Objektid programmeerimisel

"Arvutimaailm lendab lõhki! Ja suurimaks probleemiks pole mitte viirused. Pole ka häkkerid ja kräkkerid. Suurimaks probleemideks programmides on vead!"

Suurelt jaolt vigade leidmise tarvis hakatigi otsima programmeerimisse uusi vahendeid. Objektorienteeritud programmeerimine on viimase paarikümne aasta jooksul aidanud vigu leida ning tal on ka muid kasulikke omadusi.

Suuremate projektide korral aitab objektideks jagamine tõsta abstraktsiooni taset ning valmistada ja kontrollida iseseisvaid programmi osi eraldi. Samuti nagu alamprogrammide loomine aitab orienteeruda suures käskude jadas, aitab objektide loomine orienteeruda alamprogrammide rägastikus.

Valmis programmilõike saab korduvalt kasutada vaid siis, kui on võimalik nad arhiivist üles leida. Ka siin on kasu kõrgemast abstraktsioonitasemest.

Mõnikord on lihtsalt mugav reaalset elu jäljendavaid programme panna kokku objektidest, sest ka tegelikus elus on meil tegemist suhteliselt iseseisvate üksustega (auto, inimene ...).

Java programmis peab olema kogu kirjutatav kood meetodi sees. Meetod omakorda klassi ehk objektitüübi sees. Klassid omakorda moodustavad paketi.

Lihtsama programmi puhul

```
public class Tervitus{
  public static void main(String argumendid[]){
    System.out.println("Tere");
  }
}
```

on koodiks `System.out.println("Tere");`
meetodiks `main` ning klassiks `Tervitus`.

Lühikeste programmide korral saab kogu programmi teksti kirjutada `main`- meetodi sisse ning objektitüüp tema ümber ei ole millekski kasulik. Kui aga programmis tuleb luua uus tüüp ning sellest tüübist isendiga suhtlema hakata, siis saab programmi teksti muuta tavakeelele lähemaks ning isendeid kergemini eraldi kontrollida.

```
class Punkt4{
  int x, y;
  public Punkt4(){
    this(0, 0);
  }
  public Punkt4(int uus_x, int uus_y){
    x=uus_x;
    y=uus_y;
  }
  public String kirjutaAndmed(){
    return "x="+x+" y="+y;
  }
  public double kaugusNullist(){
    return Math.sqrt(Math.pow(x, 2)+Math.pow(y, 2));
  }
}

public class Punktid4{
  public static void main(String argumendid[]){
    Punkt4 p=new Punkt4(3, 4);
    System.out.println(p.kirjutaAndmed());
    System.out.println("Kaugus koordinaatide alguspunktist= "+
                       p.kaugusNullist());
  }
}
```

```
D:\Kasutajad\jaagup\java>java Punktid4
x=3 y=4
Kaugus koordinaatide alguspunktist= 5.0
```

Kas või selle näite koostamisel oli klassideks jagamine abiks. Kirjutasin programmi valmis ning panin käima. Selgus, et käivitamisel pakkus ta `x` ja `y` väärtuste 3 ja 4 juures kauguse nullist 2.2 juures. Kuna peaprogrammis on vaid palve saata oma kaugus, siis seal polnud arvutusviga teha võimalik. Seetõttu nägin, et punkt "käitub imelikult" ning teadsin kohe tema juurest viga otsima minna. Selgus, et olin ruutjuure lihtsalt kaks korda võtnud.

Pärimine

Kui soovida klassile `Punkt4` omadusi lisada, siis pole selleks vaja terve klassi koodi uuesti ümber kopeerida. Saab luua laiendatud võimalustega klassi, mis pärib klassist `Punkt4` tema omadused ning lisaks saab talle meetodeid juurde kirjutada.

```
class Punkt4Laiend extends Punkt4{
  public void liiguParemale(){
    x++;
  }
}


class Punktid4a{
  public static void main(String argumendid[]){
    Punkt4Laiend p=new Punkt4Laiend();
    System.out.println(p.kirjutaAndmed());
    p.liiguParemale();
    System.out.println(p.kirjutaAndmed());
  }
}
```

```
D:\Kasutajad\jaagup\java>java  Punktid4a
x=0 y=0
x=1 y=0
```

Lisaks ümberkirjutamise vaeva vähenemisele näitab selline toimimine ka paremini välja, millise klassi isend mida oskab ning millised oskused tal võrreldes eelnevaga on juurde tulnud. Kuna sama koodi on vaid kord, siis muutmise vajaduse puhul tuleb muutus sisse viia samuti vaid ühes kohas. Kui soovin muuta väljatrükki ilmekamaks, siis piisab, kui muudan klassi `Punkt4` meetodi `kirjutaAndmed()`. Uut koodi kasutatakse siis ka klassi `Punkt4Laiend` isendi andmete välja kirjutamisel. Samuti piisab ka vea leidmisel vaid ühes kohas parandamisest.

Kuna isend klassist `Punkt4Laiend` oskab teha kõike, mida klassi `Punkt4` isend, siis teda saab kasutada kõikjal, kus `Punkt4`-gi. Analoogia tavaelust võiks olla, et näiteks lendur suudab kõike, mis tavaline inimenegi (süüa, juua, magada, kellaaega öelda jne.), kuid lisaks sellele suudab ta veel lennukit juhtida. Kellaaega võime küsida ükskõik millise inimese käest, kuid lennukijuhtimist saame paluda vaid lendurilt. Seetõttu on omistamine inimene=lendur ehk `Punkt4=Punkt4Laiend` täiesti lubatud ja võime levinumat tegevust (nt. `kirjutaAndmed()`) küsida `Punkt4` tüüpi muutuja käest teadmata, et tegemist on tegelikult isendiga tüübist Punkt4Laiend sarnaselt nagu võime kella küsida vastutulevalt inimeselt, teadmata, et vastutulija on ametilt lendur. Kui aga on vaja paluda ülemklassile ainuomast tegevust, siis tuleb enne veenduda, et meil kasutada olev isend ka selleks võimeline on. Samuti nagu lennujuhti vajades teeme kindlaks, et rooliasuja tõepoolest ikka lendur on. Et ülemklassi meetod välja kutsuda, tuleb muutujas peituva isendi tüüp muundada vastavaks. Kui kirjutaksime `(Punkt4Laiend)p.liiguParemale()`, siis saaks kompilaator aru, et tuleks kõigepealt muutuja p poolt osutatud isendil paluda paremale liikuda ning seejärel saadud tulemus muundada tüübiks `Punkt4Laiend`. Sel juhul kompilaator vaatab, et `Punkt4`-l palutakse paremale liikuda, mida ta ei oska ning tekib veateade. Kui aga lisada veel ühed sulud, siis on tulemuseks `((Punkt4Laiend)p).liiguParemale()` ehk muutujas `p` olnud isend `Punkt4Laiend`'iks muundatuna ning temal juba on võimalik lasta paremale liikuda. Juhul, kui muutujas `p` siiski aga pole `Punkt4Laiend`i tüüpi isendit, tekib viga. Samuti nagu inimesel, kes pole lendur, pole võimalik lenduritunnistust näidata. Kui tüüp vastab oodatule, on kõik korras.

```
class Punktid4b{
  public static void main(String argumendid[]){
    Punkt4 p=new Punkt4Laiend();
    System.out.println(p.kirjutaAndmed());
    ((Punkt4Laiend)p).liiguParemale();
```

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

```
D:\Kasutajad\jaagup\java>java Punktid4b
x=0 y=0
x=1 y=0
```

Üksikjuhul paistab säärasest muundamisest vähe kasu olema, kuid sellise omistamisvõimaluse tõttu saab ühest allikast põlvnevate isendite andmeid ühes massiivis koos hoida. Lisaks saab neilt paluda seda, mida nad kõik oskavad ning kui oleme kindlaks teinud, et vastaval isendil rohkem oskusi on, saame temalt ka nende rakendamist küsida. Nii võime inimeste andmed üheskoos hoida, sõltumata sellest, kas mõned neist on lendurid või mitte. Kui oleme aga inimese käest teada saanud, et ta lendur on, siis saame tal paluda lennuki rooli asuda. Operaator `instanceof` kontrollib, kas kontrollitav isend suudab vastavale klassile seatud ülesandeid täita, s.t. kas ta on sellest klassist või tema alamklassist andes vastuseks true või false.

```
class Punktid4c{
  public static void main(String argumendid[]){
    int pArv=3;
    Punkt4 pd[] = new Punkt4[pArv];
    pd[0]=new Punkt4(2, 1);
    pd[1]=new Punkt4Laiend();
    pd[2]=new Punkt4(5, 5);
    for(int nr=0; nr<pArv; nr++){
      if(pd[nr] instanceof Punkt4Laiend)
        ((Punkt4Laiend)pd[nr]).liiguParemale();
    }
    for(int nr=0; nr<pArv; nr++){
      System.out.println(pd[nr].kirjutaAndmed());
    }

  }
}
```

```
D:\Kasutajad\jaagup\java>java Punktid4c
x=2 y=1
x=1 y=0
x=5 y=5
```

Kuna kõik klassid on ühtlasi klassi `Object` alamklassid (et see on üldine, siis võib kirjutamata jätta), siis saab kõiki struktuurtüüpide andmeid panna kokku objektimassiivi.

Ülekate

Pärimisel saab luua meetodeid juurde. Samuti on võimalik panna alamklassis ülemklassi meetod teisiti toimima. Selleks tuleb lihtsalt kirjutada samanimeline ning samade parameetritega meetod.

```
class Inimene{
  int vanus;
  public void ytleVanus(){
     System.out.println("Mu vanus on "+vanus);
  }
}

class Daam extends Inimene{
  public void ytleVanus(){
    System.out.println("Sain just "+(vanus-5)+" aastat vanaks.");
  }
}

public class Tutvustus{
  public static void main(String argumendid[]){
    Inimene naine1=new Inimene();
    Inimene naine2=new Daam();
    naine1.vanus=40;
    naine2.vanus=40;
```

```
    naine1.ytleVanus();
    naine2.ytleVanus();
  }
}
```

```
D:\Kasutajad\jaagup\java>java Tutvustus
Mu vanus on 40
Sain just 35 aastat vanaks.
```

Liidesed

Iga klassi isend suudab käituda nii selle klassi kui ka tema ülemklasside jaoks loodud olukordades. Nagu siin näiteks on iga Daam samal ajal ka Inimene, sest Daam pärineb inimesest. Ning samal ajal on nii Inimene kui Daam ka Object, sest Java hierarhias on juhul, kui ülemklassi pole määratud, ülemklassiks vaikimisi Object. Kui aga soovime, et meie loodava klassi isend sobiks peale oma ülemklasside ka muudesse kategooriatesse, tuleb kasutada liideseid.

```
interface Viisakas{
  public void tervita(String partner);
}
```

Kui loodav klass realiseerib liidest `Viisakas`, peab temas olema meetod `tervita`, mis saab parameetriks sõne. Liidese realiseerimine annab klassile kohustuse osata liideses ette antud tegevusi (ehk meetodeid). Samas annab aga selle klassi isendile asuda igal pool, kus on lubatud olla vastavat liidest realiseerival klassil.

```
class Laps extends Inimene implements Viisakas{
  public void tervita(String tuttav){
    System.out.println("Tere "+tuttav);
  }
}

class Koer implements Viisakas{
  public void tervita(String nimi){
     System.out.println("Auh!");
  }
}

public class Tutvustus2{
  public static void main(String argumendid[]){
    Laps juku=new Laps();
    juku.vanus=6;
    juku.ytleVanus();
    Viisakas kylaline1=juku;
    Viisakas kylaline2=new Koer();
    kylaline1.tervita("vanaema");
    kylaline2.tervita("Juku vanaema");
  }
}
```

```
D:\Kasutajad\jaagup\java>java Tutvustus2
Mu vanus on 6
Tere vanaema
Auh!
```

Mõnes programmeerimiskeeles (näiteks C++) eraldi liideseid ei ole. Seal lubatakse pärida korraga mitmest ülemklassist ning loodud klassi isend on samuti võimeline paiknema nii ühe kui teise ülemklassi jaoks eraldatud kohas. Kuna aga võib juhtuda, et mõlemal ülemklassil on sama nimega meetod, siis sellest võib tekkida probleeme. Selleks on javas loodud liidesed, mida realiseerides tekib vaid kohustus liideses kirjeldatud tegevusi osata. Oskusi eneseid ehk programmikoodi saab aga pärida vaid ülemklassidest. Kui on vaja mõnele klassile lisada teise klassi oskusi, siis on võimalik klassi üheks andmeväljaks võtta teise klassi isend, kellel paluda soovitud tegevusi sooritada. Ülemklasse saab olla igal klassil vaid üks, liideseid aga võib realiseerida iga klass piiramatul hulgal.

Piiritlejad

Objektorienteeritud programmides püütakse objekti kasutaja eest talle mittevajalikud tunnused ära peita, et need teda ei häiriks ning et ta ka ei saaks neid muuta ning sellega objekti toimimist häirida. Vaikimisi on võimalik nii klasse, meetodeid kui muutujaid kasutada vaid sama paketi (kataloogi) seest. Piiritleja `public` tähendab, et ligi pääseb ka väljastpoolt, `private` lubab kasutada vaid sama klassi piires ning `protected` lubab lisaks oma klassile ka alamklassides. Seda ka juhul, kui alamklassid juhtuvad teises paketis olema; `final` tähendab, et enam ei lubata muuta. Kui muutujal on ees piiritleja `final`, siis tema väärtus omistamisjärgselt enam muutuda ei saa, s.t. on konstant. Struktuurtüübi muutuja jääb `final` piiritleja korral kuni oma eluea lõpuni osutama samale isendile. Kui meetod on `final`, siis ei saa teda üle katta. Kui klass on `final`, siis ei saa talle luua alamklasse. Võtmesõna `static` meetodi või muutuja ees tähendab, et meetod või muutuja kuulub klassile, mitte isendile. Tema kasutamiseks pole vaja luua eraldi isendit. Staatilisi meetodeid ei saa üle katta. Kui harilikud muutujaid ehk väljad luuakse iga isendi jaoks uued (nt. kolmel isendil tüübist `Inimene` on igaühel oma `vanus`), siis staatilist muutujat on vaid üks (näiteks muutuja klassi `Inimene` juures, mis näitab inimeste arvu). Sellisel muutujal on väärtus siis, kui ühtki inimest pole loodud. Sellisel juhul on vastava muutuja väärtuseks 0. Samuti on tal üks ja konkreetne väärtus juhul kui inimesi on rohkem.

Lisaoskused

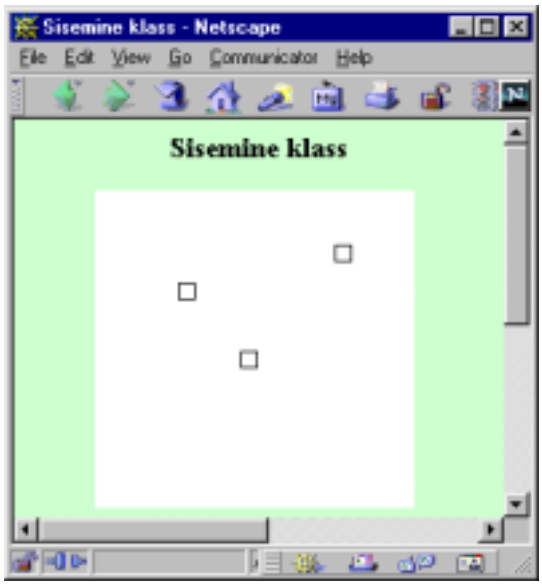
Java keeles tohib olla igal klassil vaid üks ülemklass. Kui tahta oma loodud klassis kasutada muude klasside oskusi, siis on mõistlik oma klassi luua muutuja, mille abil pöörduda vastava klassi isendi poole. Näiteks kui soovin, et rakendi pinnale tekiks tekstiväli, siis lasen rakendil lihtsalt luua isendi klassist TextField ning paigutada viimase oma pinnale. Märkimisväärne hulk olukordi õnnestub niimoodi lahendada, kus puuduvate võimaluste hankimiseks kasutatakse muude klasside isendeid.

Loodud isendile saab ligi ja tunnuseid pärida, sest meie loodud klassis paiknev muutuja osutab sinna. Nii on näiteks kerge rakendist tekstivälja sisu muuta või pärida. Samas on raske panna tekstivälja muutuse peale rakendil midagi toimuma, sest loodud tekstiväljal me rakendi kohta andmed puuduvad. Et saaks tekstiväljalt rakendile teateid saata, selleks peab esimesel teisele ligipääsuks osuti olema. Kui kirjutame rakendis `tf.addActionListener(this),` sel juhul annamegi loodud tekstiväljale võimaluse rakendile ligi pääseda ning tema meetodeid käivitada. Muul juhul peaks rakend pidevalt iga natukese aja tagant tekstivälja kontrollima, et kas seal midagi muutunud on ning siis seepeale muutuse avastama ning reageerima. Lisaks saab kõikjalt kasutada teiste klasside poolt välja pakutud staatilisi meetodeid, selleks pole isegi vastava klassi isendit vaja luua.

Sisemine klass

Sisemised klassid võimaldavad ka teisiti klassi võimalusi laiendada. Sisemisel klassil on kasutada lähteklassi eksemplari kõik muutujad ja meetodid, lisaks sellele veel need, mis sisemises klassis eneses juurde kirjeldatud on. Kuna sisemised klassid võivad olla samaaegselt ka mõne muu klassi alamklassid, siis nõnda saab ühendada kahe klassi omadused. Järgnevas näites kuulub rakendiklassi `SisemiseKlassigaRakend` sisse klass `SisemineKlass`. Hiirevajutustele reageerimiseks luuakse `init`-meetodis sisemisest klassist isend ning pannakse ta rakendile saabuvaid hiire teateid kuulama. Kuna sisemine klass pääseb ligi rakendi meetoditele ja muutujatele, saab ta ka ekraanile joonistada. Joonistusvahendi küsimiseks tuleb siiski kirjutada `SisemiseKlassigaRakend.this.getGraphics()`, sest lihtsalt `this` tähendaks sisemise klassi seest kutsutuna hoopis sisemise klassi eksemplari, kelle valduses aga ekraani pole, millele joonistada.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class SisemiseKlassigaRakend extends Applet{
  class SisemineKlass extends MouseAdapter{
    public void mousePressed(MouseEvent e){
      SisemiseKlassigaRakend.this.getGraphics().
        drawRect(e.getX(), e.getY(), 10, 10);
    }
  }
  public void init(){
   addMouseListener(new SisemineKlass());
  }
}
```



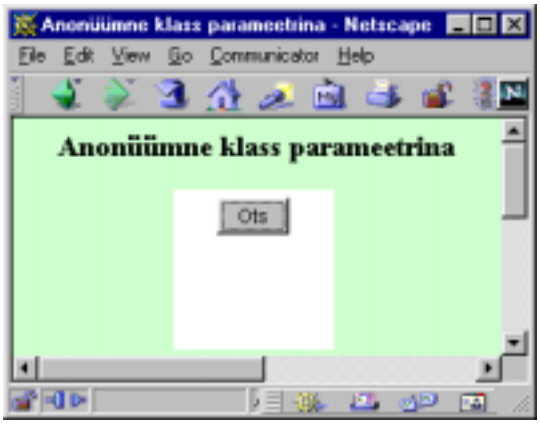
Sisemise klassi saab luua ka isendi loomise ajal.

```
import java.applet.Applet;
import java.awt.event.*;
import java.awt.*;
public class AnonyymneKlass extends Applet{
  final Button b=new Button("Algus");
  ActionListener anonyymneKuular=new ActionListener(){
    public void actionPerformed(ActionEvent e){
      b.setLabel("Ots");
    }
  };

  public void init(){
    add(b);
    b.addActionListener(anonyymneKuular);
  }
}
```

Veel lühem võimalus on toodud allpool. Siin luuakse sündmusekuularit realiseeriv isend otse `addActionListeneri` sulgude sees.

```
import java.applet.Applet;
import java.awt.event.*;
import java.awt.*;
public class AnonyymneKlass2 extends Applet{
  Button b=new Button("Algus");
  public void init(){
    add(b);
    b.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent e){
        b.setLabel("Ots");
      }
    });
  }
}
```



Arengusuunad

Arvutusvõimsuste ja programmimahtude kasvades on programmide arengut pidurdama jäänud programmeerija tööaeg ning programmides leiduvad vead. Loodeti, et tupikust päästab objektitehnika, kuid ka see ei osutunud imevahendiks. Nii nagu protseduurid aitasid liigendada jadaprogramme, aitasid objektid süstematiseerida protseduure. Kui aga klasse saab palju, on nendega samuti raske toime tulla. Javas on loodud lisatasemeks paketid, kuhu koondatakse sarnaste omadustega klassid. See aitab sobivat klassi leida. Samuti saavad ühes paketis paiknevad klassid lubada vastastikku üksteise omadusi kasutada.

Järgmiseks lahenduseks on pakutud komponenttehnoloogia. Selle järgi pannakse programm kokku kasutamisvalmis tükkidest. Näiteks võivad tükkideks olla kalender, liikuv auto, draiverite komplekt. Programmeerijal tuleb nad sobivalt ühendada ning loota, et tulemusena valmib kasutajasõbralik programm. Ühte komponenti võivad kuuluvad klassid eri pakettidest, samuti võib võrguprogrammide puhul üks komponent asetseda laiali mitmes masinas. Komponente saab luua mitmetes programmeerimiskeeltes. Nii on Java komponentideks oad (Bean), Microsoftil ActiveX komponendid. Et nad omavahel suhelda saaksid, on kirjeldatud sillad, millisel kujul komponendid andmeid vahetada saavad.

Mõningaid Java abil loodud komponente nimetatakse ubadeks (bean). Et neid võiks julgelt kõikjal kasutada, on neil samasugused turvapiirangud nagu rakenditelegi. Vabalt saab omale laadida BDK, mille abil saab hiirega komponente kokku lohistada ning rakendiks muuta. Seal saab kasutada ka neid komponente, mille loomisel pole BDK võimalusi silmas peetud, kuid järgides komponentidevahelise suhtlemise reegleid on võimalik oma komponenti teistega paremini suhtlema panna.

Kokkuvõte.

Objektorienteeritud programmeerimine loodi vigade paremaks avastamiseks, suurte programmide lihtsamaks kirjutamiseks, programmikoodi korduvkasutuseks ning reaalse maailma objektide paremaks kirjeldamiseks.

Enne isendi loomist peavad olema tema omadused ja oskused kirjeldatud klassis muutujate ja meetodite abil. Klassi isendi oskusi saab suurendada, luues klassile alamklassi. Alamklassis saab meetodeid juurde kirjeldada, samuti kasutada ülemklassi meetodeid. Kui soovida ülemklassi meetodi tööd muuta, tuleb alamklassis kirjeldada sama nime ja parameetritega, kuid alamklassi jaoks vajaliku koodiga meetod. Alamklasside isendit saab kasutada kõikjal, kuhu sobib ülemklassi isend, sest alamklasside omad mõistavad kõiki neid tegevusi sooritada, mis ülemklassis kirjeldatud on. Nad kas pärivad need oskused muutumatuna, või muudavad nende sisu meetodi ülekatmise abil. Kui soovida,

et loodava klassi isend saaks tegutseda ka mujal kui oma ülemklassile ettenähtud kohas, tuleb kasutada liidest. Liidese realiseerimiseks peavad loodavad klassis olema kirjeldatud kõik oskused, mis liideses kirjas on.

Piiritlejate `public`, `protected` ja `private` abil saab määrata ligipääsetavust. Piiritleja `static` näitab, et temaga märgitu kuulub klassi, mitte isendi juurde.

Lihtandmetüüpideks Java-s on `byte`, `short`, `int`, `long`, `float`, `double`, `char` ja `boolean`. Nendest saab kombineerida struktuurtüüpe. Lihttüübi muutujas on väärtus, struktuurtüübi omas osuti isendile. Struktuurtüübist muutuja tühivääruseks on `null`. Nii liht- kui struktuurtüüpi andmeid saab paigutada
massiivi.

Ülesandeid

Objektid

- Loo klass Inimene väljadega eesnimi ja synniaasta
- Koosta klassist kaks eksemplari, anna väljadele väärtused
- Koosta inimeste andmetest massiiv. Trüki välja inimesed, kes on sündinud varem kui 1980.

- Koosta Inimese alamklass Kodanik. Lisa väli "perekonnanimi"
- Koosta massiiv, kus on nii Inimesed kui Kodanikud
- Trüki massiivist välja kõik eesnimed, kodanikel ka perekonnanimed.

- Loo Kodanikule konstruktor andmete sisestamiseks.
- Lisa kodanikule ka sünniaastata konstruktor. Sel juhul
- tuleb sünniaasta väärtuseks -1.
- Lisa ka Inimesele konstruktor andmete sisestamiseks.
- Kutsu Kodaniku Konstruktorist välja Inimese konstruktor.

Klassid ja isendid

- Loo klass Loom väljaga "nimi" ning meetodid "tutvusta" ning "nimepikkus".
- Tee loomale alamklassid "Koer" ja "Lehm", kus tuleb katta üle meetod "tutvusta" nii, et selles väljastataks ka looma liik ning looma nime pikkus. Koosta kestprogramm loodud klasside katsetamiseks.
- Lisa klassile Loom staatiline väli loomadeArv, mille väärtus suureneb iga looma (ka koera ja lehma) lisamisel (st klassi Loom konstruktori käivitamisel). Muuda klassi Loom meetodit "tutvusta" nii, et see väljastaks loomade arvu. Kutsu klassi "Koer" meetodist "tutvusta" välja klassi "Loom" meetod "tutvusta".
- Muuda konstruktoreid nii, et isendi loomisel saaks talle koheselt ka nime omistada. Loo staatiline meetod loomade arvu väljastamiseks ning käivita see ilma isendit kasutamata.

Punktid

- Loo klass Punkt väljadega x ja y.
- Loo sellest klassist kaks isendit ja anna nende väljadele väärtused.
- Trüki mõlema punkti väljade väärtused ekraanile.
- Kopeeri ühe punkti andmed teise punkti andmeteks.
- Paiguta loodud kaks punkti massiivi.

- Loo massiiv saja punkti ning juhuslike andmetega.
- Määra viiekümnenda punkti x-koordinaadi väärtuseks 243.
- Kirjuta välja punkti andmed, mille x-koordinaadi väärtus on vähim.

Erindid, veatöötlus

Iga vähegi suurema programmi juures tekib mittestandardseid olukordi, mis tuleb programmi sujuva töö jätkumiseks lahendada. Näiteks sisestab kasutaja sobimatu numbri või pole kettal piisavalt ruumi vaheandmete salvestamiseks. Üksikjuhtudel võib probleemi lahenduse kirjutada kahtlase olukorra juurde ning tillukeste programmide juures on see täiesti kasutatav. Kui aga sarnaseid veaohtlikke kohti on palju, siis tuleb sellise lähenemise juures ka kontrollimehhanisme palju kirjutada. See võib omakorda programmi kohmakaks, väheülevaatlikuks ning raskesti muudetavaks teha. Ka on olukord keerulisem, kui vastavalt olukorrale tuleks samale eksitusele erinevalt reageerida. Näiteks sünniaasta ekslikul sisestamisel võib kasutaja tähelepanu sellele juhtida ning aastat kohe uuesti küsida. Kui aga parooli sisestamisel eksitakse, siis tuleks enne uut sisestusluba mõnda aega oodata, et poleks võimalik lühikese ajaga suurt hulka paroole läbi proovida. Samuti võib juhtuda, et veale on kasulik reageerida selle tekkimise kohast koodis küllalt kaugel. Traditsiooniliselt on sellistest vigadest teada andmiseks kasutatud muutujaid, mis informeerivad paranduslõiku vajaliku paranduse iseloomust või saadavad vea kirjelduse mingit teed pidi tema jaoks loodud töötlemise koha juurde.

Java keelde loodi ootamatustega toime tulemiseks omaette mehhanism aitamaks probleemiinfot transportida lahenduskohani. Probleemi tekkimisel saab luua ning "lendu lasta" probleemiinfot kandva isendi. Sellest kohast alates programmi täitmine katkestatakse. Kui programmilõigu ümber on pandud neid teateid töötlev püünis, siis jätkatakse programmi täitmist püünisele järgnevalt käsurealt. Püünise puudumisel lendab probleemisend programmist välja, jättes kogu teel töö tegemata. Lihtprogrammide puhul tähendab see programmi töö lõppu. Kui erind aga tekkis näiteks rakendi ülejoonistamisel meetodis paint(), siis jääb see joonistuskord lõpetamata. Näide.

Probleem tüübimuundel:

```
public class Erind1{
  public static void main(String argumendid[]){
    String s="aabits";
    int nr=Integer.parseInt(s);
    System.out.println(nr);
  }
}
```

annab töö tulemuseks:

```
C:\TEMP\java>java Erind1
Exception in thread "main" java.lang.NumberFormatException: aabits
        at java.lang.Integer.parseInt(Integer.java:409)
        at java.lang.Integer.parseInt(Integer.java:458)
        at Erind1.main(Erind1.java:4)
```

Lahtiseletatult tähendab arvuti poolt tulnud vastus seda, et lõimes nimega main tekkis erind java.lang.NumberFormatException, kuna sõna aabits ei sobi arvuks. Allpool on näha, et viga tekkis klassi Integer meetodis parseInt failis Integer.java koodireal nr. 409, see meetod oli omakorda välja kutsutud samast failist realt 458 ning see omakorda minu loodud klassi Erind main-meetodis, real nr. 4.

Kuna sõna "aabits" pole võimalik kümnendsüsteemis arvuks muundada, siis kohas, kus seda üritatakse (`Integer.parseInt(s)`) lastakse lendu erind. Kuna siin näites pole erindipüünist, siis jäävad read alates tekkinud probleemist täitmata ning numbri väärtust välja ei trükita.

Järgneva näite

```
import java.io.*;
public class Erind2{
  public static void main(String argumendid[]){
    String s="aabits";
    try{
      int nr=Integer.parseInt(s);
      System.out.println(nr);
    } catch(NumberFormatException ex){
      System.out.println("Vigane number");
    }
```

```
    System.out.println("Programmi ots");
  }
}
```

Käivitamisel paistab:

```
C:\TEMP\java>java Erind2
Vigane number
Programmi ots
```

Siinses näites satub tekkinud erind püünisesse (`try{ ... } catch ...`) ning töödeldakse. Kasutajale teatatakse, et number on vigane. Kui erind on kinni püütud, jätkub programmi töö oma tavalist rada pidi järjekorras mööda käsklauseid.

Järgnevas näites palutakse kasutajal senikaua numbrit sisestada, kuni ta sellega arvuti jaoks loetavalt hakkama saab.

```
import java.io.*;
public class Erind3{
  public static void main(String argumendid[]){
    boolean korrata=true;
    while(korrata){
      try{
        BufferedReader sisse=new BufferedReader(
          new InputStreamReader(System.in)
        );
        System.out.println("Palun number:");
        String rida=sisse.readLine();
        int nr=Integer.parseInt(rida);
        System.out.println("Number "+nr+" sisestati korralikult.");
        korrata=false;
      } catch (IOException sisendierind){
        System.out.println("Probleem klaviatuuriga");
        korrata=false;
      } catch (NumberFormatException numbriformaadierind){
        System.out.println("Valesti sisestatud number. "+
            numbriformaadierind.getMessage()+
            "\n Proovi veel!");
      }
    }
  }
}
```

Eeltoodud näites võib erind tekkida nii klaviatuurilt lugemisel kui sisestatud sõne numbriks muundamisel. Katsendilause `try{ ... }` kogub tekkinud erindid kokku ning seejärel saab neid püünistes töötlema hakata. Iga püünis püüab kinni erindi sellest tüübist, mis on kirjutatud püünise parameetriks, või tema alatüübist. Muud liiki erindi püüdmiseks peab olema talle vastav püünis.

Erindiklasside hierarhia

Erindiklassid on samuti hierarhilised nagu muudki klassid Javas. Kõige üldisem on ehk kõige kõrgemal asub klass `Throwable` (kuid seegi on `Objecti` alamklass nagu muudki), tema alamklassideks on `Error` ning `Exception`. Esimene enamjaolt parandamatute vigade jaoks, teine eriolukordade jaoks, mida on lootust lahendada. Erindi suurematekss alamklassides on `IOException` ning `RuntimeException`. Esimese alla kuuluvad kõik sisendi-väljundi (Input-Output) erindid. `RuntimeExceptioni` alla kuuluvad erindid, mille tekkimise võimalusi on programmis palju, peaaegu igas lauses, kus omistatakse või arvutatakse midagi. Näiteks ruutjuur negatiivsest arvust annab erindi tüübist `ArithmeticException` Kui viietähelisest sõnast püütakse leida seitsmendat tähte, siis tekib `IndexOutOfBoundsException`. Need mõlemad on `RuntimeExceptioni` alamklassid. Kui muud eriolukorra tekkimise võimalused tuleb alati katsendibloki abil kinni püüda või kirjutada meetodi päisesse teade, et meetodist võib väljuda vastav erind, siis klassi `RuntimeException` või ükskõik millist tema alamklassi erindit püüda pole kohustuslik. Selline kohustus lihtsalt muudaks programmi kohmakamaks. Vajadusel saab neid aga püüda nagu muidki erindeid nagu eelpool toodud näiteprogrammides Erind1 ja Erind2.

Kui tahta ühe `catch`-püünisega püüda kinni näiteks nii massiivi rajaerindit (`ArrayIndexOutOfBoundsException`) kui ka tüübimuunduserindit (`ClassCastException`), siis piisab kui püünise parameetriks kirjutada `RuntimeException`. Kuna mõlemad esimesed on viimase alaliigid, siis sobivad nad `RuntimeExceptioni` kohale samuti nagu inimese kohale sobis nii vanaema, lendur kui insener. Kui kirjutame püüniseks `catch(Exception erind)`, siis jäävad sinna kinni kõik erindid ning

`catch(Throwable probleem)` püüab kinni kõik erindid ja vead. Kui tahame, et mõne erindiklassi puhul töödeldaks selle isendit ühtemoodi, kõiki muid erindeid aga teisiti, siis tuleb spetsiifilisem püünis ettepoole panna. Näiteks:

```
try{
  lisaMassiivi(); //varem valmis olev meetod
} catch (ArrayIndexOutOfBoundsException  me){
  System.out.println("Massiiv täis");
} catch(Exception e){
  System.out.println("Eriolukord: "+e.getMessage());
  e.printStackTrace();
}
```

Erindi meetod `printStackTrace()` kirjutab välja, millises alamprogrammis (või ka omakorda alamprogrammi alamprogrammis) probleem tekkis.

Järgnevalt valik sagedamini kasutust leidvatest vea- ning erindiklassidest koos hierarhilise struktuuri ning kommentaaridega.

Klass	Seletus
Throwable	Igasugune probleem
Exception	Erind ehk parandatav eriolukord
ClassNotFoundException	Programmi ühte klassi ei leita
IOException	Sisendi-väljundi erind
FileNotFoundException	Faili ei leita
SocketException	Interneti ühenduse erind
RuntimeException	Igasugu erind, mida ei pea töötlema
ArithmeticException	Arvutuserind (nt. jagamine nulliga)
ClassCastException	Tüübimuunduserind
IllegalArgumentException	Lubamatu argument
NumberFormatException	Vigane numbriformaat
IndexOutOfBoundsException	Rajaerind (nt. masiivi olematu element)
NullPointerException	Muutuja ei osuta isendile, kuigi peaks
NoSuchElementException	Otsitud elementi ei leita
Error	Tõsisem viga
LinkageError	Viga programmi käivitamisel
ClassFormatError	Klassifail vigane
VerifyError	Programmi sees lubamatud operatsioonid
VirtualMachineError	Viga intepreteerimisel
OutOfMemoryError	Mälu otsas

Erindeid saab programmi töö käigus ka ise lendu lasta. Siis saab eriolukorra teate saata töötleva püüniseni ilma selle jaoks muid vahendeid kasutamata. Näiteks:

```
if(nr>100) throw new ArithmeticException("Liiga suur number");
```

Erindi konstruktori parameetriks antud tekst kuulub loodud erindi juurde kogu tema "eluajaks" ning selle teksti sisu saab kätte erindile saadetava teatega `getMessage()`. Seda sisu saab vajaduse korral töötlemise juures arvestada.

Nagu eelnevalt kirjas, peab peale `RuntimeException`i alla kuuluvate erinditüüpide kõikidele muudele erindi tekkevõimalustele tähelepanu pöörama. Niimoodi kompilaator ühtlasi hoolitseb, et kahtlased programmilõigud ei jääks tähelepanuta. Nende ümber peab olema katsendiblokk koos püünisega, kuhu vastavat tüüpi erind sobib, või peab meetod lubama vastavat tüüpi erindi enesest välja:

```
public void trykiKriipse(int hulk) throws Exception{
  if(hulk<0)throw new Exception("negatiivne arv");
  else for(int i=0; i<hulk; i++)System.out.print("-");
}
```

Sel juhul tuleb loodud erindiga tegelda siinset meetodit välja kutsuvas meetodis.

Soovi või vajaduse korral saab ka ise luua uusi erinditüüpe, s.t. klassi `Exception` alamklasse.

Kokkuvõte

Erindid aitavad probleemiteavet transportida tekkekohast töötlemiskohta. Erindid püüab kokku try{} katsendiblokk ning töödelda saab neid catch-püünistes. Töötlema ei pea vaid `RuntimeException`i ning tema alamklasside erindeid.