Ohjelmistojen mallintaminen

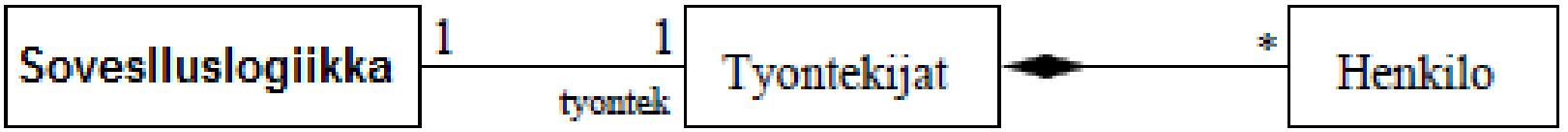
Luento 7, 23.11.

Kertaus: olioiden yhteistyön kuvaaminen

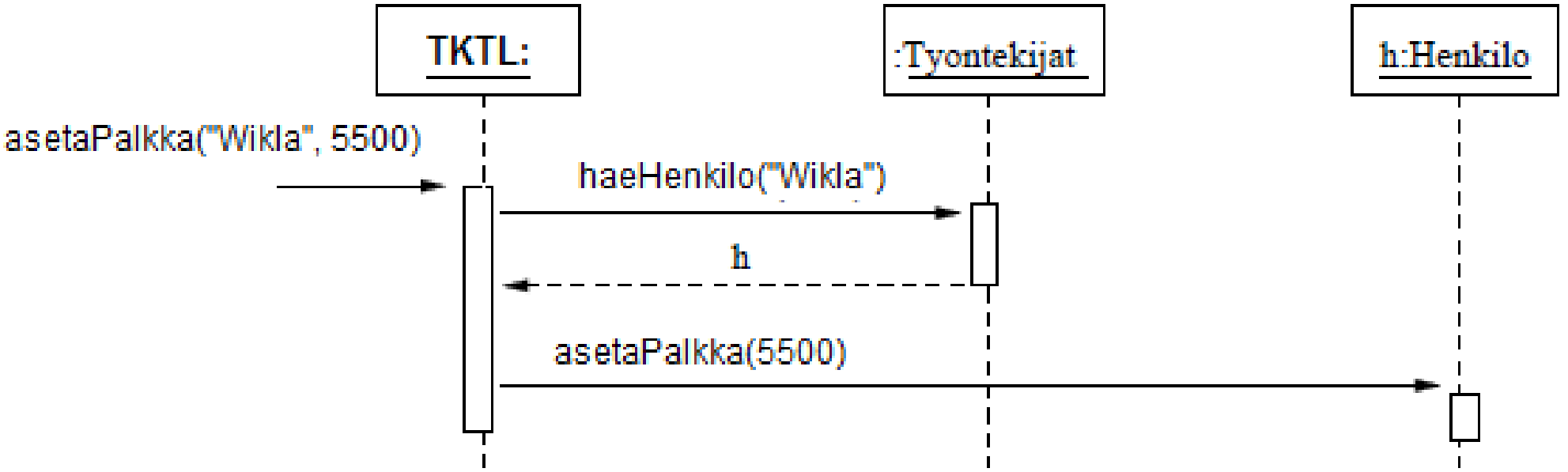
- Luokkakaavion avulla voidaan kuvata ohjelman rakenne
  - Minkälaisista luokista ohjelma koostuu ja miten luokat liittyvät toisiinsa
- Ohjelman toiminnallisuudesta vastaavat luokkien instanssit eli oliot. Oleellista olioiden yhteistyö
  - Miten oliot kutsuvat toistensa metodeita suorituksen aikana
  - Eli minkälaista yhteistyötä oliot tekevät
- Olioiden yhteistyön kuvaamiseen omat kaaviotyyppinsä:
  - Suositumpi **sekvenssikaavio**
  - ja vähemmän käytetty *kommunikaatiokaavio*
- Oliosuunnittelussa näillä kaavioilla erityisen tärkeä asema

- Jatkamme vielä hiukan sekvenssikaavioiden parissa ja esittelemme kommunikaatiokaaviot
- Seuraavassa kertauksena esimerkki viime luennolta

Järjestelmän rakenteen kuvaa luokkakaavio



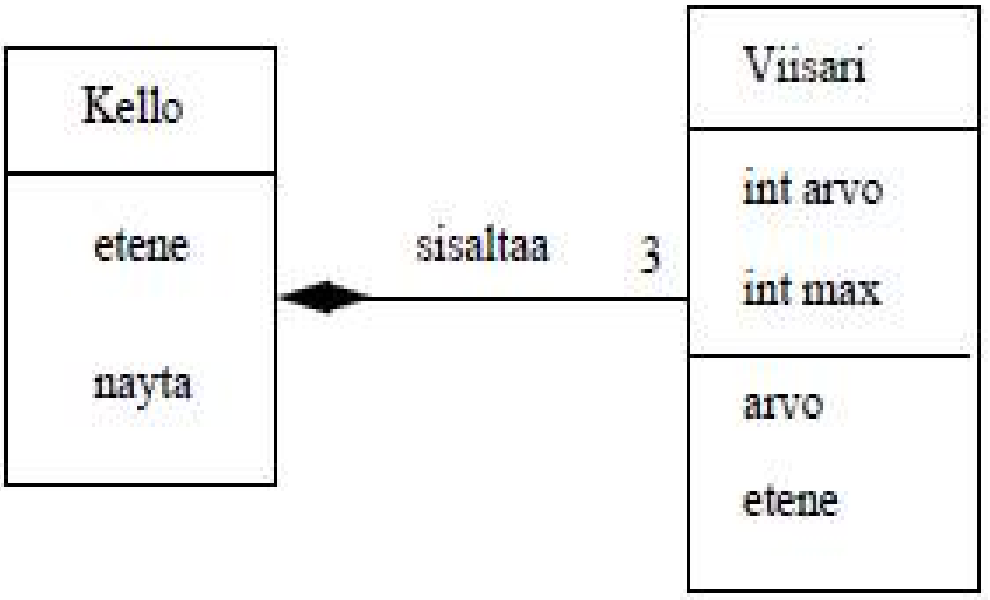
Toiminta kuvataan sekvenssikaaviolla
- jokaista toimintaskenaariota varten oma kaavio



Luokka- ja sekvenssikaaviot eivät siis "kilpaile" keskenään
- molempia tarvitaan
- molemmat tarjoavat oman näkökulmansa järjestelmään

Takaisinmallinnus ja sekvenssikaatiot

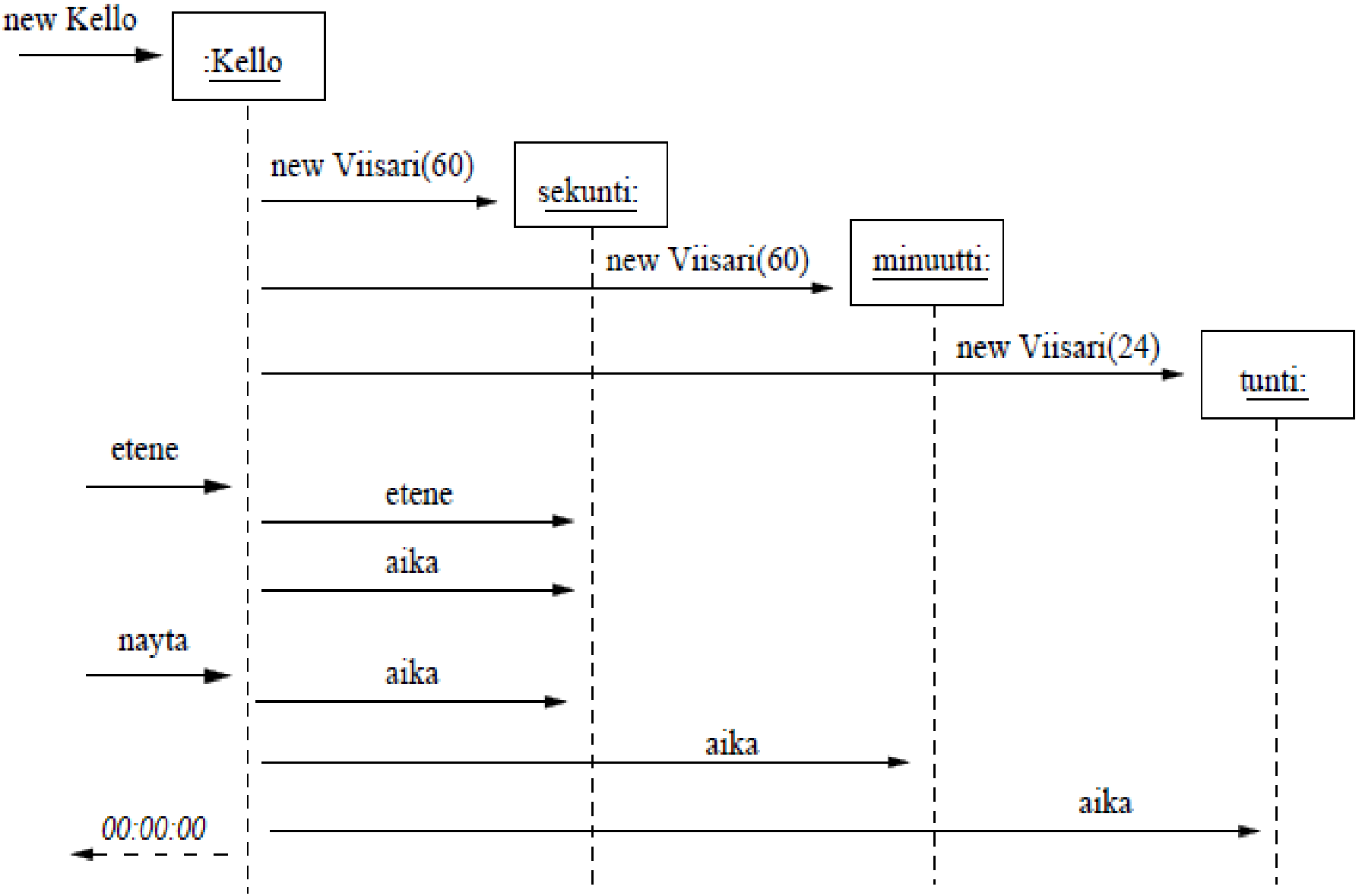
- *Takaisinmallinnuksella* (engl. reverse enginering) tarkoitetaan mallien tekemistä valmiina olevasta koodista
  - Erittäin hyödyllistä, jos esim. tarve ylläpitää huonosti dokumentoitua koodia
- Seuraavalla kalvolla Javalla toteutettu kello, joka nyt takaisinmallinnetaan
- Luokkakaavio on helppo laatia
  - Kello koostuu kolmesta viisarista
- Luokkakaaviosta ei vielä saa kuvaa kellon toimintalogiikasta joten tarvitaan sekvenssikaavioita



```
class Viisari {
  int max;      // arvo, jonka saavutettuaan
                // viisari pyörähtää nollaan
  int arvo;

  Viisari( int m ){ arvo = 0;  max = m; }

  void etene(){
     arvo++;
     if ( arvo==max ) arvo = 0;
  }

  int aika(){ return arvo; }
}
```

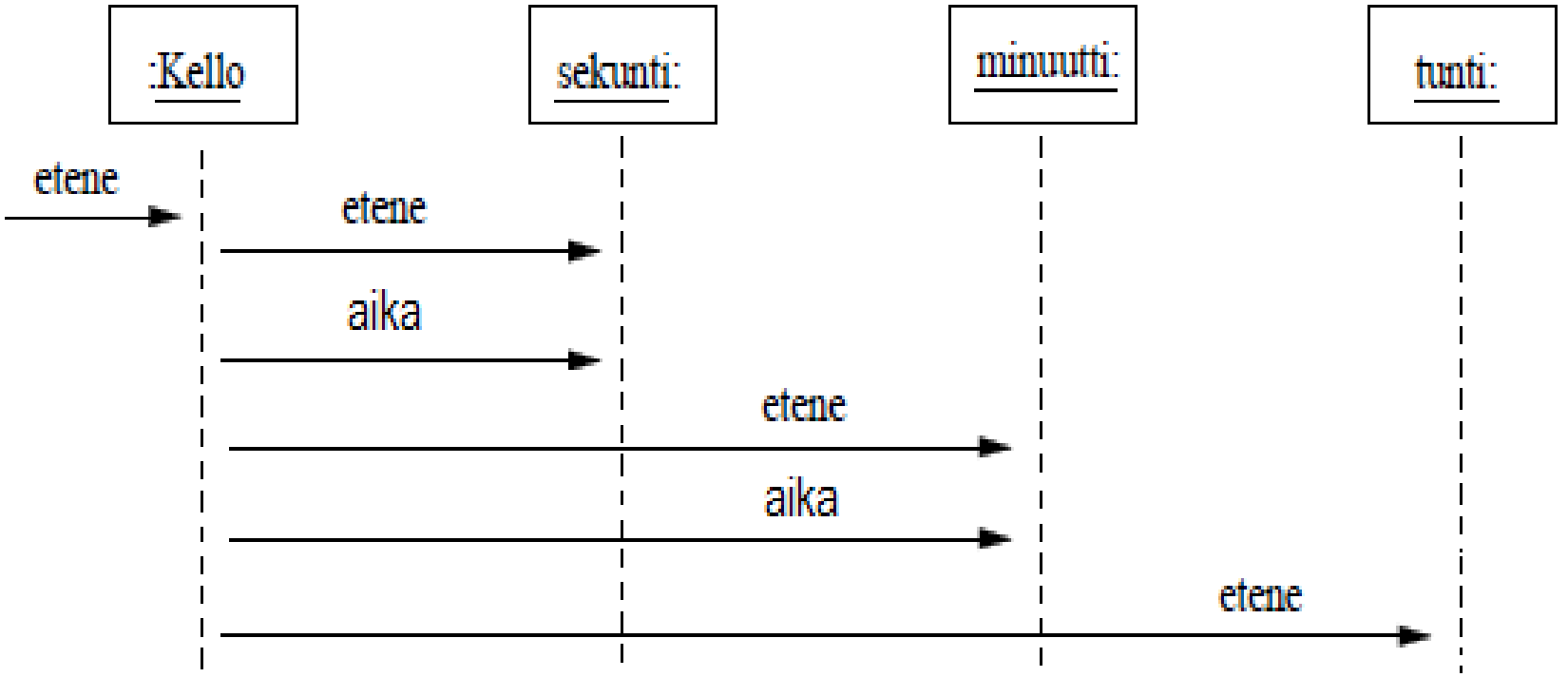
```
class Kello {
  Viisari tunti,  minuutti,  sekunti;
  Kello(){
     sekunti  = new Viisari(60);    minuutti = new Viisari(60);
     tunti    = new Viisari(24);
  }
  void etene(){
    sekunti->etene();
    if ( sekunti->aika()==0 ) {
       minuutti->etene();
       if ( minuutti->aika()==0 )   tunti->etene();
    }
  }
  void nayta(){
    System.out.print( tunti->aika() );       System.out.print(":");
    System.out.print( minuutti->aika() );  System.out.print(":");
    System.out.print( sekunti->aika() );
  }
}
```

Kellon syntyy ja lähtee käymään



Kellon eteneminen tasatunnilla

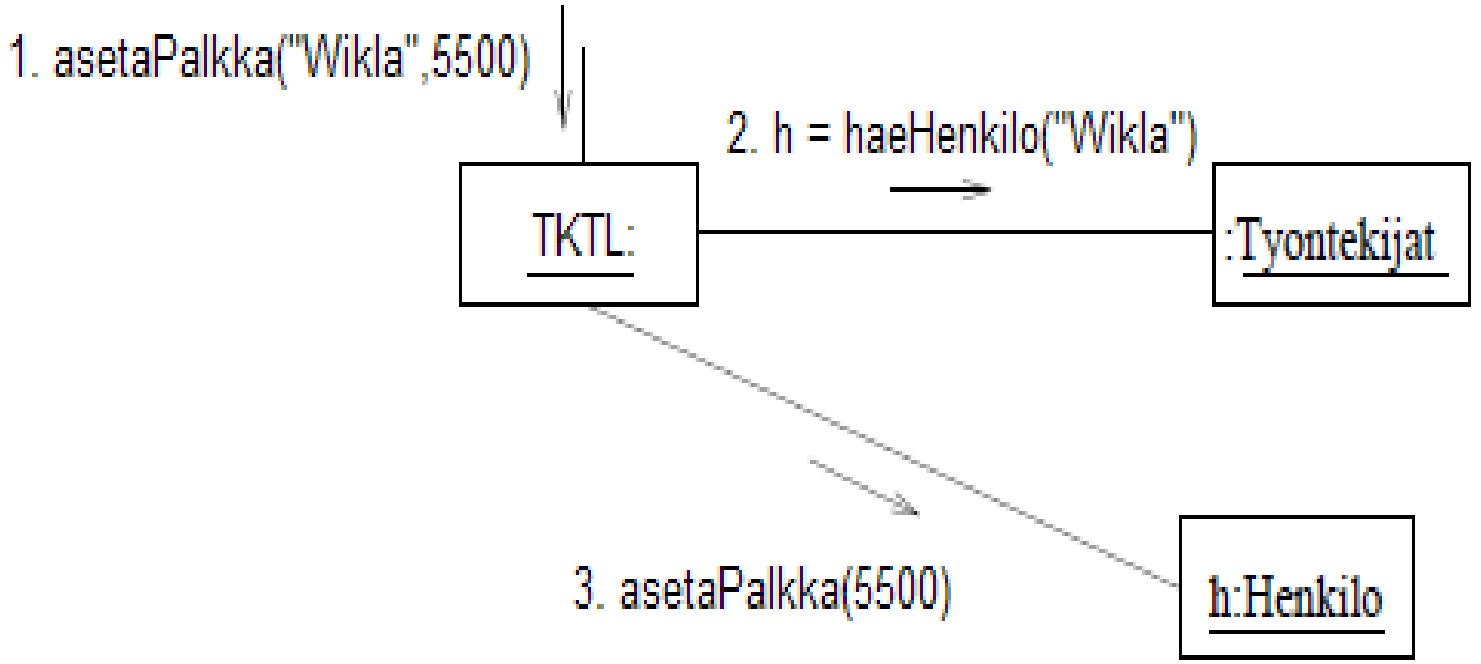
- Keskiyöllä kaikki viisarit pyörähtävät eli ”etenevät” nollaan, tilannetta kuvaava sekvenssikaavio alla



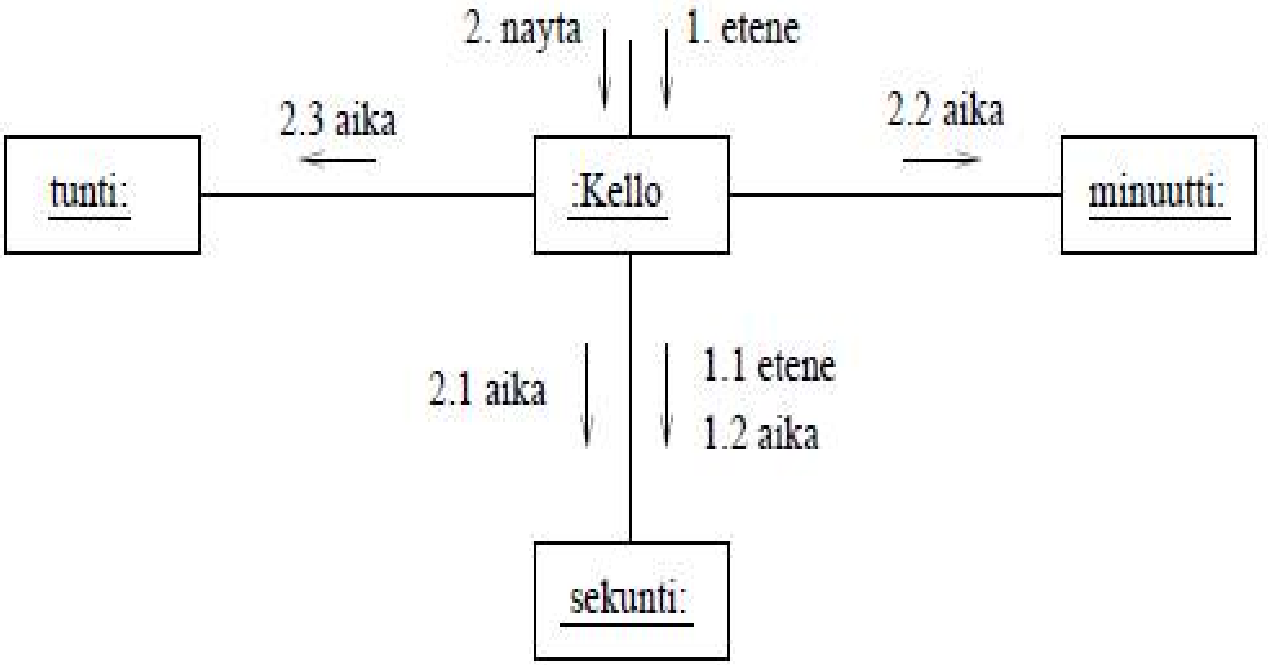
- jotta sekvenssikaavio ei kasvaisi liian suureksi, otetaan mukaan vain olennainen
  - esim. edelliseltä sivulta on jätetty pois Java-standardikirjaston out-oliolle suoritetut print()-metodikutsut

Kommunikaatiokaavio

- Toinen tapa olioiden yhteistyön kuvaamiseen on kommunikaatiokaavio (communication diagram)
- Alla muutaman sivun takainen esimerkki, jossa henkilölle asetetaan palkka
- Mukana skenaarioon osallistuvat oliot
  - Olioiden sijoittelu on vapaa
  - Kommunikoivien olioiden väliin on piirretty viiva
  - Muistuttaa oliokaaviota!
- Metodien suoritusjärjestys ilmenee numeroinnista



- Viestien järjestyksen voi numeroida juoksevasti: 1, 2, 3, …
- Tai allaolevan esimerkin (vanha tuttu Kello) tyyliin hierarkkisesti:
  - Kellolle kutsutaan metodia etene(), tällä numero 1
  - Eteneminen aiheuttaa sekuntiviisarille suoritetut metodikutsut etene() ja näytä(), nämä numeroitu 1.1 ja 1.2
  - Seuraavaksi kellolle kutsutaan metodia näytä(), numero 2
  - Sen aiheuttamat metodikutsut numeroitu 2.1, 2.2, 2.3, ...



Yhteenveto olioiden yhteistoiminnan kuvaamisesta

- Sekvenssikaavioita käytetään useammin kun kommunikaatiokaavioita
  - Sekvenssikavio lienee luokkakaavioiden jälkeen eniten käytetty UML-kaaviotyyppi
- Sekä sekvenssi- että kommunikaatiokaavioilla erittäin tärkeä asema oliosuunnittelussa
- Kaaviot kannattaa pitää melko pieninä ja niitä ei kannata tehdä kuin järjestelmän tärkeimpien toiminnallisuuksien osalta
  - Kommunikaatiokaaviot ovat yleensä hieman pienempiä, mutta toisaalta metodikutsujen ajallinen järjestys ei käy niistä yhtä hyvin ilmi kuin sekvenssikaavioista
- On epäselvää missä määrin luennolla 6 mainituja sekvenssikaavioiden valinnaisuutta ja toistoa kannattaa käyttää

- Sekvenssikaaviot on alunperin kehitetty tietoliikenneprotokollien kuvaamista varten

Yleistys-erikoistus ja periminen

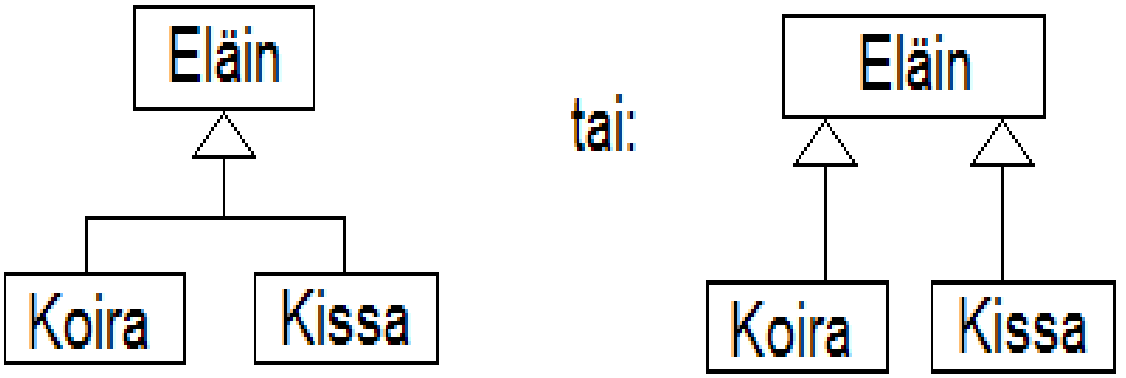
- Tähän mennessä tekemissämme luokkakaaviossa kaksi luokkaa ovat voineet liittyä toisiinsa muutamalla tapaa
- *Yhteys ja kompositio* liittyvät tilanteeseen, missä luokkien olioilla on rakenteellinen (= jollain lailla pysyvä) suhde, esim.:
  - Henkilö *omistaa* Auton (*yhteys*: normaali viiva)
  - Huoneet *sijaitsevat* Talossa (*kompositio*: musta salmiakki)
    - Musta salmiakki tarkoittaa olemassaoloriippuvuutta, eli salmiakin toisen pää olemassaolo riippuu salmiakkipäässä olevasta
    - Jos talo hajotetaan, myös huoneet häviävät, huoneita ei voi siirtää toiseen taloon
- Löyhempi suhde on taas *riippuvuus*, liittyy ohimenevämpiin suhteisiin, kuten tilapäiseen käyttösuhteeseen, esim:
  - AutotonHenkilö *käyttää* Autoa (katkoviivanuoli)
- Tänään tutustumme vielä yhteen hieman erilaiseen luokkien väliseen suhteeseen, eli *yleistys-erikostussuhteeseen,* jonka vastine ohjelmoinnissa on *periminen*

Yleistys-erikoistus ja periminen

- Ajatellaan luokkia Eläin, Kissa ja Koira
- Kaikki Koira-luokan oliot ovat selvästi myös Eläin-luokan oliota, samoin kaikki Kissa-luokan oliot ovat Eläin-luokan olioita
- Koira-oliot ja Kissa-oliot ovat taas täysin eriäviä, eli mikään koira ei ole kissa ja päinvastoin
- Voidaankin sanoa, että luokkien Eläin ja Koira sekä Eläin ja Kissa välillä vallitsee yleistys-erikoistussuhde:
  - Eläin on **yliluokka** (superclass)
  - Kissa ja Koira ovat eläimen **aliluokkia** (engl. Subclass)
- Yliluokka Eläin siis määrittelee mitä tarkoittaa olla eläin
  - Kaikkien mahdollisten eläinten yhteiset ominaisuudet ja toiminnallisuudet
- Aliluokassa, esim. Koira tarkennettaan mitä muita ominaisuuksia ja toiminnallisuutta luokan olioilla eli Koirilla on kuin yliluokassa Eläin on määritelty
- Aliluokat siis *perivät* (engl. inherit) yliluokan ominaisuudet ja toiminnallisuuden

Yleistys-erikoistus ja periminen

- Luokkakaaviossa yleistyssuhde merkitään siten, että **aliluokasta piirretään yliluokkaan kohdistuva nuoli, jonka päässä on iso ”valkoinen” kolmio**
- Jos aliluokkia on useita, voivat ne jakaa saman nuolenpään tai molemmat omata oman nuolensa, kuten alla



- Tarkkamuistisimmat huomaavat ehkä, että olemme jo törmänneet kurssilla yleistys-erikoisstussuhteeseen *käyttötapausten* yhteydessä
  - Luennolta 2: Yleistetty käyttötapaus *opetustarjonnan ylläpito* erikoistui *kurssin perustamiseen*, *laskariryhmän perustamiseen* ym...
  - Sama valkoinen kolmiosymbooli oli käytössä myös käyttötapausten yleistyksen yhteydessä

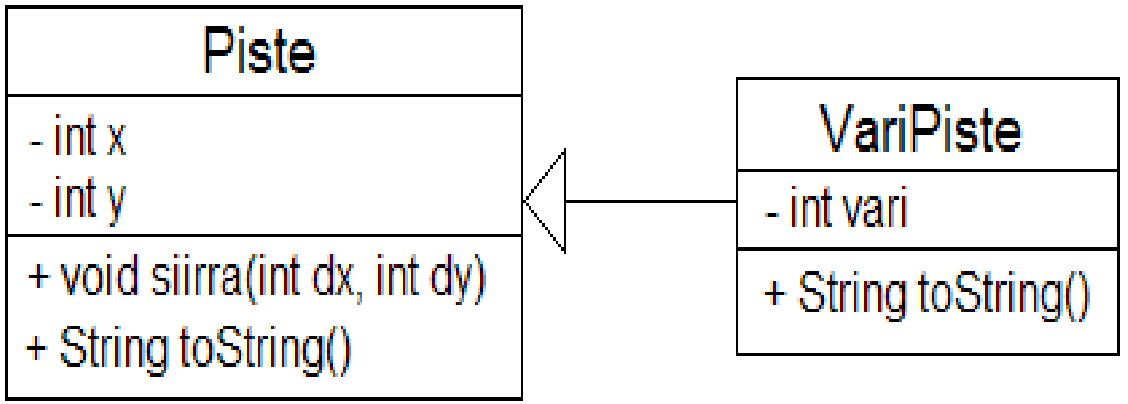
Periytyminen

- Luokkien välinen yleistys-erikoistussuhde eli yli- ja aliluokat toteutetaan ohjelmointikielissä siten, että aliluokka perii yliluokan
- Tuttu esimerkki Ohjelmoinnin jatkokurssilta, konstrutoreja ei merkitty:
  - Luokkakaavio seuraavalla sivulla

```
class Piste{
    private int x, y;
    public void siirra(int dx, int dy){
        x+=dx; y+=dy;
    }
    public String toSting(){ return ”(”+x+”)”; }
}
Class VariPiste extends Piste {
    private string vari;
    public String toSting(){ return super.toString()+” väri: ”+vari; }
}
```

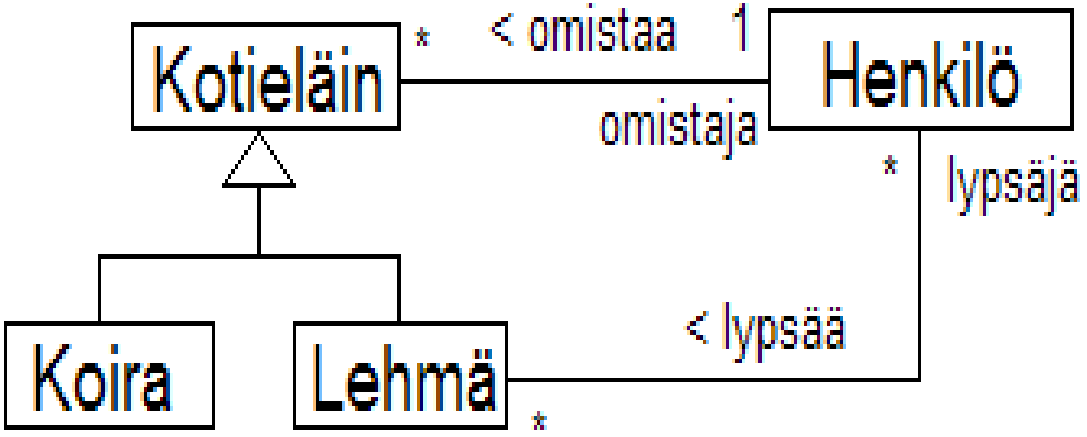
Periytyminen ja luokkakaavio

- Yliluokan Piste attribuutit x ja y sekä metodi siirra() siis periytyvät aliluokkaan VariPiste
  - Perityviä attribuutteja metodeja ei merkitä aliluokan kohdalle
- Jos ollaan tarkkoja, Piste-luokan metodi toString periytyy myös VariPiste-luokalle, joka *syrjäyttää* (engl. override) perimänsä omalla toteutuksella
  - Korvaava toString()-metodi merkitään aliluokkaan VariPiste
- Eli kuviosta on pääteltävissä, että VariPisteellä on:
  - Atribuutit x ja y sekä metodi siirra perittynä
  - Attribuutti vari, jonka se määrittelee itse
  - Itse määritelty metodi toString joka syrjäyttää yliluokalta perityn
  - Koodista nähdään, että korvaava metodi käyttää yliluokassa määriteltyä metodia



Mitä kaikkea periytyy?

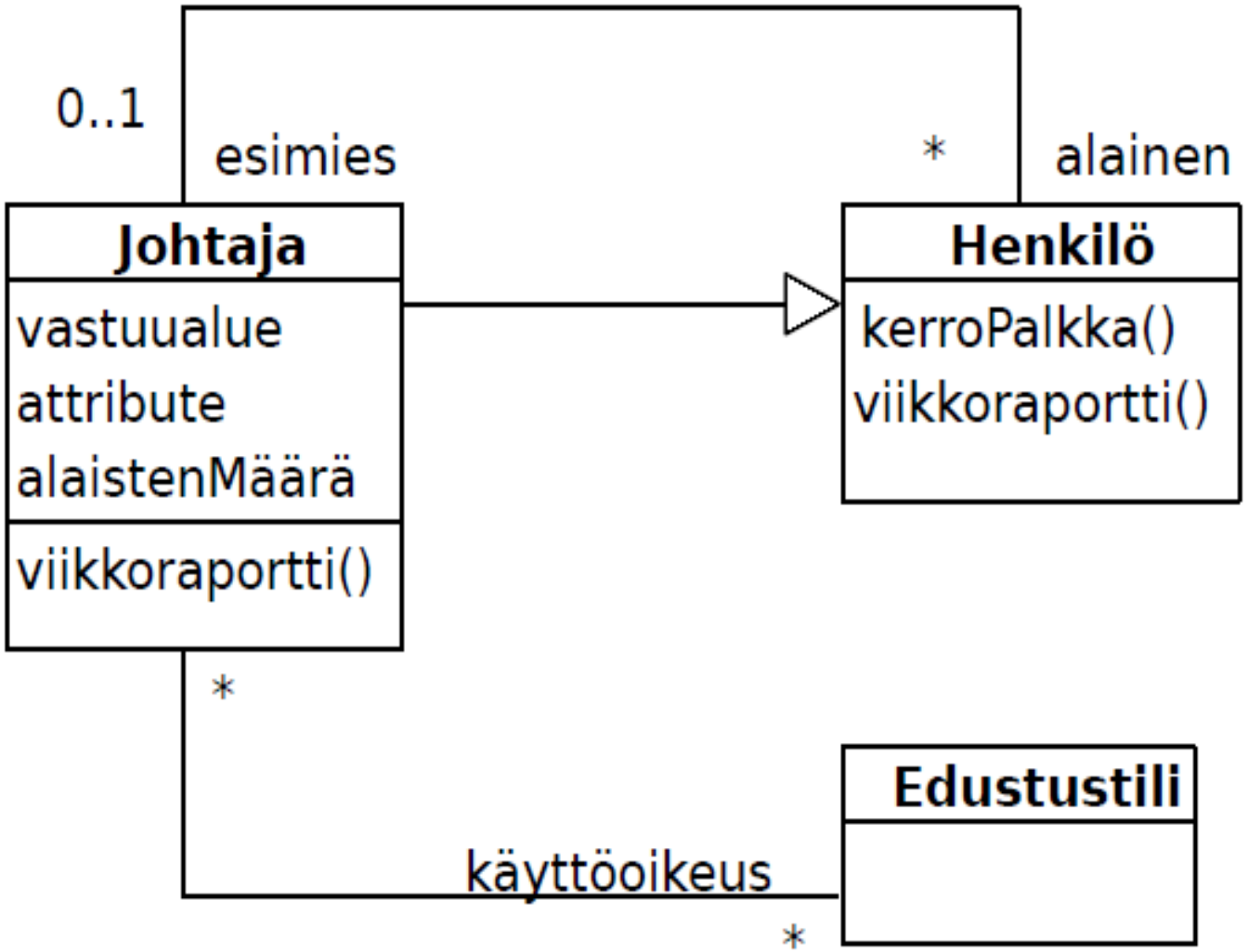
- Luokat Koira ja Lehmä ovat molemmat luokan Kotieläin aliluokkia
- Jokaisella kotieläimellä on omistajana joku Henkilö-olio
- Koska omistaja liittyy kaikkiin kotieläimiin, merkitään yhteys Kotieläin- ja Henkilö-luokkien välille
  - **Yhteydet periytyvät aina aliluokille**, eli Koira-olioilla ja Lehmä-olioilla on omistajana yksi Henkilö-olio
- Ainoastaan Lehmä-olioilla on lypsäjiä
  - Yhteys lypsää tuleekin Lehmän ja Henkilön välille



Aliluokan ja yliluokan välinen yhteys

- Yrityksen työntekijää kuvaa luokka Henkilö
  - Henkilöllä on metodit kerroPalkka() ja viikkoraportti()
- Johtaja on Henkilön aliluokka
  - Johtajalla on alaisena useita henkilöitä
  - Henkilöllä on korkeintaan yksi johtaja esimiehenä
  - Johtajalla voi olla käyttöoikeuksia Edustustileihin
  - Edustustilillä on useita käyttöoikeuden omaavia johtajia
  - Johtajan viikkoraportti on erilainen kuin normaalin työntekijän viikkoraportti
- Tilannetta kuvaava luokkakaavio seuraavalla sivulla

Osa yrityksen luokkakaaviota



Aliluokan ja yliluokan välinen yhteys

- Johtaja siis perii kaiken Henkilöltä
  - Henkilö on *alainen*-roolissa yhteydessä nollaan tai yhteen Johtajaan
  - Tästä seuraa, että *myös Johtaja-olioilla on sama yhteys, eli myös johtajilla voi olla johtaja!*
- Metodi viikkoraportti on erilainen johtajalla kuin muilla henkilöillä, siispä Johtaja-luokka korvaa Henkilö-luokan metodin omallaan
- Esim. Henkilö-luokan metodi viikkoraportti():

  Kerro ajankäyttö työtehtäviin
- Johtaja-luokan korvaama metodi viikkoraportti():

  Kerro ajankäyttö työtehtäviin

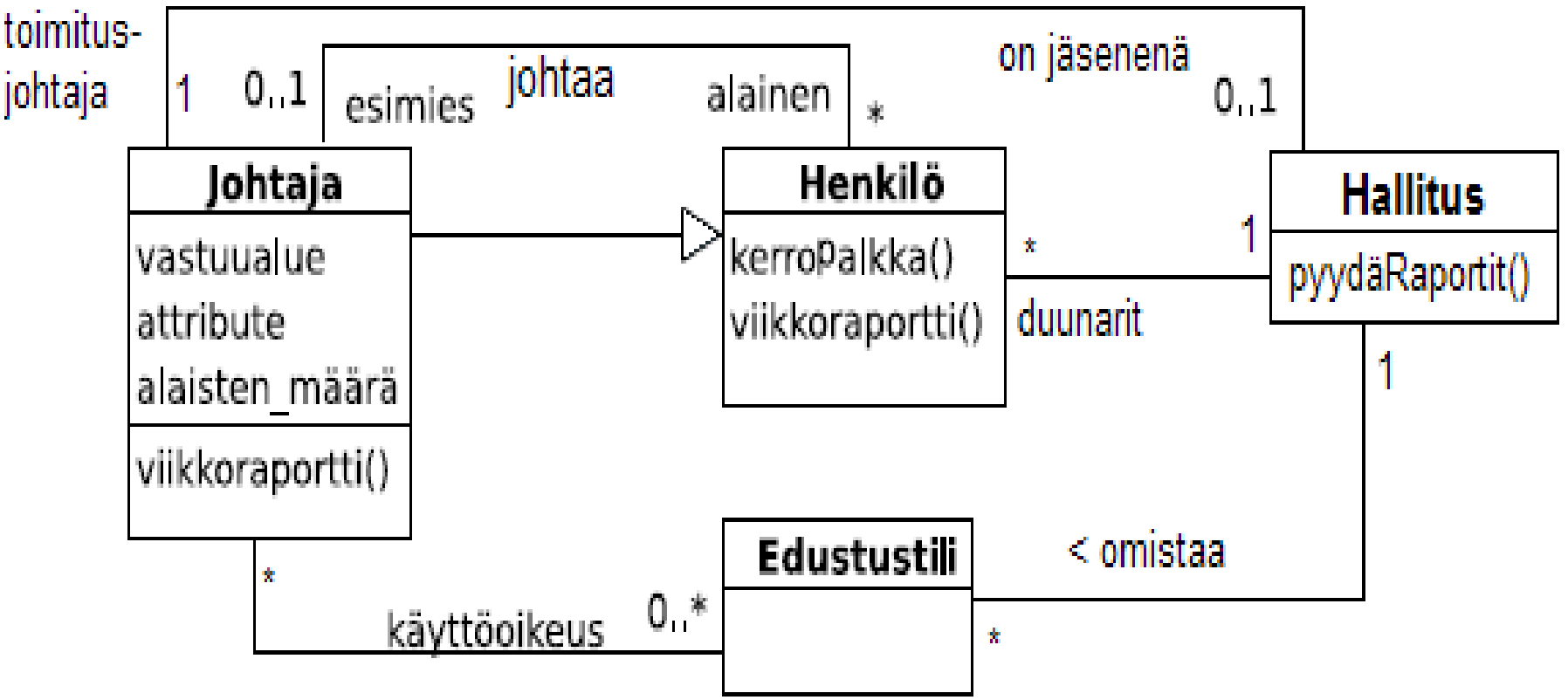
  Laadi yhteenveto alaisten viikkoraporteista

  Raportoi edustustilin käytöstä
- Yhteys käyttöoikeus Edustustileihin voi siis ainoastaan olla niillä henkilöillä jotka ovat johtajia

Laajennetaan mallia

- Yrityksen hallitus koostuu ulkopuolisista henkilöistä (joita ei sisällytetä malliin) ja yrityksen toimitusjohtajasta joka siis kuuluu henkilöstöön
  - Hallitus on edustustilien omistaja
  - Hallitus ”tuntee” toimitusjohtajaa lukuunottamatta kaikki työntekijät, myös normaalit johtajat ainoastaan Henkilö-olioina
- Hallitus pyytää työntekijöiltä viikkoraportteja
  - Viikkoraportin tekevät kaikki paitsi toimitusjohtaja

Hallitus mukana kuvassa



Olio tietää luokkansa

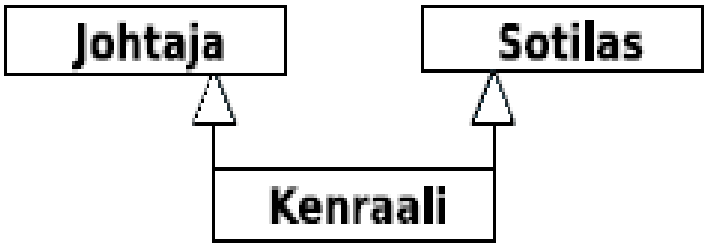
- Hallitus siis tuntee kaikki työntekijänsä, mutta ei erittele ovatko he normaaleja työntekijöitä vai johtajia
  - Hallituksen koodissa kaikkia työntekijöitä pidetään Henkilö-oliosta koostuvassa listalla *duunarit*. Johtajathan ovat myös henkilöitä!
- Hallituksen ei siis ole edes tarvetta tuntea kuka on johtaja ja kuka ei
- Pyytäessään viikkoraporttia, hallitus käsittelee kaikkia samoin:

```
class Hallitus{
  ArrayList<Henkilo> duunarit ;          // attribuutti, joka sisältää kaikki työntekijät
  Johtaja toimitusjohtaja;               // attribuutti, joka tietää toimitusjohtajan
  void pyydaRaportit(){                  // Javan for-each-lause, eli d käy läpi koko listan
    for ( d : duunarit ) {   if ( d != toimitusjohtaja ) d.viikkoraportti()  }
  }
```

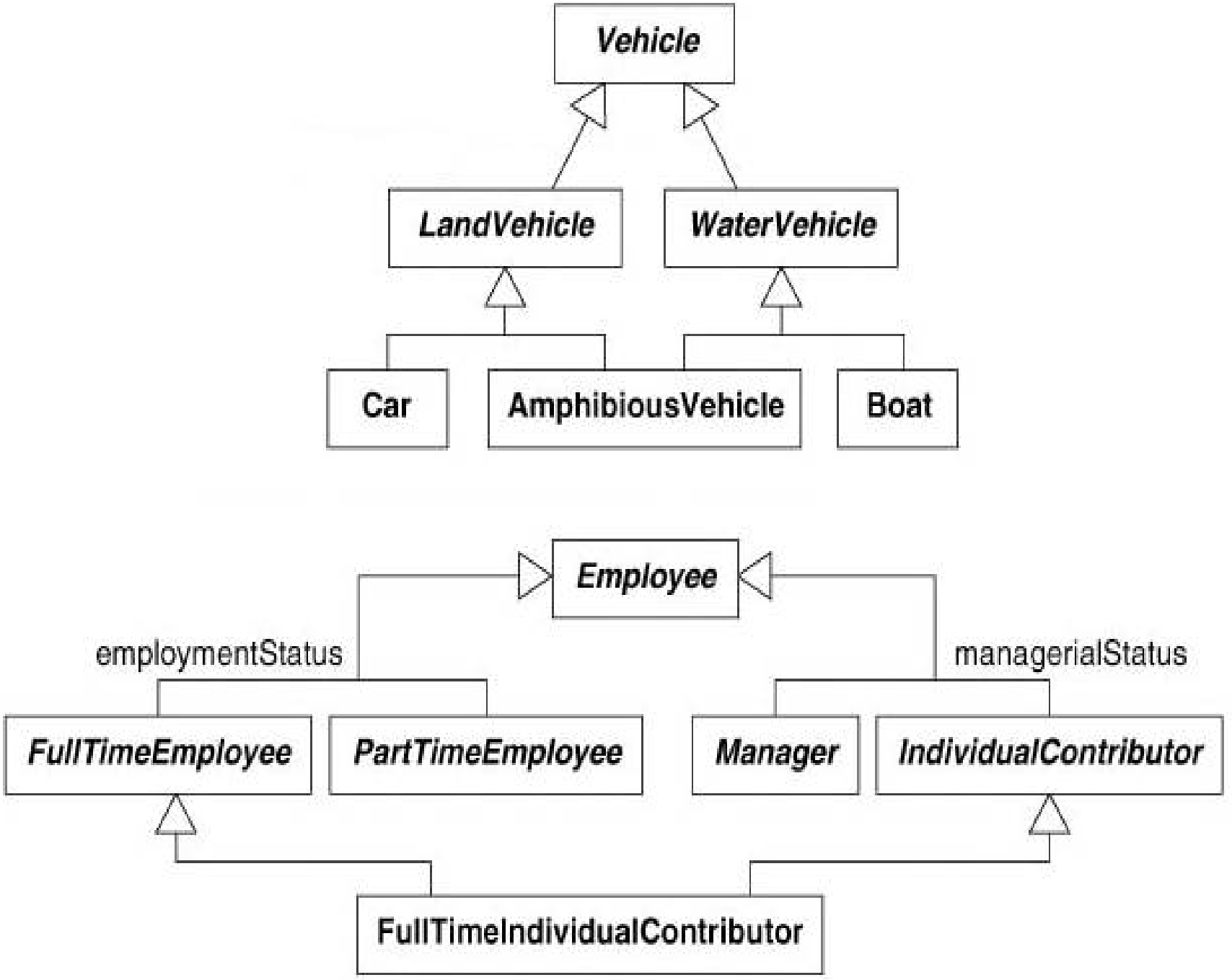
- Jokainen duunari tuntee ”oikean” luokkansa
- Kun hallitus kutsuu duunarille metodia viikkoraportti(), jos kyseessä on normaali henkilö, suoritetaan henkilön viikkoraportointi, jos taas kyseessä on johtaja, suoritetaan johtajan viikkoraportti (*polymorfismia!*)

Moniperintä

- Joskus tulee esiin tilanteita, joissa yhdellä luokalla voisi kuvitella olevan useita yliluokkia
- Esim. kenraalilla on sekä sotilaan, että johtajan ominaisuudet
  - Voitaisiin tehdä luokat Johtaja ja Sotilas ja periä Kenraali näistä
- Kyseessä *moniperintä* (engl. multiple inheritance)
- Seuraavalla sivulla: Kulkuneuvo jakautuu maa- ja merikulkuneuvoksi
  - Auto on maakulkuneuvo, vene merikulkuneuvo, amfibio sekä maa- että merikulkuneuvo
- Työntekijät voi jaotella kahdella tavalla:
  - Pää- ja sivutoimiset
  - Johtajat ja normaalit
  - Yksittäinen työntekijä voi sitten olla esim. päätoiminen johtaja



Lisää moniperintää

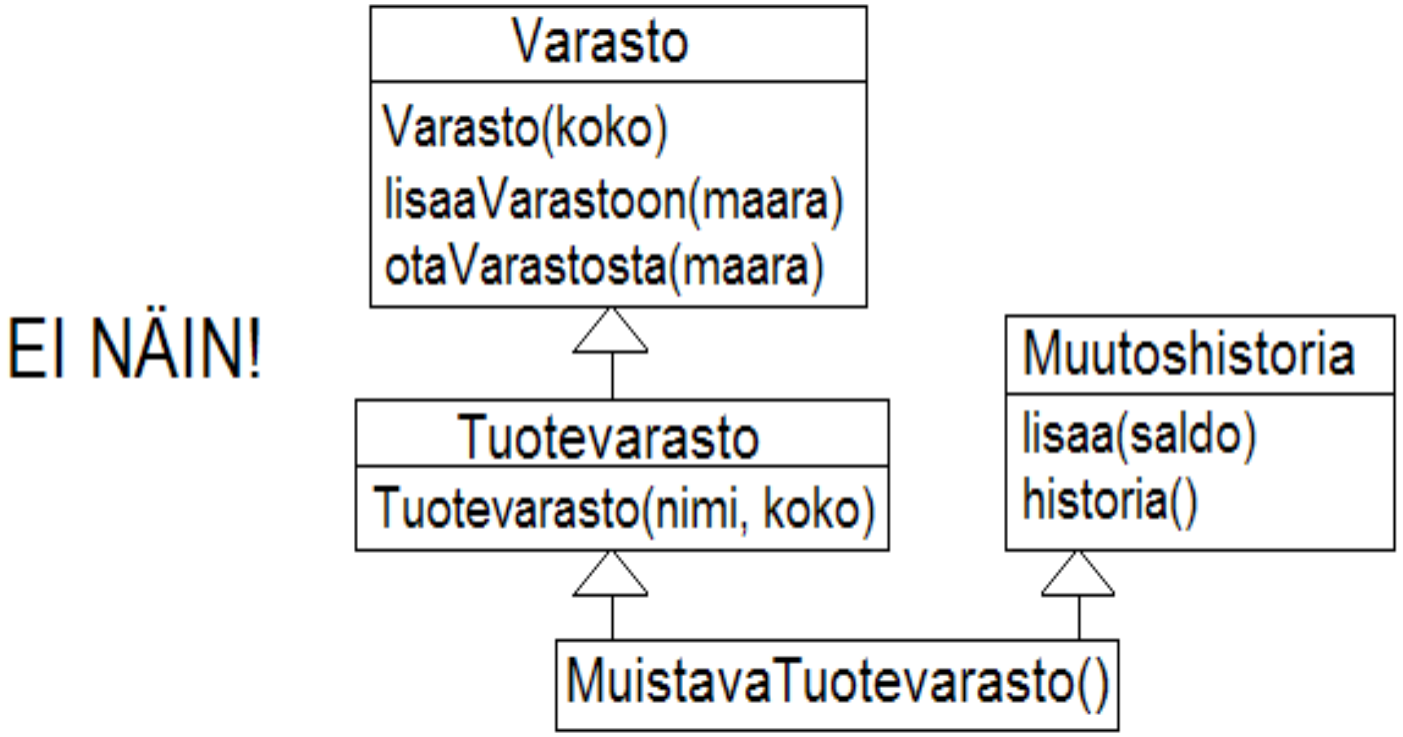


Moniperinnän ongelmat

- Moniperintä on monella tapaa ongelmallinen asia ja useat kielet, kuten Java eivät salli moniperintää
  - C++ sallii moniperinnän
  - ”moderneissa” kielissä kuten Python, Ruby ja Scala on olemassa ns. mixin-mekansismi, joka mahdollistaa ”hyvinkäyttäytyvän” moniperintää vastavan mekanismin
- Onkin viisasta olla käyttämättä moniperintää ja yrittää hoitaa asiat muin keinoin
- Näitä muita keinoja (jos unohdetaan mixin-mekanismi) ovat:
  - Moniperiytymisen korvaaminen yhteydellä, eli käytännössä ”liittämällä” olioon toinen olio, joka laajentaa alkuperäisen olion toiminnallisuutta
  - Javasta löytyvät rajapintaluokat (interface) toimivat joissain tapauksessa moniperinnän korvikkeena
- Ohjelmoinnin jatkokurssin viikon 3 laskareissa, ks http://www.cs.helsinki.fi/u/wikla/ohjelmointi/jatko/s2010/harjoitukset/3/ toteutetaan MuistavaTuotevarasto, joka on luokka johon lisätään toiminnallisuutta perimisen sijaan liittämällä siihen toinen olio

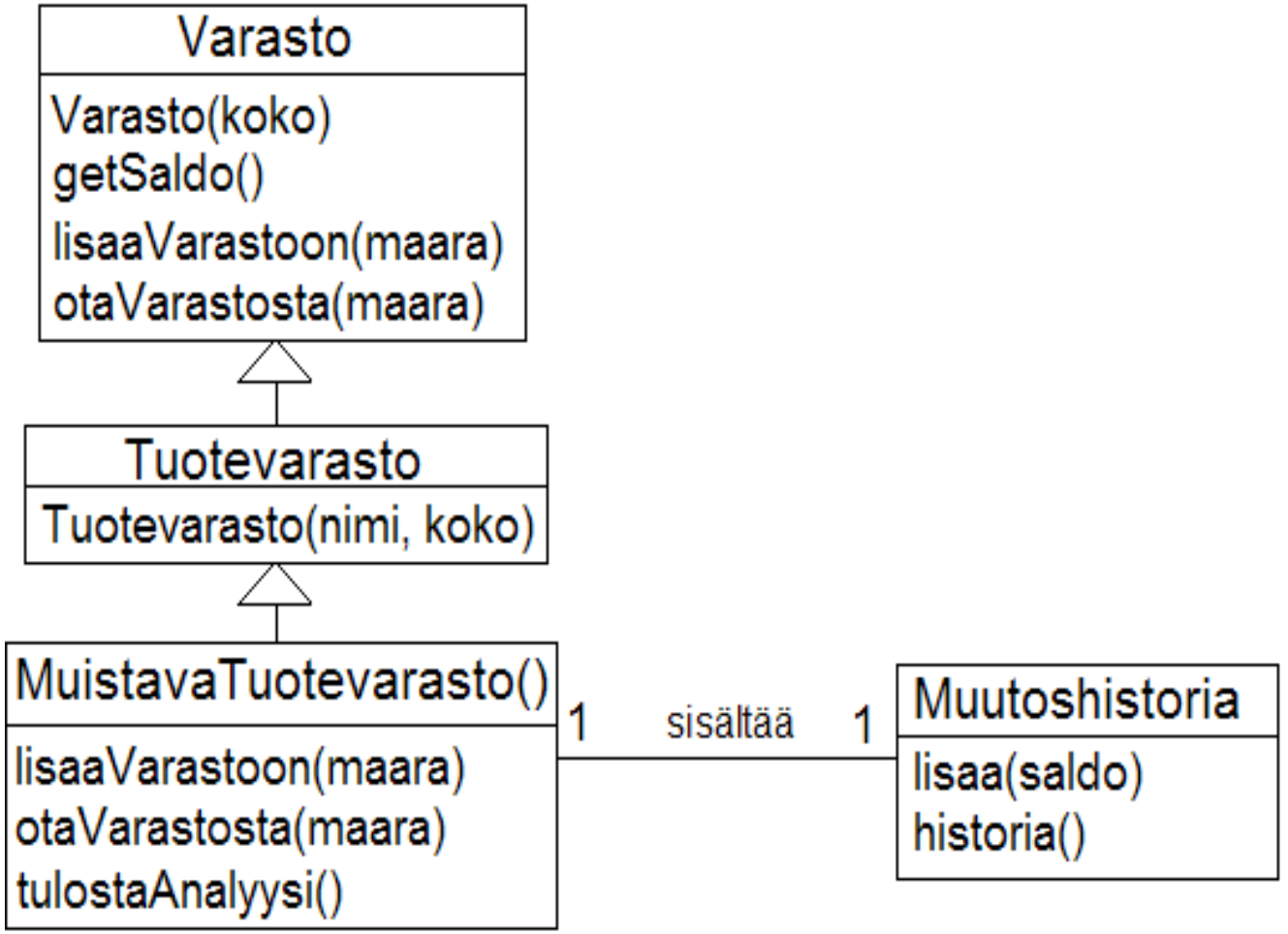
Muistava tuotevarasto

- Luokka Tuotevarasto toteutetaan perimällä vanha tuttu Varasto
  - Tuotteelle lisätään nimi
- Viikon 3 tehtävissä 3.1-3.3 toteutetaan luokka Muutoshistoria
  - Käytännössä kyseessä on lista double-lukuja, joiden on tarkotus kuvata peräkkäisiä varastosaldoja
- Tehtävässä 3.4-3.7 toteutetaan MuistavaTotevarasto joka yhdistää edellisten toiminnallisuuden
- Joku C++-ohjelmoija soveltaisi tilanteessa kenties moniperintää:



Muistava tuotevarasto

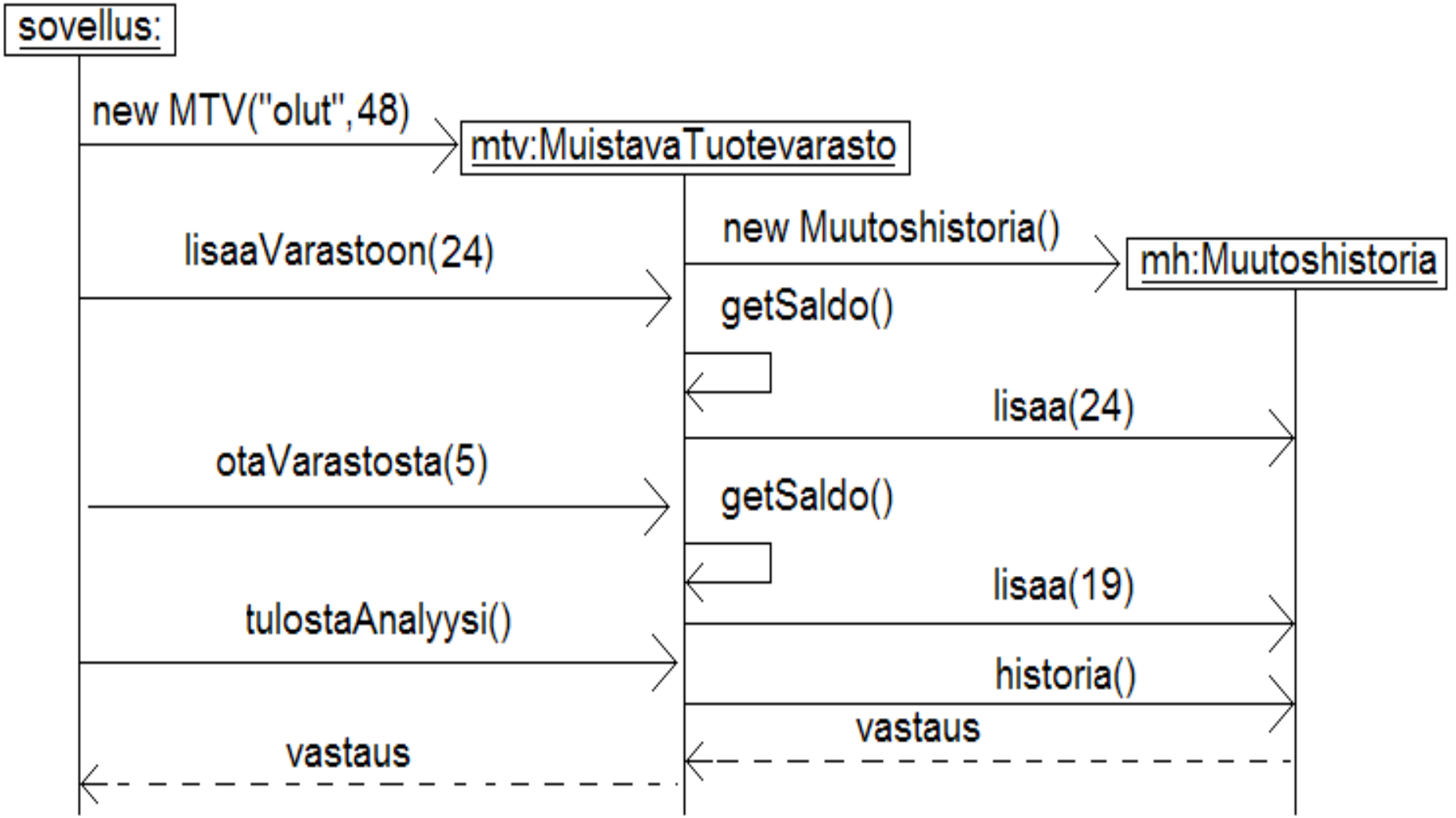
- Javassa ei moniperintää ole, ja vaikka olisikin, on parempi liittää ”muistamistoiminto” muistavaan tuotevarastoon erillisenä oliona:



- Aina kun muistavan tuotevaraston saldo päivittyy (metodien lisaaVarastoon ja otaVarastosta yhteydessä), samalla laitetaan uusi saldo muutoshistoriaan

Muistavan varaston toimintaa kuvaava sekvenssikaavio

- Sovellus luo muistavan tuotevaraston joka tallettaa olutta ja kapasiteetti on 48
- MuistavaTuotevarasto luo käyttöönsä Muutoshistoriaolion
- Aina kun varaston tilanne muuttuu, selvitetään saldo ja välitetään se muutoshistorialle
  - Kaavio ei sisällytö seuraavalla sivulla koodissa näkyviä super-kutsuja
- Analyysin tulostus delegoituu muutosvaraston hoidettavaksi



Muistavan tuotevaraston koodihahmotelma

```
MuistavaTuotevarasto extends Tuotevarasto {
    private Muutoshistoria varastotilanteet;
    MuistavaTuotevarasto(String tuote, double koko){
        super(tuote, koko);
        varastotilanteet = new Muutoshistoria();   // luodaan oma kirjanpito-olio
    }
    void lisaaVarastoon(double maara){
        super.lisaaVarastoon(maara);              // yliluokan metodi päivittää varastotilanteen
        double sadlo = getSaldo();
        varastotilanteet.lisaa( saldo );           // kerrotaan saldo kirjanpito-oliolle
    }
    String tulostaAnalyysi() {
        return varastotilanteet.historia();        // delegoidaan raportin luominen kirjanpito-oliolle
    }
}
```

Kaksi tärkeää oliosuunnittelun periaatetta

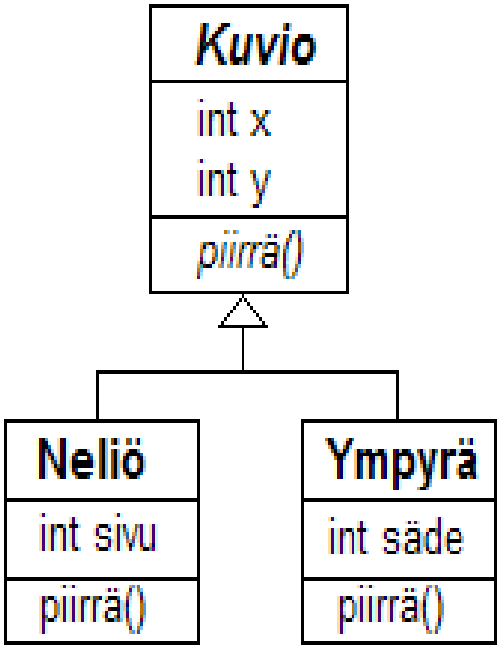
- Esimerkissä toteutuu kaksi tärkeää oliosuunnittelun periaatetta
- **Single responsibility**
  - Tarkoittaa karkeasti ottaen että **oliolla tulee olla vain yksi vastuu**
  - MuuttuvaTuotevarasto toteuttaa periaatetta koska sen vastuulla on vain varaston nykyisen tilanteen ylläpito
  - Se *delegoi* vastuun aikaisempien varastosaldojen muistamisesta Varastokirjanpito-oliolle
- **Favour composition over inheritance**
  - eli **suosi yhteistominnassa toimivia oliota perinnän sijaan**
  - Perinnällä on paikkansa, mutta sitä tulee käyttää harkiten
  - Esimerkissä käytettiin perintää järkevästi
  - Jos olisi moniperitty Tuotevarasto ja Muutoshistoria, olisi muodostettu luokka joka rikkoo single responsibility -periaatteen
- Onko näissä periaatteissa järkeä: Kyllä, ne lisäävät ohjelmien ylläpidettävyyttä
- Pitääkö periaatteita noudattaa: useimmiten. Joskus voi olla jonkun muun periaatteen nojalla viisasta rikkoa jotain periaatetta...
- Muitakin periaatteita on, joitakin niistä esitellään myöhemmin kurssilla

Abstraktit luokat

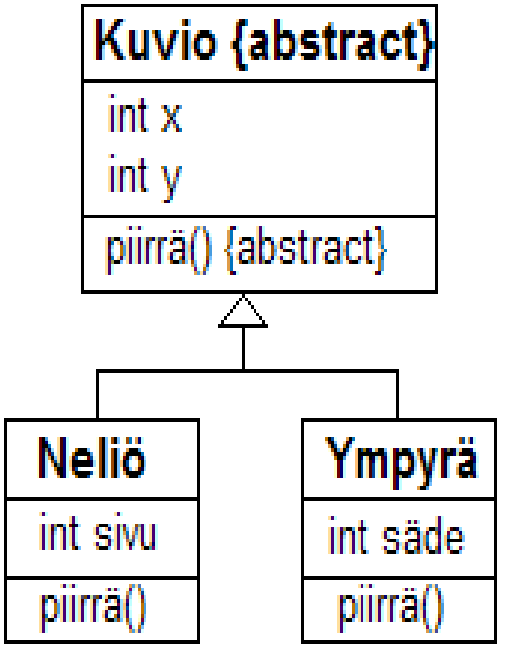
- Yliluokalla Kuvio on sijainti, joka ilmaistaan x- ja y-kordinaatteina sekä metodi piirrä()
- Kuvion aliluokkia ovat Neliö ja Ympyrä
  - Neliöllä on sivun pituus ja Ympyrällä säde
- Kuvio on nyt pelkkä abstrakti käsite, Neliö ja Ympyrä ovat konkreettisia kuvioita jotka voidaan piirtää ruudulle kutsumalla sopivia grafiikkakirjaston metodeja
- Kuvio onkin järkevä määritellä *abstraktiksi luokaksi*, eli luokaksi josta ei voi luoda instansseja, joka ainoastaan toimii sopivana yliluokkana konkreettisille kuvioille
- Kuviolla on attribuutit x ja y, mutta metodi piirrä() on *abstrakti metodi*, eli Kuvio ainoastaan määrittelee metodin nimen ja parametrien sekä paluuarvon tyypit, mutta *metodille ei anneta mitään toteutusta*
- Kuvion perivät luokat Neliö ja Ympyrä antavat toteutuksen abstraktille metodille
  - Neliö ja Ympyrä ovatkin normaaleja luokkia, eli niistä voidaan luoda olioita
- Luokkakaavio seuraavalla sivulla

Abstrakti luokka

- Luokkakaaviossa on kaksi tapaa merkitä abstraktius
  - Abstraktin luokan/metodin nimi kursiivilla, tai
  - liitetään abstraktin luokan/metodin nimeen tarkenne {abstract}



tai:




```
public abstract class Kuvio{
   int x;
   int y;
   public abstract void piirrä();
}
```

```
public class Neliö extends Kuvio {
   int sivu;
   public void piirrä(){
        graphics.drawRect(x, y, x+sivu, y+sivu);
   }
```

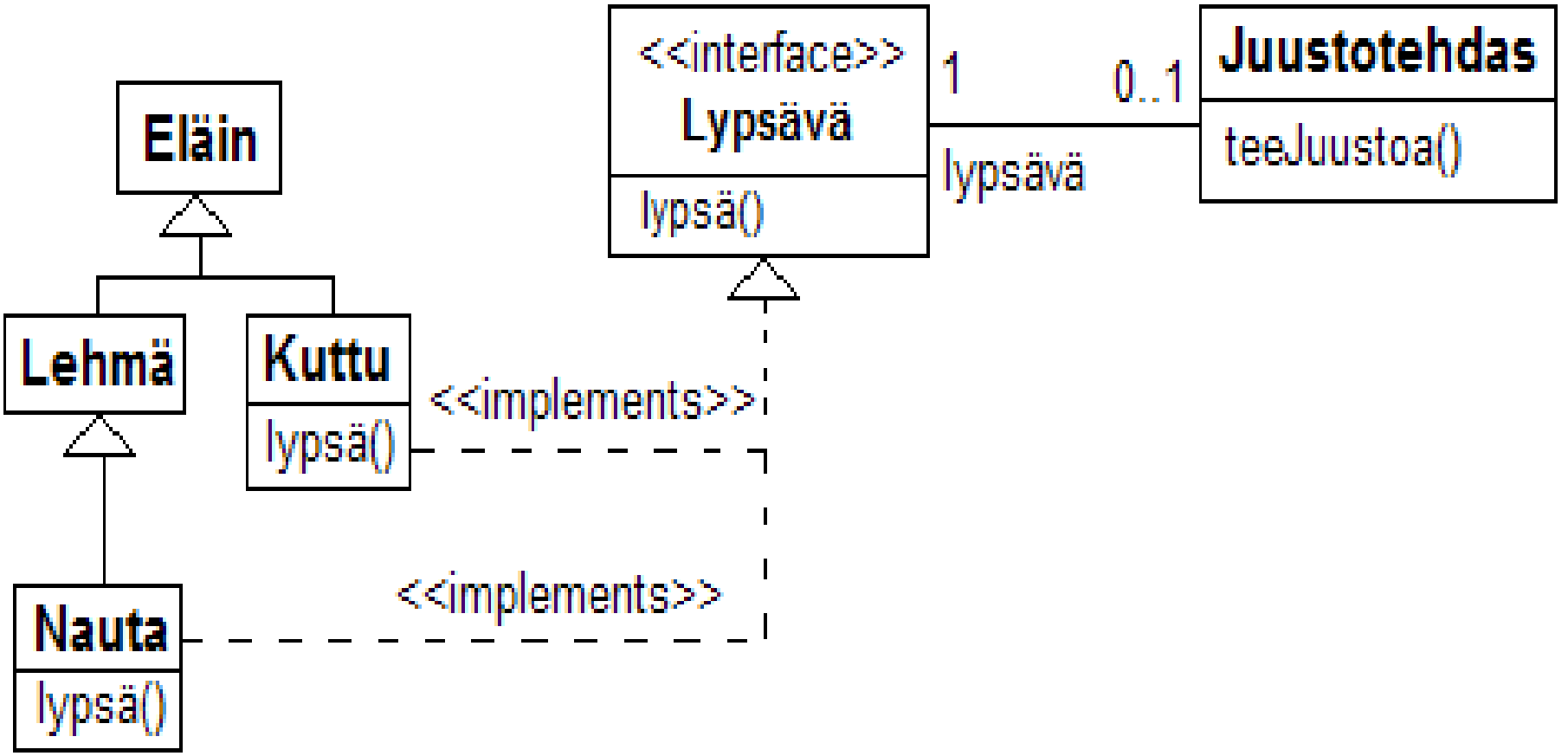
Rajapintaluokka

- Abstraktiudella ei ole mitään tekemistä moniperiytymisongelman kanssa, eli periä saa vain yhden luokan oli se abstrakti tai ei
- Javan *rajapintaluokka* (interface) on ikäänkuin abstrakti luokka joka ei sisällä attribuutteja ja jonka kaikki metodit ovat abstrakteja
- Rajapintaluokka siis listaa ainoastaan joukon metodien nimiä
- Yksi luokka voi *toteuttaa* useita rajapintoja
  - Ja sen lisäksi vielä periä yhden luokan

- Perimällä luokka saa yliluokasta attribuutteja ja metodeja
- Rajapinnan toteuttaminen on pikemminkin velvollisuus
  - Jos luokka toteuttaa rajapinnan, sen täytyy toteuttaa kaikki rajapinnan määrittelemät operaatiot
- Tai toisinpäin ajateltuna, **rajapinta on sopimus**
  - **toteuttaja lupaa toteuttaa ainakin rajapinnan määrittelemät operaatiot**

Tuttu esimerkki rajapintaluokan käytöstä

- Mallinnetaan Ohjelmoinnin jatkokurssin rajapintaesimerkki, ks:
  - http://www.cs.helsinki.fi/u/wikla/ohjelmointi/materiaali/IX_abstra/#5l
- Rajapintaluokka kuvataan luokkana, johon liitetään stereotyyppi <<interface>>
- Rajapinnan toteuttaminen merkitään kuten periminen, mutta katkoviivana
  - Voidaan tarkentaa stereotyypillä <<implements>>



Toinen esimerkki rajapintaluokan käytöstä

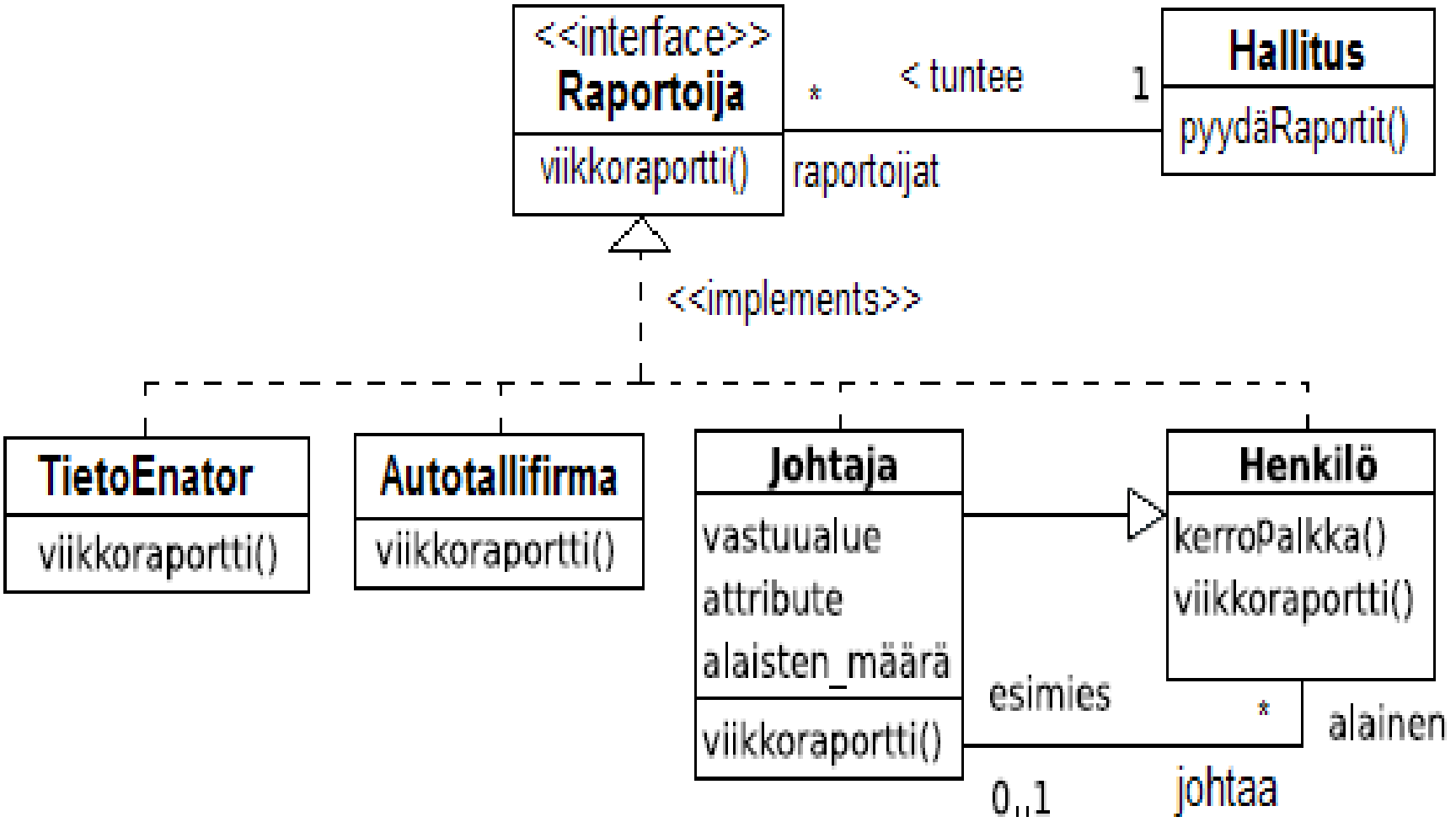
- Palataan muutaman sivun takaiseen yritysesimerkkiin
- Tilanne on nyt se, että yritys on ulkoistanut osan toiminnoistaan
- Hallitus on edelleen kiinnostunut viikkoraporteista
  - Hallitusta ei kuitenkaan kiinnosta se, tuleeko viikkoraportti omalta henkilöstöltä vai alihankkijalta
- Muuttuneessa tilanteessa hallitus tunteekin ainoastaan joukon raportointiin kykeneviä olioita
  - Jotka voivat olla Henkilöitä, Johtajia tai alihankkijoita
  - Kukin näistä toteuttaa metodin viikkoraportti() omalla tavallaan
- Tilanne kannattaa hoitaa määrittelemällä *rajapintaluokka* ja vaatia, että kaikki hallituksen tuntemat tahot toteuttavat rajapinnan

```
public interface Raportoija {

   public void viikkoraportti()

}
```

- Hallitukselle riittääkin, että se tuntee joukon Raportoijia (eli rajapinnan toteuttajia)

- Hallituksen koodi voisi sisältää seuraavan:

```
class Hallitus {
    ArrayList<Raportoija> raportoijat;
    void pyydaRaportit(){      // Javan for-each-lause, eli r käy läpi koko listan
       for ( r : raportoijat ) {   r.viikkoraportti()  }
    }
```



Perimisen kolmet kasvot

- Wiklan materiaali lainaa Koskimiehen luonnehdintaa, jonka mukaan periytymisellä on ”kahdet kasvot”:
- Uusiokäyttömekanismi
  - Periytymisellä voidaan toteuttaa aiemmin ohjelmoitujen välineiden uudelleenkäyttöä, välineitä täydennetään ja erityistetään uusien tarpeiden mukaisiksi (vrt Varasto – Tuotevarasto – MuistavaTuotevarasto)
- Käsitteiden luokittelu
  - Ongelmamaailmaa mallinnetaan peritymisen käsittein, yliluokka-aliluokka -suhde vastaa yleinen-erityinen -suhdetta. Eläin on kissan yleistys, lehmä on naudan erikoistapaus, jne.
- Kolmannet kasvot, *ne ylivoimaisesti tärkeimmät,* eivät käy kunnolla esille Koskimiehen esityksestä:
  - Näillä kasvoilla ei ole niin selkeää nimitystä kuin edellisillä kahdella
  - Seuraavalla kalvolla hiukan hahmotellaan mitä asialla tarkoitetaan
  - Asiaan liittyy *polymorfismi*
  - Juuri nämä kolmannet kasvot ovat se seikka, mikä on merkittävä muokattavuuden ja uusiokäytön mahdollistaja olio-ohjelmoinnissa

- Muutamassa esimerkissä näimme, miten on mahdollista käsitellä olioita tuntematta niiden varsinaista luokkaa
    - tunnetaan ainoastaan rajapinta tai yliluokka, eli asiat jotka oliot ainakin osaavat tehdä
- Esim. Hallituksen ei tarvitse tietää raportoijista juuri mitään
    - Koska raportoijat toteuttavat Raportoija-rajapinnan, on selvää, että niille voidaan kutsua metodia viikkoraportti()
    - Kukin olio tietää oman tyyppinsä ja osaa suorittaa omalla tavalla toteuttamansa viikkoraportti()-metodin (polymorfismin ansiosta)
    - Siispä kaikki raportoivat oikein vaikka Hallitus ei tiedä raportoijista mitään
    - Tämä mahdollistaa, että tulevaisuudessa voidaan ohjelmoida uusia luokkia, jotka saadaan liitettyä Hallitukseen kunhan ne vaan toteuttavat rajapinnan Raportoija
- Eli rajapintojen tai yliluokkien avulla saadaan **rajattua monimutkaisuutta** pois sieltä, missä siitä ei tarvitse välittää
    - Muutokset eivät haittaa niin kauan kun rajapinta toteutetaan asiallisesti
- Rajapinta/yliluokka tarjoaa **laajennuspaikan tulevaisuuden** varalle
    - Vanha ohjelma ei mene rikki uusista aliluokista tai rajapinnan toteuttajista jos ne toimivat oletusten mukaan
- *Nämä ovat syvällisiä asioita. Nyt ne tuntuvat ehkä osin käsittämättömiltä*

Kolmas oliosuunnittelun periaate

- Vähän aikaa sitten mainittiin kaksi tärkeää oliosuunnittelun periaatetta
  - Single responsibility
  - Favour composition over inheritance
- Kolmas ”kulmakivi on seuraava”:
  - **Program to an interface, not to an Implementation**, tai toisin ilmaistuna
  - **Depend on Abstractions, not on concrete implementations**
- Kyse on juuri edellisellä sivulla hahmotellusta asiasta:
  - Laajennettavuuden kannalta ei ole hyvä idea olla riippuvainen konkreettisista luokista sillä ne saattavat muuttua
  - Parempi on tuntea vain rajapintoja (tai abstrakteja luokkia) ja olla tietämätön siitä mitä rajapinnan takana on
- Tämä voidaan ajatella ”laajennettuna kapselointina”
  - Kapselointi piilottaa olioiden sisäisen toteutuksen esim. määrittelemällä instanssimuuttujat näkyvyydeltään privateiksi
  - Jos tunnetaan vaan rajapinta, ”kapseloituu” koko takana oleva olio ja tämä taas avaa uudenlaisen joustavuuden sillä rajapinnan toteuttava luokka on helppo muuttaa vaikuttamatta sen käyttäjiin