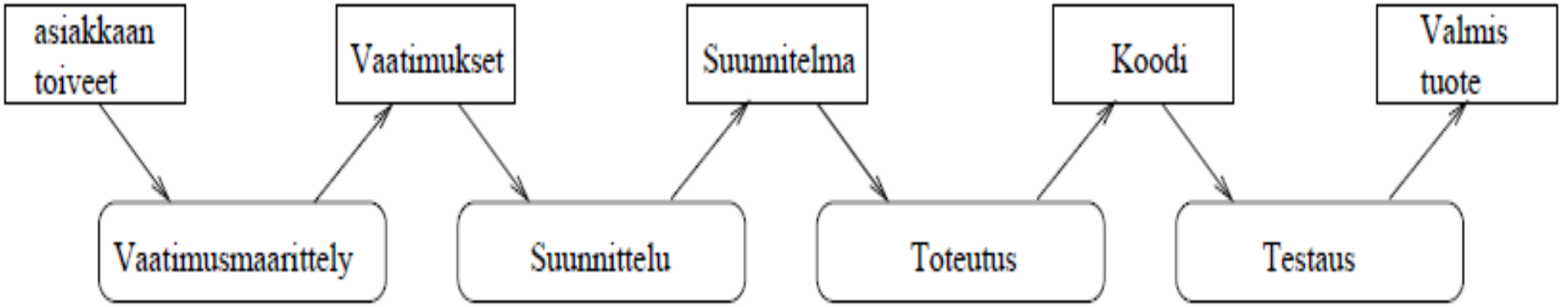
Ohjelmistotekniikan menetelmät

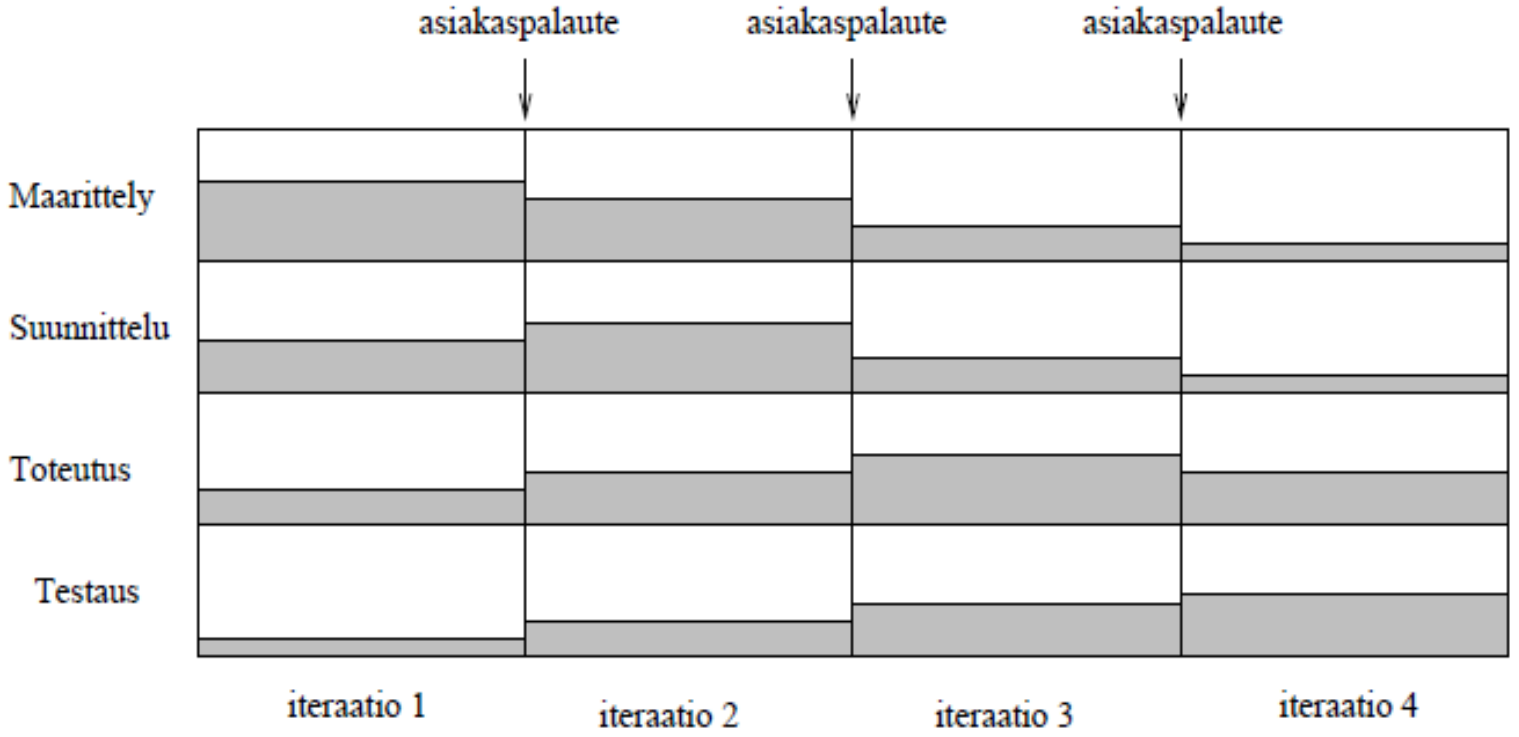
Luento 2, 3.11.

Kertausta: **Ohjelmistotuotantoprosessin vaiheet**

- Vaatimusanalyysi- ja määrittely
  - Mitä halutaan?
- Suunnittelu
  - Miten tehdään?
- Toteutus
  - Ohjelmointi
- Testaus
  - Varmistetaan että toimii niin kuin halutaan
- Ylläpito
  - Korjataan bugit ja tehdään laajennuksia

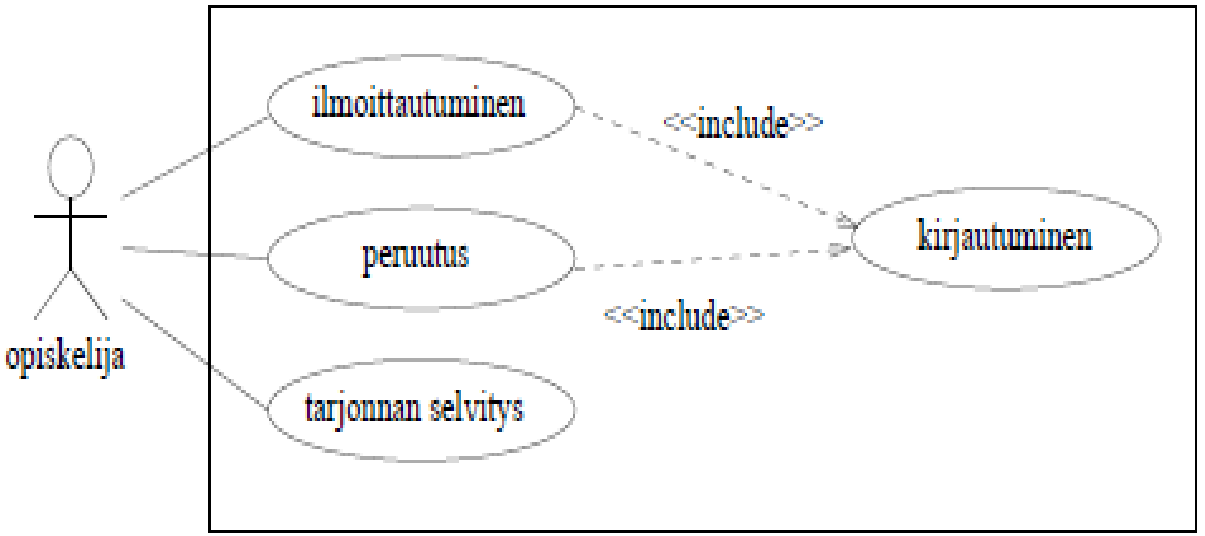
Vesiputus vs ketterä





Viimeksi: **käyttötapaukset**

- Tapa dokumentoida järjestelmän toiminnalliset vaatimukset
  - Käytetään siis *vaatimusmäärittelyssä*
- Käyttötapaus dokumentoi miten *käyttäjä* kommunikoi järjestelmän kanssa suorittaessaan jotain tehtävää
  - Yksi käyttötapaus kuvaa isomman tavoitteellisen kokonaisuuden
  - *Ei kuvaa järjestelmän sisäistä rakennetta tai toimintaa*
- Yleiskuva järjestelmän toiminnallisuudesta *käyttötapauskaaviona*
  - Ellipsit käyttötapauksia, tikkuihmiset käyttäjiä
  - Kuvassa *kirjautuminen* on apukäyttötapaus jonka *ilmoittautuminen* ja *peruutus* sisällyttävät



Käyttötapaus kuvataan tekstinä

- Ohjelmistoprojektissa pitää kuvata kaikki käyttötapaukset tekstuaalisesti saman kaavan mukaan:
  - *käyttötapauspohja*
- Esim. käyttötapaus **Ilmoittautuminen kurssille**:

- *Käyttäjä:* opiskelija
- *Tavoite:* saada kurssipaikka
- *Esiehto:* opiskelija on ilmoittautunut kuluvalla lukukaudella läsnäolevaksi
- *Jälkiehto:* opiskelija on lisätty haluamansa ryhmän ilmoittautujien listalle
- *Käyttötapauksen kulku:*

1. Opiskelija aloittaa kurssi-ilmoittautumistoiminnon
2. Järjestelmä näyttää kurssitarjonnan
3. Opiskelija tutkii kurssitarjontaa
4. Opiskeljia valitsee ohjelmiston esittämästä tarjonnasta kurssin ja ryhmän
5. **Suoritetaan käyttötapaus kirjautuminen**
6. Järjestelmä ilmoittaa opiskelijalle ilmoittautumisen onnistumisesta.

Testaus

- Testaus jakautuu eri asioihin kohdistuviin aktiviteetteihin:
  - **Yksikkötestaus**
    - Toimivatko yksittäiset metodit ja luokat kuten halutaan?
  - **Integraatiotestaus**
    - Varmistetaan komponenttien yhteentoimivuus
  - **Järjestelmä/hyväksymistestaus**
    - Toimiiko kokonaisuus niin kuin vaatimusdokumentissa sanotaan?
- **Regressiotestauksella** tarkoitetaan järjestelmälle muutosten ja bugikorjausten jälkeen ajettavia testejä jotka varmistavat että muutokset eivät riko mitään ehjää
  - Regressiotestit koostuvat yleensä yksikkö-, integraatio- ja järjestelmä/hyväksymätesteistä
- Nykyinen trendi on tehdä testeistä automaattisesti suoritettavia
- Laskareissa harjoiteltiin automaattisten yksikkötestien kirjoittamista JUnitilla

```
public class LyyraKorttiTest {
    LyyraKortti kortti;
    @Before
    public void setUp() {
        kortti = new LyyraKortti(10);
    }

    @Test
    public void konstruktoriAsettaaSaldonOikein() {
        assertEquals("Kortilla on rahaa 10.0 euroa", kortti.toString());
    }

    @Test
    public void syoEdullisestiVahentaaSaldoaOikein() {
        kortti.syoEdullisesti();
        assertEquals("Kortilla on rahaa 7.5 euroa", kortti.toString());
    }
```

Yksikkötestit

- Yksittäistä ”asiaa” (esim. toimiiko metodi oikein tietyntyylisellä syötteellä) testaavat *testitapaukset* (test case) omina metodeinaan
- Yhtä kokonaisuutta (esim. luokkaa) testaavat testitapaukset sijoitetaan yhden testiluokan sisälle
- Ennen *jokaista* testitapausta suoritettava alustuskoodi kannattaa eriyttää @Before:lla merkittyyn alustusmetodiin *setUp*

- Hyvät testit ovat kattavat eli testaavat ”kaiken” koodin monipuolisesti
- Normaalien syötteiden lisäksi on testeissä erityisen tärkeää tutkia testattavien metodien toimintaa virheellisillä syötteillä ja erilaisilla *raja-arvoilla*
- Esim. luokka Lyyrakortti
  - virheellinen syöte voisi olla esim. yritys ladata kortille negatiivinen määrä rahaa
  - Raja-arvo taas olisi edullisen lounaan ostaminen kun kortilla oleva rahamäärä on tasan edullisen lounaan hinta

Oliot ja luokat

- Kuten viikko sitten mainittiin, nykyään ohjelmistokehitys perustuu usein seuraavaan oletukseen
  - *Minkä tahansa järjestelmän katsotaan voivan muodostua olioista, jotka yhteistyössä toimien ja toistensa palveluja hyödyntäen tuottavat järjestelmän tarjoamat palvelut*

- *Mikä siis on olio?*
  - jokin ohjelman tai sen sovellusalueen kannalta mielenkiintoinen asia tai käsite, tai ohjelman osa
  - yleensä yhdistää tietoa ja toiminnallisuutta
  - omaa identiteetin, eli erottuu muista olioista omaksi yksilökseen

Oliot ja luokat

- Jokainen olio kuuluu johonkin *luokkaan*
- *Luokka kuvaa minkälaisia siihen kuuluvat oliot ovat tietosisällön ja toiminnallisuuden suhteen*
- Luokka Javassa:

```
public class Henkilo {
    private String nimi;
    private int ika;
    public Henkilo(String n) {  nimi = n;  }
    public void vanhene() {  ika++;  }
}
```

- ja Henkilö-olioita:

```
Henkilo arto    = new Henkilo(”Arto”);
Henkilo heikki = new Henkilo(”Heikki”);
arto.vanhene();
heikki.vanhene();
```

Suunnittelu ja toteutusvaiheen oliot ja luokat

- Oliota ja luokkia ajatellaan usein ohjelmointitason käsitteinä
  - Esim. Javassa määritellään luokka class Henkilo { … }
  - Ja luodaan olioita Henkilo arto = new Henkilo(”Arto”);
  - Ohjelma siis muodostuu olioista
  - Oliot elävät koneen muistissa
  - Ohjelman toiminnallisuus muodostuu olioiden toiminnallisuudesta

- Ohjelmiston suunnitteluvaiheessa suunnitellaan mistä oliosta ohjelma koostuu ja miten oliot kommunikoivat
  - Nämä oliot sitten toteutetaan ohjelmointikielellä toteutusvaiheessa

- *Mistä suunnitteluvaiheen oliot tulevat? Miten ne keksitään?*

Vaatimusanalyysivaiheen oliot ja luokat

- Vaatimusmäärittelyn yhteydessä tehdään usein *vaatimusanalyysi*
  - Kartoitetaan ohjelmiston sovellusalueen (eli sovelluksen kohdealueen) kannalta tärkeitä *käsitteitä* ja *niiden suhteita*
- Eli mietitään mitä tärkeitä asioita sovellusalueella on olemassa
  - Esim. kurssihallintojärjestelmän käsitteitä ovat
    - Kurssi
    - Laskariryhmä
    - Ilmoittautuminen
    - Opettaja
    - Opiskelija
    - Sali
    - Salivaraus
- Nämä käsitteet voidaan ajatella luokkina

Vaatimusanalyysivaiheen oliot ja luokat

- Vaatimusanalyysivaiheen luokat ovat *vastineita reaalimaailman käsitteille*
- Kun edetään vaatimuksista ohjelmiston suunnitteluun, monet vaatimusanalyysivaiheen luokista saavat vastineensa ”ohjelmointitason” luokkina, eli luokkina, jotka on tarkoitus ohjelmoida esim. Javalla
- Eli riippuen katsantokulmasta, luokka voi olla joko
  - reaalimaailman käsitteen vastine, tai
  - suunnittelu- ja ohjelmointitason ”tekninen” asia
- Tyypillisesti ohjelmatason olio on vastine jollekin todellisuudessa olevalle ”oliolle”
  - Ohjelma simuloi todellisuutta
- Ohjelmissa on myös paljon luokkia ja olioita, joille ei ole vastinetta todellisuudessa
  - Esim. käyttöliittymän toteuttavat oliot

Oliomallinnus ja Olioperustainen ohjelmistokehitys

- **Olioperustainen ohjelmistokehitys** etenee yleensä seuraavasti:
  1. Luodaan **määrittelyvaiheen oliomalli** sovelluksen käsitteistöstä
     - Mallin oliot ja luokat ovat rakennettavan sovelluksen kohdealueen käsitteiden vastineita
  2. Suunnitteluvaiheessa tarkennetaan edellisen vaiheen oliomalli **suunnitteluvaiheen oliomalliksi**
     - Oliot muuttuvat yleiskäsitteistä teknisen tason olioiksi
     - Mukaan tulee olioita, joilla ei suoraa vastinetta reaalimaailman käsitteistössä
     - Osa olioista on luonteeltaan *pysyviä* ja niitä tulee vastaamaan jokin rakenne ohjelman tietokannassa
  3. Toteutetaan suunnitteluvaiheen oliomalli jollakin **olio-ohjelmointikielellä**
- Voidaankin ajatella, että *malli tarkentuu* muuttuen koko ajan ohjelmointikieliläheisemmäksi/teknisemmäksi siirryttäessä määrittelystä suunnitteluun ja toteutukseen

Luokka- ja oliokaaviot

- Palaamme oliomallinnukseen ja olioiden käyttöön määrittely- ja suunnittelutasolla myöhemmin
- Nyt tarkastelemme miten olioita ja luokkia kuvataan UML:ssä
- Sama mallinnustekniikka kelpaa sekä teknisen tason että korkeamman tason olioille ja luokille

Luokka- ja oliokaaviot

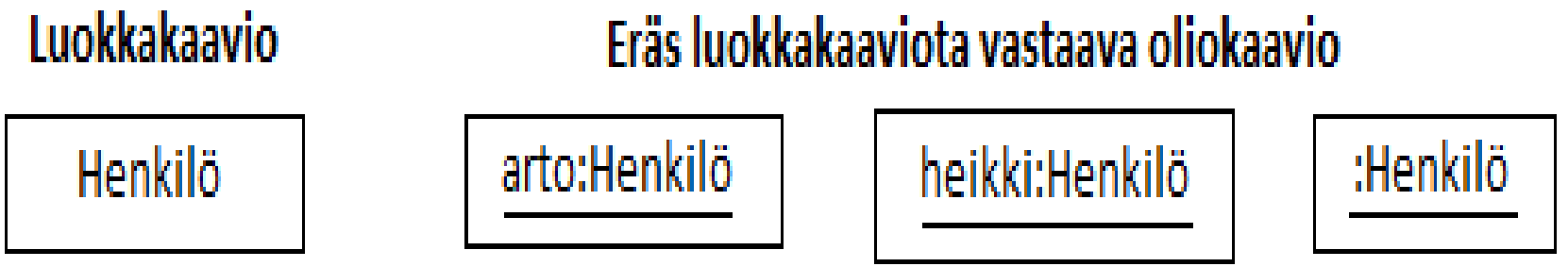
- Järjestelmän luokkarakennetta kuvaa **luokkakaavio** (engl. class diagram)
  - Mitä luokkia olemassa
  - Minkälaisia luokat ovat
  - Luokkien ja niiden olioiden suhteet toisiinsa
- Luokkakaavio on UML:n eniten käytetty kaaviotyyppi

- Luokkakaavio kuvaa ikäänkuin kaikkia mahdollisia olioita, joita järjestelmässä on mahdollista olla olemassa

- **Oliokaavio** (engl. object diagram) taas kuvaa mitä olioita järjestelmässä on tietyllä hetkellä

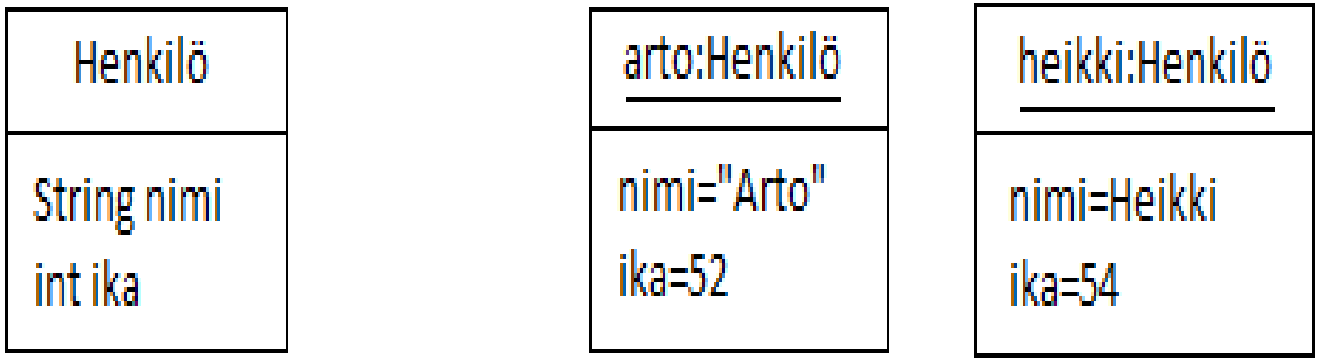
Luokka- ja oliokaaviot

- Luokkaa kuvataan laatikolla, jonka sisällä on luokan nimi
  - Kuten pian näemme, laatikossa voi olla myös muuta
- Luokasta luotuja olioita kuvataan myös laatikolla, erona on nimen merkintätapa
  - Nimi alleviivattuna, sisältäen mahdollisesti myös olion nimen
- Kuvassa Henkilö-luokka ja kolme Henkilö-olioa
  - Kolmas olioista on nimetön
- Luokkia ja olioita ei sotketa samaan kuvaan, kyseessä onkin kaksi kuvaa: vasemmalla luokkakaavio ja oikealla oliokaavio
  - Oliokaavio kuvaa tietyn hetken tilanteen, olemassa 3 henkilöä



Luokka- ja oliokaaviot: attribuutit

- Luokan olioilla on attribuutteja eli oliomuuttujia ja operaatiota eli metodeja
- Nämä määritellään luokkamäärittelyn yhteydessä
  - Aivan kuten Javassa kirjoitettaessa class Henkilö{ … } määritellään kaikkien Henkilö:n attribuutit ja metodit luokan määrittelyn yhteydessä
- Luokkakaaviossa attribuutit määritellään luokan nimen alla omassa osassaan laatikkoa
  - Attribuutista on ilmaistu nimi ja tyyppi (voi myös puuttua)
- Oliokaaviossa voidaan ilmaista myös attribuutin arvo

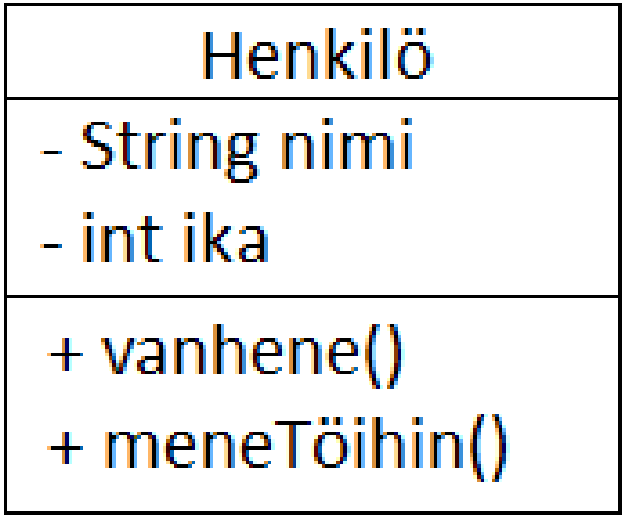


Luokka- ja oliokaaviot: metodit

- Luokan olioiden metodit merkitään laatikon kolmanteen osaan
- Luokkiin on pakko merkitä ainoastaan nimi
  - Attribuutit ja metodit merkitään jos tarvetta
  - Usein metodeista merkitään ainoastaan nimi, joskus myös parametrien ja paluuarvon tyyppi
- Attribuuttien ja operaatioiden parametrien ja paluuarvon tyyppeinä voidaan käyttää valmiita tietotyyppejä (int, double, String, …) tai rakenteisia tietotyyppejä (esim. taulukko, ArrayList).
- Tyyppi voi olla myös luokka, joko itse määritelty tai asiayhteydestä ”itsestäänselvä” (alla Väri ja Piste)

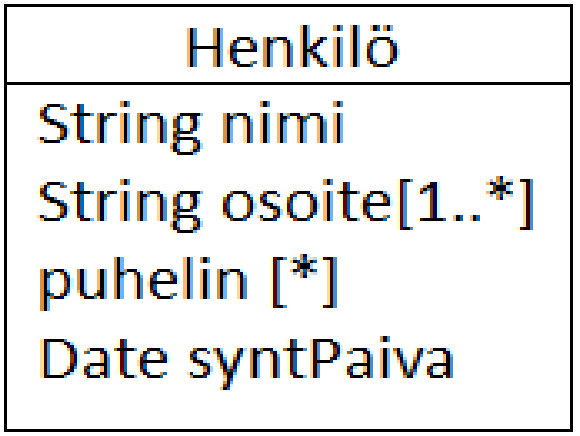
Luokkakaavio: attribuuttien ja operaatioiden näkyvyys

- Ohjelmointikielissä voidaan attribuuttien ja metodien näkyvyyttä muiden luokkien olioille säädellä
  - Javassa private, public, protected
- UML:ssa näkyvyys merkitään attribuutin tai metodin eteen: *public +, private -, protected #, package ~*
  - Jos näkyvyyttä ei ole merkitty, sitä ei ole määritelty
  - Kovin usein näkyvyyttä ei viitsitä merkitä
- Esim. alla kaikki attribuutit ovat private eli eivät näy muiden luokkien olioille, metodit taas public eli kaikille julkisia



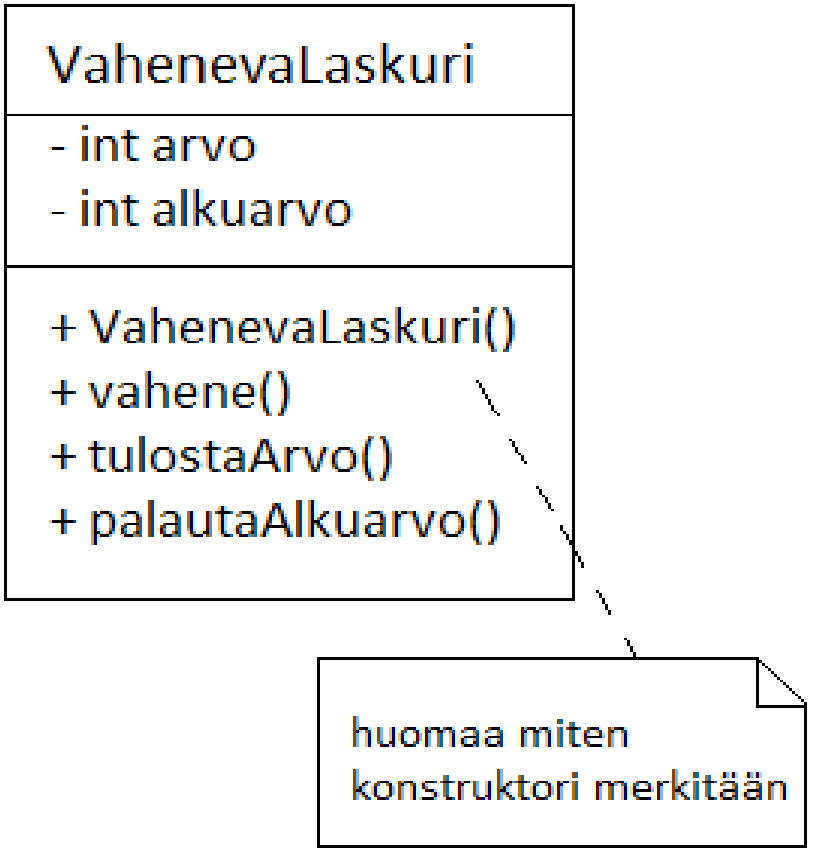
Luokkakaavio: attribuutin moniarvoisuus

- Jos attribuutti on kokoelma samanlaisia arvoja (esim. taulukko), voidaan se merkitä luokkakaavioon
  - Kirjoitetaan attribuutin perään hakasulkeissa kuinka monesta ”asiasta” attribuutti toistuu
  - * tarkoittaa tuntematonta
- Henkilöllä on vähintään 1 osoite, mutta osoitteita voi olla useita
- Puhelinnumeron kohdalle on kirjoitettu [*], joka tarkoittaa, että numeroa ei välttämättä ole tai numeroita voi olla useita
- Moniarvoisuus on asia, joka merkitään kaavioon melko harvoin



```
public class VahenevaLaskuri {
    private int arvo;
    private int alkuarvo;
    public VahenevaLaskuri(int arvo) {
        this.arvo = arvo;
        this.alkuarvo = arvo;
    }
    public void vahene() {
       if ( this.arvo>0 ) {  this.arvo--; }
    }
    public void tulostaArvo() {
        System.out.prntln( this. arvo);
    }
    public void palautaAlkuarvo() {
        this.arvo = this.alkuarvo;
    }
}
```

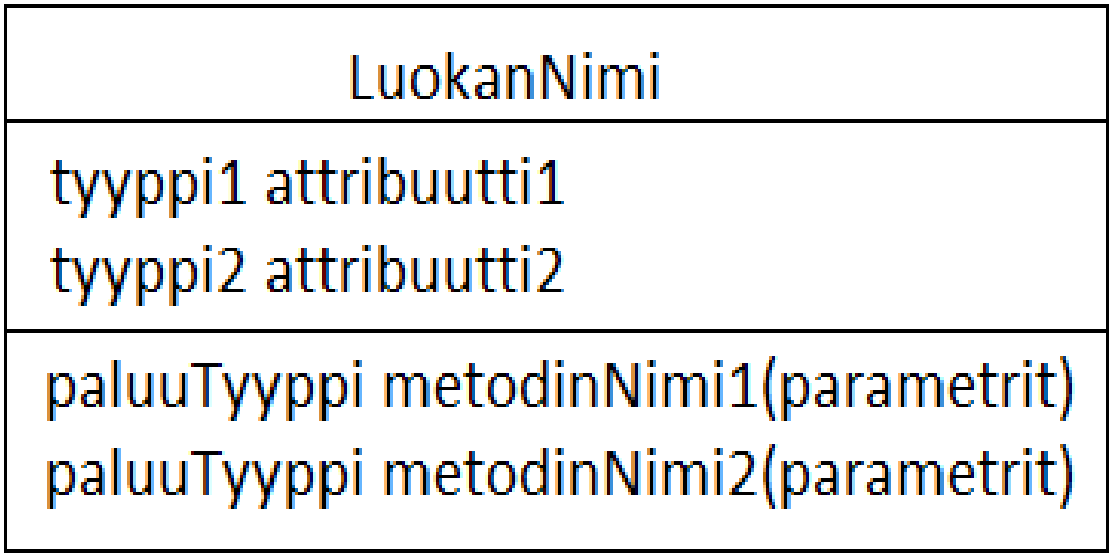
Ohjelmoinnin perusteista tuttu VahenevaLaskuri



Kuvassa mukana UML-kommenttisymboli

Kertaus: Luokan määrittely luokkakaaviossa

- Eli luokka on laatikko, jossa luokan nimi ja tarvittaessa attribuutit sekä metodit
- Attribuuttien ja metodien parametrien ja paluuarvon tyyppi ilmaistaan tarvittaessa
  - Näkyvyysmääreet ilmaistaan tarvittaessa
- Jos esim. metodeja ei haluta näyttää, jätetään metodiosa pois, vastaavasti voidaan menetellä attribuuttien suhteen



Olioiden väliset yhteydet

- Ohjelmat sisältävät useita olioita ja olioiden välillä on yhteyksiä:
  - Työntekijä *työskentelee* Yrityksessä
  - Henkilö *omistaa* Auton
  - Henkilö *ajaa* Autolla
  - Auto *sisältää* Renkaat
  - Henkilö *asuu* Osoitteessa
  - Henkilö *omistaa* Osakkeita
  - Työntekijä *on* Johtajan *alainen*
  - Johtaja *johtaa* Työntekijöitä
  - Johtaja *erottaa* Työntekijän
  - Opiskelija *on ilmoittautunut* Kurssille
  - Kello *sisältää* kolme Viisaria

- Yhteys voi olla pysyvämpiluontoinen eli rakenteinen tai hetkellinen
  - Aluksi fokus pysyvämpiluontoisissa yhteyksissä

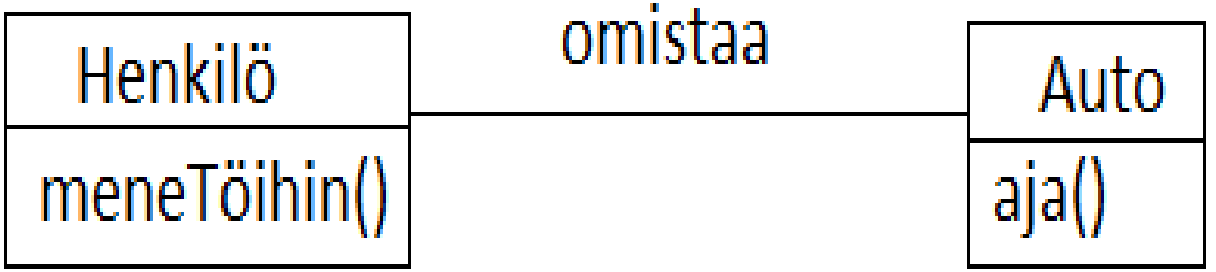
Olioiden väliset yhteydet

- Ohjelmakoodissa pysyvä yhteys ilmenee yleensä luokassa olevana *olioviitteenä*, eli oliomuuttujana jonka tyyppinä on luokka
- Henkilö omistaa Auton:

```
public class Auto{
   public void aja(){ System.out.println("liikkuu"); }
}

public class Henkilö {
   private Auto omaAuto;   // viite olioon, jonka tyyppinä Auto

   public Henkilö(Auto a) {  omaAuto = a;  }

   public void meneTöihin() {
      omaAuto.aja();          // metodissa henkilö käyttää omistamaansa autoa
   }
}
```

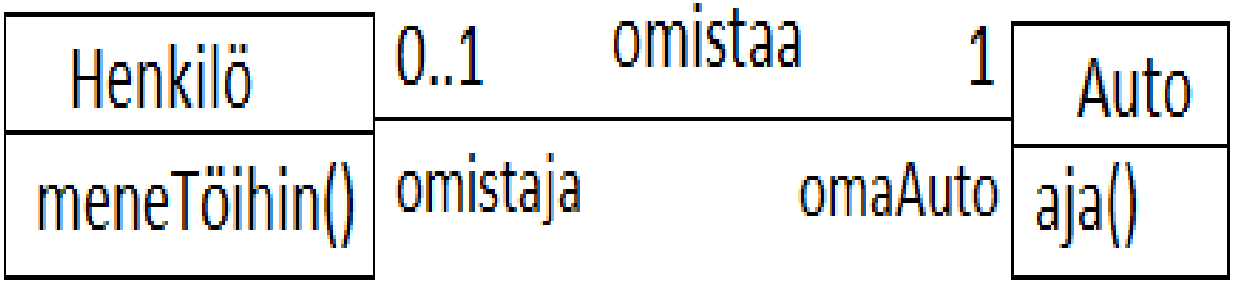
Olioiden väliset yhteydet

- Olioviite voitaisiin periaatteessa merkitä luokkakaavioon attribuuttina, kyseessähän on teknisessä mielessä attribuutti
  - Näin kuitenkaan ei ole tapana tehdä
- **Parempi tapa on kuvata olioiden välinen yhteys luokkakaaviossa**
  - Jos Henkilö- ja Auto-olion välillä voi olla yhteys, yhdistetään Henkilö- ja Auto-luokat viivalla
- Tilanne kuvattu alla
  - Yhteydelle on annettu nimi *omistaa*
  - Eli Henkilö-olio omistaa Auto-olion



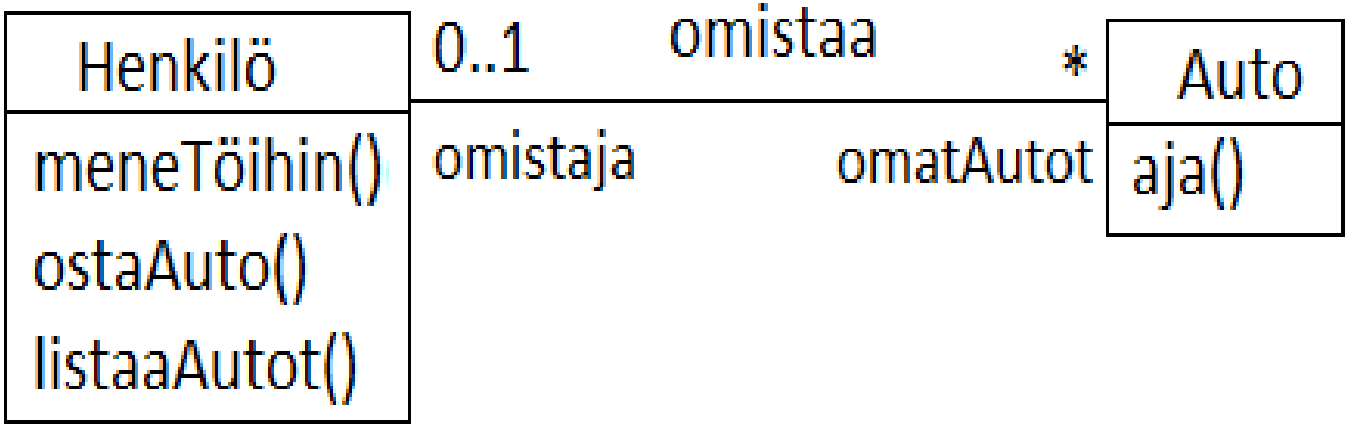
Yhteys: kytkentärajoitteet ja roolit

- Ohjelmakoodissa jokaisella henkilöllä on täsmälleen yksi auto
  - ja auto liittyy korkeintaan yhteen henkilöön
- Tämä kuvataan luokkakaaviossa *kytkentärajoitteina*
  - Alla yhteyden oikeassa päässä on numero 1, joka tarkoittaa, että yhteen Henkilö-olioon liittyy täsmälleen yksi Auto-olio
  - Yhteyden vasemmassa päässä 0..1, joka tarkoittaa, että yhteen Auto-olioon liittyy 0 tai 1 Henkilö-olioa
- Auton *rooli* yhteydessä on olla henkilön omaAuto, rooli on merkitty Auton viereen
  - Huom: roolin nimi on sama kun luokan Henkilö oliomuuttuja, jonka tyyppinä Auto
- Henkilön rooli yhteydessä on olla omistaja



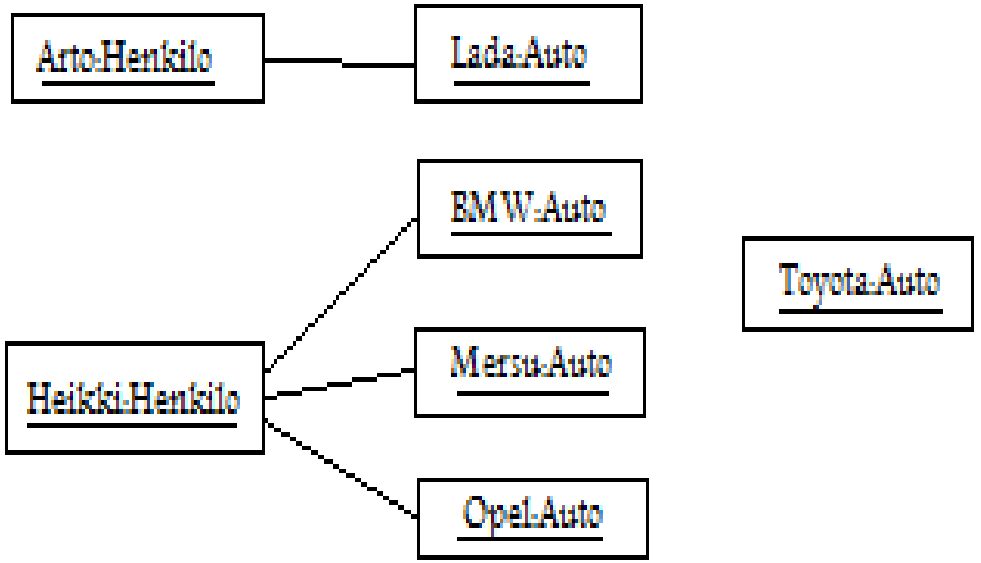
Yhteys

- Edellisessä esimerkissä yhdellä Henkilö-oliolla on yhteys täsmälleen yhteen Auto-olioon
  - Eli yhteyden Auto-päässä on kytkentärajoitteena 1
- Jos halutaan mallintaa tilanne, jossa kullakin Henkilö-oliolla voi olla mielivaltainen määrä autoja (nolla tai useampi), niin kytkentärajoitteeksi merkitään *
  - Alla tilanne, jossa henkilö voi omistaa useita autoja
  - Kullakin autolla joko 0 tai 1 omistajaa



Yhteydet oliokaaviossa

- Luokkakaavio kuvaa luokkien olioiden kaikkia mahdollisia suhteita
  - Edellisessä sivulla sanotaan vaan, että tietyllä henkilöllä voi olla useita autoja ja tietyllä autolla on ehkä omistaja
- Jos halutaan ilmaista asioiden tila jollain ajanhetkellä, käytetään oliokaaviota
  - Mitä olioita tietyllä hetkellä on olemassa
  - Miten oliot yhdistyvät
- Alla tilanne, jossa Artolla on 1 auto ja Heikillä 3 autoa, yhdellä autolla ei ole omistajaa



Yhden suhde moneen -yhteyden toteuttaminen Javassa

- Jos henkilöllä on korkeintaan yksi auto, on Henkilö-luokalla siis attribuutti, jonka tyyppi on Auto

```
public class Henkilö {

    private Auto omaAuto;   // viite olioon, jonka tyyppinä Auto

    // ...

}
```

- Jos henkilöllä on monta autoa, on Javassa yleinen ratkaisu lisätä Henkilö-luokalle attribuutiksi listallinen (esim. ArrayList) autoja:

```
public class Henkilö {

    private ArrayList<Auto> omatAutot;

    // ...

}
```

- Esimerkki seuraavalla sivulla, huomaa miten listalle lisätään uusi auto ja miten lista käydään läpi
- ArrayLististä tarkemmin Ohjelmoinnin perusteiden materiaalissa

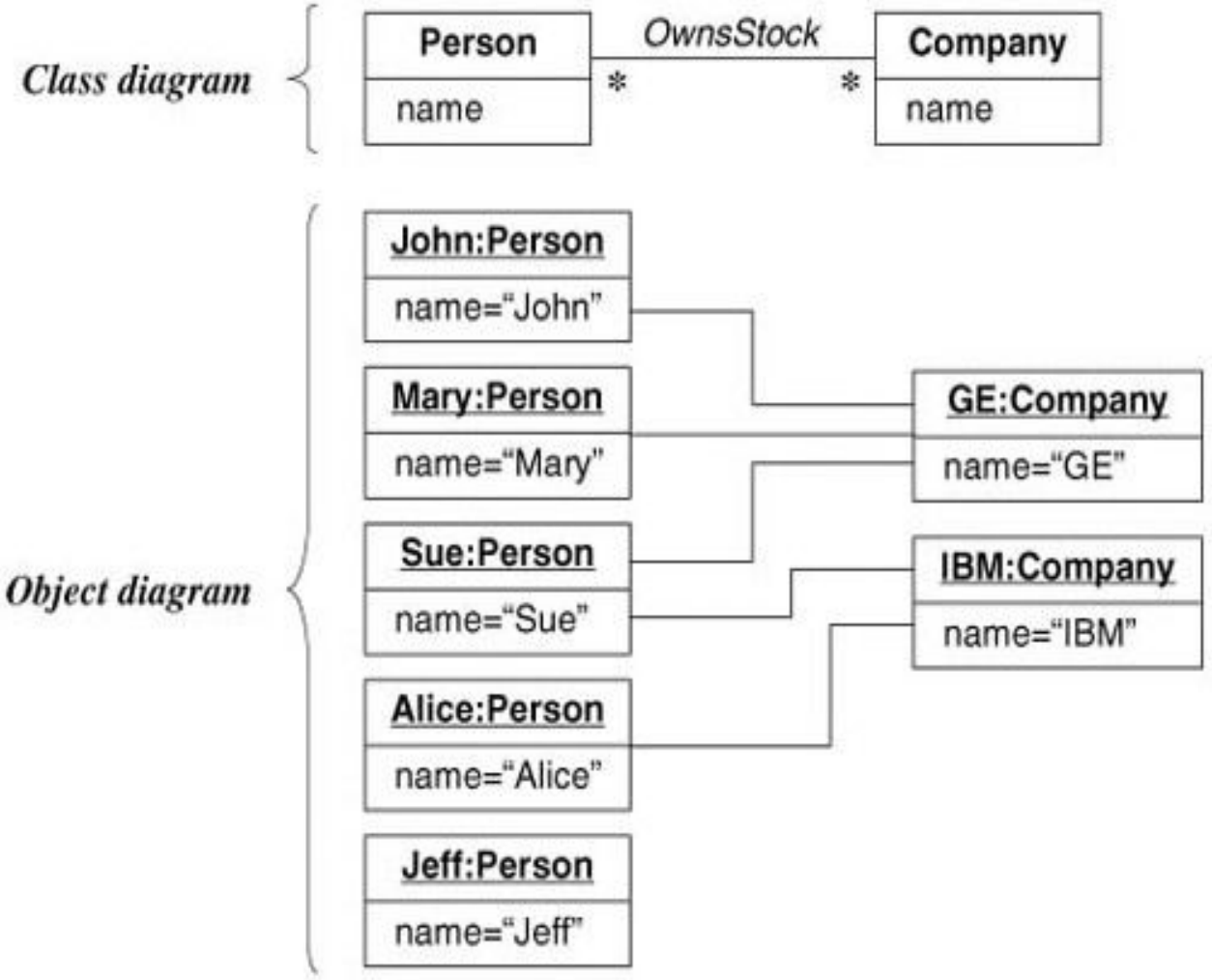
```
public class Henkilö {
    private ArrayList<Auto> omatAutot;

    public Henkilö () { omatAutot = new ArrayList<Auto>(); }
    public void ostaAuto(Auto uusi){
        omatAutot.add(uusi);
    }
    public void listaaAutot() {
         for ( Auto auto : omatAutot )  {
             System.out.prinln(auto);
        }
    }
    public void meneToihin(){
        if ( omatAutot.isEmpty()  )  {
            //  käytä julkista liikennettä
        } else {
            // … valitaan auto jolla mennään töihin
        }
    }
```

Yhteyksistä

- Kuten niin moni asia UML:ssä, on myös yhteyden nimen ja roolinimien merkintä vapaaehtoista
  - Joskus asia on niin ilmeinen, että nimeämistä ei tarvita
- Jos kytkentärajoite jätetään merkitsemättä, niin silloin yhteydessä olevien olioiden lukumäärä on määrittelemätön
  - **Kytkentärajoitteet onkin syytä ilmaista aina**

- Seuraavassa joukko esimerkkejä
- Monissa tapauksissa esitetty myös oliokaavio selkiyttämään tilannetta

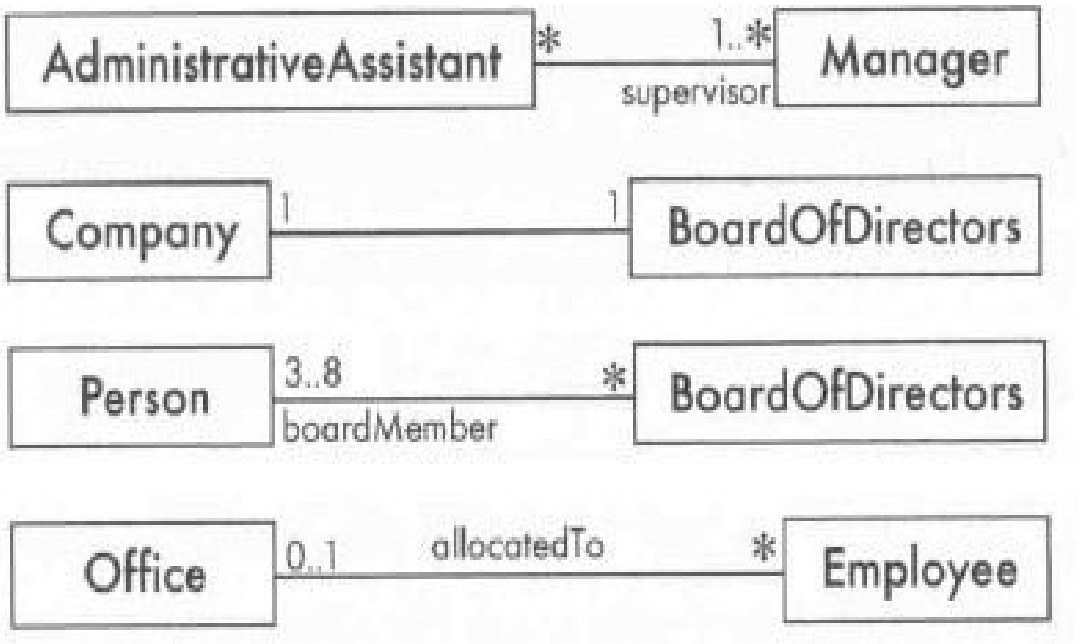
- Henkilö voi omistaa usean yhtiön osakkeita
- Yhtiöllä on monia osakkeenomistajia
- Eli yhteen Henkilö-olioon voi liittyä monta Yhtiö-olioa
- Ja yhteen Yhtiö-olioon voi liittyä monta Henkilö-olioa



- Yhtiössä työskentelee useita ihmisiä
- Ihminen työskentelee korkeintaan yhdessä yrityksessä
- Huomaa roolinimet:
  - Ihmisen rooli yhteyden suhteen on työntekijä, *employee*
  - Yrityksen rooli yhteydessä on työnantaja, *emplyer*

employee	employer
Joe Doe	Simplex
Mary Brown	Simplex
Jean Smith	United Widgets

- Lisää bisnesmaailman esimerkkejä
  - Manageria kohti on useita assistentteja, assistentin johtajana (supervisor) toimii vähintään yksi manageri
  - Yhtiöllä on yksi johtokunta, joka johtaa tasan yhtä yhtiötä
  - Johtokuntaan kuuluu kolmesta kahdeksaan henkeä. Yksi henkilö voi kuulua useisiin johtokuntiin, muttei välttämättä yhteenkään.
  - Toimistoon on sijoitettu (allocated to) useita työntekijöitä. Työntekijällä on paikka yhdessä toimisto tai ei missään



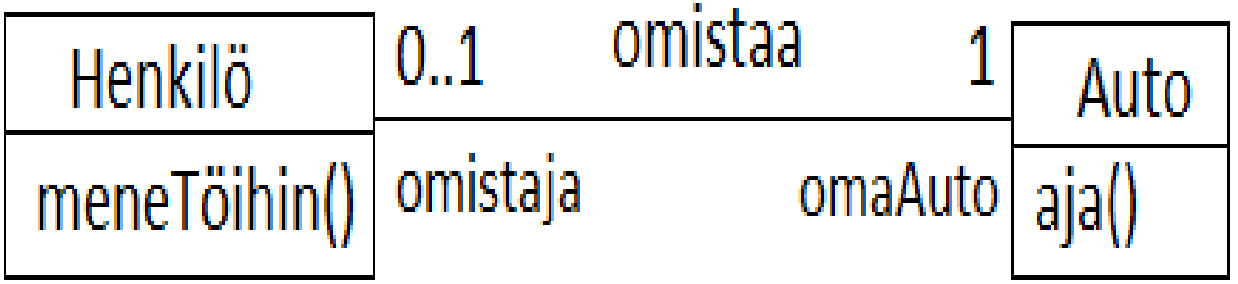
Kaksi OhPe:sta tuttua esimerkkiä

- Henkilo joka sisältää Päivämäärä-olion, luku 24.9
  http://www.cs.helsinki.fi/group/java/s15-materiaali/viikko5/
- Joukkue sisältää Pelaajia, tehtävä 102
  http://www.cs.helsinki.fi/group/java/s15-materiaali/viikko5/#102_joukkueet_ja_pelaajat

- Luokkakaaviot tehdään luennolla

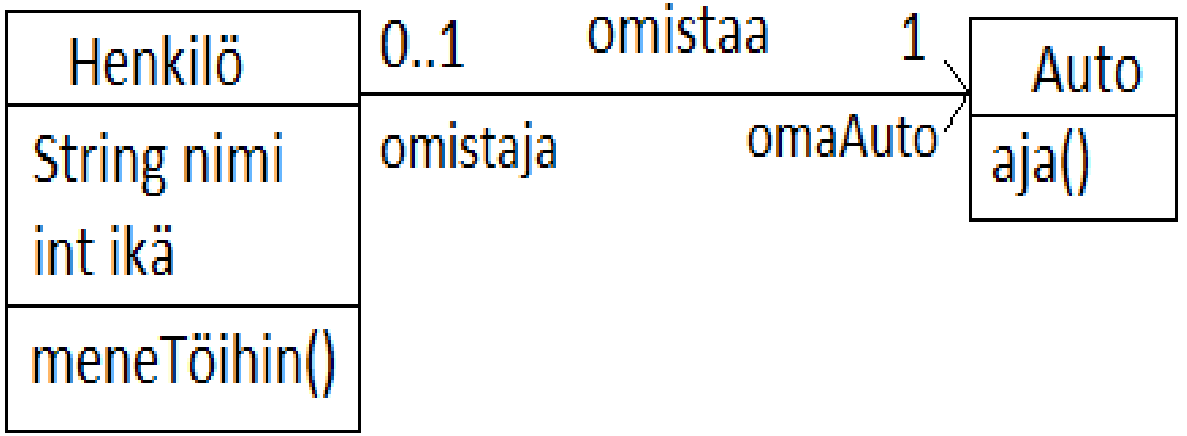
Palataan Auto ja Henkilö -esimerkkiin

```
public class Auto{
    public void aja() { System.out.println("liikkuu"); }
}

public class Henkilö {
    private Auto omaAuto;   // viite olioon, jonka tyyppinä Auto

    public Henkilö(Auto a) {  omaAuto = a;  }

    public void meneTöihin() {
        omaAuto.aja();      // metodissa henkilö käyttää omistamaansa autoa
    }
}
```



Yhteyden navigointisuunta

- Auto-luokan koodista huomaamme, että auto-oliot eivät tunne omistajaansa
  - Henkilö-oliot taas tuntevat omistamansa autot Auto-tyyppisen attribuutin omaAuto ansiosta
- Yhteys siis on oikeastaan *yksisuuntainen,* henkilöstä autoon, mutta ei toisinpäin
- Asia voidaan ilmaista kaaviossa tekemällä yhteysviivasta nuoli
  - Käytetään nimitystä navigointisuunta
  - Nuolen kärki sinne suuntaan, johon on pääsy oliomuuttujan avulla



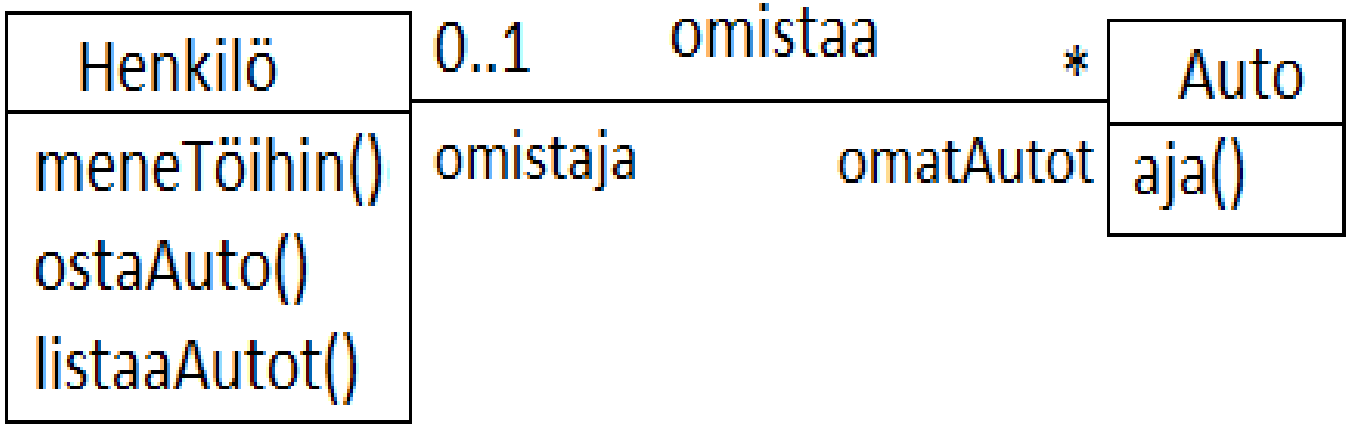
Yhteyden navigointisuunta

- Yhteyden navigointisuunnalla merkitystä lähinnä suunnittelu- ja toteutustason kaavioissa
  - Merkitään vain jos suunta tärkeä tietää
  - Joskus kaksisuuntaisuus merkitään nuolella molempiin suuntiin
  - Joskus taas nuoleton tarkoittaa kaksisuuntaista
- Määrittelytason luokkakaavioissa yhteyden suuntia ei yleensä merkitä ollenkaan

- Yhteyden suunnalla on aika suuri merkitys sille, kuinka yhteys toteutetaan kooditasolla
  - Vapaaehtoinen tehtävä: laajenna edellistä esimerkkiä niin, että Auto-olio tuntee omistajansa, ja että kuolleen henkilön auto voi saada uuden omistajan

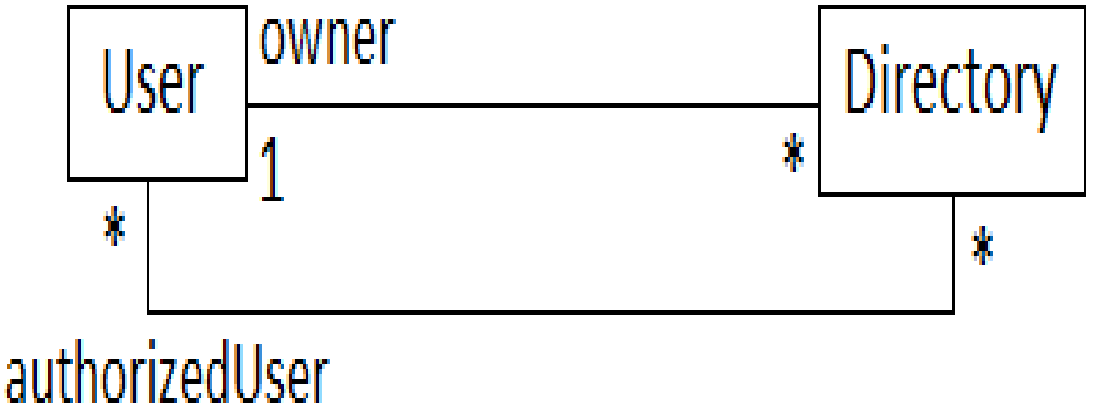
Kertaus monimutkaisemmasta yhteydestä

- Alla olevassa kaaviossa henkilö voi omistaa useita autoja
- Autolla on edelleen korkeintaan yksi omistaja
- Yhteen Henkilö-olioon voi siis liittyä monta Auto-olioa
  - kytkentärajoite * joka tarkoitaa 0...n
- Yhteen Auto-olioon liittyy 0 tai 1 Henkilö-olioa omistajan roolissa
  - kytkentärajoite 0..1
- Jos tilanne epäselvä: *piirrä oliokaavio*

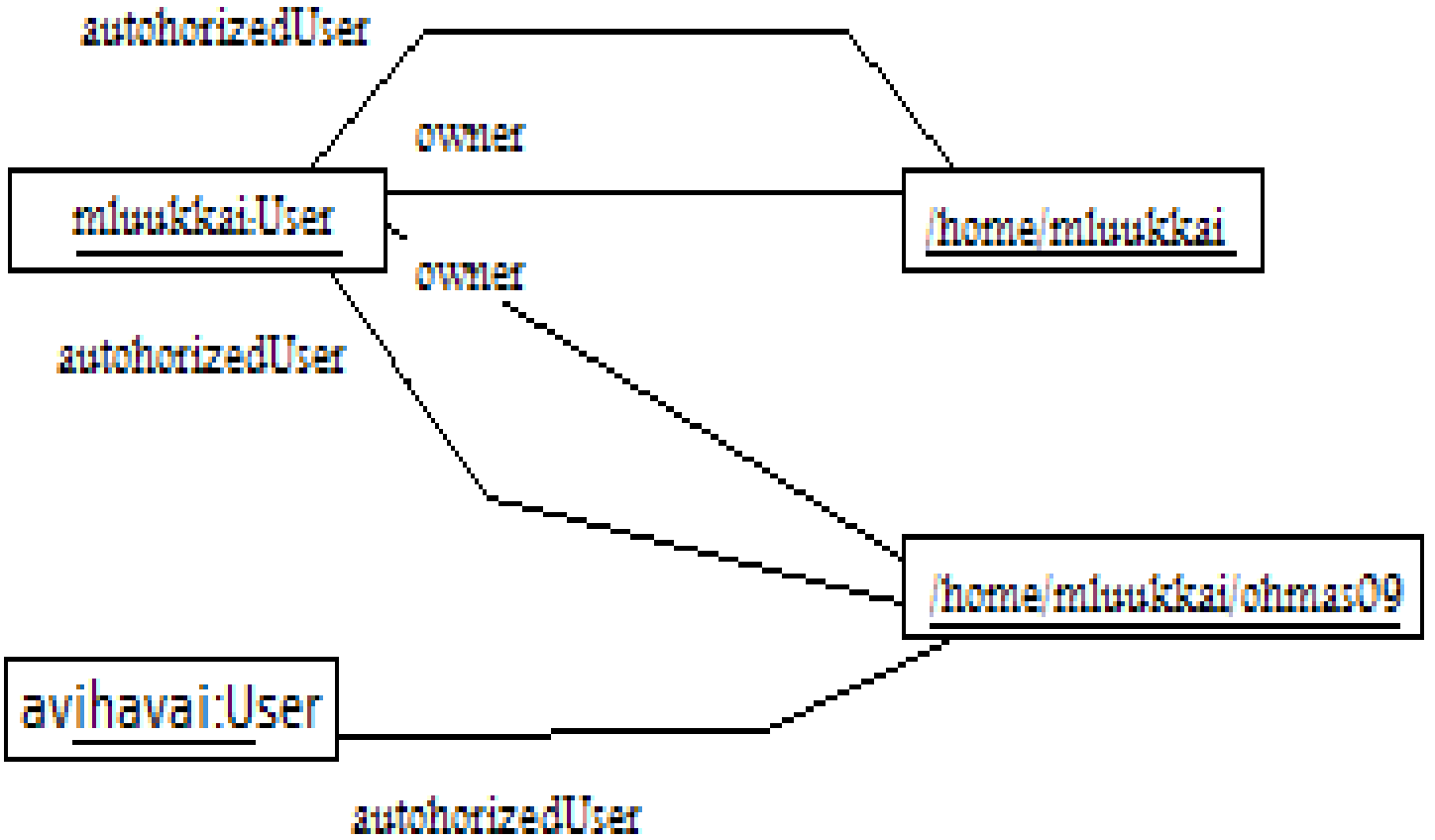


Useampia yhteyksiä olioiden välillä

- Esim. Linuxissa jokaisella hakemistolla on tasan yksi omistaja
  - Eli yhteen hakemisto-olioon liittyy *roolissa owner* tasan yksi käyttäjä-olio
- Jokaisella hakemistolla voi olla lisäksi useita käyttäjiä
  - Yhteen hakemistoon liittyy useita käyttäjiä roolissa authorizedUser
- Yksi käyttäjä voi omistaa useita hakemistoja
- Yhdellä käyttäjällä voi olla käyttöoikeus useisiin hakemistoihin
- Yhdellä käyttäjällä voi olla *samaan hakemistoon sekä omistus- että käyttöoikeus*

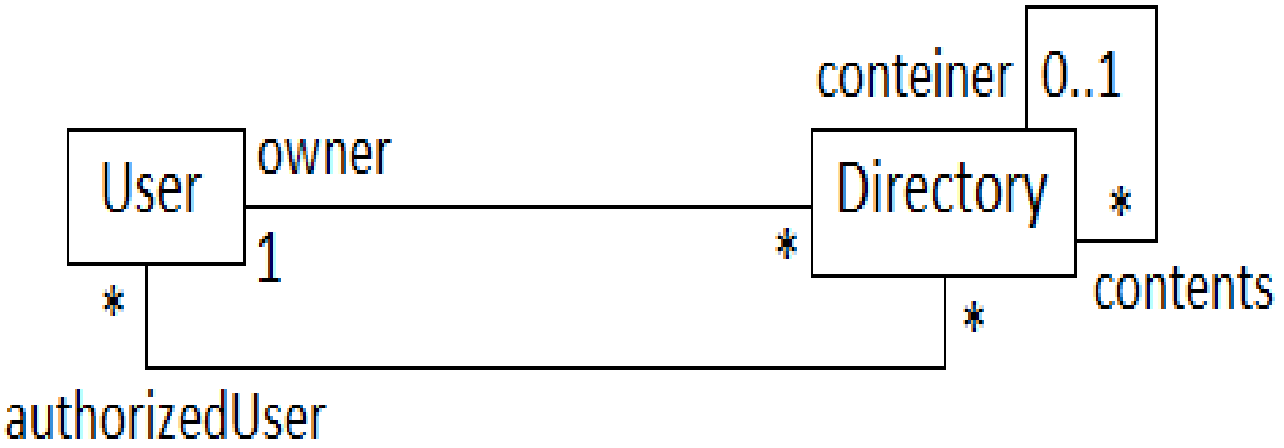


- Alla oliokaavio, joka kuvaa erään edellisen luokkakaavion mukaisen tilanteen
  - Käyttäjät mluukkai ja avihavai
  - mluukkai omistaa kaksi hakemistoa
  - mluukkai:lla myös käyttöoikeus omistamiinsa hakemistoihin
    - Samojen olioiden välillä kaksi eri yhteyttä!
  - avihavai:lla käyttöoikeus hakemistoon /home/mluukkai/ohmas09

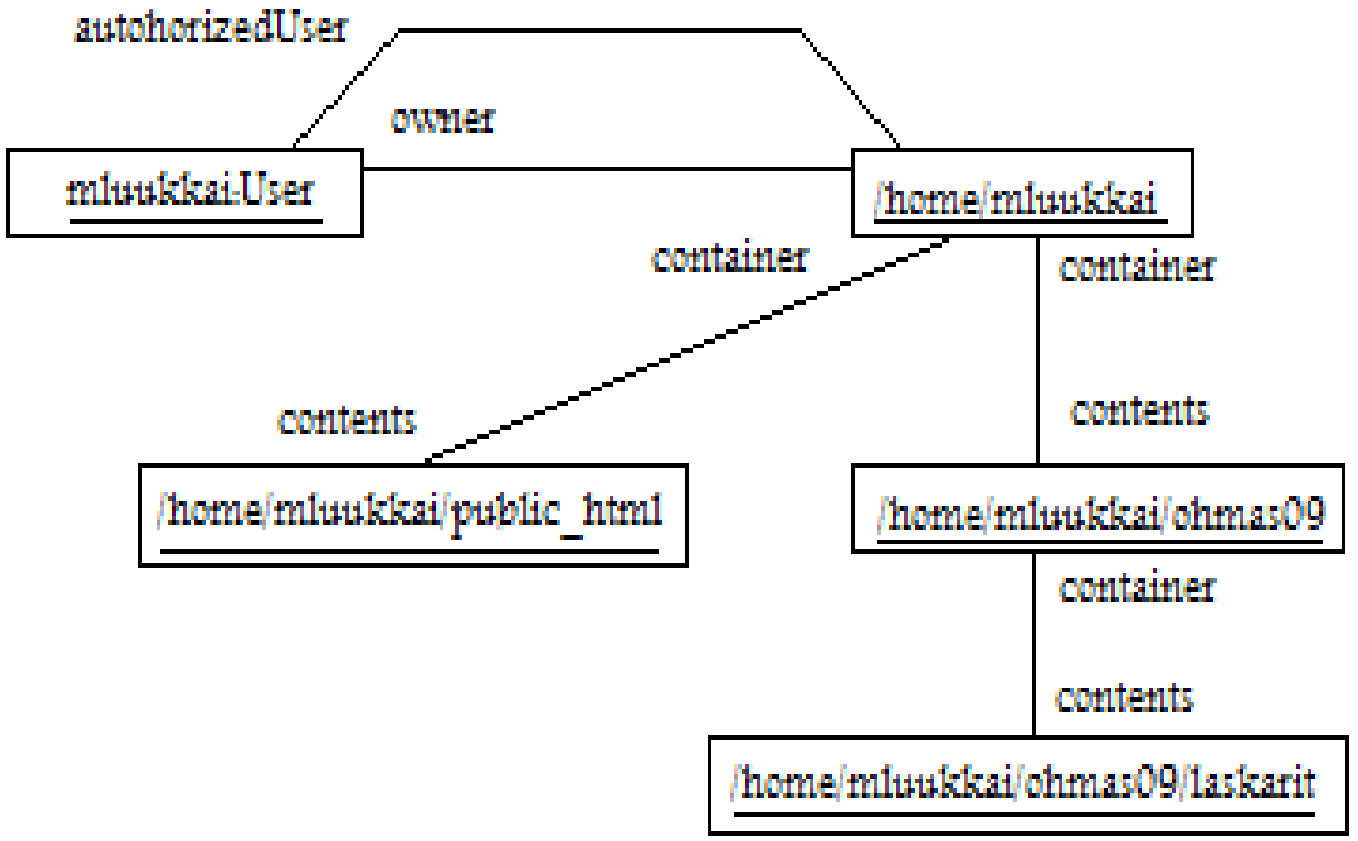


Yhteys kahden saman luokan olion välillä

- Miten mallinnetaan se, että hakemistolla on alihakemistoja?
  - Yhden olion suhteen yhteyksiä on oikeastaan kahdenlaisia
    - Hakemisto sisältää alihakemistoja
    - Hakemisto sisältyy johonkin toiseen hakemistoon
- Yhteen hakemisto-olioon voi liittyä 0 tai 1 hakemisto-olioa roolissa *container* (=sisältäjä), eli hakemisto voi olla jonkun toisen hakemiston alla
- Yhteen hakemisto-olioon voi liittyä mielivaltainen määrä (*) hakemisto-olioita roolissa *contents* (=sisältö), eli hakemisto voi sisältää muita hakemistoja

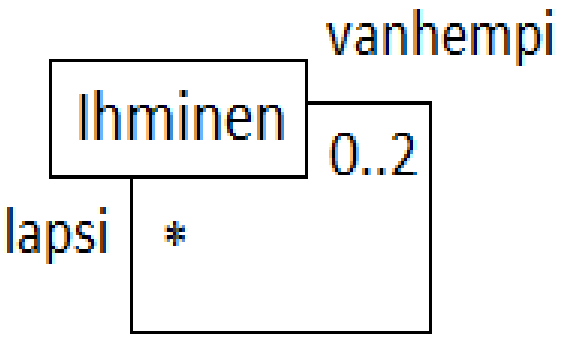


- Tilanne vaikuttaa sekavalta, selvennetään oliokaavion avulla
- /home/mluukkai sisältää kaksi hakemistoa
  - Alihakemistojen rooli yhteydessä on contents eli sisältö
  - Päähakemiston rooli yhteydessä on container eli sisältäjä
- /home/mluukkai/ohmas09 on edellisen alihakemisto, mutta sisältää itse alihakemiston



Yhteys kahden saman luokan olion välillä, toinen esimerkki

- Ihmisten välillä on suhteita, esim. vanhempi ja lapsi
- Ihmisellä voi olla 0-2 vanhempaa
  - Tietyn ihmisolion suhteen nämä ovat *roolissa vanhempi*
- Ihmisellä voi olla lapsia
  - Tietyn ihmisolion suhteen nämä *roolissa lapsi*
- HUOM: malli mahdollistaa sen, että ihminen on itsensä lapsi!
  - Tarvittaisiin esim. kommentti joka kieltää tilanteen
- Luennolla piirretään asiaa selvittävä oliokaavio

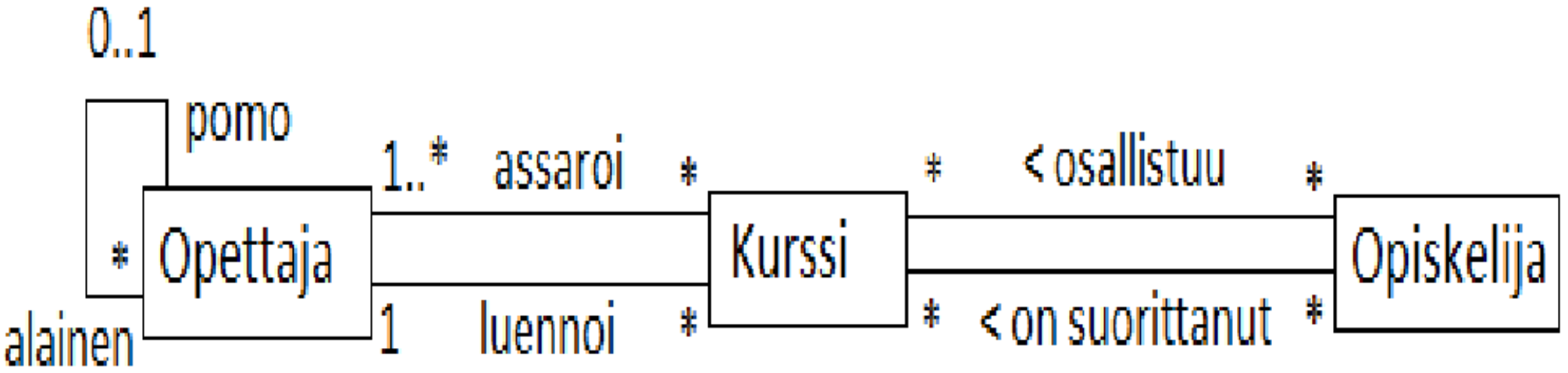


Monimutkaisempi esimerkki

- Mallinnetaan seuraava tilanne
  - Kurssilla on luennoijana 1 opettaja ja assarina useita opettajia
  - Opettaja voi olla useiden kurssien assarina ja luennoijana
  - Opettajalla voi olla pomo ja useita alaisia
    - Opettajat johtavat toisiaan
  - Opiskelija voi osallistua useille kursseille
  - Opiskelijalla voi olla suorituksia useista kursseista

- Seuraavalla sivulla luokkakaavio
  - Luennolla piirretään ehkä oliokaavio
- Yhteyksien *osallistuu* ja *on suorittanut* nimiin on merkitty lukusuunta sillä ne luetaan oikealta vasemmalle:
  - Opiskelija *osallistuu* kurssille

- Monimutkaisia ongelmia kannattaa lähestyä paloissa
  - Mietitään ensin esim. kurssin ja opettajan suhdetta
  - Sitten opiskelijan ja kurssin suhdetta
  - Lopulta opettajien keskinäisiä johtajuussuhteita
  - Kaikista eri vaiheista voidaan piirtää oma kaavionsa joka lopuksi yhdistetään
  - tai samaa kaaviota voidaan laajentaa pikkuhiljaa



Luennolla tehtävä esimerkki

- Mallinnetaan yliopisto luokkakaaviona:
  - Yliopistossa on useita tiedekuntia
  - Tiedekunnissa on useita laitoksia
  - Tiedekunta kuuluu vain yhteen yliopistoon ja laitos vain yhteen tiedekuntaan
  - Jokainen henkilökunnan jäsen on töissä tietyllä laitoksella
  - Jokaisella laitoksella on yksi henkilökunnan jäsen esimiehenä
  - Yliopisto omistaa useita rakennuksia
  - Rakennuksessa voi sijaita yksi tai useampi laitos, kaikissa rakennuksissa tosin ei ole mitään laitosta
  - Laitos sijaitsee yhdessä tai joskus myös useammassa rakennuksessa

Yhteys vai ei?

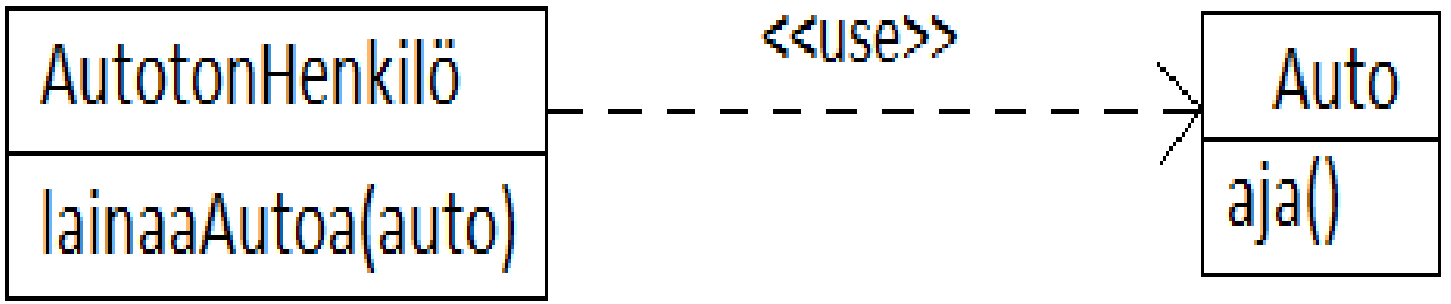
- Kuvitellaan, että on olemassa henkilöitä, jotka eivät omista autoa
- Autottomat henkilöt kuitenkin välillä lainaavat jonkun muun autoa
- Koodissa asia voitaisiin ilmaista seuraavasti:

```
public class AutotonHenkilo {
    public void lainaaJaAja( Auto lainaAuto ) {
        lainaAuto.aja();
    }
}
```

- Eli autottoman henkilön metodi lainaaJaAja saa parametrikseen auton (lainaAuto), jolla henkilö ajaa
- Auto on lainassa ainoastaan metodin suoritusajan
  - AutotonHenkilo ei siis omaa pysyvää suhdetta autoon

Ei yhteyttä vaan riippuvuus

- Kannattaako luokkien Auto ja AutotonHenkilo välille piirtää yhteys?
- Koska kyseessä ei ole pysyvämpiluontoinen yhteys, on parempi käyttää luokkakaaviossa *riippuvuussuhdetta* (engl. dependency)
- Riippuvuus merkitään katkoviivanuolena, joka osoittaa siihen luokkaan josta ollaan riippuvaisia
- Riippuvuus on tavallaan myös yhteys, mutta ”normaalia” yhteyttä ”heikompi” (= ei kestä yhtä kauaa)
- Alla on ilmaistu vielä riippuvuuden laatu
  - Tarkennin (eli stereotyyppi) <<use>> kertoo että kyseessä on käyttöriippuvuus, eli AutotonHenkilö kutsuu Auto:n metodia



Lisää riippuvuudesta

- Joskus riippuvuus määritellään siten, että luokka A on riippuvainen luokasta B jos muutos B:hen saa aikaan mahdollisesti muutostarpeen A:ssa
  - Näin on edellisessä esimerkissä: jos Auto-luokka muuttuu (esim. metodi aja muuttuu siten että se tarvitsee parametrin), joudutaan AutotonHenkilö-luokkaa muuttamaan
- Riippuvuus on siis jotain ”heikompaa” kun tavallinen luokkien välinen yhteys
  - Jos luokkien välillä on yhteys, on niiden välillä myös riippuvuus, sitä ei vaan ole tapana merkitä
  - Henkilö joka omistaa Auton on riippuvainen autosta...
- Toisin kuin yhteyksiin, **riippuvuuksiin ei merkitä kytkentärajoitteita**

Esimerkki, **Auto sisältää neljä rengasta**

```
public class Rengas{
    public void pyöri(){  System.out.println("pyörii");  }
}

public class Auto{  // neljä rengasta vasenEtu, oikeaEtu, ...
    private  Rengas vEtu;  Rengas oEtu; Rengas vTaka; Rengas oTaka;

   public Auto(){     // auto saa renkaansa syntyessään
      vEtu = new Rengas();     oEtu = new Rengas();
      vTaka = new Rengas();   oTaka = new Rengas();
    }

    public void aja(){  // ajaessa kaikki renkaat pyörivät
      vEtu.pyöri();    oEtu.pyöri():
      vTaka.pyöri();  oTaka.pyöri():
    }
```

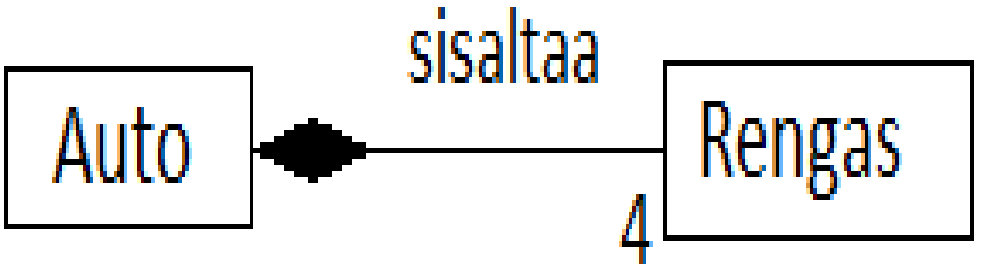
Kompositio

- Tilannehan voitaisiin mallintaa tekemällä autosta yhteys renkaisiin ja laittamalla kytkentärajoitteeksi 4
- Renkaat ovat kuitenkin siinä mielessä erityisessä asemassa, että voidaan ajatella, että ne ovat auton komponentteja
  - Renkaat sisältyvät autoon
- Kun auto luodaan, luodaan renkaat samalla
  - Koodissa auto luo renkaat
- Renkaat ovat private, eli niihin ei pääse ulkopuolelta käsiksi
- Kun roskienkerääjä tuhoaa auton, tuhoutuvat myös renkaat
- Eli ohjelman renkaat sisältyvät autoon ja niiden elinikä on sidottu auton elinikään (oikeat renkaat eivät tietenkään käyttäydy näin vaan ovat vaihdettavissa)

- Tämänkaltaista tilannetta, jota nimitetään **kompositioksi** (engl. composition), varten on oma symbolinsa, ks. seuraava sivu

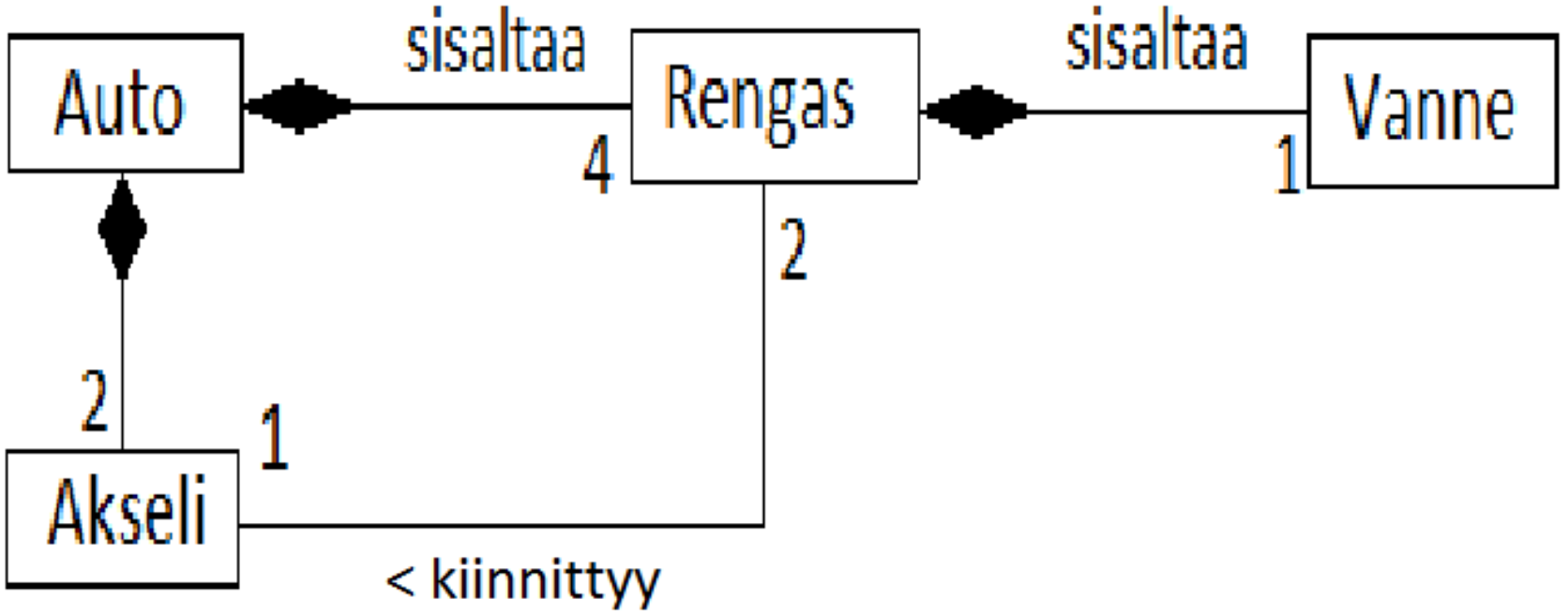
Kompositio

- Komposition symboli on ”musta salmiakkikuvio”, joka liitetään yhteyden siihen päähän, johon osat sisältyvät
- Kompositiota käytetään kun seuraavat ehdot toteutuvat:
  - *Osat ovat olemassaoloriippuvaisia kokonaisuudesta*
    - Auton tuhoutuessa renkaat tuhoutuvat
  - *Osa voi kuulua vaan yhteen kompositioon*
    - Rengasta ei voi siirtää toiseen autoon
  - *Osa on koko elinaikansa kytketty samaan kompositioon*
- Koska Rengas-olio voi liittyä nyt vain yhteen Auto-olioon, ei salmiakin puoleiseen päähän tarvita osallistumisrajoitetta koska se on joka tapauksessa 1



Monimutkaisempi esimerkki

- Tarkennettu Auto sisältää 4 rengasta ja 2 akselia
- Komposition osa voi myös sisältää oliota
  - Rengas sisältää vanteen
- Komposition osilla voi olla ”normaaleja” yhteyksiä
  - Akseli kiinnittyy kahteen renkaaseen
  - Rengas on kiinnittynyt yhteen akseliin

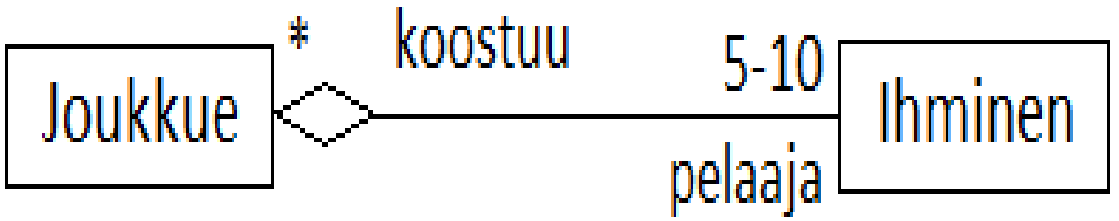


Kompositio, huomioita

- Onko kompositiomerkkiä pakko käyttää?
  - Ei, mutta usein sen käyttö selkiyttää tilannetta
- Kompositio on siis erittäin rajoittava suhde olioiden välillä, toisin kuin ”normaali” yhteys
  - Välimuotona seuraavalla sivulla esiteltävä *kooste*
- **HUOM**: auton ja renkaat sisältävä esimerkki kuvaa vaan esimerkkikoodi tilannetta mutta ei tietenkään ole realistinen kuvaamaan reaalimaailman autojen ja renkaiden suhdetta sillä normaalistihan renkaat eivät ole autoista olemassaoloriippuvaisia

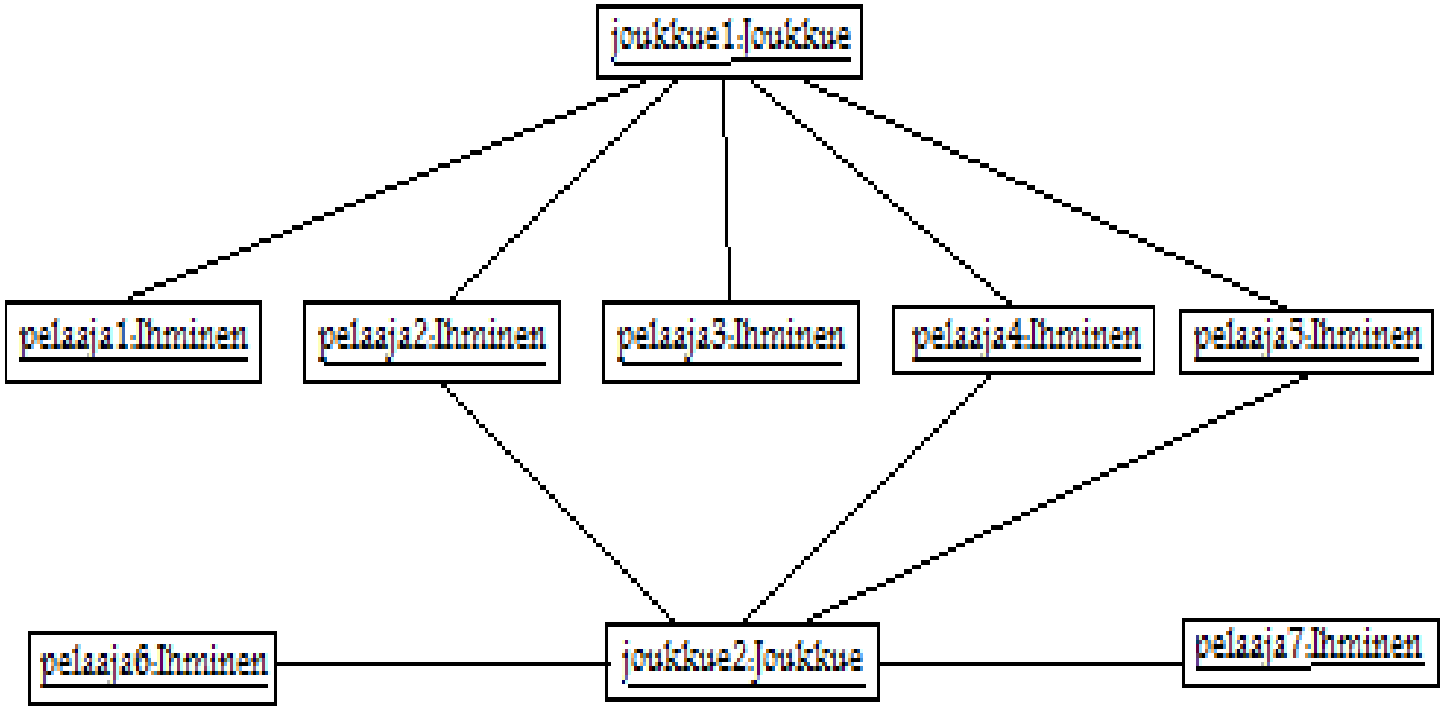
Kooste

- Koosteella (engl. agregation) tarkoitetaan koostumussuhdetta, joka ei ole yhtä komposition tapaan ”ikuinen”
  - koostesuhteisiin osallistumista ei ole rajoitettu yhteen
    - HUOM: suomenkieliset termit kooste ja kompositio ovat huonot ja jopa harhaanjohtavat
- Koostetta merkitään ”valkoisella salmiakilla” joka tulee siihen päähän yhteyttä, johon osat kuuluvat
- Esimerkki: Joukkue koostuu pelaajista (jotka ovat ihmisiä)
  - Ihminen ei kuitenkaan kuulu joukkueeseen ikuisesti
  - Joukkue ei syynnytä eikä tapa pelaajaa
  - Ihminen voi kuulua yhtäaikaa useampaan joukkueeseen
- Alla on rajattu, että joukkueella on 5-10 pelaajaa



Koosteen oliokaavio

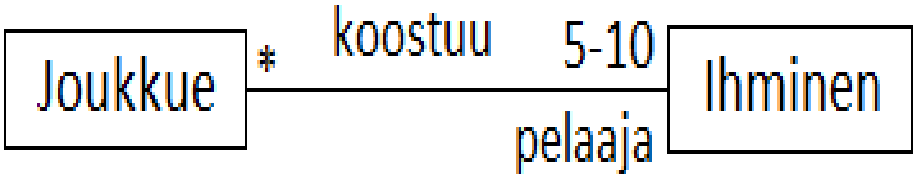
- Joukkueen 1 muodostavat pelaajat 1, 2, 3, 4 ja 5
- Pelaajat 2, 4, ja 5 kuuluvat myös joukkueeseen 2, jonka muut jäsenet ovat pelaajat 6 ja 7



Huomio koostesymboolin käytöstä

- Komposition (eli mustan salmiakin) merkitys on selkeä, kyseessä on olemassaoloriippuvuus
- On sensijaan epäselvää millon koostetta (eli valkoista salmiakkia) tulisi käyttää normaalin yhteyden sijaan
- Monet asiantuntevat oliomallintajat ovat sitä mieltä että koostetta ei edes tulisi käyttää
- Koostesuhde on poistunut UML:n standardista versiosta 2.0 lähtien
- Koostesuhde on kuitenkin edelleen erittäin paljon käytetty joten on hyvä tuntea symbooli passiivisesti

- **Tällä kurssilla koostetta ei käytetä eikä sitä tarvitse osata**
- Joukkueen ja pelaajien välinen suhde voidaankin ilmaista normaalina yhteytenä

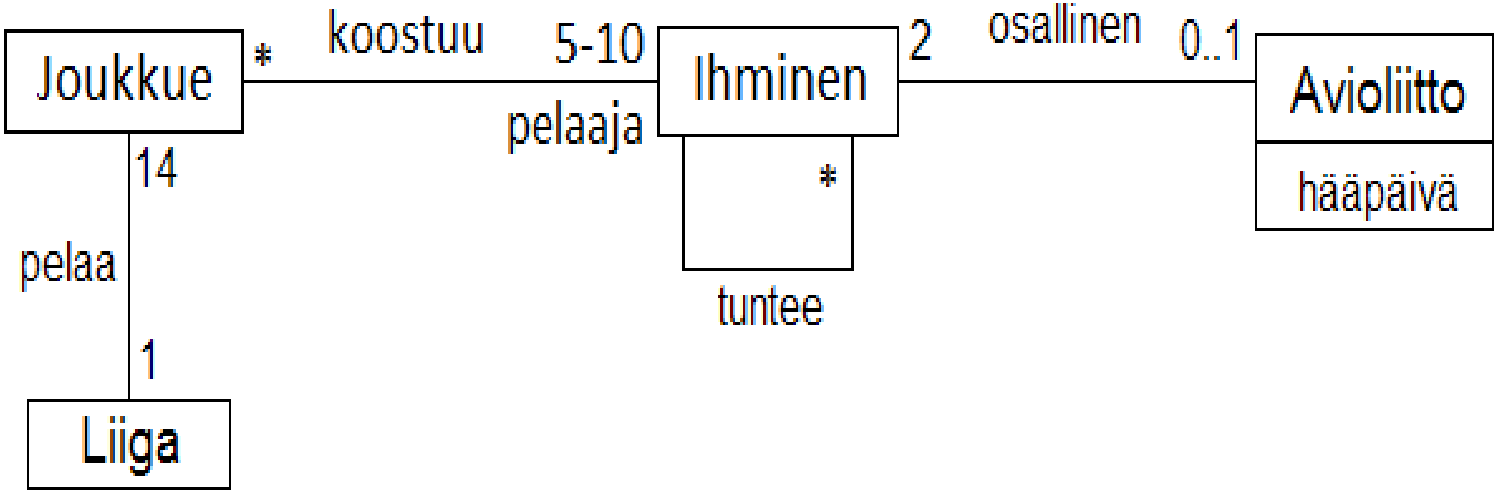


Monimutkaisempi esimerkki

- Joukkue pelaa liigassa, jossa on 14 joukkuetta
- Ihminen voi kuulua mielivaltaisen moneen joukkueeseen
- Joukkueeseen kuuluu 5-10 ihmistä
- Ihminen tuntee useita ihmisiä
- Ihminen voi olla avioliitossa, mutta vain yhdessä avioliitossa kerrallaan
- Avioliitto koostuu kahdesta ihmisestä

- Vastaus seuraavalla sivulla
- Huomaa miten yhteys tuntee on mallinnettu
  - Toisessa päässä osallistumisrajoite 1 ja toisessa *
  - Voisi olla myös * ja *
  - Tuntee on symmetrinen yhteys toisin kuin jossain aiemmassa esimerkissä ollut johtaa, joten toisen pään osallistumisrajoituksella ei ole merkitystä

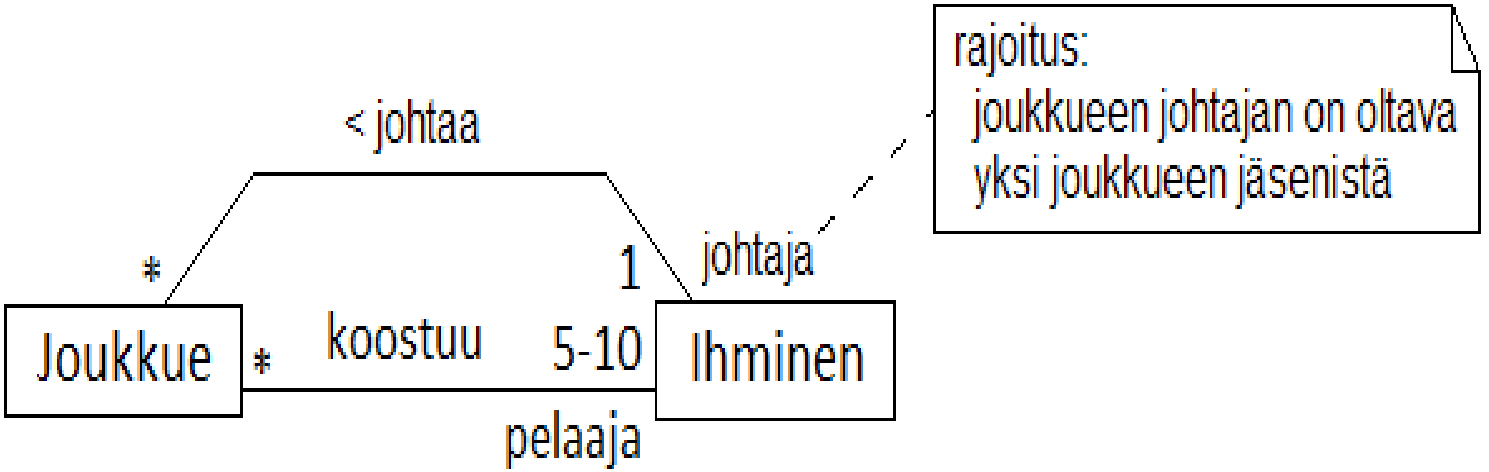
- Edellisen sivun tilannetta vastaava UML-kaavio:



- Entä jos tätä vielä laajennettaisiin seuraavasti:
  - Liigalla on sarjaohjelma
  - Sarjaohjelma sisältää 52 pelikierrosta
  - Kullakin pelikierroksella pelataan 7 ottelua
  - Ottelussa pelaa vastakkain 2 joukkuetta, joista toinen on kotijoukkue ja toinen vierasjoukkue

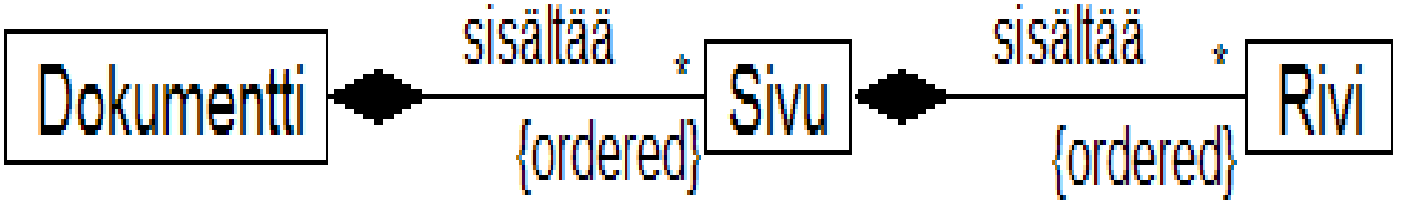
Rajoitukset

- Jos haluttaisiin mallintaa tilanne, että joku joukkueen jäsenistä on joukkueen johtaja, pelkkä luokkakaavio (siinä määrin kun tällä kurssilla UML:ää opitaan) ei riitä
- Tilanne voitaisiin mallintaa alla esitetyllä tavalla
  - Eli lisätään normaali yhteys *johtaa* joukkueen ja ihmisen välille
  - Määritellään kytkentärajoite: joukkueella on tasan 1 johtaja
  - Ilmaistaan UML-kommenttina, että joukkueen johtajan on oltava joku joukkueen jäsenistä



Yhteydessä olevien luokkien järjestys

- Esim. edellisessä esimerkissä joukkue koostuu pelaajista
  - Pelaajilla ei ole kuitenkaan mitään erityistä järjestystä yhteyden kannalta
- Joskus osien järjestys voi olla tärkeä
- Esim. dokumentti sisältää sivuja ja kukin sivu sisältää tekstirivejä
  - Sivujen ja rivien on oltava tietyssä järjestyksessä, muuten dokumentissa ei ole järkeä
- Se seikka, että osat ovat järjestyksessä voidaan ilmaista lisämääreenä {ordered}, joka laitetaan niiden olioiden päähän yhteyttä joilla on järjestys
  - Jos ei ole selvää minkä suhteen oliot on järjestetty, voidaan asia ilmaista kommentilla



Millä kaaviot kannattaa piirtää?

- Ensinnäkään, kaikkea ei tarvitse eikä kannatakaan tunkea samaan kuvaan
  - Nyrkkisääntönä voisi olla, että yhdessä kaaviossa kannattaa olla max 10 asiaa (= luokkaa, käyttötapausta, ...)
- Eli mitä työkaluja kannattaa käyttää?
- Kynä ja paperia tai valkotaulu
  - Paperi talteen tai roskiin
  - Tarvittaessa skannaus tai digikuva
- Tarjolla paljon ilmaisia työkaluja
  - Paint (suuri osa näidenkin kalvojen kuvista tehty paintilla)
  - Dia (win+linux)
  - Umbrello
  - ArgoUML
  - Openoffice
  - ...

- Paljon maksullisia vaihtoehtoja, mm.
  - Visual Paradigm, Magic Draw, Microsoft Visio, Omnigraffle (Mac)
- Magic Draw:n Community edition saatavissa ilmaiseksi, ks.
  - https://www.cs.helsinki.fi/intranet/group/cinco/teaching/md/
- Visio löytyy laitoksen ohjelmistojakelusta: http://www.cs.helsinki.fi/tietotekniikka/microsoftin-ohjelmia-opiskeluk-ytt-n
- Monet maksulliset tuotteet osaavat generoida luokkamäärittelyistä koodirunkoja ja myös toisinpäin, koodista UML:ää
- **Omat suosikkini tällä hetkellä ovat verkossa toimivat**
  - http://yuml.me luokka- ja käyttötapauskaavioihin
  - https://www.websequencediagrams.com Sekvenssikaavioihin
- Edellisissä kaaviot määritellään tekstinä. Myös seuraava verkossa toimiva editori on vähintään kelvollinen
  - https://www.draw.io/
- Ei siis ole olemassa selkeää vastausta mitä työkalua kannattaa käyttää. Tämän kurssin tarpeisiin kynä ja paperia riittää hyvin
- ”kymmenen sekunnin sääntö”