Ohjelmistotekniikan menetelmät

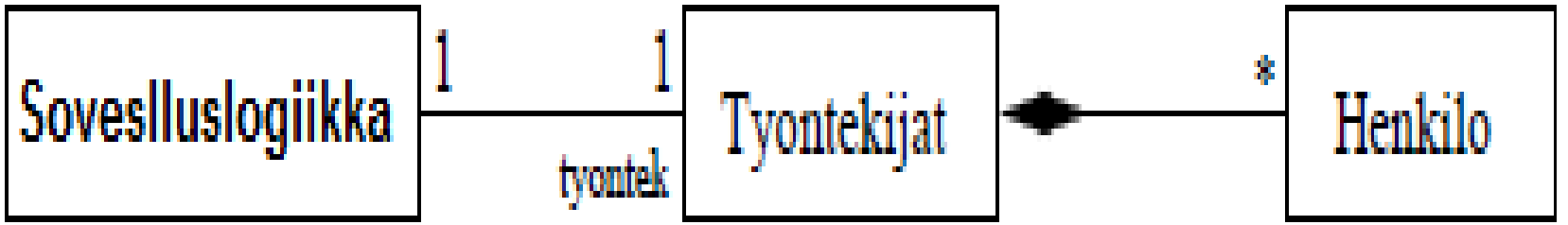
Luento 4, 17.11.

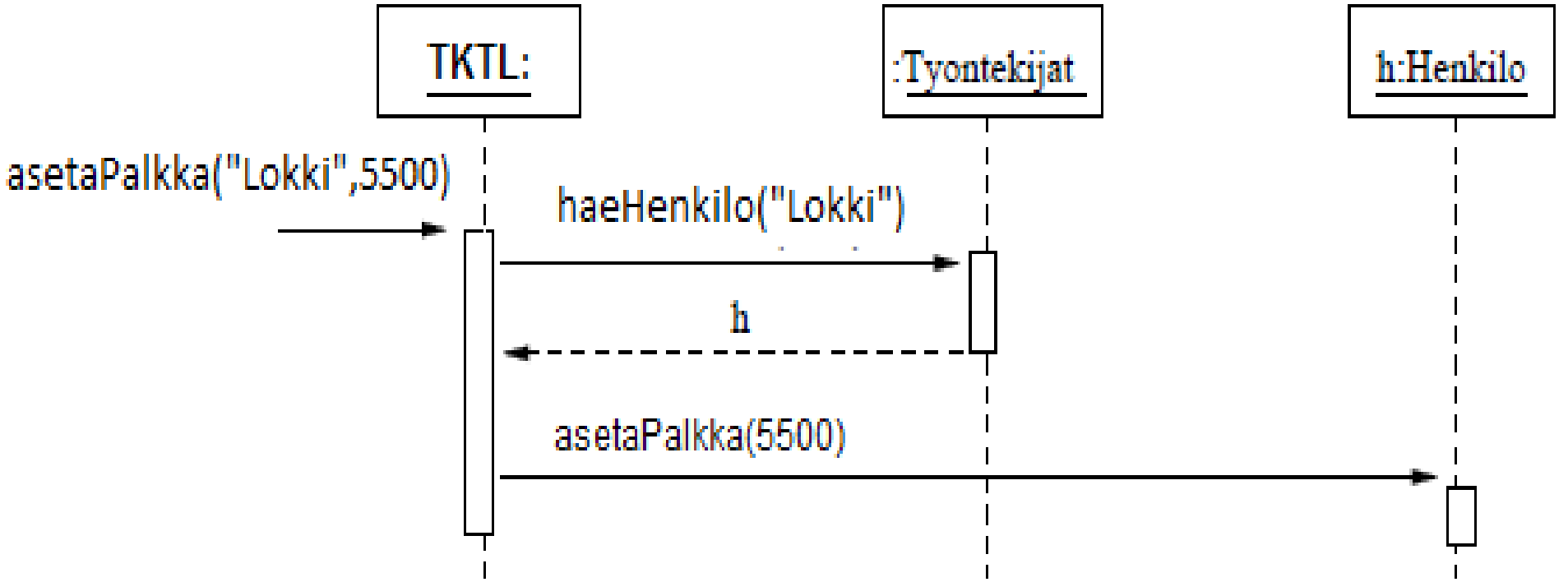
Kertaus: olioiden yhteistyön kuvaaminen

- 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 metodeja suorituksen aikana
  - Eli minkälaista yhteistyötä oliot tekevät
- Olioiden yhteistyön kuvaamiseen omat kaaviotyyppinsä:
  - Suositumpi **sekvenssikaavio**
  - ja vähemmän käytetty *kommunikaatiokaavio*
- Oliosuunnittelussa näillä kaavioilla erityisen tärkeä asema

- Seuraavilla kalvoilla esimerkki viimeviikolta

- Esimerkkisovellus:
  - Työntekijät-olio pitää kirjaa työntekijöistä, jotka Henkilö-oliota
  - Sovelluslogiikka-olio hoitaa korkeamman tason komentojen käsittelyn
- Tarkastellaan operaatiota lisaaPalkka(nimi, palkka)
  - Lisätään parametrina annetulle henkilölle uusi palkka
- Suunnitellaan, että operaatio toimii seuraavasti:
  - Ensin sovelluslogiikka hakee Tyontekijat-oliolta viitteen Henkilo-olion
  - Sitten sovelluslogiikka kutsuu Henkilo-olion palkanasetusmetodia
- Seuraavalla sivulla operaation suoritusta vastaava sekvenssikaavio
  - Havainnollistuksena myös osa luokan Sovelluslogiikka koodista






```
public class Sovelluslogiikka{
    Tyontekijat tyontek;     // attribuutti, jonka kautta sovelluslogiikka tuntee työntekijät
    public void lisaaPalkka(String nimi, int palkka ){
        Henkilo h = tyontek.haeHenkilo( nimi );
        h.asetaPalkka( palkka );
    }
}
```

Muutamia huomioita (ks. myös viikon 3 tehtävien mallivastaukset)

- Olioiden nimeäminen
  - nimi:Luokka, kumpikin osista voi jäädä pois jos asia muuten selvä
  - Olion nimiosaan voidaan viitata muualla kaaviossa, esim. paluuarvona tai metodin parametrina
- **Mistä mihin metodikutsua kuvaava nuoli piirretään?**
  - Kutsuvasta oliosta siihen olioon, kenen metodia kutsutaan
  - Esim. sovelluslogiikan koodissa metodikutsu

    Henkilo h = tyontek.haeHenkilo(nimi);

    kuvataan nuolena sovelluslogiikkaoliosta Tyontekijat-olioon
- Metodikutsujen parametrit merkataan tarvittaessa sulkuihin
- Paluuarvo merkataan tarvittaessa katkoviivalla tai metodikutsun yhteyteen
- Viittaus olioiden nimiin:
  - Esimerkissä h viittaa Henkilo-olioon, huomaa, miten h:ta käytetään metodin paluuarvona
- Vain alusta asti olemassaolevat oliot merkataan ylälaitaan, muut syntyhetken kohdalle

Yhteenveto olioiden yhteistoiminnan kuvaamisesta

- Sekvenssikaaviot ja luokkakaaviot eivät ole toistensa kilpailijoita, molemmat kuvaavat järjestelmää omasta näkökulmastaan

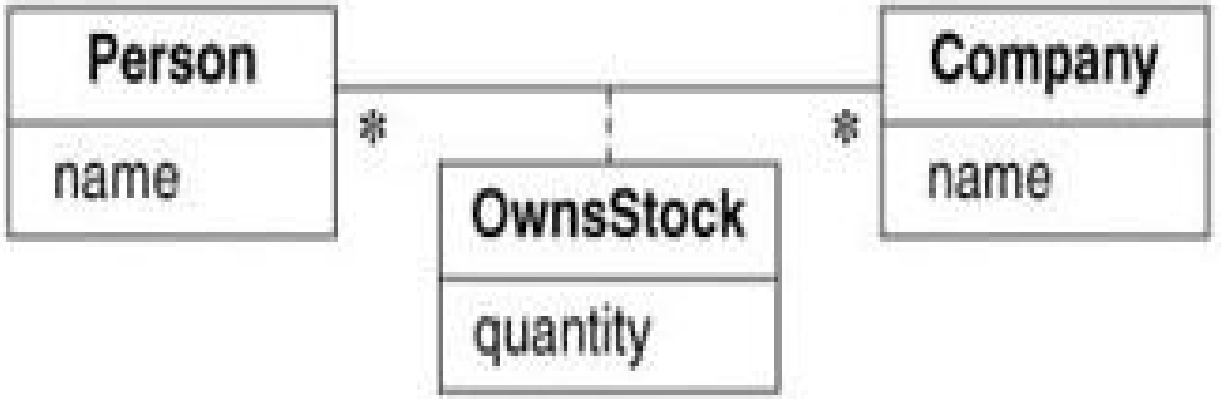
- Sekvenssikaaviot ja edellisellä luennolla nopeasti käsitellyt kommunikaatiokaaviot ovat erittäin tärkeä asema oliosuunnittelussa
  - Sekvenssikaaviot ovat huomattavasti yleisemmässä käytössä
- Kaaviot kannattaa pitää melko pieninä ja niitä ei kannata tehdä kuin järjestelmän tärkeimpien toiminnallisuuksien osalta
  - Kommunikaatiokaaviot ovat yleensä hieman pienempiä, mutta toisaalta metodikutsujen ajallinen järjestys ei käy niistä yhtä hyvin ilmi kuin sekvenssikaavioista
- On epäselvää missä määrin luennolla 3 mainittuja sekvenssikaavioiden valinnaisuutta ja toistoa kannattaa käyttää

- Sekvenssikaaviot on alunperin kehitetty tietoliikenneprotokollien kuvaamista varten

Yhteyteen liittyvät tiedot

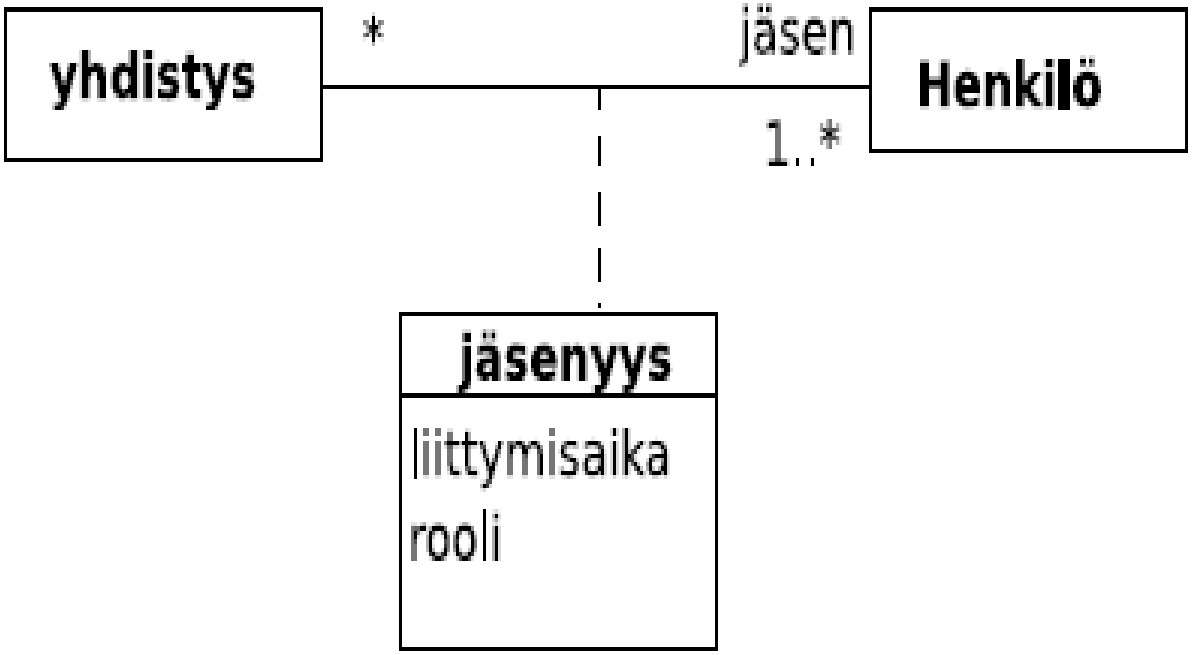
Yhteyden tietojen mallinnus

- Yhteyteen voi joskus liittyä myös tietoa
- Esim. tilanne missä henkilö voi olla (usean) yhtiön osakkeenomistaja
  - Osakkeenomistuksen kannalta tärkeä asia on omistettujen osakkeiden määrä
- Yksi tapa mallintaa tilanne on käyttää **yhteysluokkaa** (engl. association class), eli yhteyteen liittyvää luokkaa, joka sisältää esim. yhteyteen liittyviä tietoja
  - Alla yhteysluokka sisältää omistettujen osakkeiden määrän



Toinen yhteysluokkaesimerkki

- Luentomonisteessa mallinnetaan tilanne, jossa henkilö voi olla jäsenenä useassa yhdistyksessä
  - yhdistyksessä on vähintään 1 jäsen
- Jäsenyys kuvataan yhteytenä, johon liittyy yhteysluokka
  - jäsenyyden alkaminen (liittymisaika) sekä jäsenyyden tyyppi (rooli, eli onko rivijäsen, puheenjohtaja tms...) kuvataan yhteysluokan avulla



Kannattaako yhteysluokkia käyttää?

- Kannattaako yhteysluokkia käyttää?
  - Korkean tason abstrakteissa malleissa ehkä
  - Suunnittelutason malleissa todennäköisesti ei, sillä ei ole selvää, mitä yhteysluokka tarkoittaa toteutuksen tasolla
- Yhteysluokan voi aina muuttaa tavalliseksi luokaksi
- Yhteysluokka joudutaankin käytännössä aina ohjelmoidessa toteuttamaan omana luokkanaan, joka yhdistää alkuperäiset luokat joiden välillä yhteys on
  - Tämän takia yhteysluokkia ei välttämättä kannata käyttää alunperinkään

Yhteysluokasta normaaliksi luokaksi

- Alla muutaman sivun takainen osake-esimerkki ilman yhteysluokkaa
- Henkilöllä on useita ostoksia (purchase)
- Ostokseen liittyy määrä (kuinka monesta osakkeesta kyse), päiväys ja hinta
- Yksi ostos taas liittyy tasan yhteen yhtiöön ja tasan yhteen henkilöön
- Henkilö ei ole enää suorassa yhteydessä firmaan
  - Henkilö ”tuntee” kuitenkin omistamansa firmat ostosolioiden kautta

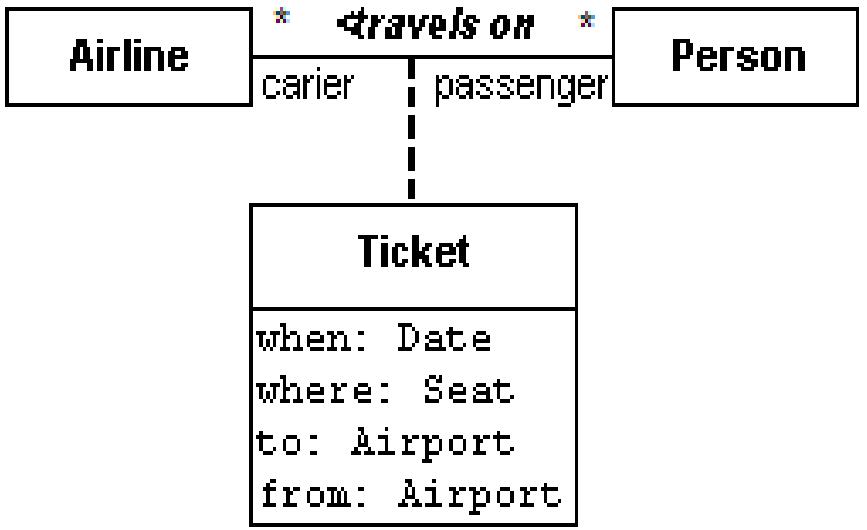
Yhteysluokasta normaaliluokaksi

- Henkilön jäsenyys yhdistyksessä on myös luontevaa kuvata yhteysluokan sijaan omana luokkanaan

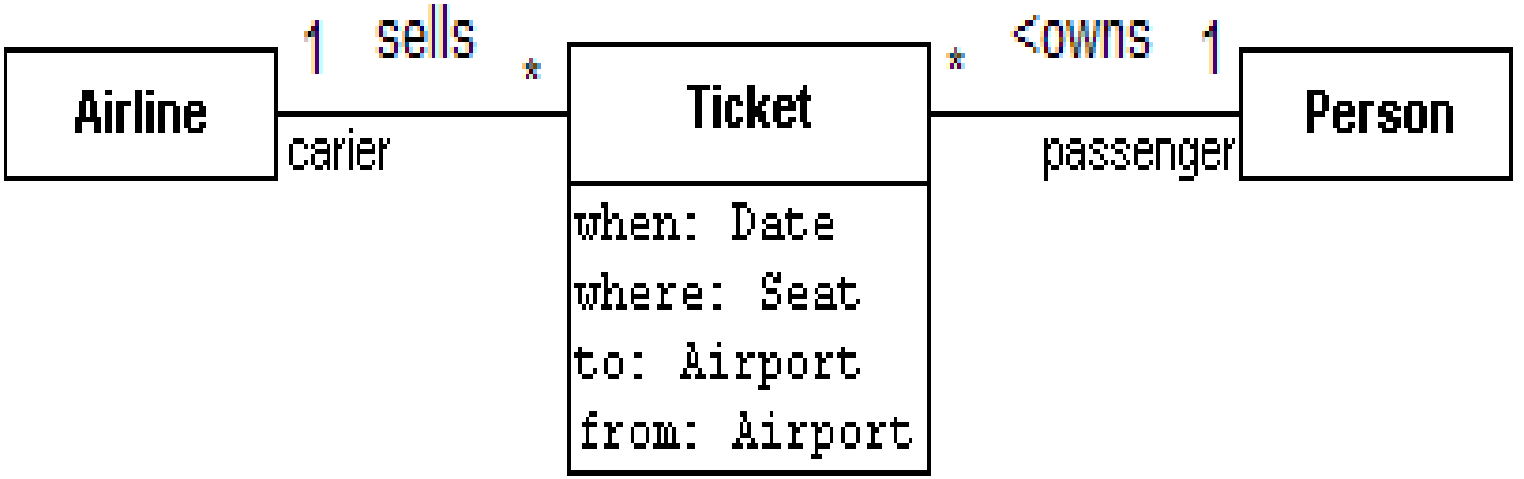
- Jäsenyyden olemassaoloriippuvuus on nyt merkitty henkilöön
- Miksi näin? Miksei yhdistykseen tai peräti molempiin?
- Periaatteessa loogisinta olisi tehdä jäsenyydestä olemassaoloriippuvainen sekä yhdistyksestä että henkilöstä, mutta UML ei salli tätä
- Olemassaoloriippuvuus saa siis olla vain yhteen olioon ja hetken mietinnän jälkeen on päätetty valita Henkilö jäsenyyden ”omistavaksi” osapuoleksi, periaatteessa Yhdistys olisi ollut yhtä hyvä valinta

Kaksi olioa ja yhteyksien lukumäärä

- Kurssin kotisivulle linkitetyssä Holubin UML quick referensessä on alla oleva esimerkki
- Eli henkilöllä voi olla *travels on* -yhteyksiä useiden lentoyhtiöiden kanssa
  - Yhteysluokkana Ticket on kerrottu matkan tiedot
- Tähän malliin sisältyy ongelma:
  - Henkilö voi olla *travels on* -yhteydessä moniin eri lentoyhtiöolioihin
  - Saman lentoyhtiöolion (esim. Finnair) kanssa ei kuitenkaan voi olla useampaa yhteyttä



- Siis: **jos luokkakaaviossa kahden luokan välillä on yhteys, voi kaksi luokkien olioa olla vain yhdessä yhteydessä kerrallaan**
  - Esim. olioiden arto:Person ja finnair:Airline välillä voi olla vain yksi yhteys, eli Arto voi lentää Finnairilla vain kerran
  - Tämä siitä huolimatta, että kytkentärajoitus on *
  - Artolla voi olla useita lippuja, mutta jokainen täytyy olla eri lentoyhtiöltä!
- Järkevin ratkaisu ongelmaan on kuvata yhteys omana luokkanaan
  - Henkilöllä voi olla useita lippuja
  - Lippu liittyy tiettyyn lentoyhtiöön ja tiettyyn henkilöön
  - Lentoyhtiön liittyy useita myytyjä lippuja



Määrittelyvaiheen luokkakaavion laatiminen

Kertaus ja esimerkki

Kertaus:
käsiteanalyysi eli menetelmä luokkamallin muodostamiseen

1. Etsi luokkaehdokkaat tekstikuvauksista (substantiivit)
2. Karsi luokkaehdokkaita (mm. poista tekemistä tarkoittavat sanat)
3. Tunnista olioiden väliset yhteyksiä (verbit ja genetiivit vihjeenä)
4. Lisää luokille attribuutteja
5. Tarkenna yhteyksiä (kytkentärajoitteet, kompositiot)
6. Etsi ”rivien välissä” olevia luokkia
7. Etsi yläkäsitteitä
8. Toista vaiheita 1-7 riittävän kauan

- Aloitetaan vaiheella 1, sen jälkeen edetään muihin vaiheisiin peräkkäin, rinnakkain tai/ja sekalaisessa järjestyksessä toistaen
- Lopputuloksena alustava toteutusriippumaton sovelluksen kohdealueen luokkamalli
  - Malli tulee tarkentumaan ja täsmentymään suunnitteluvaiheessa
  - Siksi ei edes kannata tähdätä ”täydelliseen” malliin

Esimerkki: kirjasto

- Tavoitteena on määritellä ja suunnitella tietojärjestelmä, jonka avulla hallitaan kirjaston lainaustapahtumia. Asiakasta haastattelemalla on kerätty lista järjestelmältä toivotusta toiminnallisuudesta:
  - Kirjasto lainaa alussa vain kirjoja, myöhemmin ehkä muitakin tuotteita, kuten CD- ja DVD-levyjä
  - Yksittäistä kirjanimikettä voi olla useampia kappaleita
  - Kirjastoon hankitaan uusia kirjoja ja kuluneita tai hävinneitä kirjoja poistetaan
  - Kirjastonhoitaja huolehtii lainojen, varausten ja palautusten kirjaamisesta
  - Kirjastonhoitaja pystyy ylläpitämään tietoa lainaajista sekä nimikkeistä
  - Nimikkeen voi varata jos yhtään kappaletta ei ole paikalla
  - Varaus poistuu lainan yhteydessä tai erikseen peruttaessa
  - Lainaajat voivat selata valikoimaa kirjastossa olevilla päätteillä
  - Kirjauduttuaan päätteelle omalla kirjastokortin numerolla on lainaajan myös mahdollista selailla omia lainojaan
  - Kirjaston päätteiden tarjoama toiminnallisuus on asiakkaiden käytössä myös Web-selaimen avulla

Esimerkki: kirjasto

- Etsitään substantiivit
  - **Kirjasto** lainaa alussa vain **kirjoja**, myöhemmin ehkä muitakin tuotteita, kuten **CD-** ja **DVD**-levyjä
  - Yksittäistä **kirjanimikettä** voi olla useampia **kappaleita**
  - Kirjastoon hankitaan uusia kirjoja ja kuluneita tai hävinneitä kirjoja poistetaan
  - **Kirjastonhoitaja** huolehtii **lainojen**, **varausten** ja **palautusten** kirjaamisesta
  - Kirjastonhoitaja pystyy ylläpitämään tietoa **lainaajista** sekä nimikkeistä
  - Nimikkeen voi varata jos yhtään kappaletta ei ole paikalla
  - Varaus poistuu lainan yhteydessä tai erikseen peruttaessa
  - Lainaajat voivat selata valikoimaa kirjastossa olevilla **päätteillä**
  - Kirjauduttuaan päätteelle omalla **kirjastokortin** numerolla on lainaajan myös mahdollista selailla omia lainojaan
  - Kirjston päätteiden tarjoama **toiminnallisuus** on **asiakkaiden** käytössä myös **Web-selaimen** avulla

Esimerkki: kirjasto

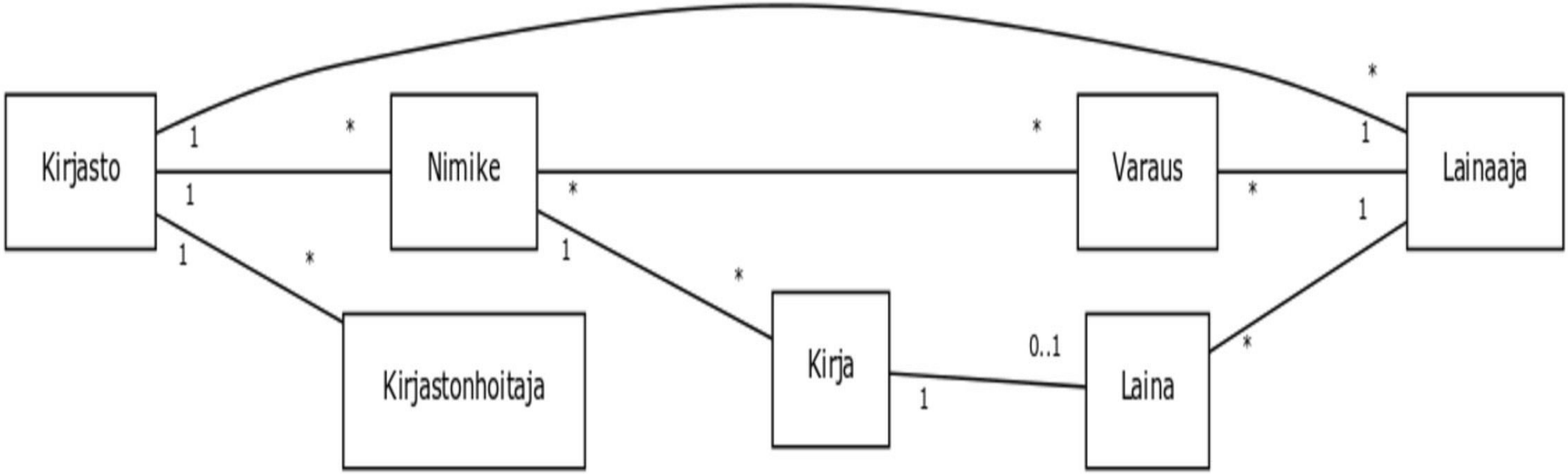
- Mietitään mitkä ovat oleellisia järjestelmän kannalta ja päädytään seuraaviin
  - Kirjasto
  - Kirjanimike
    - kirjaston valikoimassa oleva kirja
    - Lyhennetään muotoon *Nimike*
  - Kirja
    - edustaa yhtä fyysistä kopiota kirjanimikkeestä, eli tiettyä kirjanimikettä kirjastossa voi olla useampia kappaleita
  - Lainaaja
  - Kirjastonhoitaja
  - Laina
  - Varaus

Esimerkki: kirjasto

- Hylätyksi tulivat
  - CD ja DVD
    - Otetaan mukaan järjestelmään vasta jos todetaan tarpeelliseksi
  - Kappale
    - Kirjan synonyymi
  - Palautus
    - Tapahtuma missä *laina* ”poistuu”
  - Kirjastokortti
    - Oletetaan että lainaajalla on aina tasan yksi kortti
    - Saattaisi myös olla tilanteita, joissa oletettaisiin toisin. Asia syytä varmistaa ohjelmiston tilaajalta
  - Pääte ja Web-selain
    - Epäoleellisia asioita tietosisällön osalta
  - Toiminnallisuus
    - Epämääräinen käsite
  - Asiakas
    - Lainaajan synonyymi

Esimerkki: kirjasto

- Päädytään seuraavanlaiseen kaavioon



- Huomattavaa tässä on, että varaus kohdistuu aina tiettyyn nimikkeeseen, eli on oletettu että varaus ei yksilöi vielä fyysistä kirjaa johon se kohdistuu
- Eli samaan nimikkeeseen voi kohdistua useiden henkilöiden varauksia
- Laina sensijaan aina kohdistuu aina yhteen fyysiseen kirjaan, ja yksi kirja voi olla lainassa korkeintaan yhdellä henkilöllä kerrallaan

Huomioita määrittelyvaiheen luokkakaaviosta

- Substantiiveja saa yleensä siivota aika rankalla kädellä, läheskään kaikki eivät ole ”hyviä” luokkia
- Määrittelyvaiheen luokkakaaviota tehdessä ei kannata ajatella liikaa toiminnallisuutta eli sitä kuka tekee ja mitä
  - Toiminnallisuutta ajatellaan tarkemmin vasta suunnitteluvaiheessa
- Olioiden välisissä suhteissa huomio kannattaa kiinnittää pysyvämpiluonteisiin yhteyksiin
  - Esim. edellisessä tekstissä ilmenee että *Kirjastonhoitaja huolehtii lainojen, varausten ja palautusten kirjaamisesta. Nimikkeen voi varata jos yhtään kappaletta ei ole paikalla*. Luokkakaavion kannalta tässä ei ole oleellista kuka varauksen tekee vaan se että **nimikkeeseen voi liittyä varauksia**
  - Tekeminen siis välillä implikoi, että olioilla on pysyvämpiluonteinen yhteys
  - (Kirjastohoitaja-oliot eivät todennäköisesti ohjelmakoodissa tule olemaan niitä joka tekevät varauksen, kirjastohoitaja on lähinnä kirjastohoitajan tietoja tallettava entiteetti. Toiminnasta, esim. varauksen tekemisestä vastaa todennäköisesti joku muu olio)
- Olioiden välillinen hetkellinen yhteys merkitään joskus riippuvuutena
  - Riippuvuuksia merkitään kuitenkin aika harvoin määrittelyvaiheen luokkakaavioihin

Huomioita määrittelyvaiheen luokkakaaviosta

- Tekstuaalisista kuvauksista eivät läheskään aina kaikki tärkeät luokat tule esille
- Tosimaailmassa ei tietenkään aina ole mitään tekstuaalista kuvausta
  - Käsiteanalyysiä voi tehdä esim. käyttötapausten tekstuaalisille kuvauksille tai asiakkaan ”puheelle”
- Määrittelyvaiheen luokkakaavion tekemiseen ei kannata tuhlata hirveästi aikaa, on nimittäin (lähes) varmaa, että suunnittelu- ja toteutusvaiheessa luokkamallia muutetaan
  - Lopputulosta tärkeämpi on usein itse prosessi
- Suunnittelu- ja toteutusvaiheessa järjestelmään lisätään muutenkin luokkia hoitamaan esim. seuraavia tehtäviä
  - Käyttöliittymän toteutus
  - Tietokantayhteyksien hallinta
  - Sovelluksen ohjausoliot
- Määritteluvaiheen luokkakaavio toimii pohjana tietokantasuunnittelulle
  - Osa olioistahan (esimerkissämme lähes kaikki) on tallennettava

Mallinnuksen eteneminen

- Isoa ongelmaa kannattaa lähestyä pienin askelin, esim:
  - Yhteydet ensin karkealla tasolla, tai
  - Tehdään malli pala palalta, lisäten siihen muutama luokka yhteyksineen kerrallaan

- Mallinnus iteratiivisesti etenevässä ohjelmistokehityksessä
  - Ketterissä menetelmissä suositaan *iteratiivista* lähestymistapaa ohjelmistojen kehittämiseen
    - kerralla on määrittelyn, suunnittelun ja toteutuksen alla ainoastaan osa koko järjestelmän toiminnallisuudesta
  - Jos ohjelmiston kehittäminen tapahtuu ketterästi, kannattaa myös ohjelman luokkamallia rakentaa iteratiivisesti
  - Eli jos ensimmäisessä iteraatiossa toteutetaan ainoastaan muutaman käyttötapauksen kuvaama toiminnallisuus, esitetään iteraation luokkamallissa vain ne luokat, jotka ovat merkityksellisiä tarkastelun alla olevan toiminnallisuuden kannalta
  - Luokkamallia täydennetään myöhempien iteraatioiden aikana niiden mukana tuoman toiminnallisuuden osalta

Olioiden pysyväistallennus

Olioiden talletus tiedostoihin

- Suurin osa ohjelmista on sellaisia, että niiden käsittelemät tiedot on tallennettava johonkin pysyvästi
  - Kirjastoesimerkissä lähes kaikki oliot on tallennettava
  - Laskareissa käsittelemässämme olutkaupassa taas ostoskori saattaa olla sellainen, jota ei ole pakko tallettaa pysyvästi. Sovelluksen kannalta voi olla riittävää, että ostoskorioliot elävät ainoastaan yhden ostostapahtuman ajan
- Yksinkertainen ratkaisu ohjelman tietojen tallentamiseen on tiedostojen käyttö
- Käytettäessä normaaleja tekstitiedostoja, tulee ohjelmoijan päättää miten ohjelman oliot ”koodataan” tekstimuotoiseksi dataksi
  - Ohjelmoijan on kirjoitettava koodi, joka suorittaa olioiden tallentamisen tiedostoon sekä koodi, joka lukee tiedostoa ja luo tiedoston sisältöä vastaavat oliot
- Jos ohjelman tulee tallentaa olioita, joilla on yhteyksiä muihin olioihin, kuten kirjoihin liittyviä varauksia, varauksiin liittyviä asiakkaita ym., alkaa tilanne mutkistua, tiedostoon kirjoittavaa ja lukevaa koodia ei ole helppo tehdä ja koodia tulee paljon

Olioiden talletus tietokantaan

- Melkein kaikissa tilanteissa tekstitiedostoja paremman ratkaisun ohjelman tietojen tallettamiseen tarjoavat *tietokannat* (engl. Databases)
- Koska olioiden pysyväistalletus on niin yleinen ongelma, on olemassa paljon valmiita ohjelmakirjastoja, jotka helpottavat olioiden tallettamista tietokantaan
- Tietokantaratkaisuja on monenlaisia. Kurssilla *tietokantojen perusteet* tutustutaan *relaatiotietokantoihin*, noin viimeiset 30 vuotta hallitsevassa asemassa olleeseen tietokantaratkaisuun
- Viimeisen vajaan 10 vuoden aikana on kuitenkin syntynyt useita vaihtoehtoisia tietokantaratkaisuja. Tällä kurssilla tutustumme *dokumenttitietokantoihin* (engl. Document databases)
- Käytämme kurssilla MongoDB-nimistä dokumenttitietokantaa. Mongo on tällä hetkellä maailman neljänneksi käytetyin tietokanta. Mongo on suosittu erityisesti websovelluksissa
  - https://www.mongodb.org/
  - http://db-engines.com/en/ranking

JSON-dokumentit

- Useimmissa dokumenttitietokannoissa tiedot tallennetaan *JSON*:ina eli Javascript Object Notation -muodossa
  - JSON on tekstuaalinen tapa esittää Javascript-olioiden tilaa eli oliomuuttujien arvoja
- Pienintä dokumenttitietokantaan talletettavaa yksikköä sanotaan *dokumentiksi*
  - Dokumentti vastaa suurinpiirtein samaa kuin rivi relaatiotietokannassa
- JSON-dokumentit koostuvat joukosta avain-arvo-pareja
  - ks. seuraavan sivun esimerkki
- Koska JSON on tekstuaalinen esitysmuoto, voidaan JSON-dokumentteja tarvittaessa säilyttää vaikka merkkijonoissa tai tekstitiedostoissa
- JSON on erittäin yleinen formaatti tiedonsiirrossa web-selaimen ja web-palvelimien välillä

JSON-dokumentit

- Esim. opiskelijan tiedot voitaisiin esittää JSON:ina seuraavasti

  { ”nimi”: ”Arto Vihavainen”, ”opnro”: ”012345687”, ”opintopisteita”: 123 }

- Dokumenttiin liittyy nyt 3 avain-arvo-paria
  - avaimen *nimi* arvona on merkkijono ”Arto Vihavainen”
  - avaimen *opnro* arvona merkkijono ”012345687”
  - avaimen *opintopisteita* arvona kokonaisluku 123
- Aaltosulut ympäröivät JSON-dokumenttia ja avain-arvo-parit ovat dokumentin sisällä pilkulla erotettuina
- Usein JSONit esitetään siten, että jokainen avain-arvo-pari on omalla rivillään

```
{
    ”nimi”: ”Arto Vihavainen”,
    ”opnro”: ”012345687”,
    ”opintopisteita”: 123

}
```

- Huom: jatkossa emme merkitse JSON:ien avaimien ympärille hipsuja

JSON-dokumentit

- JSON-dokumentin avaimia vastaavat arvot voivat myös olla yksinkertaisia arvoja monimutkaisempia:

```
{
        nimi: ”Arto Vihavainen”,
        opnro: ”012345687”,
        puh: ”040-1234567”,
        osoite: {
                        katu: ”Mannerheimintie 10 A 1”,
                        postinumero: ”00100”
                },
        suoritukset: [ ”ohpe”, ”ohja”, ”tikape” ],
        ilmoittautumiset: [ ”otm”, ”tira” ]

}
```

- Esimerkissä avaimen *osoite* arvona on JSON-dokumentti, joka sisältää kaksi avain-arvo-paria
- Avaimien *suoritukset* ja *ilmoittautumiset* arvona taas on joukko merkkijonoja
- Tässä suhteessa JSON-dokumentit poikkeavatkin radikaalisti tavanomaisten relaatiotietokantojen riveistä, joissa yksittäisten kenttien arvot ovat yksinkertaisia arvoja

JSON-dokumentit

- Dokumenttitietokannoissa samantyyppistä asiaa edustavat dokumentit on tapana ryhmitellä *kokoelmiin* (engl. collection)
  - Esim. jos tietokantamme tallentaisi opiskelijoita ja kursseja, olisi loogista tallentaa molemmat omaan kokoelmaansa
- Kokoelma on käytännössä joukko tai taulukko JSON-dokumentteja, esim:

```
[
    {
        nimi: ”Arto Vihavainen”,
        opnro: ”012345687”,
        puh: ”040-1234567”,
        suoritukset: [ ”ohpe”, ”ohja”, ”tikape” ]

    },
    {
        nimi: ”Pekka Mikkola”,
        opnro: ”014125412”,
        opintopisteita: 125

    }
]
```

- Joukko siis esitetään listalla arvoja tai olioita ympäröitynä kulmasulkeilla

Kokoelmat

- Dokumenttitietokantojen kokoelma on vastine relaatiotietokannan tauluille
- Periaatteena on, että saman tyyppiset dokumentit, esim. kaikki opiskelijat talletetaan omaan kokoelmaansa
- Kuten edellinen esimerkki vihjaa, kokoelmassa olevat dokumentit eivät välttämättä ole rakenteeltaan täysin samanlaisia, eli niiden ei tarvitse sisältää sanoja avaimia
    - Tässä suhteessa dokumenttitietokannat poikkeavat radikaalilla tavalla relaatiotietokannoista
- Itseasiassa samassa kokoelmassa olevat dokumentit voivat sisältää vaikka täysin erilaisia avaimia, dokumenttitietokanta ei ota mitään kantaa siihen, minkälaisia tietyn kokoelman dokumentit rakenteeltaan ovat
- Sanotaankin että dokumenttitietokannat ovat *skeemattomia.* Tietokantaskeemaa, eli kuvausta kokoelmien sisältämien dokumenttien rakenteesta ei tietokannan tasolla ole ollenkaan olemassa
- Skeemattomuuden takia tietokantaan tietoa tallettava sovellusohjelma on täysin vastuussa siitä, että kantaan talletetaan sovelluksen kannalta oikean muotoista dataa, ja että kannasta haettava data osataan tulkita oikein

Dokumentin tunniste eli _id

- Jokaisella tietokantaan talletettavalla dokumentilla tulee olla kokoelman sisällä yksikäsitteinen tunniste avaimen *_id* arvona
- Kun JSON-dokumentti talletetaan tietokantaan, ja tunnisteelle ei ole annettu mitään arvoa, lisätään dokumentille automaattisesti generoitu yksikäsitteinen tunniste, esim.

```
{
    "_id" : ObjectId("563f460b227af455d2abae9e"),
    "nimi" : "Arto Vihavainen",
    "opnro" : "012345687"

}
```

- Automaattisesti generoitu tunniste on tyyppiä *ObjectId* oleva satunnainen merkkijono

Yksinkertaisen java-olion talletus dokumenttitietokantaan

- Perusideana on tallentaa tietyn luokan oliot omaan kokoelmaansa
  - Esim luokan Opiskelija oliot voitaisiin tallettaa kokoelmaan *opiskelija*
- Yksittäisen olion tallettaminen on suoraviivaista, muodostetaan dokumentti, joka tallettaa olion oliomuuttujien arvot
- Voisimme toteuttaa itse MongoDB:n Java-kirjastolla operaatiot, jotka tallettavat oliota kantaan ja lukevat kantaan talletettuja dokumentteja takaisin olioiksi
  - https://docs.mongodb.org/ecosystem/drivers/java/
  - Operaatioiden ohjelmointi olisi suoraviivaisia, mutta melko tylsää
- Onneksi Mongo tarjoaa **Morphia**-nimisen korkeamman tason kirjaston, ns. Object Document Mapperin (ODM:in), jonka avulla olioiden talletus ja lataaminen kannasta on vaivatonta
  - http://mongodb.github.io/morphia/
  - Morphia on toiminnallisuudeltaan hyvin samankaltainen kuin relaatiotietokannoille tarjolla olevat ORM:it (eli Object Relational Mapperit) kuten Hibernate ja EclipseLink tai OrmLite
- Tutustutaan nyt yksinkertaisten olioiden käsittelyyn Morphialla

Yksinkertaisen java-olion talletus dokumenttitietokantaan

- Oletetaan että haluamme tallettaa tietokantaan luokan Opiskelija oliota:

```
public class Opiskelija {
    private String nimi;
    private String opnro;
    private int opintopisteita;
    private int aloitusvuosi;
    private ArrayList<String> suoritukset;
}
```

- Otettuamme Morphia-kirjaston käyttöön projektissa, tulee luokan määrittelyn eteen liittää *annotaatio* @Entity kertomaan morphialle, että luokan oliota on tarkoitus tallentaa
- Luokalle tulee myös lisätä annotaatiolla @id varustettu ObjectId-tyyppinen oliomuuttuja dokumentin tunnisteen tallentamista varten
- Morphia edellyttää myös, että luokalla on parametriton konstruktori
- Morphiaa varten annotoitu luokka seuraavalla sivulla

Yksinkertaisen java-olion talletus dokumenttitietokantaan

- Annotoitu luokka Opiskelija

```
@Entity
public class Opiskelija {
    @Id
    private ObjectId id;
    private String nimi;
    private String opnro;
    private int opintopisteita;
    private int aloitusvuosi;
    private ArrayList<String> suoritukset;
}
```

- Oletusarvoisesti luokan oliot tulevat tallentumaan kokoelmaan nimeltä *opiskelija*. Kokoelmalle voi tarpeen vaatiessa antaa myös jonkin toisen nimen, nimemsi voitaisiin antaa esim. *students* seuraavasti:

```
@Entity(”students”)
public class Opiskelija {
```

Yksinkertaisen java-olion talletus dokumenttitietokantaan

- Tietokannan käsittelyä varten tarvitaan *Datastore*-tyyppinen olio
- Olio luodaan suunilleen seuraavasti

```
Datastore store = new Morphia().createDatastore(
        new MongoClient("http://kannan.osoite.com")), "kannan_nimi"
);
```

- Olion tallentaminen kantaan on helppoa:

```
Opiskelija arto = new Opiskelija(”Arto”, ”012345678”, 2001, 401);
arto.lisaaSuorius(”ohpe”);
arto.lisaaSuorius(”ohja”);
store.save(arto);
```

- Jos olion tila muuttuu, olio tulee tallettaa uudelleen, jotta muutos tallettuu tietokantaan

```
arto.lisaaSuorius(”tikape”);
store.save(arto);
```

Yksinkertaisen java-olion talletus dokumenttitietokantaan

- Olioa vastaava dokumentti näyttää seuraavalta

```
{
    _id: ObjectId("5640d8892c206640ebde55d6"),
    className: "com.mycompany.morphia.Opiskelija",
    nimi: "Arto",
    opnro: "012345678",
    opintopisteita: 401,
    aloitusvuosi: 2001,
    suoritukse: [
        "ohpe",
        "ohja",
        "tikape"
    ]
}
```

- Oliomuuttujien arvojen lisäksi Morphia-kirjasto on tallentanut olion yhteyteen avaimen *className* arvoksi tiedon siitä, minkä luokan olion dokumentti tallentaa

Olioiden hakeminen dokumenttitietokannasta

- Morphialla tallennettujen olioiden hakeminen tietokannasta on helppoa
- Tietojen haku tapahtuu luomalla kysely *store*-olion kautta
- Kysely, joka hakee kaikki kantaan tallennetut opiskelijat:

```
List<Opiskelija> kaikki = store.createQuery(Opiskelija.class).asList();
```

- Kyselyolion voi tarvittaessa luoda erikseen:

```
Query<Opiskelija> query = store.createQuery(Opiskelija.class);

List<Opiskelija> kaikki = query.asList();
```

- Huomaa, että samaa kyselyolioa *ei voi* uusiokäyttää monenlaisiin kyselyihin. Jatkossa on oletettu, että aina on käytössä uusi kyselyolio
- Kaikki vuoden 1999 jälkeen aloittaneet opiskelijat:

```
List<Opiskelija> osa = query.field("aloitusvuosi").greaterThan(1999).asList();
```

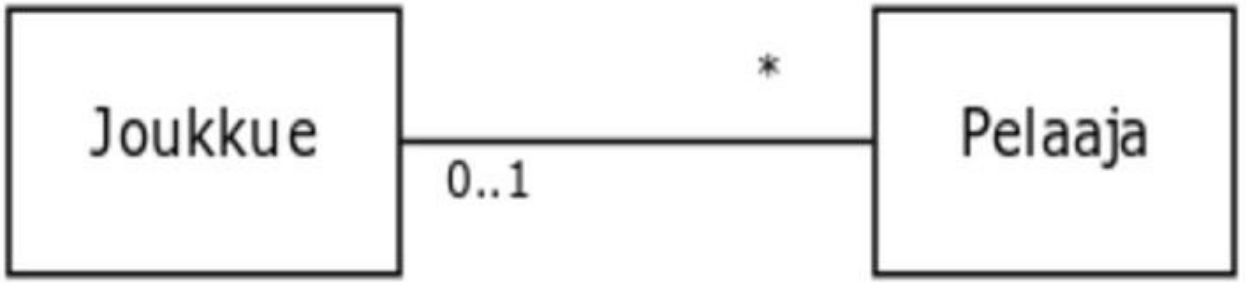
- Jos kyselyn vastauksena on odotettavissa täsmälleen yksi olio, voidaan vastaus pyytää metodilla *get*:

```
Opiskelija arto = query.field("nimi").equal("Arto Vihavainen").get();
```

- Palaamme myöhemmin hieman monimutkaisempien kyselyjen tekoon

Dokumenttien väliset yhteydet

- Tarkastellaan erilaisia mahdollisuuksia dokumenttien välisten yhteyksien tallettamiseen
- Keskitytään aluksi *yhden suhde moneen* -yhteyteen
- Esim. joukkueeseen kuuluu useita pelaajia ja pelaaja on vain yhdessä joukkueessa kerrallaan



- Jos sovelluksessa oltaisiin ensisijaisesti kiinnostuneita joukkueista ja tiettyyn joukkueeseen kuuluvista pelaajista, eli esim. *kaikkien* pelaajien listaa tarkasteltaisiin todella harvoin, voitaisiin pelaajaoliot *sisällyttää* (engl. embedded) joukkuetta esittettäviin dokumentteihin

Dokumenttien väliset yhteydet: olioiden sisällytys

- Joukkueen koodi olisi seuraava

```
@Entity
public class Joukkue {
      @Id
      ObjectId id;
      private String nimi;
      @Embedded
       private ArrayList<Pelaaja> pelaajat;
}
```

- Eli joukkueeseen liittyvään pelaajien listaan on lisätty annotaatio @Embedded
- Luokkaa pelaaja ei nyt merkitä ollenkaan tietokantaan talletettavaksi:
- public class Pelaaja {
      private String nimi;
      private int maaleja;
  }
- Joukkue luodaan ja talletetaan yllätyksettömällä tavalla:

```
Joukkue joukkue = new Joukkue(”IFK);
joukkue.lisaaPelaaja( new Pelaaja(”Pekka”, 5));
store.save(joukkue);
```

Dokumenttien väliset yhteydet: olioiden sisällytys

- Kaksi joukkuetta pelaajineen

```
{
        _id: ObjectId("5640ee842c2066422682baa6"),
       className: "com.mycompany.morphia.Joukkue",
       nimi: "HJK",
       pelaajat: [
             {  nimi: "Antti", maaleja: 7 },
             {  nimi: "Arto", maaleja: 3 }
       ]

},

{

       _id: ObjectId("5640ee842c2066422682baa7"),
       className: "com.mycompany.morphia.Joukkue",
       nimi: "IFK",
       pelaajat: [
             {  nimi: "Pekka",  maaleja: 5  }
       ]

}
```

Dokumenttien väliset yhteydet: olioiden sisällytys

- Ratkaisun huono puoli on se, että pelaajaolioihin ei ole muuta mahdollisuutta päästä käsiksi kuin hakemalla joukkueet tietokannasta
  - Tietokantatasolla pelaajien talletukseen ei ole olemassa omaa kokoelmaa
- Seuraavassa koodi, joka hakee joukkueet ja muodostaa kaikkien pelaajien listan

```
List<Pelaaja> pelaajat = new ArrayList<>();

for( Joukkue joukkue : store.createQuery(Joukkue.class).asList() ) {
    pelaajat.addAll(joukkue.getPelaajat());
}

for (Pelaaja pelaaja : pelaajat) {
    System.out.println(pelaaja);
}
```

- Ratkaisun hyviin puoliin kuuluu ainakin se, että kun joukkue on haettu tietokannasta, ovat joukkueen kaikkien pelaajien tiedot välittömästi saatavilla

Dokumenttien väliset yhteydet: pelaajatunnisteet joukkueeseen

- Usein on kuitenkin tarpeen, että kaikki oliot, esimerkissämme joukkueet ja pelaajat, talletetaan omiin kokoelmiin
- Tällöin olioiden väliset yhteydet toteutetaan linkittämällä toisiinsa liittyvät dokumentit tunnisteiden avulla
  - Dokumenttien välinen linkitys muistuttaa osin relaatiotietokantojen vierasavaimien käyttöä
- Jos riittää että joukkue tuntee pelaajat, tilanne hoidetaan seuraavasti:

```
@Entity
public class Joukkue {
    @Id
    ObjectId id;
    private String nimi;
    @Reference
     private ArrayList<Pelaaja> pelaajat;
}
```

```
@Entity
public class Pelaaja {
    @Id
    ObjectId id;
    private String nimi;
    private int maaleja;
}
```

- Eli pelaajat on nyt annotoitu siten, että ne talletetaan tietokantaan. Joukkuessa oleva oliomuuttuja *pelaajat* on nyt merkitty annotaatiolla **@Reference**. Tämän ansiosta Morphia sisällytää joukkueisiin ainoastaan pelaajien tunnisteet

Dokumenttien väliset yhteydet: pelaajatunnisteet joukkueeseen

- Luodaan kaksi joukkuetta ja niihin muutama pelaaja

```
Pelaaja p1 = new Pelaaja(”Arto”, 10);
Pelaaja p2 = new Pelaaja(”Pekka”,5);
Pelaaja p3 = new Pelaaja(”Mikko”, 17);
Joukkue ifk = new Joukkue(”IFK”);
Joukkue hjk = new Joukkue(”HJK”);
storage.save(p1, p2, p3, ifk, hjk);

ifk.lisaaPelaaja(p1);
ifk.lisaaPelaaja(p2);
storage.save(ifk);

hjk.lisaaPelaaja(p3);
storage.save(hjk);
```

- Kuten huomaamme, Datastorage-olion metodille *save* voi antaa mielivaltaisen määrän olioita tallennettavaksi
- Seuraavilla sivulla tietokantaan tallentuvat pelaaja- ja joukkuedokumentit

```
[ {
    _id: ObjectId("5641d80580905f5706e0220e"),
    className: "com.mycompany.morphia.Pelaaja",
    nimi: "Arto",
    maaleja: 10
  },
  {
    _id: ObjectId("5641d80580905f5706e0220f"),
    className: "com.mycompany.morphia.Pelaaja",
    nimi: "Pekka",
    maaleja: 5
  },
  {
    _id: ObjectId("5641d80580905f5706e02211"),
    className: "com.mycompany.morphia.Pelaaja",
    nimi: "Mikko",
    maaleja: 17
  } ]
```

```
[ {
    _id: ObjectId("5641d80580905f5706e0220d"),
    className: "com.mycompany.morphia.Joukkue",
    nimi: "IFK",
    pelaajat: [
        {   $ref: "Pelaaja",  $id: ObjectId("5641d80580905f5706e0220e") },
        {   $ref: "Pelaaja",  $id: ObjectId("5641d80580905f5706e0220f")  }
    ]
},
{
    _id: ObjectId("5641d80580905f5706e02210"),
    className: "com.mycompany.morphia.Joukkue",
    nimi: "HJK",
    pelaajat: [
        {   $ref: "Pelaaja", $id: ObjectId("5641d80580905f5706e02211") }
    ]
} ]
```

Dokumenttien väliset yhteydet: pelaajatunnisteet joukkueeseen

- Pelaajat siis tallentuvat omiin kokoelmiinsa luonnollisella tavalla, eli jokaisen oliomuuttujan arvo tallentuu pelaajan dokumenttiin
- Joukkueen pelaajalistalle tallentuu nyt ainoastaan *viite* pelaajaan. Viite on muotoa

  { $ref: "Pelaaja", $id: ObjectId("5641d80580905f5706e0220e") }

  eli viite sisältää kohteena olevan dokumentin tunnisteen, sekä tiedon että kyseessä on viite Pelaaja-tyyppisen olion tallettamaan dokumenttiin
- Viitteet tallentuvat siis melko luonnolliselta tuntuvalla tavalla
  - Relaatiotietokantoihin tottuneille tapa saattaa kuitenkin vaikuttaa aluksi oudolta, relaatiotietokannoissahan yhteys joukkueen ja pelaajien välille toteutettaisiin *pelaajaan* talletetulla joukkueen vierasavaimella
- Kyselyitä on nyt mahdollisuus kohdistaa suoraan pelaajiin tai joukkueisiin
- Esim. viisi eniten maaleja tehnyttä pelaajaa:

```
Query<Pelaaja> pQuery = store.createQuery(Pelaaja.class);

for (Pelaaja pelaaja : pQuery.order("-maaleja").limit(5).asList( )) {
      System.out.println(pelaaja);
}
```

Dokumenttien väliset yhteydet: pelaajatunnisteet joukkueeseen

- Joukkueet aakkosjärjestyksessä

```
Query<Joukkue> jQuery = store.createQuery(Joukkue.class);

for (Joukkue joukkue : jQuery.order("nimi").asList() ) {
      System.out.println(joukkue);
}
```

- Huomionarvoista on, että hakiessaan joukkueen tietokannasta, hakee Morphia kannasta samalla myös joukkueiden pelaajat:

```
Joukkue ifk = jQuery.field("nimi").equal("IFK").get();

for (Pelaaja pelaaja: ifk.getPelaajat() ) {
    System.out.println(pelaaja);
}
```

- Entä jos haluaisimme hakea kannasta joukkueen, jonka pelaajien yhteenlaskettu maalimäärä on suurin?
- Kysely ei onnistu sillä **dokumenttitietokannoissa kysely voi kohdistua ainoastaan yhteen kokoelmaan**
  - Pelaajathan ovat nyt omassa kokoelmassaan

Dokumenttien väliset yhteydet: pelaajatunnisteet joukkueeseen

- Eli kun haemme joukkueita kannasta

```
List<Joukkue> joukkueet = store.createQuery(Joukkue.class).asList() ;
```

  tekee Morphia todellisuudessa *kaksi* tietokantakyselyä, kokoelmaan joukkue ja kokoelmaan pelaaja, ja yhdistää näiden tuloksen siten, että sovellusohjelma saa käsiinsä pelaaja-oliota sisältäviä joukkue-olioita

- Koska joukkueiden ja pelaajien yhdistäminen *ei tapahdu tietokannan tasolla*, on mahdotonta tehdä kyselyä, jonka avulla kannasta saataisiin suoraan joukkue, jonka pelaajien yhteenlaskettu maalimäärä on suurin
  - Relaatiotietokantoja tuntevalle tämä on ehkä hienoinen yllätys, sillä useampien taulujen liitos on täysin standardioperaatio relaatiotietokannoissa
- Entä jos haluaisimme pystyä selvittämään tietokantatasolla eniten maaleja tehneen joukkueen? Vaihtoehtoja on kaksi
  - Siirryttäisiin ratkaisuun, jossa pelaajat olisi sisällytetty joukkueisiin
  - Talletettaisiin joukkueen pelaajien yhteenlaskettu maalimäärä joukkueen yhteyteen
- Molemmilla ratkaisuilla on omat hyvät ja huonot puolensa

Dokumenttien väliset yhteydet: joukkuetunnisteet pelaajiin

- Edellinen ratkaisu saattaa osoitteutua ongelmalliseksi, jos halutaan tietää *yksittäisen pelaajan joukkue*. Muuta mahdolisuutta tiedon hankkimiseen ei ole kuin joukkueen (sopivalla kyselyllä kaikkia ei tarvita) lukeminen tietokannasta
- Ratkaisua voi tarvittaessa laajentaa siten, että pelaaja tuntee joukkueen

```
@Entity
public class Joukkue {
    @Id
    ObjectId id;
    private String nimi;
    @Reference
    private ArrayList<Pelaaja> pelaajat;
}
```

-
```
@Entity
public class Pelaaja {
    @Id
    ObjectId id;
    private String nimi;
    @Reference
    Joukkue joukkue;
}
```

Dokumenttien väliset yhteydet: joukkuetunnisteet pelaajiin

- Luodaan 2 pelaajaa ja joukkue

```
Pelaaja p1 = new Pelaaja(”Arto”, 10);
Pelaaja p2 = new Pelaaja(”Pekka”, 5);
Joukkue ifk = new Joukkue(”IFK”);
storage.save(p1, p2, ifk);

ifk.lisaaPelaaja(p1);
ifk.lisaaPelaaja(p2);
p1.setJoukkue(ifk);
p2.setJoukkue(ifk);

storage.save(p1, p2, ifk);
```

- Huomaa, että oliot on talletettava kantaan ennen kuin pelaajan joukkue ja joukkueen pelaajat asetetaan, jos näin ei tehdä, eivät viitteet talletu tietokantaan
  - Oliot on talletettava kantaan myös pelaajien lisäyksen ja joukkueen asettamisen jälkeen
- Seuraavilla sivulla tietokantaan tallentuvat pelaaja- ja joukkuedokumentit

- Kokoelma pelaaja:

```
[
  {
          _id: ObjectId("5641d80580905f5706e0220e"),
          className: "com.mycompany.morphia.Pelaaja",
         nimi: "Arto",
         maaleja: 10,
          joukkue =  {   $ref ”Joukkue",  $id: ObjectId("5641d80580905f5706e0220d") }
    },
    {
        _id: ObjectId("5641d80580905f5706e0220f"),
          className: "com.mycompany.morphia.Pelaaja",
         nimi: "Pekka",
         maaleja: 5,
          joukkue =  {   $ref ”Joukkue",  $id: ObjectId("5641d80580905f5706e0220d") }
    }
]
```

- Kokoelma joukkue:

```
{
      _id: ObjectId("5641d80580905f5706e0220d"),
      className: "com.mycompany.morphia.Joukkue",
     nimi: "IFK",
     pelaajat: [
            {   $ref: "Pelaaja",  $id: ObjectId("5641d80580905f5706e0220e") },
            {   $ref: "Pelaaja",  $id: ObjectId("5641d80580905f5706e0220f")  }
     ]
}
```

Dokumenttien väliset yhteydet: 4 eri tapaa

- On siis *neljä* vaihtoehtoista tapaa hoitaa tilanne, missä joukkueeseen liittyy joukko pelaajia

  (1) Sisällytetään pelaajat joukkuedokumenttiin

  (2) Tallennetaan joukkueeseen pelaajien tunnisteet

  (3) Tallennetaan sekä pelaajien tunnisteet joukkueeseen että joukkueiden tunnisteet pelaajaan

  (4) Tallennetaan pelkästään pelaajaan sen joukkuetta vastaava tunniste

- Relaatiotietokannoissa yhden suhde moneen -yhteydet voidaan toteuttaa ainoastaan tavalla (4)
- Dokumenttitietokantojen yhteydessä yhteyksien toteutustapaa tuleekin miettiä sen kannalta miten sovellusta käytetään
- Itseasiassa vaihtoehtoisia tapoja on vieläkin enemmän
  - Olisi esim. mahdollista tallentaa joukkueet ja pelaajat omiin kokoelmiin ja *sen lisäksi* sisällyttää pelaajat joukkuedokumentteihin
  - Tällöin tietyt operaatiot helpottuisivat, mutta sama tieto olisi tietokannassa monessa paikassa ja tiedon ylläpitäminen hankaloituisi
- Palaamme *monen suhde moneen* -yhteyksien esittämiseen hieman myöhemmin

Olioiden tuhoaminen

- Olioiden tuhoaminen on helppoa. Riittää että muodostetaan kysely, joka palauttaa tuhottavat oliot ja annetaan kyselyolio parametriksi datastoragen metodille delete:

```
Query<Pelaaja> query = store.createQuery(Pelaaja.class);
store.delete(query.field("maaleja").lessThan(2));
```

- Ylläoleva esimerkki tuhoaa pelaajat, joiden maalimäärä on vähemmän kuin 2.

- Metodille delete voi antaa parametriksi myös viitteen kannasta haettuun olioon:

```
Pelaaja arto = query.field("name").equal("Arto").get();
store.delete(arto);
```

- Kannattaa huomata, että jos sovellus poistaa esim. jonkun pelaajaolion tietokannasta, on sovelluksen huolehdittava siitä, että pelaajaan ei viitata enää muualta tietokannan sisältä, esim. joukkueen pelaajalistasta

- Eli käytännössä edellisen lisäksi olisi suoritettava

```
Joukkue joukkue = arto.getJoukkue();
joukkue.poistaPelaaja(arto);
store.save(joukkue);
```

Olioiden päivittäminen

- Olion tietojen päivittäminen onnistuu luonnollisesti siten, että olioon tehdään muutos ja tallennetaan se uudelleen datastoragen metodilla save

```
Pelaaja arto = query.field("name").equal("Arto").get();
arto.setMaalimaara(25);
store.save(arto);
```

- Olioiden päivittämisen voi suorittaa myös tietokantatasolla. Tämä saattaa olla hyvä ratkaisu jos päivitettäviä oliota on suuri määrä
- Oletetaan, että pelaajilla olisi oliomuuttuja *palkka*, ja että haluaisimme korottaa kaikkien alle 1000 ansaitsevien pelaajien palkkaa 100:lla
- Päivitysoperaatiota varten tarvitsemme kyselyn lisäksi olion, joka määrittelee miten kyselyn palauttamien olioiden kenttiä tulisi päivittää
  - Esimerkkimme tapauksessa päivitysoperaatio on *inc* eli kasvatus

```
UpdateOperations<Pelaaja> payHundredMore =
    store.createUpdateOperations(Pelaaja.class).inc("palkka",100);

storage.update(query.field(”palkka”).lessThan(1000), payHundredMore);
```

Monimutkaisemmat kyselyt

- Tarkastellaan nyt hieman monimutkaisempien kyselyjen tekoa
- Oletetaan, että tietokantaan on tallennettu Opiskelijoita:

```
@Entity
public class Opiskelija {
	@Id
	private ObjectId id;
	String nimi;
	String opnro;
	int opintopisteita;
	int aloitusvuosi;
	@Embedded
	ArrayList<Suoritus> suoritukset;
	@Embedded
	Osoite osoite;
}
```

- Sisälletyt dokumentit määrittelevät luokat ovat:

```
public class Osoite {
	String katu;
	String postinumero;
	String kaupunki;
}
```

```
public class Suoritus {
	String kurssi;
	int arvosana;
}
```

Monimutkaisemmat kyselyt

- Haetaan opiskelijat, jotka ovat alottaneet vuoden 1999 jälkeen ja joilla on alle 200 opintopistettä

```
Query<Opiskelija> query = store.createQuery(Opiskelija.class);

query.and(
    query.criteria("aloitusvuosi").greaterThan(1999),
    query.criteria("opintopisteita").lessThan(200)
);

List<Opiskelija> tulos = query.asList();
```

- Sama kysely oltaisiin voitu rakentaa myös seuraavasti:

```
query.and(query.criteria("aloitusvuosi").greaterThan(1999));
query.and(query.criteria("opintopisteita").lessThan(200));
```

- Eli query-olioon on mahdollistaa liittää metodilla *add* uusia hakukriteerejä
- Jos haluttaisiin kohdistaa kysely ainoastaan Vihavainen-nimisiin opiskelijoihin, voitaisiin kyselyyn lisätä vielä yksi kriteeri

```
query.and(query.criteria("name").include(”Vihavainen”));
```

Monimutkaisemmat kyselyt

- Myös tai-ehdot ovat mahdollisia. Haetaan opiskelijat, joiden opintopistemäärä on yli 500 tai joiden nimi on Vihavainen tai jotka aloittivat ennen vuotta 1980

```
Query<Opiskelija> query = store.createQuery(Opiskelija.class);

query.or(
      query.criteria("aloitusvuosi").lessThan(1980),
      query.criteria("opintopisteita").greaterThan(500),
      query.criteria("name").include(”Vihavainen”)
);
```

- On myös mahdollista yhdistellä tai- ja ja-muotoisia ehtoja

```
query.and(
   query.criteria("aloitusvuosi").greaterThan(2014),
    query.or(
      query.criteria("opintopisteita").equal(0),
      query.criteria("opintopisteita").greaterThan(100)
    )
);
```

- Kysely hakee 2014 jälkeen aloittaneista ne, joilla on joko 0 tai yli 100 opintopistettä

Monimutkaisemmat kyselyt

- Koska osoite on sisällytetty opiskelijadokumenttiin, on kyselyissä mahdollista viitata myös osoitteen kenttiin. Tämä tapahtuu seuraavasti

```
query.field("osoite.kaupunki").equal("Helsinki");
```

- Kysely palauttaa kaikki Helsingissä asuvat opiskelijat.
- Kaikki opiskelijat, joiden suoritusten joukossa on *otm*:

```
query.field("suoritukset.kurssi").contains("otm");
```

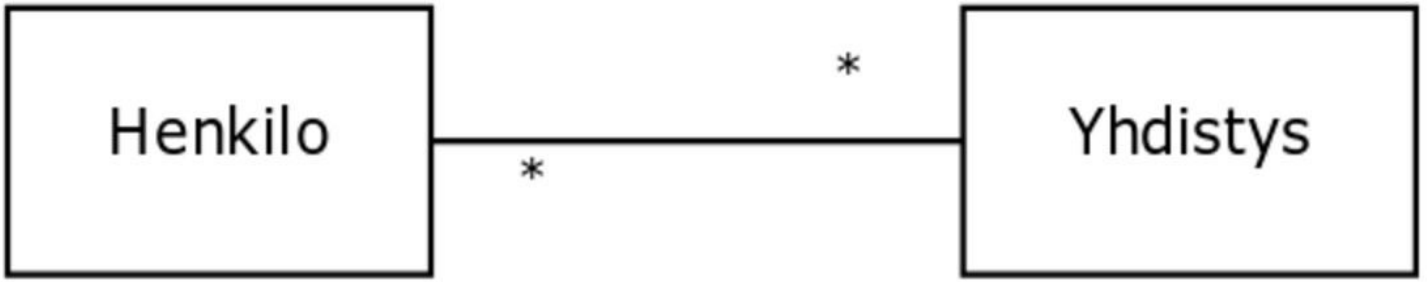
- Kaikki opiskelijat, jotka ovat suorittaneet kurssin *ohpe* arvosanalla 5:

```
Suoritus ohpe5 = new Suoritus(”ohpe”, 5);
query.field("suoritukset").hasThisOne(ohpe5);
```

- Edellinen olettaa, että luokalle Suoritus on määritelty equals-metodi, joka palauttaa true, kun verrataan samansisältöisiä oliota
- MongoDB tukee myös ns. aggregaattikyselyjä, joiden avulla tietokannasta voidaan ”laskea” erilaisia asioita, esim. *vuonna 2014 alottaneiden opiskelijoiden opintopistemäärien summa* tai *vähintään 55 opintopistettä vuodessa saaneiden opiskelijoiden prosenttiosuus aloitusvuosittain*
- Emme kuitenkaan käsittele agregaattikyselyjä tällä kertaa

Monen suhde moneen -yhteydet

- Toisin kuin relaatiotietokannant, dokumenttitietokannat pystyvät vaivattomasti esittämään myös *monesta moneen* -suhteita
- Oletetaan että meillä olisi henkilöitä ja yhdistyksiä, joiden välillä olisi monesta moneen -yhteys, eli henkilö voi kuulua moneen yhdistykseen ja yhdistyksessä voi olla monta henkilöä jäsenenä



- Tilanne hoidetaan MongoDB:llä seuraavasti
  - henkilödokumentti sisältää listan niiden yhdistysten tunnuksista, joihin henkilö kuuluu
  - yhdistysdokumentti sisältää listan niistä henkilöistä, jotka ovat yhdistyksen jäsenenä

- Henkilö-kokoelma

```
{
        _id: ObjectId(123),
        nimi: "Arto",
        yhdistykset: [ObjectId(201), ObjectId(202)]
},

{
        _id: ObjectId(124),
        nimi: "Pekka",
        yhdistykset: [ObjectId(202)]

}
```

- Yhdistys-kokoelma

```
{
        _id: ObjectId(201),
        nimi: "Kumpulan kyläyhdistys",
        jasenet: [ObjectId(123)]

},

{
        _id: ObjectId(202),
        nimi: "TKTL-alumni",
        jasenet: [ObjectId(123), ObjectId(124)]
}
```

Monen suhde moneen

- Henkilön ja yhdistyksen koodi ei sisällä mitään yllättävää:

```
@Entity
public class Henkilo {
    @Id
    private ObjectId id;
    private String nimi;
    @Reference
    List<Yhdistys> yhdistykset;
}
```

```
@Entity
public class Yhdistys {
   @Id
   private ObjectId id;
   private String nimi;
   @Reference
   List<Yhdistys> jasenet;
}
```

- Henkilöiden ja yhdistysten luominen ja niihin kohdistettavat kyselyt toimivat myös täysin samalla tavalla kuin aiemmissa esimerkeissä

- Jos yhteyteen liittyisi tietoja, esim. jäsenyyden alkamisaika. Jäsenmaksun suuruus ym. kannattaisi yhteys mallintaa omana luokkanaan:

- Tämä taas on analoginen sille, miten relaatiotietokannat toteuttavat monesta moneen -yhteydet *liitostaulujen* avulla

MongoDB konsoli

- Olemme tehneet kaikki tietokantaoperaatiomme Morphian kautta. Miten MongoDB:tä käytetään ilman apukirjastoja?
- Kaikissa Mongo-komennoissa parametri on JSON-muotoinen dokumentti
- Uusien dokumenttien luominen tapahtuu seuraavasti:

```
db.student.insert({
        ”nimi”: ”arto”,
        ”opintopisteita”: 100,
        ”osoite”: { ”katu”: ”mannerheimintie”,
                    ”kaupunki”: ”helsinki” }
});
```

- Esim hae 2014 jälkeen alottaneet opiskelijat, joilla alle 10 opintopistettä

```
db.student.find({ "$and" : [
                        { "aloitusvuosi" : { "$gt" : 2014}} ,
                        { "opintopisteita" : { "$lt" : 100}}
                    ]
                });
```

- Kyselyn JSON-muodon saa selville kutsumalla Morphian kyselylle *toString*
- Mongon käyttäminen ”natiivisti” kyselyjen tekemiseen on hieman ikävää. Tulemme kuitenkin laskareissa kokeilemaan myös JSON-muotoisia kyselyjä