Keele võimalused

Arhiivid, programne koodi uuring ja testimine.

Jar-arhiivid

Kuude ja aastatega koguneb programmilõike, millest ka uute rakenduste koostamisel kasu on. Lühematel juhtudel saab need uue valmiskoodi sisse kopeerida, kuid nõnda lähevad kapseldumisest saadavad hüved kaduma. Avastanud kord lähtefailis vea, tuleb ka kõik muud lõigud läbi käia, kus sama koodijupiga tegemist on. Kui kõik programmid kirjutatakse ühes kataloogis, siis pole suurt muret – klassi nime kaudu saab otse ka koodile ligi. Vähegi suurema programmeerimise korral tuleb koodifaile aga nii hulganisti, et neid üheskoos hoides läheks pilt väga kirjuks ning samuti ei õnnestuks eri ülesannete tarvis loodud koodilõike lahus hoida.

Edaspidi mitmel pool kasutamist leidvate klasside failid võib kokku pakkida ühte Jar-arhiivi, mida siis vajadusel mujalgi tarvitada annab. Näitena klass meetodiga nimega tühja raami avamiseks:

```
C:\kodu\jaagup\0108\k1>type ArhiiviKlass.java
import java.awt.*;
public class ArhiiviKlass{
  public static void avaRaam(String nimi){
    Frame f=new Frame(nimi);
    f.setSize(200, 200);
    f.setVisible(true);
  }
}
```

ning teine klass loodud meetodi katsetamiseks.

```
public class ArhiiviKasutaja{
  public static void main(String argumendid[]){
    ArhiiviKlass.avaRaam("Tervitusaken");
  }
}
```

Traditsioonilisel moel kompileerides ja käivitades ongi tulemuseks avatud aken.

```
C:\kodu\jaagup\0108\k1>javac Arhiivi*.java

C:\kodu\jaagup\0108\k1>java ArhiiviKasutaja
```

Soovides aknaavamismeetodiga klass arhiivi paigutada, tuleb anda korraldus

```
C:\kodu\jaagup\0108\k1>jar cf ArhiiviKlassid.jar ArhiiviKlass.class
```

Selle tulemusena luuakse uus arhiiv nimega ArhiiviKlassid.jar. Võti c tähendab uue arhiivifaili loomist (create), f – failinime. Kui sellenimeline arhiiv oleks olemas olnud, siis see kirjutataks üle.

Nüüd võib kompileeritud klassi lahtipakitud kuju maha kustutada,

```
C:\kodu\jaagup\0108\k1>del ArhiiviKlass.class
```

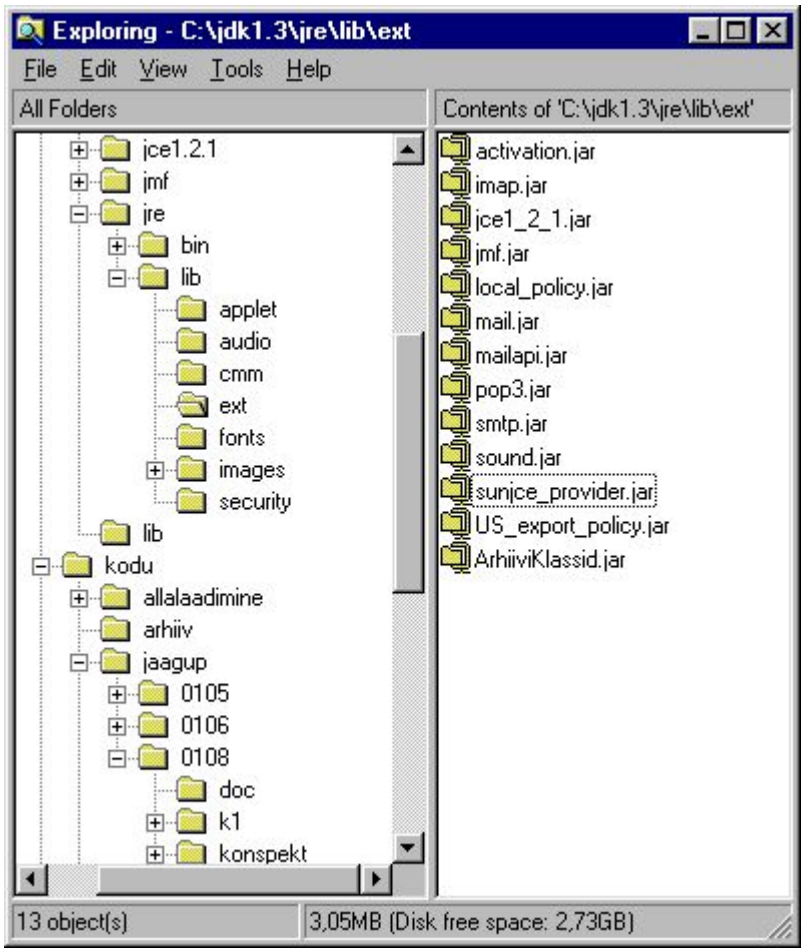
sest andmeid saab ka failist lugeda.

```
C:\kodu\jaagup\0108\k1>java -cp ArhiiviKlassid.jar;. ArhiiviKasutaja
```

Semikoolon ning punkt arhiivi nime järel tähendab, et klasside otsinguteele pannakse lisaks arhiivile ka jooksev kataloog. Muidu jääks käivitamiseks tarvilik ArhiiviKasutaja leidmata, sest seda arhiivis pole.

```
C:\kodu\jaagup\0108\k1>java -cp ArhiiviKlassid.jar ArhiiviKasutaja
Exception in thread "main" java.lang.NoClassDefFoundError: ArhiiviKasutaja
```

Kui loodud arhiivi läheb mitmel pool vaja, siis võib selle paigutada kohta, kust otsitav klass kogu virtuaalmasina piires üles leitakse. Kohaks on Java installeerimiskohast alanev kataloogipuu jre\lib\ext.



Kui nüüd raamiloomiskäsku vajatakse

```
C:\kodu\jaagup\0108\k1>java ArhiiviKasutaja
```

, siis leitakse see vabalt ligi pääsetavast kataloogist paiknevast arhiivist välja ning tulemusena võib jälle loodud raami näha.

Arhiivi võib panna ka käivitatava klassi.

```
C:\kodu\jaagup\0108\k1>jar cf ArhiiviKlassid.jar Arhiivi*.class
```

Nii võib vajaduse korral ka võõras masinas oma arhiivi sees paikneva programmi välja kutsuda.

```
C:\kodu\jaagup\0108\k1>java -cp ArhiiviKlassid.jar ArhiiviKasutaja
```

Arhiivist on kasu ka veebilehe puhul. Atribuudiga archive saab määrata, millisest failist andmed tõmmata. Sel juhul pääseb brauser hulga üksikute klasside kohale sikutamisest ning lehe laadimine võib vähem aega võtta.

```
<html><head><title>Arhiivirakend</title></head>
<body>
 <h2>Rakend arhiivist</h2>

 <applet code="ArhiiviRakend" archive="ArhiiviKlassid.jar"
   height="10" width="10">
 </applet>

</body></html>
```

```
import java.applet.Applet;
public class ArhiiviRakend extends Applet{
  public ArhiiviRakend(){
    ArhiiviKlass.avaRaam("Rakendi aken");
  }
}
```

Paketid

Üle maailma luuakse Java klasse hulgaliselt ning üsna kindlasti satuvad mõnede nimed nendest kokku. Mõistlikke arusaadavaid nimetusi on lihtsalt piiratud hulk. Selle tarvis on Java keelde loodud objektidest/klassidest veel ühe taseme võrra kõrgem grupp: paketid. Standardpakettidest oleme kindlasti kasutanud java.lang-i, java.io või java.awt vahendeid. Selliseid klasside komplekte aga saab rahus ise juurde luua. Põhikomplekti kuuluvad paketid on pandud algama nimega java, põhikomplekti laiendused sõnaga javax. Ülejäänud pakkettide nime algusse aga soovitatakse paigutada neid välja töötava kompanii veebiaadress. Pea igal pakette looval asutusel või isikul on oma väljund veebis ning juba nimede jagamisel hoolitsetakse selle eest, et sama nime alla mitut omanikku ei satuks. Domeeni sees aga tuleb loojatel juba isekeskis hoolitseda, et nimed kattuma ei hakkaks. Nõnda panen pedagoogikaülikoolis loodud paketi nimeks veebiaadressi järgi ee.tpu. Failisüsteemis paketi nimed kattuvad kataloogi nimega, nõnda tuleb siis loodud failid paigutada kompileerimis/käivituskataloogi alamkataloogi ee\tpu.

```
C:\kodu\jaagup\0108\k1>type ee\tpu\Tervitaja.java
package ee.tpu;

public class Tervitaja{
  public static void tervita(){
    System.out.println("Tervist");
  }
}
```

Faili algusse tuleb kirjutada, millise paketi klassiga tegemist on. Et paketist kasutatakse sageli klasse mõne muu programmi töö tarbeks, siis ka siinses näites pole main-meetodit, kust töö käima võiks minna. Nii nagu java.awt.Button'it võime oma töösse lisada, nii saab ka vastloodud klassi oskusi oma hüvanguks kulutada.

```
C:\kodu\jaagup\0108\k1>type Paketikatse.java
import ee.tpu.*;

public class Paketikatse{
  public static void main(String argumendid[]){
    Tervitaja.tervita();
  }
}
```

Faili alguses import-lause hoolitseb, et seal paketis asuvatele klassidele mugavalt ligi pääseks. Programm läheb tööle nagu tavaline muugi Java rakendus.

```
C:\kodu\jaagup\0108\k1>javac Paketikatse.java

C:\kodu\jaagup\0108\k1>java Paketikatse
Tervist
```

Käivituva klassi võib ka paketi sisse panna.

```
C:\kodu\jaagup\0108\k1> type ee\tpu\Alustus.java
package ee.tpu;

public class Alustus{
  public static void main(String argumendid[]){
    Tervitaja.tervita();
  }
}
```

Sellisel juhul saab paketile ligi pääsedes mugavalt seal paikneva programmi käivitada.

```
C:\kodu\jaagup\0108\k1>java ee.tpu.Alustus
```

```
Tervist
```

Paketi klassid võib lahedasti üheks arhiiviks kokku pakkida.

```
C:\kodu\jaagup\0108\k1>jar cf tekstipakett.jar ee\tpu\*.class
```

Sellisena piisab installeerimiseks vaid ühe faili sobivasse kohta kopeerimisest ning töö võibki alata.

```
C:\kodu\jaagup\0108\k1>java -cp tekstipakett.jar ee.tpu.Alustus
Tervist
```

Kui tahta arhiivile üle kogu virtuaalmasina ligi pääseda, siis võib selle kopeerida kättesaadavasse jre\lib\ext kataloogi nagu eelmiseski näites.

Soovides oma Java-programmi võõrasse masinasse paigutada, peab omanik enamasti kopeerima sinna hulga faile ning lisaks teadma, millise klassi käivitamisel kogu lugu tööle hakkab. Jar-faili mainfest-osas saab määrata, millise klassi main-meetodist programmi käivitamist alustada tuleb.

```
C:\kodu\jaagup\0108\k1>type lisateave.txt
Main-Class: ee.tpu.Alustus
```

Kui arhiivi loomisel manifest tekstifailist lisada,

```
C:\kodu\jaagup\0108\k1>jar cmf lisateave.txt tekstipakett.jar ee\tpu\*.class
```

siis käivitamisel pannaksegi arhiiv niimoodi tööle, kuidas programmi kirjutaja seda soovinud on.

```
C:\kodu\jaagup\0108\k1>java -jar tekstipakett.jar
Tervist
```

Kes on Jar-failile assotsiatsiooni loonud (või on see vaikimisi tehtud), et käivitamisel lükatakse tööle java intepretaator jar-võtmega ning parameetriks antakse jar-arhiivi nimi, siis tundubki, et tegemist on isekäivituva jar-failiga.

Erindid

Veidi seletusi erindite loomise ja kasutamise kohta.

Probleemist teada andmiseks võime soovitud kohas välja heita erindi. Järgnevad käsud jäetakse täitmata kuni erind lendab virtuaalmasinast välja või püütakse kinni. Üldjuhul tuleb meetodi päises näidata throws-teatega, kui meetodist võib erindeid välja tulla.

```
public class Erind5{
  public static void main(String argumendid[]) throws Exception {
    int vanus=8;
    if(vanus<10)throw new Exception("Liiga noor");
    System.out.println("Tere tulemast pikamaajooksule");
  }
}
```

väljund

```
Exception in thread "main" java.lang.Exception: Liiga noor
        at Erind5.main(Erind5.java:4)
```

vastab täiesti ootustele. Klassi Erind4 neljandal koodireal saadeti lendu erind ning edaspidised käsud jäid täitmata.

Kui soovitakse heita erind, mille tekkimist ei pea deklareerima, siis tuleb kasutada RuntimeExceptioni või selle alamklassi. Enamjaolt kuuluvad RuntimeExceptioni alla eriolukorrad, mis võivad ette tulla väga paljudes kohtades (nt. jagamine nulliga, massiivi piiride ületamine) ning mida deklareerides peaks peaks siis pea kõikide meetodite juurde deklaratsioonid kirjutama. Nagu näha, siin kasutatakse RuntimeExceptioni ning main-meetodis pole tarvis üles tähendada, et throws Exception või throws RuntimeException. Samas kui sinna see siiski kirjutada, siis probleeme sellest ei tekiks.

```
public class Erind6{
  public static void main(String argumendid[]) {
    int vanus=8;
    if(vanus<10)throw new RuntimeException("Liiga noor");
    System.out.println("Tere tulemast pikamaajooksule");
  }
}
```

Jällegi väljund vastavalt ootustele

```
Exception in thread "main" java.lang.RuntimeException: Liiga noor
        at Erind6.main(Erind6.java:4)
```

, neljandal real avastati, et pikema jooksmise tarvis on vanust liialt vähe.

Omaloodud erind

Kui soovime selliseid eriolukordi teada anda ja nendele reageerida, milliseid standardvahendites kirjas pole, siis võime luua oma erindialamklassi. Sellisel võime kergemini reageerida vastavalt tekkinud probleemile. Ka erindi väljatrükil näidatakse, millisest klassist erind pärit on.

```
class VanuseErind extends Exception{}

public class Erind7{
  public static void main(String argumendid[]) throws VanuseErind{
    int vanus=8;
    if(vanus<10)throw new VanuseErind();
    System.out.println("Tere tulemast pikamaajooksule");
  }
}

/* väljund:
 Exception in thread "main" VanuseErind
        at Erind7.main(Erind7.java:4)
*/
```

Soovides erindi kinnipüüdjale anda selgituse probleemi kohta, on üheks võimaluseks katta üle erindi meetod getMessage().

```
class VanuseErind2 extends Exception{
  public String getMessage(){
    return "Vanus ei sobinud";
  }
}

public class Erind7a{
  public static void main(String argumendid[]) throws Exception{
    int vanus=8;
    if(vanus<10)throw new VanuseErind2();
    System.out.println("Tere tulemast pikamaajooksule");
  }
}
```

Nõnda jõuab koodi sisse kirjutatud selgitus veateatena ekraanile või võidakse seda muul moel veatöötluses arvestada.

```
Exception in thread "main" VanuseErind2: Vanus ei sobinud
         at Erind7a.main(Erind7a.java:4)
```

Kõige viisakam võimalus programmi sees omaloodud erindile teadet kaasa panna on luua erindile uus sõneparameetriga konstruktor mis omakorda ülemklassi vastava konstruktori välja kutsub. Ülemklass hooliseb, et getMessage teate välja annaks. Et loodud klassi veel omakorda ilusti laiendada annaks, tuleks siia ka parameetriteta konstruktor kirjutada, mis ülemklassi parameetriteta konstruktori välja kutsuks. Põhjus selles, et vaikimisi kutsutakse uue isendi loomisel alati välja ülemklassi parameetriteta konstruktor, kui parajasti käivitatava konstruktori esimese käsuna pole välja kutsutud mõni muu konstruktor. Kui juhtub aga, et ülemklassil parameetriteta konstruktorit pole ning mõnda muud ka välja ei kutsuta, siis antakse veateade. Siinse näite VanuseErind3 juures on parameetriteta konstruktori kirjutamisest loobutud, arvates et sellest erindist võib luua vaid programmeerija määratud teatega isendeid ning et teateta alamklasse siia enam ei looda.

```
class VanuseErind3 extends Exception{
  VanuseErind3(String teade){
    super(teade);
  }
}

class Erind7b{
  static void main(String argumendid[]) throws Exception{
    int vanus=8;
    if(vanus<10)throw new VanuseErind3("Pole veel 10. aastane");
    System.out.println("Tere tulemast pikamaajooksule");
  }
}

/* väljund:
Exception in thread "main" VanuseErind3: Pole veel 10. aastane
        at Erind7b.main(Erind7b.java:4)
*/
```

Lõpuplokk finally

Lisaks try-le ning cactch-i(de)le võib katsendile lisada finally-ploki, mis täidetakse sõltumata sellest, kas uuritavas piirkonnas tekkis probleeme või kas neid töödeldi. Sinna kirjutatakse enamasti käsud, mis tuleb alati täita. Näiteks voogude sulgemine või andmebaasiühenduse katkestamine, mis tuleb ressursside vabastamiseks ikka läbi viia, ükskõik, kas saabuvad andmed vastasid ootustele või mitte. Järgnevates näidetes saadetakse lõpuplokis tervitused kõigile kohalolijatele sõltumata sellest kui noored või vanad nad parajasti on. Samuti tegevus, mis oleks patt ära jätta.

```
class Erind8{
  static void main(String argumendid[]) throws VanuseErind{
    int vanus=8;
    try{
     if(vanus<10)throw new VanuseErind();
     System.out.println("Tere tulemast pikamaajooksule");
    } catch(VanuseErind v){
      v.printStackTrace();
    }
    finally {
      System.out.println("Tervitus igale kohalolijale");
    }
  }
}
```

Kõigepealt püüti catch-osas erind kinni ning trükiti selle andmed, edasi tervitati finally-plokis kohalolijaid.

```
/* väljund:
VanuseErind
         at Erind8.main(Erind8.java:5)
Tervitus igale kohalolijale
*/
```

Ka juhul, kui katsendiplokis püütakse return-i abil meetodist väljuda, ei pääse lõpuploki käivitamisest.

```
public class Erind8a{
  public static void main(String argumendid[]){
    int vanus=78;
    try{
     if(vanus>70)return;  //püütakse meetodist väljuda;
     if(vanus<10)throw new VanuseErind();
     System.out.println("Tere tulemast pikamaajooksule");
    } catch(VanuseErind v){
      v.printStackTrace();
    }
    finally {
      System.out.println("Tervitus igale kohalolijale");
    }
  }
}

/* väljund:
Tervitus igale kohalolijale
*/
```

Kui kõik tingimused sobivad ning probleeme ei teki, ka siis täidetakse lõpuplokk.

```
class Erind8b{
  static void main(String argumendid[]){
    int vanus=27;
    try{
     if(vanus<10)throw new VanuseErind();
     System.out.println("Tere tulemast pikamaajooksule");
    } catch(VanuseErind v){
      v.printStackTrace();
    }
    finally {
      System.out.println("Tervitus igale kohalolijale");
    }
  }
}

/* väljund:
Tere tulemast pikamaajooksule
Tervitus igale kohalolijale

*/
```

Erindi võib pärast töötlemist edasi saata sarnaselt throw käsuga nagu algselgi juhul. Sellist moodust läheb vaja, kui samale probleemile tuleb reageerida mitmes kohas.

```
public class Erind9{
  public static void main(String argumendid[]) throws VanuseErind{
    int vanus=8;
    try{
     if(vanus<10)throw new VanuseErind();
     System.out.println("Tere tulemast pikamaajooksule");
    } catch(VanuseErind v){
      v.printStackTrace();
      throw v;
    }
  }
}

/* väljund:
VanuseErind
        at Erind9.main(Erind9.java:5)
Exception in thread "main" VanuseErind
```

```
        at Erind9.main(Erind9.java:5)

*/
```

Ka siis ei pääse finally-ploki täitmisest.

```
public class Erind9a{
  public static void main(String argumendid[]) throws VanuseErind{
    int vanus=8;
    try{
     if(vanus<10)throw new VanuseErind();
     System.out.println("Tere tulemast pikamaajooksule");
    } catch(VanuseErind v){
      v.printStackTrace();
      throw v;
    } finally {
      System.out.println("Tervitus igale kohalolijale");
    }
  }
}

/* väljund:
VanuseErind
        at Erind9a.main(Erind9a.java:5)
Tervitus igale kohalolijale
Exception in thread "main" VanuseErind
        at Erind9a.main(Erind9a.java:5)
*/
```

Kloonimine

Objektide puhul rõhutatakse kolme eristatavat omadust: tüüp, andmed ja identiteet. Tüübi alla kuuluvad klassi kirjeldamisel loodud käsklused ehk meetodid ning väljade ehk muutujate loetelu. Andmed on igal objektil omad: olgugi et mõlemad isendid võivad olla tüübist Punkt, nende x-i ja y-i väärtused on üldjuhul erinevad. Ning isegi, kui mõne punkti koordinaadid peaksid teise punkti koordinaatidega kattuma, ei ole tegemist sama objektiga, sest neil on erinev identiteet.

Soovides isendist luua koopiat, tuleb teha koopia kõigist tema väljadest. Üheks võimaluseks on koostada koopia loomiseks eraldi käsklus. Seal luua konstruktori abil uus sama tüüpi isend. Anda käskude ja parameetrite abil uuele isendile soovitud väärtused ning siis returni abil tagastada osuti uuele objektile.

Vaeva vähendamiseks on loodud liides Cloneable, mille abil on lubatud objektidest luua "mehhaaniline koopia". See tähendab, et uude koostatavasse objekti kantakse üle lihtsalt kõik vana objekti väljade väärtused. Vastava toimingu läbiviijana saab kasutada kõikide klasside ülemklassi Object protected-juurdepääsuga käsklust nimega clone, mis loobki just uue objekti ja kopeerib sinna kõikide väljade väärtused.

```
class Kloon1 implements Cloneable{
  String nimi;
  int mass;
  public Object clone(){
    Object o=null;
    try{
      o=super.clone();
    }catch(CloneNotSupportedException e){}
    return o;
  }
}
```

Et liides Cloneable määrab meetodi clone tagastustüübiks Object, siis tegelikuks kasutamiseks tuleb uus eksemplar tüübimuundusega sobivasse tüüpi ehk algse isendiga samasse tüüpi määrata.

Järgnevalt võibki näha, kuidas algsest eksemplarist kloon luuakse nõnda, et kõik väärtused samaks jäävad, kuid nendele eraldi mälupesad eraldatakse. Ning kuna pärast kloonimist on tegemist eraldi objektidega, siis massi muutmine ühe isendi juures ei muuda teise isendi massi.

```
public class Kloon1Test{
   public static void main(String[] argumendid){
     Kloon1 k1=new Kloon1();
     k1.nimi="Dolly";
     k1.mass=80;
     Kloon1 k2=(Kloon1)k1.clone();
     k2.mass=90;
     System.out.println(k1.mass+" "+k2.mass);
   }
}

/*
D:\kodu\0312>java Kloon1Test
80 90
*/
```

Kui klassil on Cloneable-liidese kaudu kord juba kloonimine lubatud, siis edasi on ka kõik vastava klassi alamklasside isendid kloonitavad. Et Kloon1 sai eelmises näites kloonitavaks muudetud, siis vastav õigus kehtib ka Kloon3-e isendite kohta.

```
class Kloon3 extends Kloon1{
  int vanus;
}
```

Ehkki nüüd kloonitakse isendit tüübist Kloon3, osatakse ülemklassi meetodite väljakutse abil ikkagi kõikidest väljadest koopiad teha.

```
public class Kloon3Test{
  public static void main(String[] argumendid){
    Kloon3 k=new Kloon3();
    k.mass=60;
    k.vanus=2;
    Kloon3 k2=(Kloon3)k.clone();
    System.out.println(k2.mass+" "+k2.vanus);
  }
}

/*
D:\kodu\0312>java Kloon3Test
60 2
*/
```

Süviti kloonimine.

Lihttüüpidega on asi selge: kui isendil olid väljad ning nendel väärtused, siis kloonimise puhul loodi uus isend, kus iga välja jaoks oli loodud uus mälupesa. Välja mälupesal paiknev väärtus ongi selle välja väärtus.

Struktuurtüüpidega on lugu keerulisem. Taoline väli võib viidata andmebaasiühendusele, avatud aknale või näiteks failile kusagil kaugel Internetis. Kui isendil ka on vastav väli küljes, siis välja kloonimine ei tee veel eraldi asetsevat objekti juurde. Tulemuseks on lihtsalt mõlema isendi küljes paiknevad osutid, mis näitavad samale tegelikule objektile. Kui nüüd ühe kloonitud isendi osuti kaudu taolise väljaspool asuva objekti parameetreid muuta, siis muutus paistab mõlemast

kloonimisega seotud isendist. Samuti nagu ühest aknast puuoksa külge seotud pekitükk talvel lindude söötmiseks paistab ka teisest aknast, sest tegemist on ikka ühe ja sama oksa ning pekiga. Ning kui pekk hakkab otsa lõppema, siis paistab see samuti mõlemast aknast kätte.

Mõnikord aga soovitakse, et kloonimise järel oleksid ka kummastki objektist viidatavad objektid teineteisest sõltumatud. Selleks tuleb ka alanevad objektid kloonida, mis mõnikord on võimalik, teinekord mitte. Siin näites püütakse kirjeldada sugupuud. Et isik ning vanemad on sama liiki ning vastav klass ise kloonitav, siis õnnestub vajadusel ka kogu sugupuu andmed kloonida. Kloonimisel tehakse kõigepealt koopia enesest ning siis ka kummastki esivanemast nende olemasolul.

```
      if(isa!=null){
        k.isa=(Kloon2)isa.clone();
      }
```

Et esivanem on samuti tüübist Kloon2, siis ka tema kloonimisel minnakse omakorda alanejaid esivanemaid kloonima.

Sarnaselt liigutakse alamelementide sisse ka väärtusi välja trükkides toString meetodi abil.

```
class Kloon2 implements Cloneable{
  String nimi;
  Kloon2 isa;
  Kloon2 ema;
  public Object clone(){
    Object o=null;
    try{
      o=super.clone();
      Kloon2 k=(Kloon2)o;
      if(isa!=null){
        k.isa=(Kloon2)isa.clone();
      }
      if(ema!=null){
        k.ema=(Kloon2)ema.clone();
      }
    }catch(CloneNotSupportedException e){}
    return o;
  }
  public String toString(){
    String s=nimi+"(";
    if(isa!=null){ s+=isa.toString(); }
    s+=", ";
    if(ema!=null){ s+=ema.toString(); }
    s+=")";
    return s;
  }
}
```

Testimisel luuakse kõigepealt väike sugupuu, kus peale Maasu leiduvad ka tema isa, ema ning isaisa. Edasi sugupuu kloonitakse ning uue puu puhul määratakse isaisa nimeks Puhvik.

```
public class Kloon2Test{
   public static void main(String[] argumendid){
     Kloon2 k1=new Kloon2();
     k1.nimi="Maasu";
     k1.isa=new Kloon2();
     k1.isa.nimi="Punik";
     k1.ema=new Kloon2();
     k1.ema.nimi="Maara";
     k1.isa.isa=new Kloon2();
     k1.isa.isa.nimi="Puhvel";
     System.out.println(k1);
     Kloon2 k2=(Kloon2)k1.clone();
     k2.isa.isa.nimi="Puhvik";
     System.out.println(k1);
     System.out.println(k2);
   }
}
```

Kui süviti kloonimist poleks toimunud, siis oleks mõlemal kloonil isaisa objekt sama ning isaisa nime muutmine ühe isendi kaudu muutnuks ta nime ka teise isendi poolt vaadates. Et aga kloonimine toimus süviti, siis klooniti ka vanavanemad ning nimemuutus ühe muutuja kaudu jättis algse vanaisa nime muutmata.

```
/*
D:\kodu\0312>java Kloon2Test
Maasu(Punik(Puhvel(, ), ), Maara(, ))
Maasu(Punik(Puhvel(, ), ), Maara(, ))
Maasu(Punik(Puhvik(, ), ), Maara(, ))
*/
```

Ülesandeid

* Koosta klass Punkt kahe koordinaadiga.
* Loo eksemplar, testi.
* Loo teine muutuja näitamaks samale eksemplarile. Muuda andmeid, testi väljatrükki mõlema muutuja kaudu.
* Realiseeri Punktil liides Cloneable, kata üle meetod clone loomaks koopia.
* Loo muutuja, mis näitaks esialgse punkti koopiale. Testi väljade muutmist, trüki tulemused ekraanile.

* Loo klass Kujund, mis sisaldab eneses nime ning Punktide nimistu. Klassile käsklused punktide lisamiseks ja muutmisks. Testi.
* Muuda Kujund Kloonitavaks. Kloonitakse nimistu, kuid veel mitte üksikuid punkte (nimistu käsk clone). Testi lisamist ja muutmist.
* Paranda koodi nõnda, et ka üksikud punktid kujundi nimistu sees kloonitaks.

Klasside uuring koodiga

Programmeerija üldjuhul teab, millised käsklused milliste parameetritega ta on oma koodi kirjutanud. Või kui täpselt ei tea, siis vaatab koodist või dokumentatsioonist järele. Kui aga pole lähtekoodi või dokumentatsiooni käepärast, samuti juhtudel, kui soovitakse automaatselt statistikat teha või mõningaid käsklusi testida, aitab välja võimalus programmikäskude abil olemasolevat klassi uurida. Järgnevas näites on võetud ette klass String ning küsitud välja kõikide sealsete meetodite nimed.

```
import java.lang.reflect.Method;
public class Klassiuuring1{
  public static void main(String[] argumendid){
    String s="Tere";
    Class c=s.getClass();
    System.out.println("Klassi nimi: "+c.getName());
    Method[] m=c.getMethods();
    System.out.println("Meetodid:");
    for(int i=0; i<m.length; i++){
      System.out.println("  "+m[i].getName());
    }
  }
}

/*
Klassi nimi:java.lang.String
Meetodid:
```

```
  valueOf
  valueOf
  valueOf
  valueOf
  valueOf
  valueOf
  valueOf
  valueOf
  valueOf
  copyValueOf
  copyValueOf
  wait
  wait
  wait
  getClass
  notify
  notifyAll
  hashCode
  compareTo
  compareTo
  equals
  toString
  length
  charAt
  getChars
  getBytes
  getBytes
  getBytes
  equalsIgnoreCase
  compareToIgnoreCase
  regionMatches
  regionMatches
  startsWith
  startsWith
  endsWith
  indexOf
  indexOf
  indexOf
  indexOf
  lastIndexOf
  lastIndexOf
  lastIndexOf
  lastIndexOf
  substring
  substring
  concat
  replace
  toLowerCase
  toLowerCase
  toUpperCase
  toUpperCase
  trim
  toCharArray
  intern
*/
```

Iga meetodi sisse saab ka mõnevõrra põhjalikumalt piiluda. Siin on vaadatud lisaks meetodite nimedele ka nende parameetrid koos tüüpidega. Samuti meetoditest välja heidetavate erinditüüpide loetelu.

```
import java.lang.reflect.Method;
public class Klassiuuring2{
  public static void main(String[] argumendid){
    String s="Tere";
    Class c=s.getClass();
    Method[] m=c.getMethods();
    System.out.println("Meetodid:");
    for(int i=0; i<m.length; i++){
      Class[] parameetrid=m[i].getParameterTypes();
      Class[] erindid=m[i].getExceptionTypes();
      Class tagastus=m[i].getReturnType();
      System.out.print(tagastus.getName()+" "+m[i].getName());
      for(int j=0; j<parameetrid.length; j++){
        System.out.print(" "+parameetrid[j].getName());
        //[I parameetri nimena tähistab näiteks täisarvude massiivi
      }
      for(int j=0; j<erindid.length; j++){
        System.out.println(" *"+erindid[j].getName());
      }
```

```
      System.out.println();
    }
  }
}
```

Väljund võib esiotsa ebatavaline tunduda, kuid kui dokumentatsioonist klassi Class meetodit getName() uurida, leia sealt, et massiivide tähistamiseks on omaette moodus olemas. Ning

```
java.lang.String valueOf [C int int
```

tähendab lihtsalt, et klassis leidub java.lang.String tüüpi objekti väljastav käsklus valueOf, mis saab parameetriteks tähemassiivi ning kaks täisarvu. Nii nagu [C tähistab tähemassiivi, nii tähistaks [[I kahemõõtmelist täisarvumassiivi.

```
/*
Meetodid:
java.lang.String valueOf [C int int
java.lang.String valueOf [C
java.lang.String valueOf java.lang.Object
java.lang.String valueOf long
java.lang.String valueOf boolean
java.lang.String valueOf char
java.lang.String valueOf int
java.lang.String valueOf float
java.lang.String valueOf double
java.lang.String copyValueOf [C int int
java.lang.String copyValueOf [C
void wait *java.lang.InterruptedException

void wait long int *java.lang.InterruptedException

void wait long *java.lang.InterruptedException

java.lang.Class getClass
void notify
void notifyAll
int hashCode
int compareTo java.lang.String
int compareTo java.lang.Object
boolean equals java.lang.Object
java.lang.String toString
int length
char charAt int
void getChars int int [C int
[B getBytes
[B getBytes java.lang.String *java.io.UnsupportedEncodingException

void getBytes int int [B int
boolean equalsIgnoreCase java.lang.String
int compareToIgnoreCase java.lang.String
boolean regionMatches int java.lang.String int int
boolean regionMatches boolean int java.lang.String int int
boolean startsWith java.lang.String
boolean startsWith java.lang.String int
boolean endsWith java.lang.String
int indexOf int int
int indexOf int
int indexOf java.lang.String
int indexOf java.lang.String int
int lastIndexOf java.lang.String int
int lastIndexOf int int
int lastIndexOf java.lang.String
int lastIndexOf int
java.lang.String substring int int
java.lang.String substring int
java.lang.String concat java.lang.String
java.lang.String replace char char
java.lang.String toLowerCase java.util.Locale
java.lang.String toLowerCase
java.lang.String toUpperCase
java.lang.String toUpperCase java.util.Locale
java.lang.String trim
[C toCharArray
java.lang.String intern
```

```
*/
```

Et Javas tulevad pärimisel käsklused kaasa, siis saab eristada klassis eneses loodud käsklusi ning kaasatulnud käsklusi. Esimesed neist saab kätte käsuga getDeclaredMethods(). Tahtes kätte saada kogu pärimispuu jooksul loodud käsklused, võetakse järgnevas näites pärast konkreetse klassi käskluste uurimist ette tema ülemklass ning nõnda kuni juureni välja. Ehk kuni enam ülemklassi võtta ei ole ning meetod getSuperclass väljastab tühitunnuse null.

```
    Class c=p.getClass();
    while(c!=null){
      ...
      c=c.getSuperclass();
    }
```

Kui tegemist pikema pärimishierarhiaga, siis võib käskude loend õige pikaks minna. Siin näidatud lihtsalt kood ja mõned esimesed käsud.

```
import java.lang.reflect.Method;
import java.awt.Panel;
public class Klassiuuring3{
  public static void main(String[] argumendid){
    Panel p=new Panel();
    Class c=p.getClass();
    while(c!=null){
      System.out.println("Klassi nimi: "+c.getName());
      Method[] m=c.getDeclaredMethods();
      System.out.println("Meetodid:");
      for(int i=0; i<m.length; i++){
        System.out.println("  "+m[i].getName());
      }
      c=c.getSuperclass();
    }
  }
}

/*
Klassi nimi: java.awt.Panel
Meetodid:
  constructComponentName
  addNotify
  getAccessibleContext
Klassi nimi: java.awt.Container
Meetodid:
  add
  ...
*/
```

Käivitamine nime järgi

Ehkki harva, kuid vahel siiski tuleb ette olukord, kus tahetakse loodava objekti tüüp määrata alles programmi käivitamise ajal. Üheks taoliseks näiteks on juhtum, kus vanema interpretaatoriversiooniga tahetakse kasutada üht klassi, uuema puhul teist. Kui aga uuem klass koodi sisse kirjutada, siis võib verifeerija teatada tundmatust klassist ning kogu koodi käivitamise keelata.

Samuti läks nime järgi eksemplari loomist vaja olukorras, kus sooviti funktsioon joonistada mitmesuguste kasutaja sisestatud valemite järgi. Et aga Javas eval-käsklus koodilõigu käivitamiseks puudub, siis aitas hädast välja kompilaator. Kasutaja sisestatud avaldise põhjal kompileeriti kokku uus klass. Edasi tuli joonistamise tarbeks klassist eksemplar luua. Vana klassi nime ei saanud kasutada, sest see võis olla juba puhvrisse laetud ning nõnda oleks graafik tulnud vana avaldise

alusel. Sobis aga lahendus, kus igal korral kompileeriti kokku uue nimega klass ning loodi vastava nimega klassist eksemplar.

Teine võimalus olnuks kasutada ClassLoaderit ning kompilaatori pakutud baitkoodist sealtkaudu klassi eksemplar luua. Rakendis aga polnud turvanõuete tõttu ClassLoaderi kasutamine võimalik, nime järgi klassi eksemplaride loomine aga küll.

Järgnevas näites paistab, kuidas võib nime järgi luua klassi eksemplari. Samuti küsitakse klassilt nime järgi välja meetod ning selle kaudu sisestatakse loodud isendile väärtus.

```
import java.lang.reflect.*;
public class MeetodiTest{
  public static void main(String[] argumendid) throws Exception{
    Class c=Class.forName("Arv");
    Arv a1=(Arv)c.newInstance();
    Method m=c.getMethod("paneSisu", new Class[]{int.class});
    m.invoke(a1, new Object[]{new Integer(7)});   System.out.println(a1.kysiSisu());
  }
}

class Arv{
  int a;
  public void paneSisu(int uusArv){
    a=uusArv;
  }
  public int kysiSisu(){
    return a;
  }
}
```

Ülesandeid

Klasside ja objektide uuring.

* Tutvu klassiuuringu näidetega.
* Trüki välja klassi java.lang.Integer meetodid.
* Käivita meetod parseInt käskluse invoke abil.

JUnit, automaattestimine

Kord valmiskirjutatud koodi haldamisele - muudatuste tegemisele ja vigade parandamisele pidada minema hulga enam ressursse kui koodi enese loomisele. Programmi töö korrektsuse kontrolliks on välja mõeldud hulgem tehnikaid, üks nendest on automaattestimine. Automaattestidega kontrollitakse üldjuhul üksikute alamprogrammide või väiksemate koodilõikude tööd, kuid on võimalik ka suuremate toimingute kontroll. Ekstreemprogrammeerimise nimelise tehnika juures peetakse programmi tööd kontrollivaid teste vähemasti sama tähtsaks kui programmi ennast, kuid oma roll on neil muudelgi puhkudel. Kui automaattestid kirjutatakse enne kui kood, siis on testid heaks abivahendiks soovitavate funktsionaalsuste kavandamisel ja kirja panemisel. Samuti saab nende kaudu küllalt hästi kontrollida näiteks kusagil eemal kolmanda osapoole loodud tükkide sobivust süsteemi.

Teise tähtsa omadusena saab automaattestide abil kontrollida soovitud funktsionaalsuse säilimist süsteemis tehtud muudatuste järel. Vähegi suuremas rakenduses tekivad soovimatud kõrvalnähud lubamatult kergesti. Automaattestide abil aga õnnestub pärast uue mooduli lisamist või vana muutmist ülejäänud süsteemi toimimine kontrollida vähemasti testidesse kirjutatud tavajuhtumite puhul.

Java puhul on tõenäoliselt levinuimaks automaattestimisvahendiks tasuta kasutatav JUnit. Programmeerija ülesandeks on kirjutada testitava rakenduse

funktsionaalsuste kontrolliks hulk terviklikke alamprogramme. JUniti keskkond käivitab need ning peab statistikat, milliste töö toimus edukalt milliste oma mitte. Kontrollimiseks võib olla koostatud kümneid, sadu või isegi tuhandeid koodilõike kuid rakendus loetakse toimivaks juhul, kui ta kõikide testide nõudmistele vastab.

JUniti testprogrammid kirjutatakse tavaliste Java klassidega, vaid ülemklassiks peab olema paketist junit.framework kaasa tulnud TestCase. Tavalisi meetodeid võib klassi lisada nõnda nagu igaüks soovib. Testolukordi tähistavad meetodid peavad algama sõnaga test. Meetodi tagumise poole nimed võib programmeerija ise määrata. Tõenäoliselt eelpool kirjeldatud klassi meetodite vaatamise ja käivitamise vahendi abil suudab JUniti testimissüsteem panna meetodid tööle ilma, et nende nimed peaksid kuhugile mujale olema sisse kirjutatud.

Olukorda testiv funktsioon üldjuhul teeb oma toimingud ning viimaste käskude hulgas assert-lausega kontrollitakse, kas saavutati eeldatav olukord. JUnit-kestprogrammi ülesandeks on meetodid käivitada ning lõpus kokku lugeda, milliste testkäskluste käivitamisel probleemid tekkisid. Siin esimeses lihtsamas näites küll vaid kontrollitakse, kas kolm pluss kaks on ikkagi viis.

Testkeskkonna vaatamiseks ja juhtimiseks on loodud mitu kasutajaliidest. Siin näites pannakse tööle kõige vähem ressursse nõudev neist ehk tekstipõhine. Olemas on veel nii awt kui Swingi graafikal põhinevad liidesed.

```
import junit.framework.*;

public class Minitest extends TestCase{
  public void testArvutus(){
    assertEquals(3+2, 5);
  }
  public static void main(String[] argumendid){
    junit.textui.TestRunner.run(new TestSuite(Minitest.class));
  }
}
```

JUniti kompileerimiseks tuleb nende kodulehelt http://www.junit.org/ alla laadida arhiiv, mille ühe osana paiknev fail junit.jar sisaldab tarvilikke klasse ning on hädavajalik kompileerimisel ja käivitamisel. Nagu õpetuses öeldud, ei tohi junit.jar-i paigutada lisamoodulite juurde jre/lib/ext kataloogi, vaid peab parameetriga käsureal kaasa andma.

Väljundist võib näha, et üks ja ainuke testitoiming sooritati edukalt.

```
/*
C:\java\jaagup\testid>javac -classpath junit.jar;. Minitest.java

C:\java\jaagup\testid>java -cp junit.jar;. Minitest
.
Time: 0,01

OK (1 test)
*/
```

Üldjuhul koostatakse testimoodulid konkreetsete moodulite, klasside, alamprogrammide või muul viisil piiritletud tervikute kontrolliks. Enne testfunktsioonide käivitamist võib mõnikord tarvilik olla rakenduse osa testimiseks ette valmistamine ning pärast testimise lõppu taas seiskamine. Selle tarvist võib testmoodulis üle katta funktsioonid setUp ning tearDown. Siis pole igas testkäskluses eraldi vaja nende toimetustega tegelda.

Järgnevas näites kontrollitakse bittväärtuste hoidlana toimiva java.util.BitSeti tööd. BitSeti puhul saab iga elemendi puhul määrata, kas sealne bitt on püsti või mitte. Algseadistamise juures määratakse bitt number 2 püstiseks. Edasiste funktsioonide juures kontrollitakse, kas bitid paiknevad ettearvatult.

```
import junit.framework.*;
import java.util.BitSet;

public class Bitihulgatest extends TestCase{
  BitSet b=new BitSet();
  public void setUp(){
    b.set(2, true);
    System.out.println("Testi algus");
  }
  public void tearDown(){
    System.out.println("Testi ots");
  }
  public void testPysti(){
    assertTrue(b.get(2));
  }
  public void testPikali(){
    assertFalse(b.get(1));
  }
  public static void main(String[] argumendid){
    junit.textui.TestRunner.run(new TestSuite(Bitihulgatest.class));
  }
}
/*
C:\java\jaagup\testid>javac -classpath junit.jar;. Bitihulgatest.java

C:\java\jaagup\testid>java -cp junit.jar;. Bitihulgatest
.Testi algus
Testi ots
.Testi algus
Testi ots

Time: 0,01

OK (2 tests)

*/
```

Testide kogum

Suurema rakenduse puhul on lihtsam kirjutada iga tervikliku osa test omaette faili. Rakenduse täiskontrolli ajal tuleb aga kõik selle kohta käivad testid käivitada. Nõnda on võimalik koostada testidest kogum ning see tervikuna käivitada.

```
import junit.framework.TestSuite;
public class TestiKogum{
  public static void main(String[] argumendid){
    TestSuite kogum=new TestSuite();
    kogum.addTest(new TestSuite(Minitest.class));
    kogum.addTest(new TestSuite(Bitihulgatest.class));
    junit.textui.TestRunner.run(kogum);
  }
}

/*
C:\java\jaagup\testid>javac -classpath junit.jar;. TestiKogum.java

C:\java\jaagup\testid>java -cp junit.jar;. TestiKogum
..Testi algus
Testi ots
.Testi algus
Testi ots

Time: 0,02

OK (3 tests)
*/
```

Ülesandeid

* Tutvu näidetega. Lae ja paki lahti JUnit. http://www.junit.org
* Lisa Minitesti testfunktsioon, mis kontrolliks, et nimes "Juku" on 4 tähte.
* Loo omaette testklass, mille initsialiseerimisel luuakse fail ning kirjutatakse sinna kümme juhuslikku arvu; arvude summa jäetakse meelde. Lisa testfunktsioon

kontrollimaks, kas failis on ikka kümme rida ning teine jälgimaks, et arvude summa on muutumatu.
* Testi sulgemisel kustutatakse fail.

* Loo liides nimede lisamiseks, otsimiseks ja kustutamiseks.
* Loo liidest realiseeriv klass.
* Loo test klassi oskuste mitmekülgseks kontrolliks.

* Loo liidese põhjal klass, mis kasutaks faili asemel andmebaasi.
* Hoolitse, et loodud testid töötaksid ka uue klassi puhul.