1. Olio-ohjelmointi

Sisällys

- Olio-ohjelmointi on eräs ohjelmointiparadigma.
- Olio-ohjelmoinnin muotoja.
- Ohjelmiston analyysi ja suunnittelu.
- Olioparadigman etuja ja kritiikkiä.

Ohjelmointiparadigmoja

- Kaiken ohjelmoinnin takana ovat tietorakenteet ja algoritmit.
- Tietokoneohjelman toteuttamiseksi on kuitenkin tarjolla useita enemmän tai vähemmän toisistaan poikkeavia lähestymistapoja (eli ohjelmointi-paradigmoja).
- Paradigmoissa kiinnitetään vaihtelevasti huomiota toimintoihin ja tietoihin.

Ohjelmointiparadigmoja

- Proseduraalinen ohjelmointi:
  - Entinen valtaparadigma, jossa ohjelma jaetaan pienempiin osiin aliohjelmiksi (proseduureiksi).
  - Rakenteinen ohjelmointi eräs tämän paradigman muoto.
    - Kehitettiin ohjelmistokriisin ratkaisuksi 1970-luvulla.
    - GOTO-lause korvattiin ohjausrakenteilla (peräkkäisyys, valinta ja toisto), jolloin päästiin eroon "spagettikoodista".
    - Ohjelman rakennetta selvennetään sisentämällä.
    - Tuloksena helpommin ymmärrettäviä ja ylläpidettäviä ohjelmia.
  - Muun muassa Fortran, COBOL, Basic, Pascal, C, C++.

Ohjelmointiparadigmoja

- Olio-ohjelmointi:
  - Nykyisin vallitseva ohjelmointiparadigma.
  - Oliokielet tukevat olioparadigmaa vaihtelevasti.
  - Muun muassa Java, C++, Smalltalk ja Eiffel.
- Proseduraalinen ja olio-ohjelmointi ovat imperatiivisen ohjelmointiparadigman edustajia.
  - Ohjelmalla tila, jota muutetaan vaiheittain käskyillä.
- Funktionaalinen ohjelmointi (Lisp) ja logiikka-ohjelmointi (Prolog) kaksi muuta pääparadigmaa.

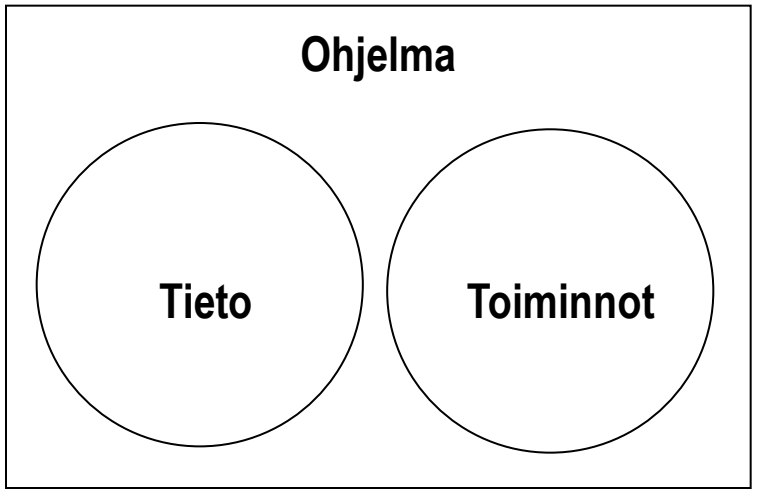
Ohjelmointiparadigmoja

- C++ on hybridikieli: oliomekanismit liitetty perinteiseen C-ohjelmointikieleen.
- Monet ohjelmointikielet ovat itse asiassa moniparadigmakieliä, joissa kieli sisältää usean ohjelmointiparadigman käsitteitä.
  - Esimerkiksi C++, PHP ja Common Lisp.
- Javaankin sisältyy rakenteisen ohjelmoinnin käsitteitä, vaikka Java luokitellaan (lähes) puhtaaksi olio-ohjelmointikieleksi.

Ohjelmointiparadigmoja

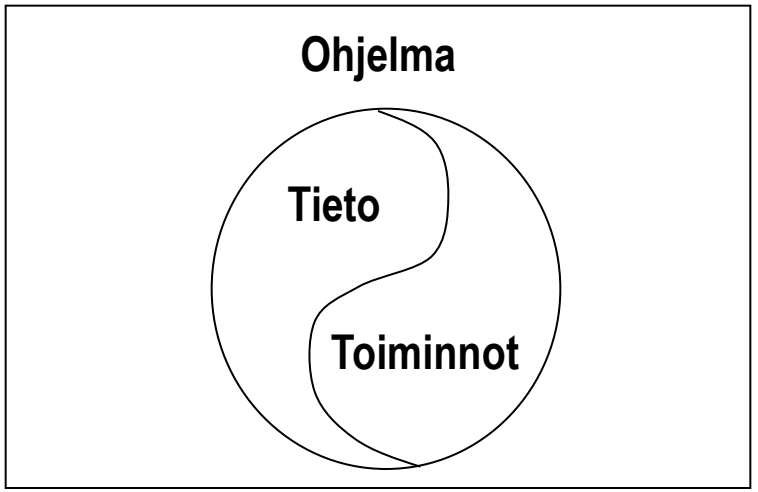
Rakenteinen ohjelmointi

- Tieto (muuttujat) ja siihen liittyvä toiminnallisuus (aliohjelmat, proseduurit, funktiot) erilliset.



Olio-ohjelmointi

- Tieto (attribuutit) ja siihen liittyvä toiminnallisuus (metodit) yhdistetty luokiksi.



Rakenteinen C++ -ohjelma

- Ohjelmoijan määrittelemä tietotyyppi, *HENKILO*-tietue, sisältää henkilön tiedot.
- Aliohjelma *tulostaHenkilo* tulostaa annetun tietueen.
- *pahis* ei ole olio, vaan *HENKILO*-tyyppinen muuttuja. Tästä syystä *pahis* on annettava tiedot tulostavalle aliohjelmalle parametrina.
- Ohjelmassa ei ole luokkamäärittelyä, vaan tietue ja aliohjelma ovat erilliset.

```
#include <iostream>
#include <string>
using namespace std;
struct HENKILO {
   string nimi;
};
void tulostaHenkilo(HENKILO apar) {
   cout << apar.nimi;
}
int main() {
   HENKILO pahis;
   pahis.nimi = "Tommy DeVito";
   tulostaHenkilo(pahis);
   return 0;
}
```

Olio-ohjelma Java-kielellä

- *Henkilo*-luokka sisältää nimen (attribuutti) ja nimen tulostamisen (metodi).
- Tiedot tulostetaan metodia pistenotaatiolla kutsumalla.
- Tietoja ei tarvitse antaa metodille parametrina, koska olio sisältää tiedot.
- Luokkamäärittely sitoo yhteen sekä tiedot että niihin liittyvät toiminnot.

```
public class Henkilo {
   public String nimi;
   public void tulosta() {
      System.out.print(nimi);
   }
}
public class HenkiloTesti {
   public static void main(String[] args) {
      Henkilo pahis = new Henkilo();
      pahis.nimi = "Tommy DeVito";
      pahis.tulosta();
   }
}
```

Olio-ohjelmoinnin muotoja

- Oliopohjainen ohjelmointi (object-based programming): Ohjelma kuvataan keskenään kommunikoivina olioina.
- Luokkapohjainen ohjelmointi (class-based programming): Olioiden lisäksi luokat.
- Olioperustainen ohjelmointi (object-oriented programming, OOP): oliot, luokat ja periytyminen.
- Kurssilla ohjelmoidaan kehittyneimmällä muodolla (OOP) Java-kieltä käyttäen.

Ohjelmiston analyysi ja suunnittelu

- Olioparadigma ei rajoitu pelkästään ohjelmointiin: Ohjelmistoja analysoidaan ja suunnitellaan olioperustaisesti (Object-Oriented Analysis/Design, OOA/D).
- Kurssilla käytetään UML:ää lähinnä OOA-tehtäviin.
- Luokkakaavioita tulee pian kalvoilla vastaan, mutta laajemmin UML opetetaan kurssin loppupuolella.
- Myöhemmillä kursseilla UML:ää enemmänkin sovelletaan kuin opetetaan.

Olioparadigman hyötyjä

- Vastaa paremmin ihmisen tapaa hahmottaa maailmaa, kuin esimerkiksi tiettyyn laskentamalliin perustuva ohjelmointiparadigma.
- Ohjelmisto on samanrakenteinen sovellusalueen käsitteiden kanssa, jolloin ylläpito on helpompaa ja ylläpitokustannukset pienempiä.
- Tukee ohjelmiston osien uudelleenkäyttöä.

Olioparadigman kritiikkiä

- Turhan raskas ja monimutkainen lähestymistapa pienten sovellusten toteuttamiseen.
- Olioparadigmaan kuuluva hierarkkinen periytymismekanismi ei sovi jokaisen sovellusalueen kuvaamiseen.
- Olioperustaisuuden odotetut hyödyt jääneet osin toteutumatta.
- Olioperustaisuudesta voi olla enemmän haittaa kuin hyötyä ensimmäisellä ohjelmointikurssilla.

2. Olio-ohjelmoinnin perusteita

Sisällys

- Esitellään peruskäsitteitä yleisellä tasolla:
  - Luokat ja oliot.
  - Käsitteet, luokat ja oliot.
  - Attribuutit, olion tila ja identiteetti.
  - Metodit ja viestit.
- Olioperustainen ohjelmointi.

Luokat ja oliot

- Luokka (class) ja olio (object) ovat olio-ohjelmoinnin keskeisimpiä käsitteitä.
- Liittyvät kiinteästi yhteen, ovat kuitenkin eri asia.
- Luokka voidaan ajatella yleiseksi malliksi, josta luodaan mallin mukaisen rakenteen ja käytöksen omaavia olioita.
- Luokka määrää siis olion attribuutit (ominaisuudet) ja metodit (käytös) eli olion piirteet.
- Abstraktisti: Olio on luokan ilmentymä (instance).

Luokat ja oliot

- Luokkien ja olioiden suhteesta seuraa:
  - Oliolla on aina luokka.
  - Luokalla voi olla 0, 1 tai useampia olioita.
- Luokka vastaa karkealla tasolla maailman käsitettä (concept).
  - Tämä selittää pitkälti olio-ohjelmoinnin suosion.

Käsitteet

- Ajatusmaailman mielikuvia, jotka auttavat hahmottamaan maailmaa ja tunnistamaan sen olioita.
- Ilmaistaan kielellisesti määritelmän avulla.
- Käsitteeseen viitataan sanalla tai termillä.
  - Esim. Käsite *koira* voidaan määritellä “Koira on nelijalkainen eläin, jolla on häntä ja joka haukkuu.”
- Käsite voi olla konkreettinen tai abstrakti.
- Käsitteet voivat olla myös kuvitteellisia.
  - Esim. *Örkki*-käsite ei ole kotoisin tästä maailmasta.

Käsitteet, luokat ja oliot

- Olio-ohjelmoinnissa sovellusalueen ja sen ympäristön käsitteitä mallinnetaan luokiksi, joista luotavat oliot muodostavat ohjelman.
- Luokkia ei voida aina tunnistaa suoraan. Tällöin analysoidaan sovellusalueen olioita (eli käsitteiden edustajia), jotta löydetään käsitteitä.
- Käsitteen “vangitseminen” sellaisenaan on lähes mahdotonta. Tämä ei ole onneksi tarpeen, koska riittää, että käsitettä vastaava luokka sisältää sovelluksen kannalta tärkeän osuuden käsitteestä.

Käsitteet, luokat ja oliot

- Luokaksi soveltuvat käsitteet yleensä substantiiveja:
  - kissa, koira, auto, henkilö, levy, kahvinkeitin, ...
- Luokkien ominaisuuksia (attribuutit) ja toimintoja (metodit) ei tulisi mallintaa luokiksi.
  - Ominaisuutta ilmaisevat substantiivit (esim. väri, koko ja ikä) ovat attribuutteja. Yksittäisellä arvolla ilmaistava käsite on usein attribuutti (esim. opiskelijanumero).
  - Toimintaa ilmaisevat käsitteet (esim. lisäys, poisto, tallentaminen ja hakeminen) ovat luonnollisesti metodeja.

Käsitteet, luokat ja oliot

- Esim. Olkoon sovellusalue kirjasto. Käsitteitä: teos, lainaus, varaus, asiakas, sakko jne. *Asiakas*-luokkaan voitaisiin sijoittaa vaikkapa vain asiakkaan nimi ja yhteystiedot. *Asiakas*-käsitteen edustajat ovat konkreettisesti olemassa. Toisaalta asiakkaat “elävät” myös *Asiakas*-luokan olioina ohjelmassa.
- Oliolla on siis kaksi roolia:
  - Sovellusalueen käsitteen edustaja.
  - Käsitettä vastaavan luokan edustaja.

Attribuutit

- Olion ominaisuudet määritellään attribuuteilla (attribute), jotka voivat esiintyä yhdessä tai useammassa luokassa.
- Esimerkiksi sekä kissoilla että linnuilla on väri. Toisaalta kissoilla on häntä ja linnuilla on pyrstö.
- Yleensä attribuutein pyritään esittämään vain tärkeimmät ominaisuudet.

Olion tila

- Oliolla on tila (state), jonka attribuutin arvot määräävät.
- Jos esimerkiksi Möykyn häntä on kippura ja Rontin häntä tavallinen, on näillä kissaolioilla eri tila, vaikka molemmat ovat mustia *Kissa*-luokan edustajia.

Möykky

Rontti

Olio	Häntä	Väri
Möykky	kippura	musta
Rontti	tavallinen	musta

Olion identiteetti

- Olioilla on myös identiteetti (identity), joka avulla olio voidaan erottaa yksikäsitteisesti muista olioista.
- Olkoon esimerkiksi Möykky ja Mörkö mustia kippurahäntäisiä *Kissoja*. Vaikka kissaolioiden tila onkin sama, ovat oliot kissojen tapaan ainutkertaisia ja erillisiä yksilöitä.

Möykky

Mörkö

Olio	Häntä	Väri
Möykky	kippura	musta
Mörkö	kippura	musta

Metodit

- Metodit (method) mallintavat olioiden käytöksen eli tietyn luokan olioille ominaiset toiminnot.
- Metodit voivat muuttaa olion tilaa.
- Esimerkiksi *Kissat* osaavat muun muassa syödä, äännellä, raapia, leikkiä ja nukkua. Nämä toiminnot ilmenevät vaikkapa Mörkö-kissan käytöksestä.
- Annetaan metodien nimet käskymuodossa (yksikön imperatiivi): *syö*(mitä), *ääntele*(ääni), *raavi*(mitä), *leiki*(millä) ja *nuku*(aika).

Metodit

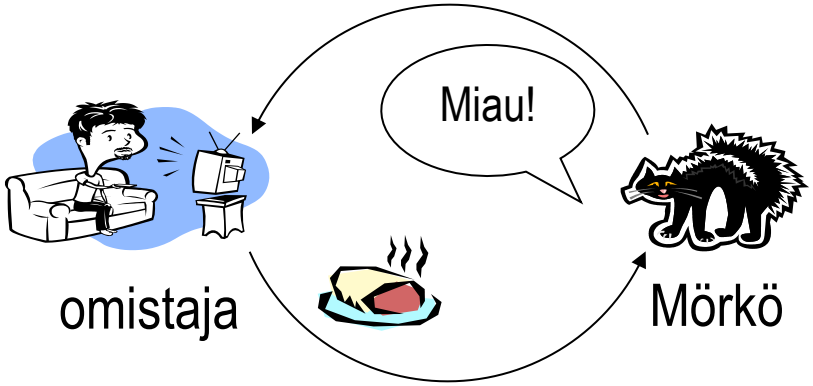
- Esim. Tietokoneohjelman *painike* voi olla aktiivinen tai passiivinen ja painiketta klikataan hiirellä. Metodeja: *aktivoi*(), *passivoi*(), *ilmoitaKlikkaus*().
- Esim. *Murtoluvut ⅓* ja *⅔* voidaan vaikkapa kertoa ja yksittäinen murtoluku esimerkiksi supistaa. Metodeja: *kerro*(murtoluku), *supista*().

Viestit

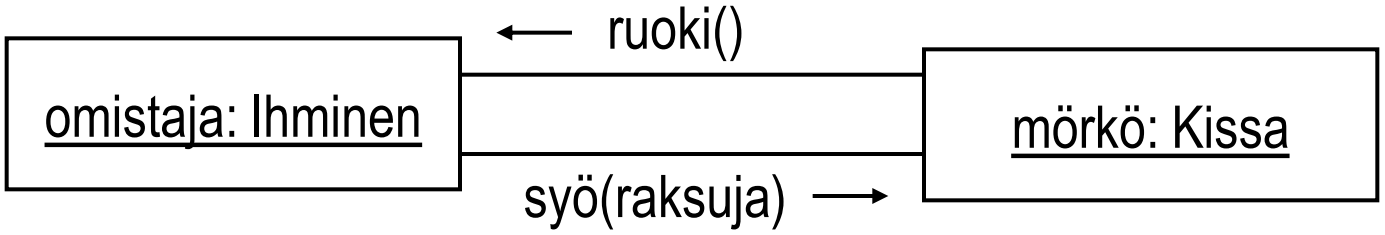
- Olioparadigmassa ohjelma on joukko keskenään “keskustelevia” olioita.
- Abstraktisti voidaan ajatella, että oliot kommunikoivat viesteillä (message).
- Luonteva ajattelumalli, koska myös reaalimaailman oliot lähettävät viestejä toisilleen.
- Lähettäjän (sender) viestin aktivoi vastaanottajan (receiver) metodin.
- Olio ymmärtää vain metodeihinsa liittyvät viestit.

Viestit

- Mörkö-kissa voi esimerkiksi naukua omistajalleen, joka sitten ruokkii käskystä kissansa raksuilla.
- Reaalimaailma:



- Oliokaaviona:



Olioperustainen ohjelmointi

- Olio on ohjelman perusyksikkö: Ohjelma kuvataan keskenään kommunikoivina olioina.
- Luokat toteutetaan olio-ohjelmointikielellä.
- Oliot luodaan jollakin operaatiolla luokastaan.
- Oliot voidaan yksilöidä esimerkiksi viitteen avulla.
- Viestiä vastaa metodin kutsuminen esimerkiksi pistenotaatiolla.
- Toteutusvaiheeseen pääsy vaatii usein luokkien ja niiden välisten suhteiden mallintamista.
  - Tämä tehdään usein UML:llä.

3. Luokat, oliot ja metodit Java-kielessä

(Lausekielinen ohjelmointi I ja II –kursseilla opitun kertausta.)

Luokat

- Java on olioperustainen ohjelmointikieli: lähes kaikki koodi sijoitetaan luokkiin.
- Luokat tunnistetaan varatusta sanasta **class**.
- Määrittely koostuu luokan otsikosta ja lohkosta:

```
public class LuokanNimi {
    // Attribuutit ja metodit luokan lohkon sisässä.
}
```

- Suorituskelpoisessa Java-luokassa oltava pää-ohjelmametodi *main*.

Oliot

- Oliot ovat viitetyyppisiä (luokkatyyppisiä) muuttujia.
- Tunnuksen esittely varaa muistia vain viitteelle.
- Olion tiedoille varataan muistia **new**-operaatiolla.
- Yleisesti: **LuokanNimi olionNimi;**

  olionNimi = new LuokanNimi();

  tai

  LuokanNimi olionNimi = new LuokanNimi();
- Esim. String nimi = **new** String("Rontti");

Metodit

- Määrittely = otsikko + runko:

```
määreet metodinNimi(parametrilista) {
    // Lauseita.
}
```

  missä määreet koostuvat muun muassa näkyvyysmääreistä (esim. **private** ja **public**) ja metodin tyypistä (esim. **void**, **int** ja **String**).

- Esim. **public static int** laskeKertoma(**int** n) { … }

Metodit

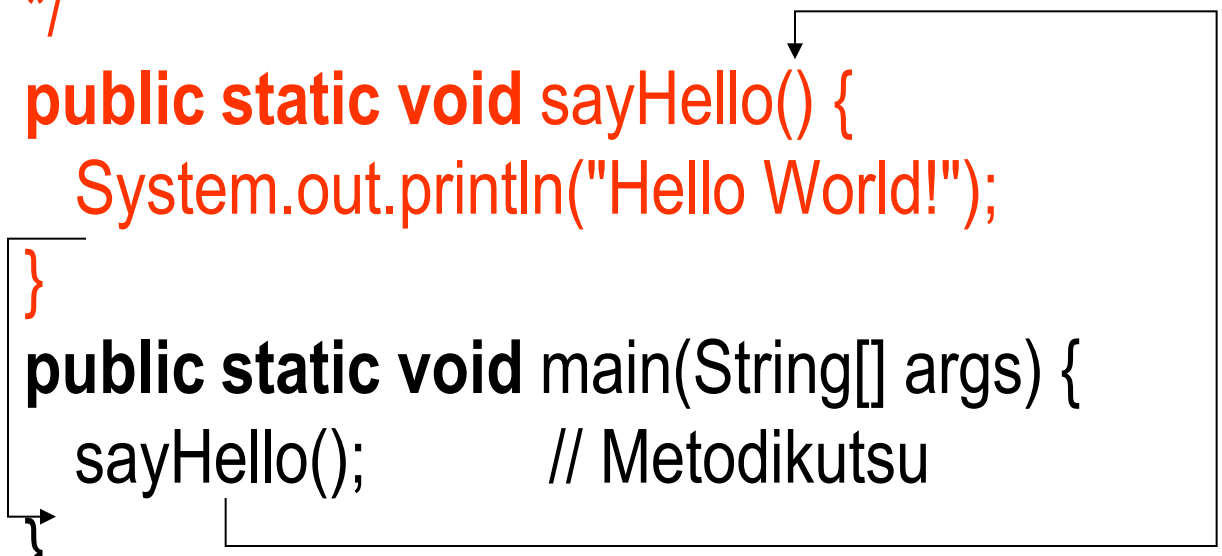
- Mahdollistavat osaltaan modulaarisuuden (modularity), jossa koodi jaetaan helposti hallittaviksi ja hyvin määriteltyjen liittymien kautta kommunikoiviksi osakokonaisuuksiksi.
- Osakokonaisuus: Nimettyyn lohkoon eristetty luokan osa, joka on olio-ohjelmoinnissa metodi.
- Liittymä: Parametrilista, jonka kautta metodille välitetään arvoja (sekä mahdollinen paluuarvo).

Metodit

- Kutsutaan luokan sisällä suoraan nimellä.
- Kutsuminen luokan ulkopuolelta joko olion tai luokan tunnuksen kautta pistenotaation avulla.
- Esim. **int** n = In.readInt(); **if** (nimi.equals("Mörkö")) …
- Lohkon tunnukset eivät näy metodin ulkopuolelle.
- Tyypittömistä (**void**-tyyppisistä) metodeista palataan automaattisesti.
- Tyypitetyt metodit palauttavat paluuarvoja **return**-lauseella.

sayHello-metodi

```
public class HelloWorld {
   /* Tulostaa tervehdyksen koko maailmalle.
   */
   public static void sayHello() {
      System.out.println("Hello World!");
   }
   public static void main(String[] args) {
      sayHello();          // Metodikutsu
   }
}
```



- Huom! Ei käänny ilman metodin **static**-määrettä, koska kutsutaan *main*-metodista.

4. Attribuutit

Sisällys

- Yleistä attribuuteista.
- Näkyvyys luokan sisällä ja ulkopuolelta.
- Attribuuttien arvojen käsittely aksessoreilla.

Yleistä

- Luokan lohkossa, mutta metodien ulkopuolella esiteltyjä muuttujia ja vakioita.
- Esittely tyypillisesti ennen metodeja.
- Esitellään lähes kuten “tavalliset” muuttujat tai vakiot.
- Lisänä näkyvyys-määreitä.

```
public class Kissa {
   // Attribuutit.
   private String vari;
   private String hanta;
   // Metodit.
   public void aantele(String s) {
      System.out.println(s);
   }
}
```

Näkyvyys luokan sisällä

- Metodien tapaan attribuuttien näkyvyyttä voidaan tarkastella luokan sisä- ja ulkopuolelta.
- Attribuutit näkyvät kaikkialla luokan sisällä (globaalisti):
  - Luokassa voi olla vain yksi tietyn niminen attribuutti.
  - Attribuutti on käytettävissä kaikissa luokan metodeissa.
  - Muistuttavat proseduraalisten ohjelmointikielien globaaleja muuttujia.

Näkyvyys luokan sisällä

- Attribuutit rikkovat modulaarisuustavoitetta samoin kuin globaalit muuttujat proseduraalisissa kielissä.
  - Runsaasti attribuutteja sisältävästä koodista on mahdoton nähdä, missä kaikkialla tietoja muutetaan.
  - Parametreilla ja paluuarvoilla tapahtuvan tiedon-välityksen saa korvata attribuuteilla vain hyvästä syystä.
- Vakiomuotoisia attribuutteja voi käyttää vapaammin.
  - Esim. **private final int** HENKIENMAXLKM = 7;

Näkyvyys luokan sisällä

- Jos metodin muuttujalla tai parametrilla on sama tunnus kuin attribuutilla, ei tapahdu nimikonfliktia, vaan metodin tunnus peittää (hide) attribuutin tunnuksen.
- Peitetty tunnus saadaan käyttöön **this**-attribuutilla.
- **this** viittaa olioon itseensä ja on aina käytettävissä automaattisesti ilman esittelyä.
- Peittämiseen harvoin tarvetta.

```
public class Peittodemo {
   // Esitellään ja alustetaan attribuutti.
   private int x = 1;
   // Peitetään ja näytetään attribuutti.
   public void tulosta() {
      // Attribuutin peittävä muuttuja.
      int x = 2;
      // Tulostaa muuttujan arvon (2).
      System.out.println(x);
      // Tulostaa attribuutin arvon (1).
      System.out.println(this.x);
   }
}
```

Näkyvyys luokan ulkopuolelta

- Luokan tiedot pyritään kätkemään, jotta tietojen muuttaminen ympäristöstä käsin ei olisi mahdollista.
- Javassa tämä tehdään **private**-määreellä, jolloin attribuutti on käytettävissä vain luokkansa sisällä.
- Koska vakioiden arvoja ei voida muuttaa, asetetaan vakioidut attribuutit joskus suoraan näkyviin **public**-määreellä.
- Pakkausten ja periytymisen yhteydessä esitellään lisää keinoja näkyvyyden määrittelyyn.

Aksessorit

- Piilotetun datan käsittelyyn tarvitaan niin sanotut aksessorit (accessors), jotka ovat metodeja attribuutin arvojen lukemiseen ja asettamiseen.
  - Vain attribuuteille, joita on tarpeen käsitellä luokan ulkopuolelta.
- Mahdollisimman lyhyitä metodeja, joissa ei tehdä mitään turhaa.
- Ei tarvitse kommentoida ellei jotain erityistä.

Aksessorit

- Lukumetodi:
  - Rungossa ainoastaan attribuutin arvon palauttava **return**-lause.
  - Tyyppi tavallisesti sama kuin attribuutin tyyppi.
- Asetusmetodi:
  - Attribuutin arvo välitetään parametrina.
  - Arvoa ei siis koskaan lueta metodissa.
  - Tulisi tarkistaa tiedon oikeellisuus. (Esimerkiksi arvoalueelle kuulumisen tai muistinvarauksen tarkistus.)
  - Virheellinen tieto voidaan jättää huomiotta, jolloin metodi on tyypitön (**void**).
  - Virheestä voidaan ilmoittaa myös paluuarvolla, mutta tämä on harvinaista.

Aksessorit

- Aksessoreita kutsutaan usein settereiksi ja gettereiksi, koska englannin kielessä aksessorien nimet aloitetaan *set*- ja *get*-sanoilla.
- Esim. Person pete = **new** Person();
  pete.setAge(10); **int** age = pete.getAge();
- Suomeksi kirjoitetussa koodissa voi käyttää esimerkiksi vastaavia *aseta*- ja *lue*-sanoja.
  - Nimen voi myös johtaa attribuutin tunnuksesta kuormittaen.
- Esim. Henkilo pete = **new** Henkilo();
  pete.asetalka(10); **int** ika = pete.luelka();
  pete.ika(10); **int** ika = pete.ika(); // kuorimittaen

Kissa-luokka (Kissa.java)

```
public class Kissa {
  /* Attribuutit ***************/
  private String vari;
  private String hanta;

  /* Aksessorit ***************/
  // Värin lukeminen.
  public String vari() {
    return vari;
  }
  // Värin asetus.
  public void vari(String v) {
    if (v != null)
      vari = v;
  }
  // Hännän tyypin lukeminen.
  public String hanta() {
    return hanta;
  }
  // Hännän tyypin asetus.
  public void hanta(String h) {
    if (h != null)
      hanta = h;
  }
  ...
}
```

- Huom! Metodeissa ei tarvita **static**-määrettä, koska luokassa ei ole *main*-metodia.

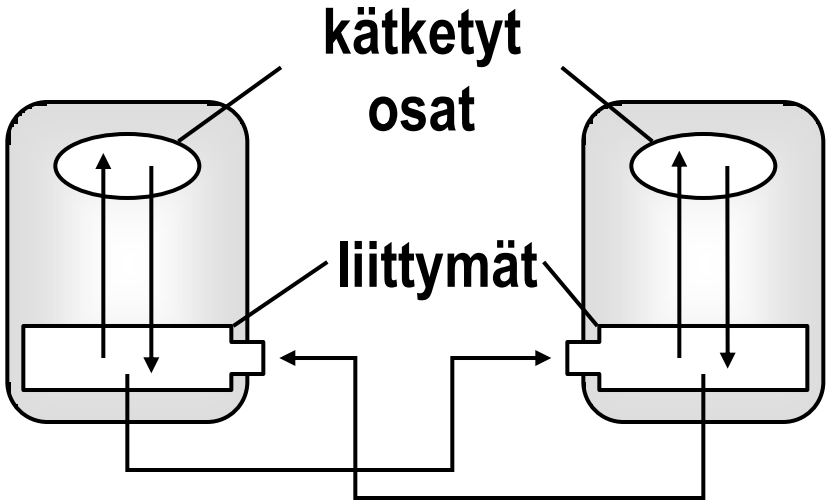
5. Kapselointi

Yleistä

- Kapseloinnilla (encapsulation) tarkoitetaan luokan tietojen ja toimintojen pakkaamista yhdeksi suojatuksi kokonaisuudeksi piirteitä piilottamalla.
- Attribuutit piilotetaan **private**-määreen ja aksessoreiden avulla. (Tiedonkätkentä on siis osa kapselointia.)
- Metodeja piilotetaan **private**-määrettä käyttäen.
- Piilotettavia ovat esimerkiksi metodit, joita tarvitaan vain luokan sisällä.

Liittymä

- Julkiset metodit muodostavat liittymän, jonka kautta luokkaa käytetään kontrolloidusti ohjelmoijan ajattelemalla tavalla.
- Kapseloitujen luokkien kommunikointi:



Kissa-luokka (Kissa.java)

```
public class Kissa {
  // Attribuutit.
  private String vari; private String hanta;
  // Metodit
  private void hairikoi() {
    // Tavaroita hajoaa...
  }
  public String vari() { return vari; }
  public void vari(String v) { if (v != null) vari = v; }
  public String hanta() { return hanta; }
  public void hanta(String h) { if (h != null) hanta = h; }
  public void aantele(String s) { System.out.println(s); }
}
```

Luokan kätketty osa

Luokan liittymä

Luokalle oma tiedosto

- Ohjelma on toistaiseksi ajateltu yhdeksi luokaksi.
- Näin lähdekooditiedostojakin on ollut vain yksi.
- Siirrytään nyt tyypilliseen käytäntöön, jossa jokaisen luokan koodi erotetaan omaan tiedostoonsa.
- Näin toimien kukin luokka on selkeämmin täysin oma kokonaisuutensa myös tiedostojen tasolla.
- *main*-metodin sisältävää luokkaa kutsutaan ajoluokaksi.
  - Käytetään tällä kurssilla usein toisen luokan testaamiseen.
  - Esim. *Kissa*-luokkaa testataan *KissaTesti*-luokassa.

KissaTesti-luokka (KissaTesti.java)

```
public class KissaTesti {
   public static void main(String[] args) {
      // Luodaan kissa.
      Kissa rontti = new Kissa();
      // Testataan metodeja.
      rontti.aantele("Miau!");
      rontti.vari("musta");
      rontti.hanta("tavallinen");
      System.out.println(rontti.vari());
      System.out.println(rontti.hanta());
   }
}
```

Testiluokan kääntäminen ja ajaminen

- Testiluokka on käännettävä yhdessä testattavan luokan kanssa. Tämä on tehtävissä eri tavoin.
- Kun molempien luokkien lähdekooditiedostot sijoitetaan samaan hakemistoon, kääntäjälle tarvitsee antaa vain testiluokan sisältävän tiedoston nimi. Esimerkki:

  javac KissaTesti.java
- Ohjelma ajetaan testiluokan nimellä. Esimerkki:

  java KissaTesti

Testiluokan kääntäminen ja ajaminen

- Luokan voi myös sisällyttää testiluokan käännökseen jostakin muusta hakemistosta joko polkumäärittelyllä tai *javac*-ohjelman *sourcepath*-parametrillä.
- Näin käännetty ohjelma suoritetaan siten, että *java*-ohjelmalle kerrotaan *classpath*-parametrin avulla tavukoodin hakemisto.
- Ajaminen ja käännös tehdään siis kuten *In*-luokan yhteydessä.
  - Kertaa tarvittaessa Lausekielinen ohjelmointi I -kurssin luentomateriaalin 8. luku.

6. Metodit

Sisällys

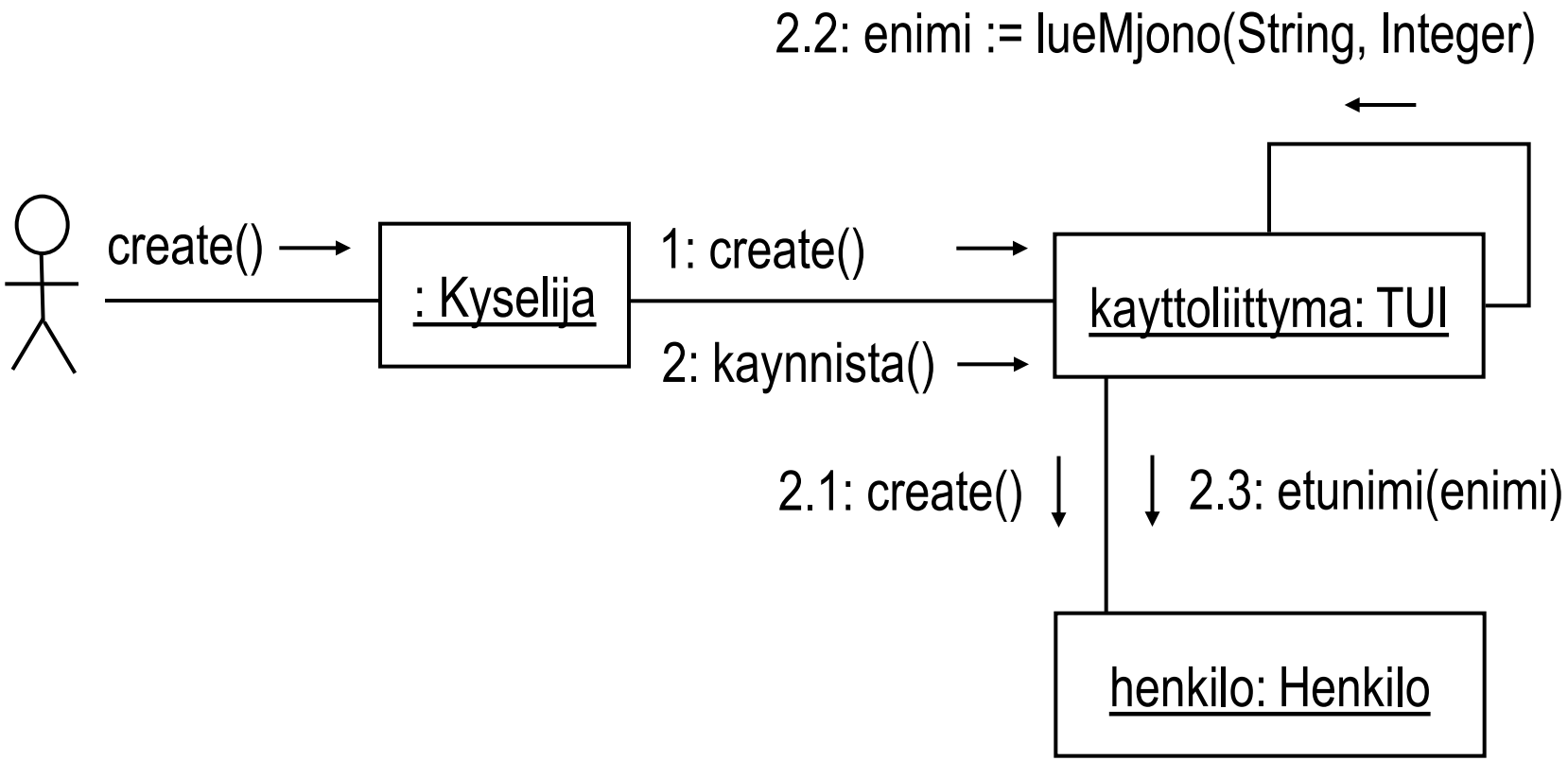
- Oliot viestivät metodeja kutsuen.
- Kuormittaminen.
- Luokkametodit (ja -attribuutit).
- Rakentajat.
- Metodien ja muun luokan sisällön järjestäminen.

Oliot viestivät metodeja kutsuen

- Olio-ohjelmoinnissa ohjelma on joukko olioita, jotka suorittavat annetun tehtävän keskenään viestimällä.
- Viestit ovat käytännössä metodikutsuja.
- Oletetaan, että tehtävänä on kirjoittaa ohjelma, joka lukee käyttäjältä tietoja.
- Ohjelmassa on ajoluokka (*Kyselija*), tekstipohjainen käyttöliittymä (*TUI*) ja *Henkilo*-luokka, jonka tietoihin kuuluu muun muassa etunimi.
  - Luokille määritelty vastuut: ajoluokka käynnistää, käyttöliittymä vuorovaikuttaa ja henkilöluokka mallintaa.
- Luokkien lähdekoodi löytyy kurssisivuilta.

Oliot viestivät metodeja kutsuen

- Henkilön tietojen kysely tapahtuu olioiden yhteistyönä:



Kuormittaminen

- Kuormittamisena (overloading) tunnettu mekanismi sallii samannimisten metodien esittelyn.
- Koska metodin tunnus voi tarkoittaa useampia asioita, on kuormittaminen eräs monimuotoisuuden (polymorphism) muodoista.
- Parametrilista erottaa kuormitetut metodit toisistaan.
- Metodeja ei voi kuormittaa antamalla erilaisia näkyvyysmääreitä tai tyyppejä.
- Käytetään erityisesti rakentajien yhteydessä.

Kuormittaminen

- Rakentajien lisäksi myös tavallisia metodeja voidaan kuormittaa:

```
// String-parametri.
public void hauku(String s) {
   System.out.println(s);
}
// String- ja int-parametrit.
public void hauku(String s, int k) {
   for (int i = 0; i < k; i++)
      System.out.println(s);
}
```

- Kätevää myös aksessorien yhteydessä:

```
// Arvon lukeminen.
public int luonne() {
   return luonne;
}
// Arvon asettaminen.
public void luonne(int l) {
   if (l > 0)
      luonne = l;
}
```

Luokkametodit (ja -attribuutit)

- Luokkametodit (class method) ovat kutsuttavissa suoraan luokan nimen kautta ilman olion luomista.
- Yleisesti Javassa:

  LuokanNimi.metodinNimi(parametrilista);

- Esim.

```
// In-luokan kautta. Tyhjä parametrilista.
double d = In.readDouble();
// Math-luokan kautta. Kaksi parametria.
int max = Math.max(1, 2);
```

Luokkametodit (ja -attribuutit)

- **static**-määreellä esiteltyjä metodeja ja attribuutteja.
- Esim. *Math*-luokan *max*-metodi:

  public static int max(**int** a, **int** b) { ... }

- **static**-määre hyödyllinen varsinkin, kun
  - kootaan yhteen toimintoja, joiden käyttö sujuvampaa suoraan ilman olion esittelyä,
  - määritellään vakioita, joiden käyttö sujuvampaa suoraan ilman olion esittelyä ja
  - on tarpeen määritellä muuttuja, jonka arvo on ajonaikana sama kaikille luokan olioille.

Luokkametodit (ja -attribuutit)

- Luokkametodeissa voidaan käsitellä vain luokka-attribuutteja ja kutsua luokkametodeja, koska luokkapiirteet ovat saatavilla ilman olioiden luomista. Tästä syystä:
  - *main*-metodin sisältävässä luokassa piirteiden esittelyihin on sisällytettävä **static**-määre ja
  - **this**-attribuuttia ei voi käyttää luokkametodeissa.
- **static**-määrettä ei tarvita muissa luokissa kuin *main*-metodin luokassa ellei olla tarkoituksella määrittelemässä luokkametodia tai -attribuuttia.

Luokkametodit (ja -attribuutit)

- Tavallisia – ilman **static**-määrettä esiteltyjä – metodeja sanotaan joskus olio- tai ilmentymä-metodeiksi (instance method), koska nämä metodit liittyvät nimenomaan luokan olioihin (ilmentymiin).
- Samoin tavallisia attribuutteja voidaan kutsua olio- tai ilmentymäattribuuteiksi (instance variable).
- Oliometodeja ei voi kutsua luokan nimen kautta, mutta luokkametodeja voidaan kutsua myös olion kautta.

Rakentajat

- Rakentaja (constructor) on metodi, jota kutsutaan aina, kun luodaan olio **new**-operaatiolla.
- Rakentajan nimi on aina sama kuin luokan nimi.
- Java tarjoaa automaattisesti kaikille luokille parametrittoman oletusrakentajan.
- Esim. Voimme luoda uuden *Kissa*-olion lauseella

  Kissa misse = **new** Kissa();

  vaikka luokkaan ei ole erikseen kirjoitettu rakentajametodia *Kissa*().

Rakentajat

- Luokalla voi olla myös parametrillisia rakentajia, koska myös rakentajia voidaan kuormittaa.
  - Parametrilliset rakentajat ohjelmoitava itse.
  - Mikäli luokalle toteutetaan parametrillinen rakentaja, Java ei enää luo automaattisesti oletusrakentajaa.
- Esim. Integer k = **new** Integer(13);

  Kissa rontti = **new** Kissa("musta", "tavallinen");

  Kissa misse = **new** Kissa();

Rakentajat

- // Parametriton oletusrakentaja.

  public LuokanNimi() {

  ...

  }

- // Parametrillinen rakentaja.

  public LuokanNimi(parametrilista) {

  ...

  }

- Rakentajalla ei ole minkäänlaista tyyppimäärittelyä: otsikkoon ei saa lisätä edes **void**-määrettä.
- Rakentajat ovat usein julkisia (**public**).

Rakentajat

- Rakentajat on tarkoitettu erityisesti attribuuttien arvojen alustamiseen.
- Rakentajissa ei tehdä mitään ylimääräistä.
  - Esimerkiksi ei kysellä alkuarvoja käyttäjältä.
- Attribuuttien alustaminen onnistuu myös esittelyn yhteydessä, mutta rakentaja tarjoaa metodina mahdollisuuden monimutkaisempiin alustustoimiin.

Rakentajat

- Esim. *Kissa*-luokan attribuuttien alustaminen:

```
...
// Alustaminen esittelyn yhteydessä.
private String vari = "musta";
private String hanta = "kippura";
…
```

tai

```
...
// Alustaminen oletusrakentajassa.
public Kissa() {
   vari = "musta";
   hanta = "kippura";
}
...
```

Rakentajat

- Parametrillisilla rakentajilla voidaan antaa attribuuteille helposti oliokohtaiset alkuarvot.
  - Asetusmetodien tapaan myös rakentajissa on syytä tarkistaa parametriarvojen järkevyys.
- **this**-attribuuttia tarvitaan peittymisen vuoksi, jos parametri nimetään attribuutin mukaan.
- Esim.

```
private String vari;
public Koira(String vari) {
    if (vari != null)
        this.vari = vari;  // attribuutti = parametri
}
```

Rakentajat

- Alkuarvojen asettaminen onnistuu ilman parametrillista rakentajaa aksessoreilla, mutta rakentajien käyttö
  - on kätevämpää ja
  - rakentajissa kannattaa suorittaa keskitetysti olion luomisen yhteydessä tarvittavat toimenpiteet.
- Huom! Koodin määrää voi vähentää edelleen hyödyntämällä aksessoreita rakentajissa.

Kissa-luokka (Kissa.java)

```
public class Kissa {
   // Attribuutit.
   private String vari;
   private String hanta;
   /* Oletusrakentaja.
    */
   public Kissa() {
      vari = "musta";
      hanta = "kippura";
   }
   /* Parametrillinen rakentaja,
    * jossa alustetaan vari (v)
    * ja hanta (h).
    */
   public Kissa(String v, String h) {
      // Koodia lyhennetty
      // kutsumalla aksessoreita.
      vari(v);
      hanta(h);
   }
   ...
}
```

KissaTesti-luokka (KissaTesti.java)

```
class KissaTesti {
   public static void main(String[] args) {
      // Luodaan kissa parametrittomalla oletusrakentajalla.
      Kissa rontti = new Kissa();
      System.out.println(rontti.vari());          // musta
      System.out.println(rontti.hanta());         // kippura
      // Luodaan kissa parametrillisella rakentajalla.
      Kissa moykky = new Kissa("valkea", "tavallinen");
      System.out.println(moykky.vari());          // valkea
      System.out.println(moykky.hanta());         // tavallinen
   }
}
```

Luokan sisällön järjestäminen

- Yleensä luokan sisältö järjestetään jollakin tavoin, koska halutaan tehdä yhdenmukaista koodia.
- Attribuutit yleensä ennen metodeja.
  - Olio-ohjelmoinnissa tiedot keskiössä.
- Esimerkkijärjestys:
  - Vakiomuotoiset attribuutit.
  - Luokka-attribuutit.
  - Attribuutit (olioattribuutit).
  - Rakentajat.
  - Luokkametodit.
  - Metodit (oliometodit).
  - *main*-metodi.

Luokan sisällön järjestäminen

- Rakentajat aina ennen muita metodeja.
- Muut metodit:
  - Java-kielen kehittäjä Sun suositteli järjestämään toiminnallisuuden mukaan: yhteenkuuluvat osat koodia lähellä toisiaan.
  - Näkyvyyden mukaan ryhmitellen.
  - Aakkosjärjestys.
- Yrityksillä omat yksityiskohtaiset tyylioppaansa.
- Oli järjestys mikä tahansa, tärkeintä on johdonmukaisuus!

7. Oliot ja viitteet

Sisällys

- Olio Java-kielessä.
- Olion luominen, elinikä ja tuhoutuminen.
- Viitteiden sijoitus.
- Viitteiden vertailu.
  - Varautuminen **null**-arvoon.
- Viite metodin paluuarvona.
- Viite metodin parametrina.
  - Muuttumattomat ja muuttuvat merkkijonot.

Olio Java-kielessä

- Java-kielessä olio määritellään joko luokan edustajaksi tai taulukoksi.
- Olio on joukko keskusmuistissa olevia tietoja.
- Oliota käsitellään siihen epäsuorasti liittyvän viitetyyppisen muuttujan eli viitteen (reference) avulla.
- Viite määrittää olion identiteetin.
- Viite toteutetaan teknisesti yhden tai useamman suojatun osoittimen (pointer) avulla.
  - Javan viitteet eri asia kuin osoittimet ja C++:n viitteet.

Olion luominen

- Tunnuksen esittely varaa muistia viitteelle (viitetyyppiselle muuttujalle), mutta ei oliolle.
- Java alustaa automaattisesti viitteen tyhjäksi (**null**-arvo) tai ilmoittaa, että viite on alustettava.
- Oliolle varataan muistia **new**-operaatiolla, joka palauttaa viitteen.
- Viitetyyppinen muuttuja ja olio liittyvät toisiinsa, kun **new**-operaation paluuarvona saatava viite sijoitetaan muuttujan arvoksi.

Olion luominen

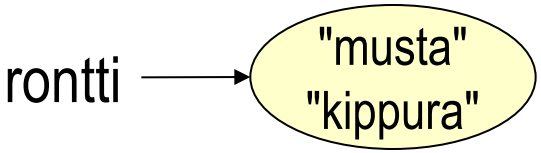
- Kissa rontti = **null**;

rontti ⟶ **null**

Olion viitteen esittely ja alustaminen tyhjäksi.

- rontti = **new** Kissa();

Lauseke **new** Kissa(); luo olion, alustaa sen attribuutit rakentajassa ja palauttaa paluuarvona olioon liittyvän tunnuksettoman viitteen.



Sijoituksen seurauksena *rontti*-viite viittaa samaan olioon kuin paluuarvona saatu viite.

Olion elinikä

- Olio on olemassa (“elää”) niin kauan kun siihen liittyy yksi tai useampia viitteitä.
- Viitetyyppiset muuttujat “elävät” alkeistyyppisten muuttujien tapaan vain esittelylohkossaan:
  - Esimerkiksi metodissa esitelty viite tuhoutuu metodista poistuttaessa, koska sen tunnus on paikallinen.
- Olion elinikä voi kuitenkin poiketa olioon muistinvarauksen yhteydessä asetetun viitteen eliniästä, koska olioon voidaan liittää myös muissakin lohkoissa kuin esittelylohkossa “eläviä” viitteitä.

Olion tuhoutuminen

- Olio tuhoutuu, kun siihen ei liity viitteitä.
- Viitteettömien olioiden poistaminen hoidetaan automaattisesti Javan virtuaalikoneen toimesta.
- Toimenpidettä kutsutaan roskankeruuksi (garbage collection).
- Virtuaalikone kerää roskia automaattisesti, kun tarvitaan lisää muistitilaa.
- Java-ohjelmoijan ei siis tarvitse huolehtia olioiden poistamisesta ohjelmallisesti.

Olion tuhoutuminen

- Javan virtuaalikonetta voi myös käskeä tuhoamaan tarpeettomat oliot *System*-luokan *gc*-metodilla.
- Monissa kielissä varattu muisti vapautetaan ohjelmoijan toimesta erityisellä operaatiolla.
- Operaattorin käyttö on tehokkaampaa kuin automaattinen muistin vapautus, mutta altistaa myös muistivuodoille (memory leak).

Viitteiden sijoitus

- Viitetyyppiseen muuttujaan *x* voidaan sijoittaa toinen viitetyyppinen muuttuja *y* lauseella:

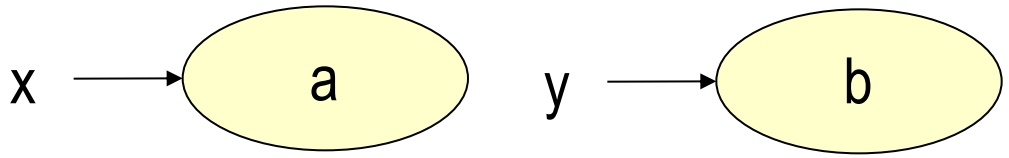
  x = *y*;

  missä muuttujien *x* ja *y* tyyppi (luokka) on sopiva.
- Sijoitus “irrottaa” viitteen *x* siihen mahdollisesti liittyvästä oliosta ja “kiinnittää” viitteen *y*:n olioon.
- Sijoituksen jälkeen muuttuja *x* viittaa muuttujan *y* viittaamaan olioon.
- Sijoitus ei kopioi *y*:n viittaaman olion tietoja *x*:n viittaaman olion tietojen päälle!
  - Kopiointiin tähän tarvitaan oma metodi.

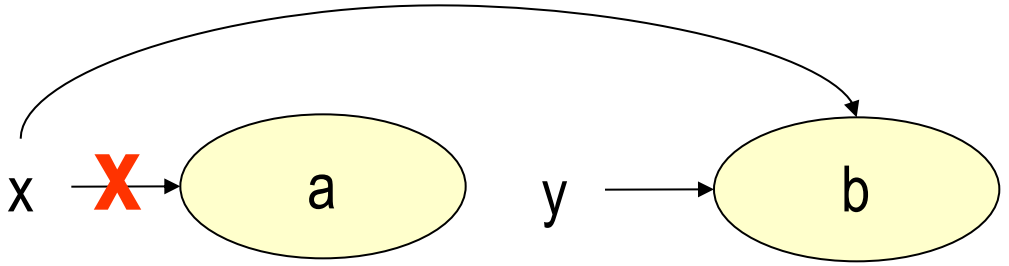
Viitteiden sijoitus

- Esimerkki. Osoittakoon viite *x* olioon *a* ja viite y olioon *b*, kun viitteiden tyyppi oletetaan samaksi.

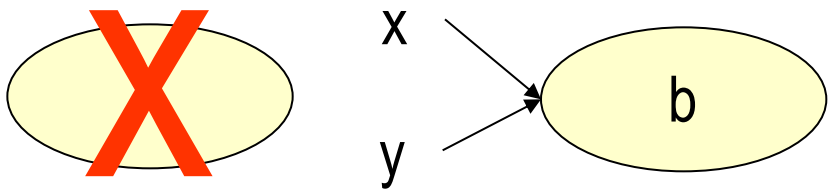
- Alkutilanne



- Sijoitus *x* = *y*;



- Lopputilanne



Viitteiden sijoitus

- Sijoituksen kautta yhteen olioon voi viitata useampia tunnuksia.
- Olion identiteetti säilyy, vaikka olioon olisi useampia viitteitä, koska viitteet liittyvät samaan olioon.
- Sijoituksen sivuvaikutuksena olioon ei välttämättä enää ole viitteitä, jolloin olio jää roskankerääjän saaliiksi.
- Olion viitteet voidaan myös poistaa asettamalla ne tyhjiksi eli **null**-arvoisiksi.

Viitteiden sijoitus

```
public class ViiteTesti {
   public static void main(String[] args) {
      // Esitellään viitteet ja varataan muistia.
      String x = new String("a");
      String y = new String("b");
      // Kiinnitetään viite x viitteen y osoittamaan olioon.
      x = y;
      System.out.println(x + " " + y);  // b b
      // Irrotetaan viitteet b-oliosta.
      x = y = null;
      System.out.println(x + " " + y);  // null null
   }
}
```

Viitteiden sijoitus

- Taulukoiden käsittelyssä on huomattava, että sijoitus ei riitä taulukon alkioiden kopiointiin.
- Taulukon sisältö on kopioitava toiseen taulukkoon joko silmukoilla tai *Arrays*-luokan metodeilla.
  - Viitetyyppiset alkiot voi olla tarpeen syväkopioida.

```
// Esitellään viite, luodaan olio
// ja liitetään viite olioon.
int[] luvut1 = { 1, 1 };
int[] luvut2 = { 2, 3, 4 };
// Sijoitetaan viite.
luvut1 = luvut2;
// Molemmat viitteet liittyvät
// jälkimmäiseen olioon.
luvut1[0] = 6;
System.out.println(luvut1[0]); // 6
System.out.println(luvut2[0]); // 6
```

Viitteiden vertailu

- Viitetyyppisiä muuttujia voidaan myös vertailla yhtä- ja erisuuruus-operaattoreilla (== ja !=).
- Vertailu kohdistuu viitteisiin eikä viitattuihin olioihin.
- Olioiden sisällön vertailuun pitää kirjoittaa erillinen metodi.
  - *Equals*- ja *compareTo*.

```
// Esitellään viitteet ja varataan muistia.
String eka1 = new String("eka");
String eka2 = new String("eka");
// Vertaillaan viitteitä. Tulostuu false,
// koska viitteet liittyvät eri olioihin.
System.out.println(eka1 == eka2);
// Vertaillaan olioita. Tulostuu true,
// koska olioiden tiedot ovat samat.
System.out.println(eka1.equals(eka2));
```

Null-arvoisiin viitteisiin varautuminen

- Metodin kutsu **null**-arvoisen tyhjän viitteen kautta aiheuttaa ajonaikaisen virheen (poikkeuksen), joka pysäyttää ohjelman suorituksen.
- Metodissa voidaan välttää virhe **if**-lauseella:

```
if (parametrinaSaatuViite != null)  {
    // Viite liittyy olioon, sitä voidaan käsitellä.
}
```

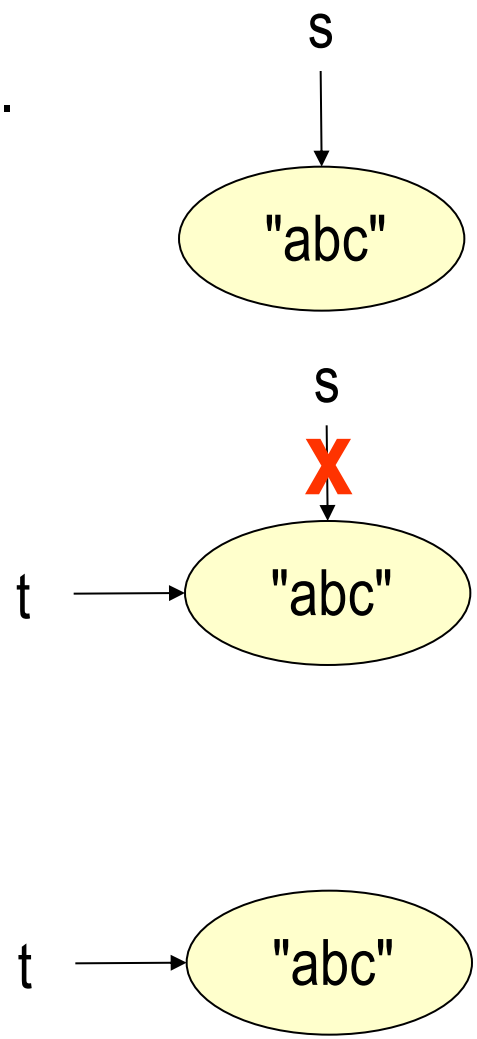
- Vaihtoehtoisesti voidaan käyttää **try-catch**-lausetta.
- Tyhjää viitettä ei huomioida, jos voidaan olettaa, että tyhjän viitteen aiheuttama poikkeus käsitellään muualla.

Viite metodin paluuarvona

- Metodin tunnukset ovat paikallisia (lokaaleja) - ne ovat olemassa vain metodin suorituksen ajan.
- Jos metodissa luotu olio halutaan säilyttää, voidaan sen viite palauttaa metodin paluuarvona ja paluuarvo sijoittaa muuttujaan kutsuvassa metodissa.
  - Olio ei tuhoudu, vaikka paikallinen viite tuhoutuu, koska olioon asetetaan toisen lohkon viite.
  - Näin toimien paikallisesti esitelty olio “elää” alkuperäistä viitettään vanhemmaksi.

Viite metodin paluuarvona

```
public class ViitePaluuarvona {
   /* Palautetaan viite luotuun merkkijono-olioon.
    */
   public static String palauta() {
      String s = new String("abc");
      return s;
   }
   public static void main(String[] args) {
      // Olio säilyy metodista poistuttaessa,
      // koska olioon liitetään uusi viite t.
      String t = palauta();
      System.out.println(t);   // abc
   }
}
```



Viite metodin parametrina

- Viitetyyppisiä muuttujia voidaan antaa metodeille parametriksi aivan kuten alkeistyyppisiä muuttujia.
  - Esimerkiksi.

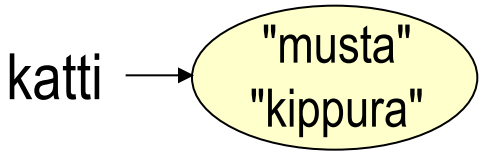
```
int kertoma(int luku) { ... }
void nimi(String nimi) { ... }
double keskiarvo(double[] luvut) { ... }
```

- Java sijoittaa sekä alkeis- että viitetyyppisen parametrin arvoksi alkuperäisen arvon kopion (pass-by-value).
  - Parametrin arvoon tehty muutos katoaa, kun metodista poistutaan.
- Parametriin liittyvän olion tilaan tehty muutos jää voimaan!
  - Olio ei kopioidu, jolloin viitetyyppinen parametri viittaa samaan olioon kuin metodikutsussa annettu viite.

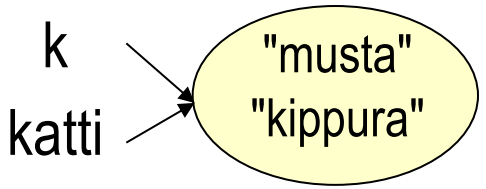
Viite metodin parametrina

```
public class ViiteParametrina {
   /* Olion tietoja muuttava metodi.
    */
   public static void leikita(Kissa k) {
      if (k != null)
         k.hanta("levoton");
   }
   public static void main(String[] args) {
      Kissa katti = new Kissa();   // musta, kippura
      // Muutetaan olion tietoja.
      leikita(katti);
      System.out.println(katti.vari());     // musta
      System.out.println(katti.hanta());  // levoton
   }
}
```

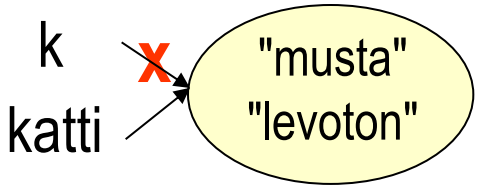
Olio luotu



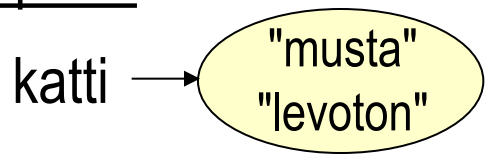
Metodiin tultaessa



Metodista poistuttaessa



Lopuksi



Viite metodin parametrina

```
public class MerkkijonoParametrina {
  /* Olion tiedot muuttuvat.
   */
  public static void muuta(StringBuffer s) {
    s.append("d");
  }
  /* Olion tiedot eivät muutu.
   */
  public static void muuta(String s) {
    s = s + "d";
  }
  public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("abc");
    String s = new String("abc");
    muuta(sb);
    System.out.println(sb);  // abcd
    muuta(s);
    System.out.println(s);   // abc
  }
}
```

- *String*-luokka on toteutettu siten, että *String*-tyyppisten muuttujien tietoihin ei voida tehdä muutoksia.
  - Lisäysoperaattori muodostaa operadeistaan uuden olion.
- Muutetun merkkijonon palauttaminen onnistuu paluuarvona tai *StringBuffer*- ja *StringBuilder*-luokkia käyttäen.

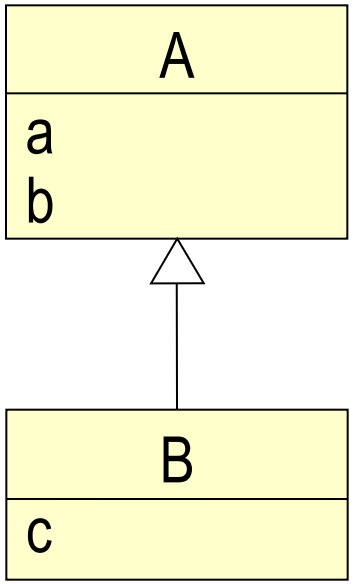
8. Periytyminen

Sisällys

- Mitä on periytyminen?
- Yksittäis- ja moniperiytyminen.
- Oliot ja perityt luokat.
- Periytymisen käyttö.

Mitä on periytyminen?

- Periytyminen (inheritance) tarkoittaa luokan *A* piirteiden siirtymistä jollekin toiselle luokalle *B*.
- Tällöin sanotaan, että *A* on *B*:n yliluokka (superclass) ja vastaavasti *B* on *A*:n aliluokka (subclass).



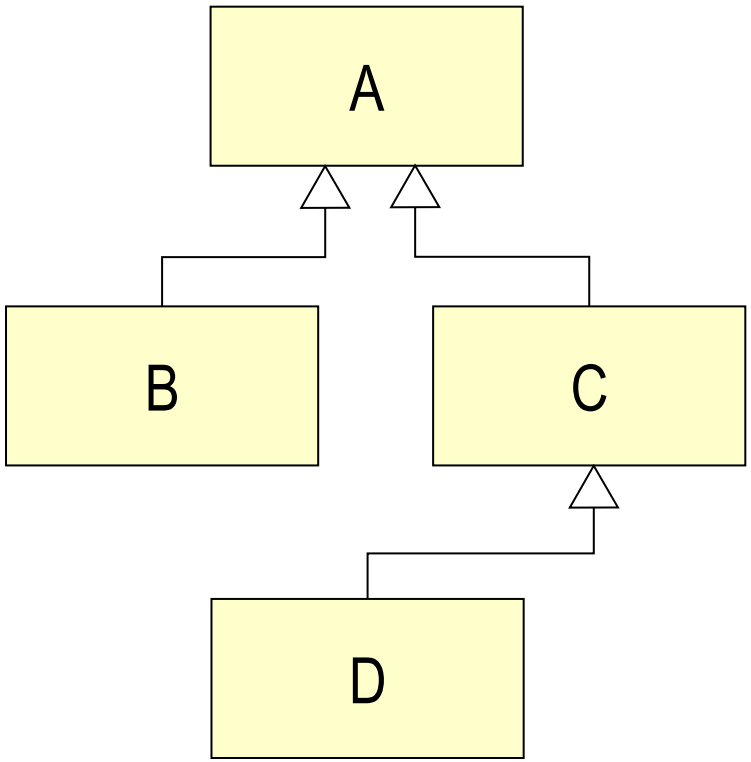
Piirteet: a, b

Piirteet: a, b, c

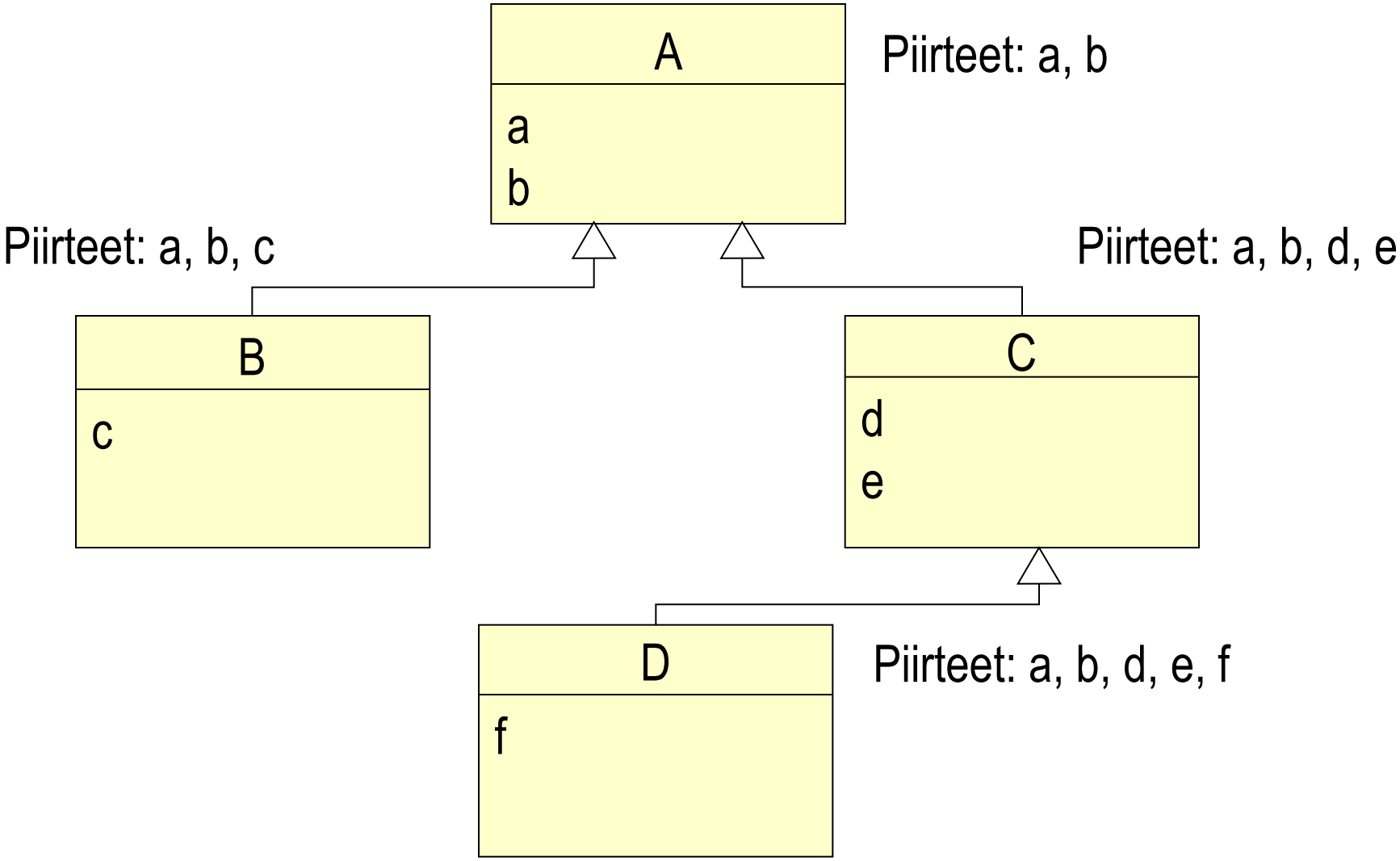
- UML:n luokkakaaviossa luokkaa symboloi laatikko, jossa annetaan luokan nimi ja usein myös luokan piirteet. (Attribuutit ja metodit erotetaan myöhemmin väliviivalla, kun luokkasymboleita ryhdytään piirtämään hieman tarkemmin.) Periytymissuhde piirretään kolmiokärkisenä yliluokkaan osoittavana nuolena.

Mitä on periytyminen?

- Yliluokalla on yksi tai useampia aliluokkia.
- Aliluokka tuntee yliluokkansa, mutta ei aliluokkiaan.
- Luokka *A* on luokan *B* esi-isä (ancestor) jos *A* on *B*:n yliluokka tai *B*:n yliluokan esi-isä.
- Luokka *B* on luokan *A* jälkeläinen (descendant), jos *B* on *A*:n aliluokka tai *A*:n jonkin aliluokan jälkeläinen.

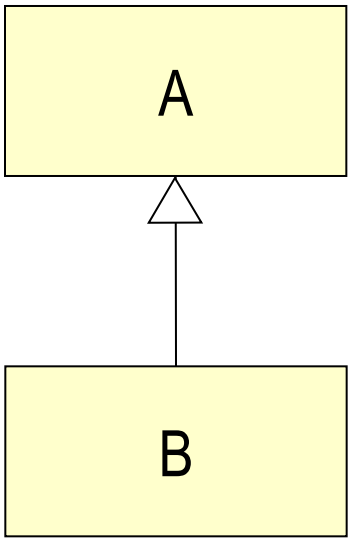


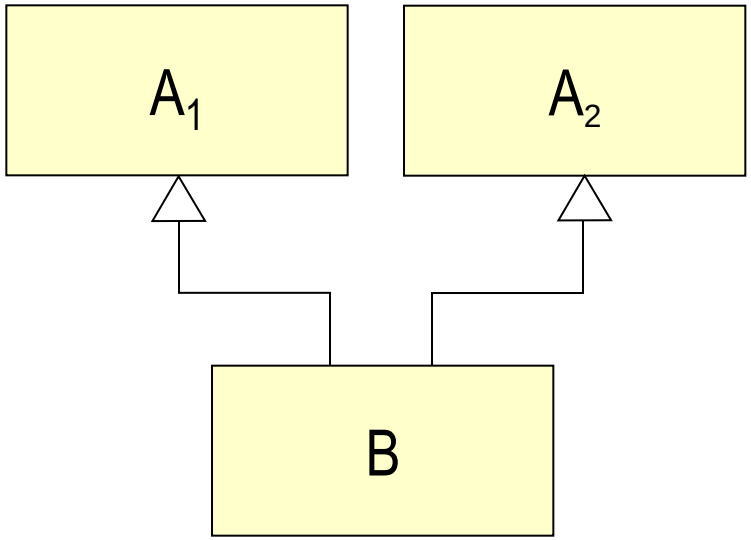
Mitä on periytyminen?



Yksittäis- ja moniperiytyminen

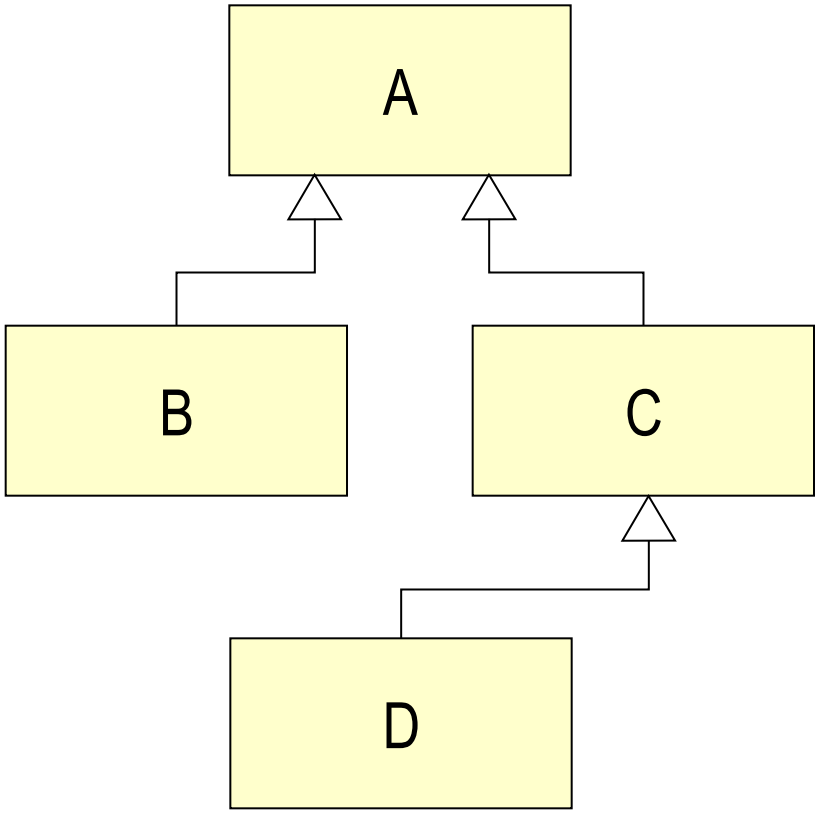
- Yksittäisperiytymisessä (single inheritance) jokaisella aliluokalla on yksi yliluokka.
- Moniperiytymisessä (multiple inheritance) aliluokalla voi olla useampia yliluokkia.





Yksittäis- ja moniperiytyminen

- Yksittäisperiytyminen tuottaa selkeän puumaisen rakenteen.
- Moniperiytyminen on kuvattavissa vaikeampiselkoisena verkkona.
- Molemmin tavoin saadaan aikaiseksi luokkahierarkia, jolla on juuri (root) tai juuria.

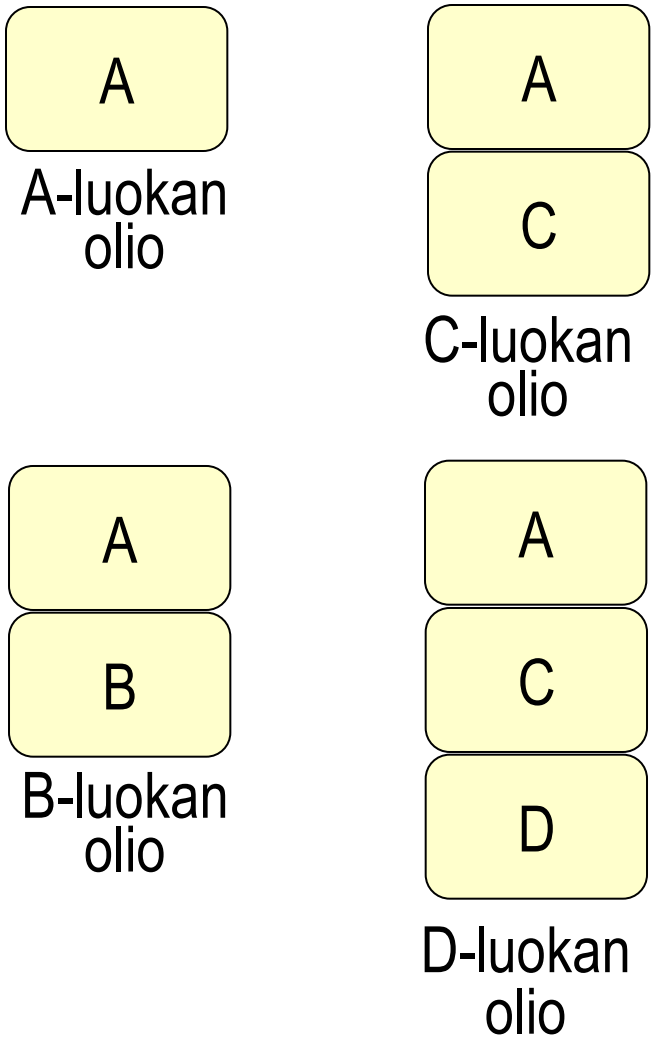


Yksittäis- ja moniperiytyminen

- Moniperiytymisestä sekä hyötyä että haittaa.
- Joissakin kielissä, Java mukaan lukien, on vain yksittäisperiytyminen.
- Tämä Javan ominaisuus voidaan kiertää osin rajapintojen avulla.
- C++ on esimerkki kielestä, jossa luokalla voi olla useita yliluokkia.
- Jatkossa periytymisen esittely rajoitetaan vain yksittäisperiytymiseen.

Oliot ja perityt luokat

- Periytymisen kautta oliolla voi olla useampia luokkia.
- Jatkossa olion perusluokaksi sanotaan luokkaa, josta olio on luotu.
- Kukin yliluokka lisää olioon uuden kerroksen piirteitä (attribuutteja ja/tai metodeja).



Periytymisen käyttö

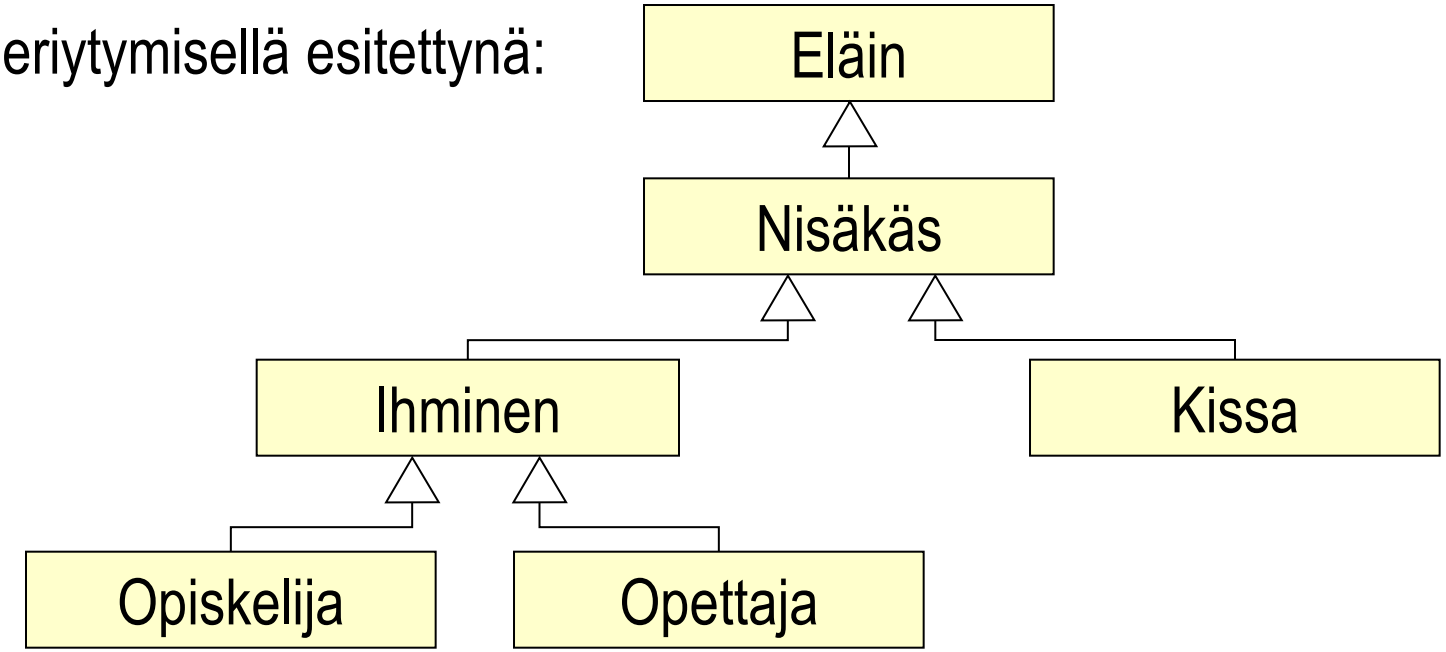
- Pragmaattinen näkökulma:
  - Periytyminen tukee koodin uudelleenkäyttöä: aliluokkien ei tarvitse toistaa perittyjä piirteitä.
  - Periytymisen avulla voidaan kehittää ohjelmistoja tarkentaen (inkrementaalisesti): aliluokkia voidaan lisätä ilman, että yliluokkia tarvitsee muuttaa.
  - Periytymisen avulla voidaan toteuttaa monimuotoisuutta.

Periytymisen käyttö

- Filosofinen näkökulma:
  - Periytyminen tukee käsitteelliseen mallintamiseen perustuvaa ohjelmistokehitystä: aliluokka erikoistus ja yliluokka yleistys.
  - Näin ollen periytymisellä voidaan mallintaa käsitehierarkioita.
  - Periytymistä kutsutaankin joskus yleistämiseksi (generalization), koska periytyminen liittyy läheisesti käsitteellisestä mallintamisesta tuttuun IsA-suhteeseen.
    - Periytyminen on luontevaa, jos lause "*Aliluokka* IsA *Yliluokka.*" tuntuu järkevältä.

Periytymisen käyttö

- Käsitehierarkia periytymisellä esitettynä:

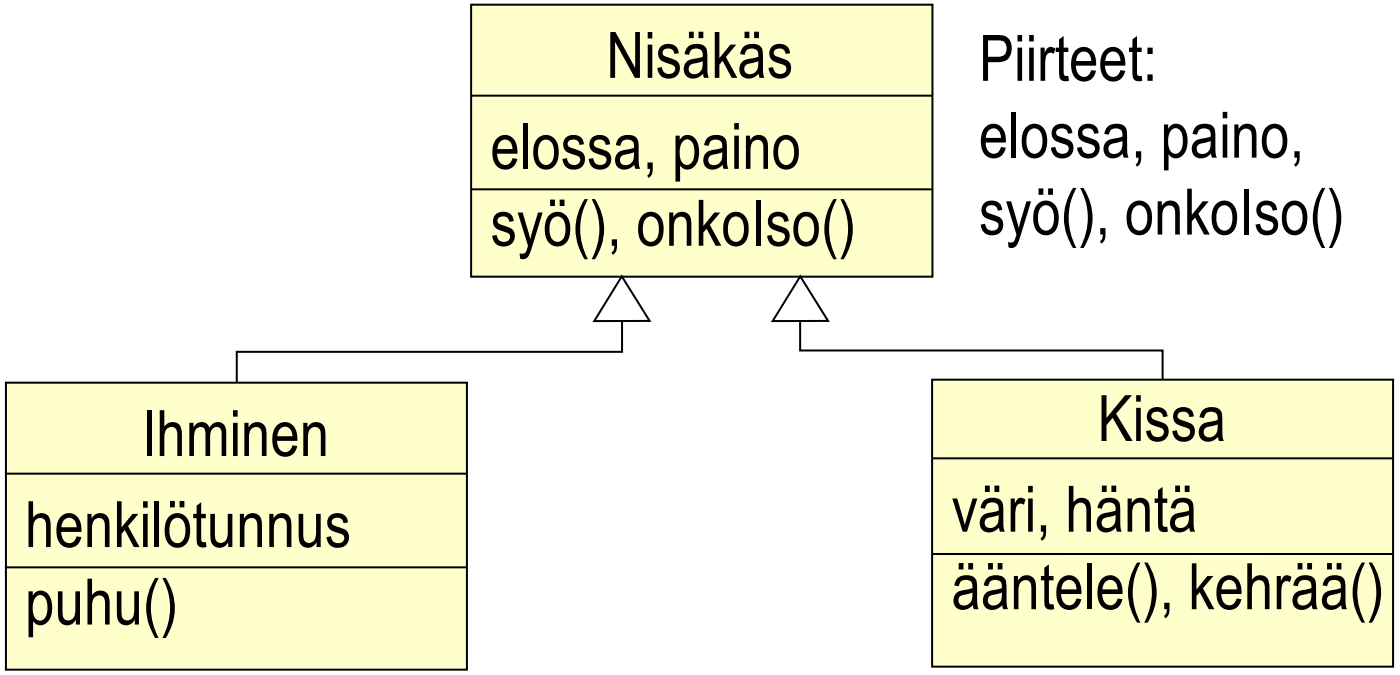


- *Nisäkäs* IsA *Eläin*
- *Ihminen* IsA *Nisäkäs, Kissa* IsA *Nisäkäs*
- *Opiskelija* IsA *Ihminen, Opettaja* IsA *Ihminen*

Periytymisen käyttö

- Aliluokan tulisi aina tuoda yliluokkaansa jokin tarkennus (tai laajennus):
  - Aliluokassa voidaan määritellä uusia piirteitä (attribuutteja ja metodeja).
  - Aliluokka voi korvata (override) yliluokassa määritellyn toiminnallisuuden toteuttamalla perityn metodin eri tavoin. Korvaaminen on eräs monimuotoisuuden muoto.
  - Aliluokka tyypillisesti rakentaa olion omalla tavallaan.

Nisäkäs, kissa ja ihminen



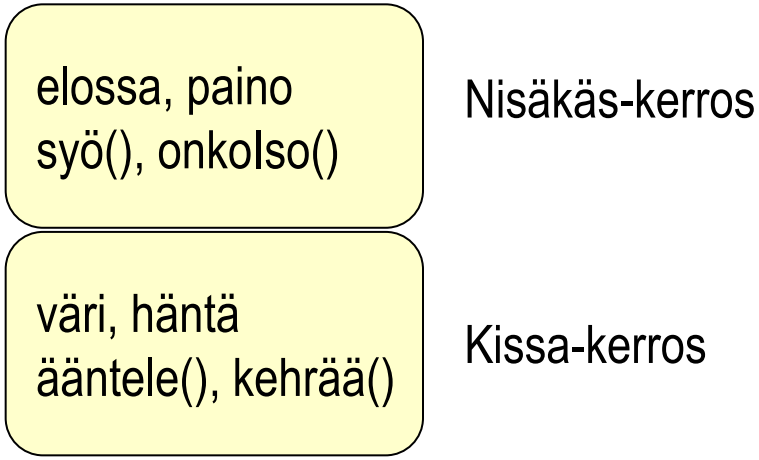
Piirteet:
elossa, paino,
syö(), onkolso()

Piirteet:
elossa, paino, syö(), onkolso(),
henkilötunnus, puhu()

Piirteet:
elossa, paino, syö(), onkolso(),
väri, häntä, ääntele(), kehrää()

Kissa-olio

- *Misse*-olio luodaan *Kissa*-luokasta: *Kissa* on *missen* perusluokka.
- Koska jokainen kissa on nisäkäs, on *misse* on sekä *Kissa* että *Nisäkäs*.
- *Misse* perii *Nisäkäs*-luokasta attribuutit *elossa* ja *paino* sekä metodit *syö*() ja *onkolso*(), jotka voidaan korvata *Kissa*-luokassa siten, että ne toimivat kissalle ominaisesti.
- Lisäksi *missellä* on kissoille ominaiset piirteet (esim. attribuutti *häntä* ja metodi *kehrää*().



9. Periytyminen Javassa

Sisällys

- Periytymismekanismi Java-kielessä.
- Piirteiden näkyvyys periytymisessä.
- Metodien korvaaminen ja **super**-attribuutti.
- Attribuutin peittäminen periytymisen kautta.
- Rakentajat ja periytyminen.
- Periytymisen estäminen.

Periytymismekanismi Java-kielessä

- Javan luokkahierarkia on yksijuurinen – kaikkien olioiden esi-isä on *Object*-luokka.
- *Object*-luokalla on joitakin hyödyllisiä piirteitä, jotka esitellään myöhemmin.
- Omia Java-luokkia ei tarvitse periyttää *Object*-luokasta: Java liittää automaattisesti tämän luokan omien luokkarakenteiden juuriluokan yliluokaksi.
- Javassa periytyminen ilmaistaan varatulla sanalla **extends**, jolla piirteet periytyvät automaattisesti.
- Perityt piirteet eivät ole välttämättä käytettävissä.

Periytymismekanismi Java-kielessä

- Yleisesti:

  määreet class Aliluokka extends Yliluokka {

  ...

  }

  missä määreet ja niiden puuttuminen säätelevät luokan näkyvyyttä (esimerkiksi **public**) ja luokan tyyppiä (**final** ja **abstract**).
- Luokat määritellään edelleen **public**-määreellä.

Nisäkäs ja kissa (Nisakas.java)

```
public class Nisakas {
   // Yliluokan attribuutit.
   private boolean elossa;
   private double paino;
   // Yliluokan metodit.
   public boolean elossa() { return elossa; }
   public void elossa(boolean e) { elossa = e; }
   public double paino() { return paino; }
   public void paino(double p) { if (p > 0) paino = p; }
   public void syo() { System.out.println("Syön kuin nisäkäs..."); }
   public boolean onkolso() { return false; }
}
```

Nisäkäs ja kissa (Kissa.java)

```
public class Kissa extends Nisakas {
   // Aliluokan attribuutit.
   private String vari;
   private String hanta;
   // Aliluokan metodit: rakentajat, aksessorit, muut oliometodit jne.
   ...
   public void kehraa() {
      System.out.println("murr, murrrr...");
   }
}
```

Nisäkäs ja kissa (Testi.java)

```
public class Testi {
   public static void main(String[] args) {
      // Luodaan kissa.
      Kissa mussu = new Kissa();
      mussu.paino(9);
      mussu.hanta("pörrö");
      System.out.println(mussu.paino());  // 9
      System.out.println(mussu.hanta());  // pörrö
      mussu.syo();                        // Syön kuin nisäkäs...
      mussu.kehraa();                     // murr, murrrr...
   }
}
```

- Huom! Kaikki tiedostot ovat samassa hakemistossa.

Piirteiden näkyvyys periytymisessä

- Javan näkyvyysmääreet (**private**, **protected** ja **public**) säätelevät piirteiden näkyvyyttä sekä luokkahierarkian sisällä että pakkausten välillä.
- Koska pakkaukset käsitellään myöhemmin, nyt riittää tietää vain seuraavaa:
  - **public**-määreellä esitellyt yliluokan piirteet näkyvät aina jälkeläisille ja ovat käytettävissä kaikissa muissa luokissa.
  - **private**-määre kätkee yliluokan piirteet sekä jälkeläisiltä että kaikilta muilta luokilta.
  - **protected**-määre on edellisten välimuoto, kun luokat on jaettu pakkauksiin.

Nisäkäs ja kissa

- Lisätään *Kissa*-luokkaan metodi:

  public boolean onkolso() { **return** paino > 10; }

- Koodi ei kuitenkaan mene läpi kääntäjästä, joka sanoo “paino has private access in Nisakas”.
- *paino*-attribuutin **private**-määre:

  private double paino;

  kätkee attribuutin kaikilta muilta luokilta mukaan lukien myös aliluokat, vaikka *Nisakas*-luokka on esitelty **public**-määreellä!

Kissa-luokan olio

- *mussu*-olion perusluokka on *Kissa*.
- *mussu* on myös *Nisakas*, mutta kätketty *paino*-attribuutti ei ole *mussun* käytettävissä.
- Attribuutti saadaan kuitenkin käyttöön epäsuorasti aksessorin avulla:

```
public boolean onkolso() {
    return paino() > 10;
}
```

~~paino~~, ~~elossa~~
paino()
paino(Double)
syo() ...

vari, hanta
aantele(String)
kehraa() ...

- Jos julkisten aksessoreiden määrittely ei ole järkevää, on luokkien attribuutit esiteltävä **protected**-määreellä ja luokat pakattava.
  - Näin kätketty tieto on huonommassa tallessa kuin **private**-määreellä esitelty.
  - **protected**-määrettä ei saa käyttää siksi, että ei jaksa kirjoittaa aksessoreita!

Korvaaminen

- Aliluokan metodi korvaa (override) yliluokan metodin, mikäli aliluokassa esitellään (näkyvyysmäärettä lukuun ottamatta) metodi täsmälleen samalla otsikolla kuin yliluokan metodi.
  - Näkyvyyttä voi laajentaa, mutta ei supistaa.
- Korvaaminen on siis eri asia kuin kuormittaminen (overloading), jossa metodin nimi pysyy samana, mutta parametrilista vaihtelee.
- Korvaamismekanismin avulla voidaan toteuttaa yliluokan määrittelemä toiminnallisuus aliluokalle ominaisella tavalla.
- Korvaamisen muoto ja sen tarve tapauskohtainen.
  - Korvattua versiota voidaan kutsua tarvittaessa.

Nisäkäs ja kissa

- Määritellään *Kissa*-luokkaan *syo*()-metodi, jossa kuvataan kuinka nimenomaan kissat syövät.
  - Metodin runko täysin uutta koodia.
- Kissa-luokan metodi:

```
public void syo() {
   System.out.println("Syön kuin kissa...");
   kehraa();
}
```

Nisäkäs ja kissa

```
public class Testi {
   public static void main(String[] args) {
      Nisakas nisse = new Nisakas();
      Kissa mussu = new Kissa();
      nisse.syo();              // Syön kuin nisäkäs...
      mussu.syo();              // Syön kuin kissa...
   }                            // murr, murrrr...
}
```

Korvaaminen

- Korvaamisen voi ajatella tunnuksen peittämisenä: aliluokan uusi versio syrjäyttää yliluokan version, mutta ei hävitä sitä.
- Peitettyyn metodiin voidaan viitata luokan **super**-attribuutilla (vertaa **this**). Yleisesti:

  super.metodinNimi(…);

- **super**-attribuutti on automaattisesti käytettävissä Javan toimesta.
  - **super**-attribuutti ei näy luokan ulkopuolelle.

Korvaaminen

- **super**-attribuutti viittaa olioon itseensä, mutta olio asetetaan yliluokkansa rooliin. (**this**-attribuuttia käytetään perusluokan roolissa.)
- Attribuutilla voidaan viitata vain yliluokan näkyviin osiin.

- Huom! Periytyminen on vain yksi luokkien välisistä suhteista (assosiaatioista) UML:lla piirretyissä luokka-kaavioissa. Assosiaatioita voidaan nimetä ja nimen yhteyteen voidaan piirtää myös nuoli, joka kuvaa assosiaation lukusuunnan.

Nisäkäs ja kissa

- Määritellään *Kissa*-luokkaan *syo*()-metodi, jossa syödään ensin kuten nisäkkäät yleensä ja sitten aletaan kehrätä tyytyväisenä.
  - Aluksi kutsutaan korvattua versiota.
- Kissa-luokan metodi:

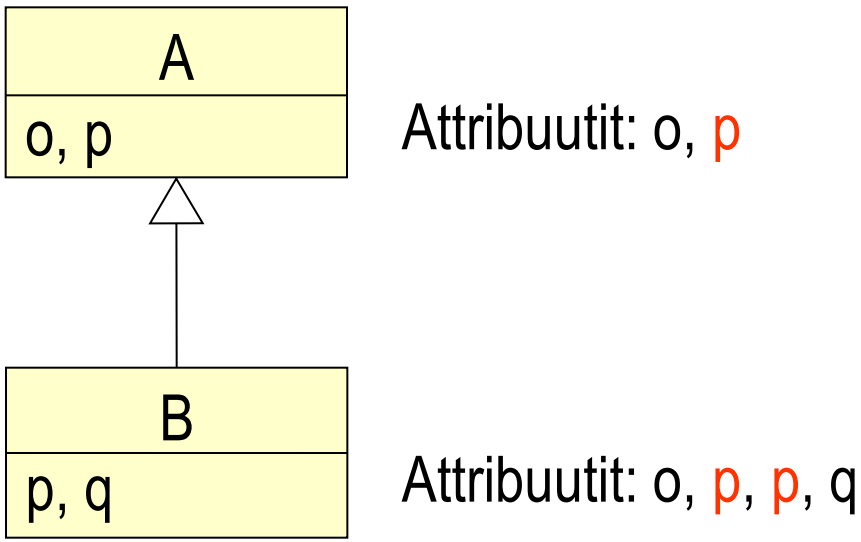
```
public void syo() {
  super.syo();
  kehraa();
}
```

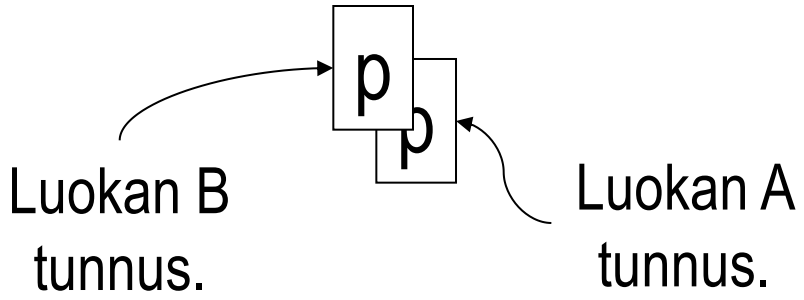
Nisäkäs ja kissa

```
public class Testi {
   public static void main(String[] args) {
      Nisakas nisse = new Nisakas();
      Kissa mussu = new Kissa();
      nisse.syo();              // Syön kuin nisäkäs...
      mussu.syo();              // Syön kuin nisäkäs...
   }                            // murr, murrrr...
}
```

Peittäminen periytymisen kautta

- Attribuutin tunnus peittää perityn attribuutin tunnuksen, kun jälkeläisessä (aliluokassa) esitellään attribuutti samalla tunnuksella kuin esi-isässä (yliluokassa).
- Vain aliluokassa näkyvä attribuutti voidaan peittää.
  - Yliluokan **public**- ja **protected**-määrein esitellyt.





Peittäminen periytymisen kautta

- Periytymisen kautta peitettyyn yliluokan attribuuttiin voidaan viitata aliluokassa **super**-attribuutin avulla.
- Peittämistä ei voi estää toisen attribuutin tyyppiä tai näkyvyysmäärettä vaihtamalla.
- Peittäminen koskee vain jälkeläistä: Esi-isän metodeissa käytetään vain esi-isän attribuuttia.
- Yleensä ottaen ei ole tarvetta eikä edes syytä esitellä attribuutteja siten, että ne peittävät esi-isän attribuutin.

Peittäminen periytymisen kautta

```
public class A {
   // Attribuutti p näkyy kaikkiin A:n jälkeläisluokkiin,
   // riippumatta siitä, että sijaitsevatko jälkeläiset
   // A-luokan pakkauksessa tai muissa pakkauksissa.
   // Attribuutti näkyy myös A:n pakkauksen muihin luokkiin.
   protected char p = 'a';
   public void tulosta() {
      System.out.println("A:n metodi: " + p);
   }
}
```

Peittäminen periytymisen kautta

```
public class B extends A {
   protected String p = "b";   // Peittää A:n attribuutin
   public void tulosta() {
      System.out.println("B:n metodi: " + p);
      System.out.println("super.p: " + super.p);
      super.tulosta();
   }
}
```

B-luokan metodi tulostaa:

```
B:n metodi: b
super.p: a
A:n metodi: a
```

Periytyminen ja rakentajat

- Rakentajat eivät periydy yliluokalta aliluokalle.
- Aliluokan oliota luotaessa kutsutaan automaattisesti yliluokan oletusrakentajaa.
- Yliluokan rakentaja alustaa yliluokan attribuutit, jonka jälkeen palataan suorittamaan aliluokan rakentajan sisältö.
- Aliluokan rakentajassa voidaan kutsua **super**-metodin avulla yliluokan rakentajaa, jolloin automaattista kutsua ei tapahdu.

Periytyminen ja rakentajat

- Yliluokan rakentajan kutsu yleisesti:

  super();

  tai

  super(parametrit);

- **public** Aliluokka() {
  super(); ...
  }

  public Aliluokka(**int** p) {
  super(p); ...
  }

- Yliluokan rakentajan kutsun on oltava aliluokan rakentajan ensimmäinen lause.
- Luokan oma rakentajaa voidaan kutsua luokan toisesta rakentajasta **this**-metodilla, jonka myös on oltava rakentajan ensimmäinen lause.
- **public** Aliluokka() {
  this(10); …
  }

Periytyminen ja rakentajat

- Rakentajat suoritetaan ketjussa:
  - Yliluokka kutsuu oman yliluokkansa rakentajaa joko automaattisesti tai ohjelmoijan toimesta kunnes päädytään *Object*-luokan rakentajaan.
  - Luokkahierarkiassa edetään takaisin suorittamalla esi-isien rakentajat, joista kukin alustaa oman osuutensa aliluokan perimistä attribuuteista.
  - Suoritetaan aliluokan rakentajan loput lauseet.
- Ketjuttamalla ei tarvitse kirjoittaa samaa koodia useampaan paikkaan.

Periytyminen ja rakentajat

```
public class A {
   private int a;
   public A() { a = 1; System.out.println("A:n 1. rakentaja"); }
   public A(int i) {
      a = i; System.out.println("A:n 2. rakentaja");
   }
   public void tulosta() { System.out.println("a: " + a); }
}
```

Periytyminen ja rakentajat

```
public class B extends A {
   private int b;
   public B() { b = 2; System.out.println("B:n 1. rakentaja"); }
   public B(int i) {
      super(); b = i; System.out.println("B:n 2. rakentaja"); }
   public void tulosta() {
      super.tulosta(); System.out.println("b: " + b);
   }
}
```

Periytyminen ja rakentajat

```
public class C extends B {
   private int c;
   public C() { c = 3; System.out.println("C:n 1. rakentaja"); }
   public C(int i) {
      super(i + 2);
      c = i; System.out.println("C:n 2. rakentaja");
   }
   public void tulosta() {
      super.tulosta(); System.out.println("c: " + c);
   }
}
```

Periytyminen ja rakentajat

```
public class Testi {
   public static void main(String[] args) {
      C s = new C();
      s.tulosta();
      C t = new C(10);
      t.tulosta();
   }
}
```

Näytölle tulostuu

```
A:n 1. rakentaja
B:n 1. rakentaja
C:n 1. rakentaja
a: 1
b: 2
c: 3
A:n 1. rakentaja
B:n 2. rakentaja
C:n 2. rakentaja
a: 1
b: 12
c: 10
```

Nisäkäs ja kissa (Nisakas.java)

```
public class Nisakas {
   ...
   public Nisakas() {
      paino = 0.1;
      elossa = true;
   }
   public Nisakas(boolean e, double p) {
      elossa(e);
      paino(p);
   }
   ...
}
```

Nisäkäs ja kissa (Kissa.java)

```
public class Kissa extends Nisakas {
   ...
   public Kissa(boolean e, double p, String v, String h) {
      // Kutsutaan yliluokan rakentajaa,
      // jolloin koodia ei tarvitse monistaa aliluokkaan.
      super(e, p);
      // Myös aksessoreita kutsumalla pääsee helpommalla.
      vari(v);
      hanta(h);
   }
   ...
}
```

Nisäkäs ja kissa (Testi.java)

```
public class Testi {
   public static void main(String[] args) {
      Kissa hassu = new Kissa(true, 2, "harmaa", "kippura");
      System.out.println(hassu.elossa());      // true
      System.out.println(hassu.paino());       // 2.0
      System.out.println(hassu.vari());        // harmaa
      System.out.println(hassu.hanta());       // kippura
   }
}
```

Periytymisen estäminen

- Luokan käyttö yliluokkana voidaan estää **final**-määreellä.
- Yleisesti:

  määreet final class LuokanNimi { ... }

  missä **määreet**-kohta on valinnainen.
- Samoin metodin korvaaminen kielletään varatulla sanalla **final**.

Norjalainen metsäkissa

```
public final class Kissa extends Nisakas {
   ...
}
// Kissan periytyminen Norjalaiseksi metsäkissaksi ei nyt
// onnistu (cannot inherit from final Kissa)
public class NorjalainenMetsakissa extends Kissa {
   public NorjalainenMetsakissa() { super(6, "tuuhea"); }
}
```

10. Abstrakti luokka

Johdanto

- Abstrakti luokka (abstract class) poikkeaa konkreettisesta luokasta (ei-abstrakti luokka) siten, että siitä ei voi luoda olioita – abstraktia luokkaa käytetään olioiden asemasta periytymisen avulla.
  - Luokalle voidaan kuitenkin kirjoittaa rakentajia.
  - Abstraktin luokan aliluokka voi olla abstrakti.
- Abstrakti luokka ei tyypillisesti toteuta kaikkia metodejaan: osalle metodeista (abstraktit) annetaan vain otsikot ja runkojen määrittely (toteutus) jätetään luokan jälkeläisille.
  - Abstraktissa luokassa ei ole pakko olla abstrakteja metodeja.

Johdanto

- Koska yksikin abstrakti metodi tekee luokasta abstraktin, konkreettisen aliluokan on toteutettava kaikki perimänsä abstraktit metodit.
- Abstrakti luokka soveltuu luontevasti abstraktien käsitteiden (esim. läheisyys, päivämäärä ja nisäkäs) kuvaamiseen.
- Abstrakti luokka mahdollistaa ohjelmointitekniikan, jolla varmistetaan, että abstraktin yliluokan konkreettisen aliluokan liittymässä on aina joukko metodeja, joita aliluokan on ollut pakko “tarkentaa” nämä metodit toteuttamalla.
  - Konkreettisen luokan aliluokan ei ole pakko korvata perittyjä metodeja.

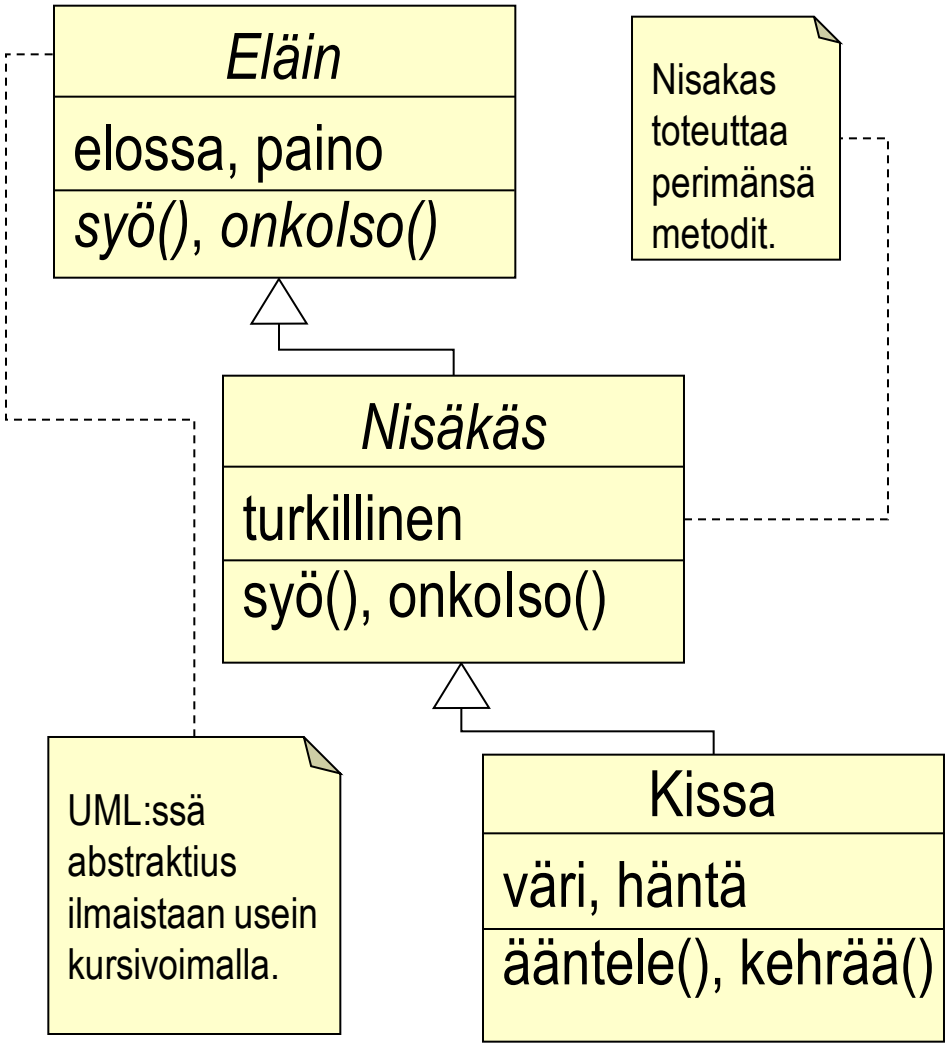
Abstraktin luokan määrittely

- Javassa varattua sanaa **abstract** käyttäen.
- Yleisesti:

  määreet abstract class LuokanNimi { ... }

  missä **määreet**-kohta on valinnainen.
- Määritellään jatkossa julkisiksi **public**-määreen avulla.
- Jälkeläisten toteuttamiksi tarkoitetut metodit annetaan **abstract**-määreellä ilman runkoa.
  - Toteutetun metodin otsikosta **abstract**-määre jätetään luonnollisesti pois.

Eläin, nisäkäs ja kissa



Piirteet:
elossa, paino, syö(), onkolso()

Piirteet:
elossa, paino, syö(), onkolso(),
turkillinen

Piirteet:
elossa, paino, syö(), onkolso(),
turkillinen,
väri, häntä, ääntele(), kehrää()

Eläin, nisäkäs ja kissa (Elain.java)

```
// Yksinkertainen abstrakti Elain-luokka.
public abstract class Elain {
   private boolean elossa;
   private double paino;
   // Rakentajat ja aksessorit.
   ...
   // Abstraktit oliometodit.
   public abstract void syo();
   public abstract boolean onkolso();
}
```

Eläin, nisäkäs ja kissa (Nisakas.java)

```
// Elain-luokasta peritty abstrakti Nisakas-luokka.
public abstract class Nisakas extends Elain {
   private boolean turkillinen;
   // Rakentajat ja aksessorit.
   ...
   // Toteutetut metodit.
   public void syo() {
      System.out.println("Syön kuin nisäkäs...");
   }
   public boolean onkolso() { return false; }
}
```

Eläin, nisäkäs ja kissa (Kissa.java)

```
// Nisakas-luokasta peritty konkreettinen Kissa-luokka.
public class Kissa extends Nisakas {
   private String vari;
   private String hanta;
   // Rakentajat, aksessorit ja muut oliometodit.
   ...
   // Korvatut metodit.
   public void syo() {
      System.out.println("Syön kuin kissa...");
      kehraa();
   }
   public boolean onkolso() { return paino() > 10; }
}
```

Eläin, nisäkäs ja kissa (Testi.java)

```
public class Testi {
   public static void main(String[] args) {
      // Tämä ei menisi kääntäjästä läpi,
      // koska abstraktista luokasta ei voida luoda olioita.
      // Nisakas nisse = new Nisakas();
      Kissa mussu = new Kissa();
      mussu.syo();                    // Syön kuin kissa...
   }                                  // murr, murrrr...
}
```

11. Rajapinnat

Sisällys

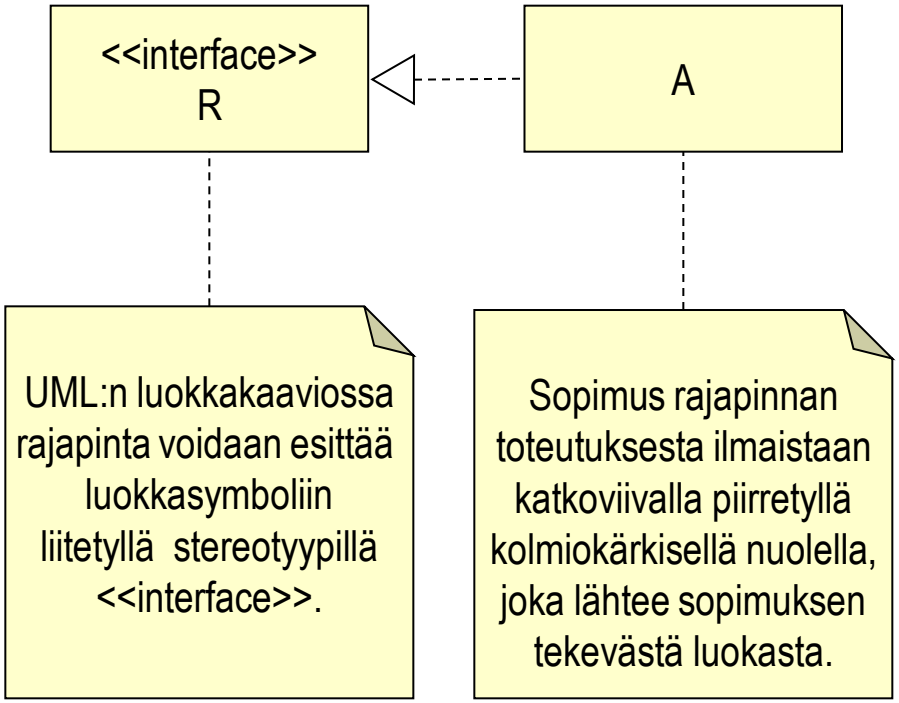
- Johdanto.
- Abstrakti luokka vai rajapinta?
- Rajapintojen hyötyjä.
- Kuinka rajapinnat määritellään ja otetaan käyttöön?
- Eläin, nisäkäs, kissa ja rajapinta.
- Moniperiytyminen rajapintojen avulla.
- Varoituksen sana.

Johdanto

- Rajapinnat (interface) muodostavat yhdessä taulukkojen ja luokkien kanssa Javan viitetyypit.
- Rajapinta muistuttaa abstraktia luokkaa.
  - Rajapinnasta ei voida muodostaa olioita.
  - Voidaan käyttää tunnuksen tyyppinä.
- Rajapinta on kuitenkin abstraktia luokkaa selvästi abstraktimpi tyyppi.
  - Kaikkien metodien oltava abstrakteja ja julkisia.
  - Attribuutit ovat aina julkisia luokkavakioita.

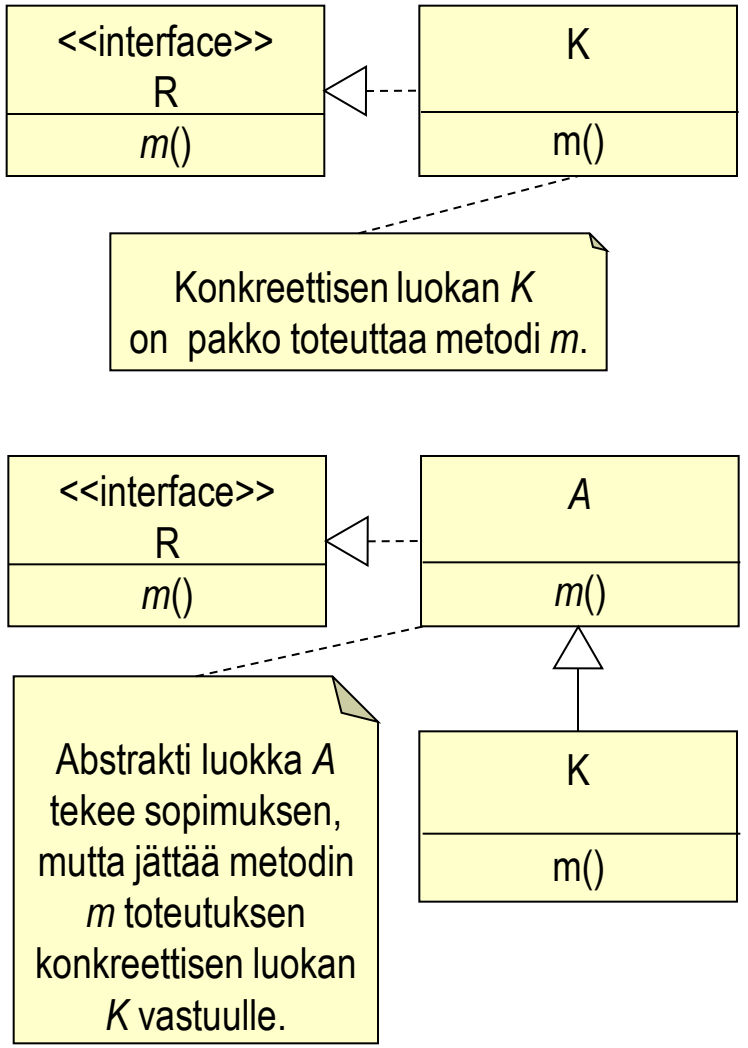
Johdanto

- Rajapintaa ei voi liittää sellaisenaan luokka-hierarkiaan, vaan jonkin luokan on toteutettava rajapinta.
  - Toteutus on koostuu avainsanalla ilmaistavasta sopimuksesta ja rajapinnan metodien toteutuksista yhdessä tai useammassa luokassa.



Johdanto

- Rajapinnan toteutuksesta sopivan konkreettisen luokan on toteutettava kaikki rajapinnan metodit.
- Abstrakti luokka voi sopia toteutuksesta, mutta siinä voidaan toteuttaa joko kaikki, vain osa tai ei yhtäkään rajapinnan metodeista.
  - Abstraktin luokan konkreettisten jälkeläisten on toteutettava kaikki perimänsä abstraktit metodit.



Johdanto

- Rajapinnan toteutus periytyy.
  - Toteutettu metodi on käytettävissä toteuttavan luokan jälkeläisissä ja se voidaan kuorimittaa tai korvata.
  - Rajapinnan kanssa sopimuksen tekevä luokan ja sen jälkeläisten katsotaan olevan myös rajapinnan tyyppiä.
    - Luokan ja sen jälkeläisten olioihin voidaan liittää rajapinnan tyyppinen viite.
- Mikä tahansa luokka, millä tahansa luokkahierarkian tasolla voi toteuttaa rajapinnan.
  - Toteutuksen periytymistä kannattaa hyödyntää mahdollisuuksien mukaan: mitä lähempänä juuriluokkaa toteutuksesta sovitaan, sitä laajempi osa hierarkiasta saa toteutuksen käyttöönsä.
  - Toisinaan on järkevää toteuttaa rajapinta vain osassa luokkahierarkian “lehtiluokkia”.

Rajapintojen hyötyjä

- Rajapinnalla voidaan määritellä liittymä ottamalla mahdollisimman vähän kantaa liittymän toteutukseen.
- Pakottaa sopimuksen tekevään luokkaan ja sen jälkeläisille liittymän, jossa on aina tietyt piirteet.
- Vaihtoehto, kun piirteiden kokoaminen abstraktiin luokkaan ja niiden hyödyntäminen periytymisen kautta ei ole käytännöllistä.
- Rajapinnoilla voidaan simuloida moniperiytymistä.

Abstrakti luokka vai rajapinta?

- Abstraktia luokkaa on syytä käyttää rajapinnan sijasta, kun sovelluksessa on samankaltaisia luokkia, joiden yhteiset piirteet voidaan koota luontevasti abstraktiin luokkaan ja periä siitä.
  - Valinta abstraktin luokan ja rajapinnan välillä on toisinaan vaikea tehtävä.
- Javan API:n luokkahierarkiat ovat moderniin tapaan matalia ja API-luokat toteuttavat tyypillisesti yhden tai useampia rajapintoja.
  - Esimerkiksi *String*-luokka periytyy suoraan *Object*-luokasta ja toteuttaa kolme rajapintaa.

Määrittely

- Rajapinta määritellään varatulla sanalla **interface**. Yleisesti (kursivoidut osat ovat valinnaisia):

```
public interface RajapinnanNimi extends
Rajapinta1, Rajapinta2, ... , RajapintaN {
  // Vakioiden määrittelyt.
  // Metodien otsikot.
}
```

- Esimerkki.

```
public interface Sahiseva {
  // Määritellään liittymä antamalla otsikot metodeille,
  // jotka mallintavat sähisevien olioiden käytöstä.
}
```

Määrittely

- Rajapintojen näkyvyysmääreet toimivat pakkausten tasolla kuten luokissa.
  - Määritellään toistaiseksi julkisiksi **public**-määrettä käyttäen.
- Rajapintoja nimetään vaihtelevasti.
  - Javan omia rajapintoja ovat esimerkiksi luokkamaisesti nimetty *Comparator* (substantiivi) ja ei-luokkamaisesti nimetty *Comparable* (adjektiivi).
- Rajapinta sijoitetaan luokkien tapaan nimensä mukaan nimettyyn tiedostoon. Esimerkiksi *Sahiseva*-rajapinta sijoitettaisiin *Sahiseva.java*-nimiseen tiedostoon.
  - Yhdessä tiedostossa vain yksi rajapinta.

Määrittely

- Rajapinnan attribuutit ovat aina julkisia luokkavakioita ja metodit ovat puolestaan aina julkisia ja abstrakteja.
- Implisiittiset määreet voidaan jättää rajanpinnan attribuuttien ja metodien esittelyistä pois.
  - Abstraktissa luokassa määreillä on aina merkitystä. Esimerkiksi **public**-määreen poistaminen muuttaa näkyvyyttä.
- Esimerkki.

  public final static double E = 2.72; $\Leftrightarrow$ **double** E = 2.72;

  public abstract void kertoma(**int** n); $\Leftrightarrow$ **void** kertoma(**int** n);

Toteutus

- Luokka ilmaisee toteuttavansa rajapinnan varatulla sanalla **implements**. Yleisesti:

  määreet **class LuokanNimi *extends***

  YliluokanNimi **implements** Rajapinta1, *Rajapinta2,*

  ... , RajapintaN {

  // Kaikkien rajapintametodien toteutus täällä,
  // jos luokka on konkreettinen. Abstrakti luokka voi
  // voi toteuttaa valinnaisen määrän metodeja.

  }

- Yllä kursivoidut osat ovat valinnaisia.

Toteutus

- Esimerkki.

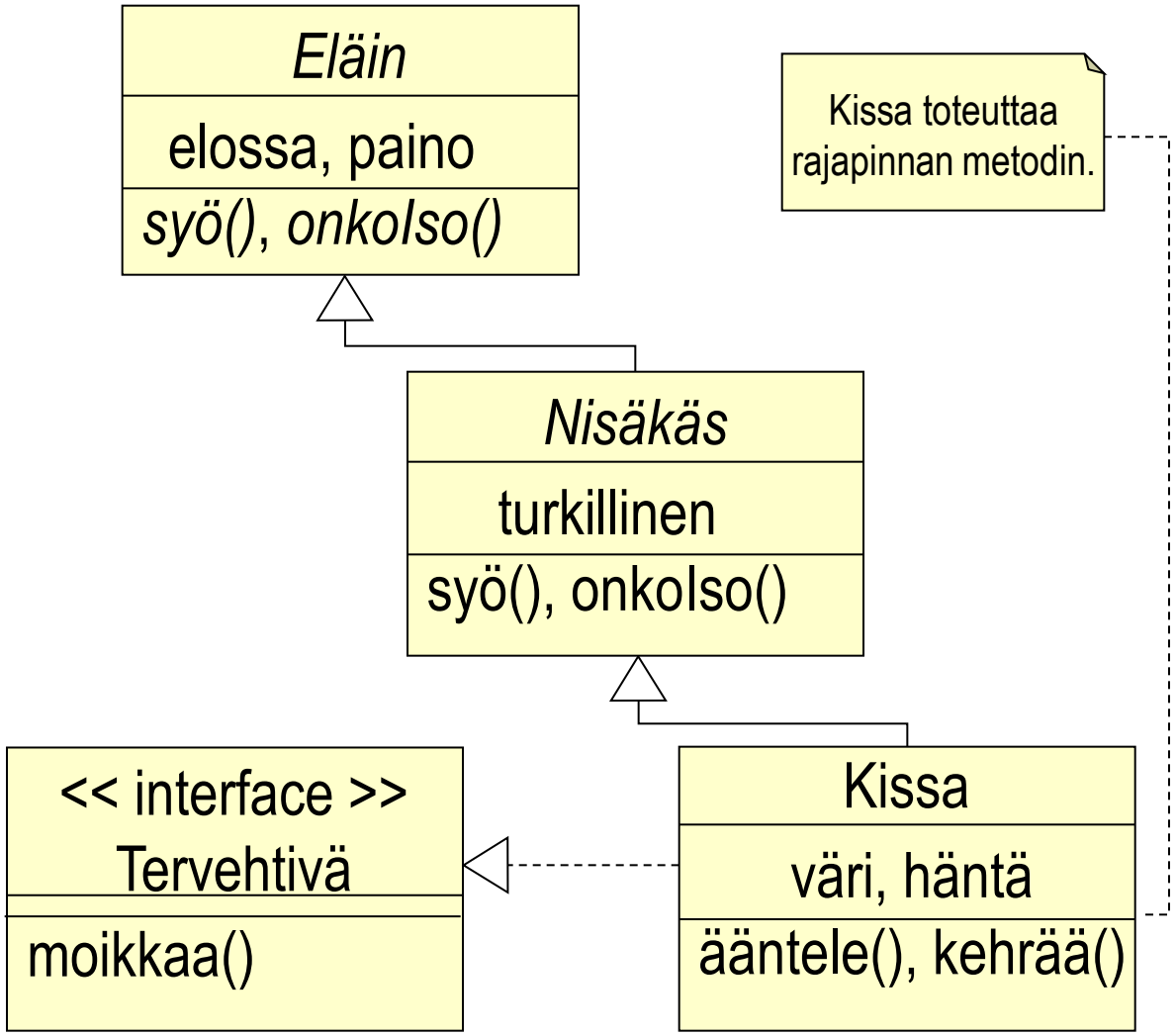
```
public class Kissa implements Sahiseva {
  // Tällä toteutetaan kaikki Sahiseva-
  // rajapinnan metodit, koska kissa on
  // konkreettinen.
}
```

- Toteutuksen jälkeen kissat ovat sähiseviä.
  - *Kissa*-luokan oliot voivat ilmaista sähinää kissoille ominaisella tavalla, mutta toisaalta on taattua, että kissa osaa aina sähistä juuri rajapinnassa luetelluilla metodeilla.
  - *Kissa*-luokan olioihin voidaan liittää *Sahiseva*-tyyppisiä viitteitä.

Tervehtiva-rajapinta

- Käytetään edelleen esimerkkinä *Nisakas*-, *Ihminen*- ja *Kissa*-luokkia.
- Sekä ihminen että kissa tervehtivät tuttuja, mutta on myös nisäkkäitä, jotka eivät osaa tervehtiä.
  - Tästä syystä tervehtimistoiminnallisuutta ei voi oikein sijoittaa *Nisakas*-luokkaan tai “väliluokkaan” *TervehtivaNisakas*, josta perittäisiin tervehtivät oliot.
- Ratkaistaan ongelma *Tervehtiva*-rajapinnalla, jonka esimerkiksi kissa ja ihminen toteuttavat.

Eläin, nisäkäs, kissa ja rajapinta



Piirteet: elossa, paino, syö(), onkolso()

Piirteet: elossa, paino, syö(), onkolso(), turkillinen

Piirteet: elossa, paino, syö(), onkolso(), turkillinen, väri, häntä, ääntele(), kehrää(), moikkaa()

Tervehtiva-rajapinta

Tervehtiva.java:

```
public interface Tervehtiva {
   // public- ja abstract-määreet saataisiin
   // myös automaattisesti.
   public abstract void moikkaa();
}
```

Kissa.java:

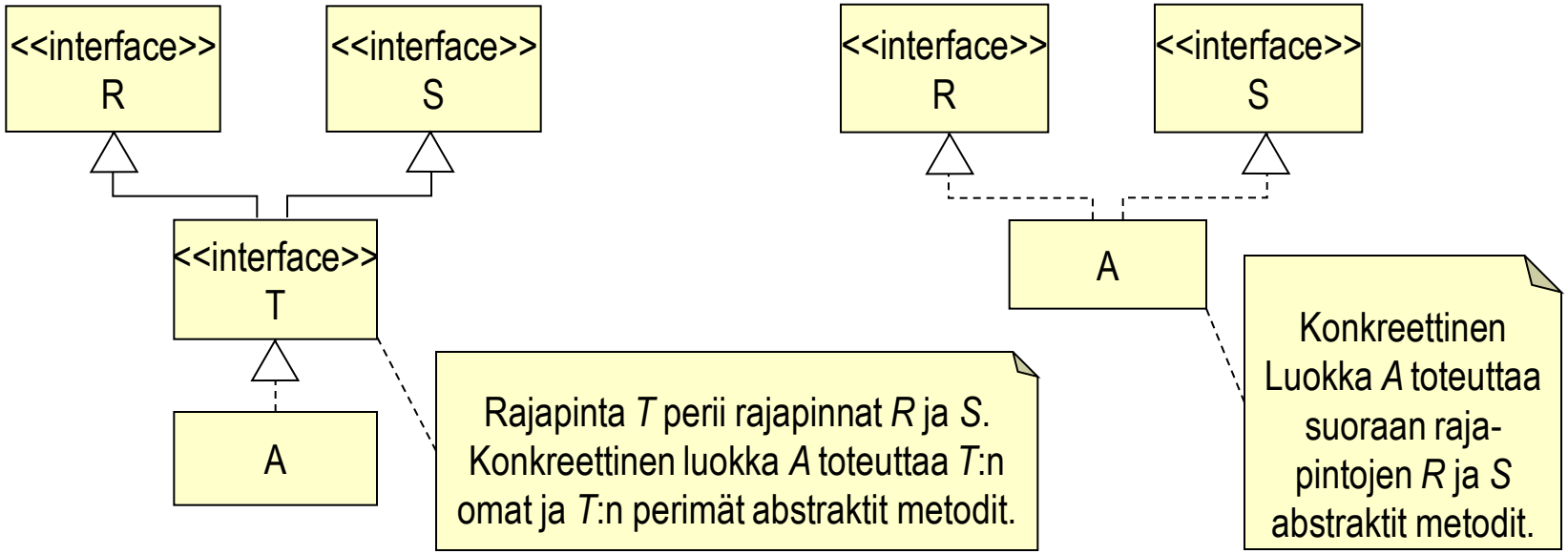
```
public class Kissa extends Nisakas
implements Tervehtiva {
      ...
      public void moikkaa() {
         System.out.println("Miu! Pusken jalkaa...");
   }
}
```

Moniperiytyminen

- Javassa on tarjolla vain yksittäisperiytyminen: *Object*-luokka on kaikkien luokkien esi-isä eikä millään luokalla voi olla useampia yliluokkia.
- Rajapinta voi kuitenkin periä yhden tai useampia rajapintoja **extends**-avainsanan avulla, jolloin rajapinnoista voidaan muodostaa luokka-hierarkioiden tapaisia rajapintahierarkioita.
- Koska rajapinnalla voi olla useita ylirajapintoja, voidaan näin toteuttaa moniperiytyminen – tosin melko rajoitetussa muodossa.

Moniperiytyminen

- Useasta rajapinnasta periytyvän rajapinnan toteutus siirtää moni-periytymisen epäsuorasti luokkahierarkiaan.
- Moniperiytymisen voidaan ajatella myös tapahtuvan, kun luokka toteuttaa suoraan useita rajapintoja (esimerkiksi *String*-luokka).



Varoituksen sana

- Rajapintaa hyödyntävien luokkien on toteutettava kaikki rajapinnan metodit, jolloin rajapintaa uusilla metodeilla laajennettaessa täytyy kaikkia rajapinnan toteuttavia luokkia muuttaa!
- Ratkaisuja:
  - Tulevaisuuden tarpeita voi huomioida “ylimääräisillä” metodeilla jo rajapintaa suunniteltaessa.
  - Vanhasta rajapinnasta voidaan periä **extends**-sanalla uusi laajempi rajapinta, jonka käyttö jää asiakkaiden päätettäväksi.

12. Monimuotoisuus

Sisällys

- Johdanto.
- Periytymismekanismi määrittää alityypityksen.
  - Viitteiden sijoitus ja vertailu.
- Staattinen ja dynaaminen luokka.
- Parametrinvälitys eräs monimuotoisuuden sovellus.
- Tyyppimuunnos.
- Rajapinnat ja alityypitys.

Johdanto

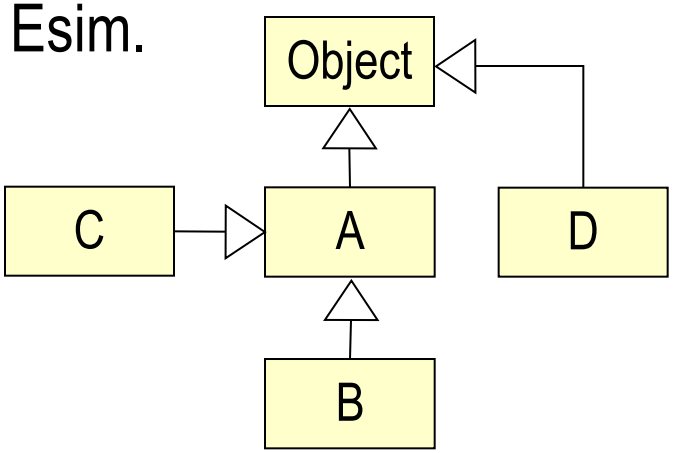
- Monimuotoisuus eli polymorfismi (polymorphism) on yleisesti sitä, että ohjelman tunnus voi tarkoittaa erilaisia asioita.
- Monimuotoisuus on monimuotoinen käsite.
  - Metodien kuormitus ja korvaaminen eräänlaista monimuotoisuutta.
- Monimuotoisuus liittyy erityisen kiinteästi periytymiseen:
  - Luokkahierarkia määrittelee alityypityksen, jossa kukin jälkeläinen edustaa alityyppiä ja kukin esi-isä ylityyppiä.
- Tarkastellaan jatkossa monimuotoisuutta Java-kielen näkökulmasta.

Alityypitys

- Määritellään tyyppien yhteensopivuus (<:) hyvin vapaamuotoisesti luokkien periytymissuhteen avulla:
  - Tyyppi *T* (luokka) on yhteensopiva itsensä kanssa.
    - *T* <: *T* (refleksiivisyys).
  - Alityyppi *S* (aliluokka) on yhteensopiva ylityyppinsä *T* (yliluokkansa) kanssa.
    - *S* <: *T*.
  - Alityyppi *S* (jälkeläinen) on yhteensopiva kaikkien ylityyppiensä *Q* (esi-isiensä) kanssa.
    - Jos *S* <: *T* ja *T* <: *Q*, niin *S* <: *Q*, (transitiivisuus).
  - Ylityyppi *Q* (esi-isä) ei ole yhteensopiva alityyppiensä *T* (jälkeläistensä) kanssa (antisymmetrisyys).

Alityypitys

- Esim.



- Koska *Object*-luokka on *A*-, *B*-, *C*- ja *D*-luokkien esi-isä (*Object* > { *A*, *B*, *C*, *D* }), niin *A*-, *B*-, *C*- ja *D*-tyypit ovat yhteensopivia *Object*-tyypin kanssa.
- Samoin koska *A*-luokka on *B*- ja *C*- luokkien esi-isä (*A* > { *C*, *B* }), niin *B*- ja *C*-tyypit ovat yhteensopivia *A*-tyypin kanssa.
- Toisaalta esimerkiksi *A*-tyyppi ei ole yhteensopiva *B*-tyypin kanssa, koska *A*-luokka ei ole *B*-luokan jälkeläinen.
- Huomaa myös, että esimerkiksi *C*- ja *D*-tyypit eivät sovi yhteen, koska niitä vastaavien luokkien välillä ei ole periytymissuhdetta.

Alityypitys

- Alityypityksen seurauksena viitteiden *x* (*X*-luokka) ja y (*Y*-luokka) sijoitus *x* = *y*; on sallittu, jos *Y* <: *X* eli *X* > *Y*.
  - Alityyppi käy ylityyppiinsä.
- Esimerkki:

```
Object o = null; A a = null;
B b = null; C c = null;
D d = null;
// Oikeellisia sijoituksia.
o = a; // A <: Object
o = b; // B <: Object
o = c; // C <: Object
o = d; // D <: Object
a = b; // B <: A
a = c; // C <: A
// Virheellisiä sijoituksia.
a = o; // Object > A
b = a; // A > B
c = d; // ei periytymissuhdetta
```

- Vertailussa operadien oltava yhteensopivia, järjestyksellä ei ole väliä.

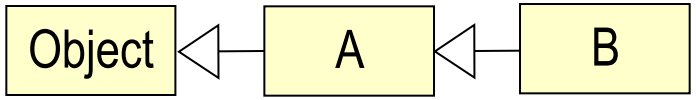
Alityypitys

- Viite ja siihen liittyvä olio voivat olla eri luokista, kunhan tyypit vain ovat sijoitusyhteensopivia.
  - Viitteeseen voi liittyä viitteen luokan olioiden lisäksi myös olioita viitteen luokan jälkeläisluokista.
- Esim.

```
Object o = null;
o = new A();
Nisakas n = new Kissa();
Object p = new String();
```

Staattinen ja dynaaminen luokka

- Erotetaan jatkossa viitteen luokka (staattinen luokka) ja olion perusluokka (dynaaminen luokka) toisistaan.
- Dynaaminen luokka vaihtelee staattisen luokan asettamissa rajoissa.
  - Staattinen luokka on dynaamisen luokan “yläraja”.



Staattinen luokka	Dynaaminen luokka voi olla
Object	Object, A, B
A	A, B
B	B

```
// Oikein (Object > A)
Object o = new A();
// Väärin (A > B)
B p = new A();
```

Staattinen ja dynaaminen luokka

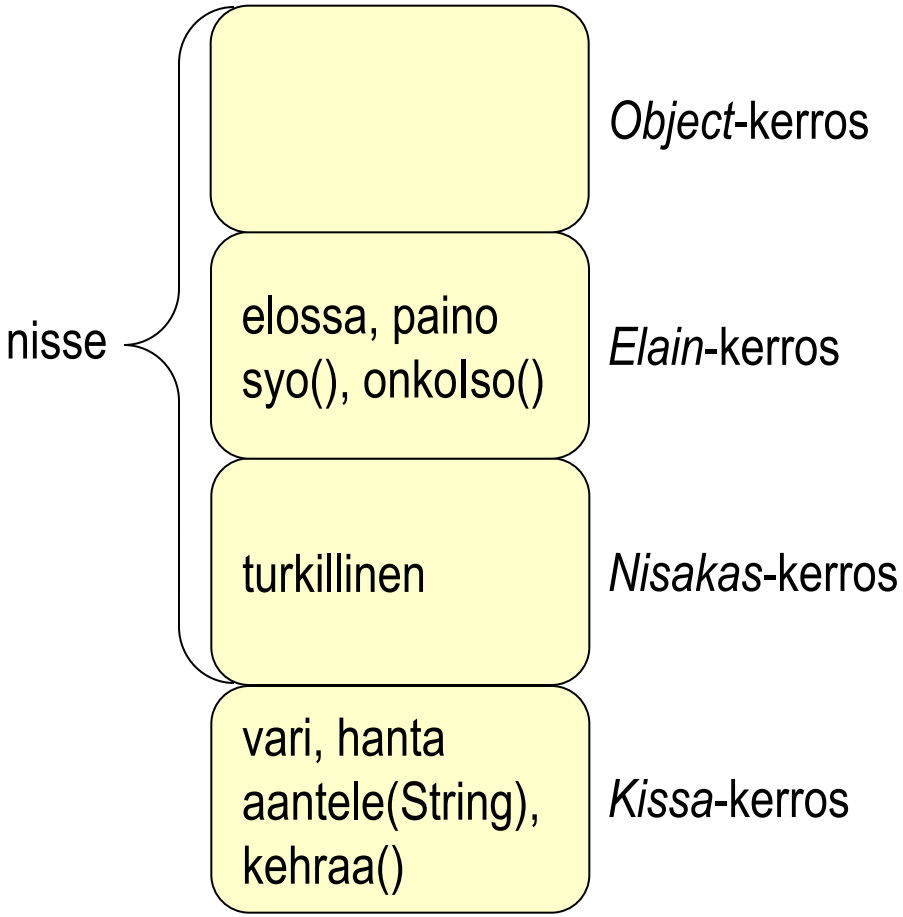
Esimerkki	Staattinen luokka	Dynaaminen luokka
Kissa molli = **new** Kissa();	Kissa	Kissa
Nisakas nisse = **new** Kissa();	Nisakas	Kissa

Object elain;
elain = **new** Kissa();
elain = **new** Koira();
	Object	
		Kissa
		Koira

// Tämä ei käänny,
// koska Object > Kissa.
Kissa mussu = **new** Object();
	
	(Kissa)
		
		(Object)

Staattinen ja dynaaminen luokka

- Java “näkee” olion viitteen luokan (staattinen luokka) kautta: ilman tyyppimuunnosta käytettävissä ovat vain staattisen luokan piirteet.
- // Asetetaan kissaan
  // Nisakas-luokan viite.
  Nisakas nisse = **new** Kissa();
  // Kissa ei osaa nyt esim.
  // kehrätä, koska staattinen
  // luokka on Nisakas.
  // Kääntäjä tuottaa virheen.
  nisse.kehraa();



Staattinen ja dynaaminen luokka

- Olio-ohjelmassa viitteeseen liittyvän olion tyyppi selviää usein vasta ohjelman ajon aikana.
  - Tässä tapauksessa myös kutsuttavan metodin versio täytyy päätellä ajon aikana. Dynaamisesti ajon aikana tapahtuvaa metodikutsujen liittämistä metodeihin kutsutaan myöhäiseksi sidonnaksi.
  - Proseduraalisissa ohjelmointikielissä aliohjelmien kutsut voidaan liittää aliohjelmien koodiin jo käännösaikana: kutsut ja määrittelyt “sidotaan” siis staattisesti (aikainen sidonta).

Parametrinvälitys

- Monimuotoisuutta hyödynnetään usein parametrinvälityksessä.
- Parametrin tyypiksi asetetaan jokin riittävän lähellä luokkahierarkian juurta oleva luokka, jolloin metodille voidaan antaa parametriksi jälkeläisluokkien viitteitä (ja olioita).

- **public void** rokota(Nisakas n) {
  // Täällä rokotetaan
  // muun muassa
  // kissoja, koiria ja ihmisiä.
  }
  …
- Kissa rontti = **new** Kissa();
  // Toimii, koska Kissa <: Nisakas.
  rokota(rontti);
  …

Parametrinvälitys

- **instanceof**-operaattorilla voidaan selvittää parametriin liittyvän olion luokka, jos tiedetään mitä luokkia oliot edustavat.
- Yleisesti:

  viite **instanceof** LuokanNimi
- Operaattori palauttaa totuusarvon **true**, mikäli viite osoittaa olioon, jonka luokka (tai esi-isä) on nimeltään *LuokanNimi*.

- ```
  public void rokota(Nisakas n) {
    if (n instanceof Kissa) {
      // Täällä rokotetaan kissa.
      …
    }
    else if (n instanceof Ihminen) {
      // Täällä rokotetaan ihminen
      …
  ```

- Huom! Operaattorin runsas käyttö ei ole suotavaa, koska yleensä pyritään mahdollisimman erillisiin luokkiin.
  ```

Tyyppimuunnos

- Eksplisiittiset tyyppimuunnokset **()**-operaattorilla ovat tuttuja jo alkeistyyppien yhteydestä.
- Esim.

```
// 1.25
double osamaara = 5 / (double)4;
// Väliaikainen muunnos,
// osamäärän palaa arvoon 1.25
// tämän lauseen jälkeen
System.out.println((int)osamaara); // 1
// Tallennetaan muunnoksen tulos.
int kokonaisosa = (int)osamaara; // 1
```

Tyyppimuunnos

- Myös viitteen luokka (staattinen luokka) voidaan tarvittaessa muuttaa toiseksi tyyppimuunnosoperaatiolla.
- Yleisesti: **(LuokanNimi)viite** ja **((LuokanNimi)viite).piirre** missä *LuokanNimi* voi olla viitteen luokan esi-isän tai jälkeläisen nimi.
  - // Luodaan Kissa-luokan olio ja asetetaan siihen Object-viite.
    Object mussu = **new** Kissa();
    // Ei käänny. Staattinen luokka on Object, joka ei kehrää.
    mussu.kehraa();
    // Kääntyy. Staattinen luokka tässä lauseessa Kissa.
    ((Kissa)mussu).kehraa();

Tyyppimuunnos

- Alkuperäinen luokka palautuu muunnoksen jälkeen, mutta tarvittaessa muunnoksen tulos voidaan sijoittaa toiseen viitteeseen, jonka luokka on sama kuin muunnoksessa käytetty luokka.
  - // Asetetaan olioon viite, jonka luokka on Kissa.

    Kissa mussu2 = (Kissa)mussu;

    // Onnistuu. Staattinen luokka Kissa.

    mussu2.kehraa();
- Yleensä luokkatyyppi muunnetaan siten, että viitteen tyyppi muunnetaan alityypikseen (downcasting), koska moni-muotoisuutta hyödynnetään usein parametrinvälityksessä.

Tyyppimuunnos

- Tyyppimuunnos on vaarallinen operaatio, koska kääntäjä tarkistaa vain onko tyyppimuunnos kieliopillisesti oikein.
- Tyyppimuunnosten avulla viite voidaan liittää täysin väärän tyyppiseen olioon, jolloin staattisen luokan piirteiden käsittely aiheuttaa ohjelman pysäyttävän poikkeuksen.
- Esim.

```
// Staattinen luokka Object, dynaaminen Kissa.
Object k = new Kissa();
// Voidaan muuntaa, koska Object > String,
// mutta ohjelma kaatuu (ClassCastException).
((String)k).length();
```

Tyyppimuunnos

- Ajonaikaisia tyyppitarkistuksia voidaan tehdä **instanceof**-operaattorilla ennen tyyppimuunnosta.
- ```
  public void rokota(Nisakas n) {
    if (n instanceof Kissa) {
      // Tyyppimuunnos turvallinen, kun tiedetään
      // dynaamiseksi luokaksi Kissa.
      ((Kissa)n).sahise(); …
    }
    else if (n instanceof Ihminen) {
      // Täällä rokotetaan ihminen
      …
  ```
  ```

Rajapinnat ja alityypitys

- Rajapinnat noudattavat alityypityssääntöjä.
- Rajapinnan toteuttava luokka ja sen jälkeläiset katsotaan rajapinnan kanssa yhteensopiviksi.
- Oletetaan, että *Kissa*-luokka toteuttaa *Tervehtiva*-rajapinnan.

```
// Staattinen ja dynaaminen luokka Kissa.
Kissa mussu = new Kissa();
// Staattinen luokka Tervehtiva, dynaaminen luokka Kissa.
Tervehtiva terve = mussu;
terve.moikkaa();    // Kääntyy; kutsutaan rajapinnan metodia.
terve.syo();        // Ei käänny; metodi ei viitteen luokassa.
```

13. Pakkaukset

Sisällys

- Pakkauksen kokoaminen (**package**).
- Pakkaukset ja hakemistorakenne.
- Pakkauksen luokkien käyttö muissa pakkauksissa (**import**).
- Pakkaukset ja näkyvyys.

Pakkauksen kokoaminen

- Luokka tai rajapinta liitetään pakkaukseensa varatulla sanalla **package**.
- Yleisesti: **package** pakkaus;
- Pakkausmäärittely on lähdekooditiedoston ensimmäinen lause.
  - Vain kommentit voivat edeltää **package**-lausetta.
- Pakkausten nimissä käytetään vain pieniä kirjaimia.
- Lähdekooditiedostossa voi olla vain yksi **package**-lause.

- Esim. liitetään *In*-luokka *oope*-pakkaukseen.

```
package oope;
import java.util.*;  // Scanner
/*
 * Scanner-luokan palveluita
 * hyödyntävä apuluokka ...
 */
final public class In {
   ...
}
```

Pakkauksen kokoaminen

- Pakkaukselle voidaan määritellä pistenotaatiolla alipakkauksia.
- Esim. luentoihin, harjoituksiin ja harjoitustyöhön liittyvät lähdekooditiedostot voitaisiin liittää *oope*-pakkauksen alipakkauksiin lauseita:

  package oope.luennot;

  package oope.harjoitukset;

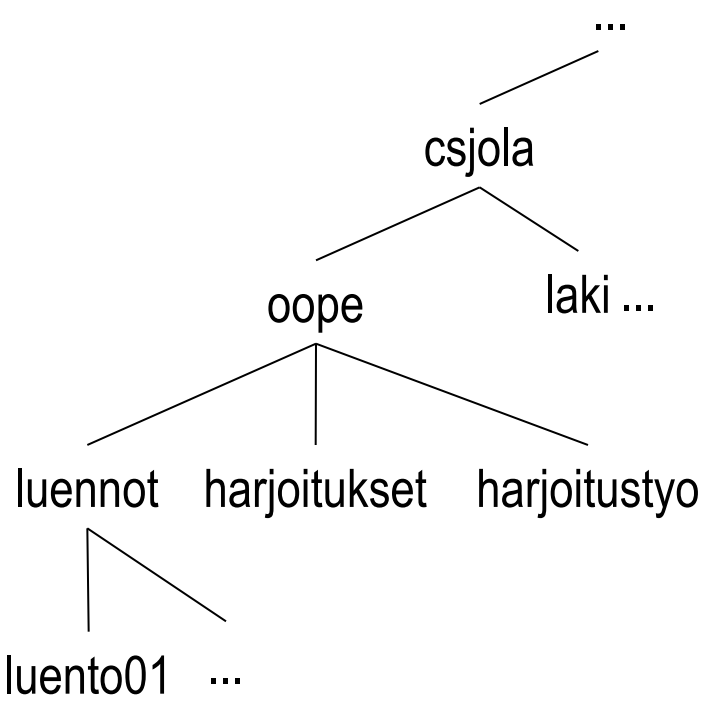
  package oope.harjoitustyo;

  asianomaisissa lähdekoodi-tiedostoissa käyttäen.

- Jokaiselle alipakkaukselle voidaan määritellä alipakkauksia.
- Pakkaukset pystytään nimeämään yksikäsitteisesti Internet-nimiavaruuden avulla.
  - Pakkauksen nimen alku esimerkiksi tekijän sähköpostiosoite käänteisessä järjestyksessä.
  - Esimerkiksi: *fi.uta.csjola*

Pakkaus ja hakemisto

- Pakkaukset jakavat nimiavaruutta puumaisesti:



- Pakkaus järjestetään massa-muistissa hakemistorakenteeksi siten, että kansiot nimetään pakkausten mukaan ja kuhunkin kansioon sijoitetaan vastaavan pakkauksen tiedostot.
- Oletetaan, että “nykyinen hakemisto” on Windows-käyttöjärjestelmässä *C:\ ... \fi\uta\csjola.* Tällöin esimerkiksi *oope*- ja *oope.luennot*-pakkausten tiedostojen tulee olla hakemistossa: *C:\ ... \fi\uta\csjola\oope* ja *C:\ ... \fi\uta\csjola\oope\luennot*.

Pakkaus ja hakemisto

- Ohjelmoijan on huolehdittava itse siitä, että lähde- ja tavukooditiedostot ovat oikeassa hakemistossa.
- Esim. “**package** fi.uta.csjola.oope;”-lauseella alkavat lähdekooditiedostot (ja vastaavat tavukooditiedostot) tulee sijoittaa alihakemistoon *fi\uta\csjola\oope*.
- Pakkauksiin jaetun ohjelman käynnistämiseksi on siirryttävä oikeaan hakemistoon tai määriteltävä Java-tulkin *classpath*-komentorivi-parametrin avulla oikea hakemisto.
- Java löytää omassa hakupolussaan olevat pakkaukset automaattisesti.
- Kansion pakkausmääreettömät tiedostot kuuluvat automaattisesti (ilman **package**-määrettä) samaan pakkaukseen.

Pakkauksen käyttöönotto

- Pakkaukseen sijoitetut luokat (tai rajapinnat) voidaan ottaa käyttöön toisessa pakkauksessa “suoraan” kirjoittamalla luokan (tai rajapinnan) nimen alkuun pakkauksen nimi.
  - Näin “viittaamalla” voidaan välttää nimikonfliktit (eri pakkauksissa samanniminen luokka).
  - Otetaan käyttöön esimerkiksi *LinkitettyLista*-luokka *fi.uta.csjola.oope*-pakkauksesta:

    ```
    fi.uta.csjola.oope.LinkitettyLista lista
    = new fi.uta.csjola.oope.LinkitettyLista();
    ```

  - Pakkauksen nimeä käyttämällä lauseita tulee pitkiä.

Pakkauksen käyttöönotto

- Toinen tapa käyttää pakkauksen sisältöä on määritellä tarvittavat luokat tai rajapinnat **import**-lauseilla, jolloin pakkauksen luokkia ja rajapintoja voidaan käyttää tavalliseen tapaan ilman pakkauksen nimeä.
- Yleisesti:

```
// Tietty pakkauksen luokka käyttöön.
import pakkaus.LuokanNimi;
// Pakkaus käyttöön kokonaisuudessaan.
import pakkaus.*;
```

missä **pakkaus** on pakkauksen tai alipakkauksen nimi.

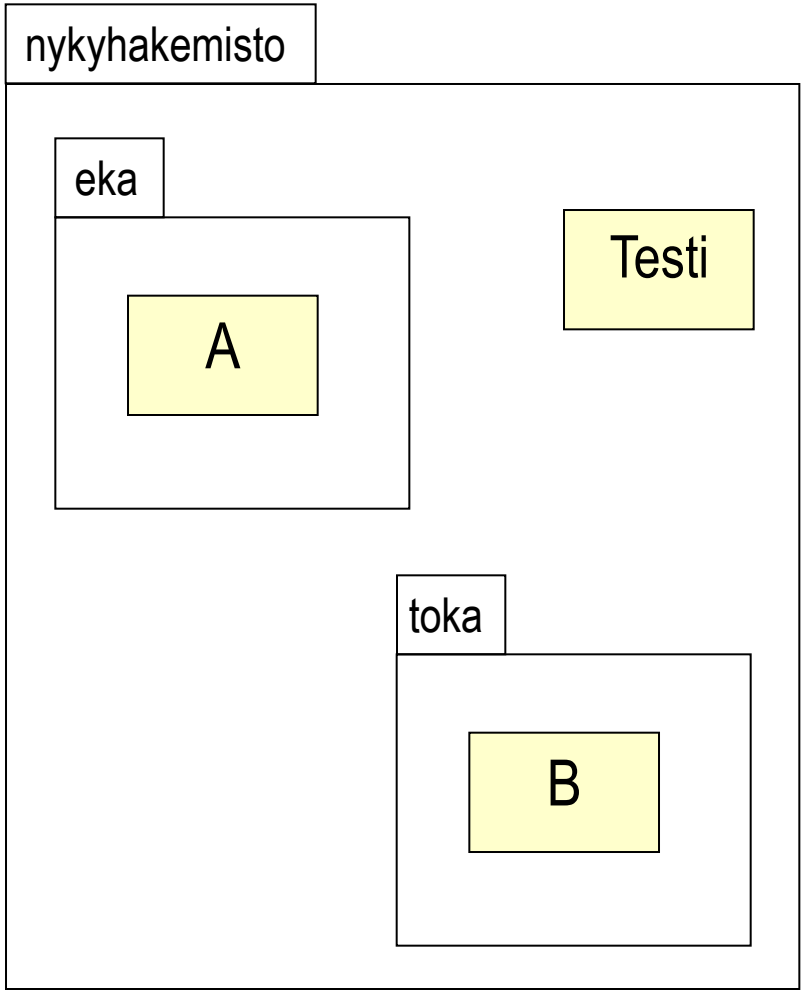
Pakkauksen käyttöönotto

- **import**-lause sijoitetaan mahdollisen **package**-lauseen jälkeen ja ennen **class**- ja **interface**-lauseita.
- **import**-lauseita voi olla useampia.
- Käytännössä kannattaa käyttää **import**-lauseita, koska niiden avulla ohjelmointi on helpompaa ja nopeampaa.

```
package kissankoti;
// Otetaan käyttöön elaimet-
// pakkauksesta Kissa-luokka
// esi-isineen.
import elaimet.*;
// Huonekaluista käyttöön Sohva.
import huonekalut.Sohva;
public class Testi {
   // Käytetään Kissa-luokkaa
   // tuttuun tapaan.
   Kissa harmi = new Kissa();
   ...
}
```

Esimerkki

- nykyhakemisto
  - Testi.java
  - Testi.class
  - eka
    - A.java
    - A.class
  - toka
    - B.java
    - B.class



Esimerkki (eka\A.java)

```
package eka;
public class A {
   public void testi() {
      System.out.println("Testi A");
   }
}
```

- Kääntäminen alihakemistossa *eka*: javac A.java

Esimerkki (toka\B.java)

```
package toka;
import eka.*;
public class B {
   public void testi() {
      A o = new A();
      o.testi();
      System.out.println("Testi B");
   }
}
```

- Kääntäminen alihakemistossa *toka*:
  javac ..\eka\A.java B.java

Esimerkki (Testi.java)

```
import eka.*;   // Otetaan A-luokka käyttöön.
public class Testi {
   public static void main(String[] args) {
      // import-lauseen avulla luokkia voidaan käyttää tutulla tavalla.
      A a = new A();
      // Lauseet ovat luokkia pakkauksen nimellä käyttäen pitempiä.
      toka.B b = new toka.B();
      b.testi();
   }
}
```

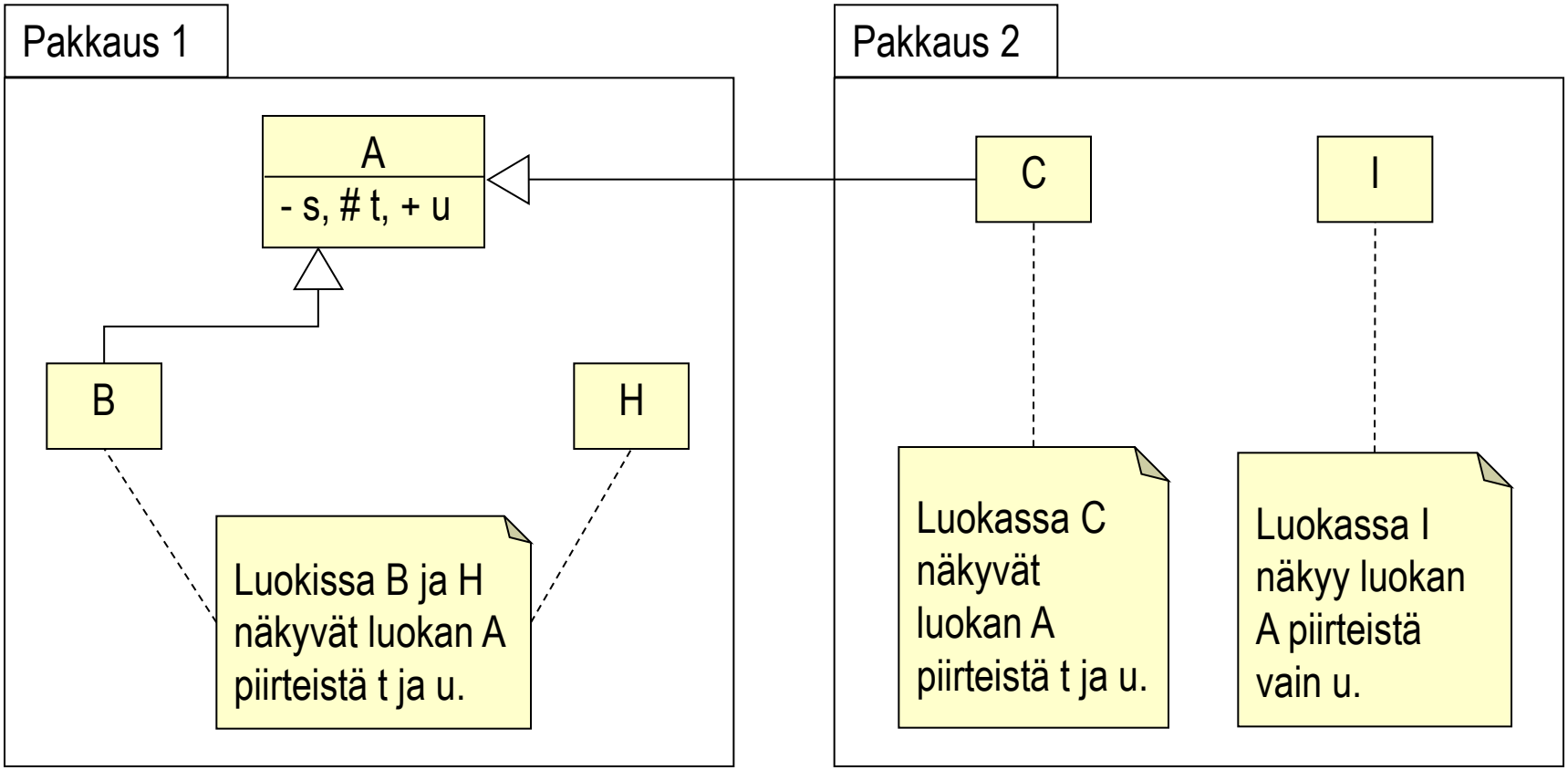
- Kääntäminen nykyhakemistossa: javac Testi.java
- Suoritus nykyhakemistossa: java Testi

Pakkaukset ja näkyvyys

- Pakkaukset rajoittavat luokkien ja niiden piirteiden näkyvyyttä muiden pakkausten luokkiin.
- Ilman näkyvyysmääreitä (niin sanottu friendly-määre) luokka ja sen piirteet eivät näy muihin kuin luokan oman pakkauksen luokkiin.
- **public**-määreellä esitellyn luokan **public**- ja **protected**-määritellyt piirteet ovat käytettävissä muissa pakkauksista.
  - **public**-määreellä esitellyt piirteet näkyvät kaikissa muissa luokissa pakkauksesta riippumatta.
  - **protected**-määreellä esitellyt piirteet näkyvät samassa pakkauksessa oleville luokille sekä luokan jälkeläisille niiden pakkauksesta riippumatta.
- Käytä pakkauksia ja **protected**-määrettä, jos julkisten aksessorien käyttö ei ole järkevää.
  - Pakkaa luokkahierarkia siten, että pakkauksessa on vain hierarkiaan kuuluvia luokkia.

Pakkaukset ja näkyvyys

public class A { // private s, protected t, public u ... }



14. Poikkeukset

Sisällys

- Johdanto.
- Tarkistettavat ja tarkistamattomat poikkeukset.
- Poikkeusten tunnistaminen ja sieppaaminen **try-catch**-lauseella.
- Mitä tehdä siepatulla poikkeuksella?
- Poikkeusten heittäminen.
- *Exception*-luokan metodeja.

Johdanto

- Ohjelman ajon aikana ilmennyt virhe pysäyttää ohjelman ellei virheeseen ole varattu asianmukaisesti.
  - Ohjelman “kaatuminen” ei ole toivottavaa, koska siitä on aina harmia käyttäjälle.
  - Jos virheestä ei voi toipua, tulisi ohjelman kaatumisen kuitenkin tapahtua hallitusti.
- Perinteiset tavat käsitellä ajonaikaisia virheitä:
  - Tunnistetaan ja vältetään, mutta ei reagoida muuten.
  - Tunnistetaan ja vältetään ja virheestä ilmoitetaan metodin paluuarvon avulla, jolloin metodia kutsuva metodi voi tarvittaessa reagoida virheeseen.

Johdanto

- Javassa virheen voidaan antaa myös tapahtua, koska virheen tuottama poikkeus (exception) voidaan tunnistaa ja siepata **try-catch**-lauseella.
  - Poikkeusta ei voida jättää täysin huomiotta: Ohjelman suoritus pysähtyy, mikäli **try-catch** ei ole viimeistään *main*-metodissa.
- Poikkeukset on mallinnettu Javan API:ssa luokiksi, joista osa on jo tuttuja:
  - *NullPointerException*: viitteen oliolle ei ole varattu muistia.
  - *ArrayIndexOutOfBoundsException*: viitataan virheelliseen paikkaan taulukossa.
  - *InputMismatchException*: väärän tyyppinen syöte.

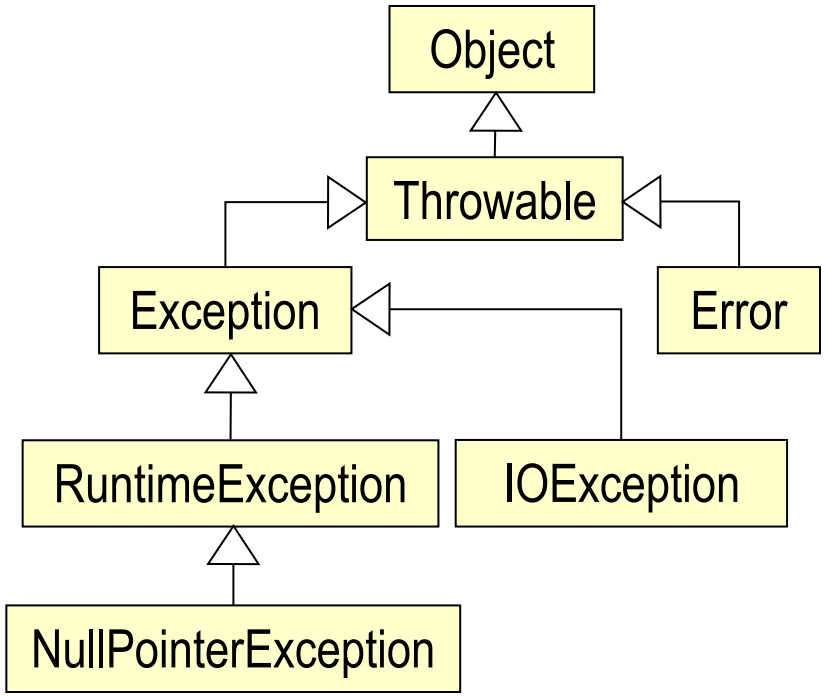
Exception-luokka

- Suuri osa poikkeuksista on *Exception*-luokan jälkeläisiä, jotka jaetaan tarkistamattomiin (unchecked) ja tarkistettaviin (checked) poikkeuksiin sen mukaan onko **try-catch**-lausetta pakko käyttää poikkeuksen ollessa mahdollinen.
  - Käytännössä myös tarkistamattomat poikkeukset on käsiteltävä jollain tavoin **try-catch**-lauseella, jotta ohjelma ei kaatuisi hallitsemattomasti, vaikka on kiistanalaista tulisiko ja voiko tarkistamattomiin poikkeuksiin ylipäätänsä reagoida.
  - Java pakottaa ohjelmoijan kirjoittamaan **try-catch**-lauseen, jos kutsuttava metodi voi palauttaa tarkistettavan poikkeuksen.

Exception-luokka

- *RuntimeException*-luokan ja sen jälkeläisluokkien mallintamat poikkeukset ovat tarkistamattomia.
- *Exception*-luokan muiden jälkeläisten poikkeukset ovat tarkistettavia.
- *Error*-luokan periytymishaara mallintaa vakavat, virtuaalikoneen toimintaan liittyvät virheet, joita ei yleensä ole syytä käsitellä.

- Pieni osa *Exception*-luokan sisältävästä luokka-hierarkiasta:



Try-catch-lause

- Poikkeuksen mahdollisesti tuottavat lauseet suljetaan **try**-lohkoon mielellään niin, ettei koodin luettavuus kärsi.
- Poikkeuksen tapahtuessa **catch**-lohkot tarkastellaan järjestyksessä ylhäältä alas.
- Poikkeus siepataan ensimmäiseen poikkeuksen tyyppiä vastaavaan **catch**-lohkoon.
  - Lohkon otsikossa annettu viite liittyy olioon, jonka on viitteen tai sen jälkeläisluokan tyyppiä.

```
try {
   // Poikkeuksen mahdollisesti
   // tuottavat lauseet.
}
catch (Tyyppi1 e) {
   // Tyypin1 poikkeusten käsittely.
}
…
catch (TyyppiN e) {
   // TyypinN poikkeusten käsittely.
}
```

Try-catch-lause

- *Exception*-tyyppisellä parametrilla varustettu **catch**-lohko sieppaa minkä tahansa poikkeuksen.
  - Käytettävä varovasti, koska voi estää virheiden havaitsemisen ja mahdollistaa huolimattoman poikkeusten käsittelyn.
- **Finally**-lohko on valinnainen ja sen lauseet suoritetaan aina poikkeuksen tapahtumisesta riippumatta.
- Ohjelman suoritus jatkuu **try-catch**-lausetta seuraavasta lauseesta.

```
try {
   // Poikkeuksen mahdollisesti
   // tuottavat lauseet.
}
catch (Tyyppi1 e) {
   // Tyypin1 poikkeusten käsittely.
}
…
catch (Exception e) {
   // Siepataan viimeistään tässä.
}
finally {
   // Tämä lohko suoritetaan aina.
}
```

Try-catch-lause

- Javan versiosta 1.7 alkaen **catch**-lohkoon voi määritellä siepattavaksi useampia virheitä putkimerkin avulla.
- Toinen uudistus on resursoitu **try-catch**-lause, jota voidaan käyttää **finally**-lohkon tapaan varattujen resurssien vapauttamiseen ne sulkemalla.
  - Resurssi varataan luomalla olio *AutoCloseable*-rajapinnan toteuttavasta luokasta.

```
...
catch (Tyyppi1 | Tyyppi2 e) {
   // Molemman tyyppisten
   // poikkeusten käsittely.
}
...
try (resurssien varaaminen) {
   ...
}

// Lukija suljetaan ennen mahdollisen
// virheen sieppausta.
try (Scanner sc
= new Scanner(new File("in.txt"))) {
   ...
}
catch (...) { ...
```

Scanner & InputMismatchException

```
import java.util.*;  // Scanner-luokka täällä.
public class InputMismatchExceptionTest {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in); // Liitetään oletussyötevirtaan.
    int luku; boolean syoteOK;
    do { // Luetaan, kunnes saadaan kokonaisluku.
      try { // Tämän lohkon aiheuttamat poikkeukset siepataan.
        syoteOK = true;
        System.out.println("Anna luku:");
        luku = sc.nextInt(); // Väärän tyyppinen syöte => poikkeus.
      }
      catch (InputMismatchException e) { // Siepataan poikkeus.
        System.out.println("Virheellinen syöte!");
        sc.nextLine(); // Poistetaan virheellinen syöte.
        syoteOK = false;
      }
    }
    while (!syoteOK);
  }
}
```

Mitä tehdä siepatulla poikkeuksella?

- Siepattu poikkeus voidaan käsitellä paikallisesti siten, että siitä tiedetään korkeintaan paluuarvon kautta kutsuvassa metodissa.
- Siepattu poikkeus voidaan myös heittää **throw**-lauseella kutsuvan metodin käsiteltäväksi.
  - Siepatun poikkeuksen heittäminen koskee lähinnä tarkistettavia poikkeuksia, jotka on joka tapauksessa pakko siepata.
  - Tarkistamattomat poikkeukset siirtyvät automaattisesti kutsuvalle metodille.
  - Esimerkiksi:

```
catch (IOException e) {
    throw e; // Heitetään tarkistettava poikkeus.
}
```

Poikkeusten heittäminen

- Siepattuun virheeseen on myös mahdollista reagoida heittämällä uusi poikkeus **catch**-lohkossa.
- Esimerkiksi:

```
catch (FileNotFoundException e) {
   throw new IllegalArgumentException("Virheellinen nimi.");
}
```

- Poikkeuksen heittäminen on hyödyllistä myös parametrien tarkistuksessa.
  - Virheellinen arvo tunnistetaan perinteisesti **if**-lauseella, mutta virheeseen reagoidaan paluuarvon sijasta heittämällä poikkeus, joka on usein *IllegalArgumentException*-tyyppiä.

Poikkeusten heittäminen

- Metodin otsikossa on määriteltävä varatun sanan **throws**-avulla metodin heittämät tarkistettavat poikkeukset.
  - Tarkistamattomia poikkeuksia ei ole pakko määritellä, mutta kommenteissa on hyvä mainita myös poikkeukset, jotka palautuvat automaattisesti ilman heittämistä.
- Yleisesti:

  määreet metodinNimi(**parametrilista**) ***throws Poikkeus1*,**
  ... , Poikkeus*N* { ...
- Kutsupaikassa on syytä varautua poikkeuksiin **try-catch**-lauseella, kun kutsuttavan metodin otsikossa esiintyy **throws**-määre.
  - Java “pakottaa” kutsuvan metodin tähän, jos metodi heittää tarkistettavan poikkeuksen.

Poikkeusten heittäminen

- Rakentaja ei voi palauttaa virhekoodeja, mutta rakentajakin voi heittää tarvittaessa poikkeuksen:

```
public Koira(String v) throws NullPointerException {
  if (v == null)
     throw new NullPointerException();
  else
     ...
}
```

Taulukon ja poikkeusten käsittelyä

```
/* Lasketaan taulukon t paikoissa [a, b] olevien alkioiden summa.
 */
public static double laskeValinsumma(double[] t, int a, int b) {
   double summa = 0;
   try { // Tässä lohkossa voi tapahtua poikkeuksia.
      for (int i = 0; i < b - a + 1; i++)
         summa += t[a + i];
   } // Indeksiarvot a ja/tai b olivat virheellisiä.
   catch (ArrayIndexOutOfBoundsException e) {
      summa = Double.NaN;
   } // Taulukolle ei oltu varattu muistia (t == null).
   catch (NullPointerException e) {
      summa = Double.NaN;
   }
   return summa;
}
```

Taulukon ja poikkeusten käsittelyä

```
/* Lasketaan taulukon t paikoissa [a, b] olevien alkioiden summa.
 * Taulukon käsittelyyn liittyvistä virheistä kerrotaan IllegalArgumentException-
 * tyyppisellä poikkeuksella, jonka maininta otsikosta on valinnaista (unchecked).
 */
public static double laskeValinsumma(double[] t, int a, int b)
throws IllegalArgumentException {
  try {
    double summa = 0;
    for (int i = 0; i < b - a + 1; i++)
      summa += t[a + i];
    return summa;
  }
  catch (Exception e) {
    throw new IllegalArgumentException(e);
  }
}
```

Taulukon ja poikkeusten käsittelyä

```
public static void main(String[] args) {
   double summa1, summa2;
   try { // Siepataan laskeValinsumma-metodin mahdolliset poikkeukset.
      double[] t1 = new double[] { 1.1, 2.2, 3.3, 4.4, 5.5 };
      // Virheellinen indeksi tuottaa ArrayIndexOutOfBoundsException-poikkeuksen.
      summa = laskeValinsumma(t1, 0, 5);
   }
   // Taulukolle ei oltu varattu muistia (t == null) tai indeksiarvo(t) oli(vat) virheellisiä.
   catch (IllegalArgumentException e) {
      // java.lang.IllegalArgumentException: java.lang.ArrayIndexOutOfBoundsException: 5
      System.out.println(e);
   }
}
```

Exception-luokan metodeja

- Perii joitakin hyödyllisiä metodeja yliluokastaan:
  - *printStackTrace*: Tulostaa standardivirhevirtaan (oletusarvoisesti näytölle) virtuaalikoneen luoman virheilmoituksen. Ilmoitus on lähes sama kuin Javan ajonaikaisen virheen vuoksi tulostama virheilmoitus.
  - *getStackTrace*: Palauttaa *StackTraceElement*-luokan viitteen, jonka kautta poikkeuksen tietoja voidaan tutkia tarkemmin.
  - *toString*: Palauttaa poikkeusta kuvailevan merkkijonon.
- Ohjelmoija voi periä omia poikkeusluokkia.

Taulukon ja poikkeusten käsittelyä

```
/* Lasketaan taulukon t paikoissa [a, b] olevien alkioiden summa.
 */
public static double laskeValinsumma(double[] t, int a, int b) {
   double summa = 0;
   try { // Tässä lohkossa voi tapahtua poikkeuksia.
      for (int i = 0; i < b - a + 1; i++)
         summa += t[a + i];
   } // Indeksiarvot a ja/tai b olivat virheellisiä.
   catch (ArrayIndexOutOfBoundsException e) {
      summa = Double.NaN; System.out.println(e.toString());
   } // Taulukolle ei oltu varattu muistia (t == null).
   catch (NullPointerException e) {
      summa = Double.NaN; e.printStackTrace();
   }
   return summa;
}
```

java.lang.ArrayIndexOutOfBoundsException: 5

```
java.lang.NullPointerException
        at TaulukkoPoikkeusDemo3.laskeValinsumma(TaulukkoPoikkeusDemo3.java:16)
        at TaulukkoPoikkeusDemo3.main(TaulukkoPoikkeusDemo3.java:33)
```

15. Ohjelmoinnin tekniikkaa

Sisällys

- For-each-rakenne.
- Lueteltu tyyppi **enum**.
- Override-annotaatio.
- Geneerinen ohjelmointi.

For-each-rakenne

- For-rakenteen variaatio taulukoiden ja muiden kokoelmien silmukoimiseen:

```
for (muuttuja : kokoelma) {
    ...
}
```

missä muuttuja on samaa tyyppiä kuin kokoelman alkio.

- Java sijoittaa muuttujaan kullakin kierroksella kokoelman seuraavan alkion arvon.

- Esimerkki: lasketaan yksiulotteisen taulukon alkioiden summa eri tavoin.

```
int[] luvut = { 1, 2, 3, 4, 5 };
...
for (int i = 0; i < luvut.length; i++)
  summa1 += luvut[i];
…
for (int luku : luvut)
  summa2 += luku;
```

For-each-rakenne

- Esimerkki: lasketaan kaksiulotteisen taulukon alkioiden summia.

```
int[][] luvut = { { 1, 2, 3 }, { 4, 5, 6 } };
   ...
   for (int i = 0; i < luvut.length; i++)
      for (int j = 0; j < luvut[i].length; j++)
         summa1 += luvut[i][j];
   ...
   // Kaksiulotteisen taulukon rivit ovat yksiulotteisia taulukoita.
    for (int[] rivi : luvut)
       for (int luku : rivi)
          summa2 += luku;
```

Lueteltu tyyppi

- Lueteltu tyyppi (enumeration) on tietotyyppi, jossa tyypin arvoille kiinnitetään nimet luettelemalla kaikki tyypin arvot.
  - Esimerkiksi pelikorttien maat ovat pata, risti, hertta ja ruutu.
- Javassa luokkatyypin erikoistapaus.
  - Määritellään avainsanalla **enum**.
  - Periytyy *Enum*-luokasta, joka on *Object*-luokan aliluokka.
  - Arvot automaattisesti julkisia luokkavakioita (**public static final**).
- Paljon rajoitteita luokkatyyppiin verrattuna.
  - Olioiden luominen ei ole mahdollista.
  - Lueteltua tyyppiä ei voi periä toisesta luetellusta tyypistä.
  - Rakentajia ei voi julkaista **public**- tai **protected**-määreillä.

Lueteltu tyyppi

- Tällä kurssilla lueteltu tyyppi
  - esitellään hyvin yksinkertaisessa muodossa ja
  - sijoitetaan luokan tapaan omaan tiedostoonsa.
- // Pelikorttien maat lueteltuna tyyppinä (Maa.java).
  // Tyypin määrittelyyn riittää jo pelkästään tyypin otsikko
  // ja arvojen esittely ne tyypin rungossa luettelemalla.
  public enum Maa {
  // Maa-tyyppisellä tunnuksella voi olla vain
  // jokin näistä arvoista.
  PATA, RISTI, HERTTA, RUUTU;
  }

Lueteltu tyyppi

```
// Viikonpäivät lueteltuna tyyppinä (Viikonpaiva.java).
public enum Viikonpaiva {
   // Luetellun tyypin rakentajaa kutsutaan arvojen esittelyn yhteydessä.
   MA("maanantai"), TI("tiistai"), KE("keskiviikko"), TO("torstai"),
   PE("perjantai"), LA("lauantai"), SU("sunnuntai");
   private String nimi;  // Viikonpäivän nimi.
   private Viikonpaiva(String uusiNimi) throws IllegalArgumentException {
      if (uusiNimi == null)
         throw new IllegalArgumentException("Virheellinen nimi!");
      nimi = uusiNimi;
   } ...
}
```

Lueteltu tyyppi

- Kullakin luetellulla tyypillä on erikoismetodit:
  - // Palauttaa luetellun tyypin E arvot taulukossa.

    public static E[] values()
  - // Muuntaa annetun merkkijonon luetellun tyypin E arvoksi.

    // Heittää IllegalArgumentException-poikkeuksen,

    // jos muunnos ei onnistu.

    public static E valueOf(String name)
- Nämä metodit eivät periydy *Enum*-luokasta, vaan kääntäjä muodostaa ne.

Override-annotaatio

- Annotaatiot (annotation) ovat kommenttien tapaan ohjelmaan liittyvää metatietoa (tietoa tiedosta).
- Annotaatioilla voidaan ohjailla Java-kääntäjän toimintaa.
- Annotaation tunnus alkaa @-merkillä.
  - Esimerkiksi @Override, @Deprecated ja @SuppressWarnings.
- Ohjelmoija lisää annotaatiot itse lähdekoodiin.
- Tällä kurssilla hyödyllisin annotaatio on Java-kielen oma *Override*-annotaatio joka pakottaa kääntäjän ilmoittamaan virheestä, jos annotoitu metodi ei korvaa yliluokan metodia.
  - Estää pienellä vaivalla kirjoitus- ja ajatusvirheitä.

Override-annotaatio

- *Override*-annotaatio annetaan omalle rivilleen välittömästi ennen metodin otsikkoa.
- Esimerkki: *Kissa*-luokka ei käänny, koska sen metodi on määritelty korvattavaksi *Override*-annotaatiolla, mutta metodin otsikko on kirjoitettu vahingossa väärin (pitäisi olla *onkolso*).

```
@Override
public boolean onkoiso() {
   return paino() > 10;
}
```

Geneerinen ohjelmointi

- Geneerinen ohjelmointi (generic programming, generics) tarkoittaa muun muassa ohjelmointitekniikkaa, jossa algoritmeja toteutetaan yleisesti sitomatta niitä käsiteltävän tiedon tyyppiin.
- Tunnetuin geneerisyyden mahdollistava mekanismi lienevät C++-kielen kaavaimet (template).
- Geneerisyys lisättiin Javaan vasta versiossa 1.5.
- Javassa geneerisyys liittyy erityisesti kokoelmaluokkien (esimerkiksi *Vector* ja *ArrayList*) käsittelyyn.
  - Algoritmien yleinen toteutus mahdollista periytymistä hyödyntäen.

Geneerinen ohjelmointi

- Geneerisesti ohjelmoiden saadaan turvallisempaa koodia:
  - lisätyyppitarkistuksien avulla havaitaan virheitä helpommin,
  - ohjelmassa tarvitaan vähemmän tyyppimuunnoksia ja
  - voidaan kirjoittaa yleiskäyttöisempiä algoritmeja.
- Geneerisyyttä käytettäessä käännös ei onnistu, jos esimerkiksi kokoelmaa käytetään vaarallisesti.
- Geneerisyyden tuottamat lisätarkistukset eivät ole käytettävissä ohjelman ajon aikana.
- Geneerinen ohjelmointi ei ole pakollista – Java-kääntäjä kuitenkin varoittaa, jos kokoelmaa käsitellään perinteisellä (vaarallisemmalla) tavalla.

Geneerinen ohjelmointi

- Tällä kurssilla riittää tietää kuinka voi halutessaan ottaa geneerisyyden käyttöön kulmasuljenotaatiolla.
- Yleisesti: LuokanNimi<TyypinNimi>

  missä geneerinen tyyppi voi olla vain viitetyyppiä.
- Esimerkiksi *Vector*-luokkaa voi käyttää turvallisemmin luomalla vektorin seuraavasti:

```
// Taulukkolistaan voidaan sijoittaa kaiken tyyppisiä viitteitä.
ArrayList<Object> v = new ArrayList<Object>(3, 2);
// Ainoastaan Integer-tyyppiset viitteet kelpaavat.
ArrayList<Integer> v = new ArrayList<Integer>(3, 2);
```

16. Javan omat luokat

Sisällys

- Johdanto.
- *Object*-luokka:
  - *toString*-, *equals*-, *clone*- ja *getClass*-metodit.
- *Comparable*-rajapinta:
  - *compareTo*-metodi.
- *Vector*- ja *ArrayList*-luokat.

Javan omat luokat

- Javan omat luokat muodostavat ohjelmointi-rajapinnan (Application Programming Interface, API), jonka avulla voidaan suorittaa mitä erilaisempia tehtäviä.
- API-luokat on jaettu pakkauksiin, joista *java.lang* on aina käytettävissä automaattisesti.
- Tässä pakkauksessa ovat kielen keskeisimmät luokat kuten esimerkiksi *Object*, *String* ja *Double*.

Javan omat luokat

- Java tarjoaa perustyypeille niin sanotut kääreluokat (wrapper classes), jotka ovat perustyypin viitetyyppisiä vastineita: **double** ja *Double*, **int** ja *Integer* jne.
  - Java muuntaa implisiittisesti alkeistyypin sitä vastaavan kääreen tyypiksi.
- *java.lang*-pakkauksesta löytyy myös *Math*-luokka, jossa joitakin matemaattisia operaatioita ja vakioita.
- Tiedostojenkäsittelyyn tarvitaan *java.io*-pakkauksen luokkia. (Tiedostot on esitelty Lausekielinen ohjelmointi -kurssin luentorungossa.)

Javan omat luokat

- *java.util*-pakkauksesta löytyvät esimerkiksi *Random*-, *Scanner*-, *Formatter*, *LinkedList*-, *Vector*-, *ArrayList*- ja *Stack*-luokat.
- Javan versiokohtainen API-dokumentaatio on luettavissa Internetissä. Esimerkiksi version 1.8 dokumentit löytyvät osoitteesta:

  http://docs.oracle.com/javase/8/docs/api/index.html
- Linkki kurssin verkkosivuilla.

Object-luokka

- Java-kielen luokkahierarkian juuriluokka.
  - Automaattisesti itse ohjelmoitujen luokkien juuriluokaksi.
- Näin kaikilla omilla luokilla muun muassa metodit:
  - *toString*: tuotetaan olion merkkijonoesitys,
  - *equals*: olioiden vertailu,
  - *clone*: luodaan olion kopio ja
  - *getClass*: olion luokan tutkiminen “metaolion” avulla.
- Metodit voidaan korvata omassa luokassa.

toString-metodi

- **public** String toString()
  - Palauttaa oletusarvoisesti merkkijonon, jossa on olion luokan nimi, @-merkki ja olion hajautuskoodi.
  - Javan API-dokumentaatiossa suositellaan, että jokainen *Object*-luokan jälkeläinen korvaa tämän metodin.

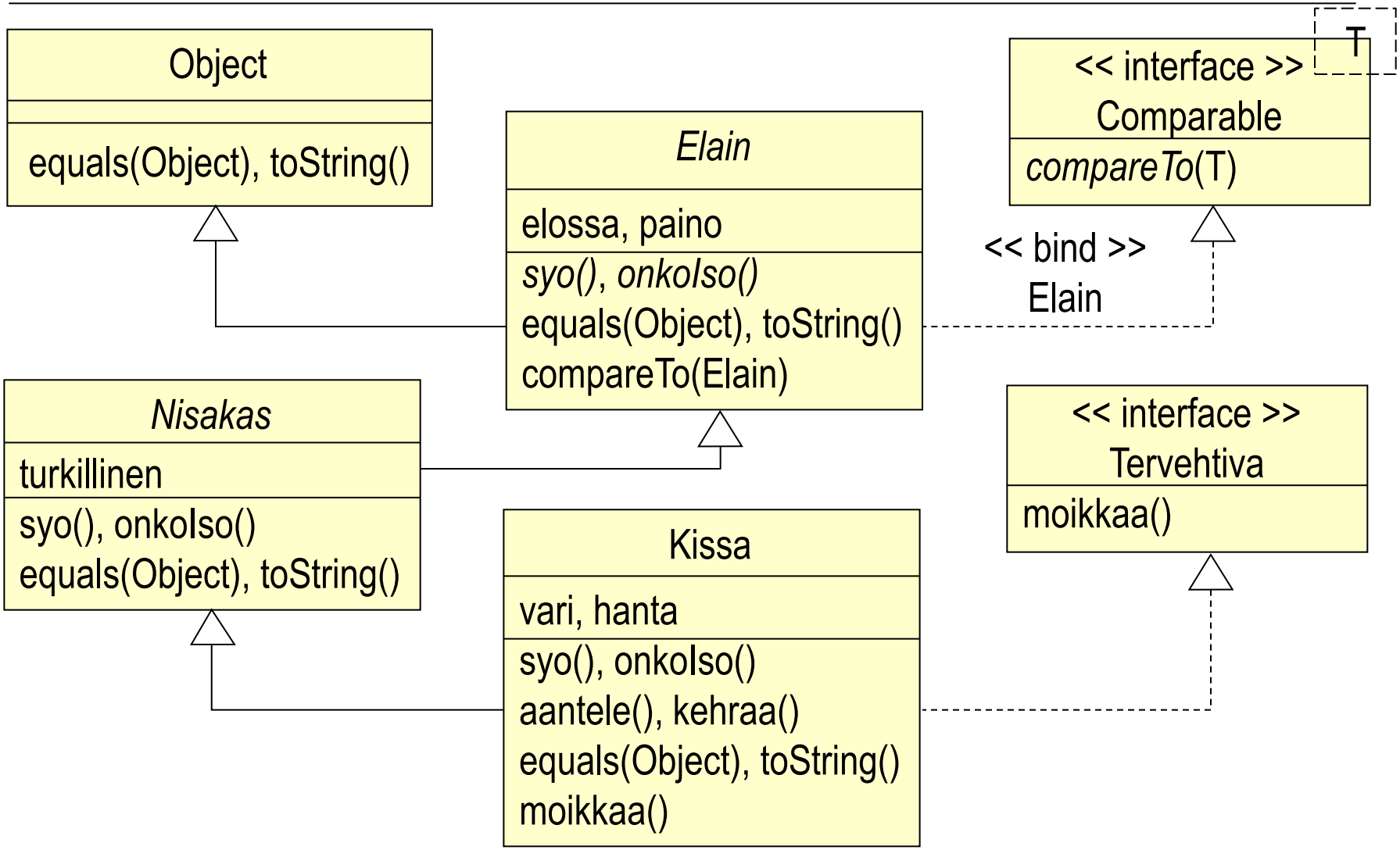
- /* Kissa-luokassa korvattu toString-metodi,
  * josta kutsutaan yliluokan korvausta.
  */

```
public String toString() {
   return super.toString() + EROTIN + vari + EROTIN + hanta;
}
```

equals-metodi

- **public boolean** equals(Object obj)
  - Tutkitaan ovatko oliot “samat” (paluuarvo **true**).
  - Olkoon viitteet x != **null**, y != **null** ja z != **null**. Tällöin:
    - x.equals(x) == **true** // refleksiivisyys
    - x.equals(y) == y.equals(x) // symmetrisyys
    - Jos x.equals(y) == **true** ja y.equals(z) == **true**,
      niin x.equals(z) == **true** // transitiivisuus
  - Vertailee oletusarvoisesti viitteitä.
  - Korvattava siten, että metodissa vertaillaan olioiden tietoja (eli attribuuttien arvoja).
    - Määrittele parametri aina *Object*-tyyppiseksi, koska muun tyyppisellä parametrilla metodi kuormitetaan.
    - Korvattaessa tulisi korvata myös *hashCode*-metodi.

Object, eläin, nisäkäs ja kissa



clone-metodi

- **protected** Object clone()
  throws CloneNotSupportedException
  - Luo kopion viestin saavasta oliosta.
  - Palauttaa *Object*-tyyppisen viitteen kopioon.
  - Metodia kutsuvan luokan tai sen esi-isän täytyy toteuttaa tyhjä *Cloneable*-rajapinta, jotta metodia voi käyttää.
  - Metodi on usein korvattava, koska se pintakopioi (shallow copy) viitetyyppiset attribuutit.
  - Metodin korvaus on annettava **public**-määreellä.
- Olion kopiointi *clone*-metodilla on monimutkaista.
  - Jotkin lähteet suosittelevat metodin välttämistä.
  - Vaihtoehdoissa myös omat ongelmansa.

getClass-metodi

- *Class*-luokka on luokkia mallintava “metaluokka”.
  - Jokaiseen olioon liittyy “metaolio”, joka sisältää tietoja sen luokasta (dynaaminen luokka).
  - *Class*-luokka ja *java.lang.reflect*-pakkaus mahdollistavat ohjelman rakenteen tutkimisen ja muokkaamisen sen ajon aikana (reflektio).
- **public final** Class<?> getClass()
  - Palauttaa olion metaolion (*Class*-luokan ilmentymä).
  - // Luodaan olio ja asetetaan viite siihen ja sen metaolioon.

    String s = **new** String(); Class m = s.getClass();

getClass-metodi

- Joitakin *Class*-luokan metodeja.
  - *getName*- ja *getSimpleName*-metodit palauttavat olion luokan nimen hieman eri muodoissa.
  - *isInstance*-metodi on **instanceof**-operaattorin vastine.
    - Paluuarvo **true**, jos metodin parametriin *p* liittyvän olion luokka *P* ja metaolion kuvaaman olion *m* luokka *M* ovat yhteensopivia (*P* <: *M*).
    - Jos *P* <: *M*, niin lauseke *p* **instanceof** *M*, on **true**.
    - **instanceof**-operaattoria joustavampi menetelmä olion luokan tunnistamiseen.

Comparable-rajapinta (java.lang)

- **public interface** Comparable<T>
  - Rajapinta on generisoitu: *T* on geneerinen tyyppinimi.
  - Esim. Toteutetaan rajapinta siten, että voidaan vertailla kaiken tyyppisiä viitteitä: ... **implements** Comparable<Object> ...
- **public abstract int** compareTo(T o)
  - Rajapinnan ainoa metodi, jonka toteuttajan on kiinnitettävä *T*.
  - Palauttaa -1, kun viestin saava olio on “pienempi kuin” parametrina saatu olio.
  - Palauttaa 0, kun viestin saava olio on “yhtä suuri kuin” parametrina saatu olio.
  - Palauttaa 1, kun viestin saava olio on “suurempi kuin” parametrina saatu olio.

Vector-luokka

- Viitetyyppisten alkioiden taulukko, jonka koko voi kasvaa tai pienetä.
- Lisätilaa varataan automaattisesti aina tarvittaessa.
- Alkioihin voi viitata indeksin avulla.
- Erittäin hyödyllinen apuluokka, joka on käytettävissä kaikissa Javan versioissa (≥ 1.0).
  - *ArrayList*-luokka modernimpi vaihtoehto.
- Sijaitsee *java.util*-pakkauksessa $\Rightarrow$ omassa ohjelmassa tarvitaan **import**-lause.

Vector-luokka

```
import java.util.Vector;
public class VektoriTesti {
   public static void main(String[] args) {
      int ALKLKM = 7;
      Vector v = new Vector(3, 2);
      for (int i = 0; i < ALKLKM; i++) {
         Integer o = new Integer(i);
         v.add(o);
         System.out.println(v.size() + "\t" + v.capacity());
      }
      while (v.size() > 0) {
         Integer o = (Integer)v.remove(0);
         System.out.println(v.size() + "\t" + v.capacity());
      }
   }
}
```

1	3
2	3
3	3
4	5
5	5
6	7
7	7
6	7
5	7
4	7
3	7
2	7
1	7
0	7

Huom! Kääntäjä varoittaa, koska kokoelman alkioiden tyyppiä ei ole annettu.

ArrayList-luokka

```
import java.util.ArrayList;
public class TaulukkolistaTesti {
   public static void main(String[] args) {
      int ALKLKM = 7;
      ArrayList<Integer> l = new ArrayList<Integer>();
      for (int i = 0; i < ALKLKM; i++) {
         Integer o = new Integer(i);
         l.add(o);
         System.out.println(l.size());
      }
      while (l.size() > 0) {
         Integer o = l.remove(0);
         System.out.println(l.size());
      }
   }
}
```

```
1
2
3
4
5
6
7
6
5
4
3
2
1
0
```

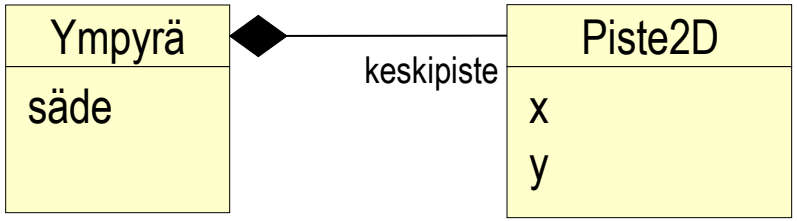
Huom! Kokoelman alkioiden tyyppi on kiinnitetty geneerisellä määreellä.

17. Kooste

Kooste

- Kooste (aggregation) on luokkien *A* ja *B* välinen suhde, joka tarkoittaa “A on B:n osa” tai “A kuuluu B:hen”.
  - Koostesuhteessa olevat luokat eivät yleensä ole periytymissuhteessa.
- Kooste toteutetaan sijoittamalla *A*-tyyppinen attribuutti luokkaan *B*.

- Esim. Ympyrän keskipiste voidaan ajatella ympyrän osaksi.




```
public class Ympyra {
   private Piste2D keskipiste;
   private double sade;
   ...
}
```

Osaoliot

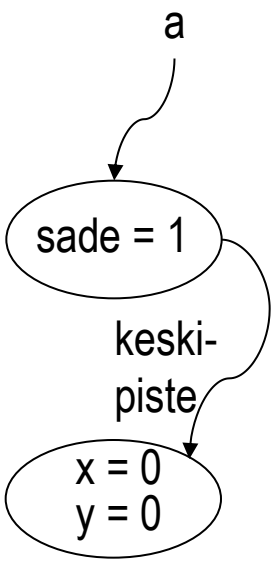
- Koostesuhteen seurauksena ohjelman ajonaikana luokan *B* oliolla on tyypillisesti *A*-luokan osaolio, jonka luominen ja tuhoaminen on usein isäntäolio vastuulla.
  - Javassa riittää yleensä luominen.

- Esim. Luodaan keskipiste rakentajassa.

```
public Ympyra() {
   // Ympyrä oletusarvoisesti
   // yksikköympyrä.
   keskipiste = new Piste2D(0, 0);
   sade = 1;
}
```

- Esim. Olioiden luominen.

```
// Ympyrää luotaessa
// luodaan myös osaolio.
Ympyra a = new Ympyra();
```

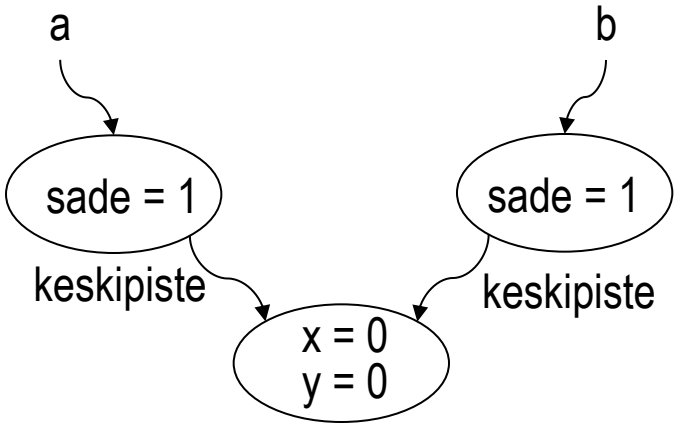


Pintakopiointi

- Koostesuhteen tuottamat osaoliot ovat haasteellisia, kun oliosta täytyy luoda kopio.
- *Object*-luokan *clone*-metodi kopioi oletusarvoisesti alkeistyyppisten attribuuttien arvot, mutta viitetyyppisten parametrien osalta tapahtuu vain pintakopiointi (shallow copy), jossa kopioidaan viitteet, jolloin alkuperäisen ja kopioidun olion viitetyyppiset attribuutit osoittavat samoihin osaolioihin.

- Esim. Ympyrän pintakopiointi.

```
// Luodaan ympyrä.
Ympyra a = new Ympyra();
// Kopioidaan pinnallisesti clone-
// metodilla ja asetetaan kopioon
// uusi viite, jolloin isäntäolioilla
// on yhteinen osaolio.
Ympyra b = (Ympyra)a.clone();
```



Pintakopiointi

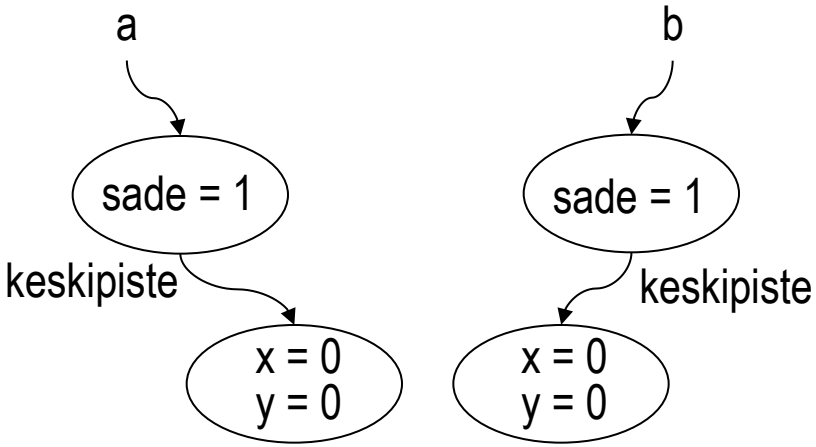
- Pintakopiointi ei ole yleensä ongelma, jos osaoliot on luotu muuttumattomista (immutable) luokista, joiden attribuuttien arvot voi asettaa vain olion luomisen yhteydessä.
  - Alkuperäinen isäntäolio ja sen kopio voivat huoletta viitata samaan osaolioon, koska osa-olion tilaa ei voi muuttaa sen luomisen jälkeen.
  - Esimerkiksi *String* on muuttumaton luokka.
- Koska muuttumaton luokka on yksinkertainen ja helppo käyttää, joissakin lähteissä suositellaan kaikkien luokkien toteuttamista mahdollisimman pitkälti muuttumattomina.
  - Oope-kurssilla tähän ei pyritä erityisesti.

Syväkopiointi

- Kopion tulisi olla alkuperäisestä oliosta riippumaton, jos osaolioina on muuttuvien (mutable) luokkien edustajia.
  - Muuttuvasta luokasta luodun olion tilaa voidaan muuttaa luomisen jälkeen esimerkiksi asetusmetodeilla.
  - Esim. *StringBuffer* on muuttuva.
- Riippumattomuus saavutetaan syväkopioinnilla (deep copy), jossa kopio-oliolle luodaan uudet osaoliot, joihin kopioidaan alkuperäisten osaolioiden tiedot.

- Esim. Ympyrän syväkopiointi.

```
// Luodaan ympyrä.
Ympyra a = new Ympyra();
// Syväkopioidaan ympyrä.
// (clone-metodi on korvattu nyt
// syväkopioivalla tavalla.)
Ympyra b = (Ympyra)a.clone();
```



Syväkopiointi

- Javan *clone*-metodin käyttö niin pinta- kuin syväkopiointiin on vaikeaa ja vaatii tarkkuutta.
  - Joissakin lähteissä *clone*-metodia kehotetaan välttämään
- Vaihtoehtoisilla lähestymis-tavoilla on omat vahvuutensa ja heikkoutensa.
- Kopiorakentajat lienevät yksinkertaisin tapa kopioida olio Javassa.
  - Ongelmana osaolioiden luokkien tunnistus.
- Kopiorakentaja saa parametrinaan viitteen kopioitavaan olioon.
- Rakentaja kopioi parametrinsa tiedot pinnallisesti tai syvällisesti.
  - Ennen kopiointia kutsutaan esi-isien kopiorakentajia ketjussa.
- Esim. Ympyrän kopiorakentaja.

```
public Ympyra(Ympyra y) {
  // Yliluokan kopiorakentajan kutsu.
  super(y);
  if (y instanceof Ympyra) {
    sade(y.sade()); …
}
```

Syväkopiointi

- Syväkopiointia hyödynnetään myös lukumetodeissa.
  - Luokan lukumetodeissa voidaan estää muuttuvista luokista luotujen osaolioiden tilan muuttaminen luokan ulkopuolella syväkopioimalla osaolio ja palauttamalla viite tähän kopioon.
- Esim. Ympyrän keskipisteen lukumetodi.

```
public Piste2D keskipiste() {
  // Luodaan uusi olio siten, että sen arvot ovat samat kuin osaolion.
  // ja liitetään kopio-olioon viite.
  Piste2D osanKopio = new Piste2D(keskipiste.x(), keskipiste.y());
  // Palautetaan viite kopioon.
  return osanKopio;
}
```

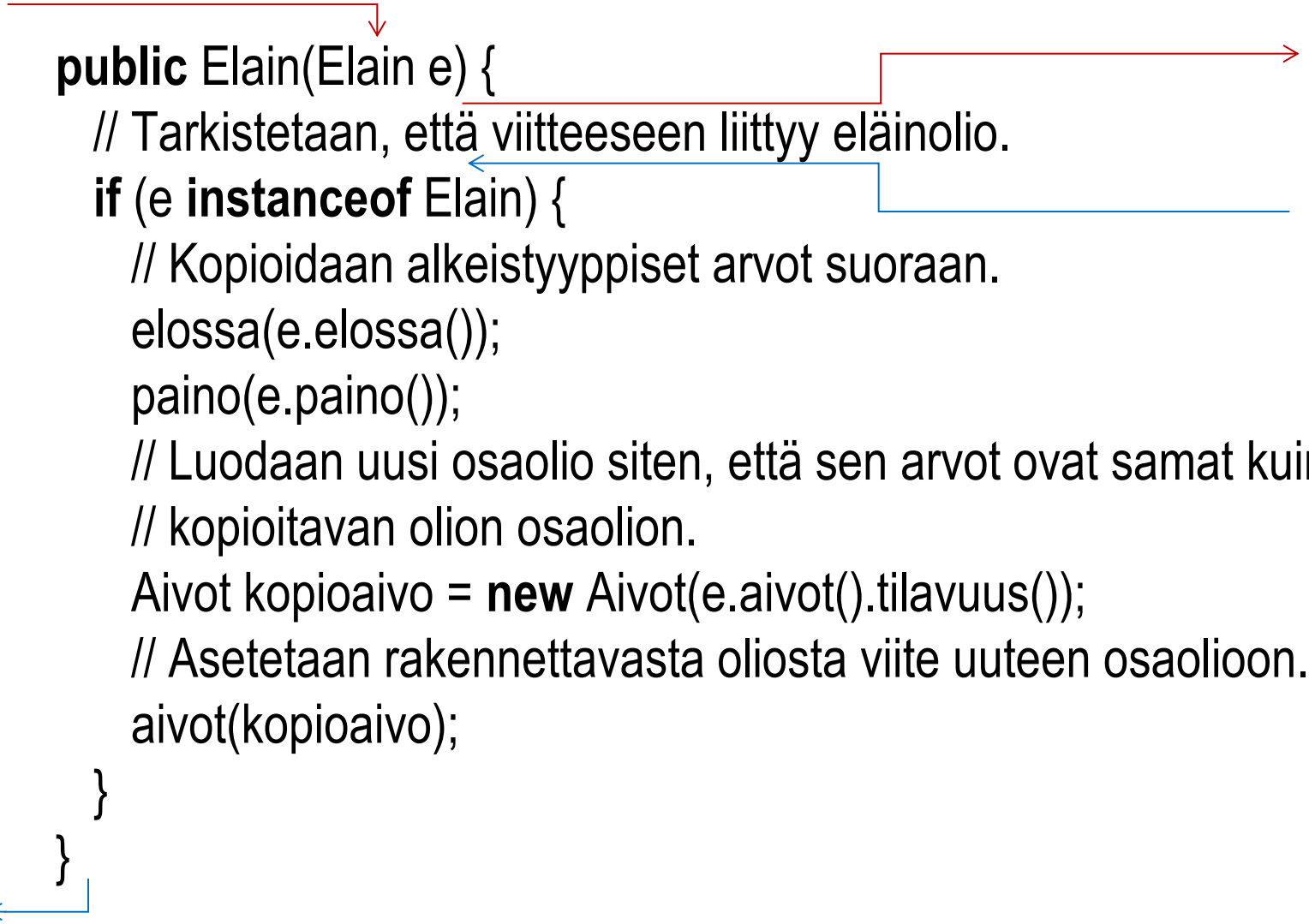
Eläimen syväkopiointi

```
Kissa hassu = new Kissa(true, 2, new Aivot(15), "harmaa", "kippura");
Kissa hassu2 = new Kissa(hassu);
…

public Kissa(Kissa k) {
   // Kutsutaan yliluokan kopiorakentajaa.
   super(k);
   // Muuttumattomille olioille
   // riittää pintakopiointi.
   vari(k.vari());
   hanta(k.hanta());
}

public Nisakas(Nisakas n) {
   // Kutsutaan yliluokan kopiorakentajaa.
   super(n);
   // Kopioidaan nisäkkään attribuutti.
   turkillinen(n.turkillinen());
}
```

Eläimen syväkopiointi



```
public Elain(Elain e) {
   // Tarkistetaan, että viitteeseen liittyy eläinolio.
   if (e instanceof Elain) {
      // Kopioidaan alkeistyyppiset arvot suoraan.
      elossa(e.elossa());
      paino(e.paino());
      // Luodaan uusi osaolio siten, että sen arvot ovat samat kuin
      // kopioitavan olion osaolion.
      Aivot kopioaivo = new Aivot(e.aivot().tilavuus());
      // Asetetaan rakennettavasta oliosta viite uuteen osaolioon.
      aivot(kopioaivo);
   }
}
```

18. Abstraktit tietotyypit

Sisällys

- Johdanto abstrakteihin tietotyyppeihin.
- Pino ja jono.
- Linkitetty lista.
- Pino linkitetyllä listalla toteutettuna.

Johdanto

- Javan omat tietotyypit ovat jo tuttuja: alkeistietotyypit (esimerkiksi **int** ja **double**) ja viitetietotyypit (luokat, rajapinnat ja taulukot).
- Abstrakti tietotyyppi (abstract data type) on tietotyypin kuvaus, jossa ei oteta kantaa tyypin toteutukseen tai edes toteutuskieleen.
- Ympäristö tuntee vain liittymän, jolla abstraktia tietotyyppiä käytetään.

Johdanto

- Abstrakti tietotyyppi koostuu
  - operaatioista: tyypin liittymä ympäristöön ja
  - tietorakenteesta: tyypin sisäinen tiedonesitystapa.
- Tietorakenne (data structure) voitava vaihtaa ilman, että operaatioiden otsikkoja tarvitsee muuttaa.
- Abstraktin tietotyypin käyttäjä ei ole kiinnostunut tietorakenteesta vaan nimenomaan tietotyypin tarjoamista operaatioista.

Johdanto

- Esim. Abstraktin tietotyypin *Aika* käyttäjää ei niinkään kiinnosta ajan esitystapa tietotyypin sisällä, vaan se kuinka voidaan vaikkapa laskea kahden ajankohdan välinen aika.
- Usein abstraktin tietotyypin tietorakenne sisältää useita tietoja, jolloin yksittäistä tietoa sanotaan alkioksi.

Johdanto

- Abstraktin tietotyypin idea on hyvin samankaltainen kuin luokkien, mutta abstraktit tietotyypit ovat olio-ohjelmointia vanhempi keksintö.
- Olio-ohjelmointi tarjoaa mainion tavan toteuttaa abstrakteja tietotyyppejä
  - operaatiot metodeiksi ja
  - tietorakenne **private**-tyyppiseksi attribuutiksi.
- Abstrakti tietotyyppi voidaan siis toteuttaa näppärästi kapseloituna luokkana.

Pino

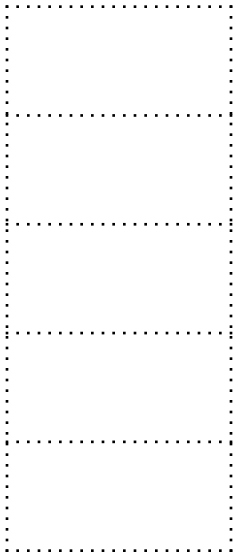
- Pino (stack) on abstrakti tietotyyppi, jossa
  - lisättävä alkio menee aina pinon päälle ja
  - poistettava alkio otetaan aina pinon päältä.
- Last-In-First-Out (LIFO) periaate.
- Lisäykset ja poistot alkio kerrallaan.
- Analogiana voi ajatella korttipakkaa pasianssissa:
  - Pakkaan voidaan lisätä kortti vain päälle (ei pakan väliin tai pohjalle).
  - Pakasta voidaan ottaa vain päällimmäinen kortti (ei pakan välistä tai pohjalta).

Pinon operaatioita

- **lisää**(Alkio a): lisää alkion *a* pinon päälle, mikäli pinoon mahtuu vielä uusi alkio.
- **poista**(): palauttaa ja poistaa pinon päällimmäisen alkion, mikäli pinossa on alkioita.
- **koko**(): alkioiden lukumäärä (≥ 0) pinossa.
- **onkoTyhjä**(): palauttaa totuusarvon **true**, jos pino on tyhjä ja totuusarvon **false**, jos pinossa ainakin yksi alkio.
- **ylin**(): palauttaa päällimmäisen alkion sitä poistamatta, mikäli pinossa on alkioita.
- Tietorakenne: esimerkiksi taulukko tai linkitetty lista.

Esimerkki

Luodaan tyhjä,
maksimissaan
5 alkiota
sisältävä pino.



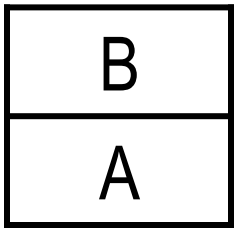
lisää(A)

koko() = 1

onkoTyhjä() = **false**

lisää(B)

koko() = 2

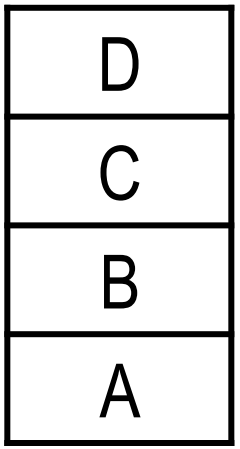


lisää(C)

koko() = 3

lisää(D)

koko() = 4

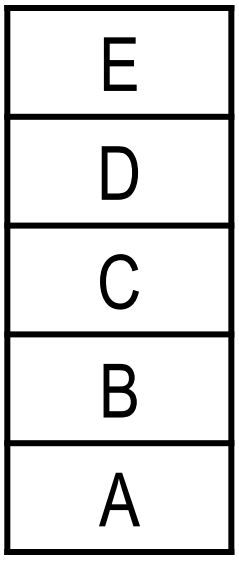


Esimerkki

lisää(E)

Pino täynnä:

koko() = 5

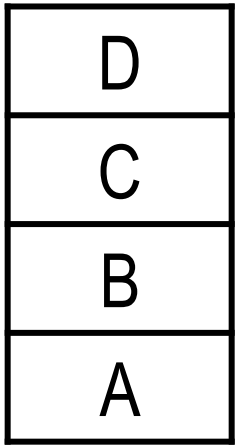


poista() = E

koko() = 4

ylin() = D

koko() = 4



poista() = D

koko() = 3

poista() = C

koko() = 2

poista() = B

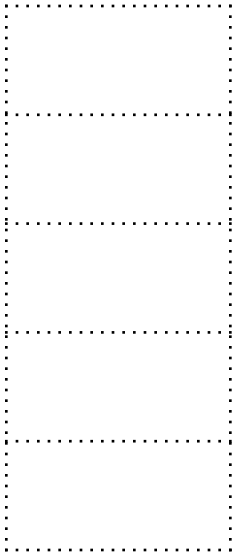
koko() = 1

poista() = A

Pino tyhjä:

koko() = 0

onkoTyhjä() = **true**



Jono

- Jono (queue) on abstrakti tietotyyppi, jossa
  - lisättävä alkio menee aina jonon loppuun ja
  - poistettava alkio otetaan aina jonon alusta.
- First-In-First-Out (FIFO) periaate.
- Lisäykset ja poistot alkio kerrallaan.
- Esim. Ruokalan jono, jossa ensiksi tullutta asiakasta palvellaan ensiksi ja uudet asiakkaat sijoittuvat kiltisti jonon loppuun.

Jonon operaatioita

- **lisää**(Alkio a): lisää alkion *a* jonon loppuun, mikäli jonossa on vielä tilaa.
- **poista**(): palauttaa ja poistaa ensimmäisen alkion jonosta, mikäli jono ei ole tyhjä.
- **koko**(): alkioiden lukumäärä (≥ 0) jonossa.
- **onkoTyhjä**(): palauttaa totuusarvon **true**, jos jono on tyhjä ja totuusarvon **false**, jos jonossa ainakin yksi alkio.
- **keula**(): palauttaa jonon ensimmäisen alkion sitä poistamatta, mikäli jono ei ole tyhjä.
- Tietorakenne: esimerkiksi taulukko tai linkitetty lista.

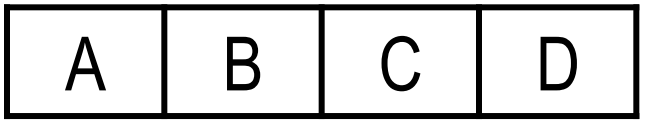
Esimerkki

- Luodaan tyhjä, maksimissaan 5 alkiota sisältävä jono.

- **lisää**(A), **koko**() = 1, **onkoTyhjä**() = **false**

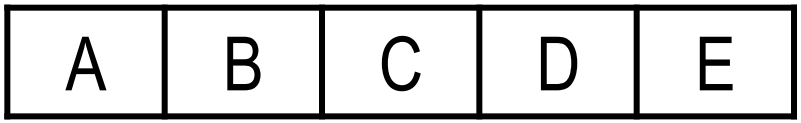
- **lisää**(B), **koko**() = 2

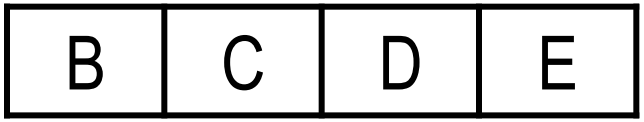
- **lisää**(C), **koko**() = 3, **lisää**(D), **koko**() = 4



Esimerkki

- **lisää**(E),
  Jono täynnä: **koko**() = 5
- **poista**() = A, **koko**() = 4,
  keula() = B, **koko**() = 4
- **poista**() = B, **koko**() = 3,
  poista() = C, **koko**() = 2,
  poista() = D, **koko**() = 1
- **poista**() = E,
  Jono tyhjä: **koko**() = 0,
  onkoTyhjä() = **true**





Linkitetty lista

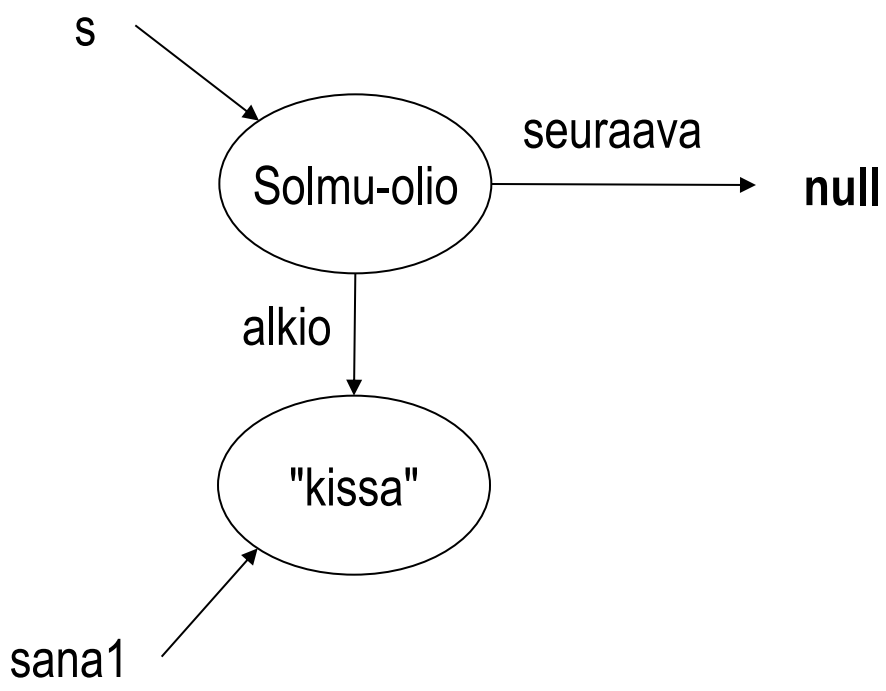
- Linkitetty lista (linked list) on tietorakenne, joka koostuu toisiinsa viittaavista solmuista (node).
- Taulukon tapaan listan alkioilla on järjestys ja alkioihin voidaan yleensä myös viitata nollasta alkavan indeksin avulla.
- Taulukosta poiketen linkitetty lista on dynaaminen tietorakenne, joka käyttää aina muistia täsmälleen tarvittavan määrän: Alkioiden lisääminen ja poistaminen muuttaa listan kokoa.
- Yksinkertaisin muoto yhteen suuntaan linkitetty lista.

Yhteen suuntaan linkitetty lista (YSLL)

- Solmu = tietoalkio + viite seuraavaan solmuun.

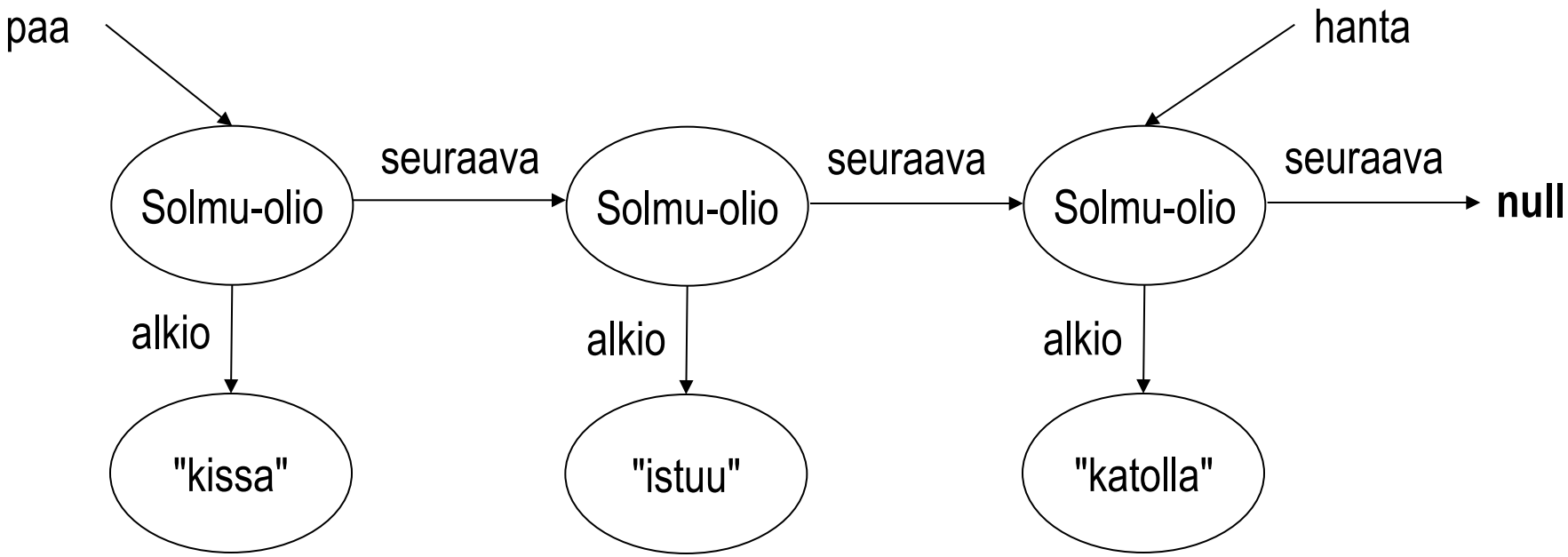
- **public class** Solmu {
  private Object alkio;
  private Solmu seuraava;
  public Solmu(Object uusi) {
  alkio = uusi;
  seuraava = **null**;
  }
  ...
  }

- String sana1 = **new** String("kissa");
  Solmu s = **new** Solmu(sana1);



Yhteen suuntaan linkitetty lista (YSLL)

- Pää (head) on listan ensimmäinen solmu.
- Häntä (tail) on listan viimeinen solmu.
- Viimeinen viite osoittaa **null**-arvoon.
- Esim. Kolmisolmuinen YSLL, jonka tietoalkiot ovat merkkijonoja.



Yhteen suuntaan linkitetty lista (YSLL)

- Operaatiot määritellään lähteestä riippuen hieman eri tavoin.
- Toteutukseenkin useita lähestymistapoja: Esimerkiksi käytetäänkö vartijasolmuja vai ei?
- Nyt pyritään mahdollisimman yksinkertaiseen ratkaisuun.

YSLL:n operaatiot

- **lisääAlkuun**(Alkio a):
  Lisää listalle uuden pään, joka sisältää alkion *a*.
- **lisääLoppuun**(Alkio a):
  Lisää listalle uuden hännän, joka sisältää alkion *a.*
- **lisää**(int indeksi, Alkio a):
  Lisää annettuun listan kohtaan uuden solmun, joka sisältää alkion *a*.
- **poistaAlusta**():
  Poistaa listan pään ja palauttaa pään alkion.
- **poistaLopusta**():
  Poistaa listan hännän ja palauttaa hännän alkion.
- **poista**(int indeksi):
  Poistaa solmun annetusta kohdasta ja palauttaa poistetun solmun alkion.

YSLL:n operaatiot

- **alkio**(int indeksi):

  Palauttaa annetussa kohdassa olevan solmun alkion.
- **korvaa**(int indeksi, Alkio a)

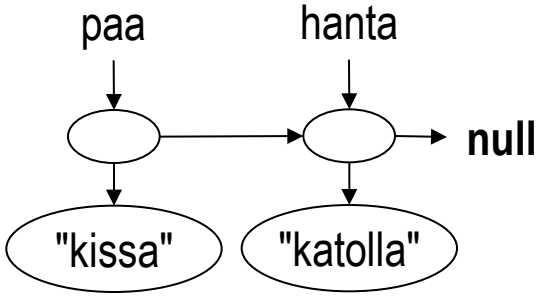
  Korvaa annetussa kohdassa olevan solmun tietoalkion annetulla alkiolla *a*.
- **koko**():

  Palauttaa listan solmujen lukumäärän.
- **onkoTyhjä**():

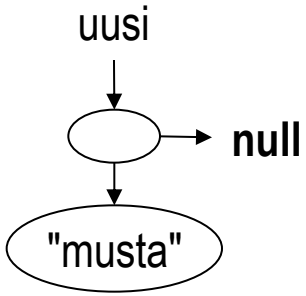
  Palauttaa totuusarvon **true**, mikäli listassa ei ole solmuja, ja totuusarvon **false**, mikäli listassa on yksi tai useampi solmu.

Listan alkuun lisääminen

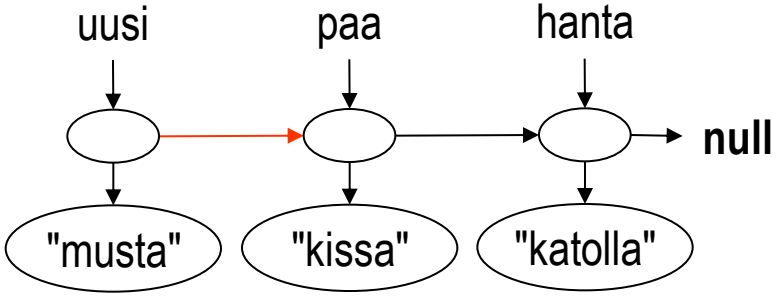
- Lista ennen lisäystä:



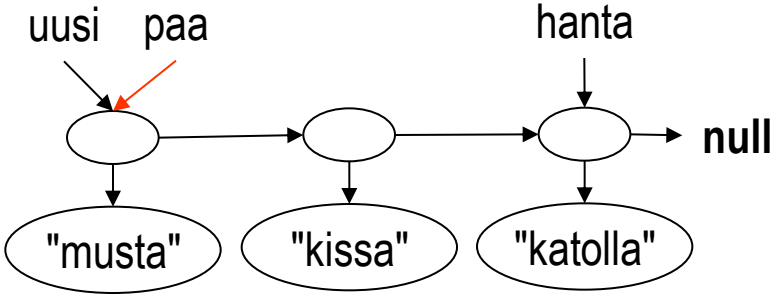
- // Luodaan uusi solmu.
  Solmu uusi = **new** Solmu(alkio);



- // Uusi solmu osoittamaan päähän.
  uusi.seuraava(paa);

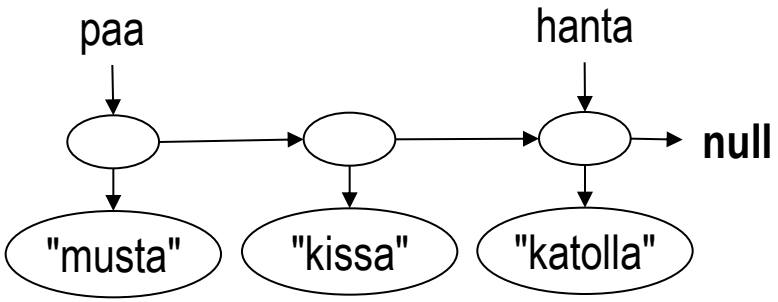


- // Uusi solmu listan pääksi.
  paa = uusi;

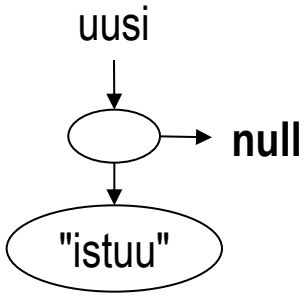


Listan keskelle (paikkaan 2) lisääminen

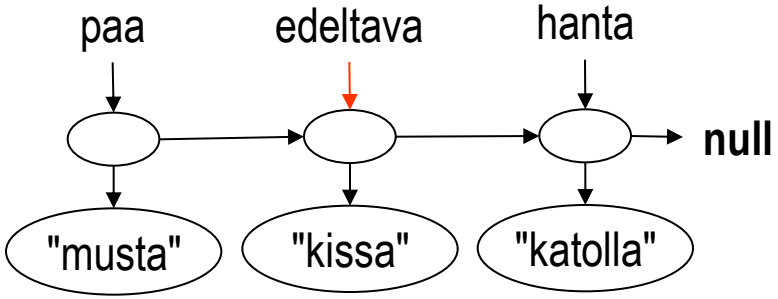
- Lista ennen lisäystä:



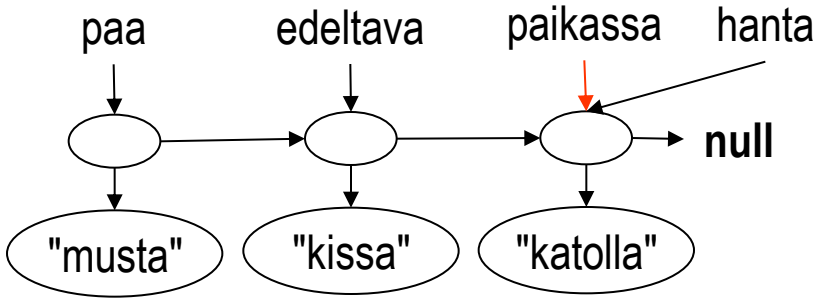
- // Luodaan uusi solmu.
  Solmu uusi = **new** Solmu(alkio);



- // Haetaan edeltävä solmu.
  Solmu edeltava = haeSolmu(paikka-1);

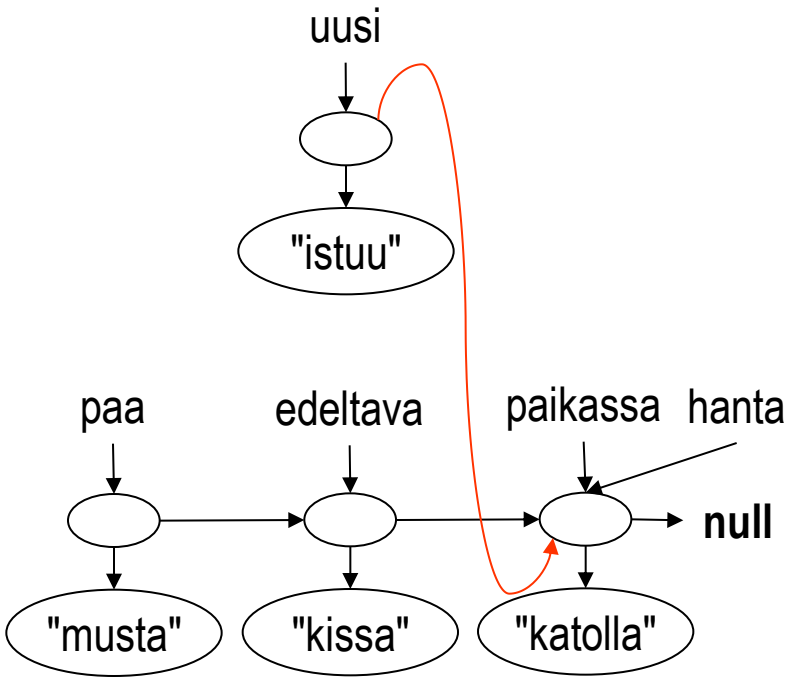


- // Paikassa oleva solmu.
  Solmu paikassa = edeltava.seuraava();

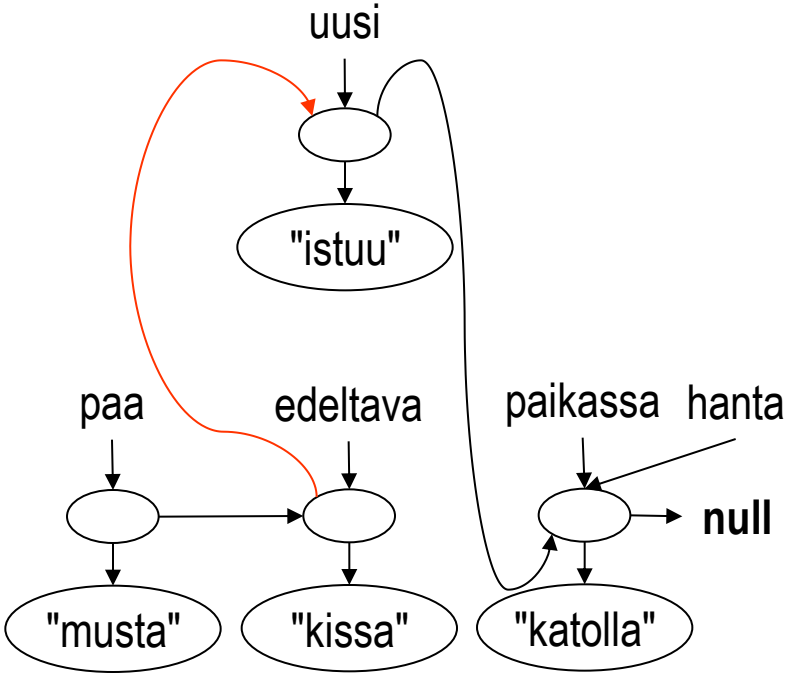


Listan keskelle (paikkaan 2) lisääminen

- // Asetetaan uusi solmu
  // viittaamaan paikan solmuun.
  uusi.seuraava(paikassa);

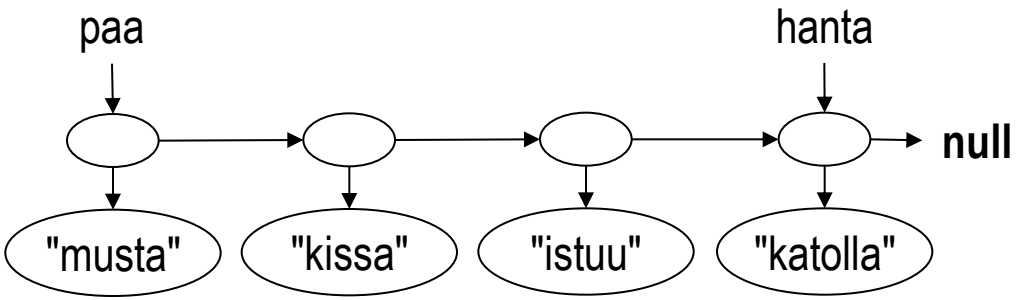


- // Asetetaan edellinen solmu
  // viittaamaan uuteen solmuun.
  edeltava.seuraava(uusi);

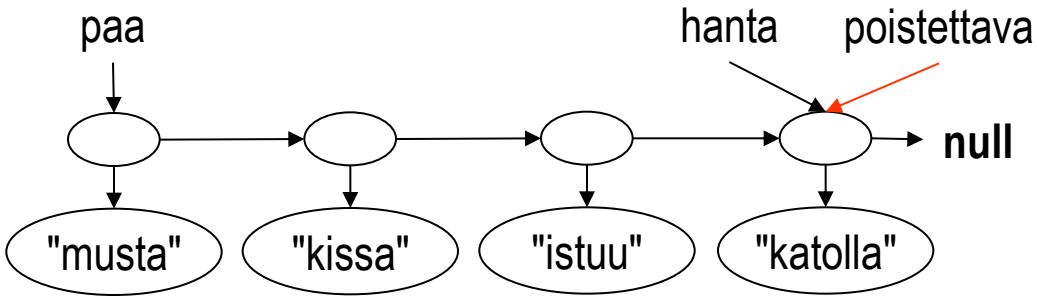


Listan lopusta poistaminen

- Lista ennen poistoa:

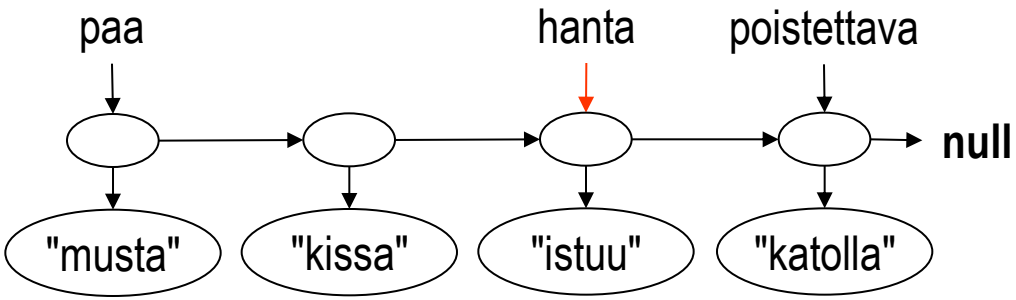


- // Asetetaan apuviite häntään, jotta poistettavaa solmua ei hukata.
  Solmu poistettava = hanta;

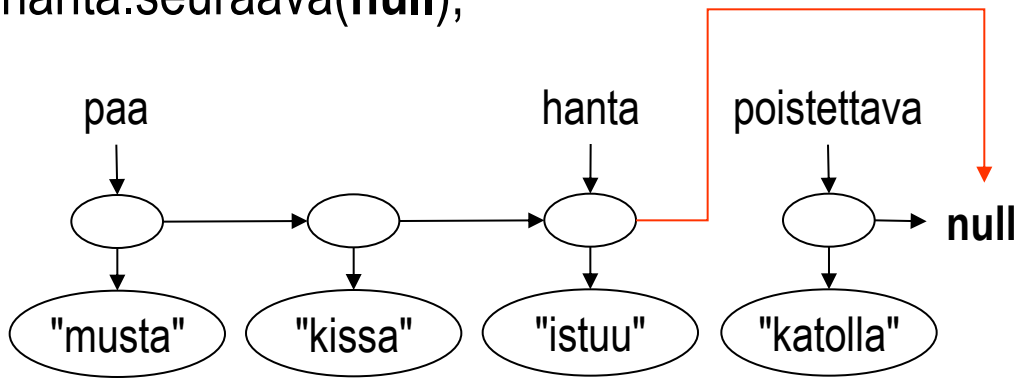


Listan lopusta poistaminen

- // Edellisestä solmusta uusi häntä.
  hanta = haeSolmu(koko - 2);

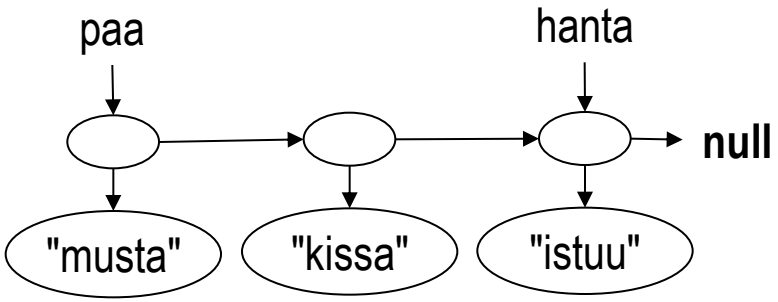


- // Poistetaan viite vanhaan häntään.
  hanta.seuraava(**null**);

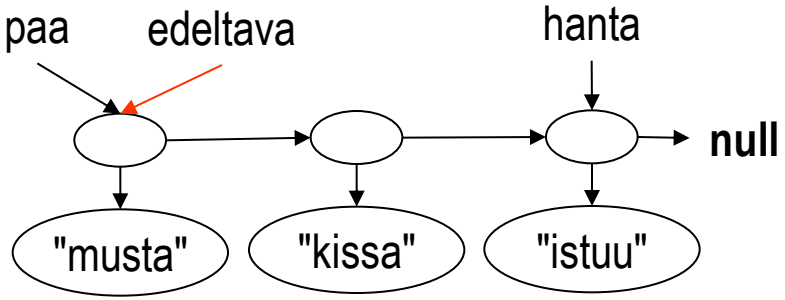


Listan keskeltä (paikasta 1) poistaminen

- Lista ennen poistoa:

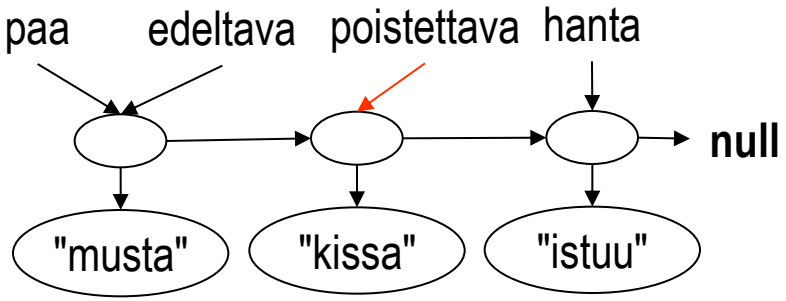


- // Haetaan poistettavaa solmua edeltävä solmu.
  Solmu edeltava = haeSolmu(paikka - 1);

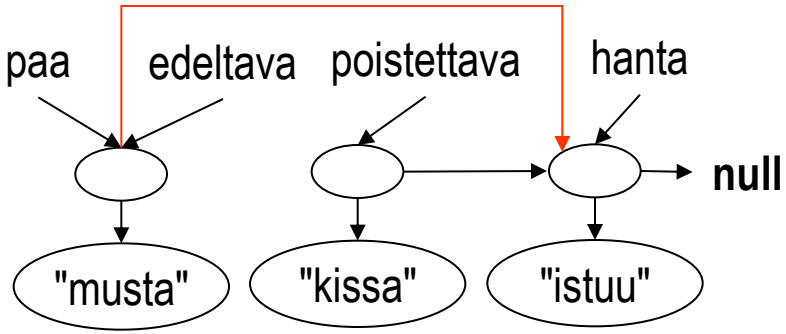


Listan keskeltä (paikasta 1) poistaminen

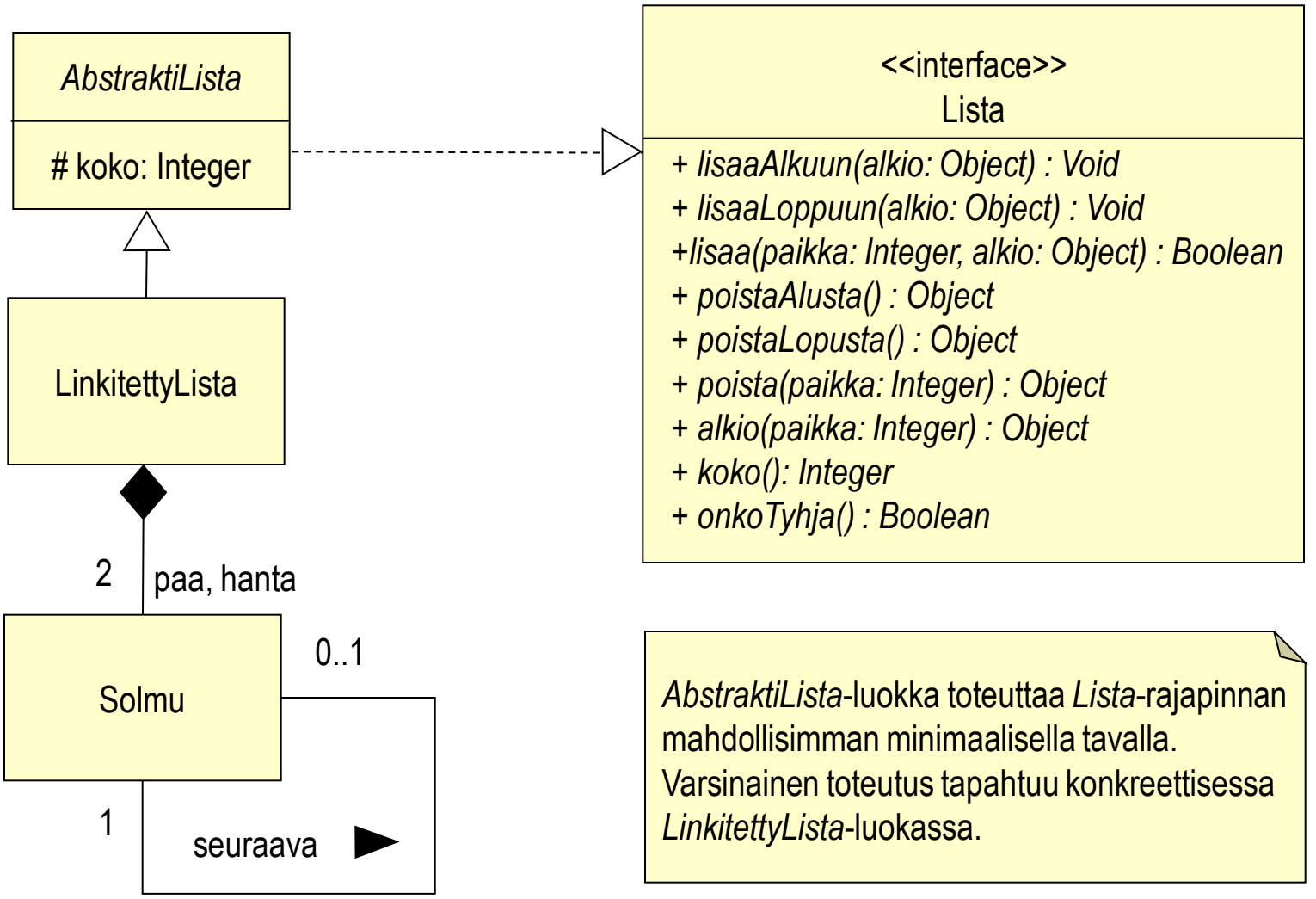
- // Poistettava solmu.
  Solmu poistettava = edeltava.seuraava();



- // Asetetaan edellinen solmu viittaamaan poistettavaa solmua
  // seuraavaan solmuun, jolloin poistettava solmu linkitetään pois.
  edeltava.seuraava(poistettava.seuraava());



LinkitettyLista-luokka



AbstraktiLista-luokka toteuttaa *Lista*-rajapinnan mahdollisimman minimaalisella tavalla. Varsinainen toteutus tapahtuu konkreettisessa *LinkitettyLista*-luokassa.

19. Unified Modeling Language (UML)

Perustuu Kai Koskimiehen Oliokirjaan ja aikaisempaan luentomateriaaliin.

Sisällys

- Johdanto.
- Luokkakaavio:
  - Luokkasymboli, attribuutit ja metodit.
  - Suhteet:
    - Assosiaatiot:
      - tavallinen assosiaatio,
      - assosiaatio ajonaikana,
      - refleksiivinen assosiaatio
      - koosteassosiaatio ja aito kooste.
    - Riippuvuussuhde.
    - Periytymissuhde ja rajapinta.

Johdanto

- Unified Modeling Language (UML) on mallinnuskieli eli täsmällinen menetelmä reaalimaailman abstrahoimiseen.
- Boochin, Rumbaugin ja Jacobsonin mallinnuskielien synteesi.
- Nykyisin Object Management Groupin (OMG) kehittämä standardi, joka kokoaa yhteen useita graafisia mallinnuskieliä, joita kutsutaan kaaviotyypeiksi.

Johdanto

- UML:ia ei ole sidottu mihinkään tiettyyn ohjelmiston suunnittelumenetelmään.
- Omimmillaan kuitenkin olioperustaisen ohjelmiston-kehityksen apuvälineenä.
- Kattaa hyvin moninaisia tarpeita, mutta toisaalta laajuutensa vuoksi UML on monimutkainen ja raskas menetelmä.

Johdanto

- Järjestelmän toimintaa voidaan kuvata korkealla tasolla käyttötapauskaavioilla.
- Useita kaavioita järjestelmän rakenteen ja käyttäytymisen mallintamiseen.
- Kaikkien kaaviotyyppien esittely ei ole kurssin puitteissa mahdollista.
- Tarkastellaan lähemmin vain luokkakaaviota, joka on rakennekaavioista keskeisin.
- Esimerkkinä muun muassa ajoneuvojen vuokrausjärjestelmä.

Johdanto

- Kaaviot koostuvat peruselementeistä ja niiden välisistä suhteista.
- Elementit ovat geometrisia kuvioita.
- Elementeillä usein myös sisäinen rakenne (esimerkiksi luokan attribuutit ja metodit).
- Suhteet ovat elementtejä yhdistäviä viivoja.
- Suhteisiin voidaan liittää tarkempaa informaatiota (esimerkiksi nimi, kertautuminen, roolit).

Johdanto

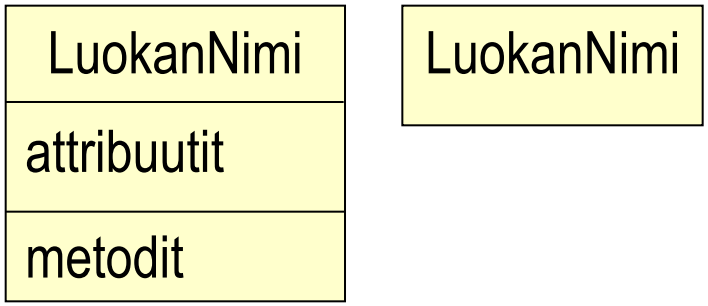
- UML:n graafinen ulkoasu ei ole tarkasti määrätty.
- Kaavioita piirretään myös vaihtelevalla tarkkuudella.
  - Olioperustaisen ohjelmistonkehityksen analyysivaiheessa käytetään paljon yleisempää esitystä kuin suunnitteluvaiheessa.
- Yksityiskohdat sivuutetaan nyt, koska tavoitteena on vain tutustua luokkakaavioihin karkealla tasolla.
- Huomaa, että harjoitustyöhön liittyy hieman tarkempi attribuuttien ja metodien esittely.

Luokkakaavio

- Luokat substantiivilla esitettäviä käsitteitä.
  - Ominaisuudet attribuutteja.
  - Toiminnot metodeja.
- Vain keskeisimmät käsitteet mallinnetaan luokiksi.
- Samoja piirteitä sisältäville luokille tehdään yliluokka, jonne yhteiset piirteet siirretään.
- Luokalla tulee olla yksikäsitteinen vastuu:
  - Esimerkiksi *Lista*-luokka säilöö tietoja ja *Käyttöliittymä* huolehtii ihmisen ja koneen vuorovaikutuksesta.
  - Jos vastuita useampia, niin luokka jaetaan osiin.
- Tarpeettomia yhteyksiä luokkien välillä vältettävä.

Luokkakaavio

- Luokkakaavio (class diagram, static structure diagram) kuvaa järjestelmään kuuluvia luokkia ja niiden välisiä suhteita.
- Tärkein järjestelmän rakenteen mallinnusväline.
- Luokat esitetään luokkasymbolin avulla.
- Luokkasymboli koostuu luokan nimestä sekä mahdollisten attribuuttien ja metodien esittelyistä.
- Luokka ja piirteet nimetään kuten Javassa.



Attribuutit

- Esittely yleisesti (mukana tärkeimmät määreet):

  {näkyvyysmääre} nimi {[kertautuminen]}

  {: tyyppi} {= alkuarvo} {lisätietomäärelista}

  missä aaltosulkeet tarkoittavat valinnaista osaa ja hakasulkeet mahdollisesti toistuvaa osaa.
- Näkyvyys: public (+), protected (#), private (-).
- Kertautumisella määritellään esimerkiksi taulukko.
  - Määre annetaan kuten assosiaatioiden yhteydessä.
  - Esimerkki: - luvut [0..*] : Integer

Attribuutit

- Tyypillisesti alkeistyyppejä, jotka annetaan usein isoilla alkukirjaimilla: Integer, Double, Boolean jne.
- Luokkatyyppisistä attribuuteista luokkasymbolissa esitetään vain kaavioon kuulumattomat (apu)luokat.
- Kaaviossa esiintyvien luokkien väliset suhteet esitetään attribuuttien asemasta assosiaatioina.
- Luokka-attribuutit alleviivataan.
- Lisätiedot kirjoitetaan aaltosulkeisiin.
  - Esimerkki: + PII : Double = 3.14 { const }

Metodit (operaatiot)

- Esittely yleisesti (mukana tärkeimmät määreet):

  {näkyvyysmääre} nimi({parametrilista})

  {: tyyppi} {lisätietomäärelista}

  missä aaltosulkeet tarkoittavat valinnaista osaa
- Parametreista voidaan antaa vain tyyppi. Tosinaan annetaan myös nimi. Tiedonvälityksen suunta (in, out, inout) määritellään harvoin.
- Esimerkki: + sayNTimes(msg: String, n: Integer)

  sayNTimes(String, Integer)

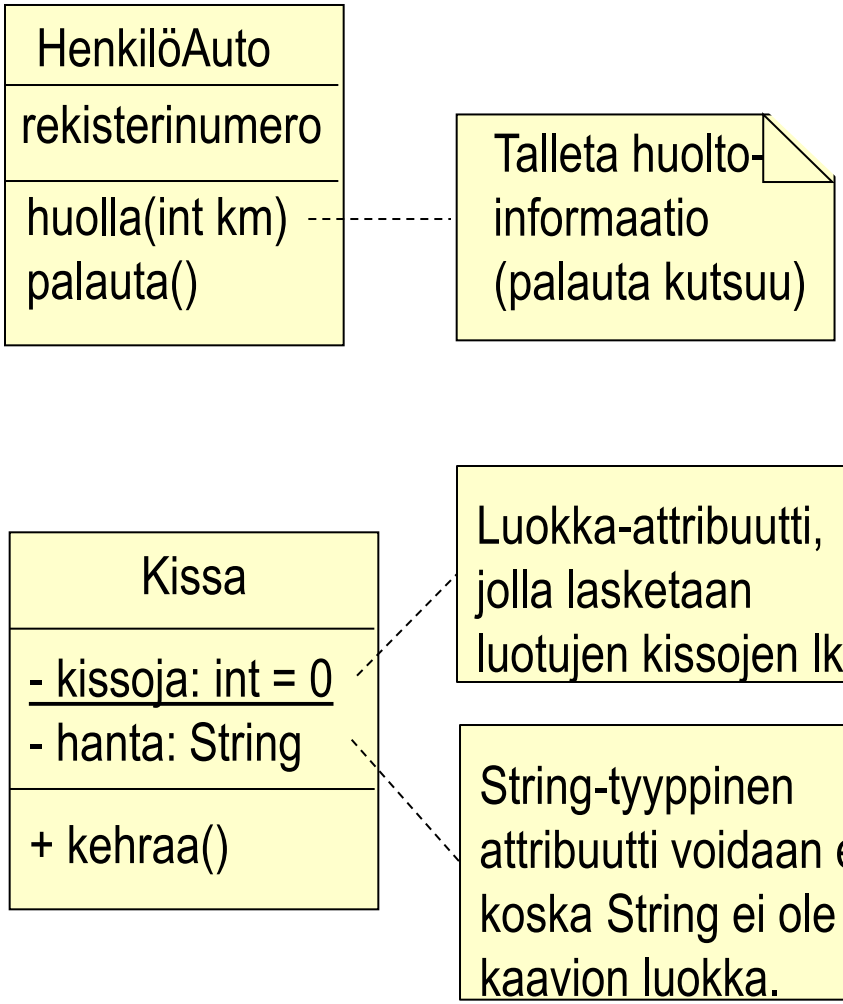
Metodit (operaatiot)

- Tyyppi annetaan kuten attribuuteille.
  - Paluuarvottomalle metodille ei anneta tyyppiä.
- Abstrakti metodi kirjoitetaan kursiivilla tai esitellään lisätiedolla {abstract}.

  Esimerkki: + *laskeEtaisyys(k: Piste) : Double*
- Luokkametodi alleviivataan.

  Esimerkki: - alusta(taulukko [][]: Character)
- ULM:ssa operaatio määritellään abstraktiksi esittelyksi ja metodi operaation toteutukseksi.

Esimerkkejä

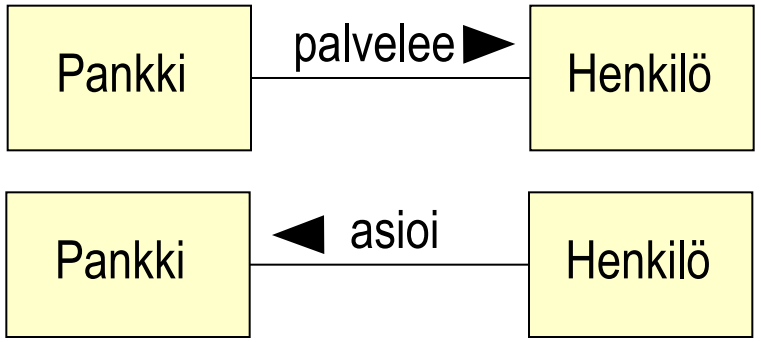


Suhteet

- Yhdistävät luokkakaavion elementtejä.
  - Piirretään erilaisina viivoina.
- Assosiaatioilla (association) kuvataan luokkien väliset suhteet, joilla on tietty pysyvyys.
  - Hetkelliset suhteet, jotka kestävät esimerkiksi vain metodin suoritusajan, kuvataan riippuvuussuhteina.
- Koosteassosiaatio ja aito kooste mallintavat “on-osa”-tyyppiset suhteet.
- Periytymissuhde ja rajapinnan toteutus ovat tuttuja.

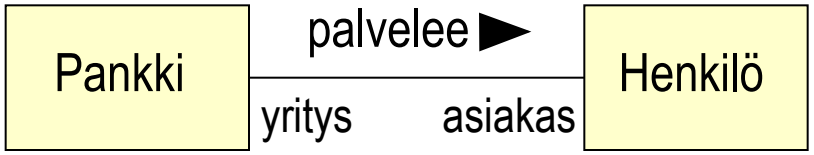
Assosiaatiot

- Tavallinen assosiaatio ilmaistaan piirtämällä viiva luokka-symbolien välille.
  - Binääriset, kahden luokan väliset yleisimpiä.
- Assosiaatiolla on usein nimi. (Asiayhteydestään selviä assosiaatioita ei tarvitse nimetä.)
- Assosiaatiolla ei ole varsinaista suuntaa.
- Nimen yhteyteen voidaan merkitä nuolisymboli, joka kuvaa lukusuunnan.
- Sama assosiaatio voidaan lukea myös toisinpäin, jos nimi vaihdetaan kuvaamaan käänteistä tilannetta.

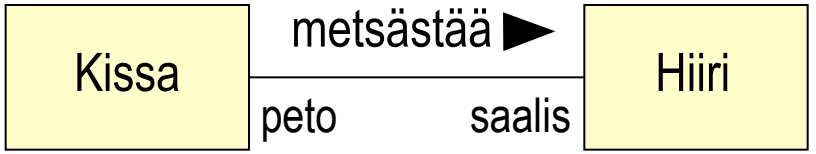


Assosiaatiot

- Assosiaation päätä kutsutaan rooliksi (role), joka voidaan myös nimetä.
- Nimi kuvaa assosiaation merkityksen vastakkaisessa päässä olevan luokan kannalta.
- Esim. Pankki näkee henkilön palveltavana asiakkaana ja henkilö pankin palvelua tarjoavana yrityksenä.



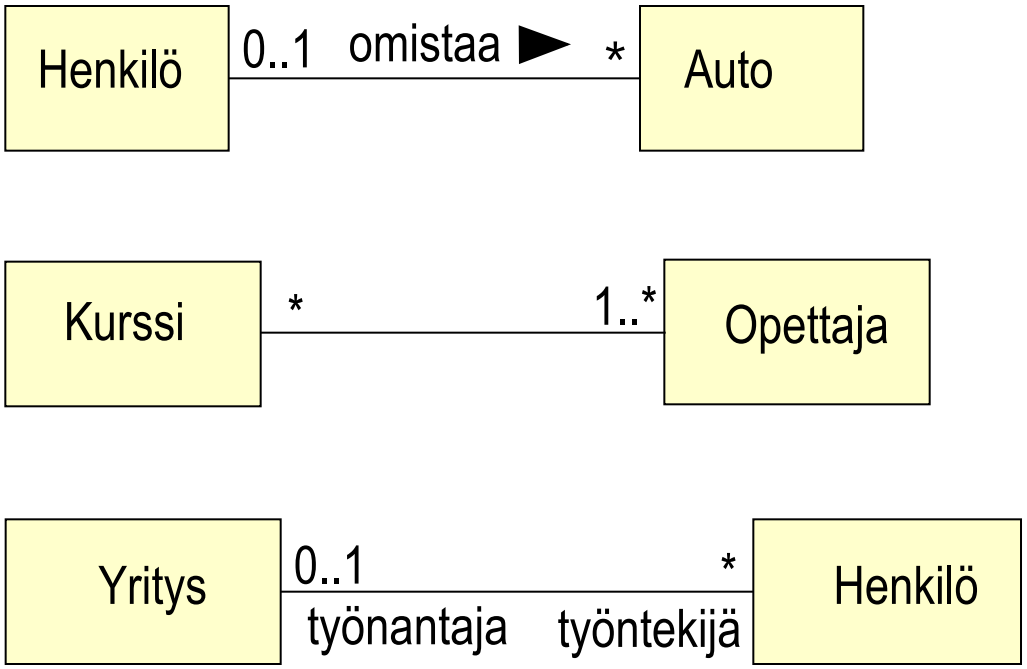
- Kissalle hiiri on saalis ja hiirelle kissa on peto.



Assosiaatiot

- Rooli ilmaisee myös assosiaation kertautumisen (multiplicity), jolla ilmaistaan montako assosiaation vastakkaisen pään luokan oliota liittyy tarkasteltavana olevan pään luokan yksittäiseen olioon.
  - * tarkoittaa mielivaltaisen monta (* == 0..*).
  - 0..1 tarkoittaa kertautumista “yksi tai ei yhtään”.
  - Jos kertautumista ei ole merkitty, se on määrittelemätön (eikä siis esimerkiksi 1).
  - Moninkertaisessa assosiaatiossa kertautuminen > 1.
  - Esim. Pankki palvelee nollaa tai useampaa asiakasta ja henkilö asioi 0, 1 tai 2 pankissa.

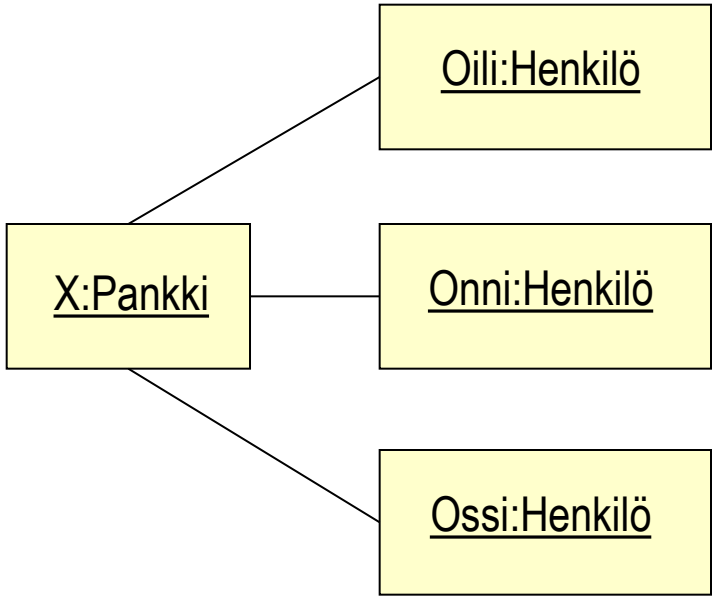
Esimerkkejä



Assosiaatiot ajonaikana

- Assosiaation ajoaikainen ilmentymä on linkki (link), joka vallitsee assosiaation olioiden välillä.
- Määritellään olioiden tupliksi: binäärisissä assosiaatioissa linkit ovat järjestettyjä oliopareja.
- Monikertaisissa assosiaatioissa linkkejä voi olla kaksi tai useampia.

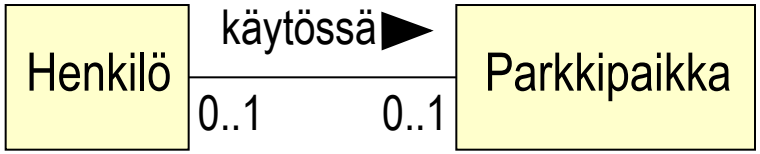
- Oliokaavio esittää luokkakaavion mahdollista ajonaikaista ilmentymää. Kaaviossa viiva vastaa linkkiä.



Assosiaatiot ajonaikana

- Assosiaatiot toteutetaan pääosin luokkatyyppisinä attribuutteina (viitteinä) assosiaation lähtöluokassa.

- Luokkakaavio:



- Toteutus Java-kielellä:

```
public class Henkilo {
   ...
   private Parkkipaikka ppaikka;
   ...
}
```

Assosiaatiot ajonaikana

- Monikertainen assosiaatio voidaan toteuttaa luokkakaavion kuulumattoman säiliöluokan avulla.

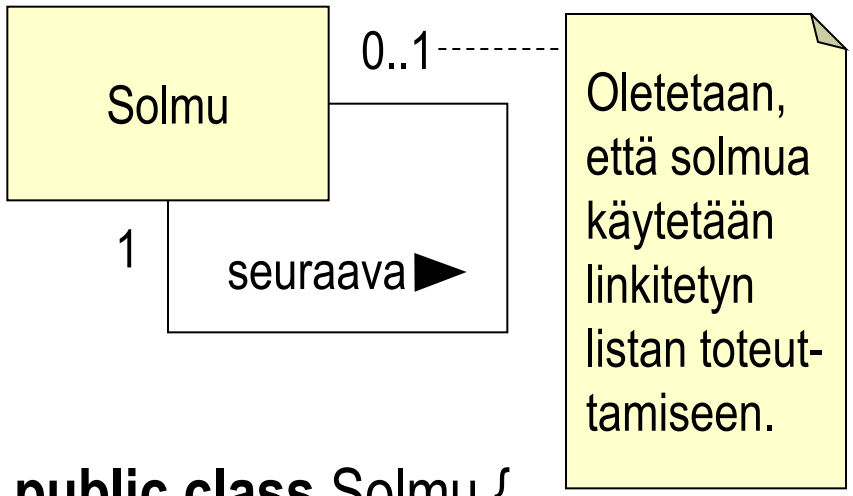
- Luokkakaavio:

- Eräs toteutus Java-kielellä:

```
public class Pankki {
   ...
   private Vector<Henkilo> asiakkaat;
   ...
}
```

Refleksiivinen assosiaatio

- Refleksiivinen (unaarinen) assosiaatio päättyy lähtöluokkaansa.
- Näin esitetty assosiaatio on hankala ymmärtää ilman nimeä.
- Roolien käyttö on myös suotavaa.



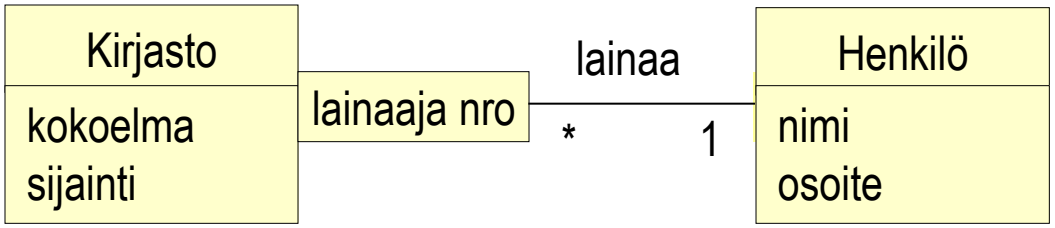

```
public class Solmu {
   // Solmun sisältämä tieto.
   private Object alkio;
   // Viite seuraavaan solmuun.
   private Solmu seuraava;
   ...
}
```

Assosiaation yksilöinti

- Jos assosiaatioon liittyy tieto, joka määrää assosiaation toisessa päässä olevien olioiden joukon, niin tämä tieto voidaan esittää niin kutsuttuna yksilöintinä (qualification).
- Yksilöinti liittyy aina moninkertaiseen assosiaatioon.
- Tavallisesti yksilöinti muuttaa kertautuvan pään kertautumisen "1":ksi (tai "0..1":ksi). Tällöin yksilöinti määrää linkin toisessa päässä olevan olion yksiselitteisesti.

Assosiaation yksilöinti

- Yksilöinti merkitään luokkasymboliin liittyvänä pienenä laatikkona, jonka sisään kirjoitetaan linkin yksilöivä tieto, joka tarkentaa olion linkin vastakkaisessa päässä.
- Usein yksilöivä tieto on vastakkaisella puolella olevan luokan attribuutti. Huomaa, että yksilöinti ei muuta sen pään kertautumista, johon se piirretään.



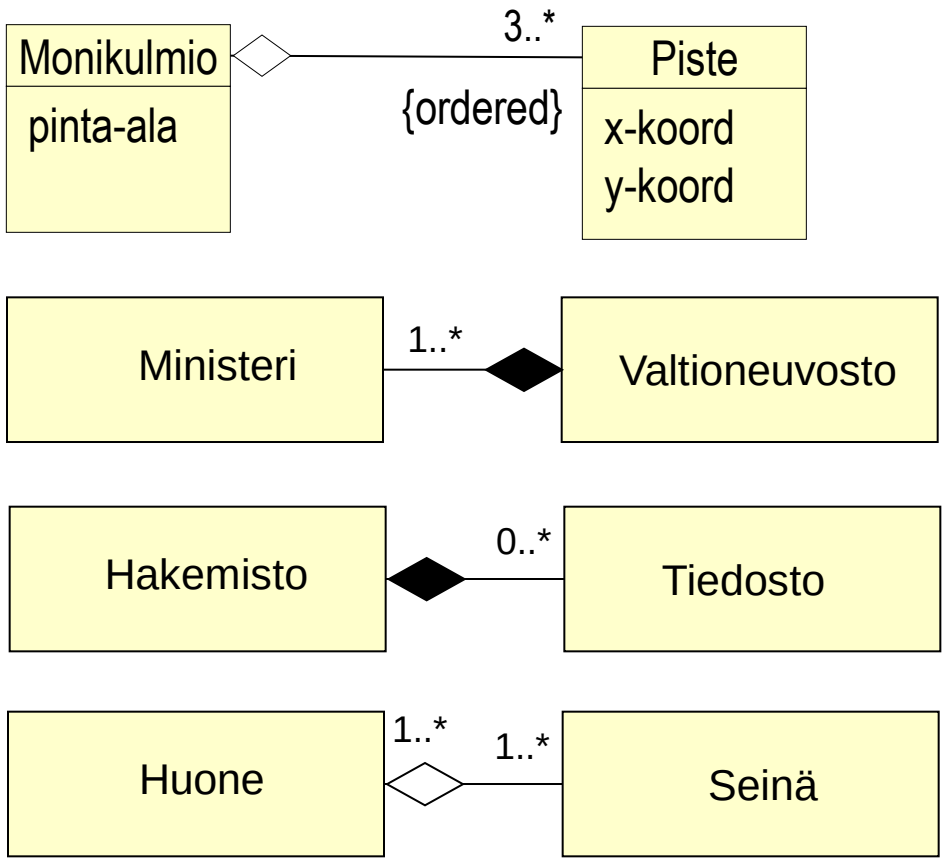
Koosteassosiaatio

- Kooste (aggregation) on erityinen assosiaatiolaji, joka esittää suhteen “on-osa” (HasA) tai “kuuluu” luokkien (ilmentymien) välillä.
- Oma symboli: pieni vinoneliö sisältävässä päässä (siis suhteen "A on-osa B:tä" B-päässä).
- Tämä symboli korvaa tavallisesti assosiaation ja roolien nimet. Koosteeseen voidaan soveltaa normaaliin tapaan kertautumista ja yksilöintiä.
- Valinta tavallisen ja koosteassosiaation välillä on toisinaan enemmänkin makuasia.

Aito kooste

- Aito kooste (composition) tarkoittaa koostesuhdetta, jossa osaolio riippuu isäntäoliostaan kahdella tavalla:
  - osa ei voi olemassa ilman isäntäänsä ja
  - osa voi olla vain yhden isännän osa.
- Isännän tulee yleensä huolehtia osan luonnista ja hävittämisestä.
- Aito kooste merkitään kuten kooste, mutta vinoneliö on musta.
  - Kertautuminen sisältävässä päässä 1 tai 0..1.

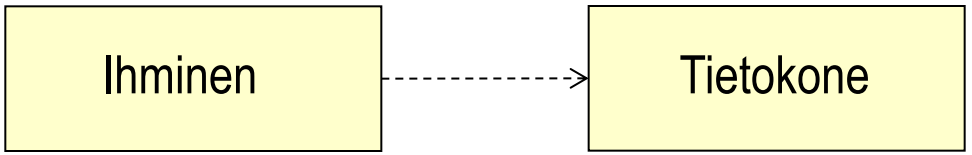
Esimerkkejä



Jos olioliitoksilla on mallin kannalta olennainen järjestys, voidaan rajoite {ordered} liittää rooliin.

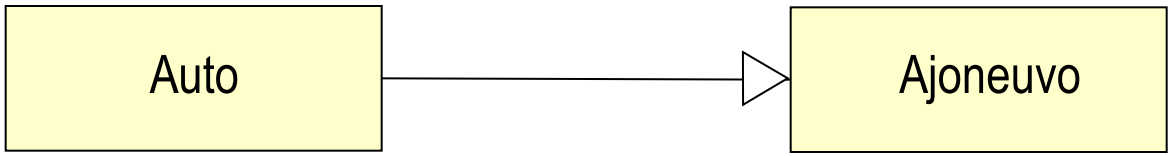
Riippuvuussuhde

- Riippuvuussuhteella (dependency) kuvataan luokkien väliset hetkelliset suhteet.
  - Muun muassa olion luominen ja metodin kutsuminen.
- Esitetään väkäpäisellä nuolella, jonka varsi piirretään katkoviivalla.
  - Nuoli piirretään kohti luokkaa, jonka palveluja käytetään.
- Esimerkki: ihminen käynnistää ja sammuttaa tietokoneen.



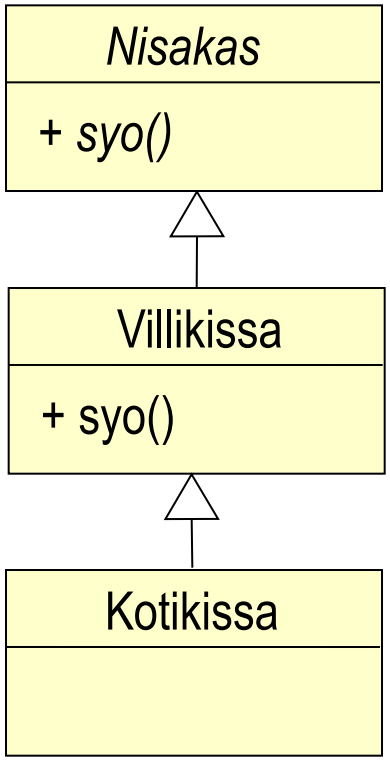
Periytymissuhde

- Periytyminen (inheritance) kuvaa erikoistumis- tai yleistyssuhteen yli- ja aliluokan välillä.
- Suhde piirretään kolmiokärkisenä yliluokkaan osoittavana nuolena.
- Jos luokalla on useita aliluokkia, nuolet voidaan piirtää joko erikseen tai yhdistettynä samaan kärkeen.



Periytymissuhde

- Perittyjä attribuutteja ei tavallisesti piirretä näkyviin jälkeläisiin.
- Abstraktin metodin (UML:n operaatio) toteuttava metodi (UML:n metodi) voidaan esitellä toteuttavassa luokassa.
- Metodin korvaaminen voidaan tarvittaessa ilmaista jälkeläisessä.

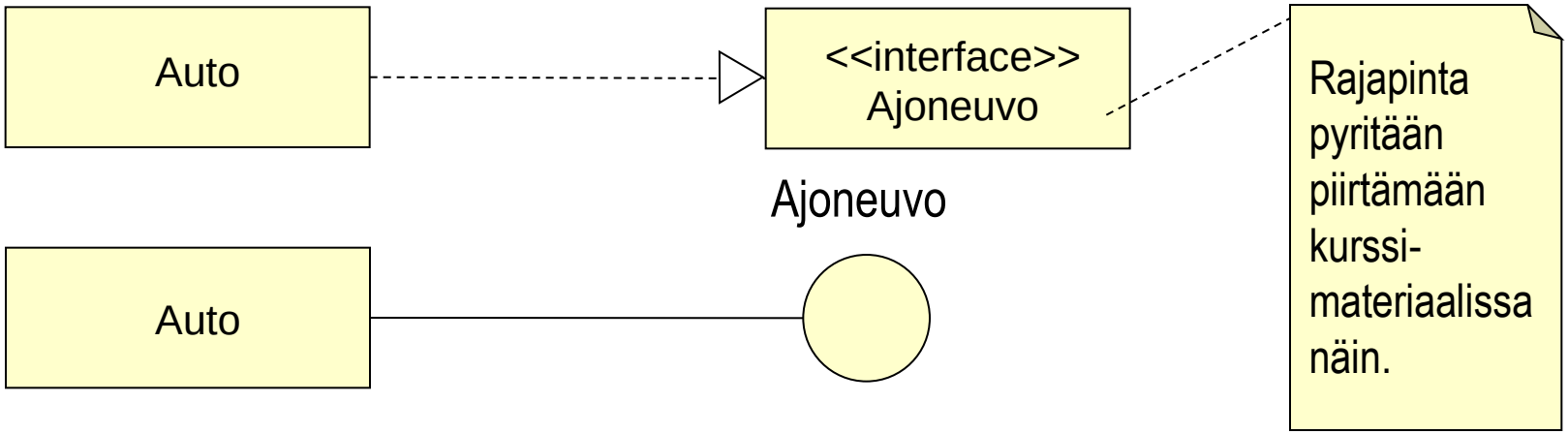


Rajapinta

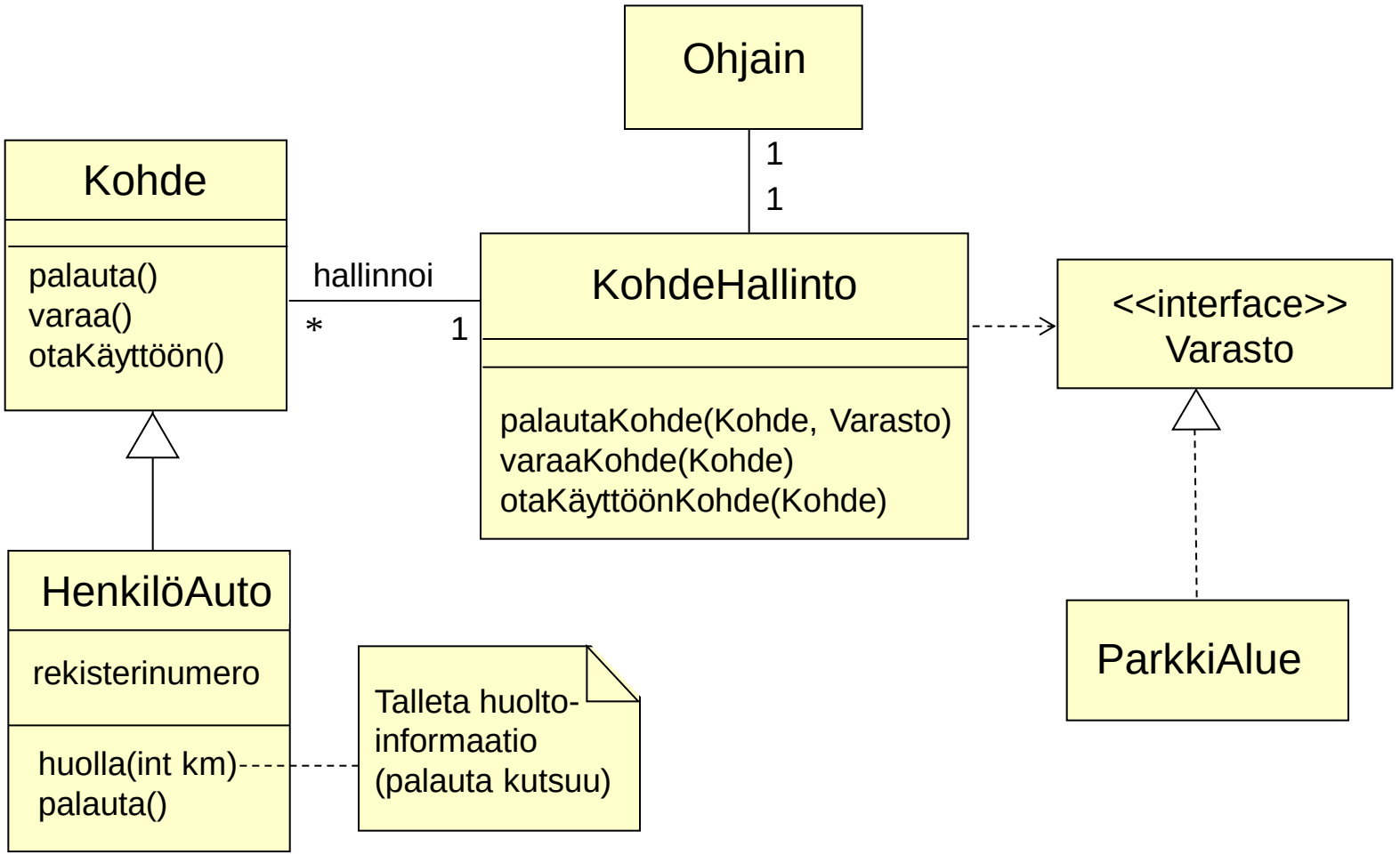
- Rajapinta esitetään joko stereotyypillä <<interface>> varustetulla luokkasymbolilla tai pyöreällä rajapintasymbolilla.
- Jos luokka toteuttaa rajapinnan, piirretään edellisessä tapauksessa toteutussuhdetta kuvaava nuoli (kuten periytyminen, mutta katkoviivalla) luokasta rajapintaan.
- Jälkimmäisessä tapauksessa rajapintaympyrä yhdistetään yksinkertaisella viivalla luokkasymboliin.

Rajapinta

- Luokkasymbolin käyttäminen rajapinnan kuvaukseen on hyödyllistä silloin, kun halutaan näkyville rajapinnan tarjoamat metodit.
- Tällöin metodi esitellään sekä rajapinnassa että sen toteuttavassa luokassa.



Ajoneuvonvarausjärjestelmä



20. *Iterator*- ja *Iterable*-rajapinnat

Iterator-rajapinta

- Mahdollistaa kokoelman järjestelmällisen läpi käynnin (iteroinnin) yhteen suuntaan.
  - **public boolean** hasNext(): kertoo onko kokoelmasta vielä saatavilla alkiota.
  - **public** E next(): palauttaa viitteen seuraavaan alkioon, jos sellainen löytyy.
  - **public void** remove(): poistaa nykyisen alkion, voidaan jättää toteuttamatta.
- Generinen tyyppi *E* on kokoelman alkion tyyppi.
- Rajapinta *java.util*-pakkauksessa.

Iterable-rajapinta

- Mahdollistaa kokoelman läpikäynnin for-each-rakenteella.
  - **public** Iterator<T> iterator(): palauttaa viitteen kokoelman iteraattoriin
- Generinen tyyppi *T* on kokoelman alkion tyyppi.
- Rajapinta *java.lang*-pakkauksessa.