on Transform3D-l käsk setTranslation, mis soovib parameetriks Vector3f-i, ehk andmeid nihke kohta iga telje suhtes.

```
BranchGroup juur = new BranchGroup();
Transform3D t3d1=new Transform3D();
t3d1.setTranslation(new Vector3f(0.5f, 0, 0));
TransformGroup tg1=new TransformGroup(t3d1);
juur.addChild(tg1);
tg1.addChild(new ColorCube(0.5));
juur.compile();
```

X-telg viib paremale, y üles ning z kasutaja poole. Täht f numbri taga tähendab, et tegemist on ühekordse täpsusega reaalarvuga (float).

Kui soovida nii nihutada kui keerata, siis tuleb need tegevused sooritada üksteise järel. Luuakse kaks TransformGroup'i ning pannakse nad üksteise järel hierarhiasse, nii et juur -> tg1(nihe) -> keere1 -> kuup. Selliselt järgnevate muunete omadused liituvad.

```
Transform3D t3d1=new Transform3D();
t3d1.setTranslation(new Vector3f(0.5f, 0, 0));
```

```
TransformGroup tg1=new TransformGroup(t3d1);

Transform3D keerd1=new Transform3D();
keerd1.rotX(Math.PI/4);
TransformGroup keere1=new TransformGroup(keerd1);

juur.addChild(tg1);
tg1.addChild(keere1);
keere1.addChild(new ColorCube(0.5));
juur.compile();
```

Sama tulemuse võib ka lühemalt saavutada, arvutades nii keeramise kui nihke summa välja ühe Transform3D objekti sees ning tulemus määrata ühe TransformGroup'i omaduseks.

```
Transform3D t3d1=new Transform3D();
t3d1.setTranslation(new Vector3f(0.5f, 0, 0));
Transform3D keerd1=new Transform3D();
keerd1.rotX(Math.PI/4);
t3d1.mul(keerd1);

TransformGroup tg1=new TransformGroup(t3d1);
juur.addChild(tg1);
tg1.addChild(new ColorCube(0.5));
juur.compile();
```

Käsk mul (multiple) korrutab kahe maatriksi väärtused (ühendab nendes paiknevad omadused) ning paneb tulemuse esimesse algse väärtuse asemel. Seda Transform3D'd saab nüüd kasutada kui algsete omaduste summat.

Liigutamine

Järgneva näite tulemusena pannakse kuup ekraanil keerlema ning lähemale-kaugemale nihkuma.

```
import java.awt.*;
import javax.swing.*;
import com.sun.j3d.utils.geometry.ColorCube;
import com.sun.j3d.utils.universe.SimpleUniverse;
import javax.media.j3d.*;
import javax.vecmath.*;

public class Kuup2a extends JApplet implements Runnable{
  TransformGroup keere1;
  double nurk;

  public Kuup2a() {
        Canvas3D c = new Canvas3D(SimpleUniverse.getPreferredConfiguration());
        getContentPane().add(c, BorderLayout.CENTER);

        BranchGroup juur = new BranchGroup();
        Transform3D keerd1=new Transform3D();
        keere1=new TransformGroup(keerd1);
        keere1.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
        juur.addChild(keere1);
        keere1.addChild(new ColorCube(0.5));
        juur.compile();

        SimpleUniverse u = new SimpleUniverse(c);
        u.getViewingPlatform().setNominalViewingTransform();
        u.addBranchGraph(juur);
        new Thread(this).start();
  }
  public void run(){
    while(true){
      try{Thread.sleep(50);}catch(Exception e){}
      nurk+=0.1;
      double kaugus=-1+Math.sin(nurk/3);
      Transform3D t=new Transform3D();
      t.rotX(nurk);
      t.setTranslation(new Vector3d(0, 0, kaugus));
      keere1.setTransform(t);
    }
  }

    public static void main(String[] args) {
```

```
      Frame f=new Frame("Keerlev kuup");
      f.add(new Kuup2a());
      f.setSize(300, 300);
      f.setVisible(true);
    }
}
```

Võrreldes eelmise näitega on objektipuu loomisel juures üks käsk
`keere1.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);`
mille tulemusena lubatakse muundusgrupi väärtusi ka programmi töö käigus muuta. Muutmine ise on toodud eraldi lõime, meetodisse run, kus igal sammul suurendatakse nurka 0.1 radiaani võrra ning vastavalt sellele määratakse kuubi keere ning nihe. Kui soovida, et kuup kaugust vaatajast ei muuda, siis võib kaugusega seotud käsud välja jätta. Samuti nagu siin run-käsus tsüklina kuupi keerates võib ka mujal mitmesugustele andmetele vastavalt. Nii võib rahumeeli paluda kujundeid paigutada vastavalt kasutaja sisestatud või võrgust saabunud andmetele.

Interpolaatorid

Liikumisi tuleb kolmemõõtmelises graafikas küllalt palju ette. Nende sujuvamaks loomiseks on kasutaja vabastatud pidevast lõimede kirjutamisest ning ta võib koostada interpolaatoreid, mis hoolitsevad muundusgrupi kasutajale sobiva oleku eest. Interpolaatori "südameks" on Alpha, mis vastavalt kasutaja poolt etteantud parameetritele muudab enese sees arvu väärtusega nulli ja ühe vahel. Interpolaator vaatab iga natukese aja tagant seda väärtust ning vastavalt sellele seab muundusgrupi parameetrid. RotationInterpolatori puhul vastab Alpha väärtusele 0-null ning 1-täisring, muud suurused vahepeal. Esimene parameeter tähendab korduste arvu (-1 vastab igavesele pöörlemisele), teine aega millisekundites, mille jooksul suureneb väärtus nullist üheni. Siinses näites siis pööreldakse igavesti, kulutades täisringile kümme sekundit. Kuna algseisu ning täisringi puhul on kuup samas asendis, siis paistabki liikumine ühtlane, kuigi vahepeal toimub muundegrupi väärtuses täisringine nihe, kui Alpha hüppab väärtuselt 1 järsku 0 peale tagasi. Interpolaator on siin näites pandud objektihierarhiasse ühes kuubiga lehena muundegrupi järglaseks. Viimane omakorda kuulub juure alla.

```
BranchGroup juur = new BranchGroup();
TransformGroup keere1=new TransformGroup();
keere1.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
RotationInterpolator keeraja1=
      new RotationInterpolator(new Alpha(-1, 10000), keere1);
keeraja1.setSchedulingBounds(new BoundingSphere());
juur.addChild(keere1);
keere1.addChild(keeraja1);
keere1.addChild(new ColorCube(0.5));
juur.compile();
```

Interpolaatoreid on mitmesuguseid. PositionInterpolator suudab kujundeid (või nende komplekti) liigutada kahe punkti vahel. Millisel ajahetkel ese kus paigas asub, seda määrab jällegi isend tüübist Alpha. Temagi muutumisi saab täpsemalt määrata. Parameetriteta konstruktori puhul kasvab väärtus sekundi jooksul nullist üheni ning siis asub jälle otsast peale. Põhjalikuma määramise puhul – nagu järgnevast näitest näha – on Alpha parameetrite järjekord selline:

1. korduste arv (-1=lõpmatus)
2. lubatud suunad(kasvamine/kahanemine)
3. ooteaeg enne käivitumist (kõik ajad millisekundites)
4. ooteaeg iga ringi algul, kasvamiskiiruse suurenemise aeg
5. üheni jõudmise aeg
6. väärtusel 1 püsimise aeg
7. vähenemiskiiruse suurenemise aeg
8. nullini jõudmise aeg
9. nullil püsimise aeg.

PositionInterpolatori konstroktoris määratakse ära Alpha, muundusgrupp muutmiseks, liikumissuund ning pikkused palju algasendist edasi ning tagasi liikuda. Liikumissuuna võib Transform3D abil keerata täpselt selliseks nagu vaja.

```
BranchGroup juur = new BranchGroup();
TransformGroup tg1=new TransformGroup();
tg1.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
Transform3D suund=new Transform3D();
suund.rotZ(Math.PI/2); //keerab ümber z-telje
```

```
Alpha a=new Alpha(-1, Alpha.INCREASING_ENABLE | Alpha.DECREASING_ENABLE ,
   0, 2000, 2000, 2000, 0, 2000, 2000, 0);
PositionInterpolator liigutaja1=
     new PositionInterpolator(a, tg1, suund, -0.6f, 0.5f);
liigutaja1.setSchedulingBounds(new BoundingSphere());
juur.addChild(tg1);
tg1.addChild(liigutaja1);
tg1.addChild(new ColorCube(0.2));
juur.compile();
```

Sarnaselt töötab ScaleInterpolator, mille abil kujundi suurust muuta võib.

```
BranchGroup juur = new BranchGroup();
TransformGroup tg1=new TransformGroup();
tg1.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
ScaleInterpolator ip1= new ScaleInterpolator(new Alpha(), tg1);
ip1.setSchedulingBounds(new BoundingSphere());
juur.addChild(tg1);
tg1.addChild(ip1);
tg1.addChild(new ColorCube(0.2));
juur.compile();
```

Liikumistrajektoori võib punktide kaupa ette anda. Selleks tuleb kirjeldada punktide asukohad ning ajad, millal mingis punktis peab olema. Aegade väärtused tuleb määrata järjest, nullist üheni (ka Alpha annab sama vahemiku) ning aegu peab olema sama palju kui punkte. Siin näites algab liikumine koordinaatide alguspunktist. Kui Alpha on jõudnud 0,7-ni selleks ajaks asub kuup punktis (1, 0, 0) ehk on liikunud ühe ühiku võrra paremale. Järgneva 0,3 Alpha-ühiku jooksul nihutatakse kuup diagonaalis vasakule üles, nii, et ajal kui Alpha=1, on kuup punktis (0, 1, 0). Siis jälle tuldud teed tagasi. Liikumistee võib ette arvutada hulga pikema, nii et saab koostatud kujundi päris keerukat rada pidi käima panna. Kui keeramisrealt (suund.rotZ (Math.PI/2)) kommentaarimärgid eest ära võtta, siis liigub kuup kõigepealt üles ning sealt vasakule alla, sest siis on kogu liikumistee ümber Z-telje täisnurga jagu vasakule pööratud.

```
TransformGroup tg1=new TransformGroup();
tg1.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
Transform3D suund=new Transform3D();
//   suund.rotZ(Math.PI/2); //keerab liikumistrajektoori ümber z-telje
Alpha a=new Alpha(-1, Alpha.INCREASING_ENABLE | Alpha.DECREASING_ENABLE ,
     0, 2000, 2000, 2000, 0, 2000, 2000, 0);
float[] ajad={0, 0.7f, 1};
Point3f[] punktid={
  new Point3f(0, 0, 0),
  new Point3f(1, 0, 0),
  new Point3f(0, 1, 0)
};
PositionPathInterpolator liigutaja1=
       new PositionPathInterpolator(a, tg1, suund, ajad, punktid);
liigutaja1.setSchedulingBounds(new BoundingSphere());
juur.addChild(tg1);
tg1.addChild(liigutaja1);
tg1.addChild(new ColorCube(0.2));
```

Kaks kuupi

Esimesed näited olid koostatud ühe kuubiga vaid seepärast, et tulemus lihtsamini kätte tuleks ning kood lühem välja näeks. Täiesti samamoodi saab kujundeid ka juurde lisada. Kui need aga otse juure külge panna, siis satuvad nad otse üksteise peale/sisse ning mõni võib sootuks teise taha peitu jääda et teda sugugi näha ei ole. Kui aga vähemalt üks objekt eemale nihutada, siis on kummalgi oma koht ning neid võib rahumeeli ekraani peal vaadelda.

```
BranchGroup juur = new BranchGroup();
Transform3D tr1=new Transform3D();
tr1.rotX(Math.PI*2/3);
tr1.setTranslation(new Vector3d(0.5, 0, 0));
TransformGroup tg1=new TransformGroup(tr1);
juur.addChild(tg1);
tg1.addChild(new ColorCube(0.2));
juur.addChild(new ColorCube(0.1));
juur.compile();
```

Muud kujundid

Lisaks värvilisele kuubile, millega algul hea katseid teha, leiab geomeetriapaketist veel silindri (Cylinder), koonuse (Cone), risttahuka (Box) ning kera (Sphere). Neile pole veel värvi antud, seetõttu tuleb see töö ise ära teha.

```
Cylinder s1=new Cylinder(0.2f, 1.2f); //raadius, pikkus
s1.setAppearance(new Appearance());
juur.addChild(s1);
```

Nagu aimata võib, saab silindrile ette anda raadiuse ning kõrguse, koonusele samuti. Risttahukale kolme külje pikkused ning kerale raadiuse.

Taust

Soovides tagapinnaks oleva musta tausta millegi muu värvi vastu vahetada, tuleb lihtsalt kirjutada millist värvi selle asemele soovitakse. Ning nagu muud elemendid, tuleb seegi juure külge haakida.

```
   Background taust=new Background(new Color3f(Color.yellow));  //kollane taust
   taust.setApplicationBounds(new BoundingSphere(new Point3d(), 1000));
   juur.addChild(taust);
```

Vaatekoha muutmine

Et ruumis ringi liikuda või lihtsalt pilti teise nurga alt vaadata, selleks võib oma vaateplatsi asukohta muuta. Järgmise paari reaga luuakse uus Transform3D, määratakse sellele parameetrid (0,6 ühikut üles, 3 ettepoole) ning palutakse vaataja silmad ehk kaamera selle koha peale paigutada. Kui ise liikuda ülespoole, siis paistab pilt selle jagu altpoolt.

```
Transform3D t=new Transform3D();
t.setTranslation(new Vector3f(0,  0.6f, 3));
u.getViewingPlatform().getViewPlatformTransform().setTransform(t);
```

Järgnevalt veidi pikem näide, kus lennutatake taevasse hulk kuupe juhuslikesse asukohtadesse tähtedeks ning seejärel võib oma asukohta muutes kui kosmoselaevaga mööda taevalaotust ringi liikuda. Iga kuubi tarvis luuakse oma muundegrupp ning selle abil nihutatakse kuup keskpunkti suhtes juhuslikult kuni +/- 50 ühiku võrra igas suunas. KeyNavigatorBehavior võimaldab vaateplatvormilt küsitud TransformGroupi muuta ning selle abil meile ruumis liikumis ette kujutada.

```
import java.applet.Applet;
import java.awt.*;
import com.sun.j3d.utils.geometry.ColorCube;
import com.sun.j3d.utils.universe.SimpleUniverse;
import com.sun.j3d.utils.behaviors.keyboard.*;
import javax.media.j3d.*;
import javax.vecmath.*;

public class Asukoht2 extends Applet {
  public Asukoht2() {
    setLayout(new BorderLayout());
    Canvas3D c = new Canvas3D(SimpleUniverse.getPreferredConfiguration());
    add(c, BorderLayout.CENTER);

    BranchGroup juur = new BranchGroup();
   for(int i=1; i<1000; i++){              //luuakse hulk juhuslikke kuupe taevasse
       Transform3D t3=new Transform3D();
       t3.setTranslation(new Vector3f((float)(100*Math.random()-50),
                                      (float)(100*Math.random()-50),
                                      (float)(100*Math.random()-50)));
       TransformGroup tg=new TransformGroup(t3);
       tg.addChild(new ColorCube(0.4));
       juur.addChild(tg);
   }

    SimpleUniverse u = new SimpleUniverse(c);
   TransformGroup tg=u.getViewingPlatform().getViewPlatformTransform();
   KeyNavigatorBehavior kb=new KeyNavigatorBehavior(tg);
   kb.setSchedulingBounds(new BoundingSphere(new Point3d(), 1000));
   juur.addChild(kb);
```

```
   Transform3D t=new Transform3D();             //Määratakse vaatekoht
   t.setTranslation(new Vector3f(0,  0, -5));
   u.getViewingPlatform().getViewPlatformTransform().setTransform(t);

   Background taust=new Background(new Color3f(Color.blue));  //sinine taust
   taust.setApplicationBounds(new BoundingSphere(new Point3d(), 1000));
   juur.addChild(taust);
   juur.compile();
    u.addBranchGraph(juur);
  }

    public static void main(String[] args) {
      Frame f=new Frame("Kuubid taevas");
      f.add(new Asukoht2());
      f.setSize(300, 300);
      f.setVisible(true);
    }
}
```

Kiri

Lihtne kahemõõtmeline tekst õnnestub ekraanile manada küllalt kergesti.

```
   juur.addChild(new Text2D("Tervitus", new Color3f(Color.green),
     "Times", 55, Font.ITALIC));
```

Selle sama teksti pöörlema panek on aga natuke suurem ettevõtmine. Samas aga on tulemus küllaltki ilus ning mõnelgi juhul võib end ära tasuda. Endiselt tuleb lubada muundegrupi väärtust programmi töö ajal kirjutada, et saaksime teksti nurka muuta. Tagapoolsed ettevõtmised pinnaga hoolitsevad, et saaksime teksti mõlemat külge näha. Vaikimisi hoiab arvuti enese energiat kokku ning ei soostu kirja tagapoolt näitama.

```
  TransformGroup keere1=new TransformGroup();
  keere1.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
  RotationInterpolator keeraja1=
            new RotationInterpolator(new Alpha(-1, 10000), keere1);
  keeraja1.setSchedulingBounds(new BoundingSphere());
  juur.addChild(keere1);
  keere1.addChild(keeraja1);

  Text2D tekst=new Text2D("Tervitus", new Color3f(Color.green),
     "Times", 55, Font.ITALIC);
  PolygonAttributes pind=new PolygonAttributes();
  pind.setCullFace(PolygonAttributes.CULL_NONE);  //et ka teine pool näha oleks
  tekst.getAppearance().setPolygonAttributes(pind);
  keere1.addChild(tekst);
  juur.compile();
```

Ka kolmemõõtmelise kirja puhul tuleb hoolitseda, et seda ekraanil näitama soostutaks. Text3D iseenesest kirjeldab vaid kuju ehk geomeetria. Nägemiseks peab looma kujundi (Shape3D) ning seadma talle geomeetria ning materjali. Et materjal ruumis paistaks, selleks lubame seda valgustada (setLightingEnable) ning paigutame ruumi tausta/hajuvalguse allika. Nii ilmub õrn kolmemõõtmeline tekst ekraanile.

```
   Text3D t=new Text3D(
     new Font3D(new Font("Helvetica", Font.PLAIN, 1), new FontExtrusion()),
     "Tere" , new Point3f(-1, 0.6f, -1.5f)
   );
   Material m=new Material();   m.setLightingEnable(true);
   Appearance a=new Appearance();   a.setMaterial(m);
   Shape3D s=new Shape3D();
   s.setGeometry(t);   s.setAppearance(a);
   juur.addChild(s);
   Color3f valgusvarv=new Color3f(Color.white);
   AmbientLight al=new AmbientLight(valgusvarv);
   al.setInfluencingBounds(new BoundingSphere());
   juur.addChild(al);
   juur.compile();
```

Tasapind

Omale sobivatest tasapindadest saab kokku ehitada kõik meile sobivad kujundid. Ka kera pole siinse programmi jaoks muud kui piisavalt tihedasti üksteise kõrvale pandud kolmnurkade kogum. QuadArray abil saab määrata loodava pinna nurgad ning siis paluda nende vahele pind koostada. Edasine toiming sama kui kolmemõõtmelise teksti puhul: geomeetria ja materjal ühendada kujundiks ning selle nägemiseks panna ta lavagraafi(puusse) ja lasta valgus peale.

```
QuadArray nurgad=new QuadArray(4, GeometryArray.COORDINATES
  | GeometryArray.COLOR_3 | GeometryArray.NORMALS);
nurgad.setCoordinate(0, new Point3f(-0.5f, 0, 0));
nurgad.setCoordinate(1, new Point3f( 0.5f, 0, 0));
nurgad.setCoordinate(2, new Point3f( 0, 0.2f, 0));
nurgad.setCoordinate(3, new Point3f( 0, 0.4f, 0));
Shape3D kujund=new Shape3D();
kujund.setGeometry(nurgad);
Appearance a=new Appearance();
a.setMaterial(new Material());
kujund.setAppearance(a);
juur.addChild(kujund);
AmbientLight taustavalgus=new AmbientLight();
taustavalgus.setInfluencingBounds(new BoundingSphere());
juur.addChild(taustavalgus);

juur.compile();
```

Joonemassiiv

Nii pindadest kui joontest võib oma tarbeks kujundi luua. Järgnevas näites luuakse kahest joonest L-täht mida võib edaspidi kasutada nii nagu tavalist kujundit, nagu näiteks kuupi või koonust.

```
import javax.media.j3d.*;
import javax.vecmath.*;
public class SuurL extends Shape3D{
  Point3d p1=new Point3d(0, 0, 0),
          p2=new Point3d(0, 1, 0),
          p3=new Point3d(0.5, 0, 0);
  Point3d[] jooned={
    p1, p2,
    p1, p3
  };
  public SuurL(){
    LineArray joonemassiiv=new LineArray(4, LineArray.COORDINATES);
    joonemassiiv.setCoordinates(0, jooned);
    setGeometry(joonemassiiv);
    setAppearance(new Appearance());
  }
}
```

Selline klass tuleb salvestada eraldi faili ning edaspidi kui tarvist seda kasutada, tuleb hoolitseda, et see SuurL asuks samas kataloogis või sisseloetavas paketis ning loodud tähe ekraanile manamiseks piisab vaid käsust

```
juur.addChild(new SuurL());
```

Valgus

Vaid osa kujundeid suudame näha siis, kui neile valgus peale ei paista. Ka ülejäänuid on võimalik pealelangeva valgusega ilmestada. Java3D on loodud liikuvate kujundite tarbeks ning värvipeensuste arvutamiseks kuigi palju aega ei kulutata, sellegipoolest võib hea tahtmise korral täiesti äratuntava pildi ekraanile manada. Fotokvaliteediga ühe pildi algandmetest välja arvutamiseks kulub võimsatel arvutitel tunde ning siiski tuleb osa tegelikkuses aset leidvaid peegeldumisi ja muid seoseid tähelepanuta jätta.

Java3D abil saab esitada haju-, suund-, punkt- ning kohtvalgust. Esimene on nagu päevavalgus, see on hajunud ühtlaselt üle ruumi ning selle abil on võimalik ka laua all ning kapi taga esemeid näha. Suundvalguse kiired on paralleelsed, samuti nagu need mis päikeselt meieni jõuavad. Punktvalgus levib ühest punktist alates igas suunas laiali. Kohtvalguse puhul saab lisaks määrata ruuminurga, kui laia koonusena see valgus laiali läheb.

Valgustatavatel kehadel peavad olema välja arvutatud pinnanormaalid. Isegi pealtnäha kumerad pinnad on arvuti tarvis pisikeste kolmnurgakujuliste tasapindade ühendid, kus iga tasapinna jaoks vastavalt

valgustatusele tema värv välja näidatakse. Nagu muud 3D objektid, tuleb ka valgus lisada elementide puusse ning määrata tema mõjupiirkond (InfluencingBounds).

```
BranchGroup juur = new BranchGroup();
Appearance a=new Appearance();
a.setMaterial(new Material());
juur.addChild(new Sphere(0.5f, Sphere.GENERATE_NORMALS, a));
AmbientLight taustavalgus=new AmbientLight();
taustavalgus.setInfluencingBounds(new BoundingSphere());
juur.addChild(taustavalgus);
```

Suundvalguse lisamine on peaaegu analoogiline

```
BranchGroup juur = new BranchGroup();
Appearance a=new Appearance();
a.setMaterial(new Material());
juur.addChild(new Sphere(0.5f, Sphere.GENERATE_NORMALS, a));
DirectionalLight suundvalgus=new DirectionalLight();
suundvalgus.setInfluencingBounds(new BoundingSphere());
juur.addChild(suundvalgus);
juur.compile();
```

Punktvalguse puhul esimene punkt näitab valgusallika asukohta, teine valguse nõrgenemist. Valgustatus mingis punktis arvutatakse valemi järgi 1/(k1+kaugus*k2+kaugus*kaugus*k3), kus k1-k3 on teise Point3f-i parameetriteks antavad väärtused. Nii, et esimene vähendab heledust igal poole, teine võrdeliselt kaugusega ning kolmas võrdeliselt kauguse ruuduga.

```
PointLight punktvalgus=new PointLight(
   new Color3f(Color.green), new Point3f(-0.6f, -0.7f, 0), new Point3f(0, 0, 1)
 );
 punktvalgus.setInfluencingBounds(new BoundingSphere());
 juur.addChild(punktvalgus);
```

Kohtvalguse puhul antakse lisaks punktvalgusele ka suund (vektori abil) ja nurk, kui laialt valgus paistab. Viimane parameeter näitab, kui palju on sõõri keskus enam valgustatud kui ääred. Kui see väärtus on 0, siis paistab valgus ühtlaselt igale poole.

```
SpotLight kohtvalgus=new SpotLight(
new Color3f(Color.green), new Point3f(-0.5f, -0.7f, 0), new Point3f(0, 0, 1),
   new Vector3f(1, 1, 0), (float)Math.PI/8, 0.5f
);
kohtvalgus.setInfluencingBounds(new BoundingSphere());
juur.addChild(kohtvalgus);
```

Materjal

Kujundite pindade omadusi määratakse materjali abil. Kui materjal või muul moel väljanägemine puudub, siis pole ka midagi näha. Eralduva valguse abil määratakse värv, mis paistab ka siis, kus keha peale muud valgust ei tule. Selliselt on määratud näiteks värvilise kuubi küljed, millega eespool mitmel pool tutvusime. Koonus ja silinder vaikimisi ei helenda, nende nägemiseks tuleb sinna kas valgus peale suunata või neile endile värvus määrata. Allpool määratakse koonuse helenduv valgus roheliseks ning seetõttu näemegi seda ekraanil rohelisena.

```
BranchGroup juur = new BranchGroup();
   Cone k1=new Cone();
   Appearance valimus=new Appearance();
   Material materjal=new Material(
     new Color3f(Color.black),  //hajuvalgus
     new Color3f(Color.green),  //eralduv
     new Color3f(Color.green),    //peegelduv valgus
     new Color3f(Color.black),  1 //läige
   );
   valimus.setMaterial(materjal);
   k1.setAppearance(valimus);
   juur.addChild(k1);
   juur.compile();
```

Muster

Lisaks ühtlasele värvile või mitme valguse üheaegsest pealepaistmisest tingitud värviüleminekule võib keha pinnaks määrata mustri. Arvutuskiiruse suurendamiseks peavad mustriks määratud pildi laius ja kõrgus punktides olema arvu 2 astmed, ehk 2, 4, 8, 16 jne. Mõne kujundi (näiteks tasapind) puhul on lisaks vaikemäärangutele võimalik mustripilti ka mitmeti kujundi peale seada: keerata väänata, võtta sellest vaid osa jne.

```
BranchGroup juur = new BranchGroup();
Appearance a=new Appearance();
TextureLoader mustrilugeja=new TextureLoader("taust1.gif", this);
ImageComponent2D pilt=mustrilugeja.getImage();
Texture2D muster = new Texture2D(Texture.BASE_LEVEL, Texture.RGBA,
                                 pilt.getWidth(), pilt.getHeight());
muster.setImage(0, pilt);
a.setTexture(muster);
juur.addChild(new Sphere(0.5f, Primitive.GENERATE_TEXTURE_COORDS, a));
juur.compile();
```

Töö käigus kujundite lisamine

Pärast programmi algset käivitamist saab kujundeid lisada vaid koos uue oksa ehk BrahchGroupiga. Sedagi vaid juhul, kui algsele juurele on lisatud omadus ALLOW_CHILDREN_WRITE, ehk maakeeli öelduna tohib siis sinna järglasi juurde panna (või ära võtta). Nii ka siin all olevas näites nupule vajutamise puhul luuakse kõigepealt uus BranchGroup, selle külge lisatakse kujund (kuup). Siis oksa näitamine optimeeritakse (compile) ning alles seejärel lisatake oks olemasoleva juure külge. Kui kõik ilusti töötab, siis on nupu vajutamise peale näha, kuidas kuubi esikülg nupuvajutuse peale ekraanile juurde tekib.

Lisada võib ka suuremat kujundite gruppi. Siis tuleb see lihtsalt enne ekraanile paigaldamist valmis teha ning alles siis üheskoos sinna paigutada. Kui soovida lisada mujale kui keskkohta (mis on ju täiesti loomulik soov), siis peab oksa ja esemete vahele paigutama (vähemalt ühe) TransformGroup'i, mille abil siis kujund sobivasse asupaika nihutada ning keerata.

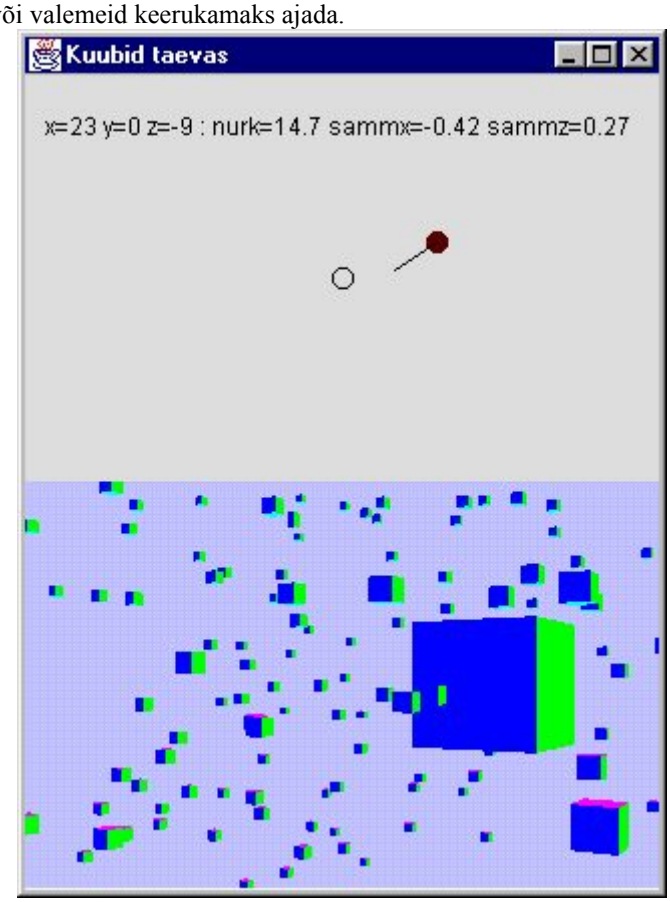
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.universe.SimpleUniverse;
import javax.media.j3d.*;
import javax.vecmath.*;

public class Kuubilisamine1a extends JApplet implements ActionListener{
  TransformGroup keere1;
  Button nupp=new Button("Lisa kuup");
        BranchGroup juur = new BranchGroup();

  public Kuubilisamine1a() {
   Canvas3D c = new Canvas3D(SimpleUniverse.getPreferredConfiguration());
   getContentPane().add(c, BorderLayout.CENTER);
   getContentPane().add(nupp, BorderLayout.NORTH);
   juur.setCapability(TransformGroup.ALLOW_CHILDREN_WRITE);
   juur.compile();

   SimpleUniverse u = new SimpleUniverse(c);
   u.getViewingPlatform().setNominalViewingTransform();
   u.addBranchGraph(juur);
   nupp.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e){
    BranchGroup oks1=new BranchGroup();
    oks1.insertChild(new ColorCube(0.3), 0);
    oks1.compile();
    juur.insertChild(oks1, 0);
  }

}
```

Ruumimängu põhi

Järgneval paaril leheküljel oleva näite järgi saab ehitada mitmeid kolmemõõtmelisi mänge – ja miks mitte ka asjalikke tutvustavaid programme, muutes väljanägemist, lisades kaunimaid ja keerukamaid kujundeid, määrates lubatud ja lubamatu liikumise piirkondi ning vajaduse korral andmeid võrgu kaudu teise kasutajani saates ja sealt teispoole teateid arvesse võttes. Samu sündmusi on näha kahel pinnal. Alumises saab kasutaja klahvide abil muuta oma asukohta ruumis, nähes, kuidas tähtedeks olevad kuubikesed tema silmade läbi paistavad. Üleval on näha tema asukohta määravad parameetrid, samuti tasandil kujutadud pinnal kasutaja asukoht ning vaatesuund universumis ning viimase keskkoht. See peaks aitama jälgida enese asukohta ning vältima eksimisi, mis kosmoses ilma vaatlusandmeteta ringi hõljudes on küllalt kerge tulema.

Peaklassi sisse on loodud kaks sisemist klassi, üks klahvivajutustele reageerimiseks ja liikumise eest hoolitsemiseks, teine ülemisel tasapinnalisel lõuendil kasutaja andmete näitamiseks. Kõikjal vajaminevad andmed kasutaja asukoha ja suuna kohta kirjeldati peaklassi alguses, nii on need kergesti kõikjale kätte saadavad ning vajaduse korral on ka näiteks võrgu kaudu andmete vahetamist lihtne korraldada.

Nooleklahvid "Üles" ja "Alla" liigutavad kasutajat edasi-tagasi vastavalt tema vaatesuunale. Teiste nooleklahvidega saab vaatesuunda vastavalt kas paremale või vasakule pöörata. Enese püstloodis kõrgemale või madalamale nihutamiseks aitavad leheküljeklahvid "Page up" ning "Page down", viltusuunas nihutamiseks lihtuse mõttes vahendeid loodud ei ole. Kuna aga asukohta saab kergesti muutujate väärtuste abil määrata, siis on sellise võimaluse lisamine täiesti võimalik. Praeguse lähenemise juures on kergemini võimalik piirduda kahemõõtmelises ruumis kehtivate matemaatiliste valemitega, muul juhul tuleb kas programmi pikemaks või valemeid keerukamaks ajada.




```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import com.sun.j3d.utils.geometry.ColorCube;
import com.sun.j3d.utils.universe.SimpleUniverse;
import com.sun.j3d.utils.behaviors.keyboard.*;
import javax.media.j3d.*;
import javax.vecmath.*;


public class Asukoht3a extends Applet {
  float x, y, z, uusx, uusy, uusz,
          nurk=0f, sammx, sammz, samm=-0.5f, nurgavahe=0.05f;
           //samm on negatiivne, kuna silmad on z-teljel miinuse poole.
```

```
Canvas3D c = new Canvas3D(SimpleUniverse.getPreferredConfiguration());
SimpleUniverse u = new SimpleUniverse(c);
TransformGroup tg=u.getViewingPlatform().getViewPlatformTransform();
Canvas tasapind=new Plaan();
Klahvikuular k=new Klahvikuular();

public Asukoht3a() {
      setLayout(new BorderLayout());
      setLayout(new GridLayout(2, 1));
      add(tasapind);
      add(c);


      BranchGroup juur = new BranchGroup();
 for(int i=1; i<1000; i++){
     Transform3D t3=new Transform3D();
     t3.setTranslation(new Vector3f((float)(60*Math.random()-30),
                                    (float)(60*Math.random()-30),
                                    (float)(60*Math.random()-30)));
     TransformGroup tg=new TransformGroup(t3);
     tg.addChild(new ColorCube(0.3));
     juur.addChild(tg);
 }
 juur.addChild(new ColorCube(2));  //suur kuup keskele


 c.addKeyListener(k);

 Background taust=new Background(new Color3f(new Color(200, 200, 255)));
 taust.setApplicationBounds(new BoundingSphere(new Point3d(), 1000));
 juur.addChild(taust);

 u.addBranchGraph(juur);
 tasapind.addFocusListener(new FocusListener(){
        public void focusGained(FocusEvent e){c.requestFocus();}
        public void focusLost(FocusEvent e){}
 }); //hoolitksetakse, et klahvidele reageeriv ruumilõuend saaks fookusesse.
}

double umarda(double arv, double koht){
    return Math.round(arv*Math.pow(10, koht))/Math.pow(10, koht);
}

public static void main(String[] args) {
    Frame f=new Frame("Kuubid taevas");
    f.add(new Asukoht3a());
    f.setSize(400, 600);
    f.setVisible(true);
}

class Klahvikuular implements KeyListener{
  public Klahvikuular(){
    arvutaSamm();
  }

  void arvutaSamm(){
    sammz=(float)(Math.cos(nurk)*samm);
    sammx=(float)(Math.sin(nurk)*samm);
  }
  public void keyPressed(KeyEvent e){
    int klahv=e.getKeyCode();
    if(klahv==KeyEvent.VK_RIGHT){  //keerab paremale
      nurk-=nurgavahe;
      arvutaSamm();
    }
    if(klahv==KeyEvent.VK_LEFT){
      nurk+=nurgavahe;
      arvutaSamm();
    }
    if(klahv==KeyEvent.VK_UP){  //edasi
      uusx+=sammx;
      uusz+=sammz;
    }
    if(klahv==KeyEvent.VK_DOWN){
      uusx-=sammx;
      uusz-=sammz;
    }
    if(klahv==KeyEvent.VK_PAGE_UP)uusy=y-samm;  //üles
    if(klahv==KeyEvent.VK_PAGE_DOWN)uusy=y+samm;
    prooviAstuda();
  }

  void prooviAstuda(){
```

```
      if(kasAsukohtSobib()){
        omista();
        liiguta();
        tasapind.repaint();
      } else {
        piiksu();
      }
    }

    /**
    * Meetodi abil määratakse, kas uude arvutatud asukohta tohib kasutaja liikuda.
    */
    boolean kasAsukohtSobib(){
       if(uusy<10) return true;
       else return false;
    }
    void omista(){
      x=uusx;
      y=uusy;
      z=uusz;
    }
    void liiguta(){
      Transform3D t1=new Transform3D();
      Transform3D t2=new Transform3D();
      t1.set(new Vector3d(x, y, z));
      t2.rotY(nurk);
      t1.mul(t2);
      tg.setTransform(t1);
    }
    void piiksu(){
      Toolkit.getDefaultToolkit().beep();
    }

    public void keyReleased(KeyEvent e){}
    public void keyTyped(KeyEvent e){}

  }


  class Plaan extends Canvas{
    public void paint(Graphics g){
      g.setColor(new Color(230, 230, 230));
      g.fillRect(0, 0, getWidth(), getHeight());
      int keskx=getWidth()/2;
      int kesky=getHeight()/2;
      int varv=100+(int)(10*y);  //vastavalt kõrgusele arvutatakse värv
      if(varv<10)varv=10;
      if(varv>240)varv=240;
      g.setColor(new Color(varv, 0, 0));
      g.fillOval(keskx+(int)(2*x)-5, kesky+(int)(2*z)-5, 10, 10);
      g.setColor(Color.black);
      g.drawString("x="+(int)x+" y="+(int)y+" z="+(int)z+" : nurk="+umarda(nurk, 2)+
         " sammx="+umarda(sammx, 2)+" sammz="+umarda(sammz, 2) , 10, 30);
           //koordinaate kirjeldav tekst
      g.drawLine(
        keskx+(int)(2*x), kesky+(int)(2*z),
        keskx+(int)(2*x+50*sammx), kesky+(int)(2*z+50*sammz)
      );
      g.drawOval(keskx-5, kesky-5, 10, 10);
    }
    public void update(Graphics g){paint(g);}
  }
}
```

Tehniline seletus

Import-käsklused on tuttavad eelsmistest näidetest ning täiesti tavalised. java.awt.event.* on siin tarvilik klahvivajutuste registreerimiseks. Canvas3D võimaldab neid kinni püüda ja vastu võtta nagu iga teinegi tavaline graafikakomponent.

```
float x, y, z, uusx, uusy, uusz,
          nurk=0f, sammx, sammz, samm=-0.5f, nurgavahe=0.05f;
```

Muutujad x, y ja z tähistavad kasutaja parasjagu kehtivat asukohta, nurk vaatenurga muutust võrreldes lähteasendiga (pilk sihitud kaugusse mööda miinusesse eemalduvat z-telge); uusx, uusy ja uusz tähistavad kasutaja uut väljaarvutatud asukohta, enne kui viimane sinna liikunud on. Niimoodi on kergesti võimalik kontrollida, kas uude kohta ta üldse liikuda tohib, samuti, kas enne uude kohta jõudmist tuleb mõni

eelnev operatsioon teostada (nt. uks avada või meloodialõik mängida). samm tähistab iga üksiku sammu pikkust ruumis. Negatiivne on ta seetõttu, et algasendis on silmad z-telje miinuspoolele suunatud ning edasi astudes paratamatult koordinaadi väärtus väheneb. Nurgavahe näitab, kui palju tuleb igal keeramisnupule vajutamisel vaatesuunda radiaanides muuta. Sammx ja sammz arvutatakse iga keeramise korral välja, et oleks teada, palju edasi või tagasi liikumise puhul vastavaid koordinaate muuta tuleb. Suurused on float tüüpi ehk ühekordse (mitte topelt) täpsusega reaalarvud, kuna Java3D on loodud enam liikumise tarvis ning siin arvestatakse enam sujuvust kui täpsust. float võtab arvutamisel vähem ressursse ning arvutamisest tekkivad erinevused pole siinse mõõtkava juures nagunii märgatavad.

```
TransformGroup tg=u.getViewingPlatform().getViewPlatformTransform();
Canvas tasapind=new Plaan();
Klahvikuular k=new Klahvikuular();
```

Kirjeldatud muutujate otstarve on järgmine:
tg tähistab kasutaja asukohta määravat TransformGroup'i. Selle omadusi muutes saab kasutajat nihutada ja keerata. Tasapinnal on näha andmed kasutaja asukohta ning klahvikuular hoolitseb selle määramise eest vastavalt vajutatud klahvidele.

```
for(int i=1; i<1000; i++){
       Transform3D t3=new Transform3D();
       t3.setTranslation(new Vector3f((float)(60*Math.random()-30),
                                      (float)(60*Math.random()-30),
                                      (float)(60*Math.random()-30)));
       TransformGroup tg=new TransformGroup(t3);
       tg.addChild(new ColorCube(0.3));
       juur.addChild(tg);
   }
   juur.addChild(new ColorCube(2));  //suur kuup keskele
```

Tähtedeks olevatele kuupidele arvutatakse igal korral uus koht. Nii x-, y- kui z-telje peal leitakse igale koordinaadi väärtusele vastama juhuslik arv –30 ning 30 vahel. Seda teeb `(60*Math.random()-30)` , mis loob juhuarvu vahemikust 0..1, korrutab selle 60-ga (vahemik 0..60) ning lahutab tulemusest 30. Sulgudes (float) võrrandi ees muudab tüübi ühekordse täpsusega reaalarvuks, et tulemus muude siin kasutatavate andmetega kokku käiks. Allpool lisatud suur kuup aitab suures tähemeres orienteeruda ning näitab ekslemisel kätte nullpunkti asukoha.

```
        tasapind.addFocusListener(new FocusListener(){
          public void focusGained(FocusEvent e){c.requestFocus();}
          public void focusLost(FocusEvent e){}
        }); //hoolitksetakse, et klahvidele reageeriv ruumilõuend saaks fookusesse.
```

Asukoha klahvidega liigutamiseks on vajalik, et vajutusi registreeriv komponent oleks fookuses. Siin näites võib aktiivseks osutuda nii ülemine kui alumine graafikakomponent. Esimesel juhul aga teated kuularini ei jõua ning tulemusena võib kasutaja nuppe muudkui vajutada ja vajutada, aga soovitud liikumistulemust ei ole. Lisatud kuulari puhul saadab tasapind koheselt omale fookuse vastuvõtmise puhul selle edasi alumisele kolmemõõtmelisele lõuendile, niimoodi ei lähe kasutaja teated kaotsi. Sarnase tulemuse võiks saada, kui paneksime ka ülemise tasapinna samale klahvikuularile teateid saatma. Viimasel on ükskõik kust teated tulevad, tema ülesandes on neile vastavalt reageerida. Samuti võib ülemisel lõuendil fookuse vastuvõtmise lihtsalt ära keelata, tulemus oleks ikka sama. Niimoodi sisemise klassina loodud fookusekuular suudab alumise lõuendi poole pöörduda kuna viimane on loodud isendimuutujana. Alamprogrammis loodud muutujate puhul on selline lähenemine raskem.

```
double umarda(double arv, double koht){
      return Math.round(arv*Math.pow(10, koht))/Math.pow(10, koht);
  }
```

on lihtsalt abifunktsioon, kus etteantud arv ümardatakse etteantud kohtadeni pärast koma. Ta pole sisuliselt kuidagi siinse programmiga seotud, kuid aitab andmete välja kirjutamisel koodi lühendada. Lahtiseletatult: Math.pow(10, koht) leiab arvu 10 sellise astme, mida koht näitab. Kui koht on 2, siis selle avaldise tulemuseks on 100. Edasi korrutatakse arv leitud väärtusega läbi ning ümardatakse. Kui arv oli enne 0,657, siis nüüd on tulemuseks 66. Et taas algsesse vahemikku tagasi jõuda, tuleb uuesti tulemus leitud kümne astmega jagada.

```
public Klahvikuular(){
  arvutaSamm();
}
```

```
void arvutaSamm(){
  sammz=(float)(Math.cos(nurk)*samm);
  sammx=(float)(Math.sin(nurk)*samm);
}
```

Nii klahvikuulari loomisel kui igakordsel vaataja suuna muutmisel palutakse tema nii x- kui z-koordinaadi suunas tehtava sammu pikkus uuesti välja arvutada. Siis saab edasi või tagasi liikumisel rahulikult neid väärtusi koordinaatidele lisada või eemadada, ilma, et peaks energiamahukat siinuse või koosiinuse arvutamist ette võtma. Nagu koolimatemaatikast teada, on koosinus sirge projektsioon ühe ning siinus teise koordinaattelje peal ehk täisnurkse kolmnurga peal vaadatuna lähis- ning vastaskülg.

```
    public void keyPressed(KeyEvent e){
      int klahv=e.getKeyCode();
      if(klahv==KeyEvent.VK_RIGHT){  //keerab paremale
        nurk-=nurgavahe;
        arvutaSamm();
      }
      if(klahv==KeyEvent.VK_UP){  //edasi
        uusx+=sammx;
        uusz+=sammz;
      }
```

Olles kasutajalt kinni püüdnud klahvivajutuse (meetod keyPressed käivitus), küsitakse sealt parameetrist edasise mugavama kasutuse huvides välja allavajutatud klahvi kood. Asutakse võrdlema, millist klahvi vajutati ning vastavalt sellele tegutsetakse edasi. Kui klahviks oli "Nool paremale", siis selle tulemusena vähendatakse nurga väärtust nurgavahe võrra ning arvutatakse välja uus samm liikumiseks nii x- kui z suunas. Paremale pööramisel tuleb lahutada seetõttu, et koordinaatteljestikul nurga suurenedes keeratakse vastupäeva, paremale poole saamiseks aga tuleb pöörata päripäeva.

```
if(klahv==KeyEvent.VK_PAGE_UP)uusy=y-samm;  //üles
```

Üles liikumisel jäävad x- ning z-koordinaat muutmata, vaid y muutub ning seda siis terve sammupikkuse ulatuses.

```
prooviAstuda();
    }

    void prooviAstuda(){
      if(kasAsukohtSobib()){
        omista();
        liiguta();
        tasapind.repaint();
      } else {
        piiksu();
      }
    }

boolean kasAsukohtSobib(){
       if(uusy<10) return true;
       else return false;
    }
```

Klahvivajutuskontrolli lõpuks kutsutakse välja meetod prooviAstuda(), kus otsustatakse, mida uute arvutatud koordinaatidega edasi tegema hakata. Tingimuses kasAsukohtSobib(), kontrollitakse, kas uutele arvutatud koordinaatidele tohib liikuda. Siin näites on tingimus lihtne, kasutajal ei lubata vaid tõusta kümne ja enama ühiku kõrgusele. Keerukamatel juhtudes saab aga siin hoolitseda selle eest, et kasutaja kõnniks ilusti mööda koridore, läbiks etteantud kontrollpunkti või ei jõuaks teisele liiklejale liiga lähedale.

Kui leiti, et uus asukoht sobib, siis omistatakse väljaarvutatud koordinaadid keha õigeteks koordinaatideks, liigutatakse ekraanil kasutaja sinna kohta ning samuti palutakse tasapind üle joonistada, et seal kasutaja kohta õiged andmed näha oleksid. Kui püüti liikuda keelatud kohale, siis selles näites tehakse sel puhul piiksu.

```
  void liiguta(){
      Transform3D t1=new Transform3D();
      Transform3D t2=new Transform3D();
      t1.set(new Vector3d(x, y, z));
      t2.rotY(nurk);
      t1.mul(t2);
      tg.setTransform(t1);
  }
```

Liigutamise tarvis arvutatakse nii nihke kui pööramise tarvis Transform3D. Nende omadused ühendatakse maatriksite korrutamise abil ning tulemus määratakse kasutaja asendiks.

```
class Plaan extends Canvas{
    public void paint(Graphics g){
      g.setColor(new Color(230, 230, 230));
      g.fillRect(0, 0, getWidth(), getHeight());
```

Üleval paiknev andmelõuend kaetakse joonistuskäsu väljakutsel kõigepealt etteantud hallika värviga kogu oma suuruses. All olev update-meetodist otse paint'i väljakutse tähendab, et ekraani vahepealset lisapuhastust ei toimu. Sama tulemuse võiks isegi veidi vähemate koodiridadega saavutada, kui määrata lõuendi taustavärviks kohe hallikas. Sel juhul update hoolitseb enne joonistuse algust vana pildi mustriga katmise eest.

```
      int keskx=getWidth()/2;
      int kesky=getHeight()/2;
```

Edaspidise joonistamise tarbeks küsitakse lõuendilt keskkoht. Sellisel juhul jääb ruumi keskpunkt lõuendi keskele ka juhul, kui kasutaja viimase suurust peaks muutma.

```
      int varv=100+(int)(10*y);  //vastavalt kõrgusele arvutatakse värv
      if(varv<10)varv=10;
      if(varv>240)varv=240;
      g.setColor(new Color(varv, 0, 0));
```

Kasutajat tähistava ringi värvus sõltub kasutaja asukoha kõrgusest. Mida kõrgemal see on, seda punasemalt joonistatakse kasutaja. Samas hoolitsetakse, et värvi määravate muutujate väärtused ei satuks äärmustesse, hoitakse, et punast värvi tähistav komponent oleks 10 ja 240 vahel. Kui 0 ja 255 vahelt väljuda, siis satutaks väärtusteni, mida pole võimalik ekraanil näidata ning väljastataks veateade.

```
      g.fillOval(keskx+(int)(2*x)-5, kesky+(int)(2*z)-5, 10, 10);
```

Joonistatakse, arvestades koordinaate lõuendi keskpunktist. Kuna ovaali joonistamisel tuleb määrata vasak serv, siin aga soovime ringi keskpunkti panna kasutaja asukohta tähistama, tuleb 10 punkti laiuse ovaali sobivasse paika loomiseks selle vasak serv viie punkti võrra vasemale nihutada.

```
      g.setColor(Color.black);
      g.drawString("x="+(int)x+" y="+(int)y+" z="+(int)z+" : nurk="+umarda(nurk, 2)+
         " sammx="+umarda(sammx, 2)+" sammz="+umarda(sammz, 2) , 10, 30);
           //koordinaate kirjeldav tekst
```

Kasutaja vaatesuunda näitav joon algab ta asukohast lõuendil ning lõpeb sinna suunas, kuhu kasutaja vaadata ja liikuda saab. Algots lõuendil sõltub niisiis lõuendi keskpunktist ja kasutaja asukohast, lõppotsa puhul aga tuleb juurde veel vaatenurgast sõltuvad väärtused. Meie õnneks on x- ning z-suunalised sammud juba välja arvutatud, piisab vaid need piisavalt suure teguriga korrutada, et need ekraanil mõistlikena välja paistaksid.

```
      g.drawLine(
        keskx+(int)(2*x), kesky+(int)(2*z),
        keskx+(int)(2*x+50*sammx), kesky+(int)(2*z+50*sammz)
      );
      g.drawOval(keskx-5, kesky-5, 10, 10);
    }
    public void update(Graphics g){paint(g);}
  }
```

Kokkuvõte

Java3D abil õnnestub luua ruumiline keskkond, kontrollida ja muuta seal nii esemete kui kasutaja asukohta ja asendit. Piltide ja sissetoodud objektide abil õnnestub sealne keskkond piisavalt tõeliseks, et võiks võiks võrreldavalt muude 3D vahenditega tunda end virtuaalkeskkonna osana. Kõik kasutatavad vahendid peavad olema ühtses andmepuus. Leiduvad vahendid nihutamiseks ja keeramiseks (TransformGroup), automaatseks asukoha ning omaduste muutmiseks (Interpolaatorid) ning sündmuste peale käitumiseks (Behavior).

Ülesandeid

3D kujundid

- Loo ekraanile mitmesse kohta mitmesuguseid kuupe
- Koosta õu laternapostide, istepinkide, põrkava palli ning liigutatava palliga. Luba kasutajal oma asukohta muuta. Katseta värve, mustreid ja interpolaatoreid.
- Anna palli koordinaadid ette tekstiväljast.

3D võrgumäng

- Mängijate asukohad määratakse serverist tulevate andmete järgi. Samaaegselt on seis näha nii kahe- kui kolmemõõtmelisel kujul.
- Platsil on sein. Serveris olevate arvutuste järgi hoolitsetakse, et seda ei saaks läbida.
- Kujunda eelnenu põhjal võimalikult tõelisena näiv võrgumäng.

Rekursiivne joonistamine

Kordused, rekursiooni baas, fraktal

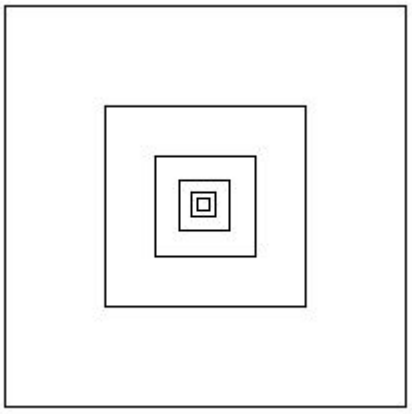
Plaan plaani peal

Ühe pargi keskel olnud plaan, kuhu see park ilusti üles joonistatud. Kõik puud ja teed olnud ilusti peal, samuti omal ka pargi plaan. Küll väiksemalt ja lihtsamalt, kuid siiski võis tähtsamaid teid ja ojasid täiesti ära tunda. Ning pisikese täpina oli sealgi näidatud plaani asukoht pargis. Nõndaviisi on see kui pildi sisse minek. Samasugust nähtust võib tähele panna, kui kaameramees filmib ning filmi peale jääb muu hulgas ka ekraan, kus parasjagu salvestatavat materjali näidatakse. Nii satub tekkinud pilt üha uuesti ja uuesti ringlema ning vaatajale tundub nagu ta saaks jälgida pikka koridori, kus avaneksid üha järgmised ja järgmised uksed ning kõigis neis oleks üks ja sama sisu. Iga korraga ainult üha väiksemalt ja väiksemalt, kuni meie silm või aparaadi eraldusvõime neid enam üksteisest eraldada ei suuda. Samasuguse tulemuse võib saada ka lihtsamate koduste vahenditega, kui üksteise vastu asendada kaks peeglit. Kumbki näitab teisele saabuvat pilti tagasi ning tekkiv koridor võib välgukiirusel väga pikaks kasvada. Olen näinud mitutteist peeglit üksteise seest paistmas ning see polnud kindlasti mitte veel võimaluste tipp. Paremate peeglite, suurema valgustuse ning põhjalikuma katsetamise juures saanuks tekkiva koridori pikkust veel hulga maad suurendada.

Kui pargis olnuks selliseid plaane neli, sel juhul oleks ka plaani peal selliseid plaane neli ning iga pisikese plaani peal neli täppi tähistamas kaartide asukohti. Ja mitut salvestatavat pilti näitavat ekraani filmides tunduks nagu eesolev koridor muudkui hargneks ja hargneks ning kaugustesse kasvaval rägastikul ei paistagi lõppu tulema.

Lõputu koridor

Mis on varem ekraanil, maastikul või paberi peal olemas, seda saab ka ise luua ning oma soovi kohaselt muuta. Kunstnikud ning teadlased on saanud niimoodi kauneid joone ja pinna vahepealseid kujundeid - fraktaleid. Arvuti võimaldab meil aidata korduvaid kujundeid uuesti ja uuesti välja joonistada ilma, et peaksime oma sõrmi tuhandete väikeste joonekeste tõmbamiseks kulutama. Ning ega käsitsi korralikust printerist selgemat tulemust ikkagi ei õnnestu saada. Ainult, et arvuti tarvis tuleb jooned kõigepealt välja arvutada, alles siis võime hakata mõtlema nende paberile kandmisele. Seetõttu tuleb hakkama saada mõnede matemaatiliste arvutustega, kuid juhul kui need tunduvad ületamatutena, võib valemid võtta juba töötavatest näidetest. Väärtusi suurendades ja vähendades ning käske lisades ja eemaldades peaks olema võimalik pea igasugused vähegi ettekujutatavad joonistused kokku kombineerida. Koridori moodustavate üksteise sisse tulevate ringide või ristkülikute joonistamise eeskiri oleks küllalt lihtne: tuleb neid senikaua üksteise otsa lükkida, kuni sisemised nii väikseks muutuvad, et sinna sisse pole enam mõistlik ega võimalik midagi paigutada.



```
import java.awt.*;
import java.applet.Applet;
public class Koridor extends Applet{
         public void paint(Graphics g){
  int x=100, y=100, laius=100, korgus=100;
  while(laius>5){
    g.drawRect(x, y, laius, korgus);
    laius=laius/2;
    korgus=korgus/2;
    x=x+laius/2;
    y=y+laius/2;
  }
 }
}
```

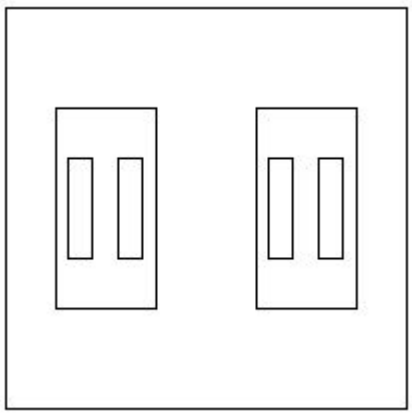
Niiviisi anti algul koridori ukse joonistamisel ette selle vasak ja ülemine serv arvatuna pildi nullpunktist. Igal järgmisel korral joonistatakse ukse sisse järgmine, mõõtmeid pidi poole väiksem uks (ehk praegusel juhul ristkülik. Vasakut serva nihutakse edasi veerandi esialgse laiuse võrra. Nii paistab, et järgmine jääb eelmise sisse keskele. Kui nihutaksime sisemist vähem, siis tunduks, nagu seisaksime suure pika koridori suhtes viltu.

Iga sammuga ühekaupa niimoodi kujundeid teise sisse või peale joonistada saab sellise tsükliga ilusti. Lihtsalt tuleb iga ringi alguses ette anda uued koordinaadid, mille järgi kujund välja joonistada. Iga sammuga mitme sisemise tüki joonistamine aga läheb praegusel viisil keerukaks. Siit aitab meid välja rekursiooniks nimetatud vahend, kus iseeneslikult kasvava kujundi loomine usaldatakse ühele alamprogrammile, kust siis seda sama vajaduse korral uuesti välja kutsutakse. Eelnev näide päistaks selle lähenemise valguses välja nii:

```
import java.applet.Applet;
import java.awt.*;
public class Koridor2 extends Applet{
          void joonistaKoridor(Graphics g, int x, int y, int laius, int korgus){
    g.drawRect(x, y, laius, korgus);
    if(laius>5)joonistaKoridor(g, x+laius/4, y+korgus/4, laius/2, korgus/2);
  }
          public void paint(Graphics g){
    joonistaKoridor(g, 100, 100, 100, 100);
  }
}
```

Nagu näha, muutus kood pigem lühemaks ning mis tähtsam - kergemini soovikohaselt muudetavaks. Loodud joonistamise käsku võib mitmelt poolt välja kutsuda ning kui on soovi omale hargnevad koridorid luua, siis tuleb vaid paar käsku ümber ja juurde teha.

```
import java.applet.Applet;
import java.awt.*;
public class Koridor3 extends Applet{
 void joonistaKoridor(Graphics g, int x, int y, int laius, int korgus){
  g.drawRect(x, y, laius, korgus);
  if(laius>15){
    joonistaKoridor(g, x+laius/8,   y+korgus/4, laius/4, korgus/2);
    joonistaKoridor(g, x+laius*5/8, y+korgus/4, laius/4, korgus/2);
  }
 }
 public void paint(Graphics g){
   joonistaKoridor(g, 100, 100, 200, 200);
 }
}
```

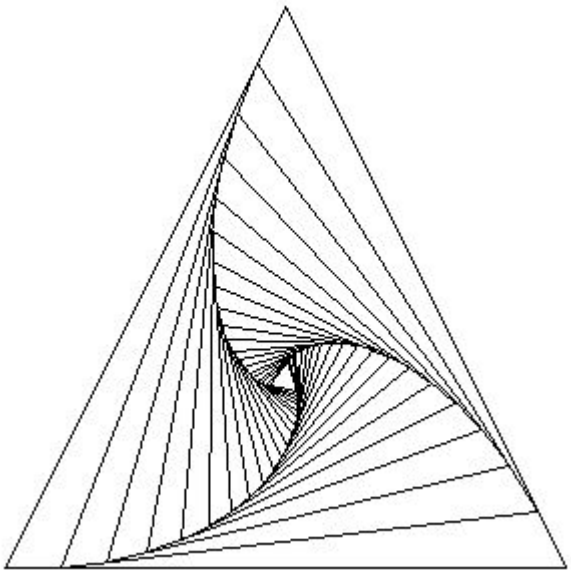


Teineteises peituvad hulknurgad

Kujundid ei pea üksteise sees mitte sama pidi olema. Küllalt lihne ning samas ilus on korduvalt veidi keeratuna hulknurki üksteise sisse joonistada. Hea ettekujutusvõime korral võib niimoodi jääda mulje kasvavast tornist või sügavusse pürgivast august. Kirjeldus:

Esimene kolmnurk joonistatakse nii, et tema kaugus rakendi servadest oleks 10 punkti. Järgmine kolmnurk joonistatakse eelmise sisse nõnda, et tema nurgapunktide leidmiseks liigutakse kümnendik mööda kolmnurga külge edasi. Matemaatiliselt leitakse uus asukoht võttes lähema punkti koordinaatidest üheksa kümnendikku ning liites sellele kaugema punkti ühe kümnendiku. Abimuutujatena kasutatakse siin jääki ja nihet, mis peavad oma väärtustes kokku andma ühe. Kõigepealt leitakse uued punktid ning seejärel omistatakse uute väärtused (ax1, ay1) joonistamisel kasutatavatele väärtustele(x1, y1). Sellline vaheetapp on vajalik, kuna algseid koordinaate läheb ka pärast uute leidmist vaja. Kõigepealt arvutatakse punkti 1 uus asukoht punktide 1 ning 2 vahelt. Kui aga pärast hakatakse kolmanda punkti uut asukohta arvutama punktide 1 ja 3 vahelt, siis peab punkti 1 vana asukoht teada olema, et leitud punkt algse joone peale satuks. Arvutatakse reaalarvudega, vaid joonistamisel teisendatakse täisarvulisteks ekraanipunktideks.

```
public class Kolmnurgad extends Applet{
  public void paint(Graphics g){
    Dimension suurus=getSize();
    double x1=10, y1=suurus.height-10,
           x2=suurus.width-10, y2=suurus.height-10,
           x3=suurus.width/2, y3=10;
    double ax1, ay1, ax2, ay2, ax3, ay3;
    int kordustearv=20;
    double nihe=0.1, jaak=1-nihe; //osa külje pikkusest
    for(int i=0; i<kordustearv; i++){
      g.drawLine((int)x1, (int)y1, (int)x2, (int)y2);
      g.drawLine((int)x2, (int)y2, (int)x3, (int)y3);
      g.drawLine((int)x3, (int)y3, (int)x1, (int)y1);

      try{Thread.sleep(100);}catch(Exception e){} //viivitus

      ax1=x1*jaak+x2*nihe;
      ay1=y1*jaak+y2*nihe;
      ax2=x2*jaak+x3*nihe;
      ay2=y2*jaak+y3*nihe;
      ax3=x3*jaak+x1*nihe;
      ay3=y3*jaak+y1*nihe;

      x1=ax1; y1=ay1;
      x2=ax2; y2=ay2;
      x3=ax3; y3=ay3;
    }
  }
}
```



Kolmnurgad üksteise seljas

Järgnevas näites asutakse kolmnurka kasvatama. Võrdhaarsele kolmnurgale antakse ette pikema külje kaks otspunkti. Nende abil arvutatakse kolmas nurk, mis paigutatakse pikema külje keskkohast küljega risti võetuna poole küljepikkuse kaugusele. Külgede kohale tõmmatakse jooned ning juhul, kui tekkinud lühem külg oli vähemalt 10 punkti pikk, võetakse see uue kolmnurga pikimaks küljeks ning arvutatakse selle järgi uued küljed. Kuna iga korraga lähevad tekkivad kolmnurgad väiksemaks, siis ühest hetkest alates on mõistlik joonistamine ära lõpetada. Kui loodava kolmnurga kõrguseks poleks mitte pool vaid kaks aluseks võetavat külge, siis tuleksid uued kolmnurgad järjest suuremad ning tuleks teiselt poolt leida ülempiir, millest suuremat kujundit pole enam mõistlik joonistada.

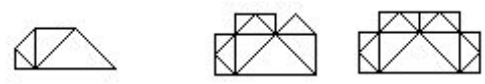
Joonistamise eest hoolitseb meetod joonistaPuu, millele antakse joonistuse sihtkoha graafiline kontekst ning joone otspunktide koordinaadid. Joonistamisega saab meetod hakkama sõltumata sellest, millises asukohas ning millise kaldega joon sinna ette antakse.

Matemaatilise poole seletus. Eelkirjeldatud kohas asuva kolmanda punkti asukoha leidmiseks tuleks kõigepealt leida pikema külje keskpunkt ning sealt edasi liikuda küljega risti poole külje pikkuse ulatuses. Üks võimalus külje keskkoha leidmiseks on arvutada nihe (vektor) ühest punktist teise (x=x2-x1; y=y2-y1), leida sellest pool ning lisada esimese punkti koordinaatidele juurde (x=x1+(x2-x1)/2; y=y1+(y2-y1)/2). Edasi

tuleks leitud punktile otsa liita punktidevahelise sirgega risti minev nihe. Nihkevektori keeramiseks saab kasutada seost (x'=x*cos(a)-y*sin(a); y'=x*sin(a)+y*cos(a)) ehk täisnurkse vastupäeva nihke korral cos(a)=0, sin(a)=1 ning (x'=-y; y'=x) ning poole punktidevahelise kauguse pikkusega ning punktidevahelise sirgega risti oleva nihke saab ((y2-y1)/2; (x2-x1)/2) ning sealtkaudu tulevad kokku ka x3 ja y3 arvutamise valemid. Kuna arvutil on suunatud y-telg alla, tavamatemaatikas aga üles, siis on märgid telje suuna muutmise eesmärgil vastupidiseks muudetud.

Ootamiskäsk Thread.sleep on vahele pandud lihtsalt seetõttu, et joonistamisel oleks näha, millises järjekorras kolmnurgad ekraanile tekivad ning millal üks joonistuspuu valmis saab ning teisega alustatakse. Kui see käsk välja kommenteerida, siis valmib pilt tunduvalt kiiremini.

```
import java.applet.Applet;
import java.awt.*;
public class Puu extends Applet{
          double kaugus(int x1, int y1, int x2, int y2){
    return Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
  }
          void joonistaPuu(Graphics g, int x1, int y1, int x2, int y2){
    int x3=x1+(x2-x1)/2+(y2-y1)/2;
    int y3=y1+(y2-y1)/2-(x2-x1)/2;
    g.drawLine(x1, y1, x2, y2);
    g.drawLine(x1, y1, x3, y3);
    g.drawLine(x3, y3, x2, y2);
    try{Thread.sleep(500);}catch(Exception e){}
    if(kaugus(x1, y1, x3, y3)>10){
      joonistaPuu(g, x1, y1, x3, y3);
      joonistaPuu(g, x3, y3, x2, y2);
    }
  }
          public void paint(Graphics g){
    joonistaPuu(g, 130, 290, 170, 290);
  }
}
```



Kuna korduva joonistamise puhul läheb punktidevaheliste nihkevektorite arvutamist ning keeramist sageli vaja, siis koostati selle tarbeks klass, mis matemaatilised arvutused enese sisse peidab ning programmeerijale jääb vaid hoolitseda sisulise poole eest. Väärtused saab sellele klassile anda vaid isendi loomisel (samuti nagu klassi java.lang.String puhul) ning iga muundamise puhul luuakse uus isend. Andmeid hoitakse ja arvutatakse täpsuse huvides reaalarvudena, kuid välja antakse joonistamise tarbeks täisarvudena. Meetodid on kahe nihke liitmiseks (pluss), teguriga korrutamiseks (korda), pikkuse arvutamiseks (pikkus) ning täisnurga jagu vastupäeva keeramiseks (keera). Kuna punkte tasandil võib andmete poolest samastada nullpunktist nendeni jõudvate vektoritega, siis sobivad klassi meetodid mõnel puhul ka punktidega ümber käimiseks.

```
public class Tasandinihe{
  /**
  * Nihke koordinaatide väärtused. Piiritleja final rea ees näitab, et
  * väärtusi pärast algset omistamist enam muuta ei saa.
  */
  final double x, y;
  public Tasandinihe(double ux, double uy){
    x=ux;
    y=uy;
  }

  /**
  * Nihe arvutatakse etteantud kahe punkti koordinaatide järgi
  */
  public Tasandinihe(double x1, double y1, double x2, double y2){
    x=x2-x1;
    y=y2-y1;
  }
  int X(){
    return (int)x;
  }
  int Y(){
    return (int)y;
  }
  double pikkus(){
    return Math.sqrt(x*x+y*y);
  }
```

```
  /**
  * Vahe samast punktist lähtuvate nihete otspunktide vahel.
  */
  double kaugus(Tasandinihe t1){
    return t1.miinus(this).pikkus();
  }

  /**
  * Väljastatakse nihke väärtuse ja teguri korrutis.
  * Nihe ise jääb muutmata.
  */
  Tasandinihe korda(double tegur){
    return new Tasandinihe(x*tegur, y*tegur);
  }

  /**
  * Väljastatakse käesoleva ning parameetrina antud nihke summa.
  * Mõlema nihke enese väärtus jääb muutmata.
  */
  Tasandinihe pluss(Tasandinihe t1){
    return new Tasandinihe(t1.x+x, t1.y+y);
  }

  Tasandinihe miinus(Tasandinihe t1){
    return this.pluss(t1.korda(-1));
  }

  /**
  * Suund keeratakse täisnurga jagu vastupäeva.
  */
  Tasandinihe keera(){
    return new Tasandinihe(-y, x);
  }
}
```

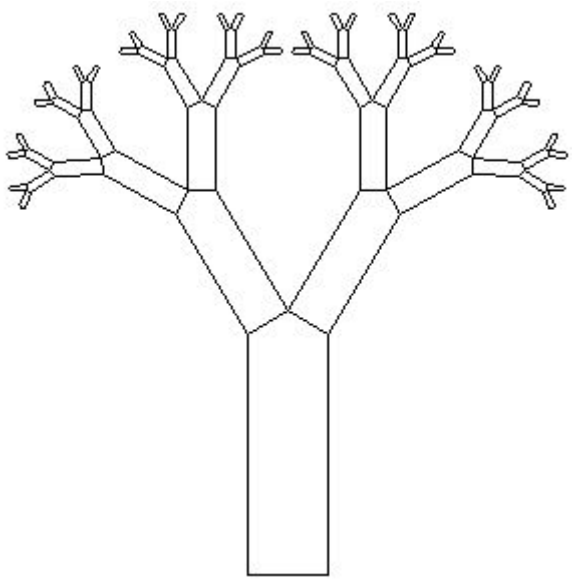
joonistamine näeks loodud Tasandinihkeklassi abil välja nii nagu allpool toodud programmis Puu2. Leitakse kaks nihet: üks ühest punktist teise ning teine nihe esimesega risti. Edaspidi saab neid kahte kasutada x ning y ühikutena uues loodud koordinaatteljestikus, kus x-teljeks oleks kahe etteantud punkti vaheline sirge.
Tasandinihe nx=p2.miinus(p1);
tähendab, et nihke x-ühiku leiame, kui lahutame teise punkti koordinaatidest esimese punkti koordinaadid. Eelmisega risti oleva y-ühiku saame aga x-i täisnurga jagu vastupäeva keerates.
Tasandinihe ny=nx.keera();
Edasi võib loodud lõike ühikutena kasutades leida joonistamise tarvis vajaliku(d) punkti(d). Niiviisi sõltubki arvutatava joonise asukoht ja suurus vaid etteantud punktidest.
Tasandinihe p3=p1.pluss(nx.korda(0.5).pluss(ny.korda(-0.3)));

```
import java.applet.Applet;
import java.awt.*;
public class Puu2 extends Applet{

  void joonistaPuu(Graphics g, Tasandinihe p1, Tasandinihe p2){
    Tasandinihe nx=p2.miinus(p1);
    Tasandinihe ny=nx.keera();
    Tasandinihe p3=p1.pluss(nx.korda(0.5).pluss(ny.korda(-0.3)));

    g.drawLine(p1.X(), p1.Y(), p2.X(), p2.Y());
    g.drawLine(p1.X(), p1.Y(), p3.X(), p3.Y());
    g.drawLine(p3.X(), p3.Y(), p2.X(), p2.Y());
    try{Thread.sleep(500);}catch(Exception e){}

    if(p1.kaugus(p3)>10){
      joonistaPuu(g, p1, p3);
      joonistaPuu(g, p3, p2);
    }

  }
  public void paint(Graphics g){
     joonistaPuu(g, new Tasandinihe(30, 290), new Tasandinihe(270, 290));
  }

}
```

Et loodav joonis enam puu moodi oleks, selleks peaks seal peale oksakohtade ka oksi endid olema. Nii tuleb kaks punkti juurde arvutada ning iga kujund luuakse juba viie punkti abil: algsed kaks oksa algul, järgmised kaks oksa lõpul ning veel üks, millest alates uut oksapaari juurde arvutada. Joone tõmbamist läheb päris palju vaja, selle tarvis sai uus alamprogramm loodud. Kui y suund kohe vastupidiseks keerata, siis võib

edaspidi harjumuspärast matemaatilist tava järgida, et see telg ikka üles suunatud on. Muidugi veel viisakam oleks suunda alles joonistamisel arvutada.

```
import java.applet.Applet;
import java.awt.*;
public class Puu3 extends Applet{

  void joonistaPuu(Graphics g, Tasandinihe p1, Tasandinihe p2){
    Tasandinihe nx=p2.miinus(p1);
    Tasandinihe ny=nx.keera().korda(-1); //y-suund vastupidiseks
    Tasandinihe p3=p1.pluss(ny.korda(3));
    Tasandinihe p4=p2.pluss(ny.korda(3));
    Tasandinihe p5=p1.pluss(nx.korda(0.5).pluss(ny.korda(3.3)));

    joon(g, p1, p2);
    joon(g, p1, p3);
    joon(g, p2, p4);
    joon(g, p3, p5);
    joon(g, p4, p5);
    try{Thread.sleep(500);}catch(Exception e){}

    if(p1.kaugus(p3)>10){
      joonistaPuu(g, p3, p5);
      joonistaPuu(g, p5, p4);
    }

  }

  void joon(Graphics g, Tasandinihe t1, Tasandinihe t2){
    g.drawLine(t1.X(), t1.Y(), t2.X(), t2.Y());
  }

  public void paint(Graphics g){
     joonistaPuu(g, new Tasandinihe(130, 290), new Tasandinihe(170, 290));
  }

}
```

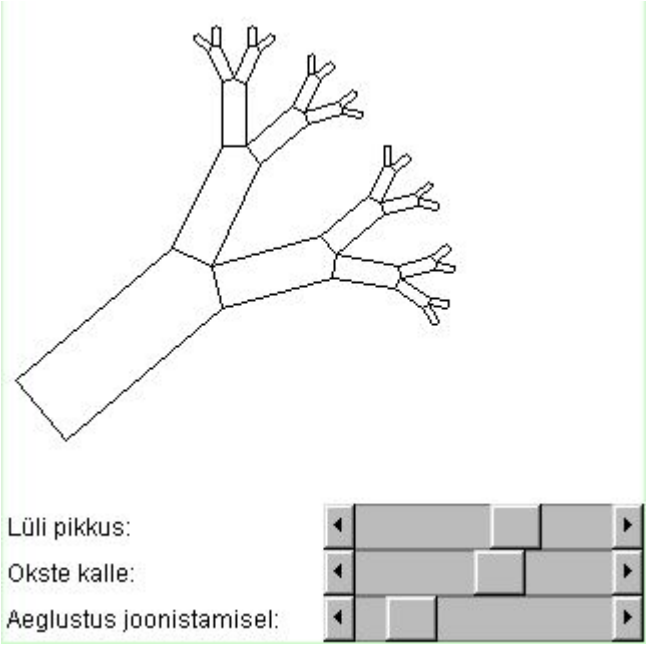


Kasutaja soovitud puu

Et kasutajale rohkem pildi kujundamise võimalusi anda, selleks võib algsed punktid pärida tema käest, samuti lasta muuta muid parameetreid nagu okste pikkus ja kalle ning joonistamise kiirus.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class Puu4 extends Applet implements MouseListener{
  int vajutusenr;
  Tasandinihe hiir1=new Tasandinihe(130, 300),
              hiir2=new Tasandinihe(170, 300);
  Scrollbar ooteaeg=new Scrollbar(Scrollbar.HORIZONTAL, 100, 100, 0, 500);
     //väärtus, nupupikkus, vähim, suurim
  Scrollbar pikkus=new Scrollbar(Scrollbar.HORIZONTAL, 200, 100, 0, 500);
  Scrollbar kalle=new Scrollbar(Scrollbar.HORIZONTAL, 200, 100, 0, 500);
```

```
  Label ooteajasilt=new Label("Aeglustus joonistamisel:");
  Label pikkusesilt=new Label("Lüli pikkus:");
  Label kaldesilt=new Label("Okste kalle:");
  public Puu4(){
    setLayout(new BorderLayout());
    Panel p1=new Panel(new GridLayout(3, 2));
    p1.add(pikkusesilt);
    p1.add(pikkus);
    p1.add(kaldesilt);
    p1.add(kalle);
    p1.add(ooteajasilt);
    p1.add(ooteaeg);
    add(p1, BorderLayout.SOUTH);
    addMouseListener(this);
  }

  public void mousePressed(MouseEvent e){
    vajutusenr++;
    if(vajutusenr==1){
      hiir1=new Tasandinihe(e.getX(), e.getY());
    }
    if(vajutusenr==2){
      hiir2=new Tasandinihe(e.getX(), e.getY());
      repaint();
      vajutusenr=0;
    }
  }
  public void mouseReleased(MouseEvent e){}
  public void mouseClicked(MouseEvent e){}
  public void mouseEntered(MouseEvent e){}
  public void mouseExited(MouseEvent e){}

  void joonistaPuu(Graphics g, Tasandinihe p1, Tasandinihe p2){
    Tasandinihe nx=p2.miinus(p1);
    Tasandinihe ny=nx.keera().korda(-1); //y-suund vastupidiseks
    Tasandinihe p3=p1.pluss(ny.korda(pikkus.getValue()/100.0));
    Tasandinihe p4=p2.pluss(ny.korda(pikkus.getValue()/100.0));
    Tasandinihe p5=p1.pluss(nx.korda(0.5).pluss(ny.korda(
      pikkus.getValue()/100.0+kalle.getValue()/1000.0
    )));

    joon(g, p1, p2);
    joon(g, p1, p3);
    joon(g, p2, p4);
    joon(g, p3, p5);
    joon(g, p4, p5);
    try{Thread.sleep(ooteaeg.getValue());}catch(Exception e){ }
    if(p1.kaugus(p3)>10){
      joonistaPuu(g, p3, p5);
      joonistaPuu(g, p5, p4);
    }

  }

  void joon(Graphics g, Tasandinihe t1, Tasandinihe t2){
    g.drawLine(t1.X(), t1.Y(), t2.X(), t2.Y());
  }

  public void paint(Graphics g){
     joonistaPuu(g, hiir1, hiir2);
  }

  public static void main(String argumendid[]){
    Frame f=new Frame("Puu joonistamine");
    f.add(new Puu4());
    f.setSize(300, 400);
    f.setVisible(true);
  }
}
```



Joonistamisel ei pruugi kõik lülid olla sugugi ühesugused. Nende ehitus võib sõltuda suurusest, kaugusest juurest kui ka näiteks juhuse läbi. Nii võib luua küllalt usutavaid pärisasjade analooge nagu lumehelves, riigipiir või kasvõi seesama puu. Ka kõverjoone tõmbamise algoritmid kasutavad sarnast lähenemist, sest tunduvalt odavam on meeles pidada kolme või nelja punkti asukohta kui joone iga punkti asukohta. Liiatigi erinevad ekraanide ja printerite joonistustihedused piisavalt, et ühe tarvis võib etteantud tihedusega punktide meeldejätmine tunduda raiskamisena, teisel puhul aga jääb tulemus silmnähtavalt konarlik. Rekursiivselt aga punkte ühendavaid sirgeid lühemateks lõikudeks jagades võib igal korral uute joonte loomise siis lõpetada, kui ollakse jõutud joonistustäpsuse tasemele. Sealt edasi arvutamine enam paremat tulemust ei saa anda.

Ka võib kasutajal lubada ette joonistada, milline peaks üks lüli välja nägema ning millistesse kohtadesse selle peal võiksid uued kinnitada. Nii oleks tulemuseks töövahend kunstniku tarvis. Kuid lihtsalt silmarõõmu võib sellistest joonistest küllaga saada, pakkudes vaatajale järelemõtlemist, millal ja kus võib jälle midagi kusagilt välja kasvama hakata.

Murdjoon

Üheks rekursiooni näiteks on murdjoone loomine. Olgu siis rakenduseesmärgiks lihtsalt kujundi servade kaunistamine või proovitagu läbi kapillaarsoonte võimalikku paiknemist nahaalusel pinnal. Kui pika joone sisse lisada jõnks ning edasi iga tekkinud joone sisse veel jõnks, siis ongi suudetud üheülbalise sirjoone asemel luua märgatavalt paindlikum kujund. Kavandatava algoritmina näeks joone jagamine väiksemateks osadeks välja järgnevalt.

```
public void murdJoon(Graphics g, int x1, int y1, int x2, int y2){
   if(kaugus(x1, y1, x2, y2)>pikimaJoonePikkus){
     //leiab joone keskkoha lähedale uue punkti ning selle
     //abil tõmbab kaks joont
   } else {
     g.drawLine(x1, y1, x2, y2);
   }
 }
```

Kuidas just uus punkt leitakse ning kuidas edasised jooned tehakse, selle tarvis on variante palju. Üks neist on toodud allpool. Joone keskoha võib leida otspunktide aritmeetilise keskmise abil.

```
int kx=(x1+x2)/2;
int ky=(y1+y2)/2;
```

Keskkoha lähedase punkti kaugus keskpunktist võiks sõltuda joone pikkusest. Et mida pikem joon, seda kaugemale võib nihke paigutada. Pika algse joone puhul pole väikest nihet kuigivõrd näha.

Lühikese algjoone puhul aga sama pikk nihe võib loodavad jooned algsest hoopis pikemaks muuta ning juhul kui joone murdmine läheb kordusesse, võib kogu lugu sootuks tsüklisse sattuda. Katsete tulemusena aga paistis, et kui loodav punkt tuleb algsest keskkohast mõlemat telge pidi mõlemas suunas kuni 0,2 algse joone pikkuse kaugusele, siis pole joone liigset väljavenitatust karta. Samas aga on muutus piisav, et silmaga sirgjoont ja murdjoont eristada.

```
int x3=kx+(int)((Math.random()*k*0.4)-k*0.2);
int y3=ky+(int)((Math.random()*k*0.4)-k*0.2);
```

Kui uus keskpunkt valmis arvutatud, siis tuleb hoolitseda, et algsetest otspunktidest loodud uue punktini joon saaks loodud. Olgu siis lihtsalt tõmmatuna või võetakse nüüd ette sama algoritm mis ennegi ja püütakse uue joone liiga suure pikkuse korral see omakorda osadeks jagada.

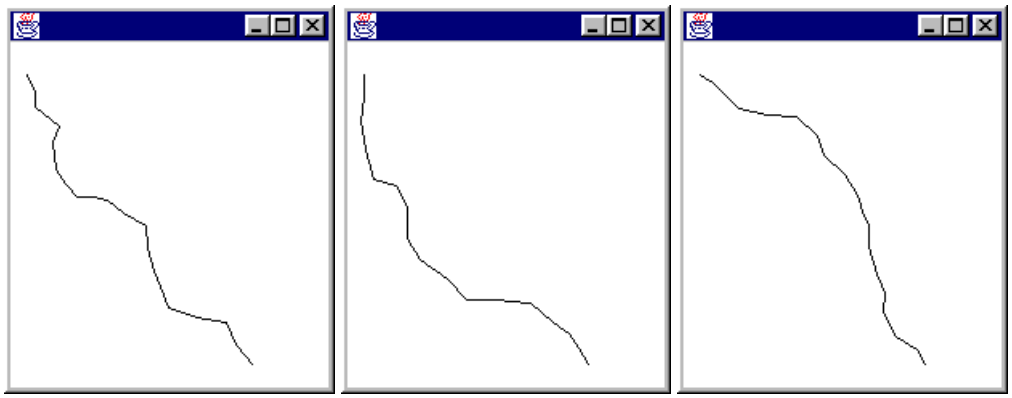
```
murdJoon(g, x1, y1, x3, y3);
murdJoon(g, x3, y3, x2, y2);
```

Eelneva algoritmi põhjal veidi pikem koodinäide, mille käivitamisel peaks ka tulemus näha olema. Kui paint'is öeldakse

```
murdJoon(g, 10, 20, 150, 200);
```

siis asutakse etteantud punktide vahele joont tõmbama. Ning iga kord, kui loodud joon tuleb pikem kui määratud pikima joone pikkus ehk 20 jagatakse joon uuesti kaheks jupiks. Lühema kahe punkti vahelise kauguse puhul veetakse joon lihtsalt ekraanile ning edasi lühemaks ei jagata.

```
import java.applet.Applet;
import java.awt.*;
public class Murdjoon2 extends Applet{

  int pikimaJoonePikkus=20;
  /**
  * Alamprogramm väljastab kahe punkti vahelise kauguse
  */
  public double kaugus(int x1, int y1, int x2, int y2){
    int dx=x2-x1;
    int dy=y2-y1;
    return Math.sqrt(dx*dx+dy*dy);
  }

  public void murdJoon(Graphics g, int x1, int y1, int x2, int y2){
    double k=kaugus(x1, y1, x2, y2);
    if(k>pikimaJoonePikkus){
      int kx=(x1+x2)/2;
      int ky=(y1+y2)/2;
      int x3=kx+(int)((Math.random()*k*0.4)-k*0.2);
      int y3=ky+(int)((Math.random()*k*0.4)-k*0.2);
      murdJoon(g, x1, y1, x3, y3);
      murdJoon(g, x3, y3, x2, y2);
    } else {
      g.drawLine(x1, y1, x2, y2);
    }
  }

  public void paint(Graphics g){
    murdJoon(g, 10, 20, 150, 200);
  }

  public static void main(String[] argumendid){
    Frame f=new Frame();
    f.add(new Murdjoon2());
    f.setSize(300, 300);
    f.setVisible(true);
  }
}
```

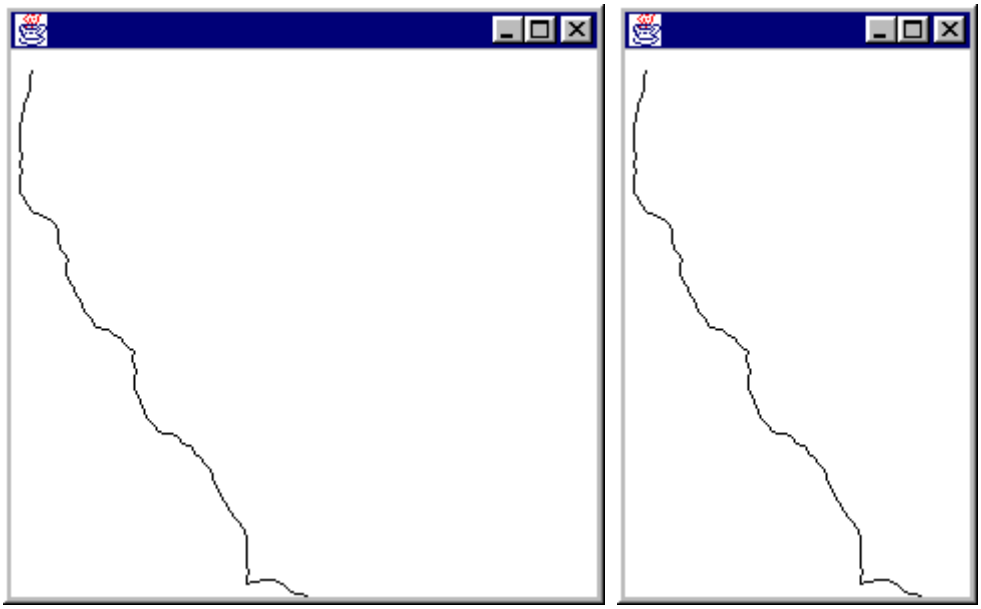


Igal joonistusel uus murdjoon

Eelnenud näites tuli igal joonistuskorral asuda joont uuesti välja arvutama. Tahtes aga masinat liigsest nuputamisest säästa ning mis tähtsamgi - loodud joont ikka ja jälle uuesti vaadata, tuleb mõeldud punktid meelde jätta. Et loodavate punktide hulk pole ette teada, siis on nende hoidmiseks massiivi asemel mugavam kasutada nimistut, näiteks standardpaketis kättesaadavat LinkedListi. Ning kuna siinses näites joone lühim pikkus ei muutu, siis võib kõik vahepunktid algul välja arvutada ning edasi vaid sobival hetkel pilt mälus paiknevate andmete põhjal välja joonistada. Punkti andmete hoidmiseks võib kasutada java.awt paketis asuvat klassi Point - vahendit mis juba olemas, ei pea hakkama enam uuesti looma. Ka punktide vahelise kauguse leidmiseks on juba käsklus olemas, Point-isendi käsklus distance teatab sobiva väärtuse. Tsükliga küsitakse nimistust ükshaaval välja punktipaarid. Kui punktide vaheline kaugus ületab lubatud pikima, siis leitakse keskkoha lähedale uue punkti koordinaadid nii nagu eelmiseski näites. Loodud p3 asetatakse endise p2 kohale ning LinkedList hoolitseb juba ise, et ülejäänud elemendid nimistus edasi liigutataks. Joonistamisel piisab punktipaaride näidatavate ekraanikoordinaatide vahele jooned tõmmata ning murdjoon ongi ekraanil.

```
import java.applet.Applet;
import java.awt.*;
import java.util.*;
public class Murdjoon4 extends Applet{

  LinkedList punktid=new LinkedList();
  int pikimaJoonePikkus=5;

  public Murdjoon4(){
    punktid.add(new Point(10, 10));
    punktid.add(new Point(200, 300));
    lisaVahePunktid();
  }

  public void lisaVahePunktid(){
   int koht=0;
   while(koht+1<punktid.size()){
    Point p1=(Point)punktid.get(koht);
    Point p2=(Point)punktid.get(koht+1);
    double kaugus=p1.distance(p2);
    if(kaugus>pikimaJoonePikkus){
      Point p3=new Point(
        (p1.x+p2.x)/2+(int)((Math.random()-0.5)*0.4*kaugus),
        (p1.y+p2.y)/2+(int)((Math.random()-0.5)*0.4*kaugus)
      );
      punktid.add(punktid.indexOf(p2), p3);
      if(p1.distance(p3)<=pikimaJoonePikkus){
        koht=koht+1;
      }
    } else {
      koht=koht+1;
    }
   }

  }



  public void paint(Graphics g){
    for(int i=0; i<punktid.size()-1; i++){
      Point p1=(Point)punktid.get(i);
      Point p2=(Point)punktid.get(i+1);
```

```
      g.drawLine(p1.x, p1.y, p2.x, p2.y);
    }
  }

  public static void main(String[] argumendid){
    Frame f=new Frame();
    f.add(new Murdjoon4());
    f.setSize(300, 300);
    f.setVisible(true);
  }
}
```



Joon püsib ka akna suuruse muutmisel

Virtuaalne Eestimaa

Järgnevalt näide, mille alusel peaks õnnestuma ehitada isearenevaid maailmu nii mängude kui õpisimulatsioonide tarbeks. Nii nagu astronoomid väidavad, et mida pole võimalik märgata, seda ei pruugi ka olemas olla, kehtib sarnane järeldus seda enam ka arvutimaailma kohta. Sugugi ei pruugi kõiki erijuhte kohe programmi töö algul välja mõelda. Kui täpsustused suudetakse vajalikul hetkel tehe piisavalt kärmesti, pole kasutajal kuigivõrd võimalusi otsustamiseks, et mudelil kaugvaate korral miskit viga oleks. Ning võimalus igas soovitud detailis pisiasjadeni välja minna jätab vaatajale uskumuse, et kõik ongi lõpuni viimistletud. Kui kontrollid tulevad majapidamist üle vaatama ning kõikjal kuhu vaadata valitseb kord ja puhtus, ei saa neil jorisemiseks põhjust olla. Olgugi, et võibolla lihtsalt keegi jälgib kontrollide teekonda ning hoolitseb, et iga võimalik vaadeldav punkt õnnestuks selleks ajaks korda teha, kui kontroll oma suurima kiiruse abil saaks sinna jõuda.

Või kõrvutuseks veel näide muinasjutuvestjast. Hea vestja suudab väljamõeldud maailma kõikide kohtade ja erijuhtude kohta vastuseid anda, ehkki ta ei pruugi olla algul kõike lõpuni välja mõelnud. Kui ta suudab hoolitseda, et loodavad paigad, ühendusteed ja sündmused eelnevatega vastuollu ei lähe, siis võib jutustaja kasvõi koos kuulajatega uusi lugusid ja kohti välja mõelda. Ikka on huvitav kuulata, kaasa mõelda ja meenutada.

Võrreldes eelmise näitega ei saa praegusel juhul kõiki punkte rakenduse töö algul välja mõelda, vaid tuleb kohti vastavalt kasutaja liikumisele juurde mõelda. Kui kilomeetritepikkune murdjoon kohe sentimeetripikkuste lõikude kaupa välja arvutada, siis kuluks mälu kõvasti ning joonistamine muutub lootusetult aeglaseks. Juba ainuüksi ühe kilomeetri peale tuleks sada tuhat punkti, pikema maa peale seda enam.

Väljamõeldud võlumaailma aluseks on Eestimaa väga ligikaudne rannajoon - nii umbes Euroopa ilmakaardilt vaadatuna. Ning kes siinsest kandist rohkem ei tea, võib nähtud pilti täiesti uskuma jääda - eriti kuna uuesti samasse kohta vaatama tulles rannajoon viimati vaadatuga võrreldes ikka samal kohal paikneb.

Kasutajaliidesesse tulid juurde nupud, et oleks võimalik nii igasse ilmakaarde kui üles ja allapoole liikuda. Horisontaalsuunas on arvestatud, et üks ühik võrdub ligikaudu ühe kilomeetriga. Kõrguse puhul aga lihtsalt muudetakse suurendust iga sammu juures sama koefitsiendi jagu. Punkti

andmete hoidmiseks kasutatakse endise täisarvulise Point'i asemel Point2D.Double-t mis võimaldab asukohti tunduvalt täpsemalt meelde jätta. Täisarvude puhul oleks siinse mõõtkava juures ühele punktile vastav üks kilomeeter, mis aga oleks külade ja linnade loomise soovi korral ilmselt liiga suur mõõtühik.

Maailmakoordinaatidest ekraanikoordinaatide arvutamiseks loodi eraldi funktsioon nagu ikka selliste arvutuste puhul tavaks. Punkti ekraanile joonistamisel arvestatakse nii punkti enese maailmakoordinaate, vaataja asukohta, suurendust kui akna suurust ja sealt tulenevat ekraani keskkoha koordinaadi väärtust. Koht, mis paikneb vaatamisel akna keskel, jääb sinna ka suurendamise või vähendamise korral.

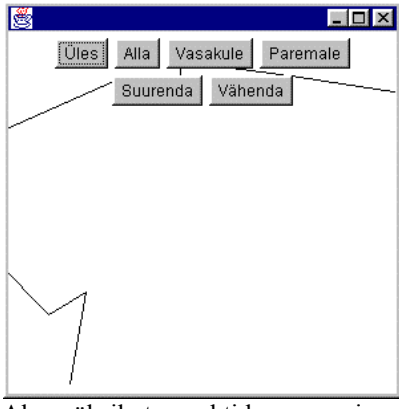
```
  int ekraaniX(double maailmaX){
    return ekeskx+(int)((maailmaX-vx)*suurendus);
  }
```

Edasi kommenteeritud rakenduse kood.

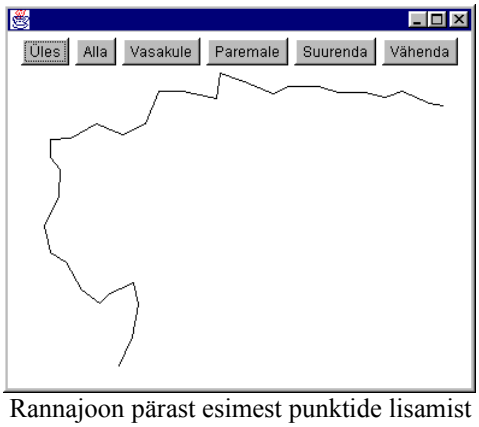
```
import java.applet.Applet;
import java.awt.*;
import java.awt.geom.*;
import java.util.*;
import java.awt.event.*;

public class Murdjoon6 extends Applet implements ActionListener{

  /** Ülesliikumisnupp. Vaataja koordinaadid vähenevad */
  Button yles=new Button("Üles");
  /** Allaliikumisnupp. Vaataja liigub lõuna suunas */
  Button alla=new Button("Alla");
  /** Nupp läände liikumiseks */
  Button vasakule=new Button("Vasakule");
  /** Nupp itta liikumiseks */
  Button paremale=new Button("Paremale");
  /**
  * Pilt suureneb liigutakse allapoole.
  * Nähtava osa joonele arvutatakse vajadusel punkte juurde.
  */
  Button suurenda=new Button("Suurenda");
  /**
  * Suurenduskordaja väheneb. Näiliselt liigutakse maapinnast kaugemale.
  */
  Button vahenda=new Button("Vähenda");
  /**
  * Kindlaksmääratud rannajoone punktide loetelu maailmakoordinaatides.
  */
  LinkedList punktid=new LinkedList();
  /**
  * Nähtav pikkus ekraanipunktides, millest alates asutakse joont poolitama.
  */
  double pikimJoonEkraanil=30;
  /**
  * Vaataja asukoha x maailmakoordinaatides.
  */
  double vx=286;
  /**
  * Vaataja asukoha y maailmakoordinaatides.
  */
  double vy=120;
  /**
  * Vaataja algne samm maailmakoordinaatides.
  */
  double vsamm=5;
  /**
  * Ekraani keskkkoha x.
  */
  int ekeskx;
  /**
  * Ekraani keskkkoha y.
  */
  int ekesky;
  /**
  * Koefitsient näitamaks, mitu ekraanipunkti vastab ühele
  * maailmakoordinaatides ühikule.
  */
  double suurendus=1;
  /**
  * Suhe, mille jagu suurenduskoefitsient suureneb või väheneb
  * alla või üles liikumisel.
  */
  double suurenduskordaja=1.1;
```

```
/**
* Kujunduse, kuularite ja algväärtuste paikasättimine.
*/
public Murdjoon6(){
  add(yles);
  add(alla);
  add(vasakule);
  add(paremale);
  add(suurenda);
  add(vahenda);
  yles.addActionListener(this);
  alla.addActionListener(this);
  vasakule.addActionListener(this);
  paremale.addActionListener(this);
  suurenda.addActionListener(this);
  vahenda.addActionListener(this);
  looAlgneJoon();
}

/**
* Algse rannajoone punktide sättimine maailmakoordinaatides.
*/
void looAlgneJoon(){
  punktid.add(new Point2D.Double(186, 249));
  punktid.add(new Point2D.Double(198, 180));
  punktid.add(new Point2D.Double(170, 197));
  punktid.add(new Point2D.Double(129, 155));
  punktid.add(new Point2D.Double(129, 61));
  punktid.add(new Point2D.Double(219, 21));
  punktid.add(new Point2D.Double(267, 27));
  punktid.add(new Point2D.Double(270, 6));
  punktid.add(new Point2D.Double(352, 17));
  punktid.add(new Point2D.Double(455, 33));
}


/**
* Liigutamine vastavalt nupuvajutustele.
*/
public void actionPerformed(ActionEvent e){
  double samm=vsamm/suurendus;
  if(e.getSource()==yles)    {vy-=samm;}
  if(e.getSource()==alla)    {vy+=samm;}
  if(e.getSource()==vasakule){vx-=samm;}
  if(e.getSource()==paremale){vx+=samm;}
  if(e.getSource()==suurenda){
    suurendus*=suurenduskordaja;
  }
  if(e.getSource()==vahenda){suurendus/=suurenduskordaja;}
  repaint();
}


/**
* Lisatavate punktide arvutus. Kui kahe punkti vaheline joon ekraanil
* kipub tulema suurem määratud väärtusest, siis leitakse
* joone keskkoha lähedale uus punkt ning tõmmatakse algse joone
* otspunktidest jooned sellesse punkti.
*/
public void lisaVahePunktid(){
 double pikimaJoonePikkus=pikimJoonEkraanil/suurendus;
 int koht=0;
 while(koht+1<punktid.size()){
  Point2D.Double p1=(Point2D.Double)punktid.get(koht);
  Point2D.Double p2=(Point2D.Double)punktid.get(koht+1);
  double kaugus=p1.distance(p2);
  if(kaugus>pikimaJoonePikkus && (kasSees(p1.x, p1.y) || kasSees(p2.x, p2.y))){
    Point2D.Double p3=new Point2D.Double(
      (p1.x+p2.x)/2+((Math.random()-0.5)*0.4*kaugus),
      (p1.y+p2.y)/2+((Math.random()-0.5)*0.4*kaugus)
    );
    punktid.add(punktid.indexOf(p2), p3);
    if(p1.distance(p3)<=pikimaJoonePikkus){
      koht=koht+1;
    }
  } else {
    koht=koht+1;
  }
 }

}

/**
```

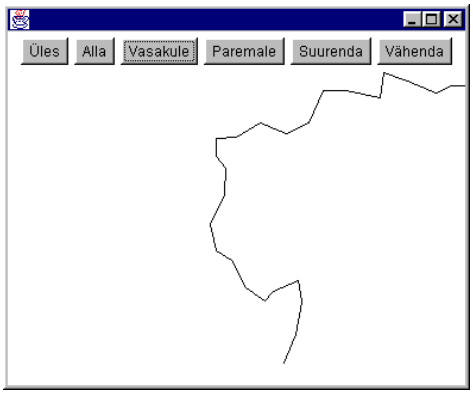
```
  * Maailmakoordinaatide teisendus ekraanikoordinaatideks, arvestatakse
  * suurendust ja kasutaja asukohta.
  */
  int ekraaniX(double maailmaX){
    return ekeskx+(int)((maailmaX-vx)*suurendus);
  }

  /**
  * Maailmakoordinaatide teisendus ekraanikoordinaatideks.
  */
  int ekraaniY(double maailmaY){
    return ekesky+(int)((maailmaY-vy)*suurendus);
  }

  /**
  * Kontroll, kas etteantud maailmakoordinaatidega punkt mahub
  * ekraanil vaatevälja.
  */
  boolean kasSees(double maailmaX, double maailmaY){
     int ex=ekraaniX(maailmaX);
     int ey=ekraaniY(maailmaY);
     return ex>0 && ex<getWidth() && ey>0 && ey<getWidth();
  }

  /**
  * Mälus olevatele andmetele vastavalt koostatakse ekraanile pilt.
  * Nähtava ala piires palutakse punkte luua niivõrd, et pikim
  * näha olev joon ei ületaks määratud pikkust.
  */
  public void paint(Graphics g){
    lisaVahePunktid();
    ekeskx=getWidth()/2;
    ekesky=getHeight()/2;
    for(int i=0; i<punktid.size()-1; i++){
      Point2D.Double p1=(Point2D.Double)punktid.get(i);
      Point2D.Double p2=(Point2D.Double)punktid.get(i+1);
      g.drawLine(ekraaniX(p1.getX()), ekraaniY(p1.getY()),
                 ekraaniX(p2.getX()), ekraaniY(p2.getY()));
    }
  }

  /**
  * Käivitus käsurealt.
  */
  public static void main(String[] argumendid){
    Frame f=new Frame();
    f.add(new Murdjoon6());
    f.setSize(300, 300);
    f.setVisible(true);
  }
}
```



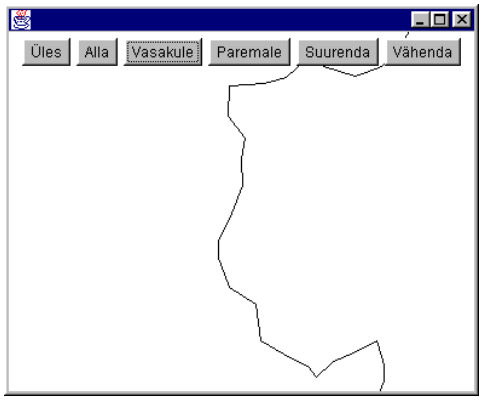
Algne üksikute punktidega rannajoon



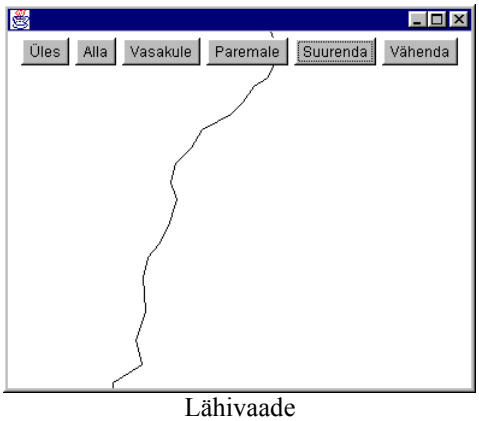
Rannajoon pärast esimest punktide lisamist



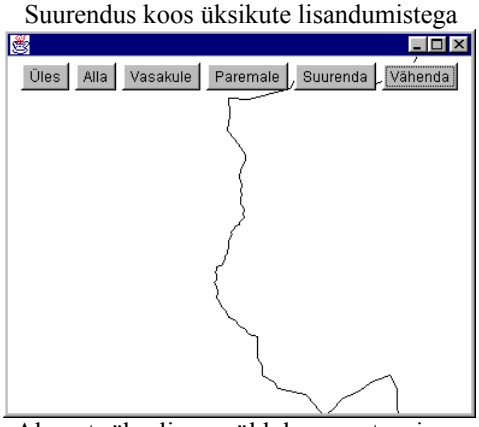
Lääneranniku nihutus pildi keskele



Suurendus koos üksikute lisandumistega



Lähivaade



Algsest säbrulisema üldplaan vaatepaigas.

Et illustreerimise mõttes oli pikima joone pikkuseks määratud 30 ekraanipunkti, siis paistab algne suurte nurkadega joon selgelt välja. Kui suurimaks lubatud pikkuseks panna aga näiteks kaks punkti, siis pole kasutajal kuigivõrd võimalust märgata, et algne nähtav kaart polegi kõikide võimalike hiljem vaadatavate kohtade pealt veel olemas. Et vaatama asumisel vastavad kohad kohe luuakse, võiks mulje jäädagi täiuslik.

Fraktali omaduste demonstreerimisel võibki lubada üha peenemaks minevat arvutust. Mingil hetkel kipub niimoodi Double 14st komakohast täpsusel väheks jääma ning tuleks leida miski täpsem võimalus koordinaatide arvutamiseks. Olgu selleks siis java.math.BigDecimal soovitud arvu komakohtade talletamiseks või mõni omaloodud vastava oskusega objekt. Kusjuures suuremat komakohtade arvu on vaja talletada vaid lähemal vaatlusel tekkivate punktide korral.

Kui eesmärgiks aga tegeliku keskkonna piisavalt tõetruu jäljendamine, siis võiks koos suurendusega muutuda ka tekkivate kujundite omadused. Kui simuleeritakse õhusõidukiga Eestimaa kohal lendamist, siis tõenäoliselt pole põhjust välja arvutada vähem kui sentimeetrise läbimõõduga objekte. Samas ei pruugi nähtav ja täienev osa piidruda sugugi vaid rannajoonega.

Ülesandeid

Virtuaalse Eestimaa täiendus

- Täienda rannajoont, lisa Eestimaale ka ida- ja lõunapiir.
- Nähtavad jooned võivad jaguneda mitmesse kogumisse. Lisa eraldi kogumina Võrtsjärve rannajoon.
- Iga kogumi juures on lisaks punktile kirjas ka vastava joone värv.

- Sinise värviga tähista tähtsamad jõed. Ka neile mõeldakse lähemal vaatamisel välja käänakud.
- Jõgede ja rannajoone käänakute juures ei lähe joone pikkus väiksemaks kui 1 meeter.
- Alates suurendusest 1 ekraanipunkt = 10 meetrit hakatakse välja mõtlema ning näitama puid. Kord loodud puud jäävad samadele kohtadele.

Naerunägu

- Joonista naeratav nägu, kelle kummaski silmas oleks samuti naerunägu.
- Joonise suurendamisel ilmub igast silmast jälle uus naerunägu välja.
- Lisaks eelmisele saab kasutaja panna joonise pidevalt suurenema ning määrata näo kallet.

Viisnurgad

- Ekraanile joonistatakse viisnurk.
- Üha väiksemad erivärvilised seest täidetud viisnurgad on üksteise sees, kusjuures sisemise viisnurga tipp läheb välimise viisnurga serva keskkohta.
- Üksteise sees on kuni 100 viisnurka. Kerimisribaga saab määrata, millise koha peal sisemise viisnurga nurk välimise serva puutub.

Maatriksarvutused joonistamisel

Korrutamine, nihutamine, keeramine, toimingute ühendamine

Enamiku joonistamiseks tarvilikku saab välja mõelda põhikoolimatemaatika ning terve talupojamõistuse abil. Kuni on tegemist üksikute paigal seisvate või sirgelt liikuvate punktidega, siis polegi enamasti muud vaja. Kui aga teisendusi ning kujundeid palju saab, siis muutub nii programmi kirjutamine kui töötamine järjest mahukamaks ning tuleb abivahendeid otsida. Keerukama kujundi kuhugi paigutamine taandub enamasti hulga punktide ümbertõstmisele. Keeramise korral tuleb selleks teha töömahukaid trigonomeetrilisi arvutusi. Kui üksteisele järgneb mitu nihutamist ning mitu keeramist, siis kasvab töö maht päris suureks ning suure punktide hulga korral seda enam.

Üheks võimaluseks arvutuste kiirendamisel on tõenäoliselt vaja minevate keerulisemate arvutuste tulemused varem välja arvutada ning neid vajadusel mälust otsida. Kuna siinuse arvutamine on vähemalt tuhat korda aeglasem kui tavaline liitmine või mälust lugemine, siis juba paari tuhande punkti juures on pildi keeramisel vahe märgata. Kuigi põhimõtteliselt võib olla vaja arvutada igasugu nurki, piisab joonistamiste juures siiski enamasti vaid ühekraadisest täpsusest. Nii on vaja eelnevalt välja arvutada siinused vaid 360 kraadi jaoks, asendusvalemeid kasutades võib kergesti piirduda ka veerandiga sellest. Kui veel veidi edasi mõtelda, siis kuna koosinus jookseb siinusest täisnurga jagu taga, siis saab samade 90 kraadi jagu välja arvutatud siinuste abil arvuti tarvis kiirete liitmis-ja lahutustehete abil leida mõlema funktsiooni väärtusi kogu olemasolevas vahemikus.

Graafikaarvutusi saab märgatavalt kiirendada ning osalt grupeerimise teel vähendada maatriksite abil. Muidu kipub nende sageli õppekavas kohustuslike tabelitega suhteliselt vähe igapäevaülesannete juures otse peale hakata olema, aga kui on vaja vähegi keerukamaid kujundeid pinnal või ruumis ümber paigutada, siis parajasti sobiva abiliidese puudumisel jõuab lõpuks paratamatult maatriksiteni välja. Silma on hakanud nende järgmised head omadused:

- Ühe kujundi puhul on vaja töömahukaid arvutusi sooritada vaid korra, ülejäänud juhtudel saab uue koordinaadi välja arvutada paari liitmis- ning korrutustehtega.
- Soovi korral võib kogu kujundi punkte käsitleda koos
- Nihutamisi, suurendamisi ja keeramisi saab vaadelda tervikoperatsioonidena, ei teki lootusetut killustatust.

Iga operatsiooni saab kirjeldada ühe maatriksina ning järjestikused operatsioonid saab ühendada lihtsalt üksteise järel seisvaid maatrikseid korrutades. Lähemad selgitused näidete varal.

Väike meeldetuletus, kuidas maatrikseid korrutatakse. Korrutatavad maatriksid paremal ning vastus vasakul. Vasakpoolse maatriksi ridadel asuvad elemendid korrutatakse parempoolse maatriksi vastavatel veergudel olevate väärtustega. Sellest järeldub, et korrutamisel peab olema vasakpoolse maatriksi ridu sama palju kui parempoolse maatriksi veerge ning korrutise tulemusena tekkivas maatriksis on ridu nii palju kui esimeses ja veerge nagu teises maatriksis.

$$\begin{bmatrix} 1*5+2*6 \\ 3*5+4*6 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 \\ 6 \end{bmatrix}$$

Samad väärtused õnnestub korrutada, kui maatriksite read ja veerud ning järjekord vahetada.

$$\begin{bmatrix} 5*1+6*2 & 5*3+6*4 \end{bmatrix} = \begin{bmatrix} 5 & 6 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

Arvutades koordinaate ümber järgneva valemi järgi

$\begin{cases} x^1 = ax + by \\ y^1 = cx + dy \end{cases}$, saab selle kirja panna maatriksitena $\begin{bmatrix} x^1 & y^1 \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} a & c \\ b & d \end{bmatrix}$.

Kui punkte on vaja üle kanda korraga rohkem kui üks, siis saab selle ette võtta järgmise korrutise abil:

$$\begin{bmatrix} x_1^1 & y_1^1 \\ x_2^1 & y_2^1 \\ x_3^1 & y_3^1 \end{bmatrix} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \end{bmatrix} \begin{bmatrix} a & c \\ b & d \end{bmatrix}$$. Iga punkti kohta tuleb lihtsalt esimeses maatriksis üks rida.

Arvestades enne toodud üldvalemit, saab välja tuua mõned erijuhud.

$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ ehk uue x-i arvutamisel on vana x-i kordaja 1 ning uue y-i arvutamisel on vana y-i kordaja 1,

pikemalt $\begin{cases} x^1 = 1*x + 0*y \\ y^1 = 0*x + 1*y \end{cases}$ ehk $\begin{cases} x^1 = x \\ y^1 = y \end{cases}$ ehk koordinaadid jäävad muutumatuks.

$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$ ehk $\begin{cases} x^1 = x \\ y^1 = -y \end{cases}$ ehk y-koordinaat muutub vastupidiseks ning punkt või joonis peegeldatakse x-telje suhtes. Soovides järjestikku kõigepealt venitada joonist x-telge pidi kaks korda pikemaks $\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$ ning seejärel y-telge pidi kaks korda pikemaks $\begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$, võib kõigepealt kaks tekkinud muutust kokku korrutada $\begin{bmatrix} 2*1+0*0 & 2*0+0*2 \\ 0*1+1*0 & 0*0+1*2 \end{bmatrix}$ ehk $\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$ ning alles siis punkti koordinaadid tulemusega läbi korrutada ning otse lõpptulemus saada.

Järgnevalt koodinäide, kuidas etteantud maatriksi järgi algsetel koordinaatidel asuvat ringi uude kohta transportida valemi järgi x^1=2x ning y1=-y. Maatriksiks on 2x2 algväärtustatud massiiv nimega m. Muutujad x ja y tähistavad punkti algseid koordinaate, keskx ja kesky näitavad joonistatava ala keskpunkti ekraanipunktides ning neid läheb vaja vaid joonistamisel. Meetod joonistaTeljed tõmbab ekraanile keskpunktis ristuvad horisontaal- ning vertikaaljoone. Käsk joonistaKujund loob etteantud koordinaatidele ringi, nii et ringi keskpunkt jääb soovitud kohale. Joonistamise arvutamine jääb käsu paint sisse. Uued koordinaadid leitakse algseid muutusmaatriksiga korrutades $\begin{bmatrix} x^1 & y^1 \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & -1 \end{bmatrix}$. Lahtiseletatult korrutatakse algsed koordinaadid x ja y kõigepealt parempoolse maatriksi vasakpoolse veeru väärtustega ning tulemuseks saadakse uue x-i koordinaat x^1. Edasi korrutatakse algsete koordinaatide maatriks teisendusmaatriksi parema veeruga ning saadakse uus y-koordinaat y^1.

$\begin{bmatrix} x*2+y*0 & x*0+y*(-1) \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & -1 \end{bmatrix}$. Kui tühjad liikmed maha arvata, siis jääb järele

$\begin{bmatrix} 2x & -y \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & -1 \end{bmatrix}$, ehk tulemus, mida soovisimegi saavutada.

Massiivile võib anda elemendid algväärtustamisel. Kui ühemõõtmelise massiivi algväärtustamisel võis väärtused paigutada lihtsalt komadega eraldatult loogeliste sulgude vahele, siis kahemõõtmelise puhul tuleb ka iga rida eraldi loogeliste sulgude vahele panna ning komadega eraldada. Nii nagu veergude elemendid on üksteisest komadega eraldatuna rida moodustavas plokis, nii loovad read üheskoos komadega eraldatult ridade massiivi ning kokku tulebki kahemõõtmeline massiiv, kus esimene indeks näitab rea- ning teine veeru numbrit.

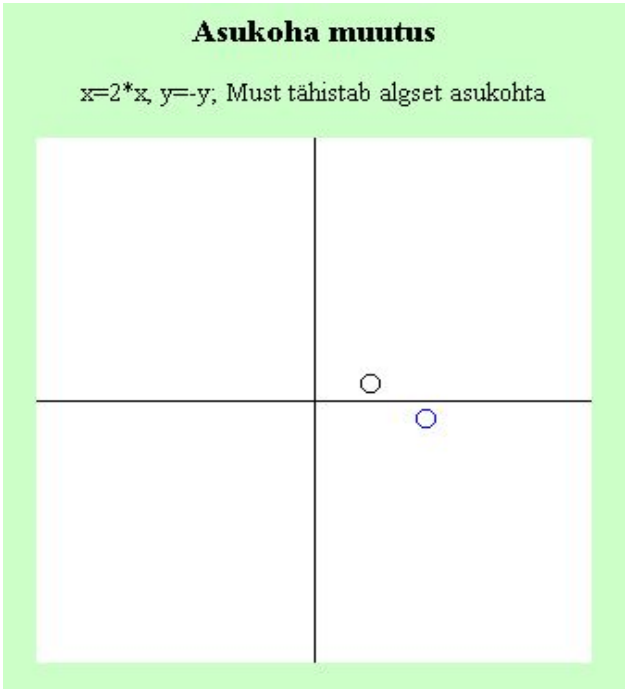
```
int[][] m=new int[][]{
    {2,  0},  //teisendus x=2x
    {0, -1}  //     y=-y
  };
```

Korrutamine (uue x-i ja y-i arvutamine):

```
int ux=x*m[0][0]+y*m[1][0];
int uy=x*m[0][1]+y*m[1][1];
```

Kuna massiivi elemendid algavad nullist, siis on maatriksi esimese rea number 0, teise rea number 1, samuti veergude puhul. Nii korrutataksegi uue x-i leidmiseks kõigepealt vana x maatriksi esimese veeru esimese elemendiga ning siis liidetakse tulemusele vana y-i korrutis teise rea esimese elemendiga. Uue y-i leidmisel sama lugu, vaid korrutatakse algsed x ja y maatriksi teise veeru elementidega. Kogu programm näeks välja järgmine:

```
import java.applet.Applet;
import java.awt.*;

public class Maatriks1 extends Applet{
  int[][] m=new int[][]{ //teisendus x=2x
    {2,  0},             //     y=-y
    {0, -1}
  };
  int x=30, y=10;
  int keskx=150, kesky=150;

  public void joonistaTeljed(Graphics g){
    g.drawLine(keskx, 0, keskx, 2*kesky);
    g.drawLine(0, kesky, 2*keskx, kesky);
  }

  public void joonistaKujund(Graphics g, int kx, int ky){
    g.drawOval(keskx+kx-5, kesky-ky-5, 10, 10);
  }

  public void paint(Graphics g){
    joonistaTeljed(g);
    joonistaKujund(g, x, y);
    int ux=x*m[0][0]+y*m[1][0];
    int uy=x*m[0][1]+y*m[1][1];
    g.setColor(Color.blue);
    joonistaKujund(g, ux, uy);
  }
  public static void main(String argumendid[]){
    Frame f=new Frame();
    f.add(new Maatriks1());
    f.setSize(300, 300);
    f.setVisible(true);
  }
}
```



Klass maatriksarvutuste tarvis.

Kui teisendusi tuleb programmi sisse enam, siis kõiki ükshaaval lahti kirjutades tuleb arvutuste peale kokku ikka palju ridu ning maatriksite lisamine ei pruugigi kuigi palju kasu tuua, pigem lihtsalt veel üks tülin juures millega arvestada. Mida rohkem ridu programmis, seda kergem on ka kusagile vigu teha. Enamlevinumate maatriksarvutuste ning ka maatriksi andmete hoidmiseks võib paaril lehel kirjutada vastava klassi või kirjutuslaiskuse puhul enesele võrgu pealt sobiv otsida. Siin näites on loodud klass suvalise soovitavate ridade ja veergude arvuga maatriksi tarvis. Peotäis konstruktoreid enamlevinud kujul andmete sisestamiseks ning meetod kahe maatriksi korrutamiseks, tegevus, mida graafikaarvutuste puhul enim vaja

läheb. Lisaks staatilised meetodid üherealise maatriksi loomiseks, et kergemini punkti asukohta määravat maatriksit koostada. Nagu korrutamise puhul näha, tuleb seal kokku kolm tsüklit üksteise sisse paigutada. Kaks tükki ridade ning veergude läbi käimiseks ning kolmas, et arvutatava lahtri tarvis tehtavad korrutised kokku liita. Maatriksi sees olevaid andmeid hoitakse ja arvutatakse reaalarvudena arvutuste täpsuse huvides, joonistamise koordinaadid neile vastavatest pesadest aga küsitakse välja täisarvudena, et oleks kergem tulemusi täisarvulistele ekraanikoordinaatidele paigutada.

```
public class Maatriks{
  double m[][];
  public Maatriks(int ridu, int veerge){
    m=new double[ridu][veerge];
  }
  public Maatriks(double a11, double a12, double a21, double a22){
    m=new double[][]{
      {a11, a12},
      {a21, a22}
    };
  }
  public Maatriks(
    double a11, double a12, double a13,
    double a21, double a22, double a23,
    double a31, double a32, double a33
   ){
    m=new double[][]{
      {a11, a12, a13},
      {a21, a22, a23},
      {a31, a32, a33}
    };
  }


/**
* Luuakse ühe rea ning kahe veeruga maatriks, algväärtusteks parameetritena
* antud väärtused. Tarvitatakse arvutigraafikas tasandil suurendamise
* ning keeramise tarvis.
*/
  static Maatriks XY(double x, double y){
    Maatriks m1=new Maatriks(1, 2);
    m1.m[0][0]=x;
    m1.m[0][1]=y;
    return m1;
  }

  static Maatriks XYZ(double x, double y, double z){
    Maatriks m1=new Maatriks(1, 3);
    m1.m[0][0]=x;
    m1.m[0][1]=y;
    m1.m[0][2]=z;
    return m1;
  }

  public int X(){ return (int)m[0][0];}
  public int Y(){ return (int)m[0][1];}
  public int ridadeArv(){return m.length;}
  public int veergudeArv(){return m[0].length;}
  public Maatriks korruta(Maatriks m2){
    if(veergudeArv()!=m2.ridadeArv())
      throw new ArithmeticException("Vigane maatriksite suurus "+
        veergudeArv()+" "+m2.ridadeArv());
    Maatriks m3=new Maatriks(ridadeArv(), m2.veergudeArv());
    for(int i=0; i<m3.ridadeArv(); i++){
      for (int j=0; j<m3.veergudeArv(); j++){
       for(int k=0; k<m3.veergudeArv(); k++){
         m3.m[i][j]+=m[i][k]*m2.m[k][j];
       }
      }
    }
    return m3;
  }
}
```

Võrreldes algse näitega muutub nüüd programmi põhiosa lihtsamaks ja lühemaks, sest pole enam vaja arvutusi koodi sisse kirjutada, nende eest hoolitseb eraldi klass. Nii algse asukoha kui muutuse saab kirjeldada maatriksina ning uus asukoht leitakse nende korrutisena. Kui eespool seisvat maatriksi klassi vaadata, siis nelja parameetriga konstruktori puhul loetakse kaks esimest esimese rea ning kaks järgmist maatriksi teise rea väärtusteks.

```
import java.applet.Applet;
```

```
import java.awt.*;

public class Maatriks2 extends Applet{
  Maatriks muutus=new Maatriks(2, 0, 0, -1);

  Maatriks asukoht=Maatriks.XY(30, 10);
  int keskx=150, kesky=150;

  public void joonistaTeljed(Graphics g){
    g.drawLine(keskx, 0, keskx, 2*kesky);
    g.drawLine(0, kesky, 2*keskx, kesky);
  }

  public void joonistaKujund(Graphics g, int kx, int ky){
    g.drawOval(keskx+kx-5, kesky-ky-5, 10, 10);
  }

  public void paint(Graphics g){
    joonistaTeljed(g);
    joonistaKujund(g, asukoht.X(), asukoht.Y());
    Maatriks uuskoht=asukoht.korruta(muutus);
    g.setColor(Color.blue);
    joonistaKujund(g, uuskoht.X(), uuskoht.Y());
  }
}
```

Keeramine

Soovides algset punkti ümber koordinaattelgede keskpunkti keerata, võib kasutada teisendusmaatriksit $\begin{bmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{bmatrix}$, kus a on soovitava keeramise nurk. Erijuhul, kui soovitakse keerata täisnurga jagu vastupäeva, tuleb teisenduseks $\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$ ehk x=-y ning y=x. All näites pööratakse PI/6 ehk 30º.

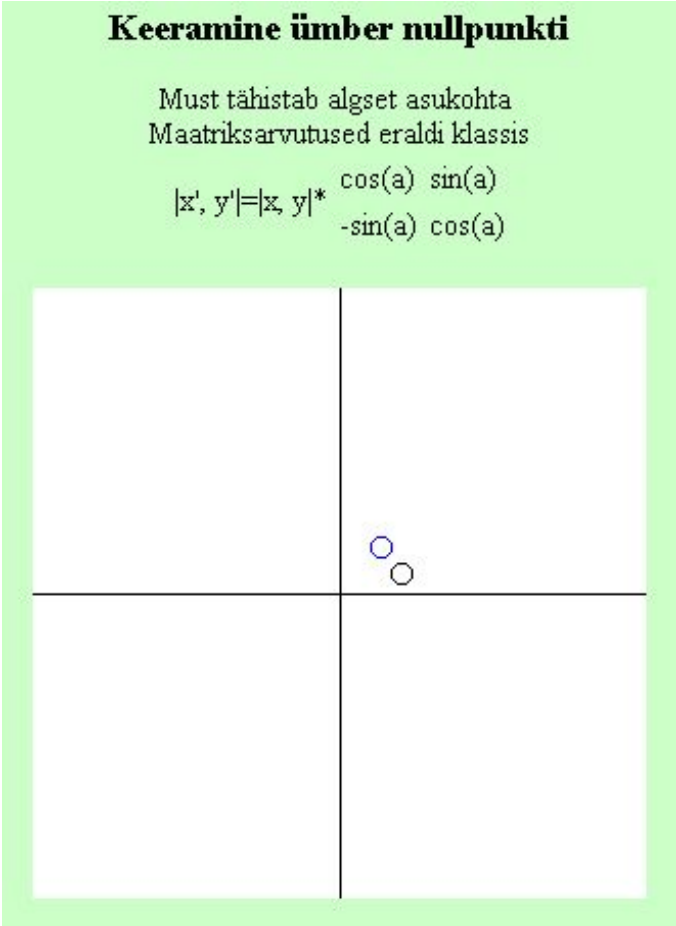
```
import java.applet.Applet;
import java.awt.*;

public class Maatriks3 extends Applet{
  double nurk=Math.PI/6;
  Maatriks muutus=new Maatriks(
    Math.cos(nurk), Math.sin(nurk),
    -Math.sin(nurk), Math.cos(nurk)
  );

  Maatriks asukoht=Maatriks.XY(30, 10);
  int keskx=150, kesky=150;

  public void joonistaTeljed(Graphics g){
    g.drawLine(keskx, 0, keskx, 2*kesky);
    g.drawLine(0, kesky, 2*keskx, kesky);
  }

  public void joonistaKujund(Graphics g, int kx, int ky){
    g.drawOval(keskx+kx-5, kesky-ky-5, 10, 10);
  }
  public void paint(Graphics g){
    joonistaTeljed(g);
    joonistaKujund(g, asukoht.X(), asukoht.Y());
    Maatriks uuskoht=asukoht.korruta(muutus);
    g.setColor(Color.blue);
    joonistaKujund(g, uuskoht.X(), uuskoht.Y());
  }
}
```

Nagu ennist kirjas, juhul kui soovitakse mitu muutust lisada ühtejärge, siis tuleb vastavate muutuste maatriksid lihtsalt ühtejärge kokku korrutada ning tulemuseks on soovitud muutusi sisaldav maatriks, millega algseid koordinaate läbi korrutades saab ühekorrga lõpptulemusele vastavad punktid teada. Mõnikord on muutuste järjekord tähtis nagu ka allpoololevas näiteks. Kokkuvõtlik nihe koosneb keeramisest ning x-koordinaadi korrutamisest koefitsiendiga. Kui näiteks x-telje lähedal asuva punkti koordinaate kõigepealt paarkümmend kraadi ümber keskpunkti keerata ning seejärel x-i väärtust mõne korra suurendada, siis

lõpppunkti y jääb sama suureks kui see oli pärast keeramist. Kui aga kõigepealt x-i suurendada ning alles seejärel sama suure nurga jagu keerata, siis lõpptulemuse y on tõenäoliselt tunduvalt suurem kui eelmisel juhul, sest kui pikemat maas olevat latti sama nurga jagu püstipoole kallutada, siis tõuseb tema ots kõrgemale kui lühikese lati puhul.

Juurde on lisatud kerimisribad, et kasutaja saaks kergemini nurka ning suurendust muuta. Iga kerimisriba liigutuse peale arvutatakse uuesti välja muutuse jaoks tarvilikud maatriksid ning tulemus joonistatakse uuesti ekraanile.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class Maatriks4 extends Applet implements AdjustmentListener{

  Scrollbar nurgasb=new Scrollbar(Scrollbar.HORIZONTAL, 314, 10, 0, 628);
  Scrollbar suurendusesb=new Scrollbar(Scrollbar.HORIZONTAL, 110, 10, 0, 200);
  Maatriks muutus;
  Maatriks asukoht=Maatriks.XY(30, 10);
  int keskx=150, kesky=150;

  public Maatriks4(){
    Panel p1=new Panel(new GridLayout(2, 2));
    p1.add(new Label("Nurk:"));
    p1.add(nurgasb);
    p1.add(new Label("Suurendus x:"));
    p1.add(suurendusesb);
    nurgasb.addAdjustmentListener(this);
    suurendusesb.addAdjustmentListener(this);
    setLayout(new BorderLayout());
    add(p1, BorderLayout.SOUTH);
    arvutaMuutus();
  }

  void arvutaMuutus(){
   double nurk=(nurgasb.getValue()-314)/100.0;
   Maatriks nurgam=new Maatriks(
    Math.cos(nurk), Math.sin(nurk),
    -Math.sin(nurk), Math.cos(nurk)
   );
   double suurendus=(suurendusesb.getValue()-100)/10.0;
   Maatriks suurendusem=new Maatriks(
     suurendus, 0,
     0, 1
   );
   muutus=nurgam.korruta(suurendusem);  //enne keerab, siis suurendab
  // muutus=suurendusem.korruta(nurgam); //enne suurendab, siis keerab
   repaint();
  }


  public void joonistaTeljed(Graphics g){
    g.drawLine(keskx, 0, keskx, 2*kesky);
    g.drawLine(0, kesky, 2*keskx, kesky);
  }

  public void joonistaKujund(Graphics g, int kx, int ky){
    g.drawOval(keskx+kx-5, kesky-ky-5, 10, 10);
  }

  public void paint(Graphics g){
    joonistaTeljed(g);
    joonistaKujund(g, asukoht.X(), asukoht.Y());
    Maatriks uuskoht=asukoht.korruta(muutus);
    g.setColor(Color.blue);
    joonistaKujund(g, uuskoht.X(), uuskoht.Y());
  }
  public void adjustmentValueChanged(AdjustmentEvent e){
    arvutaMuutus();
  }
}
```



Nihutamine

Kahemõõtmelise punkti nihutamiseks läheb tarvis kolmemõõtmelist maatriksit, kahemõõtmelisest ei piisa. Et punkti koordinaate annaks sellise maatriksiga läbi korrutada, tuleb ka seal kolmas suurus juurde võtta. Punkt muudetakse ruumiliseks, andes talle z-i väärtuseks 1. Nõnda, kui soovida punkti asukohaga x, y nihutada paika x^1, y^1, tuleks kasutada järgmist teisendust: $\begin{bmatrix} x^1 & y^1 & z^1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{bmatrix}$, kus a ning b väärtused tähistavad vastavalt x ning y koordinaatide nihet vastavaid telgi pidi. Ka muud siiamaani kasutatud teisendused võib samaks jätta, kasutades vaid maatriksi esimest kaht rida ning veergu ning jättes z-koordinaadi väärtuseks ning kordajaks arvu 1. Nii näiteks näeks kolmemõõtmeliste maatriksitena arvutatult ümber nullpunkti tasandil (ehk ümber z-telje ruumis) keeramine välja

$$\begin{bmatrix} x^1 & y^1 & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$ ning tulemused on samade nurkade puhul samad, kui kahemõõtmelise maatriksiga arvutades. Nõnda selgub ka avaldisele otsa vaadates, sest võrreldes eelmise väiksema arvutusega lisandub x-i ning y-i arvutamisel vaid liige 1*0, mis väärtuse 0 tõttu lõpptulemust ei muuda. Suuresti arvutigraafikas kasutataksegi 3*3 muutusmaatrikseid, sest nii on võimalik kõik enamasti ette tulevad teisendused ühesuguse suurusega tabelitesse kokku panna ning pole muret kahest mõõtmest kolme või pärast tagasi ülekandmise juures.

Järgnevas näites paistabki, kuidas algse punkti koordinaati kümne punkti jagu paremale ning kolmekümne jagu üles nihutada. Loomulikult saaks selle toiminguga ilma maatriksiketa tunduvalt lihtsamalt toime, kuid nii on täiendav vahend komplektis juures, mida on võimalik ühes teiste omasugustega suurest hulgast punktides koosneva kujundi asendi muutmiseks kasutada.

```
import java.applet.Applet;
import java.awt.*;

public class Maatriksnihe extends Applet{
  double nihkex=10, nihkey=30;
  Maatriks muutus=new Maatriks(
     1,      0,      0,
     0,      1,      0,
     nihkex, nihkey, 1
  );

  Maatriks asukoht=Maatriks.XYZ(30, 10, 1);
  int keskx=150, kesky=150;

  public void joonistaTeljed(Graphics g){
    g.drawLine(keskx, 0, keskx, 2*kesky);
    g.drawLine(0, kesky, 2*keskx, kesky);
  }

  public void joonistaKujund(Graphics g, int kx, int ky){
    g.drawOval(keskx+kx-5, kesky-ky-5, 10, 10);
  }

  public void paint(Graphics g){
    joonistaTeljed(g);
    joonistaKujund(g, asukoht.X(), asukoht.Y());
    Maatriks uuskoht=asukoht.korruta(muutus);
    g.setColor(Color.blue);
    joonistaKujund(g, uuskoht.X(), uuskoht.Y());
  }
}
```

Keeramine ümber määratud punkti.

Otseselt lihtsat valemit keskkohast erineva punkti ümber keeramiseks pole, selline vajadus kipub aga kirjutamise jooksul ikka tekkima. Olgu või tegemist ekraanil jalutava kilponnaga (millega paratamatult puutub kokku näiteks joonistavas programmeerimiskeeles LOGO), kes soovib mingis punktis esese asendit ning liikumissuunda muuta. Üheks võimaluseks on pidevalt joonistamisel hoida meeles kilpkonna enese keskkohta ning jalgu joonistada arvestades keskpunktist. Tahtes aga arvutamist universaalsemaks teha ning mitte pidevalt hoida meeles ning kasutada arvutamisel topeltkoordinaate, võib koostada maatriksi ka ümber määratud punkti keeramiseks. Selle saab luua, kasutades järgemööda olemasolevaid oskusi. Kõigepealt tuleb soovitav pöördekeskpunkt nihutada nullpunkti. Ümber nullpunkti keeramine käib lihtsa tuttava arvutuse teel ning kujundi (näiteks kilpkonna) õigesse algsesse kohta tagasi paigutamiseks tuleb taas kõik punktid vajaliku nihke jagu tagasi liigutada. Mujalgi toimiv reegel, et kui mõnda asja on liialt keeruline teha, siis püüa see jagada tuttavateks alamtöödeks ning nendega ükshaaval hakkama saada. Ega siis tulemuski tulemata ei jää. Matemaatiliselt näeks selline nullpunkti nihutamine, keeramine ja tagasi nihutamine välja järgnevalt, tähistades a ja b-ga keeramise keskkohta ning x-i ja y-ga keeratava punkti asukohti.

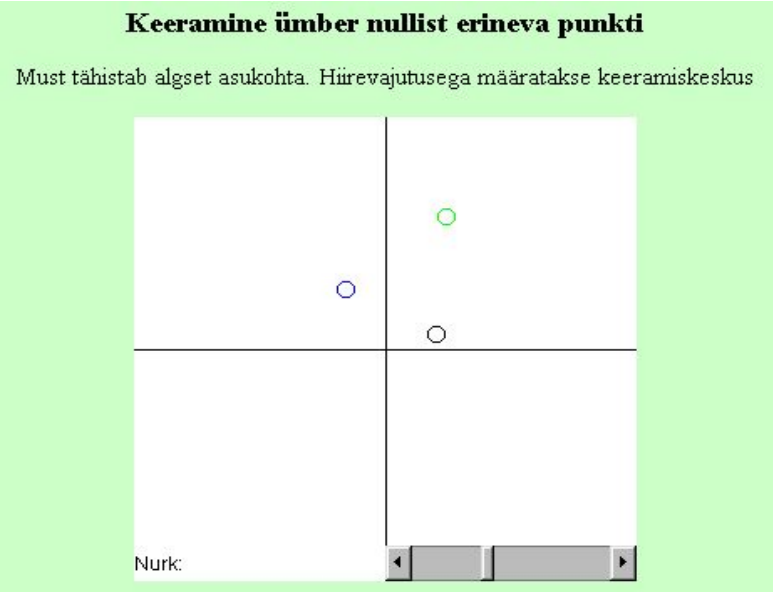
$$\begin{bmatrix} x^1 & y^1 & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -a & -b & 1 \end{bmatrix} \begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{bmatrix}$$

Kui üle kanda on rohkem kui üks punkt, siis võib arvutuse kiiruse huvides kõigepealt tagumised kolm maatriksit kokku korrutada üheks muutusmaatriksiks ning siis esimest koordinaatide maatriksit tekkinud muutustemaatriksiga läbi korrutades saame sama väikese arvutuskuluga teada uued koordinaadid, ükskõik kui keeruline ka tehtav nihe peaks olema. Kui tegemist on 3x3 reaalarvumaatriksiga, siis on arvuti jaoks ükskõik, kui ümmargused seal sees paiknevad arvud juhtuvad olema. Ning kõik tänu maatriksite korrutamise assotsiatiivsuse seadusele, mis ütleb, et $(M_1M_2)M_3 = M_1(M_2M_3)$. Nii saab tagumised muutusmaatriksid varakult tagavaraks ära korrutada ning samu teisendusi nii palju kordi rakendada, kui palju algseid asukohti ümber tõsta on.

Sama algoritmi realiseering programmina. Keeramise keskpunkti saab kasutaja hiirega määrata, samuti kerimisriba abil muuta nurka, kui palju algse asendiga võrreldes keerata.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class Maatriks4a extends Applet implements AdjustmentListener{

  Scrollbar nurgasb=new Scrollbar(Scrollbar.HORIZONTAL, 314, 10, 0, 628);
  Maatriks muutus;
  Maatriks asukoht=Maatriks.XYZ(30, 10, 1);
```

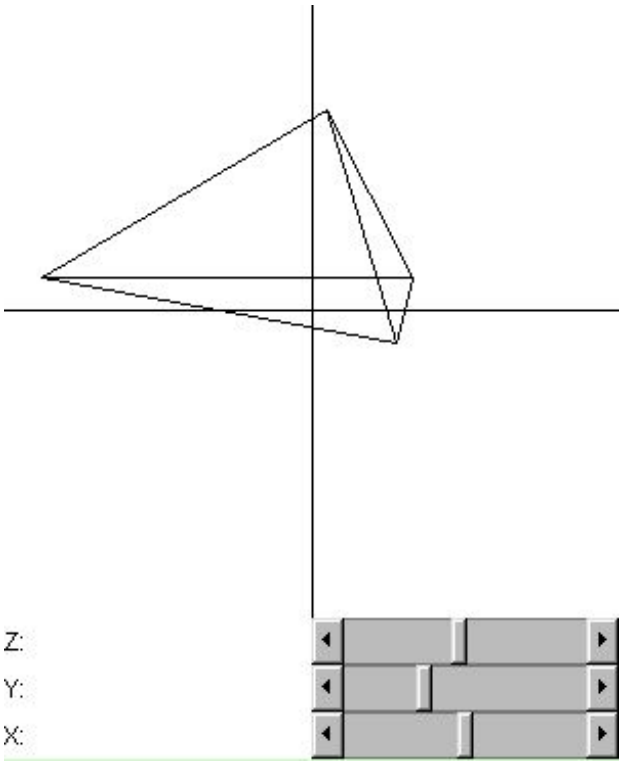
```
  Maatriks keermekeskus=Maatriks.XYZ(50, 30, 1);
  int keskx=150, kesky=150;

  public Maatriks4a(){
    Panel p1=new Panel(new GridLayout(1, 2));
    p1.add(new Label("Nurk:"));
    p1.add(nurgasb);
    nurgasb.addAdjustmentListener(this);
    setLayout(new BorderLayout());
    add(p1, BorderLayout.SOUTH);
    addMouseListener(
     new MouseAdapter(){
       public void mousePressed(MouseEvent e){
         keermekeskus=Maatriks.XYZ(
           e.getX()-keskx,
           -(e.getY()-kesky), 1
         );
         arvutaMuutus();
       }
     }
    );
    arvutaMuutus();
  }

  void arvutaMuutus(){
   double nurk=(nurgasb.getValue()-314)/100.0;
   Maatriks nurgam=new Maatriks(
     Math.cos(nurk), Math.sin(nurk), 0,
    -Math.sin(nurk), Math.cos(nurk), 0,
                  0,              0, 1
   );
   Maatriks keskelenihe=new Maatriks(
                     1,                 0, 0,
                     0,                 1, 0,
     -keermekeskus.X(), -keermekeskus.Y(), 1
   );
   Maatriks paikanihe=new Maatriks(
     1, 0, 0,
     0, 1, 0,
     keermekeskus.X(), keermekeskus.Y(), 1
   );
   muutus=keskelenihe.korruta(nurgam).korruta(paikanihe);
   repaint();
  }


  public void joonistaTeljed(Graphics g){
    g.drawLine(keskx, 0, keskx, 2*kesky);
    g.drawLine(0, kesky, 2*keskx, kesky);
  }

  public void joonistaKujund(Graphics g, int kx, int ky){
    g.drawOval(keskx+kx-5, kesky-ky-5, 10, 10);
  }

  public void paint(Graphics g){
    joonistaTeljed(g);
    joonistaKujund(g, asukoht.X(), asukoht.Y());
    Maatriks uuskoht=asukoht.korruta(muutus);
    g.setColor(Color.blue);
    joonistaKujund(g, uuskoht.X(), uuskoht.Y());
    g.setColor(Color.green);
    joonistaKujund(g, keermekeskus.X(), keermekeskus.Y());
  }
  public void adjustmentValueChanged(AdjustmentEvent e){
    arvutaMuutus();
  }

}
```



Pööramine ümber telgede.

Kui pööramine xy tasandil ehk ümber z-telje käis maatriksi järgi $\begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$, siis on näha, et peadiagonaalil on telje, mille ümber pöörati, väärtuseks 1. Ridade ja veergude muud elemendid, mis selle koordinaadiga seotud, on nullid. Sarnaselt pööratakse ka ümber teiste telgede. Ümber x-i $\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{bmatrix}$ ning ümber y-i $\begin{bmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{bmatrix}$. Kui tahame kõik koordinaatteljed pidada nullpunktist eemalduvatena, siis arvestades kruvireegli järgi, et päripäeva keeramine viib edasi ja vastupäeva tagasi, tuleks tegemist xy, yz ning zx tasanditega ning ümber y-telje keeramise maatriks tuleks siis vastassuunaline ehk $\begin{bmatrix} \cos\alpha & 0 & -\sin\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{bmatrix}$. Mitut teisendust üheskoos rakendades on oluline teisenduste järjekord. Kui kõigepealt pööratata ümber z-i ning seejärel pöörata ümber y-i, siis keerata enam mitte ümber algse kujundi y-telje, vaid pärast esimest pööret asetunud kujundi y-telje. All näites saab kerimisribade abil määrata, kui palju joonistatav tetraeeder ümber millise telje keeratud on. Kui ribad asuvad keskel, siis on kõik pöördenurgad nullid ning kujund algasendis. Koodis leitakse vastavalt iga kerimisriba asendile vastava telje suhtes pöörav muutusmaatriks ning siis korrutatakse muutused kokku.

```
muutus=mz.korruta(my).korruta(mx);
```



Ülesandeid

Maatriksarvutused

Korruta maatriksid

```
 [2 3]|2 0|
      |0 2|
(kirjalikult)
```

- Võimalda kasutajal sisestada (nt. tekstiväljadesse) punkti koordinaadid ning 2*2 teisendusmaatriks. Näita ekraanil algse ning liigutatud punkti asukohta.
-
- Kujundiks on punkte läbiv joon. Punktide koordinaadid (5 tk.) võetakse failist, teisendusmaatriks kirjutatakse tekstiväljadesse. Ekraanil näidatakse nii algset kui liigutatud joont. Keerulisemal juhul pole punktide arv ette teada.

Nupukatele:
- Kujund loetakse failist nagu eelmisel korral. Kasutaja saab hiirega määrata, ümber millise punkti, ning kerimisribaga määrata, kui suure nurga jagu kujundit keeratakse.

Pildioperatsioonid

Raster, RGB, baidid, filter, joonistuskiirus

Pildifaili loomine

Soovides joonistatud pildi andmeid talletada või mujale üle kanda, tuleb need paigutada edasiseks lugemiseks arusaadavale kujule. Loodud Image-tüüpi pildist võib andmed PixelGrabberi abil küsida täisarvumassiivi, kus iga element vastab ühele punktile pildil ning neid arve edaspidi talletada või töödelda ning hiljem MemoryImageSource abil uuesti pildiks muundada. Tahtes aga loodud kujutist mõne teise programmi abil edaspidi kasutada, tuleb see salvestada üldtunnustatud formaati. Õnnetuseks aga veel JDK1.3 puhul standardvahenditesse sisse ehitatud kujul üldlevinud failiformaatidesse salvestamist polnud, seega tuli kasutada muid teid. Lisavahendid Java Media Framework ning Java Advanced Imaging võimaldavad mitmeid lisaoperatsioone piltidega, ka salvestamist. Salvestamise tarvis on loonud koodilõike mitmed firmad ja programmeerijad, küllalt levinud on ACME GIF-ide salvestamise vahend. Sun-i Javaga kaasa tulev pakett com.sun.image.codec.jpeg võimaldab pildi paari käsuga salvestada JPEG formaati.

```
JPEGCodec.createJPEGEncoder(
      new FileOutputStream("pilt1.jpeg")
    ).encode(pilt);
```

loob väljundvoo faili nimega pilt1.jpeg ning saadab sinna muutujas pilt oleva pildi. Nõnda tekib kettale eespool loodud pilt. Kuna voogude sihtpunkte saame vabalt valida, siis võib samade vahendite abil pildi ka teise masinasse vaatamiseks saata kasutades küllalt hästi optimeeritud formaati.

```
import com.sun.image.codec.jpeg.*;
                     //kuulub SUNi JDK-sse
import java.awt.image.*;
import java.awt.*;
import java.io.*;

public class JPEGKodeerija1{
  public static void main(String argumendid[])
                              throws IOException{
    BufferedImage pilt = new BufferedImage(50, 50,
                         BufferedImage.TYPE_INT_RGB);
    Graphics piltg=pilt.createGraphics();
    piltg.setColor(Color.green);
    piltg.drawOval(5, 5, 40, 40);
    JPEGCodec.createJPEGEncoder(
      new FileOutputStream("pilt1.jpeg")
    ).encode(pilt);
  }
}
```

Alates Java versioonist 1.4 saab kasutada standardpaketti javax.imageio, kuju on koondatud korralik kogumik klasse ja käsklusi piltide lugemiseks ja kirjutamiseks. Järgnevalt näide ImageIO klassi abil pildifaili loomisest.

```
import javax.imageio.*;
import java.awt.image.*;
import java.awt.*;
import java.io.*;

public class JPEGKodeerija2{
  public static void main(String argumendid[])
                              throws IOException{
    BufferedImage pilt = new BufferedImage(50, 50,
                         BufferedImage.TYPE_INT_RGB);
    Graphics piltg=pilt.createGraphics();
    piltg.setColor(Color.green);
    piltg.drawOval(5, 5, 40, 40);
    ImageIO.write(pilt, "jpg", new File("pilt2.jpeg"));
  }
}
```

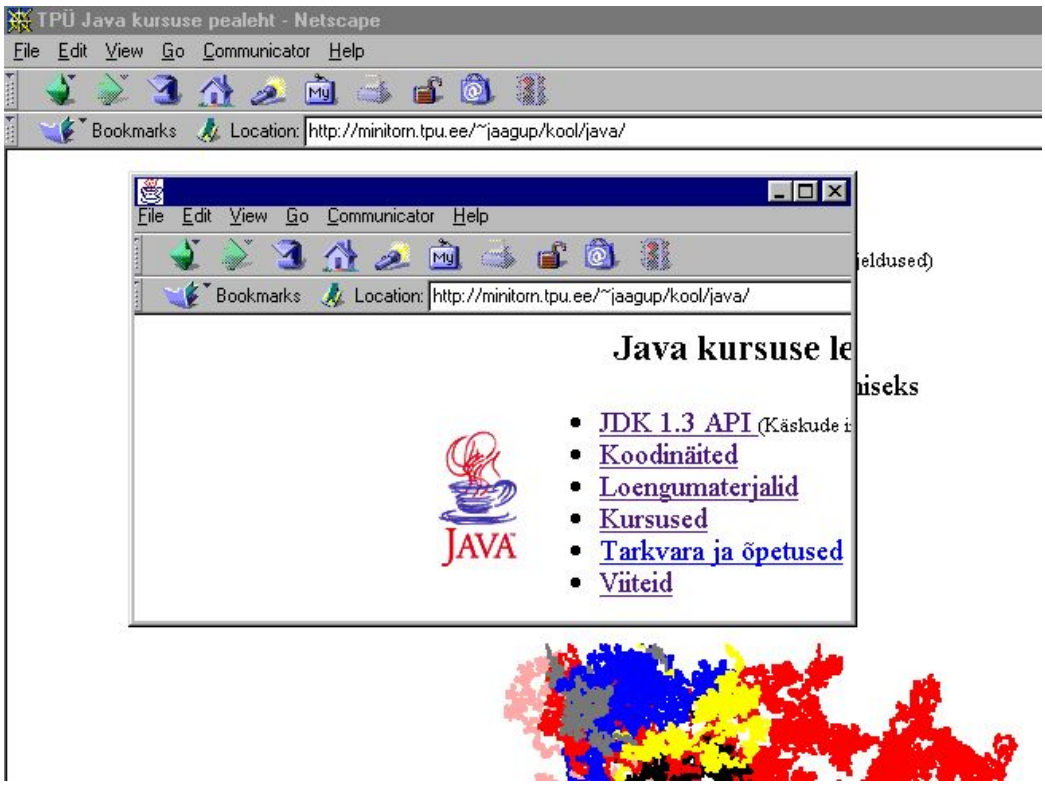
Ekraanipildi kopeerimine

Ekraanipilti kopeerida ning klaviatuuri ja hiire teateid luua aitab klassi java.awt.Robot, seda vaid juhul kui programmi vastav tegevus on lubatud. Näiteks rakendid ei tohi turvanõudeid arvestades vastavate ettevõtmistega tegelda. Ekraanipildi saab tavaliseks pildiks

```
pilt=r.createScreenCapture(
            new Rectangle(0, 0, 400, 300));
```

abil. Mis programm sellega edaspidi peale hakkab on juba programmeerija otsustada. Siin näidatakse salvestis lihtsalt ekraanile, kuid selle saab ka faili kirjutada või hoopis pildi pealt kujundeid otsides kasutaja tegevust analüüsima hakata.

```
import java.awt.*;
public class Robot2 extends Frame{
  Image pilt;
  public Robot2(){
   try{
    Robot r=new Robot();
    pilt=r.createScreenCapture(
            new Rectangle(0, 0, 400, 300));
    setSize(300, 200);
    setLocation(200, 100);
    setVisible(true);
   }catch(Exception e){}
  }
  public void paint(Graphics g){
    g.drawImage(pilt, 0, 0, this);
  }
  public static void main(String argumendid[]){
    new Robot2();
  }
}
```



Pildi muutmine

Paketi java.awt.image mitmed operatsioonid lubavad olemasoleva pildi väljanägemist muuta. Järgnevates näidetes luuakse BufferedImage ning sellele joonistatakse ringe. Edasi koostatakse esimesest pildist operatsiooni teel teine pilt ning näidatakse need kõrvuti ekraanile

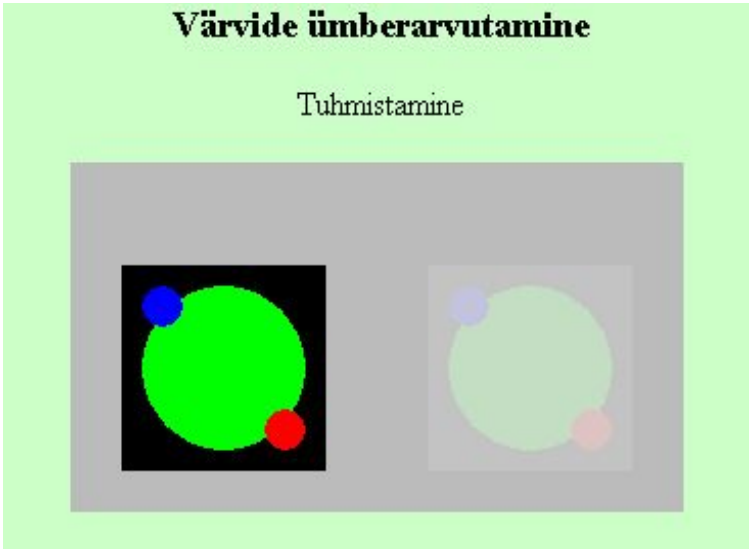
Värvide tugevus

RescaleOp võimaldab muuta pildi värve nii kõiki üheskoos kui punast, rohelist ning sinist eraldi. Siin näites luuakse

```
RescaleOp rop = new RescaleOp(0.1f, 200.0f, null);
```

, mis kõikjal jätab algsetest värvidest alles vaid 0,1 ehk 10% ning igale poole kuhu võimalik lisab iga värvi väärtusele 200. Niisuguse toimimise puhul saadakse tulemuseks valkjastuhm pilt, kuna esialgsetest väärtustest on alles vaid tühine osa ning pildi kõikide punktide kõik värvid on peaaegu maksimumväärtuse (255) lähedal. Kolmas parameeter on jäetud tühjaks (null), selle kaudu saaks soovi korral värvimuutjale edasi anda viimistlusvihjeid (RenderingHints) muutmise ressursinõuldlikkuse ja kvaliteedi kohta.

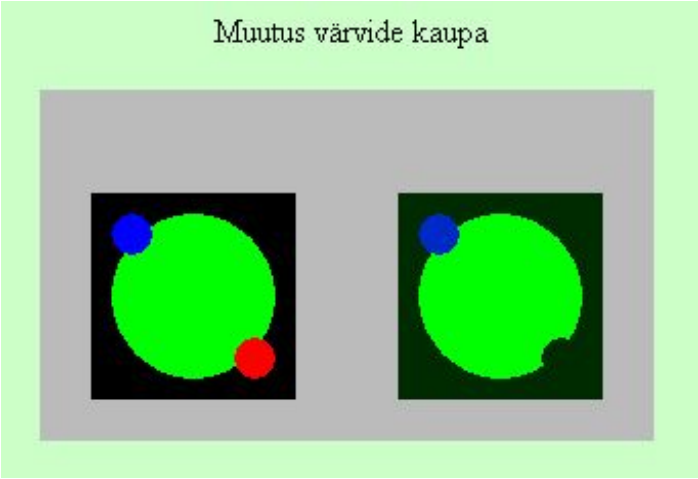
```
import java.awt.image.*;
import java.awt.geom.*;
import java.awt.*;
import java.applet.*;
public class Pildiskaleerimine1 extends Applet{
  BufferedImage pilt1=new BufferedImage(100, 100,
                      BufferedImage.TYPE_INT_RGB);
  BufferedImage pilt2=new BufferedImage(100, 100,
                      BufferedImage.TYPE_INT_RGB);
  public Pildiskaleerimine1(){
   Graphics g=pilt1.createGraphics();
   g.setColor(Color.green);
   g.fillOval(10, 10, 80, 80);
   g.setColor(Color.blue);
   g.fillOval(10, 10, 20, 20);
   g.setColor(Color.red);
   g.fillOval(70, 70, 20, 20);
   RescaleOp rop = new RescaleOp(0.1f, 200.0f, null);
                                 //väärtus*0.1 + 200
   rop.filter(pilt1,pilt2);
  }
  public void paint(Graphics g){
    g.drawImage(pilt1, 25, 50, this);
    g.drawImage(pilt2, 175, 50, this);
  }

  public static void main(String argumendid[]){
    Frame f=new Frame("Pildi värvide muutmine");
    f.add(new Pildiskaleerimine1());
    f.setSize(300, 200);
    f.setVisible(true);
  }
}
```



Väärtusi võib anda ka igale värvile eraldi. Kui ennist oli RescaleOp'i konstruktori parameetriteks kaks arvu ning null, siis siin on värvide muutmiseks ette antud kaks kolmeelemendilist massiivi ning viimistlusvihjete kohale endiselt null. Esimese massiivi iga element tähistab vasstavale värvile omistatud kordajat, teise massiiv

liikmed näitavad, palju igale värvile tugevust juurde liita. Kui mõne värvi kordajaks on 0, siis seda uuele pildile ei jõuagi. Kordaja 1 puhul jääb endine väärtus alles ning vahepealse väärtuse puhul jääb endisest alles osa.

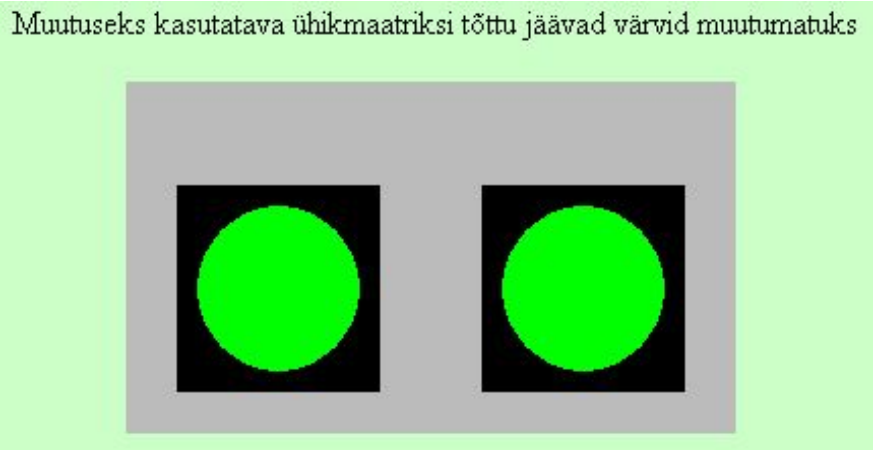
```
RescaleOp rop = new RescaleOp(
     new float[]{0, 1, 0.8f},
     new float[]{0, 50, 0}, null
);
//punane kaob, roheline jääb alles,
//sinisest 80%. Rohelisele lisatakse 50 ühikut.
rop.filter(pilt1,pilt2);
```



Värviarvutus maatriksiga

Kui punkti uus värv ei sõltu mitte ainult sama värvi tugevusest eelmisel pildil, vaid tahetakse uue punkti rohelise tugevuse arvutamisel arvestada ka eelmise punkti sinise väärtust, siis aitab BandCombineOp ning sõltuvused tuleb ette anda kahemõõtmelise massivi ehk maatriksina. Iga rea peal on kirjas, palju punkti vastava uue värvi arvutamisel tuleb arvestada algse punkti punast, rohelist ning sinist värvi.

```
float andmed[][]=new float[][]{
     {1,  0,   0},
     {0,  1,   0},
     {0,  0,   1}
};
// uus roheline = 0*vana punane+
// 1* vana roheline + 0*vana sinine
BandCombineOp bco=new BandCombineOp(
      andmed, null
);
bco.filter(pilt1.getData(), pilt2.getRaster());
```

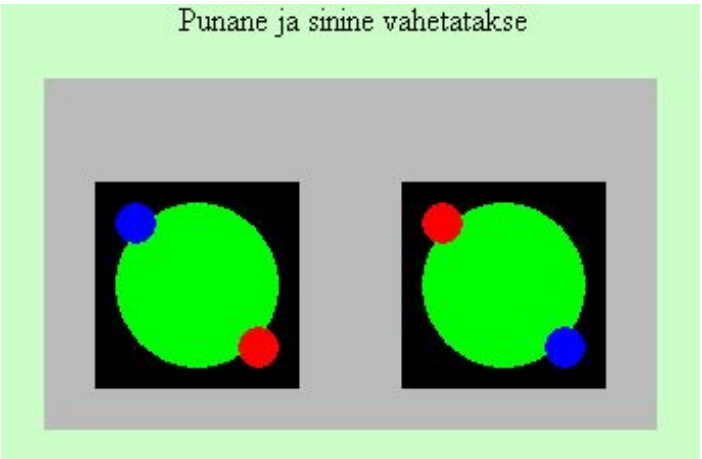


BandCombineOp'ile ei piisa piltide eneste etteandmisest. Filtrisse andmete lugemiseks tuleb sellele ette anda pildipunktide raster, mille väljastab BufferedImage meetod getData. Sihtpildist on tarvis ette anda WritableRaster ehk isend, kus pilt oma andmeid hoiab ning mille elementide muutmisel ka pildi punktid omale uue värvi saavad.

Põhivärvide vahetamine

Uue punkti punase osa arvutamisel võib endise punase sootuks arvestamata jätta ning võtta väärtus näiteks algse punkti sinise oma. Nii õnnestub värve ühendada või vahetada.

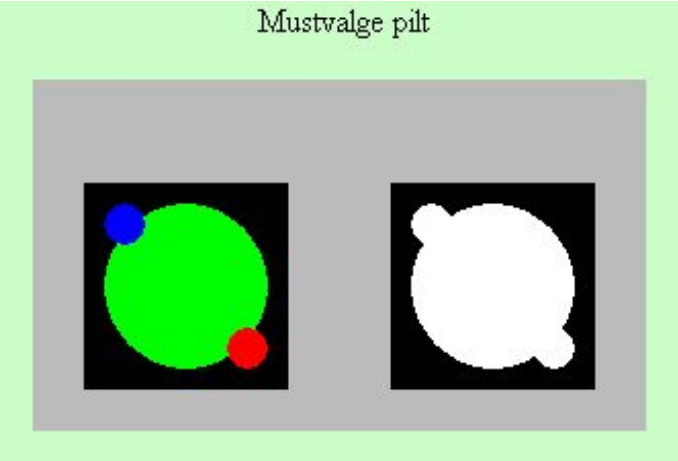
```
float andmed[][]=new float[][]{
     {0,  0,   1},   //punane ja sinine vahetatakse
     {0,  1,   0},
     {1,  0,   0}
};
```



Pilt mustvalgeks

Kui igale poole, kus ennist mingigi värv leidus panna juurde ka kõik teised värvid, siis on tulemuseks mustvalge pilt, sest värvide puudumine on must ning kõikide värvide kompott valge. Kui mõnes kohas oli ennist värve vaid osaliselt, siis nüüd oleks tulemuseks halltoon.

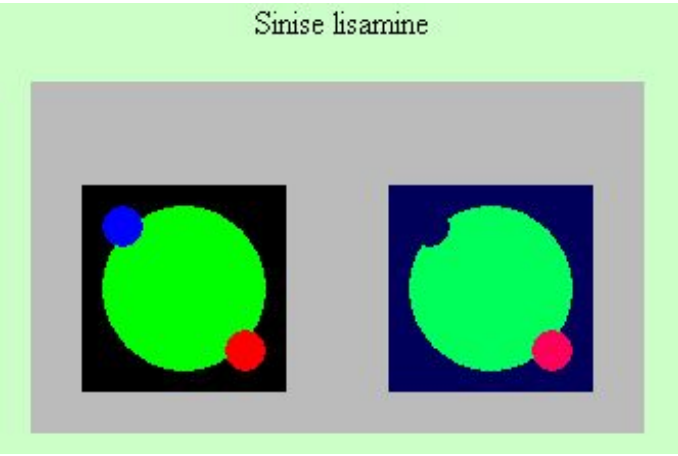
```
float andmed[][]=new float[][]{
     {1, 1, 1},
     {1, 1, 1},
     {1, 1, 1}
};
```



Põhivärvi lisamine

Tahtes üle kogu pildi põhivärvile väärtust lisada, võib massiivile juurde luua neljanda veeru, kus öeldakse, palju seda värvi juurde pannakse.

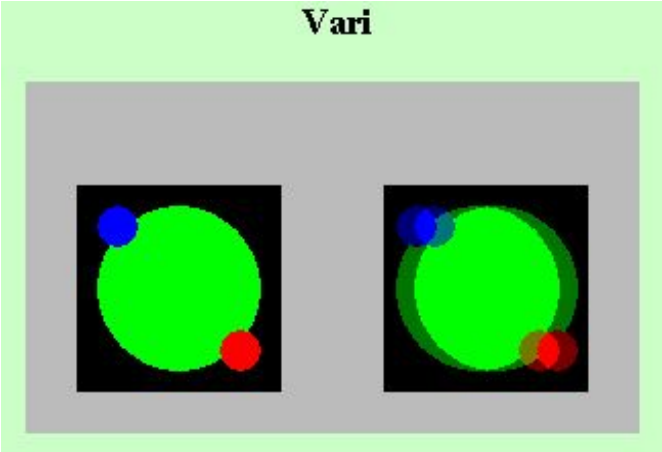
```
float andmed[][]=new float[][]{
     {1,  0,   0,  0},
     {0,  1,   0,  0},
     {0,  0,   0,  100}
};
//kõikjale lisatakse 100 ühiku ulatuses sinist (max 255)
```



Varju loomine

ConvolveOp võtab uue pildi loomisel arvesse vana pildi vastava punkti naaberpunktide väärtused. Siin antakse operatsioonile ette kümneelemendiline massiiv, mille esimene ning viimane element on väärtusega 0,5 , keskmised aga tühjad. Nii segatakse uue pildi loomisel kokku algsest punktist viis kohta vasakul ning viis kohta paremal asunud punkti värvid ningu tulemusena saadakse eelmine pilt nõnda, nagu oleks seda pildistamise ajal väristatud. Massiivi elementidega mängides võib ka varju tugevamaks või nõrgemaks muuta, arvestada ka vahepealsete punktide väärtusi või sootuks piirduda vaid ühel pool kaugel asuva punktiga. Siis tunduks, nagu oleks uus pilt tervikuna vanaga võrreldes etteantud suunas liikunud.

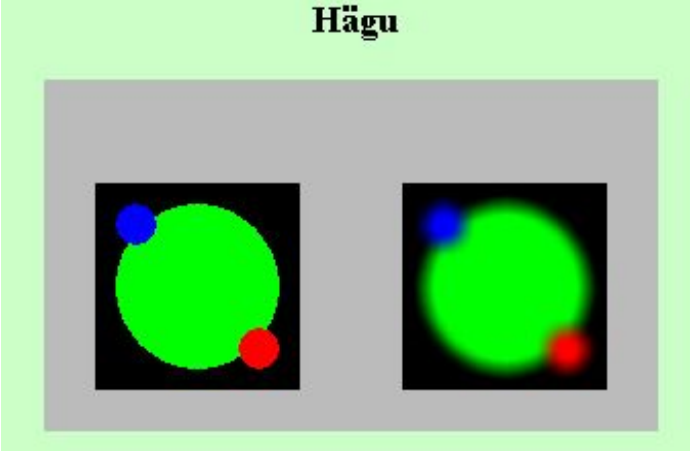
```
float andmed[]=new float[]
       {0.5f, 0, 0, 0, 0, 0, 0, 0, 0, 0.5f};
ConvolveOp co=new ConvolveOp(new Kernel(10, 1, andmed));
                             //10 veergu(x), 1 rida(y)
co.filter(pilt1, pilt2);
```



Piirjoonte hägustamine

Kui ConvolveOp'ile anda ette kahemõõtmeline massiiv, siis arvestatakse uue punkti arvutamisel endisest nii vasakul, paremal, üleval kui all asuvaid punkte. Kui võtta uue punkti aluseks ümbritsevate 10*10 punktide keskmised väärtused nagu siin, siis on tulemuseks udustatud piirjoontega pilt, sest kui terava piiri juures arvutatakse uuel pildil punkti leidmiseks mitme ümbritseva punkti keskmine, siis ei saa ju kõrvutiasetsevate punktide värvid enam nii tugevalt eristuda ning tulemuseks ongi pehme üleminek.

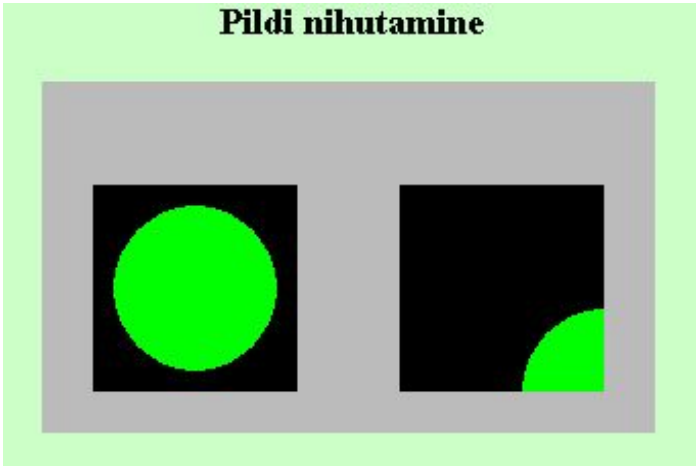
```
float andmed[]=new float[100];
for(int i=0; i<100;i++)andmed[i]=0.01f;
ConvolveOp co=new ConvolveOp(
                              new Kernel(10, 10, andmed));
```



Nihutamine

Lihtne nihutusoperatsioon sarnaneb Graphics2D poolt pakutavaga. AffineTransformOp'ile saab anda ette AffineTransformi kõikide oma võimaluste ja nende kombinatsioonidega. Võib algset pilti nihutada, suurendada, keerata ning kiiva/kaldu ajada.

```
AffineTransformOp ato=new AffineTransformOp(
      AffineTransform.getTranslateInstance(50, 50),
      AffineTransformOp.TYPE_BILINEAR
);
ato.filter(pilt1, pilt2);
```



Pildi koostamine

Kui harilikud joonistusvahendid tunduvad aeglaseks või kohmakaks jääma, siis on võimalik pilt ise üksikute punktide kaupa täisarvumassiivis välja arvutada ning siis ühe käsuga joonistatavaks pildiks muuta. All näites koostatakse värviüleminekuga pilt, kus ühtlasel õrnal rohelisel foonil suureneb kõrguse kasvades punase osa ning laiuse kasvades sinise osa. Andmete paigutamiseks tuleb luua nii pikk täisarvumassiiv, kui palju on pildi peal punkte kokku. Siin näites on kõrguseks ja laiuseks 200 punkti, nii et massiivi pikkuseks tuleb 200*200 ehk 40000. Arvestades, et iga täisarv võtab neli baiti, on tegemist 160 kilobaidiga ehk täiesti arvestatava mälumahuga. Pildipunktidele vastavad massiivi elemendid nõnda, et kõigepealt on esimese rea punktid, edasi teise rea omad jne. Iga täisarv kannab eneses pildipunkti nelja omadust Alpha-Red-Green-Blue värvimudeli järgi. Esimene määrab paistvuse, ehk kui palju koostatud värvist üldse näha on. Ülejäänud kolm osa tähistavad vastavalt iga värvi tugevust ning nagu arvutimaailmas tavaks, segatakse muud värvid nende kolme pealt kokku, sest ka inimsilm pole võimeline palju enam värve nägema kui nendest kolmest kokku segada annab. Igale komponendile eraldatakse neljabaidilisest int'ist üks bait. See määrab, et iga suuruse vähim väärtus on 0 ning suurim 255. Täisarvu baitide ühekaupa kasutamine võib olla harjumatu, kuid see võimaldab kiirust kaotamata anda edasi lihtsalt ülekantava lihttüübi väärtusega kogu ühe punkti värvi määramiseks vajaliku teabe. Kasutataks samaks otstarbeks objektide omadusi, kuluks hulk masina jõudu sealt andmete ühest kohast teise liigutamiseks ning kätte saamiseks. Lähemal vaatamisel ei peakski baitide kaupa andmete määramine väga hirmus olema, liiatigi, kui selle tarvis on soovi korral võimalik kirjutada paarirealine alamprogramm. Siin aga püüame koodi lühiduse huvides ilma selleta hakkama saada. Kui soovin kogu loodud värvi nähtavaks saada, tuleks vasakusse baiti kirjutada 255. Otse käsklust millega täisarvu baidi peale kirjutada saaks loodud ei ole, kuna see on lühidalt lahendatav. Soovides luua täisarvu, mille vasak bait on 255 ning ülejäänud nullid, võin kõigepealt luua täisarvu, mille väärtus on 255 (st. parempoolse baidi väärtus 255, ehk kõik bitid ühed ning kõik ülejäänud baidid nullid).

```
int a=255;
```

Kui edasi soovin väärtuse edasi kanda parempoolsest (esimesest) baidist vasakpoolsesse (neljandasse), siis tuleks mul algset väärtust kolme baidi ehk 24 biti jagu vasakule nihutada.

```
int b=a<<24;
```

Nii ongi loodud int, mille vasakpoolne bait on väärtusega 255 ning ülejäänud nullid. Kui sooviksin, et rohelise tugevust tähistava baidi (paremalt teise) väärtus oleks 100, siis tuleks mul numbrit 100 ühe baidi ehk kaheksa biti jagu vasakule nihutada.

```
int c=100<<24;
```

Kui soovin, et loodavas arvus oleks ühendatud kahe arvu mittenullilise väärtusega baidid (bitid), siis võin nende väärtused ühendada operaatori | abil.

```
int d=b|c;
```

Korraga võin ühendada (liita) ka rohkemate numbrite bittide väärtusi. Kui ennist on välja arvutatud punasele, rohelisele ning sinisele vastav number, massiiv punktid tähistab loodava pildi punktide jaoks leitavaid täisarve ning nr arvutatava punkti järjekorranumbrit, siis

```
punktid[nr++] = (255<<24)|(punane << 16) | roheline << 8 | sinine;
```

annab kokku ilusti täisarvu, mille esimene bait ütleb, et värvi tuleb näidata täies ulatuses, ülejäänud aga asetavad oma kohtadele igale värvile vastava tugevuse. Meetodi lõpus väljastatakse

```
createImage(new MemoryImageSource(laius, korgus, punktid, 0, laius));
```

, mis loob etteantud punktimassiivist pildi. Esimesed kolm parameetrit peaksid nimede järgi olema arusaadavad, 0 näitab, et andmeid hakatakse lugema massiivi algusest ning viimane näitab, mitme punkti kaupa rea tarvis massiivist andmeid võetakse. Harilikult on see võrdne pildi laiusega punktides. Joonistusmeetodis paint pole muud, kui pildiloomismeetodi käest pilt küsida ning see ekraanile joonistada.

```
import java.awt.image.*;
import java.awt.*;
import java.applet.Applet;
public class Pildiloomine1 extends Applet{
   public Image looPilt(){
        int laius=200;
        int korgus=200;
        int punktid[] = new int[laius*korgus];
        int nr=0;
        for (int y = 0; y < korgus; y++){
            int punane = y;
            for (int x = 0; x < laius; x++) {
                int sinine =  x;
                int roheline = 50;
                punktid[nr++] = (255<<24)|(punane << 16) | roheline << 8 | sinine;
            }
        }
        return createImage(new MemoryImageSource(laius, korgus, punktid, 0, laius));
   }

   public void paint(Graphics g){
        Image pilt=looPilt();
        g.drawImage(pilt, 0, 0, null);
   }

  public static void main(String argumendid[]){
    Frame f=new Frame();
    f.add(new Pildiloomine1());
    f.setSize(200, 220);
    f.setVisible(true);
  }
}
```

Kui kõikidele punktidele anda ühesugune sinist värvi tähistav väärtus, siis on tulemuseks ka ühtlaselt sinine pilt. Abivahend neile, kes eelmist näidet lihtsustada tahavad.

```
public Image looPilt(){
       int[] punktid=new int[200*200];
       for(int i=0; i<punktid.length; i++){
         punktid[i]=(255<<24)+100;
       }
       return createImage(new MemoryImageSource(
                       200, 200, punktid, 0, 200));
}
```

Pildi arvutamisel saab arvestada kõiki parameetreid, mis programmeerijale ette jäävad ning kasutada on. Siin tähistavad x ning y kohta, kus kasutaja viimati hiirega ekraanile vajutanud on. Punase tugevus arvutatkse vastavalt leitava punkti kaugusest hiirevajutuskohast.

```
public Image looPilt(){
        int laius=200;
        int korgus=200;
        int punktid[] = new int[laius*korgus];
        int nr=0;
        for (int y = 0; y < korgus; y++){
            for (int x = 0; x < laius; x++) {
                int punane=(int)(255*Math.sqrt(
                    (x-hx)*(x-hx)+(y-hy)*(y-hy))/laius);
                int sinine =  100;
                punktid[nr++] =
                  (255<<24)|(punane << 16) | sinine;
            }
        }
        return createImage(new MemoryImageSource(
                      laius, korgus, punktid, 0, laius));
   }
```

Lainetus

Kui punase tugevus ei sõltu mitte lihtsalt arvutatava punkti ning hiirevajutuse kaugusest vaid kauguse siinusest, mis nähtavuse suurendamise eesmärgil mingi kordajaga läbi korrutatud, siis saab tulemuseks lainetuse, sest siinus on perioodiline funktsioon ning kauguse suurenedes lainepikkuse jagu jõutakse arvutustega jälle sama kaugele tagasi. Update on üle kirjutatud seetõttu, et vajutusejärgsel pildi taasloomisel ei käiks taustavärvi vilksatus üle vaid oleks kohe uus pilt paigas.

```
import java.awt.image.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
public class Pildiloomine3 extends Applet implements MouseListener{
   int hx=100, hy=100;
   public Pildiloomine3(){
     addMouseListener(this);
   }
   public Image looLainetus(int laius, int korgus, int kx, int ky,
                                  double lainepikkus){
        int punktid[] = new int[laius*korgus];
        int nr=0;
        for (int y = 0; y < korgus; y++){
            for (int x = 0; x < laius; x++) {
                int punane=125+(int)(125*Math.sin(kaugus(x, y, kx, ky)
*2*Math.PI/lainepikkus));
//                int punane=Math.max(255-(int)(255*Math.sqrt((x-hx)*(x-hx)+(y-hy)*(y-hy))/
(laius)), 0);
                int sinine =  200;
                punktid[nr++] = (255<<24)|(punane << 16) | sinine;
            }
        }
        return createImage(new MemoryImageSource(laius, korgus, punktid, 0, laius));
   }

   static double kaugus(double x1, double y1, double x2, double y2){
     double vahex=x2-x1;
     double vahey=y2-y1;
     return Math.sqrt(vahex*vahex+vahey*vahey);
   }

   public void paint(Graphics g){
        Image pilt=looLainetus(getSize().width, getSize().height, hx, hy, 20);
        g.drawImage(pilt, 0, 0, null);
   }

   public void update(Graphics g){
     paint(g);
   }

   public void mousePressed(MouseEvent e){
     hx=e.getX();
     hy=e.getY();
     repaint();
   }
   public void mouseReleased(MouseEvent e){}
   public void mouseClicked(MouseEvent e){}
   public void mouseEntered(MouseEvent e){}
   public void mouseExited(MouseEvent e){}

  public static void main(String argumendid[]){
    Frame f=new Frame();
    f.add(new Pildiloomine3());
    f.setSize(200, 220);
    f.setVisible(true);
  }
}
```

Liikuv lainetus

Kui joonistamise parameetrid kasutajale muuta anda, siis õnnestub lainetus tema tarvis küllalt paindlikuks teha. Parameetrite muutmisel luuakse uus pildiseeria, mis ühtejärge kordudes loob illusiooni lainete liikumisest. Kaadrite arvu vähendades võib liigutamist sujuvuse hinnaga arvutile kergemaks teha. Heledusega saab määrata punase üldist tugevust, kontrastsusega et kui palju laine harjale ning lohule vastavad väärtused erinevad. Üle 255 ega 0 ei lasta väärtustel minna, muidu ei vastaks pilt enam oodatule. Pildiarvutuslõime prioriteeti on alandatud, et eelmine joonis suudaks senikaua edasi joosta kuni uue lõigu tarvis pilte arvutatakse. Uued pildid luuakse nii kerimisribaka antavate parameetrite muutmisel kui ka hiirega kuhugi vajutamisel, mis puhul leitakse lainete suubumisele uus keskkoht. Samuti tulevad uued pildid, kui joonistuskomponendi suurust muudetakse. Nii võib kasutaja valida vastavalt oma arvuti võimsusele piisavalt väikese akna, kus joonis veel mõistliku kiirusega töötada suudab.

```
import java.awt.image.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
public class Pildiloomine5 extends Applet implements MouseListener,
          Runnable, AdjustmentListener{
   int hx=75, hy=75;
   int pildilaius, pildikorgus;
   int pildinr=0;
   Scrollbar ootesb=new Scrollbar(Scrollbar.HORIZONTAL, 100, 10, 0, 500);
   Scrollbar lainepikkusesb=new Scrollbar(Scrollbar.HORIZONTAL, 20, 10, 1, 300);
   Scrollbar kaadriarvusb=new Scrollbar(Scrollbar.HORIZONTAL, 20, 3, 1, 50);
   Scrollbar heledusesb=new Scrollbar(Scrollbar.HORIZONTAL, 125, 10, 0, 255);
   Scrollbar kontrastsusesb=new Scrollbar(Scrollbar.HORIZONTAL, 100, 10, 0, 150);
   Canvas louend=new Canvas(){
     public void paint(Graphics g){
       joonista(g);
     }
     public void update(Graphics g){
       paint(g);
     }
   };

   Image pildid[];
   double lainepikkus=20;
   boolean veel;
   int kaadritearv=15;
   public Pildiloomine5(){
     louend.addMouseListener(this);
     new Thread(this).start();
     Panel p1=new Panel(new GridLayout(5, 2));
     p1.add(new Label("Aeglus:")); p1.add(ootesb);
     p1.add(new Label("Lainepikkus:")); p1.add(lainepikkusesb);
     p1.add(new Label("Kaadrite arv:")); p1.add(kaadriarvusb);
     p1.add(new Label("Heledus:")); p1.add(heledusesb);
     p1.add(new Label("Kontrastsus:")); p1.add(kontrastsusesb);
     setLayout(new BorderLayout());
     add(p1, BorderLayout.SOUTH);
     add(louend, BorderLayout.CENTER);
     lainepikkusesb.addAdjustmentListener(this);
     kaadriarvusb.addAdjustmentListener(this);
     heledusesb.addAdjustmentListener(this);
```

```
  kontrastsusesb.addAdjustmentListener(this);
}
public Image looLainetus(int laius, int korgus, int kx, int ky,
                                double lainepikkus, double faas){
     int punktid[] = new int[laius*korgus];
     int nr=0;
     for (int y = 0; y < korgus; y++){
         for (int x = 0; x < laius; x++) {
             int punane=heledusesb.getValue()+(int)(kontrastsusesb.getValue()*
                    Math.sin(kaugus(x, y, kx, ky)*2*Math.PI/lainepikkus+faas));
             if(punane>255)punane=255;
             if(punane<0)punane=0;
             int sinine =  heledusesb.getValue();
             punktid[nr++] = (255<<24)|(punane << 16) | sinine;
         }
     }
     return createImage(new MemoryImageSource(laius, korgus, punktid, 0, laius));
}


public Image[] looPildiseeria(int laius, int korgus, int kx, int ky,
               double lainepikkus, int kaadritearv){
   Image[] pildikaadrid=new Image[kaadritearv];
   for(int i=0; i<kaadritearv; i++){
     pildikaadrid[i]=looLainetus(
         laius, korgus, kx, ky, lainepikkus, 2*Math.PI*i/kaadritearv);
   }
   pildilaius=laius; pildikorgus=korgus;
   return pildikaadrid;
}

public void arvutaPildid(){
 new Thread(){
   public void run(){
    Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
    pildid=looPildiseeria(louend.getSize().width, louend.getSize().height, hx, hy,
      lainepikkusesb.getValue(),    kaadriarvusb.getValue());
    if(pildinr>=pildid.length)pildinr=0;
   }
 }.start();
}

void kontrolliSuurust(){
  if(pildilaius!=louend.getSize().width || pildikorgus!=louend.getSize().height){
     hx=louend.getSize().width/2;
     hy=louend.getSize().height/2;
     arvutaPildid();
  }
}

public void adjustmentValueChanged(AdjustmentEvent e){
  arvutaPildid();
}

public void run(){
 veel=true;
 while(veel){
  if(pildid!=null && ++pildinr>=pildid.length)pildinr=0;
  louend.paint(louend.getGraphics());
  try{Thread.sleep(ootesb.getValue());}catch(Exception e){}
 }
}

static double kaugus(double x1, double y1, double x2, double y2){
  double vahex=x2-x1;
  double vahey=y2-y1;
  return Math.sqrt(vahex*vahex+vahey*vahey);
}

public void joonista(Graphics g){
     if(g==null)return;
     if(pildid!=null && pildinr<pildid.length){
        g.drawImage(pildid[pildinr], 0, 0, null);
        kontrolliSuurust();
     } else {
        g.drawString("arvutatakse", 10, 50);
        arvutaPildid();
     }
}

public void mousePressed(MouseEvent e){
  hx=e.getX();
  hy=e.getY();
```

```
      arvutaPildid();
      repaint();
    }
    public void mouseReleased(MouseEvent e){}
    public void mouseClicked(MouseEvent e){}
    public void mouseEntered(MouseEvent e){}
    public void mouseExited(MouseEvent e){}

  public static void main(String argumendid[]){
    Frame f=new Frame();
    f.add(new Pildiloomine5());
    f.setSize(150, 250);
    f.setVisible(true);
  }
}
```

XOR tehe joonistamisel.

Kui 1980ndatel loodi küllalt sujuva liikumisega arvutianimatsioone, siis tuli läbi ajada tunduvalt väiksema jõudlusega arvutitega kui kakskümmend aastat hiljem. Sellegipoolest õnnestus saavutada küllaltki tõetruu elamus. Üheks võlusõnaks sellise liikumise juures olid spraidid ning XOR. Mobiilirakenduste ekraanidel on nimetatud tehnikad nüüdki esirinnas, mujal aga sageli põhjendamatult soiku jäänud. XOR tehte abil aga on praegugi vahel mugavam väikest joonist suurel pinnal liigutada kui graafikalehekülgi või mälupuhvreid kasutada.

Matemaatiline taust.

Tõeväärtuse puhul saab kasutada kaht võimalikku väärtust. Olgu nendeks siis arvud 0 ja 1, sõnad false ja true, värvid must ja valge, perfolindil terve koht / auk või veel mõni teineteist välistav väärtus. Levinumaid tehteid selliste väärtustega on neli ning igaüks sellistest tehetest annab vastuseks samuti väärtuse kahest võimalikust väärtusest. Ja ehk and ehk loogiline korrutamine annab tõese tulemuse vaid siis, kui mõlemad tehtes osalevad väärtused on tõesed. Igal muul juhul on tulemuseks väär väärtus, tähistatagu seda siis arvuga 0, sõnaga false või mõnel muul moel. Java keeles tähiseks üldjuhul kaks ampersandi &&. Või ehk or ehk loogiline liitmine annab tõese tulemuse juhul, kui vähemalt üks osapool on tõene. Java keeles tähiseks üldjuhul kaks püstkriipsu ||. Eitus ehk not eeldab vaid üht väärtust ning tehe keerab talle etteantud väärtuse vastupidiseks. Java keeles tähiseks hüüumärk !. Neljas tehe nimega "välistav või" ehk eXclusive OR (XOR) annab tõese tulemuse juhul, kui tehtes osaleva paari väärtused on erinevad sõltumata elementide järjekorrast. Tehte tähiseks Javas karree ^. Seda salapärast tehet annab joonistamisel enese tarbeks kasutada.

Mustvalge katsetus.

Alljärgnevalt paistab, et osalt üksteise peale on joonistatud kaks musta ruutu. Koht, kuhu musta värvi kahel korral joonistatud paistab valgena. Kui nüüd kõrvale võtta matemaatiline võrdlus, siis võiks tähistada, nagu valge oleks tehtes osalev 0 ja must 1 ning joonistamine oleks XOR-tehte tulemus. Esimese ruudu joonistamisel tekib ilusti must ruut, sest 0 ja 1 on erinevad ning igal pool on tulemuseks 1. Teise ruudu puhul aga on osalt aluspinnal nullid, osalt ühed. Kus ennist oli valge taust all, sinna tekib must värv. Kus aga enne olid musta värvi punktid siis teise musta värvi pealejoonistamisel on tehteks 1^1 ning tulemuseks 0 ehk valge. Siitkaudu paistavad välja XORi abil joonistamise kaks tähtsamat väljundit:

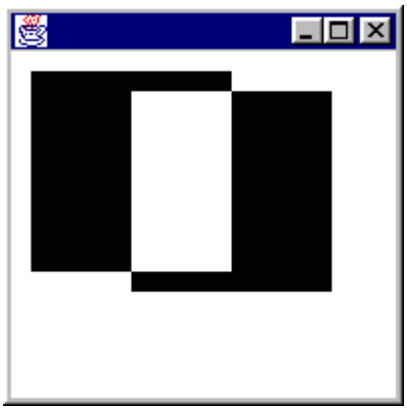
- Eelmine pilt paistab alt välja
- Teist korda kujundit samale kohale joonistades joonistatav kujund kaob ning taastub esialgne pilt.

Nende aga eeskätt viimase omaduse tõttu võimaldab selline joonistusvõte liikumise juures märgatavalt arvuti ressursse kokku hoida.

Kood näeb välja nagu iga muu tavalise joonistamisega seotud rakenduse juures. Vaid paint-meetodis on enne joonistuskäskude käivitamist antud käsuks g.setXORMode(Color.white), mis teatab, et sellest hetkest alates tuleb joonistamisel rakendada välistava või tehet. Ning et kui joonistatakse peale

sama värvi punkt kui juhtub all olema, siis olgu uue punkti värviks valge. Teist korda joonistamisel, ehk siin näites siis valge peale joonistamisel saadakse tulemuseks joonistatav värv.

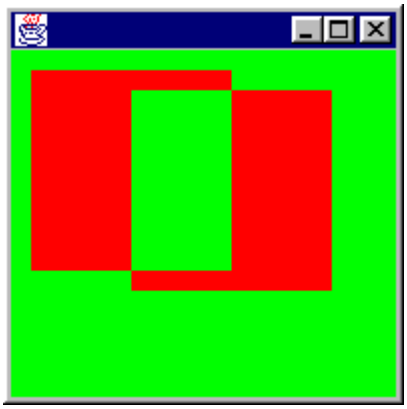
```
import java.applet.Applet;
import java.awt.*;

public class XORJoonis1 extends Applet{
  public void paint(Graphics g){
    g.setColor(Color.black);
    g.setXORMode(Color.white);
    g.fillRect(10, 10, 100, 100);
    g.fillRect(60, 20, 100, 100);
  }
  public static void main(String[] argumendid){
     Frame f=new Frame();
     f.add(new XORJoonis1());
     f.setSize(200, 200);
     f.setVisible(true);
  }
}
```



Värviline XOR

Sama lähenemine toimub ka värviliste kujundite juures. Olles kahel korral joonistanud rohelisele taustale sinise punkti, on tulemuseks taas roheline.

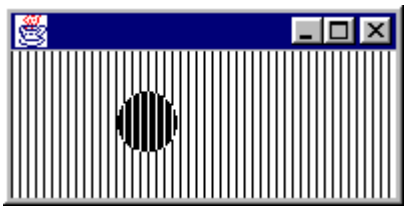
```
import java.applet.Applet;
import java.awt.*;

public class XORJoonis2 extends Applet{
  public void paint(Graphics g){
    setBackground(Color.green);
    g.setColor(Color.blue);
    g.setXORMode(Color.white);
    g.fillRect(10, 10, 100, 100);
    g.fillRect(60, 20, 100, 100);
  }
  public static void main(String[] argumendid){
     Frame f=new Frame();
     f.add(new XORJoonis2());
     f.setSize(200, 200);
     f.setVisible(true);
  }
}
```

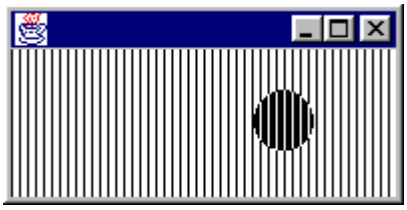


Liikumine

Enam on XORi kasu tunda liigutamise puhul. Triibulisel taustal ringi ilma vilkumata liikuma panek eeldaks vähemasti topeltpuhverdust. Nõnda aga pääseb palju väiksema joonistuspinnaga. Igal korral tuleb üle käia vaid punktid kus ring enne paiknes või kuhu ta uue sammu juures jõuab. Mõeldav oleks ka vaid ühe sammu jooksul muudetava ala puhvrisse võtmine, kuid XORi lähenemine on tuntavalt lihtsam. Eriti, kui oleks tahtmist korraga liikuma panna mitu kujundit.

```
import java.applet.Applet;
import java.awt.*;

public class XORJoonis3 extends Applet{
  public void paint(Graphics g){
    g.setColor(Color.black);
    for(int i=0; i<getSize().width; i++){
          //triibuline taust
      if(i%4==0){
        g.drawLine(i, 0, i, getSize().height);
      }
    }
    g.setXORMode(Color.white);
    for(int i=0; i<getSize().width; i++){
     g.fillOval(i, 20, 30, 30);
     try{Thread.sleep(50);} catch (Exception e){}
     g.fillOval(i, 20, 30, 30);
    }
  }
  public static void main(String[] argumendid){
     Frame f=new Frame();
     f.add(new XORJoonis3());
     f.setSize(200, 100);
     f.setVisible(true);
  }
}
```

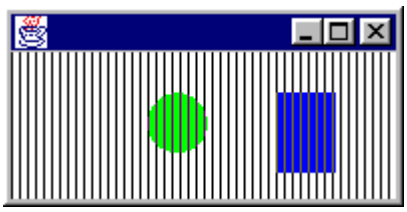


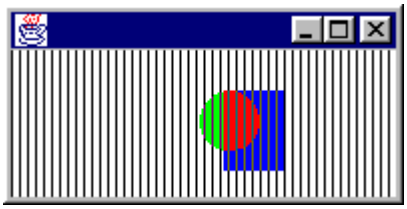


Kujundite kattumine liikumisel

Järgnevalt ongi näha mitu liikuvat kujundit. Ning taust paistab läbi nii kummastki kujundist eraldi kui üksteise peale jõudnud kujundite puhul.

```
import java.applet.Applet;
import java.awt.*;

public class XORJoonis4 extends Applet{
  public void paint(Graphics g){
    g.setColor(Color.black);
    for(int i=0; i<getSize().width; i++){
       //triibuline taust
      if(i%4==0){
        g.drawLine(i, 0, i,
                   getSize().height);
      }
    }
    g.setXORMode(Color.white);
    for(int i=0; i<getSize().width; i++){
     g.setColor(Color.green);
     g.fillOval(i, 20, 30, 30);
     g.setColor(Color.blue);
     g.fillRect(200-i, 20, 30, 40);
     try{Thread.sleep(50);}
     catch (Exception e){}
     g.setColor(Color.green);
     g.fillOval(i, 20, 30, 30);
     g.setColor(Color.blue);
     g.fillRect(200-i, 20, 30, 40);
    }
  }
  public static void main(
      String[] argumendid){
     Frame f=new Frame();
     f.add(new XORJoonis4());
     f.setSize(200, 100);
     f.setVisible(true);
  }
}
```





Liikumine omaette lõimes

Võimaluse korral pole liikumise tarvis mõeldud viivitust siiski viisakas paint-käskluse sisse kirjutada. Meetod paint on ette nähtud pildi staatiliseks kuvamiseks, viivitused selles kipuvad häirima akna suuruse muutmise sujuvust. Hoopis viisakam on paigutada joonistamine omaette lõime ning lasta sellel siis ringi joonistamiste ja kustutamiste eest hoolt kanda. Kuna joonistusalgoritmiks on XOR, siis kustutamiseks piisab samast käsust kui joonistamise puhulgi. Teistkordsel sama kujundi samasse kohta joonistamisel taastatakse algne olukord.

Lõime käivitamine on toodud meetodisse paint, sest varem ei pruugi komponendi peale veel võimalik olla joonistada. Ka run-meetodisse oleks võimalik luua kontroll, et joonistataks alles siis, kui getGraphics on tagastanud joonistamisvõimelise objekti, kuid praegune näide peaks ressursse vähem kulutama.

```
import java.applet.Applet;
import java.awt.*;

public class XORJoonis5 extends Applet implements Runnable{
  boolean algus=true;
  public void paint(Graphics g){
    g.setColor(Color.black);
    for(int i=0; i<getSize().width; i++){       //triibuline taust
      if(i%4==0){
        g.drawLine(i, 0, i, getSize().height);
      }
    }
    if(algus){
      new Thread(this).start();
      //kui paint käivitub, siis selleks ajaks on juba võimalik joonistada.
      algus=false;
    }
  }
  public void run(){
   Graphics g=getGraphics();
   g.setXORMode(Color.white);
   while(true){
    for(int i=0; i<getSize().width; i++){
     g.setColor(Color.green);
     g.fillOval(i, 20, 30, 30);
     g.setColor(Color.blue);
     g.fillRect(200-i, 20, 30, 40);
     try{Thread.sleep(50);} catch (Exception e){}
     g.setColor(Color.green);
     g.fillOval(i, 20, 30, 30);
     g.setColor(Color.blue);
     g.fillRect(200-i, 20, 30, 40);
    }
   }
  }
  public static void main(String[] argumendid){
     Frame f=new Frame();
     f.add(new XORJoonis5());
     f.setSize(200, 100);
     f.setVisible(true);
  }
}
```

Jällegi võib edaspidi siinseid näiteid oma koodis aluseks võtta ning soovi korral midagi märgatavalt ilusamat ja põhjalikumat kokku panna.

Ülesandeid

Ekraanipilt

- Kopeeritud ekraanipildi parem alumine nurgatükk näidatakse suurendatuna üle ekraani.
- Kopeeritud ekraanipildi nurgatükk liigub alaservas edasi-tagasi.
- Kopeeritud ekraanipildi serv jagatakse mõnekümne punkti pikkuse küljega ruutudeks. Need ruudud liiguvad ekraani servas ringiratast ümber ekraani.

Kopeeritud ekraanipilt

- Programmi käivitamisel kopeeritakse parasjagu nähtav ekraanipilt mällu ning näidatakse kasutajale.
- Pilt kopeeritakse mällu kogu ekraani ulatuses ning näidatakse ekraanile klassi Window abil, seega ilma välimise raamita ning peaks näima tõetruu.
- Loodud ekraanipildile hakkavad juhuslikesse kohtadesse tekkima täpikesed, kuni lõpuks on kogu pilt kirju.
- Ekraanipilt salvestatakse vähendatuna jpeg-faili.
- Väljastatakse, kas ja kus leidub ekraanil musta värvi täpp
- Ekraanipildilt otsitakse üles ja tehakse topeltklõps akende ülanurgas olevatele sulgemisristikestele.

Pildi koostamine.

- Loo MemoryImageSource abil rohekas pilt.
- Pilt on ülalt servast tumedam ning alt heledam.
- Kopeeri ekraanipilt ning keera see tagurpidi.
- Saada loodud pilt võrgu kaudu teise arvutisse vaatamiseks.

Joon pildil

- Loo MemoryImageSource abil pilt, kus mustal taustal oleks valge joon.
- Loo joon keskelt valge, servad lähevad aga sujuvalt taustaks üle.
- Pane eelmises punktis loodud joon horisontaalsuunas liikuma.

Värviüleminek

- Loo MemoryImageSource abil värviüleminek punaselt sinisele.
- Kasutaja märgib pildil kolm punkti. Ühes neist on maksimumis punane, teises roheline ning kolmandas sinine värv. Värviüleminekud pildil on sujuvad.
- Märgitud haripunktide asukohti saab kasutaja hiirega hiljem liigutada.

Muusika

Helilõigud, MIDI, saatehääl, kolmkõlad, kvanditud heli, stereo

Klippide mängimine

Esimesest versioonist peale suudavad java programmid mängida au lihtsamas formaadis faile. Versioonist 1.2 on juurde liidetud ka muude (wav, midi) helifailitüüpide mängimise võimalus. Muusika mängimiseks tuleb luua AudioClip. Selle meetoditega play, loop ning stop saab panna klipi mängima ning mängimine katkestada. Loop tähendab, et klippi mängitakse pidevalt, s.t. pärast lõpetamist hakatakse mängimisega uuesti algusest peale.

```
import java.applet.*;
public class Muusika1 extends Applet{
 public void start(){
   AudioClip lugu=getAudioClip(getCodeBase(), "spacemusic.au");
   lugu.play();
 }
}
```

Järgneva näite puhul hakatakse korduvalt mängima lehele tulles ning lehelt lahkudes lõpetatakse mängimine.

```
import java.applet.*;
public class Muusika2 extends Applet{
 AudioClip lugu;
 public void init(){
  lugu=getAudioClip(getCodeBase(), "spacemusic.au");
 }
 public void start(){
   lugu.loop();
 }
 public void stop(){
   lugu.stop();
 }
}
```

Iseseisva programmi puhul saab klipi kätte klassi `Applet` staatilise meetodi `newAudioClip` abil, meetodi parameetriks on helifaili URL.

```
import java.applet.*;
import java.net.URL;
public class Muusika3a{
  public static void main(String argumendid[]) throws Exception{
    AudioClip lugu=Applet.newAudioClip(
      new URL("http://minitorn.tpu.ee/~jaagup/kool/java/"+
               "abiinfo/tutorial/sound/example-1dot2/bottle-open.wav")
    );
    lugu.play();
    Thread.sleep(5000);
  }
}
```

Kui soovitakse mängida lugu samast masinast, siis tuleb ka see kohalik aadress enne URLiks muuta.

```
import java.applet.*;
import java.net.URL;
public class Muusika3{
  public static void main(String argumendid[]) throws Exception{
    AudioClip lugu=Applet.newAudioClip(
      new File("spacemusic.au").toURL()
    );
    lugu.play();
    Thread.sleep(5000);
  }
}
```

MIDI

Lisaks varemvalminud lugude esitamisele saab ka ise meloodiaid kokku panna. MIDI abil saame määrata, millist nooti millal mängida. Salvestatud pillide helinäidete järgi luuakse selle

tulemusena meie poolt küsitud hääl. Kui soovitakse samaaegselt kuulda mitme pilli meloodiat, tuleb need paigutada eri kanalitele. Iga kanal võib korraga teha ühe pilli häält ning harilikult on kanaleid kokku kuni 16.

Kanalist väljuvat heli saab kontrollida sinna saadetavate käskudega. Enamkasutatavad on `noteOn` ja `noteOff`. Esimene neist palub nooti mängima hakata, teine mängimise lõpetada. Meetod `noteOn` vajab kahte parameetrit: heli kõrgust ja valjust, noodi kõlamise lõpetamiseks piisab noodi numbrist. MIDI standardi järgi on igal pooltoonil oma number vahemikus 0-127. Esimese oktaavi C väärtuseks on näiteks 60, sealt saab siis vastavalt poole tooni kaupa üles ja allapoole arvutada. Valjust tähistab samuti number samast vahemikust. MIDI vahendid asuvad paketis `javax.sound`, mis kuulub standardvahendite hulka alates JDK versioonist 1.3. Operatsiooni muusikavahendite poole pöördumiseks saab kasutada klassi MidiSystem.

Üksik noot

Järgnevas näites küsitakse selle klassi kaudu helitekitamise seade ehk süntesaator ning avatakse. Viimase käest küsitakse tema külge kuuluv kanalite massiiv ning sealt omakorda kanal nr. 0. Järgnevalt palun vastaval kanalil mängida A nooti (noot nr. 69) valjusega 65. Ootan sekundi ning siis lasen noodi kõlamise lõpetada. System.exit viimase käsuna on tarvilik, kuna MIDI vahendite tarvitamisega virtuaalmasina poolt loodud lõim ei oska nootide lõpuga oma tööd lõpetada ning programm jääks viimasele reale rippuma. Analoogiline olukord on ka graafikakomponentide juures, kus programmi töö lõpetamiseks tuleb kirjutada `System.exit(0).`

```
import javax.sound.midi.*;
public class Noot{
  public static void main(String argumendid[]) throws Exception{
    Synthesizer synthesizer=MidiSystem.getSynthesizer();
    synthesizer.open();
    MidiChannel kanal=synthesizer.getChannels()[0]; //kanal 0;
    int korgus=69; //A
    int valjus=65; //keskmine
    kanal.noteOn(korgus, valjus);
    Thread.sleep(1000);
    kanal.noteOff(korgus);
    System.exit(0);
  }
}
```

Kromaatiline heliredel

Kromaatilist heliredelit võib mängida tsükli abil:

```
for(int i=40; i<120; i++){
      kanal.noteOn(i, 60);
      Thread.sleep(200);
      kanal.noteOff(i);
}
```

Kui soovida, et samal kanalil mängiks korraga mitu nooti, tuleb lihtsalt ükshaaval määrata, millised helikõrgused peavad kõlama. Kõikide mängimise saab korraga lõpetada käsuga`allNotesOff ().`

Helikõrguse ujumine

Kuigi MIDI puhul öeldakse helikõrgus numbriga ette, on ka siin võimalik toonil ujuda lasta. Seda saab käsuga `setPitchBend`, andes ette numbri, palju kõrgust muuta. Vaikimisi väärtuseks on 8192, sellisel juhul vastab noodi number tema helikõrgusele. Iga number sellest ülespoole viib helikõrgust kõrgemale, allapoole aga madalamaks. Vaikiva kokkuleppe järgi tähistab number 0 tooni võrra madalamat ning 16363 tooni jagu kõrgemat heli, kuid see kõikumise piirkond võib ka erineda. Järgnevas näites peaks tooni ülalt alla ujumine kuulda olema.

```
    kanal.noteOn(60, 70);
    kanal.noteOn(64, 70);
    for(int korgus=16383; korgus>0; korgus-=500){
      Thread.sleep(200);
      kanal.setPitchBend(korgus);
    }
```

```
    kanal.allNotesOff();
```

Pillide loetelu

Kanalil mängivat instrumenti saab muuta käsuga `programChange`, andes parameetritena ette uue pilli helipanga ning panga sees sellele pillile vastava programmijupi järjenumbri. Süntesaatorile kättesaadavad pillid saab küsida `getDefaultSoundbank().getInstruments()` abil. All näites paiknevad trükitakse tsüklis järgemööda välja pillide nimed ning mängitakse igal pillil noot.

```
Instrument[] pillid=synthesizer.getDefaultSoundbank().
                getInstruments();
    MidiChannel kanal=synthesizer.getChannels()[0];
    for(int i=0; i<pillid.length; i++){
      System.out.println(i+": "+pillid[i]);
      kanal.programChange(pillid[i].getPatch().getBank(),
                          pillid[i].getPatch().getProgram());
      kanal.noteOn(60, 50);
      Thread.sleep(500);
      kanal.noteOff(60);
    }
```

osa väljundist:

```
12: Instrument Marimba (bank 0 program 12)
13: Instrument Xylophone (bank 0 program 13)
14: Instrument Tubular Bell (bank 0 program 14)
15: Instrument Dulcimer (bank 0 program 15)
16: Instrument Hammond Organ (bank 0 program 16)
```

Rajad

Kanalitele käske andes saab edukalt mängida programmi sees üksikuid noote ja luua kõlaefekte, kuid meloodiate ja viisijuppide esitamiseks on loodud täiendavad abivahendid: sekventser, sekventsid ja rajad. Rajal (track) on kirjas saadetavad teated koos ajatemplitega. Ühele rajale võib kanda näiteks ühe pillimehe mängitavad noodid. Sekvents on radade kogumik (nagu partituur). Sekventser on programmilõik, kes sekventsi kirjutatud käsklused õigel ajal süntesaatorile heli tekitamiseks edasi saadab.

Kanalile saadetava teate hoidmiseks on `ShortMessage`. Teate sisuks võib olla nii noodi mängimise algus, selle lõpp, kanalil oleva instrumendi vahetus kui muudki, näiteks klahvivajutuse tugevuse muutus. Rajale lisamiseks tuleb teatele ümber panna MidiEvent, mis lisab sinna aja – MIDI ajaühikutes mõõdetava vahemiku alates raja algushetkest. Esimeseks teateks kanalil peab olema PROGRAM_CHANGE, selle abil määratakse, millise pilliga hakatakse vastaval kanalil häält tegema. Siin näites määratakse kanalile 0 pill number 16, ehk praeguse helipanga järgi Hammondi orel.

Sekventsi puhul tuleb määrata, milliselt seal aega arvatakse: kas kaadrite või löökide järgi. Esimest võimalust kasutatakse video kõrvale heli loomiseks, teist nootide ja taktide järgi lugu seades, viimast tähistab Sequence.PPQ nagu siin näites. Käsk `new Sequence(Sequence.PPQ, 4)` tähistab, et luuakse nootidepõhine sekvents, mille iga löök (ehk veerandnoot 2/4, 3/3 ja 4/4 taktimõõdus) jagatakse neljaks tiksuks. Tiks on vähim ajaühik MIDI mõõdustikus, seega sellise jaotuse korral on võimalik lühimaks kestuseks panna kuueteistkümnendiknoodid. Kui tahta näiteks ka kolmekümnekahendikke kasutada, siis tuleks algselt veerandnoot mitte neljaks vaid kaheksaks tiksuks jagada.

Enne rajale sündmuste lisamist tuleb rada luua sekventsi käsuga `createTrack()`. Tulemuse kuulamiseks on vaja `MidiSystem`'i käest küsida sekventser, see avada. Siis määrata, et sekventseri poolt mängitavaks sekventsiks oleks (praegu meie üherajaline) sekvents ning käsuga start panna sekventser tööle.

```
import javax.sound.midi.*;
public class Rada1{
 public static void main(String argumendid[]) throws Exception{
         ShortMessage lahti = new ShortMessage();
         ShortMessage kinni = new ShortMessage();
         ShortMessage algus = new ShortMessage();
         algus.setMessage(ShortMessage.PROGRAM_CHANGE, 0, 16, 0);
         lahti.setMessage(ShortMessage.NOTE_ON, 0, 65, 93);
         kinni.setMessage(ShortMessage.NOTE_OFF, 0, 65, 93);
         Sequence sequence=new Sequence(Sequence.PPQ, 4);
         Track track=sequence.createTrack();
```

```
          track.add(new MidiEvent(algus, 0));
          track.add(new MidiEvent(lahti, 0));
          track.add(new MidiEvent(kinni, 4));
          track.add(new MidiEvent(lahti, 8));
          track.add(new MidiEvent(kinni, 11));
          track.add(new MidiEvent(lahti, 12));
          track.add(new MidiEvent(kinni, 15));
          track.add(new MidiEvent(lahti, 16));
          track.add(new MidiEvent(kinni, 31));
          Sequencer sequencer=MidiSystem.getSequencer();
          sequencer.open();
          sequencer.setSequence(sequence);
          sequencer.start();
  }
}
```

Loodud sekvetserile saab seada mitmeid parameetreid, samuti küsida andmeid mängitava loo kohta. Kui tahan määrata, et loo mängimise kiirus on 40 lööki minutis, siis tuleb kirjutada `sequencer.setTempoInBPM(40);`
Kuna ennist olin määranud, et ühes löögis on 4 tiksu, siis eelmainitud reast järeldub, et igas minutis on 4*40 ehk 160 tiksu ning ühe tiksu pikkuseks on 60/160 sekundit. Analoogiliselt on võimalik määrata, millisest tiksust alates edasi mängitakse. Sekventseri käest saab küsida mängimise kiirust, temas sisalduva sekventsi ehk lõigu pikkust tiksudes, parasjagu mängitava tiksu numbrit ja muudki.

Kui soovin, et lõiku mitu korda järjest mängitaks, siis tuleb sekventserile panna külge kuular, mis lõputeate saabumisel paluks lõiku taas otsast mängima hakata. Piiritleja final on sekventseri ette toodud seetõttu, et teda saaks kohaliku muutujana sisemises klassis kasutada. Final ei luba muutujal vahetada isendit millele osutada (st., sellele muutujale ei saa anda uut väärtust). Muul juhul võiks juhtuda, et vahepeal pannakse sekventserile uus väärtus ning sisemise klassi sees lükatakse vale sekventser käima. Kui anonüümses sisemises klassis kasutada isendi (või klassi) tarvis kirjeldatud muutujat, siis seda probleemi ei teki.

Kordamine

Lõigu lõppu näitab kuularisse saabuv teade tüübiga nr. 47. Kui seepeale palun sekventseril uuesti otsast mängima hakata, siis tundub kasutajale, nagu lugu kordaks end järjepanu.

```
final Sequencer sequencer=MidiSystem.getSequencer();
  sequencer.open();
  sequencer.setSequence(sequence);
  sequencer.start();
  sequencer.addMetaEventListener(new MetaEventListener(){
  public void meta(MetaMessage m){
    //kui lugu läbi, siis alustatakse uuesti
    if(m.getType()==47)sequencer.start();
  }
});
```

Kui soovida, et sekventsi kordamisel erinevalt mängitaks, siis võib sinna panna mitu rada ning igal korral määrata, milliseid radasid mängitakse, milliseid mitte. Sekventseri käsu `setTrackMute` abil saab määrata, kas rada on tumm või mitte. Kui soovitakse, et ainult üks rada mängiks ning teised oleksid vait, siis tuleb see rada panna soleerima käsu `setTrackSolo` abil.

MIDI faili mahamängimine

... on lihtne, sest vastavad vahendid on küllaltki valmiskujul kättesaadavad. Kui arvutis on MIDI väljund (helikaardi kaudu) kõlaritesse või kõrvaklappidesse saadetud ning muusikafail ilusti kettal olemas, siis peaks all näha oleva nelja käsu abil olema võimalik etteantud nimega failist muusikat kuulata. Klassi MidiSystem käsk getSecuence võimaldab sekventsi lugeda failist (või mõnest muust voost). Kui avatud sekventser määrata vastavat sekventsi mängima, siis võimegi kuulata, mis faili salvestatud on.

```
import javax.sound.midi.*;
public class Midimangija1 {
  public static void main(String argumendid[]) throws Exception{
    Sequencer sekventser=MidiSystem.getSequencer();
```

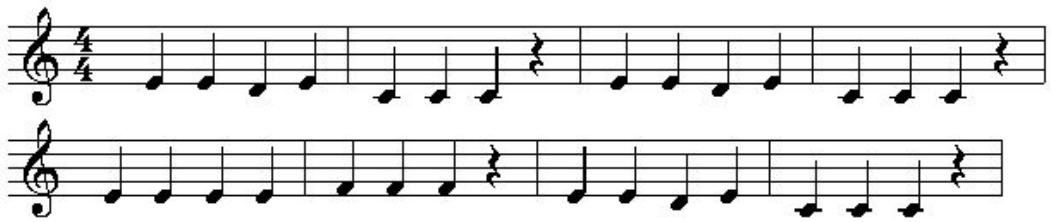
```
    sekventser.open();
    sekventser.setSequence(MidiSystem.getSequence(new java.io.File("koduke.mid")));
    sekventser.start();
  }
}
```

MIDI failis paiknevate andmete kohta võib muudki teada saada: andmete pikkust mikrosekundites ning tiksudes, radade arvu ning teated ükshaaval radade kaupa. Rajalt saab käsuga get kätte järjekorranumbri järgi MidiEvent'i. Sealt edasi getMessage väljastab teate sisu ning getTick tiksu (ajahetke), millal vastav teade süntesaatorile saadetakse. Tuleb vaadata, millist tüüpi teatega on tegemist ning vastavalt käituda. Helidega seotud teated on tüübist ShortMessage. Iga teate juures on täisarvuna kirjas kanal, käsklus ning kaks teatebaiti. Nende baitide interpretatsioon sõltub sellest, millise käsuga on tegemist.

```
import javax.sound.midi.*;
import java.util.*;
public class Midimangija2 {
  public static void main(String argumendid[]) throws Exception{
    Sequence sekvents=MidiSystem.getSequence(new java.io.File("koduke.mid"));
    System.out.println(sekvents.getDivisionType()+" pikkus: "+sekvents.getMicrosecondLength
()/1000000.0+
                    " sekundit, "+sekvents.getTickLength()+" tiksu");
    Track[] rajad=sekvents.getTracks();
    System.out.println(rajad.length+" rada");
    for(int nr=0; nr<rajad.length; nr++){
      System.out.println("Rada "+nr);
      for(int t=0; t<rajad[nr].size(); t++){
        MidiEvent me=rajad[nr].get(t);
        long tiks=me.getTick();
        MidiMessage teade=me.getMessage();
        if(teade instanceof ShortMessage){
          ShortMessage sm=(ShortMessage)teade;
          System.out.println(tiks+" ShortMessage "+
            "kanal: "+sm.getChannel()+
            " teade: "+sm.getCommand()+  //näiteks noteOff
            " bait1: "+sm.getData1()+    //kõrgus
            " bait2: "+sm.getData2()     //valjus
          );
        }
        if(teade instanceof MetaMessage){
          MetaMessage mm=(MetaMessage)teade;
          System.out.print(tiks+" MetaMessage"+
           " tyyp: "+mm.getType());
          byte[] b=mm.getData();
          for(int i=0; i<b.length; i++)System.out.print(" "+b[i]);
          System.out.println();
        }
      }
    }
  }
}

/*

Väljund:

0.0 pikkus: 15.5 sekundit, 2976 tiksu
2 rada
Rada 0
0 MetaMessage tyyp: 88 4 2 24 8
0 MetaMessage tyyp: 89 0 0
0 MetaMessage tyyp: 81 7 -95 32
2976 MetaMessage tyyp: 47
Rada 1
0 ShortMessage kanal: 0 teade: 192 bait1: 1 bait2: 0
0 ShortMessage kanal: 0 teade: 176 bait1: 7 bait2: 80
0 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 80
96 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 0
96 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 80
192 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 0
192 ShortMessage kanal: 0 teade: 144 bait1: 62 bait2: 80
288 ShortMessage kanal: 0 teade: 144 bait1: 62 bait2: 0
288 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 80
384 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 0
384 ShortMessage kanal: 0 teade: 144 bait1: 60 bait2: 80
480 ShortMessage kanal: 0 teade: 144 bait1: 60 bait2: 0
480 ShortMessage kanal: 0 teade: 144 bait1: 60 bait2: 80
576 ShortMessage kanal: 0 teade: 144 bait1: 60 bait2: 0
576 ShortMessage kanal: 0 teade: 144 bait1: 60 bait2: 80
672 ShortMessage kanal: 0 teade: 144 bait1: 60 bait2: 0
768 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 80
```

```
864 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 0
864 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 80
960 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 0
960 ShortMessage kanal: 0 teade: 144 bait1: 62 bait2: 80
1056 ShortMessage kanal: 0 teade: 144 bait1: 62 bait2: 0
1056 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 80
1152 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 0
1152 ShortMessage kanal: 0 teade: 144 bait1: 60 bait2: 80
1248 ShortMessage kanal: 0 teade: 144 bait1: 60 bait2: 0
1248 ShortMessage kanal: 0 teade: 144 bait1: 60 bait2: 80
1344 ShortMessage kanal: 0 teade: 144 bait1: 60 bait2: 0
1344 ShortMessage kanal: 0 teade: 144 bait1: 60 bait2: 80
1440 ShortMessage kanal: 0 teade: 144 bait1: 60 bait2: 0
1536 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 80
1632 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 0
1632 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 80
1728 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 0
1728 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 80
1824 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 0
1824 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 80
1920 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 0
1920 ShortMessage kanal: 0 teade: 144 bait1: 65 bait2: 80
2016 ShortMessage kanal: 0 teade: 144 bait1: 65 bait2: 0
2016 ShortMessage kanal: 0 teade: 144 bait1: 65 bait2: 80
2112 ShortMessage kanal: 0 teade: 144 bait1: 65 bait2: 0
2112 ShortMessage kanal: 0 teade: 144 bait1: 65 bait2: 80
2208 ShortMessage kanal: 0 teade: 144 bait1: 65 bait2: 0
2304 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 80
2400 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 0
2400 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 80
2496 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 0
2496 ShortMessage kanal: 0 teade: 144 bait1: 62 bait2: 80
2592 ShortMessage kanal: 0 teade: 144 bait1: 62 bait2: 0
2592 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 80
2688 ShortMessage kanal: 0 teade: 144 bait1: 64 bait2: 0
2688 ShortMessage kanal: 0 teade: 144 bait1: 60 bait2: 80
2784 ShortMessage kanal: 0 teade: 144 bait1: 60 bait2: 0
2784 ShortMessage kanal: 0 teade: 144 bait1: 60 bait2: 80
2880 ShortMessage kanal: 0 teade: 144 bait1: 60 bait2: 0
2880 ShortMessage kanal: 0 teade: 144 bait1: 60 bait2: 80
2976 ShortMessage kanal: 0 teade: 144 bait1: 60 bait2: 0
2976 MetaMessage tyyp: 47

*/
```

Nagu näha, seda MIDI faili loonud programm on kasutanud noodi kõlamise alustamiseks ning lõpetamiseks sama käsku (NOTE_ON, 144), vaid mängimise lõpetamise puhul on noodile määratud valjus 0. Kanalile 0 pole muusika kõlamisega seotud noote pandud.



MIDI faili loomine

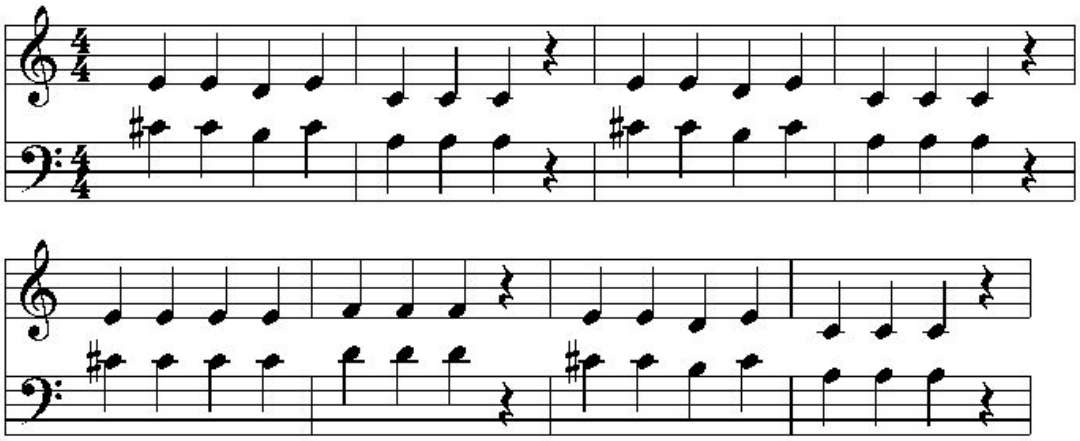
Sekventsi võib faili kirjutada ühe käsuga, andes ette MIDI formaadi (0 või 1) ning voo, kuhu andmed saata.

```
MidiSystem.write(sekvents, 1, new FileOutputStream("koduke2.mid"));
```

Nii võib rahumeeli lasta programmil loo välja mõtelda või kasutaja käest küsida ning siis kettale salvestada, et seda edaspidi kuulata või uuesti muuta. All näites on võetud ette lugu (koduke.mid), sellele lisatud rada ning algse viimase (siin ainukese) raja nootide järgi pandud uuele rajale samad noodid väikese tertsi (3 pooltooni) jagu allapoole. ShortMessage'd on ümber arvutatud, muud muutumatuna üle kantud.

```
import javax.sound.midi.*;
import java.io.*;
public class Midimangija3 {
  public static void main(String argumendid[]) throws Exception{
    Sequence sekvents=MidiSystem.getSequence(new File("koduke.mid"));
    Track algrada=sekvents.getTracks()[sekvents.getTracks().length-1]; //viimane rada
```

```
    Track uusrada=sekvents.createTrack();
    int nihe=-3;
    for(int nr=0; nr<algrada.size(); nr++){
      MidiEvent me=algrada.get(nr);
      if(me.getMessage() instanceof ShortMessage){
        ShortMessage sm=(ShortMessage)me.getMessage();
        ShortMessage sm2=new ShortMessage();
        if(sm.getCommand()==ShortMessage.NOTE_ON ||
             sm.getCommand()==ShortMessage.NOTE_OFF){
          sm2.setMessage(
            sm.getCommand(), sm.getChannel(), sm.getData1()+nihe, sm.getData2()
          );
        } else {
          sm2=(ShortMessage)sm.clone();
        }
        uusrada.add(new MidiEvent(sm2, me.getTick()));
      } else {
        uusrada.add(me);
      }
    }
    MidiSystem.write(sekvents, 1, new FileOutputStream("koduke2.mid"));
  }
}
```



Saateautomaat

Küllap olete näinud ja kuulnud süntesaatorit lugusid saatmas. Mõni koputab lihtsalt rütmi taha, teisele tuleb ette anda helistik ning selle peale mängitakse saateakordi. Leidub ka selliseid saateprogramme, mis vastavalt etteantud helistikule ise saatenoote juurde mõtlevad või lisaks sellele juba mängitava viisi pealt harmoonia ennustavad. Viimase võimaluse jätame siin välja, kuid juurde kuuluva pikema näite ja seletuse läbi töötanud lugeja peaks mõningase pusimise järel esimese kolme osaga küll hakkama saama.

Näites mängitakse saadet Juhansonide Unenäo laulule. Kasutaja saab määrata helistiku ning kuulata bassi, akorde ning taustaks mängitavat rahvalikku viiulipartiid, samuti määrata tempot. Koodi lühiduse ja kompaktsuse huvides pole viisi lisatud, kuid olles seletustest aru saanud, peaks see täiesti jõukohane olema.

Üheaegselt mängitavad rajad luuakse valmis ja pannakse kokku üheks sekventsiks. Rajad, mille heli parajasti kuulda ei soovita, pannakse vaikima. Kuna ühes helistikus mängimisel piirdub lugu kolme duuriga, siis arvutatakse nii bassi kui akordi tarvis välja rajad kõigi kolme duuri jaoks, kuid kõlada lastakse vaid siis, kui kasutaja vastavat instrumenti soovib kõlamas kuulda ning parajasti mängitav duur ja rada kokku langevad. Nii pääseb pidevast nootide arvutamisest ning uued rajad tuleb luua vaid helistiku vahetamisel. Vaid viiulipartii luuakse iga takti algul uuesti, et see tunduks värske ja kordumatuna.

Bassi partii on kõige lihtsam, sestap alustan selle loomise seletamisest. Takti jooksul on bassipartiis vaid üks noot, mängitava akordi esimene aste, mis kõlab kogu takti jooksul. Takt koosneb kolmest löögist, iga löök neljast tiksust (see määrati sekventsi loomisel). Nii peab bass alustama kõlamist tiksust 0 ning lõpetama selle viimase tiksu järel ehk kaheteistkümnenda tiksu alguses. Selle ühe noodi kõlamiseks on vaja rajale kolme teadet: tuleb määrata kanalil kõlama hakkav pill, kõlamise algus ja lõpp. Bass pannakse mängima etteantud põhinoodist 2 oktaavi ehk 24 pooltooni allpoolt.

```
void looBass(Track t, int toonika){
  try{
```

Kõigepealt tehakse tühjad teated valmis

```
   ShortMessage m[]=new ShortMessage[3];
   for(int i=0; i<m.length; i++){
     m[i]=new ShortMessage();
   }
```

seejärel antakse neile väärtused

```
   m[0].setMessage(ShortMessage.PROGRAM_CHANGE, 1, 39, 0);
   m[1].setMessage(ShortMessage.NOTE_ON, 1, toonika-24, 60);
   m[2].setMessage(ShortMessage.NOTE_OFF, 1, toonika-24, 60);
```

ning lõpuks pannakse rajale, lisades juurde, mitmenda tiksu juures nad aktiivseks peavad muutuma.

```
   t.add(new MidiEvent(m[0], 0));
   t.add(new MidiEvent(m[1], 0));
   t.add(new MidiEvent(m[2], 12));
  }catch(Exception e){e.printStackTrace();}
 }
```

Kolmkõla loomine toimub tehniliselt sarnaselt bassiga võrrelduna. Kui bassi partii koosnes takti jooksul vaid ühest noodist, siis kolmkõla tarvis kõlavad takti jooksul kolm nooti. Esimesel löögil alustatakse põhinoodiga, teisel kolmanda ning kolmandal viienda astmega. Kolmanda löögi lõpus lõpetatakse kõikide nootide kõlamine. Endise kolme teate asemel on nüüd vaja seitset: instrumendi määramiseks ning kõigi kolme noodi kõlamise alustamiseks ja lõpetamiseks.

Et võiks korraga välja arvutada ühe helistiku juures vaja minevad kolmkõla- ning bassirajad, selleks sai loodud meetod, millele antakse ette toonika ning mis väljastab sekventsi kummagi esituse radadega esimese, neljanda ning viienda astme tarvis. Kolmkõlade juures võetakse nii neljas kui viies aste (vastavalt viis ja seitse pooltooni) toonikast üles, bassi puhul alla. Tegemist on täiesti sisetundega, milliselt poolt võtta, kuid siin tundus, et helilõigu keskele sobib sügav mahlakas bass paremini ning lõpus kõrgemas toonikas justkui saabub rahu ja tasakaal. Põhjalikumate saadete puhul tuleb leidmise algoritm tõenäoliselt keerulisemaks teha, et arvestataks ilusti kõlavaid absoluutkõrgusi ning miks mitte ka viisi kulgemist.

```
Sequence pohikolmkolad(int toonika) throws InvalidMidiDataException{
   Sequence s1=new Sequence(Sequence.PPQ, 4);
   looKolmkola(s1.createTrack(), toonika);
   looKolmkola(s1.createTrack(), toonika+5); //kvart
   looKolmkola(s1.createTrack(), toonika+7); //kvint
   looBass(s1.createTrack(), toonika);
   looBass(s1.createTrack(), toonika-7); //IV ehk kvint alla
   looBass(s1.createTrack(), toonika-5);
   return s1;
 }
```

Et parasjagu valitud helistiku põhikolmkõladeks saatehääled kätte saada, ka selleks on eraldi — kuigi lühike — alamprogramm.

```
 Sequence looSekvents() throws InvalidMidiDataException{
   return pohikolmkolad(helikorgused[helistik.getSelectedIndex()]);
 }
```

Nime all helistik peitub valik ning helistik.getSelectedIndex() annab järjekorranumbri, mitmenda kasutaja oli loetelust valinud. Massiivis helistikud on tähtedega määratud, mitmendal kohal mingi helistik asetseb ning massiiv helikorgused näitab vastavate helistike toonikate asukohad MIDI skaalal. Sõnes jooksevHelistik hoitakse meeles viimati valitud helistiku nimi, et edaspidi oleks võrdlemisel teada, millal kasutaja on helistikku vahetanud, et oleks põhjust saatesekvents uuesti arvutada.

```
 JComboBox helistik=new JComboBox();
 String[] helistikud={"Bb", "F", "C", "G", "D", "A", "E"};
  int[] helikorgused={58, 53, 60, 55, 50, 57, 52};
  String jooksevHelistik="";
```

Mängitav sekvents ning mängiv sektventser,

```
  Sequence sequence;
  Sequencer sequencer;
```

graafilised vahendid töö juhtimiseks,

```
  JButton nupp=new JButton("Mängi");
  JCheckBox ruut=new JCheckBox("Korda");
  JCheckBox bass=new JCheckBox("Bass");
```

```
  JCheckBox akord=new JCheckBox("Akord");
  JCheckBox taust=new JCheckBox("Taust");
  JRadioButton[] raadionupud=new JRadioButton[3];
  String[] raadionupustring={"I", "IV", "V"};
  JScrollBar tempo=new JScrollBar(JScrollBar.HORIZONTAL, 190, 5, 40, 320);
```

rada viiulipartii paigutamiseks.

```
  Track muutuvRada;
```

Diatoonilise duuri astmetele vastavate pooltoonide arv, et hiljem oleks võimalik määrata, mitmendale astmele liikuda ning see arvutile pooltoonides selgeks teha.

```
static final int[] noodivahed={-1, 0, 2, 4, 5, 7, 9, 11, 12};
  //toonika kohal 1 väärtus 0
```

Konkreetse loo duuride järjestus.

```
  int[] duurid={0, 1, 2, 0, 1, 2, 0, 0};  // Juhansonide unenäo laul
                                          // 0- toonika, 1-IV, 2-V
  int duurinr=0; //mängitava takti järjekorranumber
```

Taustaks mängib viiul lihtsa algoritmi järgi: alustatakse kõlava duuri esimeselt astmelt ning liigutakse iga noodiga sellelt ühe võrra kas üles või alla juhuslikult võetuna. Sellisena satub rõhuline noot kokku kõlava akordiga ning taustaks kõlab ühtlane meloodiline saagimine. Veidi parandades ning ebaloogilisi järske üleminekuid vältides annaks loomulikkust suurendada, kuid ka sellisel kujul ei riiva hullusti kõrva ning viiulimängu algusest alustades tuleks tükk harjutada, et selliselegi tulemusele jõuda.

Et algoritm arvestab diatooniliste astmetega, MIDI aga loeb kõik pooltoonid samaväärseteks siis on loodud ümberarvutamiseks eraldi alamprogramm, millele antakse ette algne aste ja soovitav nihe ning mis väljastab vahe pooltoonides, lubades ka ühe oktaavi piirest välja nihkuda.

```
static int pooltoonid(int aste, int muutus){
   int loppaste=aste+muutus;
   int okt=loppaste/7;
   if(loppaste<0)okt--;
   loppaste=loppaste-7*okt;
   int vahe=noodivahed[loppaste]-noodivahed[aste];
   return vahe+12*okt;
}
```

Meetodile enesele antakse ette rada, kuhu teated panna, toonikanoodinumber ning aste, millelt mängimist alustada.

```
void looTaustaviiul(Track t, int toonika, int aste){
 try{
```

Kõigepealt luuakse mäluruum arvutustulemuste paigutamiseks soovitud arvu leitud nootide tarvis,

```
   int nootidearv=6;
   int[] samm=new int[nootidearv];
   int i=0;
   samm[i++]=0;
```

siis arvutatakse eelpool kirjeldatud algoritmi järgi väärtused nootidele, kusjuures iga järgmine asub eelmise kõrval

```
   while(i<nootidearv){
     if(Math.random()<0.5){
       samm[i]=samm[i-1]-1;
     } else {
       samm[i]=samm[i-1]+1;
     }
     i++;
   }
```

ning lõpuks asendatakse leitud astmed MIDI teadetega. Ette nagu ikka pilli määramine, sedakorda kanalile number 2.

```
   ShortMessage m=new ShortMessage();
   m.setMessage(ShortMessage.PROGRAM_CHANGE, 2, 41, 0);
   t.add(new MidiEvent(m, 0));
   for(i=0; i<nootidearv; i++){
     m=new ShortMessage();
```

Iga helikõrguse arvutamiseks liidetakse kokku helistiku põhikõrgus (toonika), pooltoonide arv kõlava duuri põhitoonini jõudmiseks (noodivahed[aste]) ning takti jooksul sellest nihkutud astmete arv pooltoonidena (pooltoonid(aste, samm[i])).

```
     m.setMessage(ShortMessage.NOTE_ON, 2,
        toonika+noodivahed[aste]+pooltoonid(aste, samm[i]), 60);
     t.add(new MidiEvent(m, i*2));
     m=new ShortMessage();
     m.setMessage(ShortMessage.NOTE_OFF, 2,
        toonika+noodivahed[aste]+pooltoonid(aste, samm[i]), 60);
     t.add(new MidiEvent(m, (i+1)*2));
   }
  }catch(Exception e){e.printStackTrace();}
 }
```

Hoolitsemaks, et mängitaks vaid nendel radadel, mille kuulamist kasutaja ootab, tuleb ülejäänud vaikima määrata. Sobivat duuri näitab raadionupp. Siin näites määrab selle valitu programm, kuid kergesti võib ka inimesel lasta määrata, mitmenda astme kolmkõla ta soovib saateks kuulata.

```
void paneSoolo(){
   for(int i=0; i<raadionupud.length; i++){
     sequencer.setTrackMute(i, !raadionupud[i].isSelected() || !akord.isSelected());
     sequencer.setTrackMute(3+i, !raadionupud[i].isSelected() || !bass.isSelected());
   }
```

Taustaviiul jäetakse kõlama vaid siis, kui vastav ruut on märgitud.

```
   sequencer.setTrackMute(6, !taust.isSelected());
 }
```

Iga uue takti mängimisel tuleb teha mõned ettevalmistused.

```
 void alusta(){
  try{
```

Kui kasutaja on helistikku vahetanud, tuleb kogu sekvents uuesti arvutada.

```
  if(!jooksevHelistik.equals(helistik.getSelectedItem())){
   sequence=looSekvents();
   jooksevHelistik=helistik.getSelectedItem().toString();
  }
```

Kui tegemist pole loo algusega, siis tuleb kustutada eelmises taktis mängitud viiulipartii

```
   if(muutuvRada!=null)sequence.deleteTrack(muutuvRada);
```

ning luua uus rada uue partii paigutamiseks.

```
   muutuvRada=sequence.createTrack();
   int pohitoon=helikorgused[helistik.getSelectedIndex()];
   int aste=1;
   if (duurid[duurinr]==1)aste=4;
   if (duurid[duurinr]==2)aste=5;
```

ja arvutada sinna noodid.

```
   looTaustaviiul(muutuvRada, pohitoon, aste);
   sequencer.setSequence(sequence);
```

Mängitava duuri raadionupp märgistada,

```
   raadionupud[duurid[duurinr]].setSelected(true);
```

leida järgmise korra jaoks järgneva takti number või suunata lugu algusse tagasi

```
   duurinr=(duurinr+1)%duurid.length;
```

ning lõpuks väljundisse lugu mängima hakata.

```
   mangi();
  }catch(Exception e){e.printStackTrace(); System.out.println(e);}
 }
```

Mängimise tarvis tuleb

```
void mangi(){
```

kõigepealt sekventser tööle panna

```
   sequencer.start();
```

alles seejärel saab määrata, millised rajad vaikivad

```
   paneSoolo();
```

ning kui kiiresti muusika liigub.

```
   sequencer.setTempoInBPM(tempo.getValue());
 }
```

Siia meetodisse saadetakse teade sekventsi lõppemise kohta. Kui kasutaja soovib jätkamist, hakatakse uuesti otsast peale, muul juhul lõpetatakse mängimine ning vabastatakse ressursid teiste programmide tarvis.

```
  public void meta(MetaMessage m){
    if(m.getType()==47 && ruut.isSelected()){
      alusta();
    } else {
      sequencer.close();
    }
  }
```

Töölesaamise käivitusprogramm on ikka standardne. Tuleb vaid hoolitseda, et init-meetod käima pandaks. Rakendina käivitamisel jääb siinne staatiline meetod sootuks täitmata. Sõltuvalt masina jõudlusest ja konfiguratsioonist võib mõnikord kohalikult kettalt iseseisva programmi või rakendina siinse saateprogrammi käivitamine sujuvama tulemuse ande kui üle võrgu tööle panek, sest sel juhul kipub vähemalt aeglasemate masinate korral õiguste kontrolliks kuuldavalt aega kuluma.

```
  public static void main(String argumendid[]){
    JFrame f=new JFrame("Saade");
```

```
    Noodirakend9a ap=new Noodirakend9a();
    ap.init();
    f.getContentPane().add(ap);
    f.pack();
    f.setVisible(true);
  }
```

Siit lehekülgedelt aga peaks olema võimalik leida piisavalt ideid ja soovitusi süntesaatori programmeerimiseks, et mõningane maitse suhu saada ning muusikateadmisi ja keerulisematel juhtudel ka raamatuid appi võttes saaks midagi täiesti kasutatavat kokku panna.

```
import javax.sound.midi.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class Noodirakend9a extends JApplet implements ActionListener, MetaEventListener{
  Sequence sequence;
  Sequencer sequencer;
  JButton nupp=new JButton("Mängi");
  JCheckBox ruut=new JCheckBox("Korda");
  JCheckBox bass=new JCheckBox("Bass");
  JCheckBox akord=new JCheckBox("Akord");
  JCheckBox taust=new JCheckBox("Taust");
  JComboBox helistik=new JComboBox();
  JRadioButton[] raadionupud=new JRadioButton[3];
  String[] raadionupustring={"I", "IV", "V"};
  JScrollBar tempo=new JScrollBar(JScrollBar.HORIZONTAL, 190, 5, 40, 320);
  Track muutuvRada;
  String[] helistikud={"Bb", "F", "C", "G", "D", "A", "E"};
  int[] helikorgused={58, 53, 60, 55, 50, 57, 52};
  String jooksevHelistik="";
  static final int[] noodivahed={-1, 0, 2, 4, 5, 7, 9, 11, 12};  //toonika kohal 1 väärtus
0
  int[] duurid={0, 1, 2, 0, 1, 2, 0, 0};  // Juhansonide unenäo laul
                                          // 0- toonika, 1-IV, 2-V
  int duurinr=0; //takti järjekorranumber

  void looKolmkola(Track t, int toonika){
  try{
   ShortMessage m[]=new ShortMessage[7];
   for(int i=0; i<m.length; i++){
     m[i]=new ShortMessage();           // Tühjad teated valmis
   }
   m[0].setMessage(ShortMessage.PROGRAM_CHANGE, 0, 25, 0);  //pill nr 25 rajale 0
   m[1].setMessage(ShortMessage.NOTE_ON, 0, toonika, 60);
   m[2].setMessage(ShortMessage.NOTE_ON, 0, toonika+4, 60); //terts rajale 0 valjusega 60
   m[3].setMessage(ShortMessage.NOTE_ON, 0, toonika+7, 60); //kvint
   m[4].setMessage(ShortMessage.NOTE_OFF, 0, toonika, 60);
   m[5].setMessage(ShortMessage.NOTE_OFF, 0, toonika+4, 60);
   m[6].setMessage(ShortMessage.NOTE_OFF, 0, toonika+7, 60);
    t.add(new MidiEvent(m[0], 0));
   for(int i=1; i<4; i++){
     t.add(new MidiEvent(m[i], (i-1)*4));  // nelja tiksu (ühe löögi) tagant
   }                                       // noodid sisse
   for(int i=4; i<7; i++){
     t.add(new MidiEvent(m[i], 12));       // 12nda tiksu juures vaikus
   }
  }catch(Exception e){
    e.printStackTrace();
  }
 }


 void looBass(Track t, int toonika){
  try{
   ShortMessage m[]=new ShortMessage[3];
   for(int i=0; i<m.length; i++){
     m[i]=new ShortMessage();
   }
   m[0].setMessage(ShortMessage.PROGRAM_CHANGE, 1, 39, 0);
   m[1].setMessage(ShortMessage.NOTE_ON, 1, toonika-24, 60);
   m[2].setMessage(ShortMessage.NOTE_OFF, 1, toonika-24, 60);
   t.add(new MidiEvent(m[0], 0));
   t.add(new MidiEvent(m[1], 0));
   t.add(new MidiEvent(m[2], 12));
  }catch(Exception e){e.printStackTrace();}
 }

 static int pooltoonid(int aste, int muutus){
    int loppaste=aste+muutus;
```

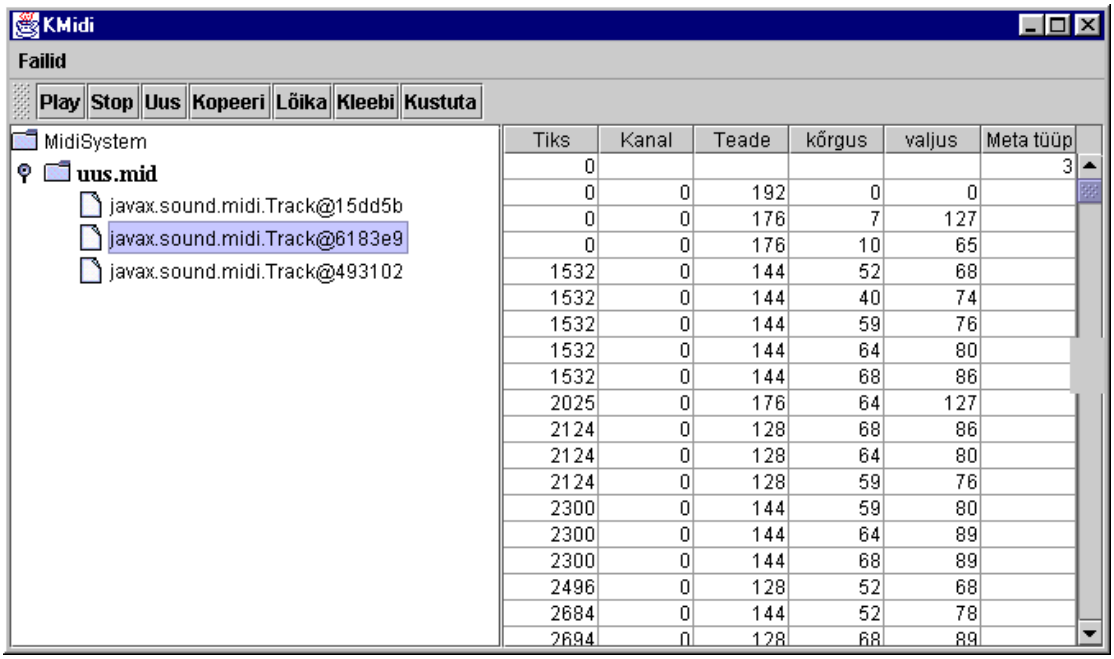
```
    int okt=loppaste/7;
    if(loppaste<0)okt--;
    loppaste=loppaste-7*okt;
    int vahe=noodivahed[loppaste]-noodivahed[aste];
    return vahe+12*okt;
  }

  void looTaustaviiul(Track t, int toonika, int aste){
   try{
    int nootidearv=6;
    int[] samm=new int[nootidearv];
    int i=0;
    samm[i++]=0;
    while(i<nootidearv){
      if(Math.random()<0.5){
        samm[i]=samm[i-1]-1;
      } else {
        samm[i]=samm[i-1]+1;
      }
      i++;
    }
    ShortMessage m=new ShortMessage();
    m.setMessage(ShortMessage.PROGRAM_CHANGE, 2, 41, 0);
    t.add(new MidiEvent(m, 0));
    for(i=0; i<nootidearv; i++){
      m=new ShortMessage();
      m.setMessage(ShortMessage.NOTE_ON, 2, toonika+noodivahed[aste]+pooltoonid(aste, samm
[i]), 60);
      t.add(new MidiEvent(m, i*2));
      m=new ShortMessage();
      m.setMessage(ShortMessage.NOTE_OFF, 2, toonika+noodivahed[aste]+pooltoonid(aste, samm
[i]), 60);
      t.add(new MidiEvent(m, (i+1)*2));
    }
   }catch(Exception e){e.printStackTrace();}
  }

  Sequence pohikolmkolad(int toonika) throws InvalidMidiDataException{
    Sequence s1=new Sequence(Sequence.PPQ, 4);
    looKolmkola(s1.createTrack(), toonika);
    looKolmkola(s1.createTrack(), toonika+5); //kvart
    looKolmkola(s1.createTrack(), toonika+7); //kvint
    looBass(s1.createTrack(), toonika);
    looBass(s1.createTrack(), toonika-7); //IV ehk kvint alla
    looBass(s1.createTrack(), toonika-5);
    return s1;
  }

  Sequence looSekvents() throws InvalidMidiDataException{
    return pohikolmkolad(helikorgused[helistik.getSelectedIndex()]);
  }

  void paneSoolo(){
    for(int i=0; i<raadionupud.length; i++){
      sequencer.setTrackMute(i, !raadionupud[i].isSelected() || !akord.isSelected());
      sequencer.setTrackMute(3+i, !raadionupud[i].isSelected() || !bass.isSelected());
    }
    sequencer.setTrackMute(6, !taust.isSelected());
  }

  void mangi(){
    sequencer.start();
    paneSoolo();
    sequencer.setTempoInBPM(tempo.getValue());
  }

  void alusta(){
   try{
   if(!jooksevHelistik.equals(helistik.getSelectedItem())){
    sequence=looSekvents();
    jooksevHelistik=helistik.getSelectedItem().toString();
   }
    if(muutuvRada!=null)sequence.deleteTrack(muutuvRada);
    muutuvRada=sequence.createTrack();
    int pohitoon=helikorgused[helistik.getSelectedIndex()];
    int aste=1;
    if (duurid[duurinr]==1)aste=4;
    if (duurid[duurinr]==2)aste=5;
    looTaustaviiul(muutuvRada, pohitoon, aste);
    sequencer.setSequence(sequence);
    raadionupud[duurid[duurinr]].setSelected(true);
    duurinr=(duurinr+1)%duurid.length;
    mangi();
```

```
  }catch(Exception e){e.printStackTrace(); System.out.println(e);}
 }

 public void init(){
      Panel nupupaneel=new Panel(new GridLayout(1, 4));
      helistik=new JComboBox(helistikud);
      helistik.setSelectedIndex(3);
      nupupaneel.add(helistik);
      ButtonGroup nupugrupp=new ButtonGroup();
      for(int i=0; i<raadionupud.length; i++){
        raadionupud[i]=new JRadioButton(raadionupustring[i]);
        nupugrupp.add(raadionupud[i]);
        nupupaneel.add(raadionupud[i]);
      }
      raadionupud[0].setSelected(true);
      ruut.setSelected(true);
      Panel mangupaneel=new Panel(new GridLayout(1, 2));
      mangupaneel.add(nupp);
      mangupaneel.add(ruut);
      JPanel tempopaneel=new JPanel(new BorderLayout());
      tempopaneel.add(new JLabel("Tempo"), BorderLayout.WEST);
      tempopaneel.add(tempo, BorderLayout.CENTER);
      JPanel valikupaneel=new JPanel(new GridLayout(1, 3));
      valikupaneel.add(bass);
      valikupaneel.add(akord);
      valikupaneel.add(taust);
      akord.setSelected(true);
      JPanel alumine=new JPanel(new GridLayout(3, 1));
      alumine.add(tempopaneel);
      alumine.add(mangupaneel);
      alumine.add(valikupaneel);
      getContentPane().add(alumine, BorderLayout.SOUTH);
      getContentPane().add(nupupaneel, BorderLayout.NORTH);
      nupp.addActionListener(this);
 }


  public void actionPerformed(ActionEvent e){
   try{
          sequencer=MidiSystem.getSequencer();
          sequencer.open();
          sequencer.addMetaEventListener(this);
          jooksevHelistik="";
          alusta();
   }catch(Exception ex){
     ex.printStackTrace();
     System.out.println(ex);
   }
  }

  public void meta(MetaMessage m){
    if(m.getType()==47 && ruut.isSelected()){
      alusta();
    } else {
      sequencer.close();
    }
  }
  public static void main(String argumendid[]){
    JFrame f=new JFrame("Saade");
    Noodirakend9a ap=new Noodirakend9a();
    ap.init();
    f.getContentPane().add(ap);
    f.pack();
    f.setVisible(true);
  }


}
```

MIDI redaktor

Järgnevalt näide, kus ühendatud Java MIDI-vahendid failide lugemiseks, mängimiseks, muutmiseks ja salvestamiseks ning Swingi võimalused kasutajaliidese loomisel. Näite on koostanud TPÜ üliõpilane Kaur Männiko graafika ja muusika programmeerimise kursuse kodutööna. Näide võimaldab avada ja luua MIDI sekventse. Vaadata radu ning neid kopeerida nii sekventsi sees kui sekventside vahel. Raja sees saab vaadata ja muuta kõiki teateid. Olles näite korralkult läbi mõelnud, peaks olema API spetsifikatsiooni järgi võimalik luua enamikke MIDI-rakendusi, mille jaoks ideid ja vajadusi on.

Graafilisest liidesest võtavad suurema osa enese alla Swingi puu ning tabel. Esimene avatud failide/sekventside ning nende sees olevate radade loeteluks. Tabelis on näha märgistatud rajal asuvad teated toimumise järjekorras koos oma tähtsamate andmetega.



Tiks	Kanal	Teade	kõrgus	valjus	Meta tüüp
0					3
0	0	192	0	0	
0	0	176	7	127	
0	0	176	10	65	
1532	0	144	52	68	
1532	0	144	40	74	
1532	0	144	59	76	
1532	0	144	64	80	
1532	0	144	68	86	
2025	0	176	64	127	
2124	0	128	68	86	
2124	0	128	64	80	
2124	0	128	59	76	
2300	0	144	59	80	
2300	0	144	64	89	
2300	0	144	68	89	
2496	0	128	52	68	
2684	0	144	52	78	
2694	0	128	68	89	

Järgnevalt koodis leiduvate tähelepanu nõudvate lõikude kirjeldus järjemööda.

Märgitud piirkonna mahamängimiseks on loodud eraldi alamprogramm. Puu käest küsitakse kasutaja märgitud komponent ning asutakse käituma vastavalt sellele, mida kasutaja märkinud oli. Mängimiseks sobivad sekvents ning rada. Esimesel puhul paikneb puu viimase lehe ehk node sees NodeObjectHolder, milles omakorda sekvents. Sekventsi saab omaette tervikuna mängima panna.

```
    public void play() {
        try {
            Object src = tree.getLastSelectedPathComponent();
...
            DefaultMutableTreeNode node = (DefaultMutableTreeNode) src;
            if ( node.getUserObject() instanceof NodeObjectHolder) {
                sekvents = (Sequence)
                  ((NodeObjectHolder)node.getUserObject()).userObject;

            } else if( node.getUserObject() instanceof Track) {
```

Raja mängitamiseks tuleb luua uus sekvents selle sekventsi parameetritega, milles mängitav rada on ning siis rajas olevad teated uue sekventsi loodavale rajale üle kanda.

```
    Track algrada = (Track)node.getUserObject();
    sekvents = (Sequence) ((NodeObjectHolder)
       ((DefaultMutableTreeNode)node.getParent()).getUserObject() ).userObject;
    sekvents = new Sequence(sekvents.getDivisionType(), sekvents.getResolution());
    Track uusrada = sekvents.createTrack();

    for (int nr = 0; nr < algrada.size(); nr++)
      uusrada.add(algrada.get(nr));
```

Sekventsi käivitamiseks piisab kahest käsust.

```
            sekventser.setSequence(sekvents);
            sekventser.start();
```

Klassil küljes olev TreeSelectionListener näitab, et KMidi eksemplar peab oskama ümber käia puu küljest tulevate teadetega. Kui puus valitakse uus koht, saabub selle kohta teade valueChanged käsu kaudu. Edasi kontrollitakse, et juhul kui valituks osutus rada ehk Track-tüüpi leht, siis luuakse uus

TableModel ning selle kaudu palutakse paremal pool asuval tabelil näidata just valitud raja sees olevaid teateid.

Nii veerunimede kui raja jaoks loodud muutujad on piiritlejaga final, et sisemise klassi loomisel võiks sealne kood kindel olla, et muutuja külge seotud objekt ikka sinna püsima jääb ning keegi seda ära vahetada ei saa. Abstraktsest tabelimudelist tabeli jaoks tarviliku objekti kokkupanekul tuleb üle katta peotäis käsklusi, mis annavad teavet tabeli sisalduse kohta või lubavad ka seda muuta. Niisuguse mudeli kaudu ei pruugi sugugi alati kõik näidatavad väärtused kahemõõtmelises massiivis asuma, vaid võidakse võtta kusagilt mujalt või sootuks käigu peal valmis arvutada. Siin näites hoitakse andmeid raja sees ning tabel on vaid sealsete väärtuste peegliks.

Tabeli mudel on loodud KMidi sees sisemise anonüümse klassina. Paistab new AbstractTableModel(){} ning nende loogeliste sulgude vahele on paigutatud kogu ülekaetavate käskude loetelu koos sisuga. Nagu käskude nimedestki näha getColumnName annab veeru järjekorranumbrile vastava pealkirja. getColumnCount ja getRowCount teatavad, palju peaks sellele mudelile vastavas tabelise ridasid ja veerge olema. Põhjalikumad käsklused on getValueAt ning setValueAt.
Nende abil saab küsida või määrata tabeli konkreetses lahtris asuvat väärtust.

Sõltumata küsitavast veerust küsitakse rajast kõigepealt reanumbrile vastav sündmus ning asutakse selle sisu uurima. Tiksu väärtus leidub sõltumata sündmuse tüübist ning see küsitakse eraldi muutujasse. Edasi toimetatakse vastavalt sündmuse tüübile. Üldjuhul muusikalise teate ehk ShortMessage puhul leitakse kanali ja käskluse koodid ning mõlemad andmebaidid. MetaMessage puhul antakse vaid tiks või tüübi kood (nt. raja lõpp 47).

Andmete seadmisel setValueAt puhul antakse ette uus väärtus ning rida ja veerg, kuhu soovitakse see väärtus paigutada. Rea järgi otsitakse üles vastav MIDI tiks, veeru järgi parameeter selle sees. Luuakse uus teade ning asendatakse muudetav väärtus, muud kopeeritakse. Edasi pannakse uus teade vana asemele.

getColumnClass ning isCellEditable toimivad nii nagu nimigi määrab. Esimene teatab, millise Java klassile vastavate andmetega tegemist on. Teine vastab, kas vastava veeru andmeid tabelis muuta lubatakse.

Edasi järgnevad mõned käsklused radade haldamiseks puus. copyTrack paigutab märgistatud raja osuti ajutisse muutujasse. getCurrentSequence leiab märgistatud kohale vastava sekventsi. Kui märgistatud on sekvents, väljastatakse osuti sellele enesele. Kui aga rada, siis leitakse sekvents, millesse vastav rada kuulub. newSequence loob uue sekventsi, leiab selle nime ning palub sekventsi puusse paigutada. pasteTrack loob märgistatud sekventsi sisse uue raja ning lisab sellesse eelnevalt meelde jäetud rajal paiknevad teated. Uuele rajale kantakse üle vaid teadete osutid, teadetest endist koopiaid ei tehta. Alles siis, kui kasutaja soovib tabeli kaudu muuta teates paiknevaid parameetreid, luuakse endise asemele uus teate eksemplar.

Sekventsi puuse lisamise eest hoolitseb addSequenceToTree. Sekventsi (faili) nimi tehakse rasvaseks HTML-i käskude abil, mida on Swingi komponentide juures lubatud kujunduseks kasutada. Kõikide sekventsis asuvate radade tarbeks luuakse puus esitamise jaoks tarvilikud DefaultMutableTreeNode tüüpi komponendid.

Järgnev kood on juba küllalt tavapärane pea iga Swingi rakenduses puhul. actionPerformed sündmuste haldamiseks. Faili lugemisel paigutatakse tekstifailis olevad andmed sekventsi, salvestamisel aga talletatakse sekventsi andmed MIDI-faili.

Menüü juures on tarvis vähemasti kolm osa. JMenuBar tähistab kogu menüüriba, JMenu ühte menüüpaneeli ning sinna külge JMenuItem, mille kaudu juba inimene käsklusi anda saab. Ning edasi juba head katsetamist ning jõudu näite põhjal omale tarviliku rakenduse koostamisel.

```
import javax.swing.*;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.io.IOException;
import java.io.File;
import java.util.*;

import java.awt.*;
import java.awt.event.*;
import java.io.*;

import javax.sound.midi.*;
import javax.swing.tree.*;
import javax.swing.table.*;
import javax.swing.event.*; //TreeSelectionListener, -event

/**
* Midi-redaktor. Koostanud TPÜ Informaatika üliõpilane Kaur Männiko
* graafika ja muusika programmeerimise kursuse kodutööna.
*/
```

```
public class KMidi extends JPanel implements ActionListener, TreeSelectionListener {
    Sequence sekvents;
    Sequencer sekventser;
    JButton btnPlay, btnStop, btnNew, btnCopy, btnCut, btnPaste, btnDelete;
    JScrollPane treeView;
    final JTree tree;
    final JTable table;
    File file;
    DefaultMutableTreeNode rootNode;
    DefaultTreeModel treeModel;

    public void play() {
        try {
            Object src = tree.getLastSelectedPathComponent();
            if (!(src instanceof DefaultMutableTreeNode))
                return;
            DefaultMutableTreeNode node = (DefaultMutableTreeNode) src;
            if (node.getUserObject() == null)
                return;

            if ( node.getUserObject() instanceof NodeObjectHolder) {
                sekvents = (Sequence) ((NodeObjectHolder)node.getUserObject()).
userObject;

            } else if( node.getUserObject() instanceof Track) {
                Track algrada = (Track)node.getUserObject();
                sekvents = (Sequence) ((NodeObjectHolder) ((DefaultMutableTreeNode)
node.getParent()).getUserObject() ).userObject;
                sekvents = new Sequence(sekvents.getDivisionType(),
sekvents.getResolution());
                Track uusrada = sekvents.createTrack();

                for (int nr = 0; nr < algrada.size(); nr++)
                    uusrada.add(algrada.get(nr));

            } else
                return; //root

            sekventser.setSequence(sekvents);
            sekventser.start();

        } catch (InvalidMidiDataException e) {e.printStackTrace();};
    }

    public void stop() {
            sekventser.stop();
    }

    public KMidi() {
        super(new BorderLayout());

        JToolBar toolbar = new JToolBar();

        btnPlay = new JButton("Play");
        btnPlay.addActionListener(this);
        toolbar.add(btnPlay);
        btnStop = new JButton("Stop");
        btnStop.addActionListener(this);
        toolbar.add(btnStop);
        btnNew = new JButton("Uus");
        btnNew.addActionListener(this);
        toolbar.add(btnNew);
        btnCopy = new JButton("Kopeeri");
        btnCopy.addActionListener(this);
        toolbar.add(btnCopy);
        btnCut = new JButton("Lõika");
        btnCut.addActionListener(this);
        toolbar.add(btnCut);
        btnPaste = new JButton("Kleebi");
        btnPaste.addActionListener(this);
        toolbar.add(btnPaste);
        btnDelete = new JButton("Kustuta");
        btnDelete.addActionListener(this);
        toolbar.add(btnDelete);

        toolbar.setOrientation(JToolBar.HORIZONTAL);
        add(toolbar, BorderLayout.NORTH);

        rootNode = new DefaultMutableTreeNode("MidiSystem");
        treeModel = new DefaultTreeModel(rootNode);
```

```
        tree = new JTree(treeModel);
        tree.addTreeSelectionListener(this);

        treeView = new JScrollPane(tree);
        treeView.setPreferredSize(new Dimension(300,200));

        add(treeView, BorderLayout.WEST);

        table = new JTable();
        JScrollPane scrollpane = new JScrollPane(table);

        add(scrollpane, BorderLayout.CENTER);

        try {
            sekventser = MidiSystem.getSequencer();
            if (!sekventser.isOpen()) {
                System.out.print("avatakse sekventser...");
                sekventser.open();
                System.out.println(" ok");
            }
        } catch (MidiUnavailableException e) {e.printStackTrace();};

    }

    public void valueChanged(TreeSelectionEvent e) {
        Object src = e.getPath().getLastPathComponent();
        if (!(src instanceof DefaultMutableTreeNode))
           return;
        DefaultMutableTreeNode node = (DefaultMutableTreeNode) src;
        if (node.getUserObject() == null)
            return;
        if(!( node.getUserObject() instanceof Track))
            return;

        final Track track = (Track) node.getUserObject();
        final String[] columnNames = {"Tiks",
                                      "Kanal",
                                      "Teade",
                                      "kõrgus",
                                      "valjus",
                                      "Meta tüüp"};

        TableModel dataModel = new AbstractTableModel() {
            public String getColumnName(int column) { return columnNames[column];}
            public int getColumnCount() { return columnNames.length; }
            public int getRowCount() { return track.size();}

            public Object getValueAt(int row, int col) {
                MidiEvent me = track.get(row);
                Long tick = new Long(me.getTick());
                MidiMessage msg = me.getMessage();

                if (msg instanceof ShortMessage){
                    ShortMessage sm = (ShortMessage)msg;
                    switch (col) {
                        case 0: return tick;
                        case 1: return new Integer(sm.getChannel());
                        case 2: return new Integer(sm.getCommand());
                        case 3: return new Integer(sm.getData1());
                        case 4: return new Integer(sm.getData2());
                        default: return "";
                    }
                }
                else if (msg instanceof MetaMessage){
                    MetaMessage mm = (MetaMessage)msg;
                    switch (col) {
                        case 0: return tick;
                        case 5: return new Integer(mm.getType());
                        default: return "";
                    }
                }
                else if (col == 0)
                        return tick;
                    else
                        return "";
            }

            public Class getColumnClass(int c) {
                //return getValueAt(0, c).getClass();
                //return Class.forName("Integer");
                return (new Long(0)).getClass();
            }
```

```
            public void setValueAt(Object aValue, int row, int col) {
                System.out.println("set value "+row+","+ col);

                MidiEvent me = track.get(row);
                Long tick = new Long(me.getTick());
                MidiMessage msg = me.getMessage();

                if (col == 0) {
                    me.setTick(((Long)aValue).longValue());
                    return;
                }
                int val = ((Long)aValue).intValue();

                if (msg instanceof ShortMessage){
                    ShortMessage sm = (ShortMessage)msg;
                    int command = sm.getCommand();
                    int channel = sm.getChannel();
                    int data1 = sm.getData1();
                    int data2 = sm.getData2();

                    int status = sm.getStatus();

                    System.out.println("enne muutmist:
"+command+","+channel+","+data1+","+ data2);
                    switch (col) {
                        //case 0: me.setTick(val);
                        case 1: channel = val; break;
                        case 2: command = val; break;
                        case 3: data1 = val; break;
                        case 4: data2 = val; break;
                    }
                    System.out.println("muudetakse:
"+command+","+channel+","+data1+","+ data2);

                    try {
                        ShortMessage sm2 = new ShortMessage();
                        sm2.setMessage(command, channel, data1, data2);
                        MidiEvent me2 = new MidiEvent(sm2, me.getTick());
                        track.remove(me);
                        track.add(me2);

                    } catch (InvalidMidiDataException ex) {
                        ex.printStackTrace();
                    }

                }
            }

            public boolean isCellEditable(int row, int col) {
                if (col == 5)
                    return false;
                else
                    return true;
            }

        };

        table.setModel(dataModel);
    }

    Track tempTrack = null;

    public void copyTrack() {
        if (tree.getLastSelectedPathComponent() == null) return;
        DefaultMutableTreeNode node = (DefaultMutableTreeNode)
tree.getLastSelectedPathComponent();
        if( node.getUserObject() instanceof Track) {
            tempTrack = (Track)node.getUserObject();

        } else
            return; //root
    }

    private Sequence getCurrentSequence() {
        Sequence seq = null;
        if (tree.getLastSelectedPathComponent() == null) return null;
        DefaultMutableTreeNode node = (DefaultMutableTreeNode)
tree.getLastSelectedPathComponent();
        if ( node.getUserObject() instanceof NodeObjectHolder) {
            seq = (Sequence) ((NodeObjectHolder)node.getUserObject()).userObject;

        } else if( node.getUserObject() instanceof Track) {
```

```
            Track algrada = (Track)node.getUserObject();
            seq = (Sequence) ((NodeObjectHolder) ((DefaultMutableTreeNode)
node.getParent()).getUserObject() ).userObject;
        }
        return seq;
    }

    private int jnr = 0;

    public void newSequence() {
        sekvents = getCurrentSequence();
        if (sekvents == null) return;
        try{
            sekvents = new Sequence(sekvents.getDivisionType(), sekvents.getResolution
());
            //addSeqenceToTree(sekvents, "uus-"+(new Integer(jnr++)).toString()
+".mid");
            addSeqenceToTree(sekvents, "uus-"+(jnr++)+".mid");
        }catch (InvalidMidiDataException e) {e.printStackTrace();}
    }

    public void cutTrack() {
        copyTrack();
        deleteTrack();
    }

    public void pasteTrack() {
        if (tree.getLastSelectedPathComponent() == null) return;
        DefaultMutableTreeNode node = (DefaultMutableTreeNode)
tree.getLastSelectedPathComponent();
        if ( node.getUserObject() instanceof Track)
            node = (DefaultMutableTreeNode)node.getParent();
        if (!( node.getUserObject() instanceof NodeObjectHolder))
            return;

        sekvents = (Sequence) ((NodeObjectHolder) node.getUserObject() ).userObject;
        Track uusrada = sekvents.createTrack();

        for (int nr = 0; nr < tempTrack.size(); nr++)
            uusrada.add(tempTrack.get(nr));  //NB! ei tee uusi eksemplare

        DefaultMutableTreeNode trackNode = new DefaultMutableTreeNode(uusrada);
        node.add(trackNode);
        treeModel.insertNodeInto(trackNode, node, 0);
    }

    public void deleteTrack() {
        if (tree.getLastSelectedPathComponent() == null) return;
        DefaultMutableTreeNode node = (DefaultMutableTreeNode)
tree.getLastSelectedPathComponent();
        if( node.getUserObject() instanceof Track) {
               Sequence sekvents = (Sequence) ((NodeObjectHolder)
((DefaultMutableTreeNode)node.getParent()).getUserObject() ).userObject;
               Track track = (Track)node.getUserObject();
               sekvents.deleteTrack(track);
               treeModel.removeNodeFromParent(node);
        } else
            return; //root
    }

    public void addSeqenceToTree(Sequence seq, String name) {
        DefaultMutableTreeNode sequenceNode = null;
        DefaultMutableTreeNode trackNode = null;
        NodeObjectHolder noh = new NodeObjectHolder("<html><b>" + name +
"</b></html>", seq);
        sequenceNode = new DefaultMutableTreeNode(noh);

        Track[] rajad = seq.getTracks();
        System.out.println(rajad.length+" rada");
        for (int nr = 0; nr < rajad.length; nr++){
            trackNode = new DefaultMutableTreeNode(rajad[nr]);
            sequenceNode.add(trackNode);
        }

        treeModel.insertNodeInto(sequenceNode, rootNode, rootNode.getChildCount());
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == btnPlay) {
            play();
        } else if (e.getSource() == btnStop) {
            stop();
```

```
        } else if (e.getSource() == btnCopy) {
            copyTrack();
        } else if (e.getSource() == btnNew) {
            newSequence();
        } else if (e.getSource() == btnCut) {
            cutTrack();
        } else if (e.getSource() == btnPaste) {
            pasteTrack();
        } else if (e.getSource() == btnDelete) {
            deleteTrack();
        } else if (e.getSource() == mnLoad) {
            loadFile();
        } else if (e.getSource() == mnSave) {
            saveFile();
        }

    }

    JFileChooser fc = new JFileChooser();

    public void loadFile() {
        int returnVal = fc.showOpenDialog(this);

        if (returnVal == JFileChooser.APPROVE_OPTION) {
            file = fc.getSelectedFile();
            try {
                System.out.println("Opening: " + file.getCanonicalPath());

                sekvents = MidiSystem.getSequence(file);
                addSeqenceToTree(sekvents, file.getName());
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        } else {
            System.out.println("Open command cancelled by user.");
        }
    }

    public void saveFile() {
        sekvents = getCurrentSequence(); //millist faili salvestada
        if (sekvents == null) return;

        int returnVal = fc.showSaveDialog(this);
        if (returnVal == JFileChooser.APPROVE_OPTION) {
            file = fc.getSelectedFile();



            try {
                MidiSystem.write(sekvents, 1, new FileOutputStream
(file.getCanonicalPath()));

                System.out.println("Saved: " + file.getCanonicalPath());
            } catch (IOException ex) {
                ex.printStackTrace();
            }
        } else {
            System.out.println("Save command cancelled by user.");
        }
    }

    JMenuItem mnUus, mnLoad, mnSave;

    public JMenuBar createMenuBar() {
        JMenuBar menuBar = new JMenuBar();
        JMenu menu = null;
        JMenuItem menuItem = null;

        menu = new JMenu("Failid");

        menuItem = new JMenuItem("Lae..");
        menuItem.addActionListener(this);
        menu.add(menuItem);
        mnLoad = menuItem;

        menuItem = new JMenuItem("Salvesta..");
        menuItem.addActionListener(this);
        menu.add(menuItem);
        mnSave = menuItem;
```

```
        menuBar.add(menu);

        return menuBar;
    }

    public static void main(String[] args) {
        // JFrame.setDefaultLookAndFeelDecorated(true);
        // Käsk kasutatav alates JDK versioonist 1.4

        JFrame frame = new JFrame("KMidi");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        KMidi kodContentPane = new KMidi();
        kodContentPane.setOpaque(true);
        frame.setJMenuBar(kodContentPane.createMenuBar());
        frame.setContentPane(kodContentPane);

        frame.pack();
        frame.setSize(600, 400);
        frame.setVisible(true);
    }

}


/**
* Vahend, mille abil hoida objekti koos tema juurde kuuluva sildiga
* Swingi puu tarvis esitataval kujul. Kasutatakse näiteks sekventsi
* hoidmiseks.
*/
class NodeObjectHolder {
    public String label = "no name";
    public Object userObject = null;

    public NodeObjectHolder() {
    }

    public NodeObjectHolder(String label, Object userObject) {
        this.label = label;
        this.userObject = userObject;
    }

    public String toString() {
        return label;
    }
}
```

Ülesandeid

Mandoliin

- Joonista ekraanile mandoliin.
- Kasutaja saab määrata joonistatavate krihvide arvu ning kaela laiust.
- Vajutades keele ja krihvi ristumiskohale, kõlab sellele vastav heli.

Kitarri mudel

- Joonista 6-keelse kitarri kaela mudel
- Mängi kitarriakord (E, H, G, D, A, E) (64, 59, 55, 50, 45, 40)
- Peenemal keelel saab määrata, milline krihv on alla vajutatud. Hääl kõlab vastavalt vajutatule
- Krihve saab valida ka teiste keelte puhul ning kuulata tulemust
- Lisaks võib valida täisakorde.

Akordion

- Joonista klaviatuuri ja 36 bassiga akordion.
- Basside ridade ja veergude arvu saab kasutaja määrata. Samuti hiire abil lõõtsa lahti ja kokku vedada.
- Lisaks eelmisele kõlab bassinuppudele vajutades neile vastav heli.

Vilepill

- Joonista vilepill. Pillil on avatud tekstiväljas määratud hulk auke.
- Lisaks eelmisele teeb pillile vajutamisel viimane häält.
- Augule vajutamisel on augud lõpust kuni sinnani avatud. Kõlab sellisele sõrmede paigutusele vastav hääl.

MIDI fail

- Mängi MIDI fail
- Väljasta radade arv
- Salvesta fail tooni jagu kõrgemalt.
- Kopeeri faili muusika iseendale sappa
- Lisa rada, kus viisi mängitakse nihkega (kaanon)

Digitaalheli

MIDI abil võib kord kokku pandud pillide helisid uuesti esitada, mängida saab vaid helikõrgusega ning kestusega. Olematute pillide hääli aga sünteesida ei õnnestu, samuti tuleb loobuda muudest heliefektidest. Kui palju kõlari membraan mingil hetkel välja venitatud on, seda otsustab meie eest helikaart või vastav süsteemiprogramm ning programmeerijal selle koha pealt kuigi palju kaasa rääkimist ei ole. "Hariliku" muusika loomise puhul ongi nii hea, sest saame rahumeeli noodikõrgustele ja -kestustele mõelda ning ei pruugi tehniliste andmete peale üleliia oma energiat kulutada. Kui aga tahta olemasolevat häält moonutada, kaja tekitada või hoopis uusi kõlasid luua, siis tuleb hakata mõtlema helilainete kuju peale. Edasi tuleb koostatud kuju kvantida, et saaks kusagil arvumassiivis hoida igale ajahetkele vastavat heligraafiku väärtust sarnaselt nagu matemaatikaski võime esitada funktsiooni x-idele vastavate y-ite massiivina, kus x-teljel oleks aeg ning y-il helirõhu kõrvalekalle tasakaaluasendist. Kvantimissagedusest (kandesagedusest) sõltub helikõvera edasiandmise kvaliteet. Mängitav kõver luuakse punkte ühendavatest sirgete järgi ning mida tihedamalt on punkte, seda lähedasemalt õnnestub säilitada esialgse lindistatud või välja arvutatud kõvera kuju.

Lihtne piiks

Järgnevalt on püütud kokku panna võimalikult lihtne heli, mida ka kõrvaga kuulda oleks. Kvantimissageduseks on määratud 1000 mõõtmist sekundis. Helirõhud on kordamööda määratud maksimumile ja miinimumile. Sedasi on väljastatava heli sageduseks pool kvantimissagedusest ehk 500 hertsi. Mõõtmistäpsuseks on üks bait ning heli väljastatakse ühe kanalina (mono, mitte stereo). Nõnda saab mõõtmistulemused panna ilusti baidimassiivi, kus igale kvandile vastab üks bait. AudioFormat'i abil määratakse, millist tüüpi andmed tulemas on. AudioSystem'i abil küsitakse operatsioonisüsteemilt SourceDataLine, kuhu saadetud heliandmed jõuavad lõpuks kasutaja kõrvadeni. Pärast voo avamist võime sinna andmeid saata kirjutades andmeid loodud voogu.

```
import javax.sound.sampled.*;
public class Piiks1{
 public static void main(String[] args) throws Exception {
   int     kandesagedus =1000;
   byte[] andmed=new byte[5*kandesagedus]; //5 sekundit
   for(int i=0; i<andmed.length; i++){
     if(i%2==0)andmed[i]=(byte) 127;
     else      andmed[i]=(byte)-128;
   }
```

```
   AudioFormat formaat = new AudioFormat(AudioFormat.Encoding.PCM_SIGNED,
               kandesagedus, 8, 1, 1, kandesagedus, false); //8bitine heli, 1, kanal, 1
bait raami kohta
   SourceDataLine line = (SourceDataLine) AudioSystem.getLine(
       new DataLine.Info(SourceDataLine.class, formaat,  AudioSystem.NOT_SPECIFIED)
   );
   line.open(formaat);
   line.start();
   line.write(andmed, 0, andmed.length);
   System.exit(0);
 }
}
```

Siinusekujuline laine

Soovides koostada keerukama kujuga helilaineid, peab ka kandesagedus suurem olema. Kui soovida helilainele anda siinuse lainjat kuju, peab ühe täisvõnke kohta rohkem mõõtmisi olema kui et ainult üks laine harja ning teine põhja tarvis. All näites ~20 mõõtmist täisvõnke kohta peaks juba silmaga kaugelt vaadates enamjaolt sinusoidi sarnase kuju andma ning ka kõrvaga kuulates ei tohiks karedus kuigivõrd tunda anda. Liiatigi kui nii tehniliste vahendite kui inimkõrva poolsed tasandused juurde arvata. Ka suurema kvantimissageduse kuid üheaegsete mitmete keerulisemate helide korral ei mõõdeta ühe heli andmeid oluliselt täpsemalt.

Veidi seletusi arvutamise kohta. Kui iga baidi järjekorranumbrist võtta siinus ning selle väärtus panna kvandile, siis saaksime sinusoidi, mille lainepikkus oleks 2*Pi baiti ehk 10000 Hz kandesageduse puhul tuleks helisageduseks 10000/6,28 ehk ~ 1600 Hz. Helitugevus oleks aga imetilluke, sest kasutada olevast 256-ühikulisest (8-bitisest) piirkonnast on tarvitatud vaid vahemikku miinust ühest üheni ehk siinuse väljundväärtust. Vahemikku saab kergesti suurendada, korrutades tulemuse kordajaga (siin juhul 100). Sel juhul kõigub väärtus miinus saja ning +100 vahel ning kasutatakse suurem osa (200/256) väljundpiirkonnast. Soovides helilainet kiiremini võnkuma panna, tuleb sama aja (baitide) jooksul suurendada arvu, millest siinust võetakse, kiiremini. Suurema kandesageduse puhul on aga ühe baidi jaoks eraldatud aeg väiksem. Baidi järjekorranumbri ja kandesageduse suhe aga näitab aega sekundites ning kui soovime, et kandesageduse muutumisel jääks helisagedus samaks, siis peab siinuse arvutamisel kandesagedus olema murrujoone all. Math.sin(2*Math.PI*nr/kandesagedus) puhul tehtaks parajasti üks võnge sekundis, et saaksime kõrgemat (kuuldavat) heli, selleks tuleb siinuse parameetriks olev väärtus soovitud sagedusega läbi korrutada. Nii saamegi lainekujulise heli arvutamiseks valemi
andmed[nr]=(byte)(100*Math.sin(2*Math.PI*nr*sagedus/kandesagedus));

```
import javax.sound.sampled.*;
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class Piiks2Rakend extends Applet implements ActionListener{
 Button nupp=new Button("Piiksu");
 public Piiks2Rakend(){
   setLayout(new BorderLayout());
   add(nupp);
   nupp.addActionListener(this);
 }
 public void actionPerformed(ActionEvent e){
  try{
   int  kandesagedus =10000;
   int  sagedus=440;  //440 hertsi
   int  nr=0;
   byte[] andmed=new byte[5*kandesagedus]; //5 sekundit
   while(nr<andmed.length){
     andmed[nr]=(byte)(100*Math.sin(2*Math.PI*nr*sagedus/kandesagedus));
     nr++;
   }
   AudioFormat formaat = new AudioFormat(AudioFormat.Encoding.PCM_SIGNED,
               kandesagedus, 8, 1, 1, kandesagedus, false); //8bitine heli, 1, kanal, 1
bait raami kohta
   SourceDataLine line = (SourceDataLine) AudioSystem.getLine(
       new DataLine.Info(SourceDataLine.class, formaat,  AudioSystem.NOT_SPECIFIED)
   );
   line.open(formaat);
   line.start();
   line.write(andmed, 0, andmed.length);
   line.close();
  }catch(Exception ex){ex.printStackTrace();}
```

```
 }
 public static void main(String[] argumendid){
   Frame f=new Frame("Piiks");
   Piiks2Rakend p=new Piiks2Rakend();
   f.add(p);
   f.setSize(200, 100);
   f.setVisible(true);
   p.actionPerformed(null);
 }
}
```

Tõusev heli

Soovides heli panna ühtlaselt tõusma, peab selle sagedust püsiva ajavahemiku järel (kahe) kordistama. Kaks korda suurem helisagedus annab oktaavi jagu kõrgema heli. Nõnda võib heli arvutamisel siinuse parameetrina aja (andmebaidi järjekorranumbri) kordajana kasutada astmefunktsiooni, mille astendajat pidevalt kasvatada.

```
andmed[nr]=(byte)(100*Math.sin(2*Math.PI*nr*
          Math.pow(2, nr*tous_oktaavites/andmed .length)*sagedus/kandesagedus));
```

Nõnda esineb aega tähistav number valemis kahes kohas. Esmalt annab sin (nr*tegur), kus tegur=2*Pi*sagedus/kandesagedus välja ühtlase sinusoidi, mille järgi võib pidevat samal kõrgusel püsivat tooni kuulata. Tooni pidevaks kergitamiseks tuleb siinuse parameeter läbi korrutada teisegi, nüüd juba ajast sõltuva koefitsiendiga, mille tulemusena saigi kokku eelpool toodud valem.

```
import javax.sound.sampled.*;
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class Piiks2aRakend extends Applet implements ActionListener{
 Button nupp=new Button("Piiksu");
 public Piiks2aRakend(){
   setLayout(new BorderLayout());
   add(nupp);
   nupp.addActionListener(this);
 }
 public void actionPerformed(ActionEvent e){
  try{
   int  kandesagedus =10000;
   int  sagedus=100;
   int  nr=0;
   double tous_oktaavites=2;
   byte[] andmed=new byte[5*kandesagedus];
   while(nr<andmed.length){
     andmed[nr]=(byte)(100*Math.sin(2*Math.PI*nr*
          Math.pow(2, nr*tous_oktaavites/andmed.length)*sagedus/kandesagedus));
     nr++;
   }
   AudioFormat formaat = new AudioFormat(AudioFormat.Encoding.PCM_SIGNED,
               kandesagedus, 8, 1, 1, kandesagedus, false);
   SourceDataLine line = (SourceDataLine) AudioSystem.getLine(
       new DataLine.Info(SourceDataLine.class, formaat,  AudioSystem.NOT_SPECIFIED)
   );
   line.open(formaat);
   line.start();
   line.write(andmed, 0, andmed.length);
   line.close();
  }catch(Exception ex){ex.printStackTrace();}
 }
 public static void main(String[] argumendid){
   Frame f=new Frame("Tõusev toon");
   Piiks2aRakend p=new Piiks2aRakend();
   f.add(p);
   f.setSize(200, 100);
   f.setVisible(true);
   p.actionPerformed(null);
 }
}
```

Kahebaidine kvant

Kvaliteetsema heli edastamiseks kaheksabitisest kvandist ei piisa. Jutu ning lihtsama heli saab selle abil küllalt hästi edasi anda, kuid linnuhäälte või orkestrimuusika puhul läheb märgatav kõrvale kuuldav osa kaduma. Kui mõõtmistäpsust suurendada, jääb rohkem algsest helist alles. Arvutusvalemite põhimõte jääb ikka samaks, kuid kui enne kaheksabitise heli korral oli kvandi suurimaks võimalikuks väärtuseks 127, siis kuueteistbitise juures annab maksimumi 32767. Seda tuleb valemite juures arvestada, muidu jääb hääl väga vaikseks või pole seda lihtsama helitehnika juures üldse kuulda.

Suurema mõõtmistäpsuse puhul tuleb lõivu maksta suurema mahuga. Kui ennist sai läbi aetud iga kvandi juures ühe baidiga, siis nüüd läheb vaja kahte.

```
byte[] andmed=new byte[5*kandesagedus*mitmeBitineHeli/8]; //5 sekundit
```

Kvandile vastavat numbrit arvutatakse sama valemi järgi, vaid valjust on niivõrd suurendatud, et see jääks parajalt kuulatavatesse piiridesse

```
int tulemus=(int)(valjus*Math.sin(Math.PI*nr*
       Math.pow(2, 0.5*Math.sin(nr*Math.PI/andmed.length))*sagedus/kandesagedus));
```

Baidimassiivis säilitamiseks ning edasiandmiseks tuleb täisarv kahe baidi vahel jagada. Siinses kodeeringus pannakse ettepoole viimane (madalam) bait, taha esimene (kõrgem) bait, kuid sõltuvalt formaadist võib järjekord ka teistpidine olla. Viimase baidi kättesaamiseks jäetakse alles vaid sellele baidile vastavad bitid, muud asendatakse nullidega. Tüübimuunduse abil baidiks muundamise järel jõuabki soovitud väärtus sihtmassiivi kohale. Täisarvust teise baidi kättesaamiseks nihutatakse see kõigepealt kaheksa biti jagu paremale esimese baidi kohale ning tehakse eelpool toodud tehe. Arvus 0xFF ehk 255 on esimese baidi kõik bitid ühed ning & tehtega algsest arvust vaid need ühed alles jättes, millele 255 ühed vastu panna on, tulebki kokku ühebaidiline väärtus. Nüüd piisab see vaid andmemassiivi üle kanda, et seda pärast kuulata annaks.

```
     andmed[nr++]=(byte)(tulemus & 0xFF);       //viimane bait
     andmed[nr++]=(byte)(tulemus >> 8 & 0xFF);  //eelviimane bait
```

Ning näide ise:

```
import javax.sound.sampled.*;
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class Piiks3Rakend extends Applet implements ActionListener, Runnable{
 Button nupp=new Button("Piiksu");
 Checkbox korda=new Checkbox("Korda");
 public Piiks3Rakend(){
   setLayout(new BorderLayout());
   add(nupp);
   add(korda, BorderLayout.SOUTH);
   korda.setState(true);
   nupp.addActionListener(this);
 }
 public void actionPerformed(ActionEvent e){
   new Thread(this).start();
 }
 public void run(){
  try{
   int  kandesagedus =44100;
   int  sagedus=400;
   int  mitmeBitineHeli=16;
   int  kanaliteArv=1;
   int  valjus=7000;  //max 32767
   int  nr=0;
   byte[] andmed=new byte[5*kandesagedus*mitmeBitineHeli/8]; //5 sekundit
   while(nr<andmed.length){
     int tulemus=(int)(valjus*Math.sin(Math.PI*nr*
       Math.pow(2, 0.5*Math.sin(nr*Math.PI/andmed.length))*sagedus/kandesagedus));
     andmed[nr++]=(byte)(tulemus & 0xFF);       //viimane bait
     andmed[nr++]=(byte)(tulemus >> 8 & 0xFF);  //eelviimane bait
        }
   AudioFormat formaat = new AudioFormat(AudioFormat.Encoding.PCM_SIGNED,
               kandesagedus, mitmeBitineHeli, kanaliteArv,
               kanaliteArv*mitmeBitineHeli/8, kandesagedus, false);
   SourceDataLine line = (SourceDataLine) AudioSystem.getLine(
       new DataLine.Info(SourceDataLine.class, formaat,  AudioSystem.NOT_SPECIFIED)
   );
```

```
    line.open(formaat);
    line.start();
    do{
      line.write(andmed, 0, andmed.length);
    }while(korda.getState());
    line.close();
   }catch(Exception ex){ex.printStackTrace();}
  }
  public static void main(String[] argumendid){
    Frame f=new Frame("Piiks");
    Piiks3Rakend p=new Piiks3Rakend();
    f.add(p);
    f.setSize(200, 100);
    f.setVisible(true);
    p.actionPerformed(null);
  }
}
```

Digitaalheli redaktor.

Mõnerealiste koodinäidetega õnnestus läbi proovida märgatav osa kvanditud heliga ümberkäimise tehnilistest käskudest. Ehkki lõpuks taandub suurem osa tegemisi otseste kvantide väärtuste väljaarvutamise ning tulemuste analüüsi peale, õnnestub kõrvale ehitatud kestprogrammi abil matemaatiliselt kuivana tunduvat arvutamist kasutajale märgatavalt mugavamaks muuta. Heliredaktorit on märkimisväärne osa meist kasutanud, paljudel pole aga aimu, kui kerge või keeruline võiks olla selline rakendus kokku panna. Siin on püütud alustada võimalikult lihtsast näitest ning korduste teel jõutud rohkemate võimalustega rakenduseni. Siin ettetulevaga sarnaseid küsimusi peab lahendama enamiku kasutajalt sisestust ootava helidega seotud rakenduste puhul.

Lihtsaim heliredaktor õnnestub kirjutada mõneteistkümne reaga ning tema ainuke oskus on koodi otse sisse kirjutatud. Olgu selleks siis faili salvestamine vaiksemaks, tagurpidi või kahe helifaili sisu kokkuliitmine. Üldiseks malliks ikka, et tuleb algsed failid sisse lugeda, leida juurdepääs üksikute kvantide andmete juurde. Seal soovitud muutused sisse viia või olemasolevate andmete põhjal uus massiiv kokku kirjutada ning lõpuks tulemus kasutajale ette mängida või faili salvestada. Rakendused asuvad keerukamaks minema selle osa juurest, kus kasutajal antakse voli määrata, milliseid muutusi andmetega ette võtta.

Märgistusala

Esimeseks näiteks on redaktor, kus kasutaja saab faili programmi mällu lugeda, seal andmeid kustutada ja kopeerida ning siis tulemuse uude faili talletada. Heli mängimise ega helikõvera vaatamise võimalust ei pakuta. Sobiva lõigu märkimiseks on loodud eraldi klass Margistusala1. Klassi nähtavaks osaks on lõuend, kus kasutaja saab omale soovitava lõigu märkida. Jäetakse lihtsalt meelde protsendid, kust maalt kuhu maale sedakorda kasutaja oma lõigu märkis. Alale eraldatud laiust tähistatakse 100% ehk 1,0-ga. Kui kasutaja juhtus märkima näiteks loo keskmise kolmandiku, siis see talletatakse muutujates väärtused 0,33 ja 0,66 abil vastavalt algus- ja lõppprotsendi tarbeks. Muutuja kaudu kannatab isendile ette anda mälus hoitava andmeploki kogupikkuse ning sellele vastavalt teatavad meetodid algKaader ja loppKaader, mitmendast kaadrist mitmenda kaadrini kasutaja märgistatud ala paikneb. Et testiks on klassil küljes ka main-meetod, saab komponenti ka eraldi katsetada ning veenduda, et kasutajal soovitud lõike ka märgistada õnnestub.

```
import java.awt.*;
import java.awt.event.*;
public class Margistusala1 extends Panel implements MouseListener{
  int kogupikkus=50000;
  double algusprotsent=0.1, loppprotsent=0.4;
  Color margistusvarv=Color.black;
  public Margistusala1(){
    addMouseListener(this);
  }
  public void paint(Graphics g){
    g.setColor(margistusvarv);
    g.fillRect(
      (int)(algusprotsent*getWidth()), 0,
      (int)((loppprotsent-algusprotsent)*getWidth()), getHeight()
    );
  }
  public void mousePressed(MouseEvent e){
    algusprotsent=e.getX()/(double)getWidth();
  }
  public void mouseReleased(MouseEvent e){
    loppprotsent=e.getX()/(double)getWidth();
```

```
    if(loppprotsent<algusprotsent){
      double abi=algusprotsent; algusprotsent=loppprotsent; loppprotsent=abi;
    }
    repaint();
  }
  public int algKaader(){
    return (int)(algusprotsent*kogupikkus);
  }
  public int loppKaader(){
    return (int)(loppprotsent*kogupikkus);
  }
  public void mouseClicked(MouseEvent e){}
  public void mouseEntered(MouseEvent e){}
  public void mouseExited(MouseEvent e){}
  public static void main(String[] arg){ //ala testimine
    Frame f=new Frame();
    f.add(new Margistusala1());
    f.setSize(300, 200);
    f.setVisible(true);
  }
}
```

Iseenesest ei pea loodud märgistusala sugugi olema seotud helitöötlusprogrammiga. Sarnaselt on sinna külge võimalik haakida mõnda muud osa andmete väljarealdamist vajavat rakendust - olgu selleks või nimede eraldamine telefoniraamatust. Järgnevalt aga näha, kuidas heliredaktor eelnevalt kirjeldatud märgistusala vajab.

Kopeerimine ja kleepimine

Et rakenduses saaks pärast avamist midagi heliga peale hakata, selleks tuleb mõni olemasolev helifail sisse lugeda. Tänuväärseks vahendiks on AudioInputStream, mille abil vähemasti teoreetiliselt on võimalik ühtse lähenemise kaudu kätte saada andmebaite kõigist formaatidest, millega Java hakkama saama peaks. Kaheksabitise heli puhul olen märganud, et AudioInputStreamil on raskusi tuvastamisega, kas andmeid hoitakse märgita või märgiga arvudena ning tõenäoliselt võib leida ebakõlasid ehk muudegi vormingute puhul, kuid üldiselt on klassi võimalused täiesti kasutatavad.

Järgnevalt eeldatakse, et tegemist on ühebaidise ehk kaheksabitise heliga ning helifaili kvandid loetakse ükshaaval failist ja pannakse mälupuhvrisse. Kui andmed otsas (read() väljastab -1), siis muudetakse puhvri sisu baidimassiiviks. Nõnda mäluvoo abil on mugav toimida, kui pole teada saabuvate andmete pikkust.

```
AudioInputStream sisse=AudioSystem.getAudioInputStream(new File(tfLae.getText()));
ByteArrayOutputStream malu=new ByteArrayOutputStream();
int nr=sisse.read();
while(nr!=-1){      //loetakse voo sisu mälupuhvrisse
  malu.write(nr);
  nr=sisse.read();
}
andmed=malu.toByteArray();
ala.kogupikkus=andmed.length;
formaat=sisse.getFormat();
```

Mälust faili kirjutamisel luuakse kõigepealt AudioInputStream mälus olevast andmemassiivist ning saabunud voog suunatakse soovitud formaadis faili kettal.

```
  AudioInputStream ais=new AudioInputStream(
   new ByteArrayInputStream(andmed), formaat, andmed.length
 );
AudioSystem.write(ais, AudioFileFormat.Type.WAVE, new File(tfSalvesta.getText()));
```

Kasutaja soovitud redigeerimisoperatsioonides tuleb mälus olevate andmetega lihtsalt soovitud muutused ette võtta. Soovides, et märgitud ploki kohal oleks helis vaikus, tuleb kõikide selle ploki andmete väärtused kirjutada ühesugusteks. Kui tegemist on märgiga täisarvuga kvantidega, siis sobib null vaikimisväärtuseks täiesti. Kui märgiga osa puudub (tüübiks näiteks PCM_UNSIGNED), siis võiks vaikuse korral olla väärtus pool vastava arvu bittidega tekitatavast maksimumväärtusest.

```
    for(int i=ala.algKaader(); i<=ala.loppKaader(); i++){
      andmed[i]=(byte)0;
    }
```

Kopeerimise puhul luuakse kõigepealt sobiva pikkusega puhver ning siis kopeeritakse ükshaaval andmeplokis märgitud baidid puhvrisse.

```
    puhver=new byte[ala.loppKaader()-ala.algKaader()];
    for(int i=0; i<puhver.length; i++){
      puhver[i]=andmed[ala.algKaader()+i];
    }
```

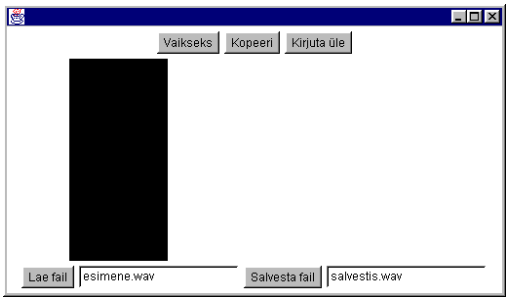
Andmete ülekirjutuse juures kirjutatakse puhvris olevad baidid andmemassiivi baitidele peale alates kursori asukohast. Nii nagu vanematel tekstiredaktoritel on tunduvalt tavalisem ülekirjutus kui vahelekirjutus, nii ka heliredaktori puhul on andmeid eelmiste peale lihtsam panna kui vahele. Nõnda ei pea hakkama ülejäänud andmeid edasi liigutama. Ning et pool muna on parem kui tühi koor, siis piirdume siin pealekirjutusega.

```
    if(puhver==null)return;
    for(int i=0; i<puhver.length; i++){
      andmed[ala.algKaader()+i]=puhver[i];
    }
```

Edasi lihtsakoelise redaktori kood tervikuna.

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.sound.sampled.*;
public class Digiheliredaktor1 extends Panel implements ActionListener{
  byte[] andmed;
  byte[] puhver;
  AudioFormat formaat;
  Button kopeeri=new Button("Kopeeri");
  Button kirjutaYle=new Button("Kirjuta üle");
  Button tyhjenda=new Button("Vaikseks");
  Margistusala1 ala=new Margistusala1();
  Button lae=new Button("Lae fail");
  TextField tfLae=new TextField("esimene.wav", 20);
  Button salvesta=new Button("Salvesta fail");
  TextField tfSalvesta=new TextField("salvestis.wav", 20);
  public Digiheliredaktor1(){
    Panel p=new Panel();
    p.add(tyhjenda);
    p.add(kopeeri);
    p.add(kirjutaYle);
    Panel alapaneel=new Panel();
    alapaneel.add(lae);
    alapaneel.add(tfLae);
    alapaneel.add(salvesta);
    alapaneel.add(tfSalvesta);
    setLayout(new BorderLayout());
    add(p, BorderLayout.NORTH);
    add(ala, BorderLayout.CENTER);
    add(alapaneel, BorderLayout.SOUTH);
    kopeeri.addActionListener(this);
    kirjutaYle.addActionListener(this);
    tyhjenda.addActionListener(this);
    lae.addActionListener(this);
    salvesta.addActionListener(this);
  }
  public void actionPerformed(ActionEvent e){
    if(e.getSource()==lae){laeFail(); }
    if(e.getSource()==salvesta){salvestaFail(); }
```

```
    if(e.getSource()==tyhjenda){tyhjendaLoik(); }
    if(e.getSource()==kopeeri){kopeeriLoik(); }
    if(e.getSource()==kirjutaYle){kirjutaLoikYle(); }
  }
  public void laeFail(){
   try{
    AudioInputStream sisse=AudioSystem.getAudioInputStream(new File(tfLae.getText()));
    ByteArrayOutputStream malu=new ByteArrayOutputStream();
    int nr=sisse.read();
    while(nr!=-1){      //loetakse voo sisu mälupuhvrisse
      malu.write(nr);
      nr=sisse.read();
    }
    andmed=malu.toByteArray();
    ala.kogupikkus=andmed.length;
    formaat=sisse.getFormat();
   }catch(Exception viga){
     viga.printStackTrace();
   }
  }

  public void salvestaFail(){
   try{
      AudioInputStream ais=new AudioInputStream(
       new ByteArrayInputStream(andmed), formaat, andmed.length
     );
    AudioSystem.write(ais, AudioFileFormat.Type.WAVE, new File(tfSalvesta.getText()));
   }catch(Exception viga){
     viga.printStackTrace();
   }
  }

  public void tyhjendaLoik(){
    for(int i=ala.algKaader(); i<=ala.loppKaader(); i++){
      andmed[i]=(byte)0;
    }
  }

  public void kopeeriLoik(){
    puhver=new byte[ala.loppKaader()-ala.algKaader()];
    for(int i=0; i<puhver.length; i++){
      puhver[i]=andmed[ala.algKaader()+i];
    }
  }

  public void kirjutaLoikYle(){
    if(puhver==null)return;
    for(int i=0; i<puhver.length; i++){
      andmed[ala.algKaader()+i]=puhver[i];
    }
  }


  public static void main(String[] argumendid){
    Frame f=new Frame();
    f.add(new Digiheliredaktor1());
    f.setSize(400, 300);
    f.setVisible(true);
  }
}
```



Redigeeritud heli kuulamine eraldi programmis.

Teine ring

Märgistusalale lisati helikõvera joonistusoskus. Nii pole vaja vaid ligikaudselt protsente piiluda. Kui on tegemist muutuva valjusega heliga, siis õnnestub kergesti välise pildi järgi eristada, millal uus rõhuline koht algab. Märgistusvärv joonistatakse alla ning helikõver selle peale, nõnda õnnestub mõlemat üheaegselt vaadata.

Joonistamisel liigutakse tsükliga läbi kogu andmeploki, tõmmates joone eelmisest punktist jooksvasse.
(andmed[i]&0xff)/250.0*getHeight() lahtiseletatuna:

0xff on täisarv, mille bitid kahendsüsteemis on kõik ühed. Kui byte-tüüpi väärtus &-operaatoriga sellise arvuga ühendada, siis tulemuseks on int-väärtus, millel samad bitid kui algsel byte-väärtusel, ent väärtuseks on positiivne arv 0 ja 255 vahel byte -128 ja +127 asemel. 250ga läbijagamine ning ala kõrgusega läbi korrutamine hoolitseb, et helilained oleksid võrdelised näidatava ala suurusega. Kui andmeid pole veel määratud, siis on vastava muutuja väärtuseks null ning helikõverat pole vaja ega võimalik joonistada.

Andmete edastamiseks märgistusalale loodi meetod seaAndmed. Seal antakse ette uus andmemassiiv, mille peale jääb märgistusala muutuja osutama, andmeid ei kopeerita. Lisaks jäetakse meelde massiivi pikkus ning nullitakse muud vajalikud muutujad. repaint() teatab, et pilt tuleb joonistada uuendatud andmete järgi.

```
import java.awt.*;
import java.awt.event.*;

public class Margistusala2 extends Panel implements MouseListener{
  private int kogupikkus=50000;
  byte[] andmed;  //heli andmed
  double algusprotsent=0.1, lopprotsent=0.1;
  Color margistusvarv=Color.green;
  Color joonistusvarv=Color.black;

  public Margistusala2(){
    addMouseListener(this);
  }
  public void paint(Graphics g){
    g.setColor(margistusvarv);
    g.fillRect(
      (int)(algusprotsent*getWidth()), 0,
      (int)((lopprotsent-algusprotsent)*getWidth()), getHeight()
    );
    g.drawLine(  //kursor
      (int)(algusprotsent*getWidth()), getHeight()/2,
      (int)(algusprotsent*getWidth()), getHeight()
    );
    if(andmed!=null){
      g.setColor(joonistusvarv);
      for(int i=1; i<andmed.length; i++){
        g.drawLine(
          (int)((i-1.0)/andmed.length*getWidth()),
          (int)(getHeight()-(andmed[i-1]&0xff)/250.0*getHeight()),
          (int)(((double)i)/andmed.length*getWidth()),
          (int)(getHeight()-(andmed[i]&0xff)/250.0*getHeight())
        );
      }
    }
  }

  public void seaAndmed(byte[] uuedAndmed){
    andmed=uuedAndmed;
    kogupikkus=andmed.length;
    algusprotsent=0;
    lopprotsent=0;
    repaint();
  }

  public void mousePressed(MouseEvent e){
    algusprotsent=e.getX()/(double)getWidth();
  }

  public void mouseReleased(MouseEvent e){
    lopprotsent=e.getX()/(double)getWidth();
    if(lopprotsent<algusprotsent){
      double abi=algusprotsent; algusprotsent=lopprotsent; lopprotsent=abi;
    }
    repaint();
  }
```

```
  public int algKaader(){
    return (int)(algusprotsent*kogupikkus);
  }

  public int loppKaader(){
    return (int)(loppprotsent*kogupikkus);
  }
  public void mouseClicked(MouseEvent e){}
  public void mouseEntered(MouseEvent e){}
  public void mouseExited(MouseEvent e){}
}
```

Redaktorile tulid juurde nupud vahelekleepimise ning vahelt lõikamise tarbeks. Samuti õnnestub maha mängida nii kogu andmemassiivis olevat heli kui kasutaja märgitud helilõiku.

Nii lõikamise kui kleepimise puhul kasutatakse massiivide kopeerimiseks käsku System.arraycopy, mis pidada suurte andmemahtude korral olema arvutile valutum kui tuhandete üksikute baitide ükshaaval kopeerimine. Nagu allpool koodist näha, koostatakse lõikamise puhul kõigepealt uus massiiv, mis on algsest väljalõigatava osa jagu lühem. Edasi kopeeritakse märgitud osa puhvrisse, et vajadusel õnnestuks see sobivasse kohta kleepida. System.arraycopy soovib enesele viis argumenti. Esimeseks massiiv kust kopeerida, praegusel juhul muutuja nimega andmed. Edasi järjekorranumber, mitmendast elemendist kopeerimist alustada, ehk ala.algKaader(). Siis tuleb massiiv, kuhu kopeeritavad andmed paigutada - nimeks siin juhul puhver. Siis järjekorranumber, kust alates paigutatakse elemente uude massiivi. Et soovime puhvrisse paigutada lõigatu alates algusest siis selleks väärtuseks 0. Ja lõpuks kopeeritavate elementide arv ehk märgistatud lõigu pikkus.

Järgnevate kopeerimistega veel algsest andmeplokist nii märgistuseelne kui märgistusjärgne lõik uude massiivi ning lõikus ongi valmis.

```
  public void loikaLoik(){
    byte[] uus=new byte[andmed.length-(ala.loppKaader()-ala.algKaader())];
    puhver=new byte[ala.loppKaader()-ala.algKaader()];

    System.arraycopy(andmed, ala.algKaader(), puhver, 0, (ala.loppKaader()-
ala.algKaader()));
    System.arraycopy(andmed, 0, uus, 0, ala.algKaader());
    System.arraycopy(andmed, ala.loppKaader(), uus,
                     ala.algKaader(), andmed.length-ala.loppKaader());
    andmed=uus;
    uuendaAla();
  }
```

Kleepimine näeb lõikamisega küllalt sarnane välja. Vaheks ainult, et uus massiiv on eelmisest pikem ning puhvri sisu tuleb kursorile eelneva ning kursorile järgneva ploki vahele paigutada.

```
  public void kleebiLoik(){
    if(puhver==null)return;
    byte[] uus=new byte[
       andmed.length+puhver.length-(ala.loppKaader()-ala.algKaader())];
    System.arraycopy(andmed, 0, uus, 0, ala.algKaader());
    System.arraycopy(puhver, 0, uus, ala.algKaader(), puhver.length);
    System.arraycopy(andmed, ala.loppKaader(), uus,
              ala.algKaader()+puhver.length, andmed.length-ala.loppKaader());
    andmed=uus;
    uuendaAla();
  }
```

Kui jooksvalt võimalik oma töö vilju kuulata, siis sujub heli töötlemine ladusamalt. Hea on järele proovida, et kõik plaanitu ka kõrvade jaoks sobilikult kokku saab ning tulemust võib ka enne faili salvestamist ja selle maha mängimist kontrollida.

Siin näites ei hoita helikanalit pidevalt redigeerimisprogrammi all kinni, vaid küsitakse kanal alles siis, kui kavatsetakse mängima hakata. Tähtsaim kanali küsimise juures on formaat, milles kavatsetakse andmeid teele saatma hakata. Selleks jääb praegusel juhul sama algsest helifailist loetud formaat. Käsuga write saadetakse andmed helikaardi poole teele ning drain ootab, kuni saadetud andmed on jõutud maha mängida. Edasi võib kanali selleks korraks sulgeda, et liialt palju arvuti ressursse programm enese käes ei hoiaks.

```
  public void mangi(){
    try{  //Kontrollib, et opsüsteemilt on võimalik kanal küsida.
     kanal=(SourceDataLine)AudioSystem.getLine(
      new DataLine.Info(SourceDataLine.class, formaat, AudioSystem.NOT_SPECIFIED)
```

```
     );
     kanal.open();
     kanal.start();
     kanal.write(andmed, 0, andmed.length);
     kanal.drain();
     kanal.close();
    }catch(Exception ex){
      ex.printStackTrace();
    }
  }
```

Märgistatud lõigu mängimine erineb eelmisest vaid selle poolest, et kogu massiivi pikkuse asemel tuleb määrata baidid/kaadrid, kust alates ja kui palju neid helikaardile mängimiseks saata tuleb.

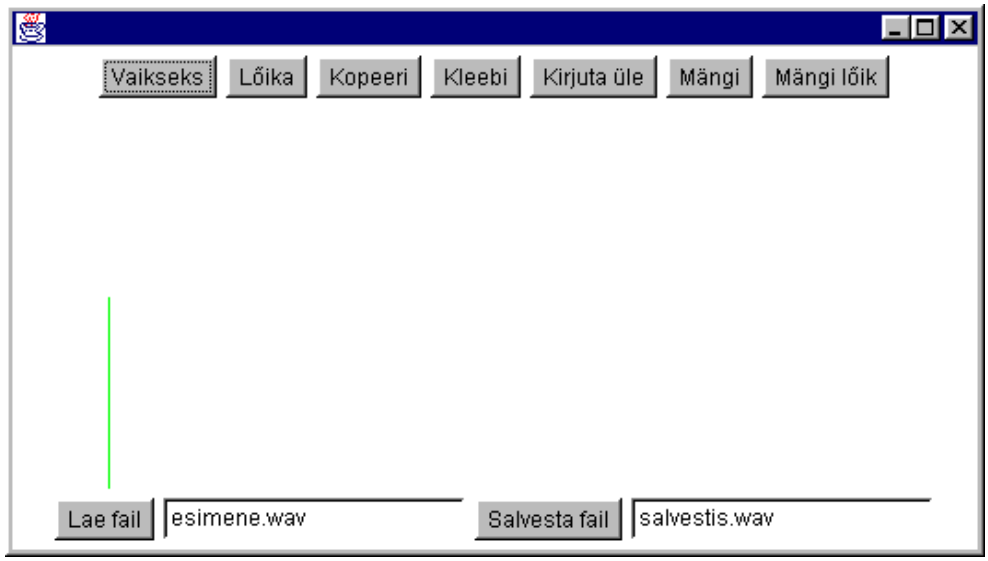
```
     kanal.write(andmed, ala.algKaader(), ala.loppKaader()-ala.algKaader());
```

Ning teise redaktori kood tervikuna.

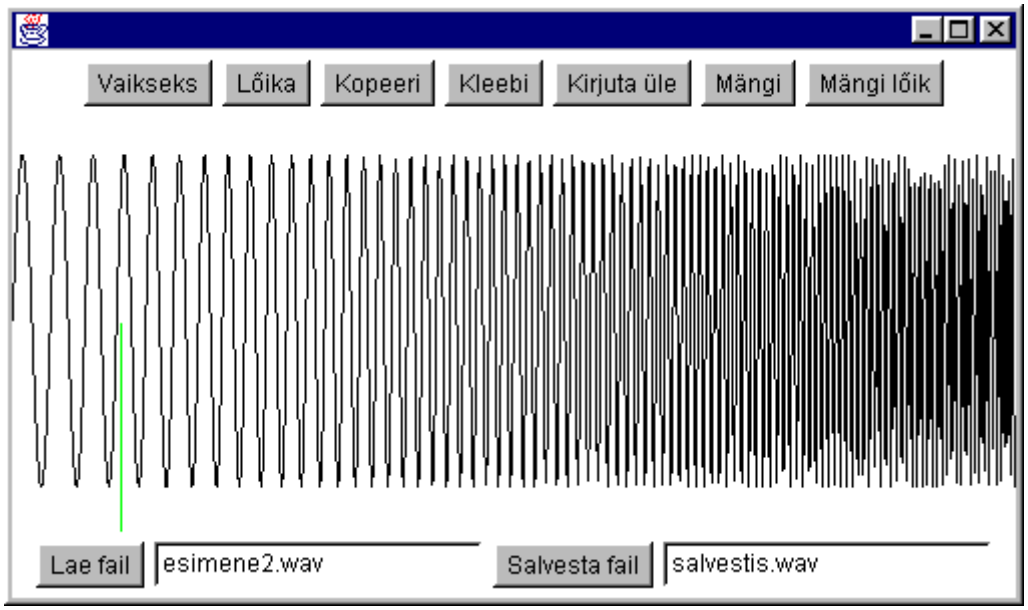
```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.sound.sampled.*;
public class Digiheliredaktor2 extends Panel implements ActionListener{
  byte[] andmed;
  byte[] puhver;
  AudioFormat formaat;
  Button kopeeri=new Button("Kopeeri");
  Button loika=new Button("Lõika");
  Button kleebi=new Button("Kleebi");
  Button kirjutaYle=new Button("Kirjuta üle");
  Button tyhjenda=new Button("Vaikseks");
  Button mangi=new Button("Mängi");
  Button mangiLoik=new Button("Mängi lõik");
  Margistusala2 ala=new Margistusala2();
  Button lae=new Button("Lae fail");
  TextField tfLae=new TextField("esimene.wav", 20);
  Button salvesta=new Button("Salvesta fail");
  TextField tfSalvesta=new TextField("salvestis.wav", 20);
  SourceDataLine kanal;
  public Digiheliredaktor2(){
    Panel p=new Panel();
    p.add(tyhjenda);
    p.add(loika);
    p.add(kopeeri);
    p.add(kleebi);
    p.add(kirjutaYle);
    p.add(mangi);
    p.add(mangiLoik);
    Panel alapaneel=new Panel();
    alapaneel.add(lae);
    alapaneel.add(tfLae);
    alapaneel.add(salvesta);
    alapaneel.add(tfSalvesta);
    setLayout(new BorderLayout());
    add(p, BorderLayout.NORTH);
    add(ala, BorderLayout.CENTER);
    add(alapaneel, BorderLayout.SOUTH);
    loika.addActionListener(this);
    kopeeri.addActionListener(this);
    kleebi.addActionListener(this);
    kirjutaYle.addActionListener(this);
    tyhjenda.addActionListener(this);
    lae.addActionListener(this);
    salvesta.addActionListener(this);
    mangi.addActionListener(this);
    mangiLoik.addActionListener(this);
  }
  public void actionPerformed(ActionEvent e){
    if(e.getSource()==lae){laeFail(); }
    if(e.getSource()==salvesta){salvestaFail(); }
    if(e.getSource()==tyhjenda){tyhjendaLoik(); }
    if(e.getSource()==loika){loikaLoik(); }
    if(e.getSource()==kopeeri){kopeeriLoik(); }
    if(e.getSource()==kleebi){kleebiLoik(); }
    if(e.getSource()==kirjutaYle){kirjutaLoikYle(); }
    if(e.getSource()==mangi){mangi(); }
    if(e.getSource()==mangiLoik){mangiLoik(); }
  }
  public void laeFail(){
   try{
    AudioInputStream sisse=AudioSystem.getAudioInputStream(new File(tfLae.getText()));
```

```
    System.out.println(sisse.getFormat());
    ByteArrayOutputStream malu=new ByteArrayOutputStream();
    int nr=sisse.read();
    while(nr!=-1){      //loetakse voo sisu mälupuhvrisse
      malu.write(nr);
      System.out.print(nr+" ");
      nr=sisse.read();
    }
    andmed=malu.toByteArray();
    //ala.kogupikkus=andmed.length;
    ala.seaAndmed(andmed);
    formaat=sisse.getFormat();
    uuendaAla();
   }catch(Exception viga){
     viga.printStackTrace();
   }
  }

  public void salvestaFail(){
   try{
      AudioInputStream ais=new AudioInputStream(
       new ByteArrayInputStream(andmed), formaat, andmed.length
     );
    AudioSystem.write(ais, AudioFileFormat.Type.WAVE, new File(tfSalvesta.getText()));
   }catch(Exception viga){
     viga.printStackTrace();
   }
  }

  public void loikaLoik(){
    byte[] uus=new byte[andmed.length-(ala.loppKaader()-ala.algKaader())];
    puhver=new byte[ala.loppKaader()-ala.algKaader()];

    System.arraycopy(andmed, ala.algKaader(), puhver, 0, (ala.loppKaader()-
ala.algKaader()));
    System.arraycopy(andmed, 0, uus, 0, ala.algKaader());
    System.arraycopy(andmed, ala.loppKaader(), uus,
                     ala.algKaader(), andmed.length-ala.loppKaader());
    andmed=uus;
    uuendaAla();
  }


  public void tyhjendaLoik(){
    for(int i=ala.algKaader(); i<=ala.loppKaader(); i++){
      andmed[i]=(byte)0;
    }
    uuendaAla();
  }

  public void kopeeriLoik(){
    puhver=new byte[ala.loppKaader()-ala.algKaader()];
    for(int i=0; i<puhver.length; i++){
      puhver[i]=andmed[ala.algKaader()+i];
    }
  }

  public void kleebiLoik(){
    if(puhver==null)return;
    byte[] uus=new byte[
       andmed.length+puhver.length-(ala.loppKaader()-ala.algKaader())];
    System.arraycopy(andmed, 0, uus, 0, ala.algKaader());
    System.arraycopy(puhver, 0, uus, ala.algKaader(), puhver.length);
    System.arraycopy(andmed, ala.loppKaader(), uus,
                ala.algKaader()+puhver.length, andmed.length-ala.loppKaader());
    andmed=uus;
    uuendaAla();
  }

  /**
  * Joonistusala uuendamine pärast andmete muutust.
  */
  void uuendaAla(){
    ala.seaAndmed(andmed);
  }

  public void kirjutaLoikYle(){
    if(puhver==null)return;
    for(int i=0; i<puhver.length; i++){
      andmed[ala.algKaader()+i]=puhver[i];
    }
    uuendaAla();
  }
```

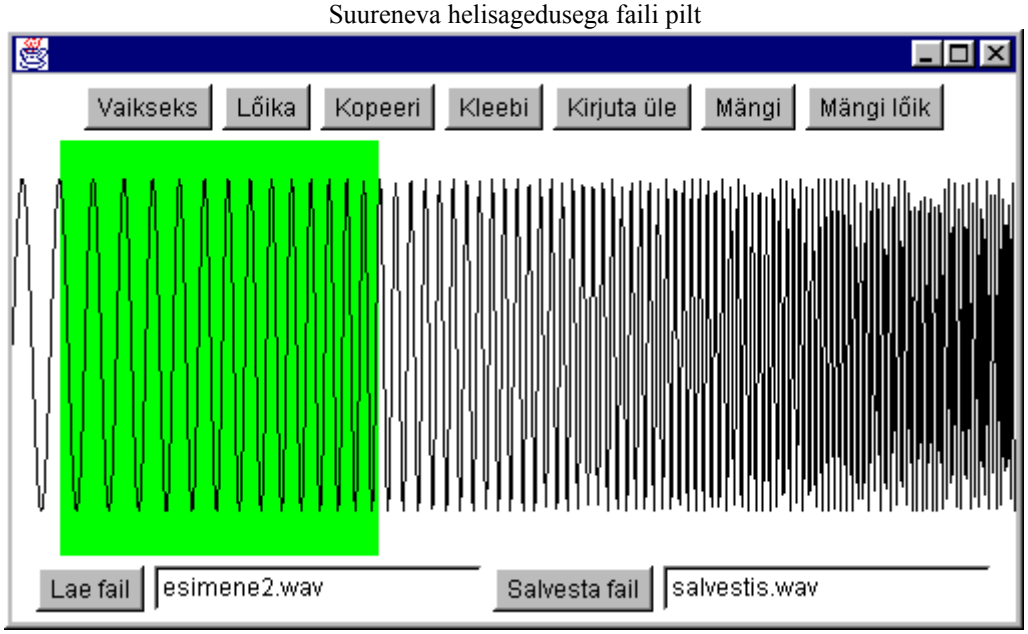
```
  public void mangi(){
    try{  //Kontrollib, et opsüsteemilt on võimalik kanal küsida.
     kanal=(SourceDataLine)AudioSystem.getLine(
      new DataLine.Info(SourceDataLine.class, formaat, AudioSystem.NOT_SPECIFIED)
     );
     kanal.open();
     kanal.start();
     kanal.write(andmed, 0, andmed.length);
     kanal.drain();
     kanal.close();
    }catch(Exception ex){
      ex.printStackTrace();
    }
  }

  public void mangiLoik(){
    try{  //Kontrollib, et opsüsteemilt on võimalik kanal küsida.
     kanal=(SourceDataLine)AudioSystem.getLine(
      new DataLine.Info(SourceDataLine.class, formaat, AudioSystem.NOT_SPECIFIED)
     );
     kanal.open();
     kanal.start();
     kanal.write(andmed, ala.algKaader(), ala.loppKaader()-ala.algKaader());
     kanal.drain();
     kanal.close();
    }catch(Exception ex){
      ex.printStackTrace();
    }
  }

  public static void main(String[] argumendid){
    Frame f=new Frame();
    f.add(new Digiheliredaktor2());
    f.setSize(400, 300);
    f.setVisible(true);
  }
}
```



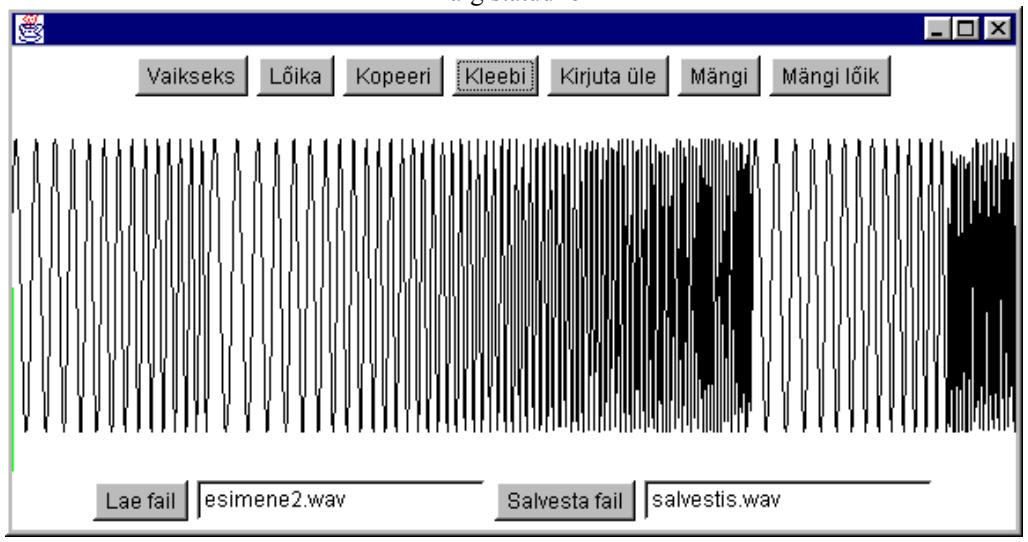
Avatud tühi redaktor



Suureneva helisagedusega faili pilt



Märgistatud lõik



Madalama sagedusega heli kopeerituna kõrgesageduslikuma vahele.

Kolmas ring

Tegemist pole sugugi veel täieliku redaktoriga, kuid siinset rakendust peaks saama juba mitmete "päris" helide juures rakendada. Arvestatud on nii mono- kui stereoheli ning kaheksa- kui kuueteistbitiseid märgiga ning märgita kvante, kusjuures kahebaidise kodeeringu puhul võib kõrgem bait olla nii ees- (big endian) kui tagapool (little endian). Mõnedki kontrollid on jäänud koodi lühiduse huvides peale panemata, kuid eks rakenduse eriomaduste lisamise järel tulevad nagunii kontrolli vajavad kohad paremini esile.

Võrreldes eelnenuga hoitakse märgistusala komponendis lisaks andmetele meeles formaati, sest samad andmed võivad eri formaatide puhul hoopis eri tulemuse anda. Näiteks stereo puhul loetakse samast andmemassiivist kvante kordamööda ning samade andmete monona mängimine ei pruugi viisi sugugi äratuntavaks jätta.

Märgistuse alustus- ja lõpetuskohta ei hoita enam meeles protsendina vaid kaadri järjekorranumbriga. Nii on võimalik täpsemalt määrata, milline koht märgistatud on. Terviküksusena käivad koos kaadrid, mida on mõtet vaid korraga kopeerida, lisada või eemaldada. Kui ühebaidise ja ühe kanaliga heli puhul oli kaadri suuruseks üks bait, siis kahebaidise stereoheli puhul on kaadri suuruseks juba neli baiti ning seda nelikut pole pea kusagil töötluse juures mõistlik lahutada.

Joonistusala on märgistusala komponendi sisse loodud eraldi sisemise lõuendiklassina. Nii on mugavam paluda alal enese sisse helikõver joonistada ning samal ajal saab märgistusala alla serva paigutada kerimisribad asukoha ja koefitsientide määramiseks ilma, et peaks programmis liialt energiat kulutama arvutamisele hoolitsemaks et lainekõver kogemata kerimisribade alla ei satuks.

Suuremaks arvutamist nõudvaks ülesandeks on soovitud kanalilt soovitud järjekorranumbriga helikvandi küsimine. Kvandi väärtus väljastatakse täisarvuna. Tulgu see siis ühe või kahebitine ehk märgiga või märgita.

```
int kysiKanaliltKvant(int kanal, int kaader){
```

Kõigepealt leitakse kaadri algus - koht massiivi algusest kaadri suuruse ja kaadri järjekorranumbri jagu baite edasi.

```
int kaadriAlgus=kaader*formaat.getFrameSize();
```

Edasi kaadri seest kvandi algus. Esimese kanali (kanali number 0 puhul) langevad kaadri ja kvandi algus kokku, muul juhul tuleb kvandi laiuse jagu edasi liikuda.

```
int kvandiAlgus=kaadriAlgus+kanal*formaat.getSampleSizeInBits()/8;
```

Kaheksabitise heli korral sobibki vastava baidi väärtused arvuks kirjutatuna võtta - juhul kui on tegemist märgita täisarvudega salvestuse puhul.

```
if(formaat.getSampleSizeInBits()==8){
  return andmed[kvandiAlgus] & 0xFF;
}
```

Kuueteistbitise heli korral tuleb järgnevast kahest baidist kokku kombineerida kvandi asukohta määrav arv.

```
int b1=andmed[kvandiAlgus] & 0xFF;
int b2=andmed[kvandiAlgus+1] & 0xFF;
```

Märgita arvude puhul on arvutus veidi lihtsam.

```
if(formaat.getEncoding().equals(AudioFormat.Encoding.PCM_UNSIGNED)){
```

Kui kõrgem bait eespool, siis tuleb seda loodavas arvus ühe baidi jagu vasakule nihutada.

```
 if(formaat.isBigEndian()){
  return b1 << 8 | b2;
 } else {
```

Muul juhul tuleb nihutamiseks võtta madalam bait.

```
  return b2 << 8 | b1;
 }
}
```

Kui kvante esitatakse märgiga arvude abil, siis tuleb hoolitseda, et vasakpoolne märgibitt kaduma ei läheks. Üheks võimaluseks peaks kõigepealt olema omistada väärtus kahebaidisele täisarvule short, kus vasakpoolseim bitt hoolitseb märgi eest. Edasine muundamine int-muutujaks enam väärtust ei muuda.

```
if(formaat.getEncoding().equals(AudioFormat.Encoding.PCM_SIGNED)){
             if(formaat.isBigEndian()){
              return (int)((short)(b1 << 8 | b2));
             } else {
              return (int)((short)(b2 << 8 | b1));
             }
            }
```

Kui tegemist tundmatu kodeeringuga, siis väljastatakse 0.

```
            return 0;
        }
```

Eraldi tähelepanu väärib kerimisribadest väärtuste välja küsimine. Esimeses ribas hoitakse väärtusi nullist sajani ning loetud sisu saab ette kujutada protsendina loo kogupikkusest. Nõnda siis leitakse joonistuseAlgKaader nihkeprotsendi ja kaadrite arvu korrutisena.

Nii x- kui y-suunalise suurenduse puhul on pandud kerimisriba väärtus arvu kaks astendajat muutma. Nõnda suurendab kümne pügala jagu skaalal liikumine laiust kaks korda ning järgmise kümne pügala jagu liikumine jälle kaks korda. Selline lähenemine peaks vaatajale loomulikum tunduma.

```
        public void adjustmentValueChanged(AdjustmentEvent e){
           joonistuseAlgKaader=sb1.getValue()*kaadriteArv/100;
           piksleidKaadriKohta=Math.pow(2, sb2.getValue()/10.0-7);
           suurendusKordaja=Math.pow(2, sb3.getValue()/10.0);
           ala.repaint();
        }
```

Ka kanali joonistamise tarbeks on eraldi alamprogramm loodud. Nõnda õnnestub paint-meetodit mõnevõrra lihtsustada. Meetodile antakse ette graafiline kontekst kuhu joonistada, kanal kust andmeid võtta ning ala suurus ja asukoht helikõvera paigutamiseks.

```
        void joonistaKanal(Graphics g, int kanal, int vasak, int yla,
                 int laius, int korgus){
```

Et poleks põhjust kogu helifaili jagu kaadreid läbi käia, vaid võiks piirduda ainult nähtavate kaadritega, siis leitakse kohe algul mahtuvate kaadrite arv

```
           int mahtuvateKaadriteArv=(int)(laius/piksleidKaadriKohta);
```

Kogu helikõver moodustatakse üksikutest joontest. Et joone puhul on vaja teada kahte otspunkti, siis on mõistlik punktid ühekaupa välja arvutada ning iga kord eelmise punkti andmed meeles pidada, et oleks võimalik vanast punktist uude joon tõmmata.

Kvandi väärtuse enese saab käsuga kysiKanaliltKvant. Muid tehteid kasutatakse helikõvera parajaks etteantud alasse paigutamiseks. Muutuja keskArv hoiab eneses vastava vormingu helikvantide väärtust vaikuseolekus, ehk väärtust, mis peaks sattuma kõvera keskele vertikaalsihti mööda. Suurima võimaliku väärtuse ehk maxArvuga läbi jagamine ning korgus/2-ga korrutamine venitab graafiku etteantud piiridesse. Kerimisribast sõltuva suurendusKordaja abil saab seda edaspidi täiendavalt skaleerida.

```
           int vanax=0;
           int vanay=(int)(yla+korgus/2-(kysiKanaliltKvant(kanal, joonistuseAlgKaader)-
keskArv)*korgus*suurendusKordaja/2/maxArv);
```

Samm näitab, mitme kaadri võrra igal korral edasi liigutakse. Kui kaadreid peaks paiknema laiuse ekraanipunktil rohkem kui üks, siis suurendatakse sammu niivõrd, et iga järgnev arvutatav kaader satuks uuele ekraanipunktile. Selliselt on võimalik hoolitseda, et ka pikkade helifailide andmete ligikaudne ekraanile joonistamine liialt kaua aega ei võtaks. Kui iga kaadri jaoks pole ekraanil punkti, siis paratamatult jääb kujutis mõnevõrra ebatäpne. Küll aga peaksid näha olema tähtsamad kõikumised valjuses ehk amplituudis. Kui iga kaadri asukohta tähistada punktiga ekraanil, tuleks kujutis mõnevõrra täpsem, kuid samas oleks heliosa ekraanil tihedate joonte tõttu pea ühtlaselt must ning joonistamine

nõuaks masinalt enam jõudu. Sammu aga ühest väiksemaks ei lasta minna, sest järgmine alumine täisarvuline väärtus oleks null ning nõnda jääks programm igavesse tsüklisse.

```
          int samm=(int)(1.0/piksleidKaadriKohta);
          if(samm<1){samm=1;}
          for(int k=joonistuseAlgKaader+1; k<kaadriteArv &&
k<joonistuseAlgKaader+mahtuvateKaadriteArv; k+=samm){
```

Edasi juba uus koordinaatide arvutus, sama põhimõttega kui ülalpoolgi. Ning ka x-suunal tuleb kaadri numbri, esimese joonistatava kaadri numbri ning venituskoefitsendi abil ekraanile sobiva punkti asukoht leida.

```
            int uusx=(int)((k-joonistuseAlgKaader)*piksleidKaadriKohta);
            int uusy=(int)(yla+korgus/2-(kysiKanaliltKvant(kanal, k)-
                    keskArv)*korgus*suurendusKordaja/2/maxArv);
            g.drawLine(vanax, vanay, uusx, uusy);
            vanax=uusx; vanay=uusy;
          }
        }
```

Ning nagu ikka - märgistusala kood tervikuna. Loodetavasti aitavad siin sees paiknevad kommentaarid mõistmist samuti veidi selgemaks teha.

```
import java.awt.*;
import java.awt.event.*;
import javax.sound.sampled.*;

/**
* Abiklass Digiheliredaktor3 tarbeks. Võimaldab näidata helilaine kuju
* ning märgistada kasutaja soovitud ploki andmetest.
*/
public class Margistusala3 extends Panel implements MouseListener, AdjustmentListener{
        /**
        *  Heliandmete interpreteerimiseks vajalik formaat. Eeldatavalt antakse
        *  ette koos andmetega.
        */
        AudioFormat formaat;
        /**
        * Massiiv näidatavate andmete hoidmiseks.
        */
       byte[] andmed=new byte[0];
       /**
       * Märgistatud ala alustava kaadri järjekorranumber.
       */
       protected int algusKaader;
       /**
       * Märgistatud ala lõpetava kaadri järjekorranumber.
       */
       protected int loppKaader;
       /**
       * Ala helikõvera tegelikuks näitamiseks.
       */
       protected JoonistusAla ala=new JoonistusAla();
       /**
       * Märgistatud piirkonna taustavärv. Samuti kursori värv.
       */
       Color margistusvarv=Color.green;
       /**
       * Helikõvera värv.
       */
       Color joonistusvarv=Color.black;
       /**
       * Koefitsient näitamaks mitu pikslit laiuse suunas on ühe kaadri
       * andmete näitamiseks. Ühest väiksem väärtus tähistab, et
       * sama ekraanipunkti peale peab mahtuma mitu kaadri väärtust.
       */
       double piksleidKaadriKohta=1;
       /**
       * Koefitsient vertikaalsihis suurenduse tarbeks. Kui väärtuseks on 1, siis
       * võtab heli oma maksimumväärtuste puhul piikide tipud kogu kanali andmete
        * joonistamiseks ette nähtud ala ulatuses.
       */
       double suurendusKordaja=1;
       /**
       * Kaadri number, millest alates on helikõver joonistusalas näha.
       */
       int joonistuseAlgKaader=0;
       /**
```

```
        * Helikaadrite arv etteantud andmeplokis. Arvutatakse uue ploki etteandmisel.
        */
        int kaadriteArv;
        /**
        * Helikõvera väärtus hariliku vaikuse puhul. Märgiga täisarvuga
        * kvantide puhul väärtuseks 0, märgita kvantide puhul pool täisväljalöögist.
        */
        int keskArv;
        /**
        * Helikvandi suurim võimalik väärtus jooksva vormingu puhul.
        */
        int maxArv;
        /**
        * Kerimisriba määramaks, millisest kohast alates asutakse helikõverat ekraanil
näitama.
        * Kerimisriba väärtusi saab kujutada protsentidena kogu lõigu pikkusest.
        */
        Scrollbar sb1=new Scrollbar(Scrollbar.HORIZONTAL, 1,  1, 0, 100);
        /**
        * Kerimisriba määramaks helikõvera horisontaalsuunalist väljavenitatust.
        * Joonistamisel kasutatakse logaritmilist skaalat.
        */
        Scrollbar sb2=new Scrollbar(Scrollbar.HORIZONTAL, 50, 1, 0, 100);
        /**
        *  Kõrgusesuunalise helikõvera väljavenitatuse määramine.
        */
        Scrollbar sb3=new Scrollbar(Scrollbar.HORIZONTAL, 10, 1, 0, 100);

         /**
         *  Konstruktor paigutuse ja kuularite sättimiseks.
         */
        public Margistusala3(){
                setLayout(new BorderLayout());
                add(ala);
               ala.addMouseListener(this);
               Panel ribaPaneel=new Panel(new GridLayout(1, 6));
               ribaPaneel.add(new Label("Asukoht:"));
               ribaPaneel.add(sb1);
               ribaPaneel.add(new Label("Laius:"));
               ribaPaneel.add(sb2);
               ribaPaneel.add(new Label("Kõrgus:"));
               ribaPaneel.add(sb3);
               add(ribaPaneel, BorderLayout.SOUTH);
               sb1.addAdjustmentListener(this);
               sb2.addAdjustmentListener(this);
               sb3.addAdjustmentListener(this);
        }

        /**
        * Soovitud kanali ja kaadri järgi helikvandi küsimine
        * täisarvuna. Väärtus antakse vastavalt andmetega
        * kaasas olevale vormingule.
        */
        int kysiKanaliltKvant(int kanal, int kaader){
            int kaadriAlgus=kaader*formaat.getFrameSize();
            int kvandiAlgus=kaadriAlgus+kanal*formaat.getSampleSizeInBits()/8;
            if(formaat.getSampleSizeInBits()==8){
              return andmed[kvandiAlgus] & 0xFF;
            }
            int b1=andmed[kvandiAlgus] & 0xFF;
            int b2=andmed[kvandiAlgus+1] & 0xFF;
            if(formaat.getEncoding().equals(AudioFormat.Encoding.PCM_UNSIGNED)){
             if(formaat.isBigEndian()){
              return b1 << 8 | b2;
             } else {
              return b2 << 8 | b1;
             }
            }
            if(formaat.getEncoding().equals(AudioFormat.Encoding.PCM_SIGNED)){
             if(formaat.isBigEndian()){
              return (int)((short)(b1 << 8 | b2));
             } else {
              return (int)((short)(b2 << 8 | b1));
             }
            }
            return 0;
        }



         /**
         * Kanali helikõvera joonistus vastavalt etteantud parameetritele.
         */
```

```
        void joonistaKanal(Graphics g, int kanal, int vasak, int yla, int laius, int
korgus){
          int mahtuvateKaadriteArv=(int)(laius/piksleidKaadriKohta);
          int vanax=0;
          int vanay=(int)(yla+korgus/2-(kysiKanaliltKvant(kanal, joonistuseAlgKaader)-
keskArv)*korgus*suurendusKordaja/2/maxArv);
          int samm=(int)(1.0/piksleidKaadriKohta);
          if(samm<1){samm=1;}
          for(int k=joonistuseAlgKaader+1; k<kaadriteArv &&
k<joonistuseAlgKaader+mahtuvateKaadriteArv; k+=samm){
            int uusx=(int)((k-joonistuseAlgKaader)*piksleidKaadriKohta);
            int uusy=(int)(yla+korgus/2-(kysiKanaliltKvant(kanal, k)-keskArv)
*korgus*suurendusKordaja/2/maxArv);
            g.drawLine(vanax, vanay, uusx, uusy);
            vanax=uusx; vanay=uusy;
          }
        }



        /**
        * Märgistusalale uute andmete seadmine või andmemuutusest teatamine.
        * Leitakse edaspidiseks joonistamiseks tarvilikud konstandid.
        */
        public void seaAndmed(byte[] uuedAndmed, AudioFormat uusFormaat){
                andmed=uuedAndmed;
                formaat=uusFormaat;
                algusKaader=0;
                loppKaader=0;
                kaadriteArv=andmed.length/formaat.getFrameSize();
                maxArv=255;
                if(formaat.getSampleSizeInBits()==16){
                  maxArv=65535;
                }
                 if(formaat.getEncoding()==AudioFormat.Encoding.PCM_SIGNED){
                   keskArv=0;
                   maxArv=maxArv/2;
                 } else {
                   keskArv=maxArv/2;
                 }
                ala.repaint();
        }

        /**
        * Kerimisribade muutuste peale konstantide väljaarvutus.
        */
        public void adjustmentValueChanged(AdjustmentEvent e){
           joonistuseAlgKaader=sb1.getValue()*kaadriteArv/100;
           piksleidKaadriKohta=Math.pow(2, sb2.getValue()/10.0-7);
           suurendusKordaja=Math.pow(2, sb3.getValue()/10.0);
           ala.repaint();
        }

        /**
        * Alguskaadri leidmine vastavalt hiirevajutuse asukohale.
        */
        public void mousePressed(MouseEvent e){
          algusKaader=(int)(e.getX()/piksleidKaadriKohta+joonistuseAlgKaader);
        }

        /**
        * Lõppkaader vastavalt hiirevajutuse asukohale. Kui märgistati
        * paremalt vasakule, siis hoolitsetakse, et algus
        * oleks ikka enne lõppu.
        */
        public void mouseReleased(MouseEvent e){
                loppKaader=(int)(e.getX()/piksleidKaadriKohta+joonistuseAlgKaader);
                if(loppKaader<algusKaader){
                  int abi=algusKaader; algusKaader=loppKaader; loppKaader=abi;
                }
                ala.repaint();
        }

        /**
        * Andmebaidi järjekorranumber, millest alates
        * hakkab märgistatud alguskaader. Väärtused langevad
        * kokku vaid ühe kanali ja ühebaidise heli puhul.
        * Tarvilik Digiheliredaktorile teadmaks, millisest
        * baidist alates tuleb andmeid muutma või kopeerima asuda.
        */
        public int algKaader(){
          return algusKaader*formaat.getFrameSize();
        }
```

```
        /**
        * Märgistuse lõppkaadri esimese baidi järjekorranumber.
        */
        public int loppKaader(){
          return loppKaader*formaat.getFrameSize();
        }

        /**
        * Tühi meetod hiireliidese realiseerimiseks.
        */
        public void mouseClicked(MouseEvent e){}
        /**
        * Tühi meetod hiireliidese realiseerimiseks.
        */
        public void mouseEntered(MouseEvent e){}
        /**
        * Tühi meetod hiireliidese realiseerimiseks.
        */
        public void mouseExited(MouseEvent e){}
        /**
        * Lõuend tegelikuks joonistamiseks.
        */
        class JoonistusAla extends Canvas{
           /**
           * Kuvatakse nii kursor, märgistatud ala kui
           * kõige peale helikõver ise.
           */
           public void paint(Graphics g){
             if(formaat==null){return;}
             g.setColor(margistusvarv);
             g.fillRect(
               (int)((algusKaader-joonistuseAlgKaader)*piksleidKaadriKohta), 0,
               (int)((loppKaader-algusKaader)*piksleidKaadriKohta), getHeight()
             );
             g.drawLine(  //kursor
              (int)((algusKaader-joonistuseAlgKaader)*piksleidKaadriKohta),
                 getHeight()/2,
              (int)((algusKaader-joonistuseAlgKaader)*piksleidKaadriKohta), getHeight()
             );
             if(andmed!=null){
                g.setColor(joonistusvarv);
                if(formaat.getChannels()==1){
                  joonistaKanal(g, 0, 0, 0, getWidth(), getHeight());
                } else {
                  joonistaKanal(g, 0, 0,
                     0, getWidth(), getHeight()/2);
                  joonistaKanal(g, 1, 0, getHeight()/2, getWidth(), getHeight()/2);
                }
             }
        }
   }
}
```

Redaktoriklassil enesel juurde tulnud mitmebaidiliste helide sisselugemisvõimalus, samuti lõigu mängimine korduvalt. Lisaks leiduvad nüüd koodi juures Javadoci-stiilis kommentaarid, mis teevad mahu küll pikemaks, kuid peaksid aitama meelde tuletada, milleks miski koodilõik mõeldud on.

Andmete failist sisselugemise puhul on tähtsaimaks teada, mitme baidi jagu tuleb mälus ruumi varuda. Vormingu käest on võimalik küsida kaadri suurus ning helisisendvoo käest kaadrite arv. Nende korrutis annabki heli hoidmiseks tarvilike baitide arvu. Edasi pole muud kui andmed ühe read-käskluse abil sisse lugeda.

```
andmed=new byte[formaat.getFrameSize()*(int)sisse.getFrameLength()];
sisse.read(andmed);
ala.seaAndmed(andmed, formaat);
```

Kui märgistatud lõik juhtub olema lühemapoolne, siis ühekordne kuulamine tekitab vaid tunde, et "oli midagi". Täpsemaks mõistmiseks aga tuleb lõiku vähemasti paar-kolm korda kuulata. Eriti hea aga, kui õnnestub lõik pidevalt mängima panna ning siis muutusi tehes pidevalt jälgida, mis operatsioonide tulemusena muutus. Kasutaja saab kordust määrata märkeruudu kaudu.

```
Checkbox loiguKordus=new Checkbox("Lõigu kordus");
```

Et omasoodu ja pidevalt korduv lõik muud programmi kinni ei jooksutaks ning oleks võimalik mängimise ajal maha võtta näiteks kordust tähistava märkeruudu märgistust, selleks peab korduv mängimine paiknema omaette lõimes. Et redaktori juures praegu muid omaette töötamist nõudvaid lõike pole, siis võis lõigu paigutada rakenduse enese run-meetodisse ning eraldi lõimel paluda seda jooksutada.

```
    if(e.getSource()==mangiLoik){new Thread(this).start(); }
```

Meetod run ise sarnaneb terve ploki mängimise näitega. Vaid juures on määratlus, kust alates ning kuhu maani anmeid kanalisse saatma peab. Ning ümber tsükkel, mis hoolitseb, et märgitud ruudu puhul lõiku muudkui korrataks ja korrataks.

```
  public void run(){
    try{  //Kontrollib, et opsüsteemilt on võimalik kanal küsida.
     kanal=(SourceDataLine)AudioSystem.getLine(
      new DataLine.Info(SourceDataLine.class, formaat, AudioSystem.NOT_SPECIFIED)
     );
     kanal.open();
     kanal.start();
     do{
       kanal.write(andmed, ala.algKaader(), ala.loppKaader()-ala.algKaader());
     }while(loiguKordus.getState());
     kanal.drain();
     kanal.close();
    }catch(Exception ex){
      ex.printStackTrace();
    }
  }
```

Ning kommenteeritud redaktori kolmanda versiooni kood tervikuna.

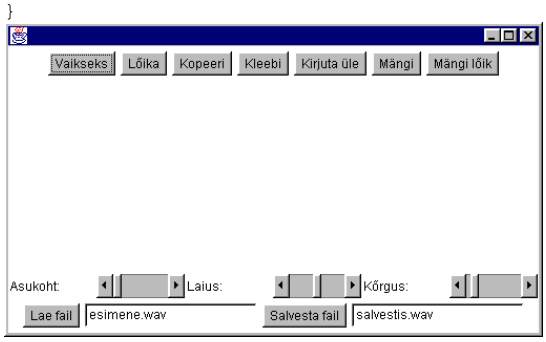
```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.sound.sampled.*;

/**
* Redaktori põhiklass. Suudab näidata, redigeerida ja
* salvestada PCM-kodeeringus (nt. wav) 8 ja 16 bitise heli
* ning 1 või kahe kanaliga helifaile.
*/
public class Digiheliredaktor3 extends Panel implements ActionListener, Runnable{
  /**
  *  Mängimis- ja töötlemiskõlbulikud andmed.
  *  Sisu tarvitatakse vastavalt sisseloetud
  *  formaadile.
  */
  byte[] andmed;
  /**
  * Abipuhver kopeerimise ja lõikamise tarbeks.
  */
  byte[] puhver;
  /**
  * Sisseloetud faili formaat. Kirjeldus näiteks
  * PCM_SIGNED, 22050.0 Hz, 16 bit, stereo, little-endian, audio data
  */
  AudioFormat formaat;
  /**
  * Kopeerimisnupp. Märgitud andmed kopeeritakse puhvrisse.
  */
  Button kopeeri=new Button("Kopeeri");
  /**
  * Lõikamisnupp.
  */
  Button loika=new Button("Lõika");
  /**
  * Kleepimisnupp.
  */
  Button kleebi=new Button("Kleebi");
  /**
  * Ülekirjutusnupp. Puhvris olevad andmed kirjutatakse
  * andmeploki peale kursorist alates. Heli pikkus ei muutu.
  */
  Button kirjutaYle=new Button("Kirjuta üle");
  /**
```

```
* Märgitud piirkonnas asendatakse kvantide väärtused nullidega.
*/
Button tyhjenda=new Button("Vaikseks");
/**
* Kogu mälus oleva andmeploki sisu mängitakse inimesele korra ette.
*/
Button mangi=new Button("Mängi");
/**
* Märgistatud ploki sisu mängitakse kuulajale.
*/
Button mangiLoik=new Button("Mängi lõik");
/**
* Märkeruudu seesolek jätab märgitud mängitava lõigu kordama.
* Kordamine lõppeb ruudu vabastamisel.
*/
Checkbox loiguKordus=new Checkbox("Lõigu kordus");
/**
* Ala sisseloetud helifaili andmete näitamiseks, märgistamiseks
* ning muutuste näitamiseks.
*/
Margistusala3 ala=new Margistusala3();
/**
* Faili laadimisnupp
*/
Button lae=new Button("Lae fail");
/**
* Laetava faili nimi.
*/
TextField tfLae=new TextField("esimene.wav", 20);
/**
* Faili salvestusnupp. Kodeering jääb samaks kui laadimisel.
*/
Button salvesta=new Button("Salvesta fail");
/**
* Salvestatava faili nime sisestus.
*/
TextField tfSalvesta=new TextField("salvestis.wav", 20);
/**
* Kanal kasutaja kuulatava heli teele saatmiseks.
*/
SourceDataLine kanal;
/**
* Paigutuse ja kuularite paika sättimine.
*/
public Digiheliredaktor3(){
  Panel p=new Panel();
  p.add(tyhjenda);
  p.add(loika);
  p.add(kopeeri);
  p.add(kleebi);
  p.add(kirjutaYle);
  p.add(mangi);
  p.add(mangiLoik);
  p.add(loiguKordus);
  Panel alapaneel=new Panel();
  alapaneel.add(lae);
  alapaneel.add(tfLae);
  alapaneel.add(salvesta);
  alapaneel.add(tfSalvesta);
  setLayout(new BorderLayout());
  add(p, BorderLayout.NORTH);
  add(ala, BorderLayout.CENTER);
  add(alapaneel, BorderLayout.SOUTH);
  loika.addActionListener(this);
  kopeeri.addActionListener(this);
  kleebi.addActionListener(this);
  kirjutaYle.addActionListener(this);
  tyhjenda.addActionListener(this);
  lae.addActionListener(this);
  salvesta.addActionListener(this);
  mangi.addActionListener(this);
  mangiLoik.addActionListener(this);
}

/**
* Sündmuse valik vastavalt nupuvajutusele.
*/
public void actionPerformed(ActionEvent e){
  if(e.getSource()==lae){laeFail(); }
  if(e.getSource()==salvesta){salvestaFail(); }
  if(e.getSource()==tyhjenda){tyhjendaLoik(); }
  if(e.getSource()==loika){loikaLoik(); }
  if(e.getSource()==kopeeri){kopeeriLoik(); }
  if(e.getSource()==kleebi){kleebiLoik(); }
```

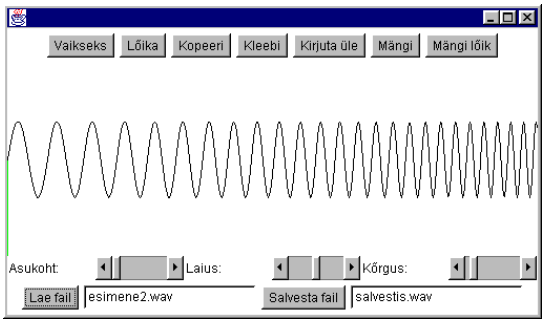
```
    if(e.getSource()==kirjutaYle){kirjutaLoikYle(); }
    if(e.getSource()==mangi){mangi(); }
    if(e.getSource()==mangiLoik){new Thread(this).start(); }
  }

  /**
  *  Faili mällu lugemine. Andmete pikkus aimatakse kaadri suuruse ning
  *  kaadrite arvu järgi. Formaat jäetakse meelde.
  */
  public void laeFail(){
   try{
    AudioInputStream sisse=AudioSystem.getAudioInputStream(new File(tfLae.getText()));
    formaat=sisse.getFormat();
    andmed=new byte[formaat.getFrameSize()*(int)sisse.getFrameLength()];
    sisse.read(andmed);
    ala.seaAndmed(andmed, formaat);
    System.out.println(formaat);
   }catch(Exception viga){
     viga.printStackTrace();
   }
  }

  /**
  * Helifaili talletamine kettale. Vorming ja parameetrid
  * jäävad endiseks, kasutaja sai sisu lisada, muuta või lühendada.
  */
  public void salvestaFail(){
   try{
      AudioInputStream ais=new AudioInputStream(
       new ByteArrayInputStream(andmed), formaat, andmed.length
     );
    AudioSystem.write(ais, AudioFileFormat.Type.WAVE, new File(tfSalvesta.getText()));
   }catch(Exception viga){
     viga.printStackTrace();
   }
  }


  /**
  * Märgitud lõigu kopeerimine mällu ning eemaldamine mängitavast jadast.
  * Kuna märgistusalas hoolitsetakse, et märgistus algab ja
  * lõpeb alati kaadri esimese baidiga, siis õnnestub siin
  * vastavate algKaadri ja lõppKaadri vaheliste baitide töötlemine
  * sõltumata kanalite ning kvandis asuvate baitide arvust.
  */
  public void loikaLoik(){
    byte[] uus=new byte[andmed.length-(ala.loppKaader()-ala.algKaader())];
    puhver=new byte[ala.loppKaader()-ala.algKaader()];

    System.arraycopy(andmed, ala.algKaader(), puhver, 0, (ala.loppKaader()-
ala.algKaader()));
    System.arraycopy(andmed, 0, uus, 0, ala.algKaader());
    System.arraycopy(andmed, ala.loppKaader(), uus,
                     ala.algKaader(), andmed.length-ala.loppKaader());
    andmed=uus;
    uuendaAla();
  }


  /**
  * Lõigu tühjendus, kõikide seal asuvate baitide väärtuseks
  * antakse 0.
  */
  public void tyhjendaLoik(){
    for(int i=ala.algKaader(); i<=ala.loppKaader(); i++){
      andmed[i]=(byte)0;
    }
    uuendaAla();
  }


  /**
  * Märgitud lõigu andmed kopeeritakse puhvrisse.
  * Mängitava lõigu sisu ei muutu.
  */
  public void kopeeriLoik(){
    puhver=new byte[ala.loppKaader()-ala.algKaader()];
    for(int i=0; i<puhver.length; i++){
      puhver[i]=andmed[ala.algKaader()+i];
    }
  }

  /**
```

```
* Puhvrist lisatakse kursori kohale vahele lõik. Andmete
* hoidmiseks luuakse uus massiiv ning sinna paigutatakse
* tulemus sobival kujul.
*/
public void kleebiLoik(){
  if(puhver==null)return;
  byte[] uus=new byte[
     andmed.length+puhver.length-(ala.loppKaader()-ala.algKaader())];
  System.arraycopy(andmed, 0, uus, 0, ala.algKaader());
  System.arraycopy(puhver, 0, uus, ala.algKaader(), puhver.length);
  System.arraycopy(andmed, ala.loppKaader(), uus,
            ala.algKaader()+puhver.length, andmed.length-ala.loppKaader());
  andmed=uus;
  uuendaAla();
}

/**
* Joonistusala uuendamine pärast andmete muutust.
* Ala hoolitseb seeläbi juba ise enese pildi uuendamise eest.
*/
void uuendaAla(){
  ala.seaAndmed(andmed, formaat);
}


/**
* Puhvris olev lõik kirjutatakse kursorist alates
* algse heli andmete asemele.
*/
public void kirjutaLoikYle(){
  if(puhver==null)return;
  for(int i=0; i<puhver.length; i++){
    andmed[ala.algKaader()+i]=puhver[i];
  }
  uuendaAla();
}


/**
* Mängitakse mälus olevad andmed kogu pikkuses.
*/
public void mangi(){
  try{  //Kontrollib, et opsüsteemilt on võimalik kanal küsida.
   kanal=(SourceDataLine)AudioSystem.getLine(
    new DataLine.Info(SourceDataLine.class, formaat, AudioSystem.NOT_SPECIFIED)
   );
   kanal.open();
   kanal.start();
   kanal.write(andmed, 0, andmed.length);
   kanal.drain();
   kanal.close();
  }catch(Exception ex){
    ex.printStackTrace();
  }
}


/**
* Eraldi lõimes lükatakse mängima märgitud lõik.
* Kui märkeruut sees, siis jäädakse kordama, muul
* juhul kõlab lõigu sisu ühe korra.
*/
public void run(){
  try{  //Kontrollib, et opsüsteemilt on võimalik kanal küsida.
   kanal=(SourceDataLine)AudioSystem.getLine(
    new DataLine.Info(SourceDataLine.class, formaat, AudioSystem.NOT_SPECIFIED)
   );
   kanal.open();
   kanal.start();
   do{
     kanal.write(andmed, ala.algKaader(), ala.loppKaader()-ala.algKaader());
   }while(loiguKordus.getState());
   kanal.drain();
   kanal.close();
  }catch(Exception ex){
    ex.printStackTrace();
  }
}

/**
* Redaktori käivitus käsurealt.
*/
public static void main(String[] argumendid){
```

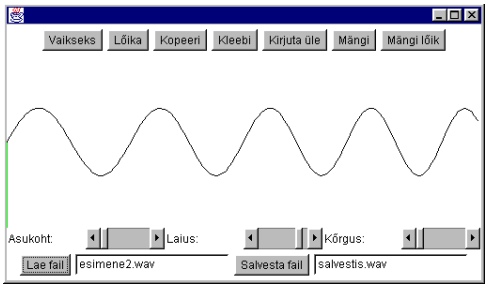
```
    Frame f=new Frame();
    f.add(new Digiheliredaktor3());
    f.setSize(400, 300);
    f.setVisible(true);
  }
}
```



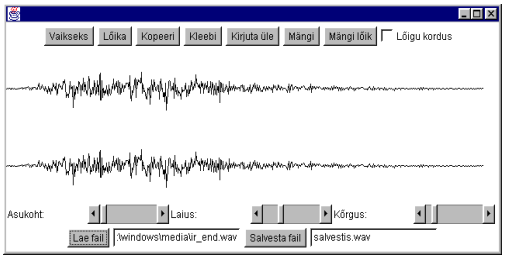
Enne faili avamist



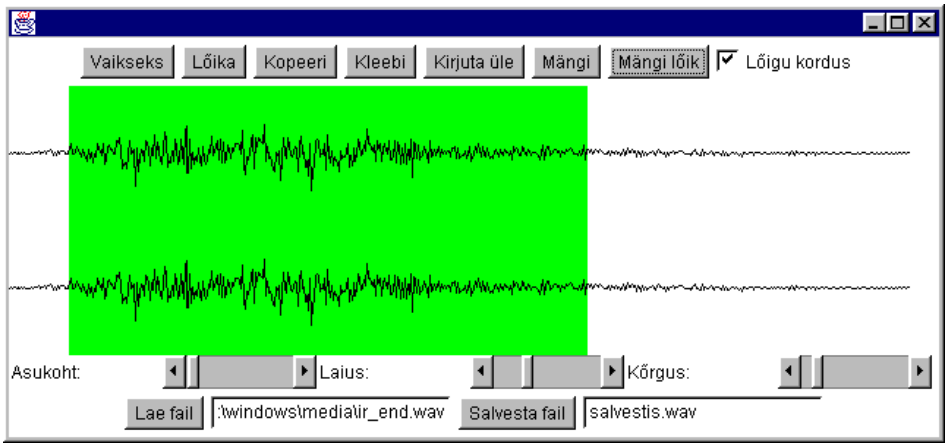
Kõrgenev ehk tiheneva sagedusega heli



Venitatud laiusega lõik



Heli kahel kanalil



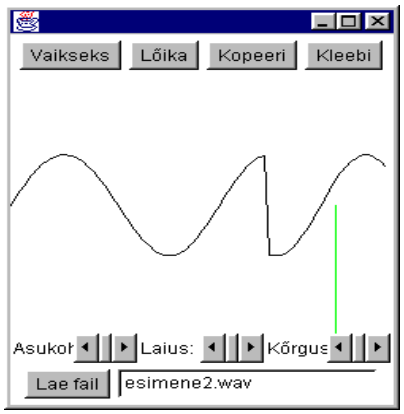
Korduse mängimine

```
D:\arhiiv\konspektid\gm\naited>java Digiheliredaktor3
PCM_SIGNED, 22050.0 Hz, 16 bit, stereo, little-endian, audio data
```

Heli väljatrükitud formaat.

Kommenteeritud rakendus pole veel kaugeltki täiuslik, kuid ega eesmärgiks polegi ühe järjekordse helitöötlusprogrammi loomine, milliseid juba nii vaba kui kommertstarkvara hulgas mitmeid kätte saada on. Pigem tuleb ise kirjutada selliseid lõike, mis olemasolevat lindistust soovitud suunas analüüsivad või siis kasutaja toimetuste saateks sobilikke helisid kokku panevad.

Nagu ehk katsetuste juures kuulda võisite, kippusid lõikamise ja kleepimise juures krõpsud sisse tulema. Põhjuseks, et nõnda lõikamisel ei tea baite kopeeriv koodilõik helikõvera kujust midagi ning sujuva joone sisse tekivad järsud murdekohad.



Krõpsuga koht

Üheks krõpsude vähendamise võimaluseks on kokkuliidetavad pooled veidi üksteise peale paigutada ning eelnev heli mõne võnke jooksul summutada ning järgnev heli sama aja sees vaikusest paari võnkega üles võimendada.

Kui kokku liidetavad helid on ligikaudugi sama amplituudi ja võnkesagedusega, siis võib õnnestuda võnkumised kokku ühendada samas faasis ning nii silmade kui kõrvade jaoks tundub, et helid oleksid nagu algusest peale kokku kuulunud.

Kokkuvõte

Java muusikavõimalused on tasapisi laienenud ning alates versioonist 1.3 õnnestub nii MIDI vahenditega orkestrilugusid kokku panna kui ise digitaalhelisid luua ja töödelda. Nagu mujal, nii ka siin pole programmeerimiskeel imevahend, selle kaudu lihtsalt õnnestub arvuti vastavatele seadmetele ligi pääseda ning soovitud helisid salvestada või kuuldavale tuua. Muusikalised ja matemaatilised tagamaad tuleb ikka enesel läbi mõelda ning pärast esmavahenditega tutvumist tulebki rakenduse loomisel enam sellele pühenduda. Lugudele saate koostamisel tuleb mõelda harmoonia ning taustalõikude peale neid sobivasse helistikku paigutades. Olenevalt muusikastiilist saab sageli kolme põhiduuriga enamikuga soovitust toime ning alles ilustuste ja kaunistuste juures tuleb muud oskused ja vahendid meelde tuletada ning arvutile selgeks teha. Olemasolevat noodifaili muutes tuleb arvestada, millistel radadel asuvate häältega me mida ette võtta soovime.

Digitaalhelis tuleb helilaine kuju ise välja arvutada. Nõnda kulub küll juba lihtsate häälte tekitamiseks paras kogus matemaatikat ning vähegi ilusamate helide loomiseks tuleb hulga arvutada ning tarkadest raamatutestki tarkust juurde ammutada, kuid põhimõtteliselt on võimalik luua või olemasolevatest kokku panna pea iga heli, mis vähegi ettekujutatav võiks olla.

Ülesandeid

Heli kõrgus

- Mängi signaali sagedusega 440 Hz.
- Pane signaali kõrgus sõltuma hiire asukohast ekraanil.
- Hoolitse, et kõrguse muutumine oleks sujuv.
- Korraga kõlab kaks heli. Ühe kõrgus sõltub hiire x- ning teine y-koordinaadist.
- Salvesta loodud heli faili.

Helifaili muundamine

- Salvesta helifail kaks korda üksteise järele.
- Salvesta helifail algsest poole vaiksemalt.
- Salvesta helifail iseenesega kajanihkega.

Lõppsõna

Eelnevatel lehekülgedel pole kirjas kaugeltki kõik Java ja graafika ja muusikaga seonduv, kuid märgatavale osale tekkivatest ideedest peaks siinsest kirjutisest omale teostusaluse leidma. MIDI poolest tasub kokku võtta kogu eneses leiduv muusikaalane fantaasia ning lihtsa taustaviiuli asemel terve orkester viisile saatjaks välja mõelda. Rääkimata lihtsatest ja ilusatest kõllidest, mida õnnestub oma rakendusele kasutaja meeleolu ilmestamiseks juurde lisada. Digitaalhelinäidetest peaks piisama, et akustikaõpiku abil lihtsast kajast või valjenemisest tunduvalt keerulisemaidki efekte kokku panna.

Graafika vallas pean tähtsaks tegelikest nähtustest koostatud arvutimudeleid, mis võimaldavad nii näidata kui katsetada olukordi, mida käepäraste vahenditega tülikas või suisa võimatu on luua. Olgu siis tegemist õpetajaga koolitunnis, autojuhiga liikluses või piloodiga taeva all. Kel piltide ilusama kujundusega või sealt info eraldamisega vaja rinda pista, sel tuleb millalgi lähemalt tutvuda Java Advanced Imaging lisapaketiga. Küllalt palju aga õnnestub ka ise „põlve otsas“ pildist eraldatud piksleid uurides ja muutes kokku panna. Video töötlejad ja ülekandjad vajavad Java Media Frameworki, kus suur osa vajalikku tööd juba ära tehtud, piisab vaid öelda, mida ja kuidas näidata vaja.

Eraldi tähelepanu väärib graafika mobiilirakenduste juures. Viimase paari aastaga on Java loojate unistused selles valdkonnas hakanud täituma ning võib loota, et mõne aasta jooksul jõutakse üksikutest töötavatest vidinatest ka lahendusteni, mis paljude kasutajatele tarvilikke lisandusi miniseadmetesse toovad.

Kes on kõik eelnevad näited ja seletused läbi lugenud, mõelnud, proovinud ja enese jaoks edasi arendanud, sel kuluks veidi puhata, tagasi vaadata ning kogunenud killukestest enesele edasiseks tervik mõelda, millele edaspidi kindlalt toetuda, nii et julgeb pea iga siinse valdkonnaga seotud toimetuse ette võtta. Kui loetud vähem, aga kasvõi mõni seletuse või koodi lõik on aidanud oma tegemisi edasi viia ehk teadmisi korrastada, ka siis on õpitust kasu olnud.

Tänan lugemast!