Olio-ohjelmointi Javalla

Versio 1.0

Antti Herala, Erno Vanhala ja Uolevi Nikula

Lappeenrannan teknillinen yliopisto
LUT School of Business and Management
Laboratory of Innovation and Software
PL 20
53851 Lappeenranta

Lappeenrannan teknillinen yliopisto
Lappeenranta University of Technology

LUT School of Business and Management
Laboratory of Innovation and Software

LUT Scientific and Expertise Publications

Oppimateriaalit – Lecture Notes 5

Antti Herala, Erno Vanhala, Uolevi Nikula

Olio-ohjelmointi Javalla

ISBN 978-952-265-753-4
ISBN 978-952-265-754-1 (PDF)
ISSN-L 2243-3392
ISSN 2243-3392

Lappeenranta 2015

Esipuhe

Tämä opas jatkaa Lappeenrannan teknillisellä yliopistolla (LUT) kirjoitettujen ohjelmointioppaiden sarjaa. Opiskelijamme ovat törmänneet Ohjelmoinnin perusteet -opintojaksolla Python 3 -oppaaseen ja Käytännön ohjelmoinnissa C-oppaaseen. Tämä opas jatkaa edellämainittujen oppaiden luomaa jatkumoa ja tarjoaa opiskelijoille tietoa olio-ohjelmoinnista Java-ohjelmointikielellä. Opas edellyttää, että opiskelijalla on jo perustason osaaminen esim. LUTin ohjelmointikurssien muodossa. Mikäli tätä osaamista ei ole, on suositeltavaa aloittaa opiskelu Python 3 -ohjelmointioppaan parissa.

Oppaan esimerkit on rakennettu Ubuntu 14.04 -käyttöjärjestelmällä, NetBeans 8 ja Java 8 ympäristöissä. Java on periaatteessa laitteistoriippumaton kieli, joten esimerkit ovat suoraan käytettävissä myös muissa käyttöjärjestelmissä ja kehitysympäristöissä. Oppaan esimerkit toimivat myös vanhemmassa Java 7:ssa.

Opas on kaksiosainen tämän ensimmäisen oppaan keskittyessä olio-ohjelmoinnin perusteisiin Javalla ja toisen osan laajentaessa tätä osaamista graafisella käyttöliittymällä, kartoilla ja muilla ohjelmointia tukevilla asioilla kuten versionhallinnan käytöllä.

Tämä opas on lisensoitu vastaavalla Attribution-NonCommercial-ShareAlike 4.0 International -lisenssillä (CC BY-NC-SA 4.0) ja tehty yhteistyössä Liikenneviraston kanssa.

Sisällysluettelo

1. Luokat ja olio-ohjelmointi....6
1.1. Ohjelmointiparadigmoista....6
1.2. Mitä on olio-ohjelmointi?....7
1.3. Luokat....8
1.4. Entäs oliot?....10
1.5. Rakentaja(t)?....11
1.6. Muutama sana jäsenmuuttujista....12
1.7. Näkyvyysmääre: yksityinen (private) ja julkinen (public)....13
1.8. Ja lopuksi Javaa....14
1.8.1. Javan tietotyypit....14
1.8.2. Tietorakenne: Taulukko....15
1.8.3. System.out.println()....16
1.9. Kerrataan lopuksi....17
1.9.1. Main-funktion käyttö....19
2. Lisää luokista ja muuttujista....20
2.1. Olion tuhoaminen....23
2.1.1. Purkaja ja muistin vapauttaminen....23
2.1.2. finalize()....24
2.2. Viittaukset olioihin....25
2.2.1. Viittaus ja osoitin....25
2.3. Tyyppimuunnoksia....26
2.4. Lopuksi Javaa....26
2.4.1. Tyyppimuunnoksia muuttujilla....27
2.4.2. Tietorakenne: Lista ja vektori....29
2.5. Standardivirrat....33
2.6. Kerrataan....35
3. Kirjastot ja tietovirrat....36
3.1. Tietovirrat (I/O streams)....36
3.1.1. Serialisointi....37
3.2. Javaa....37
3.2.1. Tietovirrat Javalla....38
3.2.2. Tietorakenteet: Enumeration ja map....40
4. Oliopohjainen suunnittelu ja UML....43

4.1. Pariutuminen (coupling) ja koheesio (cohesion)....44
4.2. UML....44
4.2.1. Luokkakaavio (class diagram)....45
4.3. Javasta....47
4.3.1. this....47
4.3.2. Package ja nimiavaruudet....50
4.3.3. Enum....51
5. Periytyminen ja abstraktio....53
5.1. Abstrakti luokka....54
5.2. Näkyvyysmääre: suojattu (protected)....56
5.3. Toiminnan korvaaminen....57
6. Virhetilanteita ja periytymisiä....59
6.1. Rajapinta (interface)....62
6.2. Virheiden käsittely....63
6.3. Javaa....64
6.3.1. Rajapinta vs. abstrakti luokka....65
6.3.2. Foreach....67
6.3.3. Virheiden käsittely....67
6.3.4. final....70
7. Kopiointi ja sijoitus (sekä luokkamuuttujat)....72
7.1. Viitekopiointi....72
7.2. Matalakopiointi....74
7.3. Syväkopiointi....75
7.4. Kopiointi käytännössä....76
7.5. Sijoittaminen....81
7.6. Luokkamuuttujista ja static-avainsanasta....82
8. Sisäiset luokat....84
9. Java ja internet....87
10. Lähteet....93
11. Liite 1: Asennusopas....94

1. Luokat ja olio-ohjelmointi

1.1. Ohjelmointiparadigmoista

"*Programming paradigms are heuristics used for algorithmic problem solving. A programming paradigm formulates a solution for a given problem by breaking it down to specific building blocks and defining the relationships among them.*" - Yuila Stolin ja Orit Hazzan [Students' Understanding of Computer Science Soft Ideas: The Case of Programming Paradigm, SIGCSE Vol. 39 Number 2, 2007]

Ohjelmoinnin suunnitteluun ja ohjelmointiin on tarjolla useita erilaisia paradigmoja, joista tämän oppaan käyttäjät ovat oletettavasti päässeet jo tutustumaan proseduraaliseen ohjelmointiin. Seuraavassa käsitellään lyhyesti proseduraalinen, funktionaalinen ja oliopohjainen paradigma, jotka ovat hyvin paljon käytettyjä ohjelmoinnissa.

Ohjelmoinnin päätarkoitus on luoda ratkaisu ongelmalle niin, että se voidaan tuottaa koneellisesti ja sitä voidaan toistaa useita kertoja. Ohjelma siis ratkoo käyttäjän ongelmia algoritmisesti. Ohjelmointiparadigmat ovat olemassa, jotta jokaisen ohjelmoijan ei tarvitse itse keksiä pyörää uudelleen, vaan ne tarjoavat ohjelmoijalle alustan, jonka avulla ongelmaa voidaan lähteä ratkomaan. Paradigmat myös antavat ohjelmalle hyvän rakenteen, jolloin isoja ja pieniä ongelmia voidaan ratkoa samalla tavalla. Tämän vuoksi ohjelmointiparadigmoja käytetään hyvin paljon ohjelmoinnissa, sillä ne luovat pohjan kaikille ohjelmointikielille ja ratkaisutavoille.

Taulukko 1. Ohjelmointiparadigmat

Paradigma	Rakentuu	Rakenneosat liittyvät toisiinsa
Proseduraalinen	Proseduureista tai käskysarjoista	Hierarkisesti
Oliopohjainen	Luokista	Periytymällä
Funktionaalinen	Funktioista	Yhdistämällä

Olioparadigma on yksi eniten käytetyistä ohjelmointiparadigmoista ja sen ympärille onkin kasautunut monia kieliä, joita käytetään olio-ohjelmointiin, esimerkiksi Java, C++, C#, Python, PHP ja Smalltalk. Oliopohjainen ohjelma koostuu siis luokista, jotka voivat periä tietojaan ja taitojaan toisilta luokilta ja sitä kautta olioista, jotka suorittavat tehtäviä ja ovat toisten olioiden

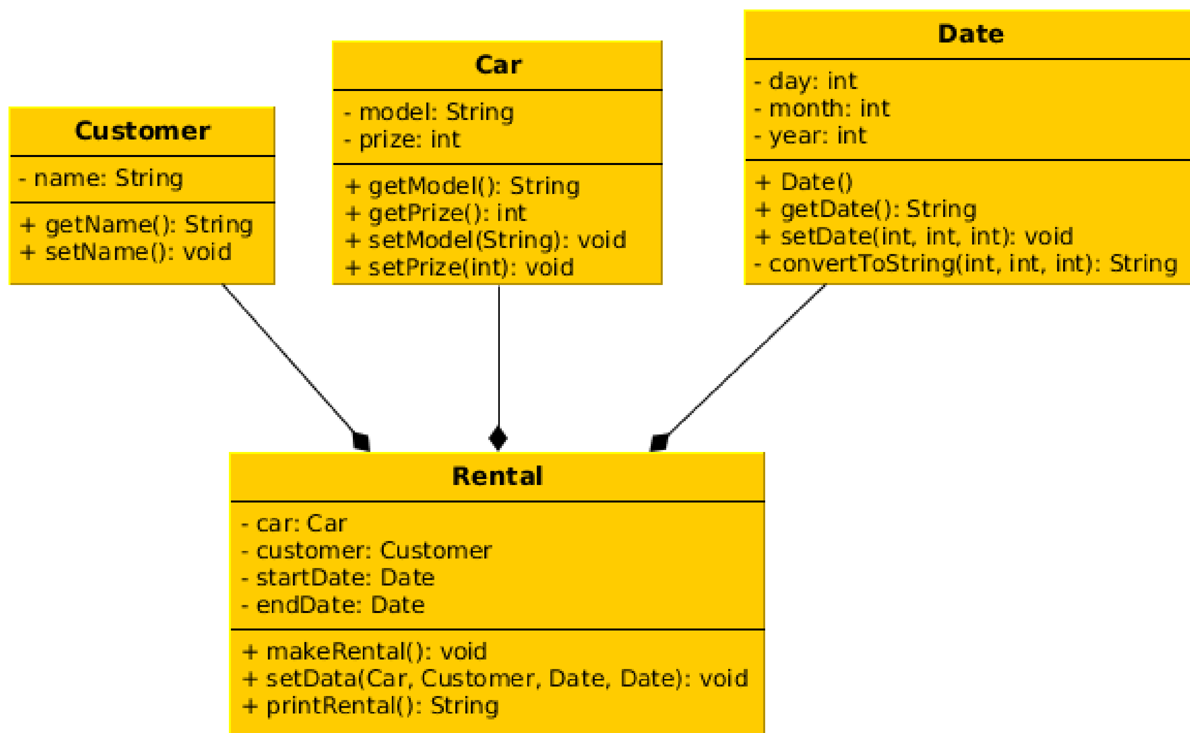
käytössä. Koska olioistakin rakentuva ohjelma voi olla hyvinkin monimutkainen, sitä mallinnetaan UML-luokkakaavioiden avulla, joihin palataan myöhemmin tässä oppaassa luvussa 4.

1.2. Mitä on olio-ohjelmointi?

"*Olio-ohjelmointi on ohjelmoinnin lähestymistapa, jossa ohjelmointiongelmien ratkaisut jäsennetään olioiden yhteistoimintana. Oliot sisältävät toisiinsa loogisesti liittyvää tietoa ja toiminnallisuutta.*" - Wikipedia [URL: http://fi.wikipedia.org/wiki/Olio-ohjelmointi]

Olio-ohjelmointi on oliopohjaisesta paradigmasta lähtöisin oleva ohjelmointitapa, jossa ohjelmatoteutus noudattaa luokkapohjaisuutta. Ongelma jaetaan osiin niin, että toteutuksessa on yhdessä toimivia luokkia, jotka ratkaisevat itsenäisiä osaongelmia. Tämänkin esittämiseen kaikkein yksinkertaisin tapa on esimerkki.

> Ajatellaan autovuokraamoa, missä autovuokraamon omistaja haluaa vuokraamon toimintaa avustavan ohjelman, mikä pitää kirjaa tapahtuneista vuokrauksista. Koska pohdimme asiaa olio-ohjelmoinnin pohjalta, lähdemme toki miettimään, mitä tekijöitä liittyy yhteen varaukseen. Varaukseenhan liittyy aina vuokran tekevä asiakas, siihen liittyy vuokrattava auto ja tottakai on tiedettävä haku- ja palautuspäivämäärät. Yksinkertaisuuden vuoksi asiakkaalla on järjestelmään tallennettuna vain nimi, jolla asiakas identifioidaan. Autolla vuorostaan on tietoinaan auton malli sekä sen vuokraushinta. Päivämäärä on ilmoitettu järjestelmässä muodossa pp.kk.vvvv. Nyt järjestelmää voidaan havainnollistaa UML-luokkakaavion avulla, johon on merkitty tarvittavat luokat, niiden operaatiot sekä tiedot. Älkää huoliko, UML:ää käsitellään edempänä lisää.



Kuva 1.1. Autovuokraamo

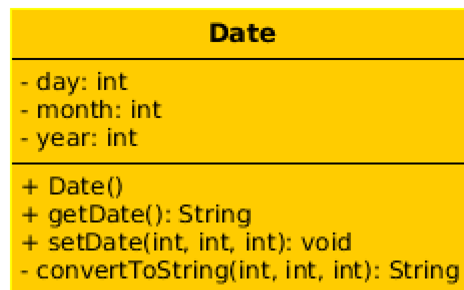
Kuvassa nähdään, että varaus (Rental) muodostuu kolmesta muusta luokasta, asiakkaasta (Customer), autosta (Car) sekä kahdesta päivämäärästä (Date). Aina kun autovuokraamossa tapahtuu varaus, järjestelmään ilmoitetaan varauksen tapahtuneen. Tällöin järjestelmä luo Rental-luokasta uuden instanssin, olion, ja pyytää käyttäjää syöttämään tarvittavat tiedot. Tietojen avulla järjestelmässä syntyy olioita asiakkaasta, autosta ja päivämääristä ja ne linkitetään suoraan liittymään varaukseen. Tällöin meillä on käytössä varaus-olio, joka tietää tarkalleen, kuka vuokrasi jonkin tietyn auton ja milloin sitä voidaan odottaa viimeistään takaisin.

1.3. Luokat

"*Start by thinking of a class as an object. Imagine a class as a container, like that glass or plastic bottle you drink your favorite beverage from. Even something this simple has specifications - it has a size. It can only hold so much. A beverage company puts something in it. You take it out. It's used for storing something. More commonly, classes can store things and do things.*" - Guy M. Haas [URL: http://www.bfoit.org/itp/JavaClass.html]

Kuten edellä olevassa esimerkissä voidaan huomata, luokkien avulla voidaan muodostaa suuriakin kokonaisuuksia liittämällä niitä yhteen. Miten tällaiset luokat voidaan sitten luoda käytännön tasolla? Jotta luokka voitaisiin luoda, on sille luotava *rakentajat*, *metodit* sekä *jäsenmuuttujat* ja

jokainen luokan metodi ja muuttuja tarvitsevat omat *näkyvyysmääreensä*. Kaikki nämä termit tullaan esittelemään tässä luvussa, mutta käydään asiaa esimerkin avulla, jossa käytetään aikaisemmasta esimerkistä tuttua Date-luokkaa.



Kuva 1.2. Date-luokka UML:nä

Esimerkki 1.1. Date-luokka Javalla

```
class Date {
   private int day;
   private int month;
   private int year;

   public Date() {
      day = 1;
      month = 1;
      year = 1970;
   }
   public void setDate(int new_day, int new_month, int new_year) {
      day = new_day;
      month = new_month;
      year = new_year;
   }
   public String getDate() {
      return convertToString();
   }
   private String convertToString() {
      String s_day = Integer.toString(day);
      String s_month = Integer.toString(month);
      String s_year = Integer.toString(year);

      String total = s_day + "." + s_month + "." + s_year;
      return total;
   }
}
```

Kuvan ja koodiesimerkin välinen yhteys on käytännössä hyvin yksinkertainen ja se selkenee opasta eteenpäin selatessa. Luokalla on kolme yksityistä muuttujaa (*private variable*) day, month ja year, kaksi julkista metodia (*public method*) setDate() ja getDate(), yksityinen metodi (*private method*) convertToString() sekä Date-rakentaja (*constructor*). Kuten edellä olevassa esimerkissä, on luokan

rakenne tavallisesti se, että ylimpänä määrittelyssä ovat jäsenmuuttujat, sitten rakentajat ja lopuksi metodit. Rakentaja löytyy aina jokaiselta luokalta ja se on hyvä merkitä luokkakaavioon, varsinkin jos luokalla on useampia rakentajia.

1.4. Entäs oliot?

"*The terms class and object are definitely related to one another, but each term holds its own distinct meaning." … "The term class refers to the actual written piece of code which is used to define the behaviour of any given class." … "The term object, however, refers to an actual instance of a class."* - Varoon Sahgai

[URL:http://www.programmerinterview.com/index.php/java-questions/difference-between-object-and-class/]

Luokka on käsite, johon oppaan lukijat ovat varmastikin jo aikaisemmin törmänneet tutustuessaan ohjelmoinnin maailmaan. Luokka on ohjelmoinnissa kuin malli tai suunnitelma, jonka mukaan siitä luodut instanssit, oliot, käyttäytyvät ja rakentuvat. Luokka antaa siis pohjan attribuuteille ja metodeille, joita luokasta luodut oliot tulevat käyttämään, eli sen voidaan sanoa olevan muotti olioille. Samalla tavoin kuin oikean elämän muotit, luokka antaa olioille muodon ja tarkoituksen pakottamatta muuttujiin arvoja, kuten muotti ei pakota käyttämään vain tiettyä materiaalia valamisessa. On tosin mahdollista, että kaikkia muotteja ei ole tarkoitettu kaikille materiaaleille ja sama pätee hyvin myös luokkien käyttämiseen; kaikki luokat eivät sovellu aivan kaikkiin tarkoituksiin. Luokka siis muodostaa abstraktin mallin, missä se määrittelee toiminnallisuus- ja tietotyyppien avulla kokonaisuuden, josta voidaan luoda ilmentymiä.

Kuten edellä mainittiin, oliot ovat luokista luotuja instansseja eli ilmentymiä. Olioita on helppo ymmärtää oikean elämän esimerkkien avulla, esimerkkinä talon rakentaminen. Kun ajatellaan taloa, voidaan ajatella erilaisia taloja: tasakattoisia, viistokattoisia, harjakattoisia tiili-, puu- tai elementtitaloja. Kaikissa taloissa on erilainen katto, mutta jokaisessa sellainen kuitenkin on eli voidaan ilmaista, että yksi talon attribuuteista eli ominaisuuksista on katto. Lisäksi talon seinät voidaan rakentaa erilaisista materiaaleista, mutta jokaisella talolla on kuitenkin seinät ja niitä on tavallisesti neljä, mutta voi olla myös vain kolme tai vaikka seitsemäntoista. Esimerkkejä voidaan miettiä pidemmällekin, mutta ajatus taisi tulla jo selväksi: *luokalla on attribuutteja, eli ominaisuuksia*. Luokkien ja olioiden toiminta on hyvin lähellä talon rakentamista, sillä luokalla on ominaisuuksia ja toiminnallisuuksia, jotka olio valmistuessaan saa omakseen. Ajatellaan siis Talo-luokkaa, jossa on määritelty, että talolla on katto, sillä on jokin määrä seiniä ja ne on rakennettu jostakin materiaalista. Näin mielletään talo hyvin abstraktilla tasolla. Kun Talo-oliota lähdetään rakentamaan, voidaan oliolle määritellä millainen katosta on tultava ja montako seinää rakennetaan

mistäkin materiaalista. Tällöin luodaan niin sanotusti jotakin, joka on jo oikeasti olemassa ja käytettävissä. Luokka on siis yleisesti ymmärretty, abstraktin tason esitys ja olio on käytettävissä oleva instanssi, jolla on luokan määrittämät ominaisuudet. Luokat ja oliot muodostavat keskeisen osan koko olioparadigmasta.

1.5. Rakentaja(t)?

"*A class contains constructors that are invoked to create objects from the class blueprint. Constructor declarations look like method declarations - except that they use the name of the class and have no return type.*" - Oracle [URL:
http://docs.oracle.com/javase/tutorial/java/javaOO/constructors.html]

Rakentaja nimensä mukaisesti rakentaa olion luokan mallin pohjalta kutsuttaessa. Rakentajan näkyvyysmääre on tavallisesti julkinen, poislukien erikoistapaukset, joista myöhemmin abstraktion yhteydessä luvussa 5. Luokan rakentajassa tavallisesti olion tila alustetaan (esimerkki 1.1.), eli sen yksityisille muuttujille annetaan oletusarvoja niin, että olio ei sisällä tyhjää tietoa (null). Tämä on tuttua jo C-ohjelmoinnin puolelta.

Tietysti olisi myös näppärää, jos luokkaa luodessa voisi lähettää alustusparametreja oliota varten, jolloin ei olisi erikseen tarvetta ensin luoda oliota ja sitten vasta asettaa sille toivottuja arvoja. On mahdollista luoda rakentaja, joka ottaa sisäänsä parametreja ja alustaa olion yksityiset muuttujat ilman erillistä funktiota. Esimerkki seuraa:

Esimerkki 1.2. Date-luokan rakentaja parametrien kanssa

```
public Date(int new_day, int new_month, int new_year) {
   day = new_day;
   month = new_month;
   year = new_year;
}
```

On huomattava, että yhdellä luokalla voi olla useita rakentajia, kunhan kaksi rakentajaa eivät saa täysin samat tietotyypit omaavia parametreja. Esimerkiksi rakentajat Date() ja Date() eivät voi olla samassa luokassa, mutta rakentajat Date() ja Date(int x) voivat. Tällöin ensimmäistä, parametritonta rakentajaa kutsutaan *oletusrakentajaksi* (*default constructor*).

Jotta ohjelmaa voitaisiin testata, on ohjelman yhteen - ja vain yhteen - luokkaan lisättävä main-funktio. Tämän funktion avulla ohjelma voidaan ajaa kuin tavallinen ohjelma ja sen luokkia voidaan

käyttää. Esimerkissä luodaan kaksi instanssia Date-luokasta käyttäen ensin oletusrakentajaa ja sen jälkeen parametrit omaavaa rakentajaa päätyen samaan lopputulokseen.

Esimerkki 1.3. Date-luokassa oleva ohjelman main-funktio.

```
public static void main(String[] args) {
      Date date = new Date();
      System.out.println(date.getDate());
      date.setDate(17, 3, 2014);
      System.out.println(date.getDate());

          Date date2 = new Date(17, 3, 2014);
      System.out.println(date.getDate());
}
```

Tuloste

```
run:
1.1.1970
17.3.2014
17.3.2014
BUILD SUCCESSFUL (total time: 0 seconds)
```

1.6. Muutama sana jäsenmuuttujista

"*Instance variables belong to an instance of a class. Another way of saying that is instance variables belong to an object, since an object is an instance of a class. Every object has it's own copy of the instance variables.*" - Varoon Sahgai

[URL: http://www.programmerinterview.com/index.php/c-cplusplus/whats-the-difference-between-a-class-variable-and-an-instance-variable/]

Jäsenmuuttujia voi olla olio-ohjelmoinnissa kahdenlaisia, joko olion jäsenmuuttujia tai luokan jäsenmuuttujia. Selkeyden vuoksi tässä oppaassa puhutaan jäsenmuuttujista (*instance variables*) ja luokkamuuttujista (*class variables*), joiden eroista kertoo seuraava seuraava tekstilaatikko. Luokkamuuttujiin tullaan palaamaan tarkemmin myöhemmin oppaassa luvussa 7, mutta seuraavassa muutama sana jäsenmuuttujista.

Jäsenmuuttuja vs. luokkamuuttuja

Jäsenmuuttuja on jokaisella luokasta muodostetulla oliolla oleva yksilöllinen muuttuja. Luokkamuuttuja vuorostaan on muuttuja, joka on jokaiselle luokasta luodulle oliolle yhteinen. Esimerkiksi edellä käytetyssä Date-luokassa olisi mahdollista luoda vuosiluvuksi luokkamuuttuja, jotta vältytään turhalta toistolta ja vuosilukua ei tarvitsisi syöttää jokaiselle oliolle ja se olisi mahdollista muuttaa jokaiselle oliolle samaa aikaa. Luokkamuuttujaa kuvaa Javassa avainsana *static*.

Edellä olevassa esimerkissä (1.1.) käsiteltiin jo hieman jäsenmuuttujia ja sitä, miten ne esitellään luokan rakenteessa. Jäsenmuuttujat esitetään tavallisesti luokan määrittelyn alussa ja ne alustetaan viimeistään luokan rakentajassa. Jäsenmuuttujat siis pitävät sisällään tiedon olion tilasta ja tarpeellisesta tiedosta, josta olio muodostuu. Kuten esimerkissä on huomattavissa, koko date-olion toiminta perustuu täysin jäsenmuuttujien ympärille, mihin olio-ohjelmointi käytännössä perustuu. Jokaisella oliolla on omat jäsenmuuttujansa, jotka ovat näkyvissä ainoastaan oliolle itselleen ja näihin muuttujiin on mahdollista päästä käsiksi suoraan tai epäsuorasti olion erilaisilla metodeilla. Kuvattuun tiedon piilottamiseen on käytetty avainsanoja yksityinen (*private*) ja julkinen (*public*), joista enemmän seuraavassa.

1.7. Näkyvyysmääre: yksityinen (private) ja julkinen (public)

"*The visibility of a property or method can be defined by prefixing the declaration with the keywords public or private. Class members declared public can be accessed everywhere." ... "Members declared as private may only be accessed by the class that defines the member.* " php.net [URL: http://php.net/manual/en/language.oop5.visibility.php]

Näkyvyysmääreet ovat olio-ohjelmoinnin suola. Niiden avulla on mahdollista piilottaa tietoa olion toiminnasta muilta olioilta niin paljon kuin on tarpeen, mutta myös tarjota rajapinta ulospäin. Tällöin mahdollistetaan yksinkertainen rajapinta monimutkaisellekin toteutukselle paljastamatta olion toimintaa ulospäin. Näkyvyysmääreiden voidaan ajatella toimivan *need-to-know*-periaatteella, jolloin jokainen olio tarjoaa ulospäin juuri niin paljon tietoa, kuin ulkopuolisen on tarpeen tietää. Kaikki tieto, jota ulkopuolisen ei tarvitse tietää, pysyy visusti piilotettuna olion sisällä.

Julkinen ja yksityinen näkyvyysmääre tulee antaa jokaiselle luokan muuttujalle ja metodille, sillä olio tarvitsee tiedon siitä, voiko se esittää muuttujaa tai metodia ulospäin rajapinnassaan vai ei. Jäsenmuuttujat ovat tavallisesti olion yksityisiä muuttujia, kuten esimerkissä 1.2., sillä ne pitävät sisällään olion tilan ja sen tiedot, jotka on tarkoitettu vain olion itsensä käyttöön. Jos muuttujan näkyvyysmääre olisi julkinen, muuttujaa voisi muokata myös olion ulkopuolelta, mikä sotii olio-ohjelmoinnin perusaatetta vastaan. Periaate on, että oinen olio ei saa muokata toisen olion tilaa ilman, että muokattava olio on tietoinen muokkauksesta. Yksityisiä muuttujia tosin on joskus tarpeen muokata olion ulkopuolelta, mutta se ei ole syy vaihtaa muuttujan näkyvyysmäärettä. Yksityisen jäsenmuuttujan kanssa olio tarjoaa ulospäin *lukija- ja muuttajametodit* (*setter and getter methods*), joiden avulla muuttujaa voidaan muokata tai sen arvo voidaan lukea. Tämä voi kuulostaa monimutkaiselta ja ehkä turhankin hankalalta menettelyltä, mutta olio-ohjelmoinnissa tietoa pyritään suojaamaan mahdollisimman paljon.

1.8. Ja lopuksi Javaa

Jokaisen oppaan luvun lopuksi tulee lukuun liittyviä Java-esimerkkejä sekä Javan perustoiminnallisuuksia. Ensimmäisen luvun lopuksi käsitellään Javan tietotyypit ja avataan hieman luokan muodostamista.

1.8.1. Javan tietotyypit

Java, kuten C ja C++, on vahvasti tyypitetty kieli, joten jokaiselle muuttujalle on alustettava tietotyyppi ennen kuin sitä voidaan käyttää. Javassa on kahdeksan erilaista primitiivistä tietotyyppiä, jotka on kuvattu taulukossa 2.

Taulukko 2. Javan primitiiviset tietotyypit

Tietotyyppi	Koko	Vaihteluväli	Käyttö	Oletusarvo
byte	8 bittiä	-128 … 127	Suositellaan käytettäväksi suurissa taulukoissa muistin säästämiseksi	0
short	16 bittiä	-32,768 … 32,767	Muistin säästäminen	0
int	(oletus) 32 bittiä	$-2^{31} \ldots 2^{31} - 1$	Tavallisin kokonaislukutietotyyppi	0
long	64 bittiä	$-2^{63} \ldots 2^{63} - 1$	Kun int ei ole tarpeeksi suuri tarvittaviin lukuihin	0L
float	32 bittiä		Liukulukujen käyttö muistin säästämisessä	0.0f
double	64 bittiä		Tavallisin tietotyyppi liukuluvuille ja Javan oletustietotyyppi liukuluvuille	0.0d
boolean		false / true	Ehtojen toteutumissääntöihin ja lippuihin	false
char	16 bittiä	'\u0000' … '\uffff' (Unicode)	Merkkien säilömiseen	'\u0000'
String			Olio merkkijonojen säilömistä ja käyttöä varten	null

Javan perustietotyyppeihin lukeutuu myös merkkijonoluokka eli String, joka ei varsinaisesti ole primitiivinen tietotyyppi, mutta se luokitellaan sellaiseksi Javan tarjoaman tuen takia. String-luokka on poikkeuksellinen Javan luokka, sillä siitä olion rakentaminen on mahdollista ilman *new*-avainsanaa. Merkkijono-olio on mahdollista siis luoda yksinkertaisesti lainausmerkkejä käyttäen:

```
String s = "This is a string."
```

String-muotoiset oliot ovat täysin muuttamattomia (*immutable*), eli niihin ei voida asettaa uutta arvoa vaan luokasta on muodostettava aina uusi instanssi. Tämä ei kuitenkaan aiheuta muistivuotoja, sillä Java tuhoaa käyttämättömät oliot tietyin väliajoin automaattisen muistinhallinnan avulla.

Läheisesti Javan perustietotyyppeihin liittyvät myös luokat Integer ja BigDecimal. Integer-luokkaa voidaan hyödyntää lukujen kanssa, jos halutaan muuttaa niiden vaihteluväliä. Esimerkiksi int-tietotyypin vaihteluväli on tavallisesti $-2^{31} \ldots 2^{31} - 1$ (signed), mutta Integer-luokka mahdollistaa vaihteluvälin muuttamisen väliin $0 \ldots 2^{32} - 1$ (unsigned). Tämä on mahdollista myös tietotyypille long. Vaihteluvälin muuttaminen kasvattaa kokonaisluvun ylärajaa ns. poistamalla siltä alaraja eli siirtämällä alaraja nollaan. Tällöin suurin mahdollinen kokonaisluku, joka voidaan tallettaa Long-tyyppiseen muuttujaan on 18446744073709551615 ja suurin mahdollinen kokonaisluku Integer-tyyppisessä muuttujasssa on 4294967296. BigDecimal on hyödyllinen luokka desimaalilukujen kanssa, sillä tarkkoja arvoja käytettäessä, esimerkiksi valuutan kanssa, ei float- ja double-tietotyyppejä voida käyttää niiden epätarkkuuden ja pyöristysten takia.

1.8.2. Tietorakenne: Taulukko

Oppaan ensimmäisessä luvussa käsitellään Javan tietorakenteista yksinkertaisin eli taulukko, joka tietorakenteena tulisi olla jokaiselle tämän oppaan lukijalle jo tuttu. Taulukko on olio, joka pitää sisällään kiinteän määrän tietyn tyyppistä dataa, joita kutsutaan elementeiksi. Jokaisella taulukon elementillä on indeksinumero, jonka avulla sen sisältöön päästään käsiksi. Tämä mahdollistaa esimerkiksi koko taulukon läpikäymisen yhdellä *for*-silmukalla. Indeksinumerointi itsessään alkaa nollasta, jolloin ensimmäisen elementin indeksi on 0 ja esimerkiksi kahdeksannen elementin indeksi on 7. Taulukko on myös staattinen tietorakenne, eli luomisen jälkeen taulukon koko on kiinteä ja sitä ei ole mahdollista muuttaa. Esimerkki selventää:

Esimerkki 1.4. Taulukon luonti ja alustus

```
public class Array {
   public static void main(String[] args) {
      // declare an array
      int[] array;

      // set arrays length to 10
      array = new int[10];

      // initialize the array
```

```
        for(int i = 0; i < array.length; i++) {
            array[i] = 100 * i;
        }

        // print out the elements
        for(int i = 0; i < array.length; i++) {
            System.out.println("Element at index " + i + ": " + array[i]);
        }
    }
}
```

Tuloste

```
run:
Element at index 0: 0
Element at index 1: 100
Element at index 2: 200
Element at index 3: 300
Element at index 4: 400
Element at index 5: 500
Element at index 6: 600
Element at index 7: 700
Element at index 8: 800
Element at index 9: 900
BUILD SUCCESSFUL (total time: 0 seconds)
```

Taulukkoesimerkki on luokka nimeltä Array, joka sisältää vain pääfunktion main(), eli se muodostuu aivan samalla tavalla kuin mikä tahansa yksittäinen C-kielinen ohjelma. Luokan sisällä alustetaan viite taulukkoon array, joka tulee pitämään sisällään kokonaislukuja. Seuraavaksi viitteen päähän alustetaan 10 alkion taulukko ja taulukkoon lisätän kokonaislukuarvoja for-silmukan avulla. Lopuksi esimerkki tulostaa kaikki taulukkoon tallennetut arvot uuden silmukan avulla käyttäen Javan yleistä tulostusfunktiota System.out.println().

1.8.3. System.out.println()

Edellisessä esimerkissä havaitaan hieman mielenkiintoisen näköinen tulostusmetodi: System.out.println(). Alkuosa tuosta funktiosta, eli System.out on standarditietovirta (käsitellään seuraavassa luvussa), jonka avulla voidaan lähettää merkkijonoja vastaanottavalle laitteelle, tässä tapauksessa näytölle. println() on vuorostaan funktio, jonka avulla on mahdollista lähettää tietovirralle merkkijono ja funktio lisää automaattisesti rivinvaihdon merkkijonon perään. Merkkijonoa on mahdollista kasvattaa pidemmäksi käyttämällä +-merkkiä liitettävien osien välillä, mutta tällöin on muistettava, että +-merkki ei lisää välilyöntiä tulostukseen, vaan se on erikseen kirjoitettava mukaan liitettäviin merkkijonoihin. Tämä toiminnallisuus on mahdollista välttää käyttämällä System.out.print()-funktiota, joka tulostaa annetun rivin ilman rivinvaihtoa.

Taulukon hyödyllisiä kirjastometodeja, joiden kaikki mahdolliset parametrit on mahdollista löytää Oraclen Javadokumentaatiotsta osoitteesta:
http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html

- binarySearch()
  - Etsimismetodi, jonka avulla taulukosta voidaan etsiä elementtejä. Ottaa parametreikseen taulukon sekä etsittävän tiedon ja palauttaa löytyneen elementin indeksin tai negatiivisen luvun, jos indeksiä ei löydy.
- copyOf()
  - Kopioi taulukon ja antaa kopiolle uuden pituuden. Ottaa parametrikseen kopioitavan taulukon sekä uuden pituuden ja palauttaa taulukon kopion.
- equals()
  - Metodi, jonka avulla voidaan tarkistaa onko taulukko identtinen toisen taulukon kanssa. Ottaa parametrikseen vertailtavat taulukot.
- sort()
  - Järjestää taulukon alkiot kasvavaan numerojärjestykseen, vaatii parametrikseen järjestettävän taulukon.
- toString()
  - Muuttaa taulukon sisällön String-muuttujaksi.

On huomattava, että **taulukko on hyvin rajallinen tietorakenne** ja Java sisältää useita dynaamisempia tietorakenteita, kuten vektorin (*java.util.Vector*) ja listan (*java.util.ArrayList*), joista lisää seuraavassa luvussa.

1.9. Kerrataan lopuksi

Oppaan tässä luvussa käsiteltiin luokan ja olion määritelmiä sekä luokan fyysistä rakentamista. Luokka määritellään Javalla:

```
public class Example {
}
```

Luokan jäsenmuuttujat määritellään tavallisesti luokassa ensimmäisenä ja niiden näkyvyysmääre on yksityinen:

```
public class Example {
   private int ex_1;
   private double ex_2;
   private String ex_3;
}
```

Jäsenmuuttujien jälkeen luokalle muodostetaan rakentajat, jossa alustetaan luokan jäsenmuuttujat joko oletusarvoilla tai rakentajaan saapuvilla parametreilla ja niiden näkyvyysmääre on julkinen:

```
public class Example {

   private int ex_1;
   private double ex_2;
   private String ex_3;

   public Example() {
      ex_1 = 5;
      ex_2 = 10.228;
      ex_3 = "Example";
   }

   public Example(int par1, double par2, String par3) {
      ex_1 = par1;
      ex_2 = par2;
      ex_3 = par3;
   }
}
```

Lopuksi luokka tarvitsee vielä lukija- ja muuttajametodit (*setter- and getter-methods*), joiden näkyvyysmääre on myös julkinen. Metodit on asetettu tavallista tiiviimmin tilan säästämiseksi ja niiden yksinkertaisen toiminnan takia:

```
public class Example {

   private int ex_1;
   private double ex_2;
   private String ex_3;

   public Example() {
      ex_1 = 5;
      ex_2 = 10.228;
      ex_3 = "Example";
   }

   public Example(int par1, double par2, String par3) {
      ex_1 = par1;
      ex_2 = par2;
      ex_3 = par3;
   }

        public int getInt() {return ex_1;}
        public void setInt(int new_int) {ex_1 = new_int;}
        public double getDouble() {return ex_2;}
        public void setDouble(double new_double) {ex_2 = new_double;}
        public String setString() {return ex_3;}
        public void setString(String new_string) {ex_3 = new_string;}
}
```

Jos luokasta haluaa muodostaa ajettavan instanssin, on ohjelmaan, **luokan sisään**, lisättävä main-funktio aivan niin kuin C-kielessä. Tämä on yksinkertaisesti luokan määrittelyn loppuun lisättävä pätkä:

```
public static void main(String[] args) {
    // TODO code application logic here
}
```

1.9.1. Main-funktion käyttö

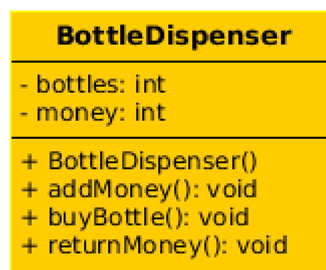
Tässä yhteydessä on huomattava, että jokainen luokka ei tarvitse toimiakseen erillistä main-funktiota, vaan koko ohjelma tarvitsee vain yhden pääluokan, jonka sisällä on main-funktio. Eli ohjelmassa voi olla 15 luokkaa, mutta siinä on vain yksi pääluokka eli ajettava luokka, joka pitää sisällään main-funktion. Näin ohjelmassa olevat luokat toimivat luotavien olioiden alustoina ja niitä käytetään tarpeen mukaan pääluokasta.

Esimerkissä esitetty luokka on hyvin yksinkertainen ja karkea, mutta luokkaan liittyy myös erilaisia yksityisiä metodeita, jotka eivät näy ulospäin. Näiden metodien tarkoituksena on auttaa olion rajapinnan rakentamisessa, sisäisessä toiminnassa sekä muiden käskyjen toteutuksessa. Esimerkkejä tällaisista luokista on oppaan tulevissa luvuissa. Seuraavassa luvussa puhutaan lisää luokkien toiminnasta ja jäsenmetodeista sekä tyyppimuunnoksista.

2. Lisää luokista ja muuttujista

"*Classes are an expanded concept of data structures: like data structures, they can contain data members, but they can also contain functions as members.*" - Cplusplus.com [URL: http://www.cplusplus.com/doc/tutorial/classes/]

Jatketaan luokista siitä, mihin viime luvussa jäimme. Kuten muistamme, luokka koostuu jäsenmuuttujista, rakentajista sekä lukija- ja muuttajametodeista. Tosin, jos luokat olisivat vain näin yksinkertaisia rakenteita, ei olio-ohjelmointi olisi loppujen lopuksi kovinkaan ekspressiivinen tapa ohjelmoida. Pelkkien yksinkertaisten jäsenmuuttujien ja rakentajien kanssa toimiminen muuttuu nopeasti hyvinkin suppeaksi, joten esittelemme oppaan tässä luvussa enemmän mahdollisuuksia luokan muodostamiseen. Käytetään esimerkkinä yksinkertaista limsa-automaattia:



Kuva 2.1. Limsa-automaatti

Kuten kaaviosta nähdään, limsa-automaatilla on rakentaja BottleDispenser(), kaksi jäsenmuuttujaa bottles ja money sekä kolme jäsenmetodia. Esimerkki on vielä yksinkertainen, joten luokalta puuttuvat lukija- ja muuttajametodit, mutta ovatko ne tarpeen? Keskustellaan asiasta myöhemmin hieman lisää. Muodostetaan aluksi luokka Javan avulla:

Esimerkki 2.1. Limsa-automaatti Javalla

```
public class BottleDispenser {

   private int bottles;
   private int money;

   public BottleDispenser() {
      bottles = 50;
      money = 0;
   }

   public void addMoney() {
      money += 1;
```

```
        System.out.println("Klink! Money was added into the machine!");
    }

    public void buyBottle() {
        bottles -= 1;
        System.out.println("KACHUNK! Bottle appeared from the machine!");
    }

    public void returnMoney() {
        money = 0;
        System.out.println("Klink klink!! All money gone!");
    }
}
```

Luokan rakenne on hyvin yksinkertainen - tosin ei limsa-automaattikaan kovin monimutkainen laite ole. Tässä kyseisessä automaatissa on luotaessa sisällä 50 pulloa ja 0 rahaa ja siinä on mahdollista syöttää rahaa, ostaa pulloja sekä tyhjentää rahat automaatista. Metodit - eli jäsenfunktiot - ovat tässä esimerkissä jokainen määritetty näkyvyydeltään julkisiksi, jotta muut luokat voivat pyytää oliolta toiminnallisuuksia. Jos ajatellaan elävän elämän limsa-automaattia, addMoney()-metodi aktivoituu, kun käyttäjä syöttää kolikon automaattiin, buyBottle()-metodi aktivoituu painiketta painettaessa ja returnMoney() palautuspainiketta painettaessa. Tällöin voitaisiin ajatella metodien olevan yksityisiä, sillä käyttäjä ei pääse metodeihin kiinni vaan painikkeisiin, mutta jos olio-ohjelmoinnissa halutaan muiden olioiden käyttävän olion metodeita (kuvitteelliset painikkeet tässä tapauksessa), on metodien oltava näkyviä ulospäin. Metodien piilottaminen olisi kuin painikkeiden ja valuuttakolon piilottaminen automaatin sisäpuolelle.

Ajatellaan luokkarakennetta hieman pidemmälle. Jokainen automaatissa oleva pullohan on oma olionsa, joten eikö olisi mukavaa käyttää myös ohjelmatasolla olioita osoittamaan pullojen laatua ja määrää. Pullolla on tiedossa sen valmistaja ja sisällön nimi sekä sisällön kokonaisenergiamäärä. Voidaan siis luoda luokka Pullo:

Bottle

- name: String
- manufacturer: String
- total_energy: int

+ Bottle()
+ Bottle(String name, String manuf, int totE)
+ getName(): String
+ getManufacturer(): String
+ getEnergy(): int

Kuva 2.2. Pullo-luokka

Pullo-luokalla on nyt yksityiset muuttujat nimi, valmistaja ja kokonaisenergiamäärä. Tämän lisäksi pullolla on kaksi rakentajaa - oletusrakentaja ja parametrillinen rakentaja - sekä kolme lukijametodia. Muuttajametodeja ei kyseisellä luokalla ole olemassa, sillä pullon ominaisuuksia on mahdoton muuttaa sen jälkeen kun se on luotu ja täytetty. Nyt voimme siis käyttää Pullo-olioita, joilla voimme täyttää pulloautomaatin tämän rakentajassa.

Ennen koodiesimerkkiä puhutaan tovi myös pulloautomaatin lukija- ja muuttajametodeista. On totta, että pullokone on oltava mahdollista täyttää, muuten se olisi kovin kertakäyttöinen. Pullokoneen täyttö tosin ei ole laitteen oma toiminto, vaan sen tekee laitteen puolesta toinen olio. Pullojen lukumäärä (pl. pullojen loppuminen) ei kosketa yhtään oliota koneen ulkopuolella, joten kenenkään ei tarvitse tietää pullokoneen pullojen määrää. Sama pätee koneen sisältämään rahamäärään, joka jo turvallisuussyistä on suotavaa pitää piilotettuna ulkopuolisilta. Tämä siis tarkoittaa, että pullokone ei tarvitse lukija- tai muuttajametodeita muuttujilleen.

Esimerkki 2.2. Pulloautomaatti Javalla

```
public class BottleDispenser {

    private int bottles;
    // The array for the Bottle-objects
    private Bottle[] bottle_array;
    private int money;

    public BottleDispenser() {
        bottles = 50;
        money = 0;

        // Initialize the array
        bottle_array = new Bottle[bottles];
        // Add Bottle-objects to the array
        for(int i = 0;i<bottles;i++) {
            // Use the default constructor to create new Bottles
            bottle_array[i] = new Bottle();
        }
    }

    public void addMoney() {
        money += 1;
        System.out.println("Klink! Money was added into the machine!");
    }

    public void buyBottle() {
        bottles -= 1;
        System.out.println("KACHUNK! Bottle appeared from the machine!");
    }
```

```
    public void returnMoney() {
        money = 0;
        System.out.println("Klink klink. All money gone!");
    }
}
```

Esimerkkiin on nyt lisätty taulukko, johon tallennetaan luotuja Pullo-olioita. Tämä vaatii tietotyypin asettamisen Pullo-olioksi, jotta ko. pulloja on mahdollista lisätä koneeseen. Kyseistä luokkaa on mahdollista laajentaa edelleen, mutta esimerkin rajoissa emme sitä tässä tee. Esimerkissä myös havaitaan, että buyBottle()-metodissa limsa-automaatista vähennetään pullojen määrää, mutta säilytettäviin pulloihin tämä ei ohjelmassa vaikuta. Tarvittaisiin siis menetelmä, jonka avulla voisimme vähentää taulukosta yhden pullon aina, kun pullo myydään pois. Siitä seuraavassa.

2.1. Olion tuhoaminen

"*You are terminated.*" - T-800, Terminator

Edellisessä esimerkissä luotiin taulukko, joka piti sisällään kaikki koneessa olevat Pullo-luokan instanssit. Kun ajatellaan pulloautomaatin toimintaa, käyttäjän ostaessa pullon, pullojen määrä koneessa vähenee yhdellä ja samalla yksi Pullo-instanssi häviää koneesta ja ilmestyy koneen ulkopuolelle. Esimerkissä pullojen määrä vain väheni ilman, että Pullo-olioiden määrä vähentyi. Tähän siis tarvittaisiin toiminto, joka "tuhoaisi" Pullo-olion koneen sisältä, kun se siirtyy koneen ulkopuolelle. Kyseisen toiminto on rakentaja vastakohta eli *purkaja* (*destructor*).

2.1.1. Purkaja ja muistin vapauttaminen

"*In object-oriented programming, a destructor is a method which is automatically invoked when the object is destroyed.*" - Wikipedia
[URL: http://en.wikipedia.org/wiki/Destructor_(computer_programming)]

Kuten jokaisella luokalla on oltava rakentaja, on sillä oltava myös purkaja, jotta olio voidaan hallitusti tuhota ja sen käyttämä muisti vapauttaa. Purkaja ei tuhoa itse oliota vaan valmistelee olion poistettavaan kuntoon sulkemalla kaikki avoinna olevat tietovirrat ja vapauttamalla olion käyttämät resurssit. Tällöin olion hävittämisen yhteydessä tulisi hävittää myös kyseiseen olioon liittyvät oliot, koosteoliot. Esimerkkitapauksen pulloautomaattia hävitettäessä tulisi ensin hävittää sen sisällä olevat Pullo-oliot.

Monet oliokielet tarjoavat mahdollisuuden määritellä luokalle purkamisoperaatio, jota kutsutaan kun oliota ollaan hävittämässä. Varsinkin C++-kielessä olion purkamisoperaatio on välttämätön

resurssien hallinnan vuoksi, sillä se ei sisällä mitään automaattista muistinhallintaa, toisin kuin esimerkiksi Java. Java on ohjelmointikieli, joka käyttää automaattista muistinkäsittelyä apunaan olioiden tuhoamiseen. Tämä tarkoittaa sitä, että C++:ssa voi ohjelmoija itse määrittää hetken, milloin olio tuhotaan, mutta Javassa on mahdollista vain asettaa olio tuhottavaksi ja järjestelmä tuhoaa olion automaattisesti sitten, kun näkee sen itse tarpeelliseksi.

2.1.2. finalize()

"*Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.*" - Javadoc
[URL: http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#finalize()]

Vaikka Javassa ei ole mahdollista määrittää täsmällisesti, milloin olio tuhotaan, on sille silti mahdollista luoda niin kutsuttu purkaja, *finalize()*-metodi. Luokalle luodaan parametriton finalize()-jäsenmetodi, jota järjestelmä kutsuu automaattisesti silloin, kun oliota ollaan tuhoamassa. finalize() vastaa siitä, että olioon liittyvät tietovirrat suljetaan ja muut resurssit vapautetaan juuri ennen olion tuhoamista. Javassa myös osaolioiden tuhoaminen suoritetaan automaattisesti myöhemmin, jos niitä ei ole viitattu mihinkään muualle (viittauksisa heti seuraavaksi), eli ne pyörivät tietokoneen muistissa ilman viittausta mihinkään olioon.

Esimerkissä 2.2. tarvittaisiin jokin metodi, jonka avulla pullojen määrää koneessa voitaisiin vähentää. Koska kyseisessä luokassa on käytössä niinkin vajavainen tietorakenne kuin taulukko, joudutaan Pullo-olioiden poistaminen tekemään epäsuorasti kopioimalla alkuperäisen taulukon tiedot yhden elementin lyhyempään taulukkoon.

Esimerkki 2.3. Pullo-olion tuhoaminen

```
private void deleteBottle() {
   bottle_array = copyOf(bottle_array, bottles);
}
```

Jos limsa-automaatti olisi käyttänyt tietorakenteenaan listaa tai vektoria, olisi pullon poistaminen hyvin paljon suoraviivaisempaa.

Olion tuhoamisen kannalta käytettävä tietorakenne on merkityksetön, sillä prosessi toteuttaa aina saman tehtävän: hävittää viittauksen olioon. Kuten aikaisemmin todettiin, Javan muistinkäsittelijä tuhoaa olion, johon ei viitata, automaattisesti. Ohjelmoija ei voi tuhota oliota; ainoastaan viittaukset olioihin voidaan tuhota ja sitä kautta olio tuhoutuu ajan myötä.

2.2. Viittaukset olioihin

"*Olioihin pääsee käsiksi viittausten eli referenssien (reference) avulla. Joissakin teoksissa viittaus on suomennettu sanalla viite. Viittaus on siis osoite olioon.*" - PHKK:n opetusmateriaali [URL: http://edu.phkk.fi/opiskelu/ohjperjava/olio_ohj_per.htm]

Viittaaminen muistuttaa käytännön tasolla hyvin paljon arvon sijoittamista muuttujaan, mutta kuten edellä olevassa lainauksessa on sanottu, viittaus viittaa olioon, se ei ole arvon asettamista muuttujaan. Oliot ovat tavallisesti hyvinkin erikokoisia ja vievät muistissa vaihtelevan määrän tilaa, minkä vuoksi niille ei voida määritellä yhtä tietotyyppiä, vaan ne ovat aina omia tietotyyppejään. Käytetään esimerkkinä String-luokkaa:

```
String name;
```

Edellä on luotu viittaus String-olioon ja sen sisältönä on null. Seuraavaksi luodaan uusi olio, johon viittaus liitetään:

```
name = “Erno”;
```

Viittaus viittaa nyt String-luokasta luotuun olioon, jonka sisältönä on merkkijono "Erno". Jos haluamme muuttaa olion sisältämää merkkijonoa, on se mahdotonta, sillä String-oliot ovat muuttamattomia, mutta voimme muuttaa viittausta:

```
name = “Timo”;
```

Nyt muistialueella on olemassa kaksi String-oliota, "Erno" ja "Timo" ja viittaus name viittaa olioon "Timo", kun taas olioon "Erno" ei viittaa mikään. Tällöin järjestelmä toteaa seuraavalla tarkastuskierroksella, että "Erno"-olioon ei ole yhtäkään viittausta ja sitä ei ole tarpeen säilyttää, joten se tuhotaan.

Muuttamaton ja muutettavissa oleva (*immutable and mutable*)
Oppaassa on nyt muutamaan kertaan mainittu muuttamaton olio. Mitä tämä tarkoittaa? Muuttamattoman olion sisältöä ei voida muuttaa, sillä se ei sisällä muuttajametodeita ja se voidaan ainoastaan luoda tai tuhota. Jos olion sisältöä on tarpeen muokata, tällöin on luotava uusi olio ja siirrettävä nykyinen viittaus uuteen olioon. Muutettavissa olevan olion sisältöä on mahdollista muuttaa tarjottujen muuttajametodien avulla. Java toimii tässä suhteen siis hieman samalla tavalla kuin Python ja eritavalla kuin C tai C++.

2.2.1. Viittaus ja osoitin

Osoitin eli pointteri pitäisi olla oppaan lukijalle jo tuttu käsite. Osoitin on muuttuja, joka ilmoittaa tietyn kohdan tietokoneen muistissa. Esimerkiksi C++-kielessä on mahdollista käyttää sekä

osoittimia että viittauksia ja on huomattavaa, että osoittimilla voidaan tehdä enemmän kuin viittauksilla - enemmän toiminnallisuutta ja tuhoa. Osoittimen ja viittauksen ero on pääasiassa se, että osoitin osoittaa aina tiettyyn muistipaikkaan, kun taas Javassa viittaus osoittaa aina olioon, oli se missä muistipaikassa tahansa. Osoittimen päässä olevaa tietoa voidaan siis muuttaa ilman, että se vaikuttaa osoittimeen, mutta viittauksella näin ei voida tehdä, sillä muutokset tapahtuvat viittauksen kautta. Tähän on kuitenkin suhtauduttava varauksella, sillä viittauksen päässä oleva tieto saattaa muuttua jostakin muustakin syystä kuin vain ohjelman takia, esimerkiksi käyttöjärjestelmän toimesta.

2.3. Tyyppimuunnoksia

"*Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a long value into a simple integer then you can type cast long to int.*" - C-tutorial [URL: http://www.tutorialspoint.com/cprogramming/c_type_casting.htm]

Muuttujien tyyppimuunnokset ovat vahvasti tyypitetyillä ohjelmointikielillä ohjelmoidessa tavallisia ja erittäin tarpeellisia. Esimerkiksi käyttäjiltä pyydetty syöte on aina merkkijono, joten sen käyttäminen laskennassa vaatii tyyppimuunnoksen. Kuitenkin tyyppimuunnos pitää sisällään aina vaaroja, sillä esimerkiksi liukuluvun muuttaminen kokonaisluvuksi aiheuttaa tiedon häviämistä (tässä tapauksessa pyöristämistä), mikä ei ole suotavaa. Lisäksi myös muunnokset suuremmista tietotyypeistä pienempiin voivat aiheuttaa sen, että muuttujassa oleva data ei mahdu uuteen tietotyyppiin, kuten esimerkiksi int-muuttujan muuntaminen short-muotoon ei ole Javassa tuettua ilman eksplisiittistä tyyppimuunnosta, mutta silti mahdollista. Vaikka tällaisen tyyppimuunnoksen tekeminen on mahdollista, tulisi ohjelmoijan aina valvoa, että tahatonta tiedon häviämistä ei tapahdu.

Tyyppimuunnokset ovat mahdollisia ja tarpeellisia myös olioiden välillä, varsinkin kun olio on luotu käytettävän luokan kantaluokasta, josta puhumme enemmän periytymisen yhteydessä.

2.4. Lopuksi Javaa

Tämän luvun lopuksi on kasattu esimerkkejä siitä, miten tyyppimuunnokset hoituvat Javan avulla sekä käydään läpi tietorakenteet lista ja vektori sekä kerrataan hieman, mitä opimme.

2.4.1. Tyyppimuunnoksia muuttujilla

Javassa sallittuja sijoituksia tyypistä riippumatta ovat sijoitukset pienemmästä tietotyypistä suurempaan, eli esimerkiksi short-muuttujan voi asettaa int-muuttujaan ja float-muuttujan voi asettaa double-muuttujaan.

Esimerkki 2.4. Tietotyyppien sijoittaminen

```
short _short  = -23;
int _int    = 123;
long _long   = 121234;
char _char   = 'q';
float _float  = 3.14f;
double _double = 121.2311;

// sallittuja         kiellettyjä
_int    = _short;       _short = _int;
_long   = _int;         _int   = _long;
_double = _short;       _short = _double;
_double = _float;       _float = _double;
_int = _char;           _char  = _int;
_double = _char;        _short = _char;
                        _char  = _short;
```

Mikäli kuitenkin on tarpeen tehdä kiellettyjä tyyppimuunnoksia ja suoraan sijoittaminen ei ole mahdollista, esimerkiksi int-muuttuja long-muuttujaksi tai double-muuttujasta float-muuttujaksi, onnistuu se eksplisiittisellä tyyppimuunnoksella. Tällöin on kuitenkin oltava huolellinen, jotta tietoa ei häviä tyyppimuunnosten takia vain siksi, että sijoitettava luku on liian suuri sopimaan haluttuun tietotyyppiin.

Esimerkki 2.5. Eksplisiittinen tyyppimuunnos

```
short  _short  = -23;
int    _int    = 123;
long   _long   = 121234;
char   _char   = 'q';
float  _float  = 3.14f;
double _double = 121.2311;

_short = (short)_int;
_int   = (int)_long;
_short = (short)_double;
_float = (float)_double;
_char  = (char)_int;
_short = (short)_char;
_char  = (char)_short;
```

Järkevissä tapauksissa nämä tyyppimuunnokset ovat täysin perusteltuja, kunhan vain ohjelmoija pitää tarkasti hallussa sen, että muunnettavat arvot eivät hukkaa tietoa siksi, että ne eivät sovi

uuteen tietotyyppiin. Nämä tietotyyppeihin tallennettavat rajat on nähtävissä ensimmäisen luvun Javan tietotyyppitaulukossa.

Ohjelmoijaa saattaa myös kiinnostaa merkkijonojen muuntaminen erilaisiksi luvuiksi esimerkiksi käyttäjän syötettä pyydettäessä. Jotta olisi mahdollista luoda merkkijonosta luku, joudutaan käyttämään hyödyksi Javan wrapper-luokkia, joihin on kiedottu sisään primitiivisen tietotyypin tieto. Wrapper-luokkien tunnusmerkkinä ohjelmoitaessa on se, että nimeltään ne ovat identtiset primitiivisten tietotyyppien kanssa (poikkeuksena int/Integer ja char/Character), mutta ne alkavat isolla kirjaimella.

Esimerkki 2.6. Wrapper-luokat

```
Short _short = new Short((short)-23);
Integer _int = new Integer(123);
Long _long = new Long((long)121234);
Character _char = new Character('q');
Float _float = new Float(3.14);
Double _double = new Double(121.2311);
String _string = "Hello world!";

_short = _int.shortValue();
_int = _long.intValue();
_short = _double.shortValue();
_float = _double.floatValue();
_string = _char.toString();
_string = _double.toString();
_int = Integer.parseInt(_string);
_double = Double.parseDouble(_string);
```

Kuten voi huomata, wrapper-luokkien kanssa toimiminen ei ole aivan niin selkeää kuin pelkillä primitiivisillä tietotyypeillä. Tietotyyppien muuntaminen onnistuu käyttämällä wrapper-olioiden jäsenmetodeja, jotka eivät ota parametreja, mutta palauttavat halutun tietotyypin. Sama pätee merkkijonoksi muuttamiseen, sillä toString()-metodi on periytetty kantaoliosta Object, eli se löytyy jokaisesta wrapper-oliosta. Rakentajat Short() ja Long() vaativat eksplisiittisen tyyppimuunnoksen, sillä Javassa kokonaisluvun oletustietotyyppi on int ja Short ja Long luokkien rakentajat vaativat short- ja long-tyyppiset luvut.

Enemmän jäsenmetodeita ja informaatiota wrapper-luokista löytyy javadocista.

Integer: http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html

Boolean: http://docs.oracle.com/javase/7/docs/api/java/lang/Boolean.html

2.4.2. Tietorakenne: Lista ja vektori

Lista ja vektori ovat hyvin samankaltaisia *java.util.*-kirjastosta löytyviä tietorakenteita. Oppaassa listalla tarkoitetaan Javan ArrayList-luokkaa, ei List-luokkaa, sillä ArrayList implementoi List-luokan, eli ArrayList sisältää kaiken sen toiminnallisuuden, mitä List sisältää. Tämä pätee myös Vector-luokkaan, joka myös implementoi Listin, eli käyttää List-luokan tarjoamaa rajapintaa. Implementointi, periytyminen ja rajapinnat käsitellään oppaan tulevissa luvuissa. Vektorin ja listan pääasiallinen ero on se, että vektori on synkroninen tietorakenne kun taas lista ei ole, minkä vuoksi on tärkeää pohtia tapauskohtaisesti kumpaa tietorakennetta käytetään. Tästä johtuen lista on nopeampi tietorakenteena vektoriin verrattuna, eli se on nopeampi iteroida läpi ja käyttää. Tämä vertailu ei kuitenkaan kuulu kurssin aihealueisiin eikä tavoitteisiin, joten jätämme asian syvällisemmän pohdinnan tähän.

Vaikka Java onkin C:n ja C++:n kaltaisesti vahvasti tyypitetty kieli, voi listaan ja vektoriin silti asettaa eri tietotyyppejä omaavia muuttujia kuten Pythonissa. Esimerkiksi samaan listaan voi laittaa sekä merkkijonoja, kokonaislukuja ja liukulukuja, mutta niiden iteroinnissa on oltava erittäin tarkkana, ettei ohjelma synnytä ajonaikaista virhettä. On totta, että eri tyyppisiä muuttujia on mahdollista säilöä samaan tietorakenteeseen, mutta jos jokaiselle alkiolle pyrkii suorittamaan saman tehtävän, voi ohjelma hyvin nopeasti törmätä yhteensopivuusongelmiin, pyrkien etsimään esimerkiksi kokonaisluvulta merkkijonon metodia. Tämä ei kuitenkaan ole ohjelmoinnin kannalta suositeltava tapa luoda tietorakenteita, vaikka niin on mahdollista tehdä. Myöhemmin käydään läpi oikeaoppinen tapa alustaa tietorakenne. Käytetään esimerkkinä listan iterointia, jossa halutaan pystyä tulostamaan ensin listan alkion sisältämä arvo ja sen jälkeen tietotyyppi. Tietotyyppi TYPE on Javan wrapper-luokan luokkamuuttuja, eli se on yhteinen jokaiselle luokasta luodulle instanssille.

Esimerkki 2.7. Sekalaista tietoa tietorakenteessa

```
import java.util.ArrayList;

public class ListSample {

   public static void main(String[] args) {
      ArrayList list = new ArrayList();
      list.add("Hello world!");
      list.add(69);
      list.add(3.14);
      list.add("IT can be fun!");

      String cond = "Hello world!";

      for(int i = 0; i < list.size(); i++) {
         if(list.get(i).equals(cond)) {
```

```
            System.out.println(list.get(i));
        }
        System.out.println(((Integer) list.get(i)).TYPE);
    }
  }
}
```

Tuloste

```
run:
Hello world!
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be
cast to java.lang.Integer
        at listsample.ListSample.main(ListSample.java:26)
Java Result: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

Esimerkissä näemme, kuinka tietorakenteeseen voidaan lisätä ensin merkkijono, sitten kokonaisluku, liukuluku ja taas merkkijono, eli String, Integer, Double, String. Koska lista tallentaa nämä tietotyypit Object-muodossa, eli kaikkien luokkien kantaluokkana (asiaan palataan periytymisessä), niitä on mahdollista tallentaa rakenteeseen. Tällöin esimerkiksi tiedon tulostaminen toimii yksinkertaisesti System.out.println()-funktion avulla. Jos listasta halutaan tulostaa myös alkion tietotyyppi, kohdataan suuria ongelmia. Kuten edellä sanottiin, TYPE on wrapper-luokan ominaisuus, jolloin sen tulostaminen vaatii eksplisiittisen tyyppimuunnoksen, tällä kertaa oliona. Ongelmana on, että kaikki alkiot eivät ole Integer-tyyppisiä, vaan ensimmäisenä iteraatiosta tulee vastaan String. Tällöin kääntäjä ei voi tehdä tyyppimuunnosta ja palauttaa virheen:

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be
cast to java.lang.Integer
        at listsample.ListSample.main(ListSample.java:29)
Java Result: 1
```

Samankaltaiseen tilanteeseen jouduttaisiin, jos jokaiselle alkiolle olisi suoritettava vain String-oliolla oleva metodi. Joudumme siis toteamaan tässä vaiheessa, että vaikka usean tietotyypin mahduttaminen samaan tietorakenteeseen on mahdollista, ei se käytännön näkökulmasta ole kovinkaan järkevää.

Esimerkissä tulee ilmi myös Javan erikoinen tapa vertailla olioita, sillä tavanomainen, primitiivisillä tietotyypeillä toimiva vertailu “==” ei toimi olioiden kanssa. Javassa on rakennettu Object-luokan metodi equals(), jonka avulla on mahdollista vertailla olioiden sisältöä.

Esimerkki 2.8. Olioiden vertailu

```
public static void main(String[] args) {
    String line1 = "Hello World!";
    String line2 = new String("Hello World!");
```

```
        Integer int1 = new Integer(47);
        Integer int2 = new Integer(47);

        System.out.println(line1 == line2);
        System.out.println(line1.equals(line2));
        System.out.println(int1 == int2);
        System.out.println(int1.equals(int2));

        int1 = 47;
        int2 = 47;

        System.out.println(int1 == int2);
    }
```

Tuloste

```
run:
false
true
false
true
true
BUILD SUCCESSFUL (total time: 0 seconds)
```

Vahinkoja tietysti sattuu ja tapahtuu, jolloin ei voida olla varmoja, minkä tyyppistä tietoa ohjelma lisää listaan, varsinkin jos luemme datan tiedostosta ja lisäämme alkioita suoraan listaan. Tietorakenne on mahdollista tiivistää niin, että sinne ei voi lisätä kuin tietyn tietotyypin muuttujia alustamalla se ko. tietotyypin mukaan. **Tämä on ohjelmoinnin kannalta suositeltava tapa.**

```
ArrayList<Integer> list = new ArrayList();
```

Tällöin ajonaikainen virhe tulee jo tiedostosta luettaessa, jolloin voimme lisätä lukijametodiin virheenkäsittelijän, joka huomaa poikkeustapauksen ja ilmoittaa siitä käyttäjälle kaatamatta ohjelmaa. Virheiden käsittelyyn palataan myöhemmin luvussa 6.

Vektori ja lista sisältävät useita yhteisiä ja hyödyllisiä metodeja, joista tässä esitellään muutamia:

- add()
  - Lisää parametrina annetun elementin rakenteen loppuun tai parametrina annettuun indeksiin.
- clear()
  - Tyhjentää koko rakenteen.
- contains()
  - Palauttaa arvon `true`, jos rakenteesta löytyy parametrina annettu elementti.
- get()
  - Palauttaa elementin parametrina annetusta indeksistä.
- indexOf()

  - Palauttaa parametrina annetun elementin ensimmäisen instanssin indeksin tai -1, jos elementtiä ei löydy.
- isEmpty()
  - Palauttaa true, jos rakenne on tyhjä.
- remove()
  - Poistaa parametrina annetun indeksin tai elementin rakenteesta.
- set()
  - Korvaa rakenteessa olevan elementin parametrina annetulla elementillä parametrina annetussa indeksissä.
- size()
  - Palauttaa rakenteen koon.

Täysi lista listan ja vektorin metodeista:

- http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html
- http://docs.oracle.com/javase/7/docs/api/java/util/Vector.html

Esimerkki 2.9. Lista ja vektori, perustoiminnallisuuksia

```
import java.util.ArrayList;

ArrayList<String> al = new ArrayList<String>();
System.out.println("Initial size of al: " + al.size());

al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
al.add(1, "A2");
System.out.println("Size of al after additions: " + al.size());

System.out.println("Contents of al: " + al);
al.remove("F");
al.remove(2);
System.out.println("Size of al after deletions: " + al.size());
System.out.println("Contents of al: " + al);
```

Tuloste

```
run:
Initial size of al: 0
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]
```

```
BUILD SUCCESSFUL (total time: 0 seconds)
```

Esimerkki toimii täysin identtisenä vektorille, kunhan viitteen tyyppi ja rakentaja vaihdetaan Vector-tyyppiseksi.

2.5. Standardivirrat

Standardivirrat ovat tietokonejärjestelmissä olevat virrat, jotka lukevat syötettä näppäimistöltä ja kirjoittavat tulosteen näytölle. Java sisältää kolme erilaista standardivirtaa: standardisyöte (*System.in*), standardituloste (*System.out*) sekä standardivirhe (*System.err*). Nämä oliot on määritelty automaattisesti ja niitä ei ole tarpeen alustaa tai avata millään tavalla. Standardituloste ja -virhe ovat molemmat tulostevirtoja, mutta mahdollistavat tällöin samanaikaisen virheen näytölle tulostuksen ja tiedostoon kirjoittamisen, esimerkiksi virhelogeja generoidessa. Standardivirrat ovat tyypiltään tavuvirtoja, vaikka käyttökohteen vuoksi voitaisiin niiden olettaa olevan tekstivirtoja. Tämän vuoksi esimerkiksi tekstin lukeminen näppäimistön syötteenä vaatii erillisen luokan, InputStreamReaderin, kietomisen tietovirran ympärille. Enemmän asiaa tavuvirtojen ja tekstivirtojen eroista seuraavassa luvussa.

Esimerkki 2.10. Syötteen lukeminen standardivirrasta

```
public class ReaderExample {
   public static void main(String[] args) {
      try {
         BufferedReader in;
         in = new BufferedReader(new InputStreamReader(System.in));
         String inputLine;
         System.out.print("Enter input: ");

         while((inputLine = in.readLine()) != null) {
            if(inputLine.contains("Term")) {
               System.out.println(inputLine);
               break;
            }
            System.out.println(inputLine);
            System.out.print("Enter input: ");
         }
      }
      in.close();

      } catch (IOException ex) {
         System.out.println("Reader has detected an error!");
      }
   }
}
```

Tuloste

```
run:
```

```
Enter input: Is it dead?
Is it dead?
Enter input: Terminated.
Terminated.
BUILD SUCCESSFUL (total time: 12 seconds)
```

Esimerkissä huomattiin taas muutamia uusia sanoja, *try* ja *catch*. Nämä liittyvät olennaisesti virheiden käsittelyyn, jossa try-lohkossa pyritään tekemään suoritus ja catch-lohko ottaa kiinni syntyvät virheet, jotka sille on määritelty. Edellä olevassa esimerkissä on Javan kääntäjän vaatima virheenkäsittely, sillä aina kun dataa luetaan tai kirjoitetaan, vaatii kääntäjä automaattisesti sen ympärille virheenkäsittelyn. Tässä tapauksessa vaatimuksen aiheuttaa in.readLine()-metodi, joka lukee siis System.in-virrasta käyttäjän syötettä ja tuottaa virheen, jos mitään luettavaa ei löydy. Tämän virheen pakollisuus johtuu yksinkertaisesti siitä, että ohjelmoidessa ulkoisten resurssien käyttäminen on aina epävarmaa ja johtaa hyvin useasti virheisiin. Esimerkissä nähdäänkin jo lukemisen ja kirjoittamisen aikana tapahtuva virhe, eli *IOException* (*Input-Output Exception*), jossa ilmoitetaan ohjelman ajonaikaisesta virheestä, jos haettavaa syötettä ei löydy tai virheestä, jos luettavaa tiedostoa ei löydy tai kirjoitettavaan tiedostoon ei ole oikeuksia kirjoittaa.

Toinen tapa lukea käyttäjän syötettä komentoriviltä on Scanner-luokka. Scanner on siitä hyödyllinen luokka, että edellä esitetyssä esimerkissä on mahdollista lukea käyttäjän syötettä rivi kerrallaan. Joskus, varsinkin komentorivillä, on tarpeellista lukea käyttäjältä useampia syötteitä kerralla yhdeltä riviltä, jolloin kokonaisen rivin lukeminen olisi työlästä ja se pitäisi rikkoa osiin manuaalisesti. Scanner-luokka mahdollistaa käyttäjän syötteen lukemisen sana - välilyöntien tai muun määriteltävän merkin erottama merkkijono - kerrallaan tai kokonainen rivi kerrallaan. Scannerin avulla on myös mahdollista eristää syötteestä kokonaislukuja ja liukulukuja, sekä se mahdollistaa tietyn sisällön etsimisen syötteen joukosta. Scanner antaa siis huomattavasti enemmän mahdollisuuksia syötteen keräämiseen.

Esimerkki 2.11. Scanner-luokka ja käyttäjän syöte

```
public static void main(String[] args) {
    Scanner s = new Scanner(new BufferedReader(
          new InputStreamReader(System.in)));

    System.out.print("Input: ");
    System.out.println("If there's next element: " + s.hasNext());
    System.out.println("Any characters: " + s.next());
    System.out.println("Integer: " + s.nextInt());
    System.out.println("Boolean: " + s.nextBoolean());

    s.close();
}
```

Tuloste

```
run:
Input: Running.... 15 true
If there's next element: true
Any characters: Running....
Integer: 15
Boolean: true
BUILD SUCCESSFUL (total time: 9 seconds)
```

Huomaa, että esimerkki on vain main-funktio ja toimiakseen se olisi siirrettävä olemassa olevan luokan sisälle.

2.6. Kerrataan

Oppaan tässä luvussa käsiteltiin pääasiassa luokan jäsenmetodien käyttäminen, sillä niitä on huomattavasti enemmän kuin pelkästään muuttaja- ja lukijametodeita. Luokan jäsenmetodit ovat tärkeä osa olio-ohjelmointia, sillä niiden avulla on olion on mahdollista toimia niin, ettei sen toimintaa voi nähdä ulospäin. Jäsenmetodien avulla piilotetaan kaikki sellainen toiminta, mistä ulkopuolisten olioiden ei tarvitse tietää.

Viittausten käyttäminen näyttää samanlaiselta ohjelmointitasolla kuin arvon asettaminen muuttujaan, mutta on hyvä muistaa, että viittauksista puhuttaessa puhutaan tietokoneen muistissa olevista olioista, ei muuttujista. Viittaaminen olioon tarkoittaa, että viitteen avulla olion jäsenmetodeita on mahdollista käyttää myös olion ulkopuolella (jos metodin näkyvyysmääre sen sallii), eli olion tarjoamaa ulkoista rajapintaa, olion metodeja, on mahdollista käyttää.

Lisäksi tässä luvussa käytiin läpi perustietotyyppien tyyppimuunnoksia, sillä ne ovat hyvin tarpeellisia vahvasti tyypitetyissä kielissä. Lisäksi luvussa esiteltiin Javan tietorakenteista lista ja vektori sekä muutamia niiden jäsenmetodeja.

Kuten esimerkeistä on huomattu, Java vaatii hyvin paljon erilaisten kirjastojen käyttämistä toimiakseen luontevasti. Seuraavassa kappaleessa käsitellään lisää I/O-toimintoja eli tietovirtojen hallintaa, tiedostoon kirjoittamista. Luvussa käydään myös läpi vielä muutamia tietorakenteita sekä hyödyllisiä kirjastoja.

3. Kirjastot ja tietovirrat

"*In programming, a library is a collection of precompiled routines that a program can use. The routines, sometimes called modules, are stored in object format. Libraries are particularly useful for storing frequently used routines because you do not need to explicitly link them to every program that uses them.*" - Webopedia [URL: http://www.webopedia.com/TERM/L/library.html]

Kirjastoihin luetaan tavallisesti sekä standardikirjastot että kolmannen osapuolen tuottamat kirjastot. Standardikirjastot ovat kirjastoja, jotka sisältävät ohjelmointikielelle tarkoitetut funktiot ja luokat valmistajasta riippumatta. Standardikirjastot ymmärretään tavallisesti osaksi ohjelmointikieltä, vaikka ovatkin erillinen kokonaisuus ja ne sisältävät tavallisesti kielen yleisemmin käytetyt algoritmit, tietorakenteet sekä I/O-toiminnot. Esimerkki standardikirjastosta on C-kielen *stdio.h*-kirjasto, eli "*standard input output*"-kirjasto, joka määrittelee kielen I/O-toiminnot, vastaava standardikirjasto Javassa on *java.io*.

Standardikirjastojen lisäksi on olemassa erilaisia ohjelmoijan omia kirjastoja sekä kolmansien osapuolten tuottamia kirjastoja, kuten käyttöliittymäkirjastot ja karttapohjakirjastot. Nämä kirjastot tuovat ohjelmointiin lisää ominaisuuksia, jotka ovat liian suurikokoisia ja epätavanomaisia tavalliseen standardikirjastoon. Tällaisia kirjastoja ovat esimerkiksi Qt, GTK+, JavaFX, Oskari ja Google Maps.

3.1. Tietovirrat (I/O streams)

"*An I/O Stream represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.*" - Alessandro D'Ottavio
[URL: http://alessandrodottavio.wordpress.com/2012/04/05/encoding-in-java-io/]

Tietovirtojen avulla on mahdollista liikutella erilaista dataa ohjelmaan (input) ja ohjelmasta ulos (output). Niiden kanssa on mahdollista liikutella bittejä, primitiivisiä tietotyyppejä ja olioita sekä muokata ja manipuloida dataa hyödyllisin tavoin. Siitä huolimatta, mitä ja millaista tietoa tietovirta siirtää, on sen toiminta silti vain datan liikuttelua lähteestä toiseen. Syötevirran (*input stream*) avulla dataa luetaan lähteestä, oli lähde sitten käyttäjä, tiedosto tai verkko. Tulostevirtaa (*output stream*) käytetään, kun siirretään dataa varastoon tai näytille, esimerkiksi näytölle tai tiedostoon. Tietovirrat voivat kytkeä myös kaksi oliota toisiinsa, missä molemmat oliot ovat jo olemassa ja kommunikoivat keskenään datapakettien avulla, esimerkiksi verkon yli kommunikointi (*networking*). Tietovirtojen

toiminnasta enemmän Javan esimerkkien avulla kappaleessa 3.2., mutta ennen sitä muutama sana seralisoinnista.

3.1.1. Serialisointi

"*Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.*" - Tutorialspoint
[URL: http://www.tutorialspoint.com/java/java_serialization.htm]

Olioiden kirjoittaminen tietovirtoihin on tavallisesti hyvin vaikea hahmottaa, sillä olio voidaan ajatella pakettina ja tietovirta on helppo hahmottaa putkena, jota pitkin dataa siirretään. Varsinkin suurten olioiden kanssa ajatellaan, että oliota ei ole mahdollista mahduttaa tietovirtaan vaan olion sisältämä tieto pitäisi lähettää erikseen. Tämä onkin tavallisesti hyvin vaikea tehtävä vahvasti tyypitetyissä kielissä, sillä jokainen I/O-toimintoja suorittava olio odottaa parametrinaan juuri tietyn tyyppistä dataa, jolloin varsinkin ohjelmoijan omia olioita on mahdotonta saada sopimaan lähettävään olioon.

Serialisaatio on luotu, jotta olioita olisi mahdollista kirjoittaa virtoihin ja sitä kautta esimerkiksi tiedostoon. Serialisoinnin ansiosta olio on mahdollista muuttaa bittijonoksi niin, että sen sisältämä data ja rakenne pysyvät muuttumattomina ja se voidaan lähettää eteenpäin tulostevirtaan tai lukea syötevirrasta. Tämä mahdollistaa olion tallentamisen tietokoneen muistiin ilman, että olion tietoja olisi tarpeen tallentaa erikseen tiedostoon. Serialisoinnin avulla oliot voidaan siis tallentaa ja ladata tiedostosta ilman tuskastuttavaa olioiden uudelleen luontia datan pohjalta.

3.2. Javaa

Oppaan tässä luvussa käsiteltiin enemmän kirjastoja sekä tietovirtoja. Luvussa käytiin suhteellisen pieni määrä teoriaa, sillä kyseessä olevat aihealueet ovat itsessään hyvin yksinkertaisia, mutta niiden järkevä käyttäminen vaatii harjoittelua. Javan kirjastoista löytyy enemmän tietoa javadocista (http://docs.oracle.com/javase/7/docs/api/overview-summary.html) ja seuraavassa mainitaan muutamia yleisiä:

Taulukko 3. Javan kirjastoja

Kirjasto	Kuvaus	Esimerkki
java.net	Sisältää luokkia, joiden avulla voidaan toteuttaa kommunikaatiota verkon yli	java.net.URL luo merkkijonosta urlin ohjelman käyttöön ja voi myös avata yhteyden ja ylläpitää yhteyttä
java.io	Mahdollistaa Javan I/O-tietovirrat	java.io.FileInputStream mahdollistaa syötteen lukemisen tiedostosta
java.lang	Sisältää fundamentaalisia Javan toiminnan takaavia luokkia	java.lang.String sisältää String-luokan määrittelyn
java.util	Pitää sisällään mm. tietorakenteisiin tarvittavat luokat	java.util.Vector sisältää vektorin määrittelyn

3.2.1. Tietovirrat Javalla

Ennen tietovirtojen varsinaista käsittelyä on huomattava, että aina kun tietovirtoja käytetään kumpaan suuntaan tahansa, on ne käytön jälkeen suljettava. Tämän avulla vältetään suuri osa erilaisista resurssivuodoista. Javan tietovirrat voidaan jakaa kahteen luokkaan, tavuvirtoihin (*byte streams*) ja tekstivirtoihin (*character streams*).

Tavuvirrat ovat Javassa matalimman tason tietovirtoja, joiden käsittelyyn on olemassa InputStream- ja OutputStream-luokat *java.io*-kirjastossa. Tavuvirtoja käytetään siirtämään 8-bitin paketteja, eli tavuvirta ottaa lukiessaan syötteeksi aina yhden tavun ja kirjoittaa ulos yhden tavun. Kun tavuja luetaan tiedostosta, tavuvirta käyttää FileInputStream-luokkaa, joka ottaa rakentajan parametriksi joko tiedoston nimen tai File-olion. Tiedostoon kirjoitettaessa käytetään FileOutputStream-luokkaa, joka on rakenteeltaan samanlainen kuin syötteen vastaava luokka.

Seuraavat esimerkit sisältävät vain main-funktion, eivätkä ole sidottuja luokkien sisään. Jotta ohjelmoija saa esimerkin ajettua, tulee main-funktio sijoittaa luokan sisään.

Esimerkki 3.1. Tavuvirrat tiedostoon ja tiedostosta

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class ReadAndWrite {
   public static void main(String[] args) {
      try {
```

```
            FileInputStream in;
            FileOutputStream out;

            in = new FileInputStream("example.txt");
            out = new FileOutputStream("outputEx.txt");
            int c;

            while((c = in.read()) != -1) {
               out.write(c);
            }
            in.close();
            out.close();
         } catch (IOException ex) {
            System.out.println("File not found!");
         }
      }
   }
```

Toinen tietovirtatyypeistä Javassa on tekstivirta, joka nimensä mukaisesti käsittelee vain tekstiä. Tavuvirrat voivat käsitellä kaikkea mahdollista dataa, mutta tekstivirrat käsittelevät ainoastaan merkkejä. Esimerkki 3.1. pätee suoraan sellaisenaan myös tekstivirroille sillä erolla, että FileInputStream muuttuu FileReaderiksi ja FileOutputStream muuttuu FileWriteriksi.

Tavu- ja tekstivirrat ovat kuitenkin käytännössä hyvin hitaita, koska ne hakevat tietoa lähteestä vain yksi tavu tai merkki kerrallaan. Tämän vuoksi Javassa on mahdollista luoda puskuroituja tietovirtoja, joissa on mahdollista käsitellä suurempaa tietomäärää kerrallaan. Puskuroidun tietovirran tarkoitus on, että data luetaan tai kirjoitetaan muistissa olevaan puskuriin, jolloin kaikkia merkkejä ja tavuja ei tarvitse lukea muistista tai levyltä. Puskuroituja tietovirtoja varten Javassa on luokat BufferedReader/BufferedWriter tekstivirroille ja BufferedOutputStream/ BufferedInputStream tavuvirroille. Puskuroitu tietovirta voidaan luoda antamalla sille rakentajan parametriksi tavallinen tietovirta.

Esimerkki 3.2. Puskuroidut tietovirrat

```
public class ReadAndWrite {
   public static void main(String[] args) {
      try {
         BufferedReader in;
         BufferedWriter out;

         in = new BufferedReader(new FileReader("example.txt"));
         out = new BufferedWriter(new FileWriter("outputEx.txt"));
         String inputLine;
         while((inputLine = in.readLine()) != null) {
            out.write(inputLine);
         }
         in.close();
         out.close();
```

```
            } catch (IOException ex) {
               System.out.println("File not found!");
            }
         }
      }
```

Kuten esimerkistä huomataan, puskuroitu tietovirta mahdollistaa lukemisen rivi kerrallaan.

Edellä mainittujen tietovirtojen lisäksi Javassa on mahdollista luoda tietovirta, joka käsittelee primitiivisiä tietotyyppejä eli datavirta (*data stream*) tai tietovirta, joka käsittelee olioita eli oliovirta (*object stream*). Näitä tietovirtavariaatioita edustavat luokat DataInputStream/DataOutputStream sekä ObjectInputStream/ObjectOutputStream. Oliovirtoja käytettäessä on huomattava, että liikuteltavan olion on oltava serialisoituva, eli se on muutettava tavumuotoon siirron ajaksi.

3.2.2. Tietorakenteet: Enumeration ja map

"*The Enumeration interface isn't itself a data structure, but it is very important within the context of other data structures. The Enumeration interface defines a means to retrieve successive elements from a data structure.*" - tutorialspoint
[URL: http://www.tutorialspoint.com/java/java_data_structures.htm]

Enumeration ei ole puhdas tietorakenne, mutta se nivoutuu yhteen tietorakenteiden kanssa, sillä sen avulla on mahdollista iteroida läpi erilaisia tietorakenteita ja käsitellä niiden sisältämiä arvoja yksi kerrallaan, muuttamatta itse rakenteen sisältöä. Enumeration ei sisällä kuin kaksi jäsenmetodia, joiden avulla on mahdollista käydä läpi rakenteen sisältö, nextElement() ja hasMoreElements(). Kuvataan Enumerationin käyttöä esimerkin avulla.

Esimerkki 3.3. Enumeration ja vektori

```
public class EnumEx {
   public static void main(String[] args) {
      Enumeration weekdays;
      Vector week = new Vector();
      week.add("Sunday");
      week.add("Monday");
      week.add("Tuesday");
      week.add("Wednesday");
      week.add("Thursday");
      week.add("Friday");
      week.add("Saturday");
      weekdays = week.elements();
      while(weekdays.hasMoreElements()) {
         System.out.println(weekdays.nextElement());
      }
   }
```

```
}
```

Tuloste

```
run:
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
BUILD SUCCESSFUL (total time: 1 second)
```

Enumeration on jo hieman vanhanaikainen iterointimenetelmä, mutta se on edelleen käytössä monissa erilaisissa luokissa, joten sen olemassaolo on hyvä tietää. Enumerationia ei myöskään pidä sekoittaa avainsanaan *enum*, josta lisää myöhemmin luvussa 4.

Tietorakenne map muistuttaa hyvin paljon Pythonin sanakirjaa, mihin on mahdollista tallentaa avain/arvo-pareja. Tällöin on siis mahdollista linkittää yksikäsitteisiä avain-arvoja arvokenttiin ja näin ollen päästä kiinni tallennettuun arvoon avaimen avulla.

Esimerkki 3.4. Map ja sen avain/arvo-parit

```
public class MapSample {

   public static int askCageNmb() {
      Scanner s = new Scanner(System.in);
      System.out.print("Enter cage number: ");
      int cage = s.nextInt();

      s.close();
      return cage;
   }

   public static void main(String[] args) {
      System.out.println("*** ZOO! ***");
      Map m = new HashMap();
      m.put(12, "Giraffe");
      m.put(16, "Elephant");
      m.put(5, "Monkey");
      m.put(65, "Rhino");
      m.put(44, "Zebra");
      System.out.print("Animals and cages: ");
      System.out.println(m);
      int cage = askCageNmb();

      System.out.println("In cage " + cage + " lives " + m.get(cage));

   }
}
```

Tuloste

```
run:
*** ZOO! ***
Animals and cages: {16=Elephant, 65=Rhino, 5=Monkey, 12=Giraffe, 44=Zebra}
Enter cage number: 12
In cage 12 lives Giraffe
BUILD SUCCESSFUL (total time: 5 seconds)
```

4. Oliopohjainen suunnittelu ja UML

"*It's a process of planning a software system where objects will interact with each other to solve specific problems. The saying goes, 'Proper Object oriented design makes a developer's life easy, whereas bad design makes it a disaster.*" - Maria Sukesh
[URL: http://www.codeproject.com/Articles/567768/Object-Oriented-Design-Principles]

Olioparadigma aiheutti muutoksia ohjelman luontitavassa eli ongelman ratkaisumallissa ja sitä kautta synnytti olio-ohjelmoinnin. Kuitenkin, jotta olioiden ohjelmoiminen järkevässä mittakaavassa olisi mahdollista, oli myös ohjelmistojen suunnittelun muututtava noudattamaan uutta ohjelmointitapaa. Olioparadigmasta kehittynyt oliopohjainen suunnittelu ei kuitenkaan nimestään huolimatta ole suunnittelutapa tai sen avulla tehty analyysi ei anna mitenkään eriävää ongelmakuvausta kuin muutkaan eri paradigmoihin perustuvat suunnittelumenetelmät. Oliopohjainen suunnittelu eroaa muista suunnittelumenetelmistä sillä, että se tuottaa lopputuloksenaan ratkaisumallin, joka nojaa toimintansa olioiden toimintaan ja niiden välisiin yhteyksiin. Aikaisemmissa luvuissa on nähty jo esimerkkejä siitä, miten tällaista oliosuunnitelmaa voidaan kuvata käyttämällä UML-suunnittelukielen mukaisia rakenteita.

Oliopohjaisessa suunnittelussa on olemassa suunnittelufilosofia SOLID, joka sisältää viisi periaatetta. Periaatteiden avulla olio-ohjelmasta todennäköisesti muodostuu järkevä kokonaisuus, jota voidaan muokata ja laajentaa ajan kuluessa. SOLID tarjoaa työkalun lähdekoodin siivoamiseen, kunnes siitä saadaan sekä luettavaa että laajennettavaa ja se toimii osana ketterää ohjelmistokehitystä.

Taulukko 4. SOLIDin periaatteet
[URL: http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)]

Etukirjain	Lyhenne	Tarkoitus	Kuvaus
S	SRP	Single Responsibility Principle	Jokaisella luokalla tulisi olla vain yksi vastuualue
O	OCP	Open/Closed Principle	Luokka on avoin laajennukselle, mutta suljettu muutoksille
L	LSP	Liskov Substitution Principle	Kantaluokan olio voidaan aina korvata lapsiluokan oliolla ilman muutoksia ohjelmakoodiin
I	ISP	Interface Segregation Principle	Useat räätälöidyt rajapinnat ovat parempi ratkaisu kuin yksi yleinen rajapinta
D	DIP	Dependency Inversion Principle	Luota abstraktioon, älä kiinteään rakenteeseen

4.1. Pariutuminen (coupling) ja koheesio (cohesion)

Ohjelmistoja kehitettäessä pariutumisen tarkoitus on mitata sitä, kuinka läheisesti ohjelmiston komponentit ovat riippuvaisia toisistaan. Kuten aikaisemmassa luvussa mainittiin, olioparadigma rakentuu luokista, joten pariutumisella mitataan tässä tapauksessa sitä, miten erillään luokat toimivat toisistaan eli kuinka modulaarisia ne ovat. Pariutumista mitataan tavallisesti asteikolla matalasta korkeaan, mutta myös löysästä tai heikosta tiukkaan tai vahvaan. Hyvään ohjelmiston suunnitteluun kuuluu kiinteänä osana matalaan pariutumiseen pyrkiminen, sillä se lisää luokkien ja olioiden uusiokäyttöä ja mahdollistaa ohjelmiston laajentamisen ilman suuria rakenteellisia muutoksia. Matala pariutuminen liitetään tavallisesti korkeaan koheesioon, jolloin luotua ohjelmaa on helppo ylläpitää ja sen lähdekoodi on helppolukuista.

Koheesion avulla mitataan ohjelmoinnissa sitä, kuinka paljon ohjelman rakenneosat keskittyvät yhden toiminnallisuuden toteuttamiseen. Olioparadigmassa tämä mitataan sillä, kuinka paljon yhden luokan toiminnallisuudet liittyvät luokalle tarkoitettuun tehtävään. Luokan tarjoamat metodit tulee siis olla rakennettu niin, että ne liittyvät vain ja ainoastaan luokan toimintaan ja sen tarjoamaan rajapintaan. Korkean koheesion avulla on mahdollista muuttaa luokan ja olion toimintaa ilman, että se vaikuttaa muiden olioiden toimintoihin. Sen avulla on myös mahdollista käyttää samaa oliota useiden eri tehtävien osatoteutuksessa, eli se on hyvin uudelleenkäytettävä.

4.2. UML

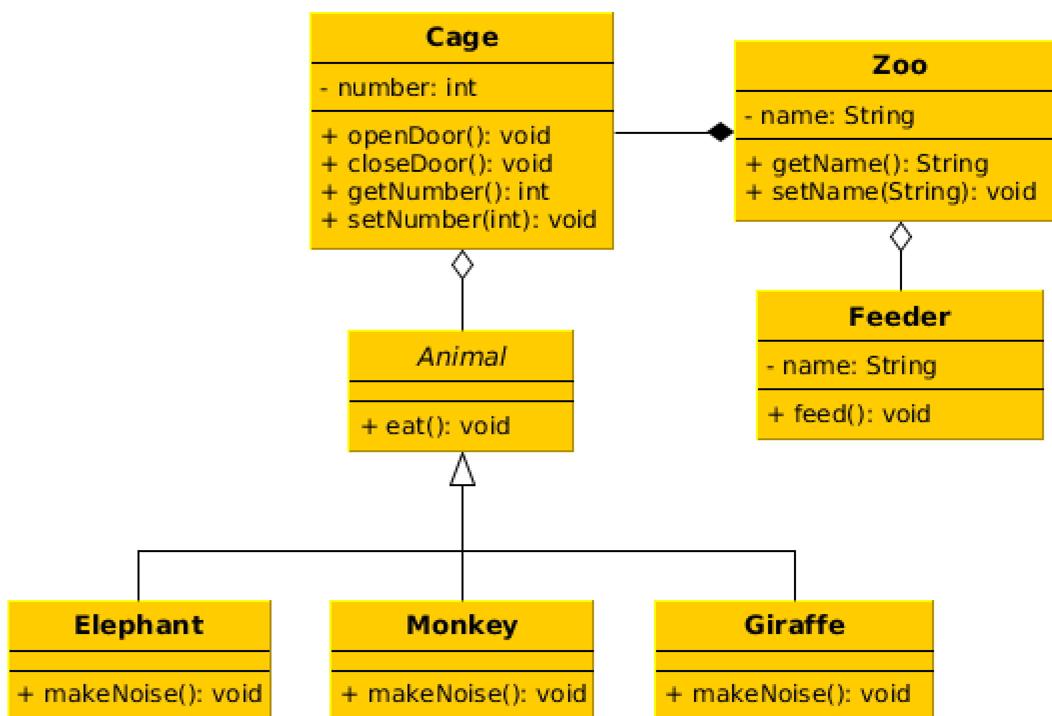
"*The Unified Modeling Language™ - UML - is OMG's most-used specification, and the way the world models not only application structure, behavior, and architecture, but also business process and data structure.*" - uml.org [URL: http://uml.org/]

Kuten edellisessä kappaleessa sanottiin, UML on suunnittelukieli, jolla on sekä syntaksi että semantiikka. UML:n avulla ongelmia kuvatessa on hyvä siis muistaa, että se sisältää ohjeita ja sääntöjä siihen, mitä ja miten asioita voidaan kuvata. UML ei kuitenkaan ole vain kieli, jonka avulla esitetään osien erilaisia konsepteja, vaan se mahdollistaa myös niiden kontekstien esittämisen. UML kielenä rakentuu pääasiassa erilaisista kuvaajista (*diagram*), joiden avulla voidaan kuvata ohjelmia hyvin yksityiskohtaisesta erittäin geneeriseen kuvaukseen. Tämän oppaan tarkoituksena on tutustuttaa lukija vain luokkakaavion ominaisuuksiin ja toimintaan, sillä UML on kokonaisuutena liian laaja käytäväksi tällä kurssilla. Enemmän UML:n toiminnasta on mahdollista etsiä sekä verkosta että lähdekirjallisuudesta, kuten UML 2.0 In a Nutshell [URL: http://www.it-ebooks.info/book/154/] tai muu vastaava kirja.

4.2.1. Luokkakaavio (class diagram)

"*In software engineering, a class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.*" - Wikipedia
[URL: http://en.wikipedia.org/wiki/Class_diagram]

Luokkadiagrammin avulla voidaan esittää järjestelmä erittelemällä se luokkiin, jäsenmuuttujiin ja -metodeihin sekä niiden välisiin kiinteisiin yhteyksiin. Luokkakaaviot ovat UML:n yksi eniten käytetyistä kaavioista ja niiden avulla on mahdollista luoda kehittäjille yhteinen kuva järjestelmän koostumuksesta. Toisaalta luokkakaaviolla voidaan esittää järjestelmä niin tarkasti, että ohjelmakoodia on mahdollista generoida suoraan kaaviosta.



Kuva 4.1. Luokkakaavio eläintarhasta

Lähdetään liikkeelle luokasta `Zoo`, jolla on jäsenmuuttuja `name` ja sen näkyvyysmääre on yksityinen. Luokkakaaviossa yksityinen näkyvyysmääre merkitään merkillä "-" ja julkinen näkyvyysmääre merkillä "+", joka asetetaan ennen jäsenmuuttujaa tai -metodia. Lisäksi luokan jäsenmuuttuja `name` on tietotyyppiä `String`, joka on ilmoitettu muuttujan nimen perään kaksoispisteellä erotettuna. Luokan jäsenmetodit ilmoitetaan samankaltaisesti kuin

jäsenmuuttujakin, mutta kaksoispisteen avulla erotetaan metodin paluuarvo ja sulkujen sisälle ilmoitetaan parametrin tietotyyppi.

Luokkien Zoo ja Cage välillä nähdään viiva, jonka päässä on musta salmiakki. Tämä tarkoittaa kompositiota (*composition*), eli vahvaa riippuvuutta toisesta luokasta. Komposition avulla voidaan ilmaista, että toinen luokista kuuluu toiselle luokalle, eli on osa sitä. Tässä esimerkin tapauksessa huomataan, että luokka Cage on osa Zoo-luokkaa. Onhan kuitenkin ihan järkeenkäypää, että eläintarha muodostuu häkeistä, samalla tavoin kuin ihminen muodostuu ruumiinosista. Kompositiolla toisiinsa liitetyt luokat pysyvät tavallisesti yhdessä koko elinkaaren ajan, eli jos eläintarha tuhotaan, todennäköisesti myös häkit tuhotaan ja sama pitää paikkaansa myös ihmisen kanssa. Kompositio ymmärretään tavallisesti kuvaamalla, että jokin luokka "muodostuu" toisesta luokasta ja sitä heikompi yhteystyyppi on aggregaatio (*aggregation*).

Aggregaatio on kuvattu esimerkissä valkoisella eli tyhjällä salmiakilla. Vaikka aggregaatio esittää heikompaa yhteyttä kuin kompositio, esittää se silti omistajuussuhdetta toiseen luokkaan. Esimerkissä luokka "Feeder" on yhteydessä eläintarhaan aggregaatiolla, eli ruokkija on osa eläintarhaa, mutta ei kuitenkaan osaluokka. Tästä näemme, että ruokkijaa ei tuhota silloin, kun eläintarha tuhotaan, mikä on tietenkin täysin loogista. Sama pätee yhteyteen luokkien Animal ja Cage välillä, missä eläin kuuluu tiettyyn häkkiin (voidaan tietysti kiistellä siitä, kuuluuko eläin häkkiin), mutta eläintä ei tuhota häkin tuhoamisen yhteydessä. Aggregaatio voidaan siis ymmärtää niin, että häkki "omistaa" eläimen samalla tavalla kuin yhtiö tai eläintarha "omistaa" työntekijöitä.

Esimerkin kuvassa oleva Animal-luokka näyttää hieman erilaiselta kuin muut luokat. Tämä johtuu siitä, että Animal-luokka on abstrakti luokka, eli siitä ei voida luoda olioita vaan se toimii tässä yhteydessä vain kantaluokkana, josta eläinoliot periytetään. Periytymisen piirrosmerkkinä käytetään valkoista kolmiota, joka osoittaa aliluokasta kantaluokkaan. Rajapintaa taas esittää täysin samanlainen piirrosmerkki, mutta rajapinnassa on katkoviiva, kun periytymisessä on kiinteä viiva.

Kantaluokasta Animal eriytetään kolme luokkaa, Giraffe, Elephant ja Monkey, jotka saavat kaikki Animal-luokan määrittelemät jäsenmetodit omaan käyttöönsä. Eli ne perivät toimintonsa kantaluokasta sekä määrittelevät omia toimintojaan ja voivat myös uudelleenmääritellä (*override*) kantaluokan toimintoja. Periytymisestä, abstraktista luokasta ja rajapinnoista enemmän seuraavassa luvussa.

4.3. Javasta

Oppaan tässä luvussa käytiin teoriapuolella asioita, jotka liittyvät kiinteästi olioparadigmaa noudattavan ohjelman suunnitteluun. Koodipuolella taas käsitellään avainsanat *this* ja *enum* sekä Javan oma näkyvyysmääre *package*.

4.3.1. this

this-avainsana ei liity tarkemmin ottaen Javalla ohjelmointiin eikä edes olio-ohjelmointiin, sillä se on universaali avainsana useissa ohjelmointikielessä. Tässä oppaassa this esitellään Javan avulla, mutta lukijan on hyvä muistaa, että se on käytännöllinen avainsana kaikessa ohjelmoinnissa, tosin esimerkiksi Pythonissa tämä avainsana on *self*.

Ajatellaan hypoteettinen skenaario, jossa ollaan luokan metodin sisällä ja ohjelman toiminnallisuuden vuoksi olisi tarpeellista viitata senhetkiseen olioon itseensä. Tässä kohtaa on huomattava, että kääntäjä tekee hieman salaista työtä ilman ohjelmoijan lupaa, sillä metodille on kääntäjän toimesta lähetetty tieto siitä oliosta, jonka avulla metodikutsu aktivoitiin. Huomaa kuitenkin, että jos haluat käyttää metodikutsua metodin sisältä, on se mahdollista tehdä ilman this-avainsanaa. Avainsana on tarpeellinen silloin, kun halutaan eksplisiittisesti viitata senhetkiseen olioon, esimerkiksi *return*-lausekkeessa. Palauttamalla this-avainsanan metodista, palautuu sieltä viittaus senhetkiseen olioon. Selvennetään asiaa esimerkin avulla.

Esimerkki 4.1. this-avainsanan käyttö olion palauttamiseen

```
class Person {
   public void eat(Apple apple) {
      Apple peeled = apple.getPeeled();
      System.out.println("Yummy!");
   }
}

class Peeler {
   public static Apple peel(Apple apple) {
      System.out.println("Peeling...");
      return apple;
   }
}

class Apple {
   public Apple getPeeled() {
      return Peeler.peel(this);
   }
}

public class PassingThis {
```

```
    public static void main(String[] args) {
        new Person().eat(new Apple());
    }
}
```

Tuloste

```
run:
Peeling...
Yummy!
BUILD SUCCESSFUL (total time: 0 seconds)
```

Esimerkissä nähdään, että käytössä on kolme luokkaa: Person, Peeler ja Apple. Ohjelman tehtävä on, että Person haluaa syödä omenan, joka on kuorittava ennen syömistä. Suoritusjärjestys menee aluksi Person-olion eat-jäsenmetodiin, jossa omena kuoritaan. Apple-olio taas pyytää Peeler-oliota kuorimaan senhetkisen omenan eli itsensä, jolloin omena on lähetettävä kuorimelle this-avainsanaa käyttäen. Muuten tämä lähetys eri luokkien välillä ei olisi mahdollista. Peeler kuorii omenan, palauttaa sen Apple-oliolle, joka palauttaa sen Person-oliolle, joka lopulta syö omenan. Toinen mahdollinen käyttötapa avainsanalle this on rakentajien kutsuminen rakentajien sisältä, kuten seuraavassa esimerkissä.

Esimerkki 4.2. this rakentajien käytössä

```
public class Flower {

    private int petal_count;
    private String s;

    public Flower(int count) {
        petal_count = count;
        System.out.println("Constructor with int parameter.");
    }

    public Flower(String ss) {
        s = ss;
        System.out.println("Constructor with String parameter.");
    }

    public Flower(String s, int count) {
        this(count);
        this.s = s;
        System.out.println("Constructor with int and String parameters.");
    }

    public Flower() {
        this("Hi!", 47);
        System.out.println("Default constructor.");
    }

    private void printPetalCount() {
        System.out.println("Petalcount = " + petal_count + " s = " + s);
```

```
    }

    public static void main(String[] args) {
        Flower flower = new Flower();
        flower.printPetalCount();
    }
}
```

Tuloste

```
run:
Constructor with int parameter.
Constructor with int and String parameters.
Default constructor.
Petalcount = 47 s = Hi!
BUILD SUCCESSFUL (total time: 1 second)
```

Esimerkissä Flower-luokka sisältää neljä erilaista rakentajaa, joista yksi on oletusrakentaja ja kolme muuta parametrillisia. Rakentajat eivät ole kovin monikäyttöisiä, mutta tämän esimerkin tarpeisiin ne ovat riittäviä. Ohjelman käynnistyessä kutsutaan oletusrakentajaa, jonka sisällä on suoritus, missä kutsutaan this-avainsanan avulla kahden parametrin omaavaa rakentajaa. Tässä huomataan, että suoritus jää odottamaan, että kutsuttu rakentaja valmistuu ennen kuin alkuperäinen rakentaja valmistuu.

this-avainsanan avulla siis kutsutaan uudelleen luokan rakentajaa (eri rakentaja kuin aluksi, muuten olisi kyseessä päättymätön rekursio) ja lähetetään sille sekä merkkijono että luku. Tämä rakentaja vuorostaan kutsuu kolmatta rakentajaa this-avainsanan avulla, jolle se lähettää pelkästään kokonaisluvun. Tämän peräänhän olisi loogisinta kutsua seuraavaa rakentajaa, jolle syötetään merkkijono. Kaiken tämän ketjuttamisen kanssa on kuitenkin huomattava, että kahta rakentajaa ei ole mahdollista kutsua peräkkäin, mutta ns. sisäkkäin se on kuitenkin mahdollista, eli yhtä rakentajaa voidaan kutsua toisessa rakentajassa, kun kyseessä on sama olio. Tämän vuoksi kahden parametrin rakentajassa on ensin kutsuttu rakentajaa kokonaisluvun kanssa ja sen jälkeen asetettu merkkijono this-avainsanan avulla merkkijonomuuttujaan.

Tulostuksesta voidaan huomata, missä järjestyksessä rakentajat suoritetaan, sillä ensimmäisenä kutsuttu rakentaja suoritetaan loppuun vasta kolmantena. Rakentajat siis suoritetaan loppuun sitten vasta, kun edellinen on tehtävänsä suorittanut. Lopuksi huomataan, että ketjutetuilla rakentajilla syntyy samanlainen olio kuin syntyisi vain yhdellä rakentajalla.

4.3.2. Package ja nimiavaruudet

"*A package is a grouping of related types providing access protection and name space management. Note that types refers to classes, interfaces, enumerations, and annotation types.*" - Oracle
[URL: http://docs.oracle.com/javase/tutorial/java/package/packages.html]

`package`-avainsana eli paketti on tullut varmasti jo tähän mennessä Javaa ohjelmoidessa tutuksi, sillä jokainen Javalla tehty luokka vaatii, että se sijaitsee jossakin paketissa. Mitäs tämä sitten tarkoittaa? Paketti liittyy hyvin läheisesti kirjastojen käyttämiseen ja *import*-avainsanaan, sillä ne ovat käytännössä Javan kirjastoja. Paketti siis nimensä mukaisesti pitää sisällään jotakin ja tässä tapauksessa paketti pitää sisällään luokkia, jotka on järjestetty saman nimiavaruuden (*namespace*) alle. Nimiavaruuksien avulla on mahdollista sitoa useita symboleita ja tunnuksia ns. saman katon alle ja tällä tavoin sallia kahden samannimisen instanssin (esimerkiksi luokan) olemassaolon. Avainsana *namespace* löytyy myös muista ohjelmointikielistä kuten C++ ja PHP, mutta niissä avainsanaa käytetään hieman eri tavalla kuin Javassa. Nimiavaruuksia on mahdollista luoda myös itse (mutta ei ole tämän kurssin asiaa) ja niitä saadaan käyttöön avainsanan `import` avulla, josta enemmän seuraavassa.

Aikaisemman luvun esimerkkiä käyttäen otamme käsittelyyn luokan ArrayList. Kun otamme listan käyttöön ohjelmassa, ilman *import*-sanaa uuden listan luominen joudutaan tekemään julistamalla koko pakkausketju:

```
java.util.ArrayList list = new java.util.ArrayList();
```

Toisaalta vaihtoehtona on tuoda pakkausketju nimiavaruudeksi ohjelmaan *import*in avulla ja julistaa:

```
import java.util.*;
ArrayList list = new Arraylist();
```

Paketteihin sisällytetään toisiinsa liittyviä luokkia, jotta niiden käyttö olisi yksinkertaisempaa. Pääsyy pakettien käyttöön on nimiavaruuksien hallinta, joiden avulla kaksi luokkaa voi sisältää samannimisen jäsenmetodin tai kahdella luokalla voi olla sama nimi, niin kauan kuin ne ovat eri nimiavaruuksien sisällä.

Tämän lisäki Java tarjoaa paketeille uuden näkyvyysmääreen, joka voidaan asettaa luokan jäsenmuuttujille. Tällöin kaikki luokat, jotka ovat saman paketin sisällä, voivat käyttää ko. jäsenmuuttujaa vapaasti eli näkyvyysmääre on *public*, mutta kaikki paketin ulkopuoliset luokat näkevät muuttujan määreellä *private*. Tällainen käyttäytyminen madaltaa olioiden pariutumista,

mutta on suotavaa, että luokka tarjoaa rajapinnassaan muuttaja- ja lukijametodit, jotta arvoa ei tarvitse muokata suoraan. On muistettava, että toisen olion tilan/arvojen muuttaminen ilman omistavan olion lupaa tulisi olla kiellettyä kaikissa tapauksissa.

4.3.3. Enum

Edellisessä kappaleessa käsiteltiin tietorakennetta Enumeration, mutta *Enum* ei nimestään huolimatta liity siihen. enum on yleisesti ohjelmoinnissa käytössä oleva rakenne, jonka avulla on mahdollista tallentaa järjestelmään kiinteitä arvoja. Esitetään enum tarkemmin esimerkin avulla.

Esimerkki 4.3. enumin käyttö

```
package pkgenum;

enum Day {
   SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
   THURSDAY, FRIDAY, SATURDAY
}

public class EnumEx {

   Day day;

   public Enum(Day day) {
      this.day = day;
   }

   public void printDay() {
      switch(day) {
         case MONDAY:
            System.out.println("Whee, it's monday!");
            break;
         case FRIDAY:
            System.out.println("It's friday! Bottoms up!");
            break;
         case SATURDAY: case SUNDAY:
            System.out.println("Finally, weekend.");
            break;
         default:
            System.out.println("There are only mondays and fridays.");
            break;
      }
   }

   public static void main(String[] args) {
      Enum first = new EnumEx(Day.MONDAY);
      first.printDay();
      Enum third = new EnumEx(Day.WEDNESDAY);
      third.printDay();
      Enum fifth = new EnumEx(Day.FRIDAY);
      fifth.printDay();
```

```
        Enum sixth = new EnumEx(Day.SATURDAY);
        sixth.printDay();
    }
}
```

Tuloste

```
run:
Whee, it's monday!
There are only mondays and fridays.
It's friday! Bottoms up!
Finally, weekend.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Esimerkissä huomataan, että enum määritellään samalla tavalla kuin mikä tahansa luokka, mutta sen sisään asetetaan vain ne arvot, jotka halutaan asettaa kiinteiksi arvoiksi. Tällöin on mahdollista luoda uusia muuttujia enumin avulla ja niiden sisältöä voidaan vertailla switch-case-rakenteen avulla. enum on näppärä keino tallentaa järjestelmään yksinkertaisia vakioita niin, että koodi pysyy helppolukuisena.

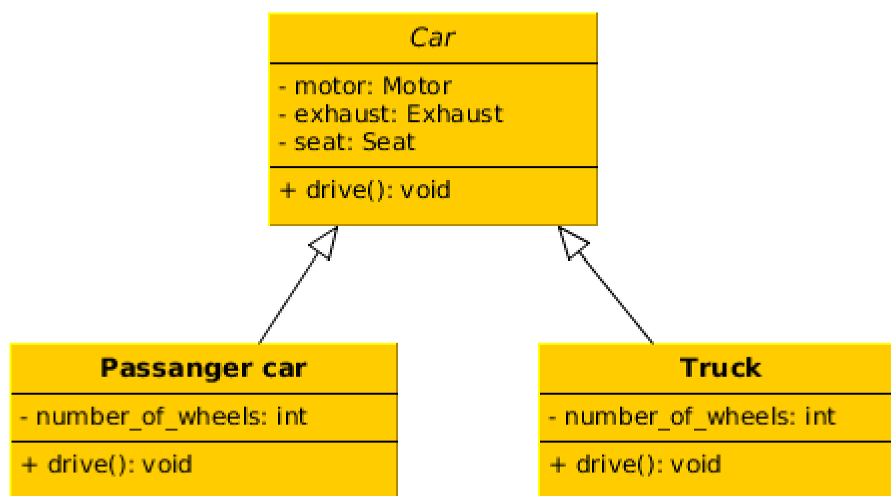
Javassa enum on huomattavasti tehokkaampi kuin vastaava avainsana muissa kielissä, sillä Javassa enum muodostaa itsestään olion, jolloin se voi sisältää jäsenmetodeja ja -muuttujia. Näiden ansiosta on enumin avulla mahdollista luoda myös täysin itsenäisiä olioita ja toiminnallisuuksia.

5. Periytyminen ja abstraktio

“*In object-oriented programming (OOP), inheritance is when an object or class is based on another object or class, using the same implementation (inheriting from a class) or specifying implementation to maintain the same behavior (realizing an interface; inheriting behavior). It is a mechanism for code reuse and to allow independent extensions of the original software via public classes and interfaces.*” - Wikipedia
[URL: http://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)]

Luokat ja oliot ovat näppärä työkalu ratkaisemaan erilaisia ohjelmallisesti ratkaistavia ongelmia. On kuitenkin turhaa luoda ensin luokka, jolla ratkaistaan yksi osaongelma ja toinen luokka, jolla on käytännössä samanlainen toiminnallisuus, mutta luokan perusajatus on kuitenkin hieman erilainen. Voidaan esimerkkinä ajatella henkilöautoa ja kuorma-autoa, jotka ovat periaatteessa samanlaisia, eli ne sisältävät esimerkiksi moottorin, renkaat ja ohjaamon. Kuitenkin niiden renkaiden määrä ja auton paino sekä dimensiot ovat huomattavasti erilaiset, eikä niitä saa ajaa samalla kortilla, joten niitä ei voida luoda samasta mallista tai samalla tuotantolinjalla. Olisi siis kätevää luoda muotti, joka kerää molempien ajoneuvojen yhteiset ominaisuudet ja jatkaa molempien kehittämistä erillisissä linjoilla.

Kuvassa 5.1. nähdään siis, että henkilöauto ja kuorma-auto on periytetty *Car*-luokasta. Tällöin *Car*-luokka pitää sisällään molempien autotyyppien perusominaisuudet ja perivät luokat saavat nyt kantaluokan ominaisuudet ja voivat lisätä omia ominaisuuksiaan. Kuten edellisessä kappaleessa mainittiin, kursivoitu luokan nimi tarkoittaa sitä, että luokka on abstrakti, jolloin siitä ei ole mahdollista luoda instansseja.



Kuva 5.1. Erilaisten autoluokkien periyttäminen

5.1. Abstrakti luokka

"*An abstract class is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.*" - Oracle
[URL: http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html]

Kuvasta 5.1. esitettiin, miten on kaksi käytännössä hyvin lähekkäistä luokkaa voidaan periyttää samasta kantaluokasta. Kantaluokka on esimerkissä abstrakti, joka määrittää auton perustoiminnallisuudet ja jakaa ne eteenpäin periville luokille. Kantaluokan ei ole pakko olla abstrakti, mutta esimerkin kannalta se on suotavaa. Miksi luokka on abstrakti? Ajatellaan autotehdasta, jossa halutaan tuottaa autoja. Onko mahdollista tuottaa liukuhihnalta ulos auto, jolla on vain auton perusominaisuudet, mutta esimerkiksi korimalli ja renkaiden lukumäärä ei olisi varmasti tiedossa? Vastaus kysymykseen on ei. Joten selkeästi abstraktion tarkoituksena on mahdollistaa yhteinen kanta usealle samankaltaiselle luokalle, mutta samaan aikaa estää instanssien luominen ilman kokonaista toteutusta.

Luokan lisäksi on mahdollista luoda abstrakteja metodeita, mutta abstraktin metodin sisältävä luokka on tällöin myös abstrakti. Abstraktilla metodilla tarkoitetaan metodia, jolle ei ole määritelty toteutusta, mutta sen parametrit ja paluuarvo on määritelty. Käytämme esimerkkinä edellä esitettyä kaaviota autoista.

Esimerkki 5.1. Abstraktio ja periytyminen koodin tasolla

```
abstract class Car {
    protected String manufacturer;
    protected String model;
    protected int year;

    protected void printData() {
        System.out.println(manufacturer + " " + model + " " + year);
    }
    protected void turnLeft() {
        System.out.println(this.getClass().getSimpleName() +
                ": Turning left...");
    }
    protected void turnRight() {
       System.out.println(this.getClass().getSimpleName() +
                ": Turning right...");
    }
    protected abstract void drive();
}

class PassangerCar extends Car {
    public PassangerCar() {
        manufacturer = "Honda";
        model = "Civic";
```

```
        year = 2000;
    }
    @Override
    protected void drive() {
        System.out.println(this.getClass().getSimpleName() + ": Vrooooom!");
    }
}

class Truck extends Car {
    public Truck() {
        manufacturer = "Volvo";
        model = "VN";
        year = 2005;
    }
    @Override
    protected void drive() {
        System.out.println(this.getClass().getSimpleName() + ": BRUM BRUM!!");
    }
}

public class Cars {

    public static void main(String[] args) {
        PassangerCar car = new PassangerCar();
        Truck truck = new Truck();
        car.printData();
        car.drive();
        truck.drive();
        car.turnLeft();
        truck.turnRight();
    }
}
```

Tuloste

```
run:
Honda Civic 2000
PassangerCar: Vrooooom!
Truck: BRUM BRUM!!
PassangerCar: Turning left...
Truck: Turning right...
BUILD SUCCESSFUL (total time: 0 seconds)
```

Esimerkissä havaittiin, mitä tarkoittaa abstraktin luokan luominen sekä abstraktin metodin käyttö. Abstraktissa luokassa *Car* määritellään abstrakti metodi drive(), jolloin jokaisella *Car*-luokan perivällä luokalla (avainsana *extends*) on oltava toiminnallisuus tarjottuna kyseiselle metodille. Abstraktista luokasta on kuitenkin mahdollista tarjota metodeja, joita perivät luokat voivat käyttää, kuten esimerkin turnRight() ja turnLeft(). Esimerkissä havaitaan, että näkyvyysmääreenä ei ole nyt julkinen eikä yksityinen vaan peritymisen yhteydessä hyvin usein käytetty näkyvyysmääre *suojattu* (*protected*).

5.2. Näkyvyysmääre: suojattu (protected)

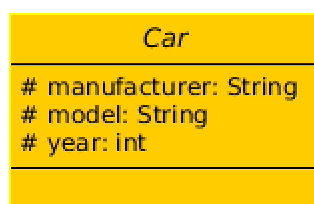
Suojattu näkyvyysmääre on kolmas (Javan tapauksessa neljäs) ja viimeinen näkyvyysmääre olio-ohjelmoinnissa. Suojattu näkyvyysmääre tarjoaa enemmän rajapintaa luokan toiminnasta ulospäin kuin yksityinen, mutta rajoittaa näkyvyyttä silti enemmän kuin julkinen. Suojattu näkyvyysmääre on hyvin usein käytössä periytymisen yhteydessä, sillä suojattu näkyvyys tarjoaa rajapinnan kyseessä olevan luokan lisäksi myös luokan periville luokille, jolloin esimerkissä 5.1. olevat suojatut muuttujat `manufacturer`, `model` ja `year` ovat muuttujia, joita myös luokan aliluokat voivat hyödyntää, kuten esimerkissä on nähtävissä `PassangerCar`-luokan rakentajassa.

Tavallisesti jopa suojatun näkyvyysmääreen käyttäminen muuttujien määreenä ei ole suotavaa olioajattelun näkökulmasta (suositus olisi yksityinen näkyvyysmääre), mutta suunniteltaessa on aina huomioitava, onko muuttujien piilottaminen aliluokilta tarpeellista ja hyödyllistä, varsinkin silloin, jos aliluokat tarvitsevat muuttujia toimintaansa varten. Tämä ilmentää tavallisesti huonoa suunnittelua. Piilottaminen olisi mahdollista muuttamalla muuttujien näkyvyysmääre yksityiseksi ja luomalla muuttaja- ja lukijametodit, joiden näkyvyysmääre olisi suojattu. Selvennetään näkyvyyttä taulukon avulla, johon otetaan mukaan myös Javan `package`-näkyvyysmääre, vaikkei se lukeudu olio-ohjelmoinnin tavallisiin määreisiin.

Taulukko 5. Näkyvyysmääreet (K=näkyy/E=ei näy)

Määre	Luokka	Paketti	Aliluokka	Maailma
public	K	K	K	K
protected	K	K	K	E
package	K	K	E	E
private	K	E	E	E

Suojattu näkyvyysmääre esitetään UML-kaavioissa käyttämällä #-merkkiä.

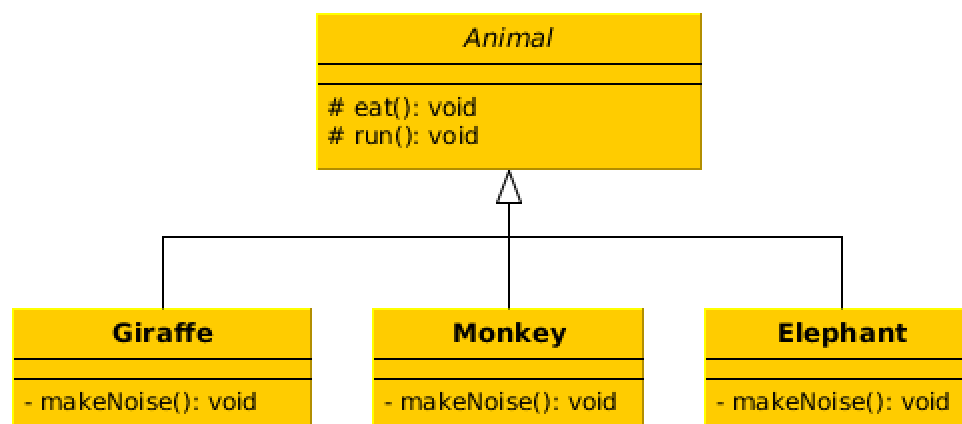


Kuva 5.2. Suojattu muuttuja UML-kaaviossa

Näkyvyyttä määritellessä on hyvä muistaa, että kantaluokasta perityn metodin näkyvyysmäärettä ei voida kiristää aliluokassa, mutta sitä voidaan löysätä. Kantaluokan suojattu metodi ei siis voi olla aliluokan yksityinen metodi, mutta se voi olla aliluokan julkinen metodi. Tällainen näkyvyysmääreillä kikkailu ei kuitenkaan pitäisi olla tarpeellista ja ilmentää huonoa suunnittelua, sillä abstrakti luokka tarjoaa rajapinnan, jota aliluokan on käytettävä täysin tai jätettävä rajapinta käyttämättä.

5.3. Toiminnan korvaaminen

On olemassa erilaisia mahdollisuuksia, joissa on toivottavaa, että kantaluokka tarjoaa toiminnallisuuden aliluokille, mutta yksi useasta aliluokasta tarvitsee erilaisen toteutuksen. Käytetään esimerkkinä edellisen luvun eläintarha-esimerkkiä, jossa esiteltiin kolme eläintä; apina, kirahvi ja elefantti. Eläimet periytettiin abstraktista Eläin-luokasta, jossa eläimille tarjottiin yhteinen eat()-metodi, mutta jokaisella eläimellä on oma makeNoise()-metodi, sillä jokainen eläin ääntelee eri tavalla. Lisäksi Eläin-luokka tarjoaa metodin run(), joka on jokaiselle eläimelle sama.



Esimerkki 5.3. Eläinten periytyminen

Esimerkki 5.4. Eläinten periytyminen ja toiminnan korvaaminen

```
abstract class Animal {
   protected void eat() {
      System.out.println("Munch munch.");
   }
   protected void run() {
      System.out.println("Step step step...");
   }
}
```

```
class Monkey extends Animal {
   @Override
   protected void run() {
      System.out.println("Jump jump jump...");
   }
   private void makeNoise() {
      System.out.println("Kree kree!!");
   }
}

class Giraffe extends Animal {
   private void makeNoise() {
      System.out.println("Moo moo!!");
   }
}

class Elephant extends Animal {
   private void makeNoise() {
      System.out.println("Troot troot!");
   }
}

public class Animals {

   public static void main(String[] args) {
      Monkey m = new Monkey();
      Giraffe g = new Giraffe();
      Elephant e = new Elephant();
      g.run();
      e.run();
      m.run();
   }
}
```

Tuloste

```
run:
Step step step...
Step step step...
Jump jump jump...
BUILD SUCCESSFUL (total time: 0 seconds)
```

Esimerkissä nähdään, kuinka Elephant ja Giraffe käyttävät Animal-luokan metodia run(), mutta Monkey-luokka korvaa kantaluokan toiminnan, mahdollistaen poikkeuksellisen toiminnan lisäämisen ilman, että kantaluokan on tarvetta muuttua. Kuten voidaan huomata, menetelmä on täysin sama kuin abstraktin metodin kanssa, mutta tällä kertaa kantaluokka tarjoaa toiminnallisuutta abstraktiuden sijaan. Voidaan siis puhua kantaluokan oletustoiminnallisuudesta.

6. Virhetilanteita ja periytymisiä

Edellisessä luvussa esitetyt esimerkit olivat hyvin yksinkertaisia helpon ymmärrettävyyden vuoksi. Tässä luvussa viedään periytymisen käsitettä hieman pidemmälle haastavampien esimerkkien avulla, tosi nämäkään esimerkit eivät tuo esille sitä, miten kompleksisia luokkarakenteet voivat todellisuudessa olla. Tämän lisäksi esitellään periytymiseen liittyvät avainsanat *interface* ja *implements*. Luvun lopuksi käydään läpi vielä virheitä ja niistä toipumista. Aloitetaan esimerkillä, jossa luodaan erilaisia polkupyöriä.

Esimerkki 6.1. Polkupyöräluokat

```
class Bicycle {
   private int cadence;
   private int gear;
   private int speed;

   public Bicycle(int c, int g, int s) {
      cadence = c;
      gear = g;
      speed = s;
   }
   public void setCadence(int newValue) {cadence = newValue;}
   public void setGear(int newValue) {gear = newValue;}
   public void applyBrake(int decr) {speed -= decr;}
   public void speedUp(int incr) {speed += incr;}
   public void printDescription() {
      System.out.println("\nBike is in gear " + this.gear
      + " with a cadence of " + this.cadence
      + " and travelling at a speed of " + this.speed + ".");
   }
}

class MountainBike extends Bicycle {
   private String suspension;
   public MountainBike(int c, int g, int s, String susp) {
      super(c, g, s);
      this.suspension = susp;
   }
   public String getSuspension() {return suspension;}
   public void setSuspension(String susp) {suspension = susp;}
   @Override
   public void printDescription() {
      super.printDescription();
      System.out.println("The mountainbike has a " + getSuspension()
      + " suspension.");
   }
}

class RoadBike extends Bicycle {
   private int tireWidth;
```

```
    public RoadBike(int c, int g, int s, int w) {
        super(c, g, s);
        tireWidth = w;
    }
    public int getTireWidth(){return tireWidth;}
    public void setTireWidth(int newTireWidth){tireWidth = newTireWidth;}
    @Override
    public void printDescription(){
        super.printDescription();
        System.out.println("The RoadBike" + " has " + getTireWidth() +
            " MM tires.");
    }
}

public class BikeTest {

    public static void main(String[] args) {
        Bicycle bike1, bike2, bike3;

        bike1 = new Bicycle(20, 10, 1);
        bike2 = new MountainBike(20, 10, 5, "Dual");
        bike3 = new RoadBike(40,20,8,23);

        bike1.printDescription();
        bike2.printDescription();
        bike3.printDescription();
    }
}
```

Tuloste

```
run:
Bike is in gear 10 with a cadence of 20 and travelling at a speed of 1.

Bike is in gear 10 with a cadence of 20 and travelling at a speed of 5.
The mountainbike has a Dual suspension.

Bike is in gear 20 with a cadence of 40 and travelling at a speed of 8.
The RoadBike has 23 MM tires.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Kuten esimerkissä nähdään, kantaluokan metodeita on mahdollista kutsua aliluokasta käyttämällä avainsanaa *super*. Tämän lisäksi aliluokasta voidaan kutsua kantaluokan rakentajaa eksplisiittisesti, varsinkin silloin kun kantaluokan rakentaja tarvitsee parametreja. Tällöin peritystä luokasta luotu olio sisältää myös kantaluokan olion, joka esitetään seuraavassa esimerkissä avainsanan *instanceof* avulla.

Esimerkki 6.2. Luodun luokan instanssin tyyppi

```
System.out.println(bike1 instanceof Bicycle);
System.out.println(bike2 instanceof MountainBike);
System.out.println(bike3 instanceof MountainBike);
```

```
System.out.println(bike3 instanceof Bicycle);
```

Tuloste

```
run:
true
true
false
true
BUILD SUCCESSFUL (total time: 0 seconds)
```

Avainsana *instanceof* kertoo, onko annettu olio instanssi pyydetystä luokasta. Esimerkissä nähdään, että bike1 on Bicycle-luokan instanssi ja bike2 on MountainBike-luokan instanssi. Näiden lisäksi bike3 ei ole MountainBike-luokan instanssi, sillä bike3 on luotu luokasta RoadBike, joka on periytetty luokasta Bicycle (jonka instanssi se myös on), jolloin MountainBike ei liity luokan rakentamiseen. Avainsanan avulla on yksinkertaista jaotella pääohjelman toiminnallisuus koskemaan vain tietyn luokan instansseja, jolloin voidaan olla varma esimerkiksi metodin olemassaolosta. Käytetään vanhaa esimerkkiä edellisestä luvusta, jossa pyrimme tulostamaan luokkien tyyppejä:

Esimerkki 6.3. instanceof

```
import java.util.ArrayList;

public class ListSample {

   public static void main(String[] args) {
      ArrayList list = new ArrayList();
      list.add("Hello world!");
      list.add(69);
      list.add(3.14);
      list.add("IT can be fun!");

      for(int i = 0; i < list.size(); i++) {
         System.out.print(list.get(i) + ": ");
         if(list.get(i) instanceof Integer) {
            System.out.println(((Integer) list.get(i)).TYPE);
         }
         else if(list.get(i) instanceof Double) {
            System.out.println(((Double) list.get(i)).TYPE);
         }
         else {
            System.out.println((list.get(i)).getClass().getSimpleName());
         }
      }
   }
}
```

Tuloste

```
run:
Hello world!: String
```

```
69: int
3.14: double
IT can be fun!: String
BUILD SUCCESSFUL (total time: 0 seconds)
```

6.1. Rajapinta (interface)

"*There are a number of situations in software engineering when it is important for disparate groups of programmers to agree to a "contract" that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, interfaces are such contracts.*" - Oracle
[URL: http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html]

Rajapintojen tarkoituksena on mahdollistaa kahden täysin erillisen ohjelmiston toiminta niin, että kumpikaan ei tiedä toisen tarkasta toteutuksesta. Tällaista kahden ohjelmiston välistä kanavaa kutsutaan rajapinnaksi, joka on kuin luokka, mutta ilman mitään toteutusta. Rajapinta voi siis sisältää vakioita, metodien märittelyitä, oletusmetodeita, staattisia metodeita ja sisäisiä luokkia. Rajapintaa ei ole mahdollista luoda, kuten ei abstraktia luokkaakaan, mutta rajapinta on mahdollista implementoida olion käyttöön.

Ajatellaan esimerkkinä futuristista autoa, joka osaa ajaa täysin ohjelmallisesti ilman ihmisen syötteitä. Auton valmistajan luoma ohjelma sisältää käskyjä kuten käynnistys, pysäytys, kiihdytys, kääntyminen, jne. ja auto toimii täysin käskyjen avulla. Auton valmistuttua toinen yritys luo järjestelmän, jonka avulla autoa on mahdollista ajaa GPS:n kautta kulkevalla datalla. Tällöin auton valmistajan on ollut tärkeää julkaista rajapinta eli API (Application Programming Interface), jossa määritellään millaisilla käskyillä autoa on mahdollista liikuttaa. Tämän rajapinnan avulla GPS-järjestelmän kehittäjät saavat selville metodien ja vakioiden nimet ja tarpeelliset tiedot auton toiminnan varmistamiseksi.

Esimerkin tapauksessa auton valmistajan ei tarvitse tietää mitään GPS-järjestelmästä ja GPS-järjestelmän luojan ei tarvitse tietää, miten erilaiset toiminnot on autoon implementoitu, ainoastaan se, miten järjestelmä voi saada auton toimimaan kuten halutaan. Rajapinnat (tuttavallisesti apit) ovat hyvin tärkeä osa erilaisten järjestelmien toteutuksessa, koska pyörää ei tarvitse lähteä keksimään uudestaan järjestelmää kehitettäessä. Hyvänä esimerkkinä toimii karttapalvelut, jotka luottavat tällä hetkellä pääasiassa Openlayers-karttarajapintaan (http://openlayers.org/), jonka avulla on mahdollista saada mikä tahansa kartta näkymään esimerkiksi web-sivulla. Tätä rajapintaa käytetään niin Bing-kartoissa (http://www.bing.com/maps/), Oskarissa (http://oskari.org/) kuin Googlen

Maps-palvelussa (https://www.google.fi/maps/preview). Rajapintojen avulla on mahdollista kytkeä toisiinsa kaksi täysin erilaista järjestelmää, joista yksi järjestelmä tarvitsee toisen apua toimiakseen.

6.2. Virheiden käsittely

"*An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons.*" - Tutorialspoint
[URL: http://www.tutorialspoint.com/java/java_exceptions.htm]

On täysin tavallista, että ohjelmaa luotaessa syntyy virheitä. Staattisia virheitä, kuten eriävät muuttujien nimet tai metodikutsut ovat tavallisia ja kääntäjä huomaa ne yleensä jo koodia kääntäessä. Näiden lisäksi on kuitenkin olemassa myös dynaamisia eli ajonaikaisia virheitä, joita kääntäjä ei voi havaita sillä ne ilmenevät, kuten nimestä voi päätellä, ajon aikana. Näiden virheiden käsittely on ohjelman toiminnan kannalta hyvinkin kriittistä, jolloin esimerkiksi on mahdollista ilmoittaa löytyykö tiedostoa vai ei. Ilman virheiden käsittelyä ohjelma vain nostaa kädet pystyyn ja kaatuu heti, kun tiedostoa ei löydy.

Virheet aiheuttavat ohjelman suorituksen estymisen joko kokonaan tai osittain, sillä virheen tapahtuessa ohjelmalla ei ole kykyä ymmärtää seuraavaa suoritettavaa askelta. Ohjelman huomatessa virheen, luovuttaa se suorituksensa ylemmälle taholle, kuten virheenkäsittelijälle. Esimerkkinä voidaan käyttää jakolaskua, jossa jakajan on aina oltava erisuuri kuin nolla. Jos jakajaan on kuitenkin ilmestynyt arvo nolla ja ohjelma pyrkii sitä väkisin jakamaan (muistutus: nollalla ei voi jakaa, ks. matematiikka), joudutaan poistumaan senhetkisestä suorituksesta ja kertomaan ohjelmalle, että näin ei voi tehdä.

Ohjelmalle kerrotaan kielletystä tekemisestä luomalla tai "heittämällä" virhe, joka on Javassa olio. Tämä virhe pysäyttää alkuperäisen suorituksen ja viittaus heitettyyn virheeseen lähetetään käsiteltäväksi. Virheen ilmaantuessa ohjelma etsii virheelle sopivan virheen käsittelijän (*exception handler*), jonka avulla on määritelty virheestä toipuminen, kuten esimerkiksi tiedon häviämisen estäminen. Virheenkäsittelijä tarjoaa ohjelmalle mahdollisuuden toipua heitetystä virheestä kaatumatta.

Virheisiin liittyy monia avainsanoja useissa eri ohjelmointikielissä, mutta Javassa näitä ovat *throw*, *try*, *catch* ja *finally*. Avainsana *throw* on työkalu virheen generointiin; sillä on mahdollista lähettää uusi virhe käsiteltäväksi, *try* ja *catch* ovat pari, jossa *try*-lohko pyrkii tavalliseen suoritukseen, mutta virheen sattuessa lähettää suorituksen *catch*-lohkolle. *finally*-lohkon avulla voidaan määrittää, mitä ohjelma suorittaa sen jälkeen, kun se poistuu *try*-lohkosta (normaalisti tai virheen kautta), jolloin

sen sisään voidaan määrittää resurssien sulkemisen (esimerkiksi tiedostokahva) ja olla varmoja siitä, että resursseja ei jää auki.

6.3. Javaa

Tämän luvun lopuksi käydään läpi esimerkin kautta, miten Javan *interface* ja *implements* avainsanat toimivat ja miten niitä käytetään. Kuvataan asia luomalla rajapinta Instrument, jonka implementoi useampi soitinluokka.

Esimerkki 6.4. Instrumentti-rajapinta

```
enum Note {
   MIDDLE_C, C_SHARP, B_FLAT
}

interface Instrument {
   public void play(Note n);
   public void adjust();
}

class Wind implements Instrument {
   public void play(Note n) {
      System.out.println(“Wind” + ".play() " + n);
   }
   public void adjust() {
      System.out.println(“Wind” + ".adjust()");
   }
}

class Percussion implements Instrument {
   public void play(Note n) {
      System.out.println(“Percussion” + ".play() " + n);
   }
   public void adjust() {
      System.out.println(“Percussion” + ".adjust()");
   }
}

class Stringed implements Instrument {
   public void play(Note n) {
      System.out.println(“Stringed” + ".play() " + n);
   }
   public void adjust() {
      System.out.println(“Stringed” + ".adjust()");
   }
}

class Brass extends Wind {}

class Woodwind extends Wind {}
```

```
public class Orchestra {

   public static void tune(Instrument i) {
      i.play(Note.MIDDLE_C);
   }
   public static void tuneAll(Instrument[] e) {
      for(Instrument i : e) { // foreach
         tune(i);
      }
   }
   public static void main(String[] args) {
      Instrument[] orchestra = {
         new Wind(),
         new Percussion(),
         new Stringed(),
         new Brass(),
         new Woodwind()
      };
      tuneAll(orchestra);
   }
}
```

Tuloste

```
run:
Wind.play() MIDDLE_C
Percussion.play() MIDDLE_C
Stringed.play() MIDDLE_C
Brass.play() MIDDLE_C
Woodwind.play() MIDDLE_C
BUILD SUCCESSFUL (total time: 0 seconds)
```

Esimerkissä huomataan, että rajapinnan määrittelyssä ei ole mahdollista lisätä toiminnallisuutta metodeille, vaikka ne olisivatkin jokaisella rajapinnan implementoivalla luokalla sisällöltään samanlaiset. Rajapinta kuitenkin vaatii, että jokainen sen implementoiva luokka sisältää samat toiminnallisuudet kuin rajapinta itse ja tämä ominaisuus löytyy ainakin Javasta, C#:sta ja PHP:stä. Javassa on myös huomattava, että jokainen luokka voi implementoida useita rajapintoja, mutta silti sen on sisällettävä jokaisen rajapinnan sisältämät metodit. Esimerkissä voidaan myös havaita, että rajapinnan implementoivan luokan aliluokat eivät enää joudu implementoimaan rajapinnan toiminnallisuuksia. Tämä on omalta osaltaan hyvin loogista, sillä aliluokalla on jo implementaatio rajapinnan metodeille kantaluokassaan.

6.3.1. Rajapinta vs. abstrakti luokka

Abstrakti luokka käsiteltiin aikaisemmin ja sen toteutuksessa huomattiin, että abstrakti luokka tuo ohjelmoijalle mahdollisuuden luoda metodeita määrittelevä kantaluokka, jonka metodeilla ei välttämättä ole toteutusta. Rajapinta taas tuo ohjelmoijalle mahdollisuuden viedä abstraktiota eteenpäin ja luoda täysin abstrakti luokka, jossa mitään implementaatiota ei ole. Rajapinta siis

määrittelee mallin, mutta ei suoritusta, jolloin sen voidaan sanoa olevan malli, millainen sen implementoiva luokka tulee rakenteeltaan olemaan.

Rajapinta on muutakin kuin vain äärirajoille viety abstrakti luokka, sillä se mahdollistaa ns. moniperinnän. Moniperinnässä luokka perii toiminnallisuutensa kahdesta tai useammasta kantaluokasta, mikä on tavallisesti olio-ohjelmoinnin vastaista ja ei ole suotavaa käytännössä, joskus kuitenkin välttämätöntä. Abstraktilla luokalla moniperintä ei ole mahdollista toteuttaa, mutta rajapintojen avulla tämä onnistuu, koska rajapinta ei sisällä toteutusta.

Esimerkki 6.5. Moniperintää rajapintojen avulla

```
interface CanFight {
   void fight();
}

interface CanSwim {
   void swim();
}

interface CanFly {
   void fly();
}

class ActionCharacter {
   public void fight() {System.out.println("Fighting!");}
}

class Hero extends ActionCharacter implements CanFight, CanSwim, CanFly {
   public void swim() {}
   public void fly() {}
}

public class MultipleInterfaces {

   public static void t(CanFight x) {x.fight();}
   public static void u(CanSwim x) {x.swim();}
   public static void v(CanFly x) {x.fly();}
   public static void w(ActionCharacter x) {x.fight();}
   public static void main(String[] args) {
      Hero h = new Hero();
      t(h);
      u(h);
      v(h);
      w(h);
   }
}
```

Tuloste

```
run:
Fighting!
Fighting!
```

```
BUILD SUCCESSFUL (total time: 1 second)
```

Esimerkissä luodaan kolme rajapintaa sekä ActionCharacter-luokka. ActionCharacter-luokasta periytetty Hero-luokka implementoi kaikki kolme rajapintaa ja tarjoaa kaikille rajapinnoille toteutuksen. Kuten esimerkissä voidaan havaita, Hero-luokan kantaluokka ActionCharacter implementoi metodin fight(), joka on määritelty rajapinnassa CanFight. Koska implementaatio on kantaluokassa, se löytyy myös aliluokasta, jolloin rajapinnan vaatima toteutus löytyy myös sieltä.

Pääluokassa, jossa ohjelma ajetaan, havaitaan myös, miten kantaluokka ja rajapinnat voivat toimia myös tietotyyppeinä tietoa liikuteltaessa. Esimerkiksi metodi u(), joka ottaa sisäänsä tietotyypin CanSwim-olion, voi käyttää ainoastaan metodia swim(), eli se kohtelee parametrina otettua oliota kuin rajapinnasta CanSwim luotua oliota. Tämähän siis ei ollut määritelmän puolesta mahdollista, sillä rajapinnasta oliota ei voi luoda, mutta näin se on mahdollista kiertää. Parametrien avulla oheisessa esimerkissä tehdään siis eksplisiittinen tyyppimuunnos, eli muutetaan Hero-olio joksikin muuksi olioksi.

6.3.2. Foreach

Esimerkissä 6.4. nähdään myös Javan erilainen implementaatio for-lausekkeesta, joka toimii foreachin tavoin (eli siis samalla tavalla kuin PHP:n foreach tai Pythonin for). Lausekkeella on mahdollista iteroida läpi tietorakennetta ja suorittaa sama käsky jokaiselle alkiolle.

```
public static void tuneAll(Instrument[] e) {
     for(Instrument i : e) { // foreach
        tune(i);
     }
   }
```

Syntaksissa nähdään, että senhetkinen alkio on vasemmalla puolella kaksoispistettä ja oikealla puolella on iteroitava tietorakenne, johon alkio sisältyy.

6.3.3. Virheiden käsittely

Käydään aluksi läpi yksinkertainen esimerkki, joka osoittaa try-catch-finally lohkojen yhteistoiminnan.

Esimerkki 6.6. Virheen kiinniotto ja suorituksen jatkaminen

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class HandleException {
```

```
    public static void main(String[] args) {
        PrintWriter out = null;

        try {
            System.out.println("Entering try-statement");

            out = new PrintWriter(new FileWriter("Output.txt"));

            for(int i=0;i < 10; i++) {
                out.println(i);
            }
        } catch (IOException e) {
            System.err.println("Caught IOException: " + e.getMessage());
        } finally {
            if(out != null) {
                System.out.println("Closing PrintWriter...");
                out.close();
            }
            else {
                System.out.println("PrintWriter is not open.");
            }
        }
    }
}
```

Virheetön tuloste

```
run:
Entering try-statement
Closing PrintWriter...
BUILD SUCCESSFUL (total time: 0 seconds)
```

Tuloste, jos virhe tapahtuu

```
run:
Entering try statement
Caught IOException: OutFile.txt
PrintWriter is not open
BUILD SUCCESSFUL (total time: 0 seconds)
```

Erilaisia poikkeuksia löytyy valmiina Javasta iso liuta, joten niitä ei käydä suuremmin läpi tässä yhteydessä. Kuitenkin, omaa ohjelmaa luotaessa on aina pohdittava jokainen mahdollinen virhe läpi ja sille on lisättävä käsittelijä jollekin tasolle ohjelmassa, tai sitten tätä tehdään testatessa. Tietenkään nämä valmiit käsittelijät eivät voi kattaa kaikkia mahdollisia virhetilanteita, joita ohjelmoija voi kohdata, jonka vuoksi on oltava mahdollista luoda myös omia virheitä sekä niiden käsittelijöitä. Tästä enemmän esimerkissä.

Esimerkki 6.7. Oman virheen luonti ja käsittely

```
class MyException extends Exception {
    public MyException() {}
    public MyException(String msg) {super(msg);}
}
```

```
public class DualConstructors {

   public static void f() throws MyException {
      System.out.println("Throwing MyException from f().");
      throw new MyException();
   }

   public static void g() throws MyException {
      System.out.println("Throwing MyException from g().");
      throw new MyException("Originated in g().");
   }

   public static void main(String[] args) {
      try {
         f();
      } catch (MyException e) {
         e.printStackTrace(System.out);
      }
      try {
         g();
      } catch (MyException e) {
         e.printStackTrace(System.out);
      }
   }
}
```

Tuloste

```
run:
Throwing MyException from f().
dualconstructors.MyException
        at dualconstructors.DualConstructors.f(DualConstructors.java:18)
        at dualconstructors.DualConstructors.main(DualConstructors.java:28)
Throwing MyException from g().
dualconstructors.MyException: Originated in g().
        at dualconstructors.DualConstructors.g(DualConstructors.java:23)
        at dualconstructors.DualConstructors.main(DualConstructors.java:33)
BUILD SUCCESSFUL (total time: 0 seconds)
```

Esimerkissä nähdään, kuinka virhe MyException perii luokalta Exception ominaisuutensa ja tarjoaa kaksi rakentajaa. Esimerkin metodi f() heittää virheen ilman parametria, joten se ei tulosta mitään viestiä, mutta g() ottaa merkkijonomuotoisen parametrin ja tulostaa sen virheen sattuessa. Tämä ominaisuus on peritty suoraan Exception-luokasta, jota kutsutaan super()-metodin avulla.

Esimerkissä havaitaan myös uusi tuttavuus nimeltä *throws*, joka sijaitsee metodin määritelmän perässä. Tällä määrittelyllä tarkennetaan metodin määrittelyssä se, millaisen virheen metodi tulee heittämään. Kuten esimerkissä voidaan nähdä, metodin g() sisällä käytetään *throw*-avainsanaa, eli metodissa generoidaan uusi virhe. Tämä ilmoitetaan kääntäjälle jo metodin määrittelyssä, jotta

kääntäjä osaa odottaa ja vaatia tätä virhettä metodilta, vaatien virheen käsittelyn kutsun yhteyteen. Näin varmistetaan, että metodin generoima virhe tulee käsiteltyä, eikä se vain unohdu.

6.3.4. final

Java sisältää myös avainsanan *final*, joka muuttuu sitä mukaa, missä sitä käytetään. Avainsanaa voidaan käyttää muuttujien, metodien tai luokkien kanssa, joissa se aiheuttaa erilaisia toimintoja.

Luokkien kanssa final-avainsana tarkoittaa sitä, että luokasta ei ole mahdollista periyttää enää uusia luokkia. Luokan julistaminen lopulliseksi parantaa sen turvallisuutta ja tehokkuutta, joten suuri osa Javan kirjastoista on julistettu lopullisiksi, esimerkiksi String-luokka. Lopullisen luokan sisältämät metodit ovat automaattisesti lopullisia, jotka noudattavat samaa kaavaa kuin lopullinen luokka, jolloin metodin toimintaa ei ole mahdollista korvata eikä metodia ole mahdollista piilottaa. Katsotaan pieni esimerkki piilottamisesta ja toiminnan korvaamisesta.

Esimerkki 6.8. Metodin korvaaminen ja piilottaminen

```
class Base {
   public void m1() {}
   public final void m2() {}

   public static void m3() {}
   public static final void m4() {}
}

class Derived extends Base {
   public void m1() {} // Ok, overriding
   public void m2() {} // Forbidden

   public static void m3() {} // Ok, hiding m3
   public static void m4() {} // Forbidden
}
```

Lopullinen muuttuja on nimensä perusteella jo helppo ymmärtää, sillä se on muuttuja, jonka arvoa ei voida enää muuttaa. Semantiikaltaan se muistuttaa hyvin paljon C++:n *const*-avainsanaa (joka löytyy myös Javasta avainsanana), mutta final-avainsana on siitä kiero, että sitä ei tarvitse alustaa heti, jolloin muuttuja voidaan jättää tyhjäksi ja ensimmäinen siihen asetettu arvo on lopullinen.

Lopullinen metodi tarkoittaa vuorostaan sitä, että metodia ei voida enää korvata kantaluokasta perityssä luokassa, kuten esimerkissä nähdään. Samalla final-avainsana estää metodin piilottamisen, jolla tarkoitetaan staattisen metodin, luokkamuuttujia käyttävän metodin, toiminnan korvaamista. Luokkamuuttujsita ja avainsanasta *static* lisää seuraavassa luvussa.

Lopullinen luokka vuorostaan tarkoittaa, että luokkaa ei voida käyttää kantaluokkana eli siitä ei voi enää periytyä uusia luokkia. Olio-ohjelmoinnissa periytyminen on voimakas työkalu ja yksi perusajatuksista, mutta joissakin tapauksissa luokka on luotava niin, että siitä ei voida enää periä ominaisuuksia. Lopullinen luokka on myös yksi tapa vahvistaa aikaisemmin mainittuja suunnittelumetodeja (luku 4, taulukko 4), joissa kehoitetaan käyttämään kompositiota ennemmin kuin periytymistä. Käyttämällä avainsanaa `final` luokan muodostamisessa vahvistetaan hyviä suunnitteluperiaatteita.

7. Kopiointi ja sijoitus (sekä luokkamuuttujat)

"*Classes are used to model concepts in object-oriented systems. Instances or objects of these are created and used throughout an application. It is not unusual that we are interested in making a copy of an object during the run time. How does one make such a copy?*" -Venkat Subramaniam [URL: http://www.agiledeveloper.com/articles/cloning072002.htm]

Joskus ohjelmoidessa on tarpeellista luoda kopioita olemassa olevista olioista. Kopioita voidaan olio-ohjelmoinnin maailmassa tehdä useita erilaisia, kuten viitekopioita, matalakopioita ja syväkopioita. Tämän lisäksi olioiden sijoittaminen toistensa paikalle on mahdollista. Käydään tässä luvussa läpi jokainen näistä kopiointitavoista ja pohditaan lopussa toteutusta Javalla.

7.1. Viitekopiointi

Viitekopiointi on kopiointitavoista yksinkertaisin ja sitä käytetään etenkin oliokielissä, missä muuttujat ovat viittauksia olioihin. Viitekopiointi on yksinkertainen ja nopea tapa suorittaa kopiointia ja sitä tapahtuu varsinkin funktiokutsuissa, sillä funktion parametri on aina kopio alkuperäisestä. On huomattava, jos funktion parametriksi annetaan primitiivinen tietotyyppi, kuten int, annetaan funktiolle tällöin kopio alkuperäisestä ja alkuperäinen ei muutu, vaikka funktiolle annettu arvo muuttuisikin.

Esimerkki 7.1. Primitiivisten tietotyyppien kopiointi

```
public class CopyTest {

   public static void badSwap(int par1, int par2) {
      int temp = par1;
      par1 = par2;
      par2 = temp;
   }

   public static void main(String[] args) {
      int var1 = 10, var2 = 5;
      System.out.println(var1 + " " + var2);
      badSwap(var1, var2);
      System.out.println(var1 + " " + var2);
   }
}
```

Tuloste

```
run:
10 5
10 5
BUILD SUCCESSFUL (total time: 0 seconds)
```

Jos samaa operaatiota esimerkistä lähdetään tekemään olioiden avulla, systeemi monimutkaistuu lisää. Javassa lähetetään myös oliot kopioina, mutta itse olio ei kopioidu vaan sen viite. Kuulostaa monimutkaiselta, joten pyritään selventämään tätä esimerkin kautta, jolloin asian pitäisi aueta paremmin.

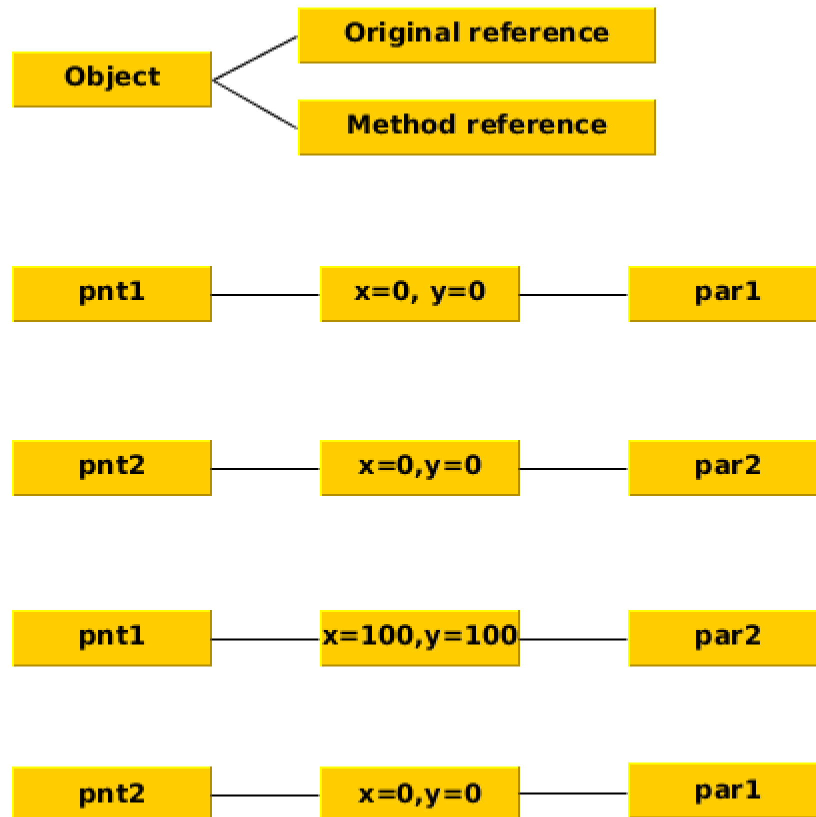
Esimerkki 7.2. Viitteiden kopiointi

```
public class CopyTest {

   public static void tricky(Point par1, Point par2) {
      par1.x = 100;
      par1.y = 100;
      Point temp = par1;
      par1 = par2;
      par2 = temp;
   }

   public static void main(String[] args) {
      Point pnt1 = new Point(0,0);
      Point pnt2 = new Point(0,0);
      System.out.println("X: " + pnt1.x + " Y: " +pnt1.y);
      System.out.println("X: " + pnt2.x + " Y: " +pnt2.y);
      tricky(pnt1, pnt2);
      System.out.println("X: " + pnt1.x + " Y: " +pnt1.y);
      System.out.println("X: " + pnt2.x + " Y: " +pnt2.y);
   }
}
```

Tuloste

```
run:
X: 0 Y: 0
X: 0 Y: 0
X: 100 Y: 100
X: 0 Y: 0
BUILD SUCCESSFUL (total time: 0 seconds)
```

Esimerkissä haluamme vaihtaa kahden muuttujan paikkaa, sama kuin edellä kokonaislukujen kanssa. Nyt onnistuimme vaihtamaan olion sisältämiä arvoja ilman paluuarvoja funktiosta, mutta edelleen huomaamme, että ne ovat samoin päin kuin alussa, eli saimme arvot muuttumaan, mutta emme saaneet olioita vaihtumaan. Havainnollistetaan kuvan avulla, mitä esimerkissä juuri tapahtui.



Kuva 7.1. Viitekopiointia

Kuvassa on alkuperäinen tilanne, jossa olio on jo olemassa ja siihen liittyy alkuperäinen viittaus sekä viittaus, joka on metodissa. Eli Object on Point-olio, alkuperäinen viittaus on pnt1 ja metodin viittaus on par1. Alkutilanne ennen vaihtoa pnt1 ja par1 viittaavat samaan olioon sekä pnt2 ja par2 toiseen olioon. Vaihdon jälkeen huomaamme, että viite par2 viittaa nyt samaan olioon kuin pnt1 ja sama toisin päin, eli ainoastaan metodin viittaukset kääntyivät, alkuperäisiin viittauksiin tai olioihin tämä ei vaikuta.

7.2. Matalakopiointi

Matalakopiointi on Javassa oletuskopiointitapa, johon se tarjoaa clone()-metodin Object-luokassa. Matalakopioinnin tarkoitus on luoda kopio olemassa olevasta oliosta ja tarjota kopiolle käyttöön kopioitavan instanssin kaikki kentät. Matalakopioinnin tuloksena on olemassa kaksi identtistä oliota, jotka jakavat samat resurssit keskenään.

Ajatellaan esimerkkinä matalakopioinnista autotallia, jossa on useampi auto. Halutaan kopioida kyseessä oleva autotalli, jolloin meillä olisi lisää tilaa autoille, mutta emme kuitenkaan halua kopioida autotallin sisällä olevia autoja. Kopioikaamme siis matalasti pelkkä autotalli, jättäen autot

molempien autotallien yhteiseen käyttöön. Ohessa on kuva järjestelmän tilasta ennen ja jälkeen matalakopioinnin.

Before

After

Kuva 7.2. Matalakopiointia

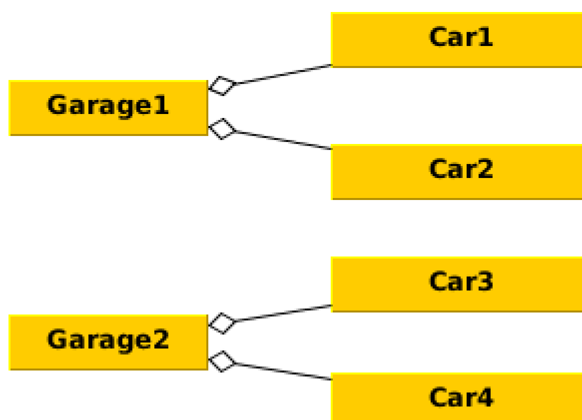
Matalakopiointi on hyvin selkeä operaatio, jonka tuloksena saadaan aina kaksi oliota. Näiden olioiden jakamat tietorakenteet ja resurssit voivat silti muodostaa ongelmia, sillä muutos tietorakenteessa tarkoittaa muutosta jokaisessa matalakopioidussa oliossa. Joskus tämä voi olla hyvä asia, mutta se tuottaa myös ongelmia olioiden toiminnassa. Joissakin tapauksissa olioiden toiminnan takaaminen vaatii jokaisen oliokohtaisen tietorakenteenkin kopioimista, jolloin matalakopiointi ei riitä vastaamaan tätä operaatiota. Siksi meillä on käytössä myös syväkopiointi.

7.3. Syväkopiointi

Syväkopioinnissa kopioidaan alkuperäisen olion lisäksi kaikki olion ulkopuoliset, mutta siihen liittyvät resurssit ja tietorakenteet. Tämä on olion kannalta paras mahdollinen kopiointitapa, sillä se mahdollistaa kahden identtisen, toisistaan irrallaan olevan olion muodostamisen. Kuitenkin syväkopioinnin järkevä käyttö on hankalaa, sillä se kopioi sellaisiakin tietorakenteita, joita ei olisi tarpeen kopioida, esimerkiksi kopio kirjaston kirjasta kopioisi myös koko kirjaston ja kaikki sen kirjat tai auton kopiointi kopioisi autotallin ja kaikki sen sisältämät autot. Seuraavassa on vielä esimerkkiä siitä, miten syväkopiointi tapahtuu.

Before

After



Kuva 7.3. Syväkopiointi

7.4. Kopiointi käytännössä

Käytännössä olioiden kopioimiseen on kaksi mahdollista suoritustapaa. Ensimmäinen on käyttää Javan tarjoamaa clone()-metodia, mutta se suorittaa tavallisesti vain matalakopiointia. Toinen keino on luoda oliolle kopiomuodostin, jossa kopioitava instanssi alustetaan.

Esimerkki 7.3. Kopiomuodostin

```
public final class Galaxy {
   private double amass;
   private final String name;

   public Galaxy(double mass, String name) {
      amass = mass;
      this.name = name;
   }

   public Galaxy(Galaxy another) {
      this(another.getMass(), another.getName());
   }

   public double getMass() {return amass;}
   public String getName() {return name;}
   public void setMass(double mass) {amass = mass;}
```

```
    public static void main(String[] args) {

        Galaxy e220 = new Galaxy(25, "e220");

        Galaxy copy_e220 = new Galaxy(e220);
        copy_e220.setMass(400);
        System.out.println("Original mass: " + e220.getMass());
        System.out.println("Copy's mass: " + copy_e220.getMass());
    }
}
```

Tuloste

```
run:
Original mass: 25.0
Copy's mass: 400.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

Esimerkissä luotiin kopiomuodostin, joka on identtinen tavallisen muodostimen kanssa, mutta ottaa parametrikseen samasta luokasta luodun instanssin. Esimerkki on hyvin yksinkertainen, eikä siinä oleviin olioihin kiinnity muita olioita tai tietorakenteita. Lisäksi esimerkin kopiomuodostimessa on hieman ongelmia, joita voidaan lähteä pohtimaan seuraavan esimerkin avulla, jossa kopioitavaa luokkaa laajennetaan.

Esimerkki 7.4. Uusi kopiomuodostin

```
class Brain {
    public Brain() {}
    public Brain(Brain another) {}
}

public class Person {

    private Brain brain;
    private int age;

    public Person(Brain abrain, int age) {
        brain = abrain;
        this.age = age;
    }

    public Person(Person another) {
        age = another.age;
        brain = new Brain(another.brain);
    }
    public String toString() {
        return "This is a person with " + brain;
    }

    public static void main(String[] args) {
        Person sam = new Person(new Brain(), 1);
```

```
        Person bob = new Person(sam);
        System.out.println(sam);
        System.out.println(bob);
    }
}
```

Tuloste

```
run:
This is a person with person2.Brain@190690e
This is a person with person2.Brain@1a8c064
BUILD SUCCESSFUL (total time: 0 seconds)
```

Samalla tavalla kuin edellisessä esimerkissä, kopioimme yhdestä oliosta toisen. Nyt esimerkkiä laajennettiin kattamaan Person-luokka, johon liittyy Brain-luokka niin, että ihmisellä on aivot. Esimerkissä henkilö kopioidaan ja kopiomuodostimessa myös henkilön aivot kopioidaan, jolloin meillä on ohjelmassa kaksi ihmistä, joilla on molemmilla omat aivonsa. Tästä saataisiin muodostettua matalakopio jättämällä uusien aivojen muodostaminen kopioimisen ohessa pois ja asetettaisiin vain parametrina saadut aivot kopion aivoiksi. Tällaisella ratkaisulla voidaan ratkoa kopiointiongelmia tiettyyn pisteeseen asti, mutta entä jos aivot voisivat olla joko tavalliset aivot tai hieman älykkäämmät aivot. Siitä seuraavassa.

Esimerkki 7.5. Kopiomuodostin jatkettuna

```
class Brain {
    public Brain() {}
    public Brain(Brain another) {}
}

class SmarterBrain extends Brain {
    public SmarterBrain() {}
    public SmarterBrain(SmarterBrain another) {}
}

public class Person {

    private Brain brain;
    private int age;

    public Person(Brain abrain, int age) {
        brain = abrain;
        this.age = age;
    }

    public Person(Person another) {
        age = another.age;
        brain = new Brain(another.brain);
    }
    public String toString() {
        return "This is a person with " + brain;
    }

    public static void main(String[] args) {
```

```
        Person sam = new Person(new SmarterBrain(), 1);
        Person bob = new Person(sam);
        System.out.println(sam);
        System.out.println(bob);
    }
}
```

Tuloste

```
run:
This is a person with person2.SmarterBrain@193229
This is a person with person2.Brain@b368f
BUILD SUCCESSFUL (total time: 0 seconds)
```

Esimerkki rakennettiin jatkamaan edellistä, mutta nyt tulostuksesta huomataan, että aivojen kopioiminen jäi hieman kesken johtaen siihen, että älykkäiden aivojen omistajasta kopioitui vain keskivertoaivojen omistaja. Tällaisissa tapauksissa voidaan pohtia kopioimista esimerkiksi *instanceof* -avainsanan avulla, mutta se ei ole välttämättä paras ratkaisu. Jotta perитynkin luokan kopiointi olisi mahdollista, on kopiointi vietävä hieman pidemmälle ja siinä on käytettävä sekä kopiomuodostinta että clone()-metodia.

Esimerkki 7.6. Lopullinen kopion luominen

```
class Brain implements Cloneable {
   public Brain() {}
   public Brain(Brain another) {}
   public Object clone() throws CloneNotSupportedException {return super.clone();}
}

class SmarterBrain extends Brain {
    public SmarterBrain() {}
    public SmarterBrain(SmarterBrain another) {}
    public Object clone() throws CloneNotSupportedException {
        SmarterBrain another = (SmarterBrain) super.clone();
        return another;
    }
}

class Person implements Cloneable {

    private final Brain brain;
    private int age;

    public Person(Brain abrain, int age) {
        brain = abrain;
        this.age = age;
    }

    public Object clone() {
        return new Person(this);
    }

    protected Person(Person another) {
```

```
        Brain refBrain = null;
        try {
            refBrain = (Brain) another.brain.clone();
        } catch (CloneNotSupportedException ex) {}
        brain = refBrain;
        age = another.age;
    }

    public String toString() {
        return "This is a person with " + brain;
    }
}

class SkilledPerson extends Person {
    private String theskills;
    public SkilledPerson(Brain brain, int age, String skills) {
        super(brain, age);
        theskills = skills;
    }
    protected SkilledPerson(SkilledPerson another) {
        super(another);
        theskills = another.theskills;
    }
    public Object clone() {
        return new SkilledPerson(this);
    }
    public String toString() {
        return "SkilledPerson: " + super.toString();
    }
}

public class User {
    public static void play(Person p) {
        Person another = (Person) p.clone();
        System.out.println(p);
        System.out.println(another);
    }

    public static void main(String[] args) {
        Person sam = new Person(new Brain(), 1);
        play(sam);

        SkilledPerson bob = new SkilledPerson(new SmarterBrain(), 1, "Writer");
        play(bob);
    }
}
```

Tuloste

```
run:
This is a person with person2.Brain@1dbe1c9
This is a person with person2.Brain@17d56b
SkilledPerson: This is a person with person2.SmarterBrain@bc9f58
SkilledPerson: This is a person with person2.SmarterBrain@1816fb6
BUILD SUCCESSFUL (total time: 0 seconds)
```

Esimerkissä luodaan henkilö, jolla on tavalliset aivot, jonka jälkeen tämä henkilö kopioidaan. clone()-metodia kutsuttaessa nähdään, kuinka se kutsuu sisällään olion kopiomuodostinta, jossa taas kutsutaan aivojen clone()-metodia. Aivot siis kopioituvat ensin ja saavat tyyppinsä omistavalta oliolta, jolloin kopion valmistus alkaa aivoista. Näiden valmistuttua palautetaan henkilöolio takaisin alkuperäiselle kutsujalle, jolloin meillä on valmis kopio henkilöstä omilla aivoillaan. Samalla tavalla tapahtuu jälkimmäinen kutsu, jossa luodaan lahjakas ihminen, jolle annetaan paramterina älykkäämmät aivot rakentajaan. Kopiomuodostimessa kutsutaan henkilön kopiomuodostinta täysin periytymisen sääntöjen mukaisesti ja clone()-metodi toimii täysin identtisesti kantaluokan toimintaan nähden. Esimerkin avulla on siis mahdollista luoda erilaisia kantaluokasta periytettyjä oliota ja kantaluokan olioita, jotka voivat muodostua erilaisista olioista, jotka kuitenkin ovat liitoksissa toisiinsa. Esimerkissä voi myös havaita useita tuttuja avainsanoja, kuten *implements*, *final, static* ja *extends*.

7.5. Sijoittaminen

Sijoittaminen voidaan helposti ymmärtää tavalliseksi arvon asettamiseksi muuttujaan ja se on käytännössä vain kopiointiin verrattavaa olion muokkausta. Kopioinnissa luodaan aina uusi olio, joka on samanlainen kuin vanha olio, mutta sijoittamisen tarkoituksena on muokata vanha jo olemassa oleva olio muistuttamaan toista oliota.

Javan sijoittamisessa on huomattava, ettei jokaista oliota voi sijoittaa (tai sen viittausta muuttaa) mihin vain, sillä viittaukset on alustettava tietotyyppikohtaisesti. On siis huomattavaa, että periytetty luokka voidaan asettaa kantaolion viitteen päähän, mutta kantaolioon ei voi viitata periytetyn olion viitteen kautta.

Esimerkki 7.7. Olioiden sijoittamista

```
class A {}
class B extends A {}
class C extends A {}

public class SampleAssignment {

   public static void main(String[] args) {
      A a = new A();
      B b = new B();
      C c = new C();

      a = c; // Legal
      b = c; // Illegal
      b = a; // Illegal
   }
}
```

7.6. Luokkamuuttujista ja static-avainsanasta

"*In object-oriented programming with classes, a class variable is a variable defined in a class (i.e. a member variable) of which a single copy exists, regardless of how many instances of the class exist.*" - Wikipedia [URL: http://en.wikipedia.org/wiki/Class_variable]

Luokkamuuttujista oli puhetta jo ensimmäisessä luvussa, mutta niiden käytöstä ei ollut vielä tarkempaa keskustelua. Lähdetään liikkeelle esimerkin voimin.

Esimerkki 7.8. Luokkamuuttujan toiminta

```
public class SampleStatic {
   public static int amount = 0;
   public SampleStatic() {
      amount++;
   }
   public void printAmount() {
      System.out.println(amount);
   }
   public static void main(String[] args) {
      SampleStatic ex1 = new SampleStatic();
      ex1.printAmount();
      SampleStatic ex2 = new SampleStatic();
      ex1.printAmount();
      SampleStatic ex3 = new SampleStatic();
      ex1.printAmount();
   }
}
```

Tuloste

```
run:
1
2
3
BUILD SUCCESSFUL (total time: 0 seconds)
```

Kuten esimerkissä nähdään, staattinen muuttuja eli luokkamuuttuja muuttuu aina, kun olioita luodaan. Luokkamuuttuja ei ole olion omistama muuttuja, vaan se jaetaan jokaisen luodun instanssin kanssa, jolloin esimerkissä jokaisen olion rakentaja kasvattaa lukua yhdellä. Tätä ominaisuutta voidaan käyttää esimerkiksi Singleton-suunnittelumallissa (luokasta on olemassa vain yksi instanssi). Enemmän tietoa Singeltonista kiinnostuneille löytyy Wikipediasta osoitteessa: http://en.wikipedia.org/wiki/Singleton_pattern. Staattisesta metodista (esimerkissäkin oleva main) huomataan, että sen sisällä ei ole mahdollista käyttää this-avainsanaa, sillä metodi ei tiedä, mikä

instanssi sitä kutsui. Staattista metodia ei myöskään kutsuta olemassa olevan instanssin avulla vaan sitä voidaan kutsua ilman oliota.

Esimerkki 7.9. Singleton-luokka

```
public class SampleSingleton {
   private static SampleSingleton instance = null;

   protected SampleSingleton() {}

   public static SampleSingleton getInstance() {
      if(instance == null) {instance = new SampleSingleton();}
      else {System.out.println("Instance already exists.");}
      return instance;
   }
   protected void printMethod() {
      System.out.println(this);
   }
   public static void main(String[] args) {
      SampleSingleton sample = SampleSingleton.getInstance();
      sample.printMethod();
      SampleSingleton sample2 = SampleSingleton.getInstance();
      sample2.printMethod();
   }
}
```

Tuloste

```
run:
samplesingleton.SampleSingleton@1db9742
Instance already exists.
samplesingleton.SampleSingleton@1db9742
BUILD SUCCESSFUL (total time: 0 seconds)
```

8. Sisäiset luokat

"*In my humble opinion, the most important feature of the inner class is that it allows you to turn things into objects that you normally wouldn't turn into objects. That allows your code to be even more object-oriented than it would be without inner classes.*" - Tony Sintes from ObjectWave
[URL: http://www.javaworld.com/article/2077411/core-java/inner-classes.html]

Sisäiset luokat ovat olio-ohjelmoinnissa hyvin laaja käsite, mutta niitä käsitellään hyvin rajatusti tässä oppaassa. Lukijalle suositellaan etsimään oppaan ulkopuolelta tietoa sisäisistä luokista, upotetuista luokista, abstrakteista sisäisistä luokista ja niin edelleen, sillä ne ovat jo niin edistynyttä asiaa, ettei niitä tämän oppaan rajoissa voida esitellä. Sisäiset luokat ovat yksinkertaisesta luonteestaan huolimatta hyvin monimutkainen rakenne, johon jokaisen ohjelmoijan kannattaa tutustua esimerkiksi täältä: http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html.

Sisäiset luokat ovat näppärä keino koota yhteen paikkaan toisiinsa kiinteästi liittyviä luokkia ja näin muokata niiden näkyvyyttä. On kuitenkin tärkeää ymmärtää, että sisäinen luokka ei tarkoita samaa kuin aikaisemmassa luvussa käsitelty kompositio. Vaikka sisäinen luokka vaikuttaa vain keinolta piilottaa luokan suoritus toisen luokan sisään, on huomattava, että sisäinen luokka tietää ulkoisen luokan olemassaolosta ja pääsee käsittelemään sitä. Tämän kautta tuotettu lopputuote sisältää mahdollisesti selkeämpää ja paremmin toimivaa koodia, mutta sisäinen luokka ei kuitenkaan ole takuu tästä. Käydään sisäistä luokkaa läpi esimerkin avulla.

Esimerkki 8.1. Sisäinen luokka

```
public class Parcell {

   class Contents {
      private int i = 11;
      public int value() {return i;}
   }
   class Destination {
      private String label;
      Destination(String loc) {label = loc;}
      public String readLabel() {return label;}
   }
   public Destination to(String s) {
      return new Destination(s);
   }
   public Contents contents() {
      return new Contents();
   }
   public void ship(String dest) {
      Parcell.Contents c = contents();
      Parcell.Destination d = to(dest);
```

```
      System.out.println(d.readLabel());
    }
    public static void main(String[] args) {
      Parcell p = new Parcell();
      p.ship("Tasmania");
    }
}
```

Tuloste

```
run:
Tasmania
BUILD SUCCESSFUL (total time: 0 seconds)
```

Esimerkissä nähdään, että Parcell-luokka pitää nyt sisällään kaksi luokkaa, eli Contents ja Destination. Nämä sisäiset luokat liittyvät hyvin kiinteästi Parcel-luokan suoritukseen, joten niiden on järkevämpää sijaita luokan sisällä kuin sen ulkopuolella. Varsinkin, jos Parcel-luokka tarvitsee useamman kohteen tai sen sisältö vaihtuu suorituksen aikana.

Edellisessä esimerkissä nähdään, kuinka sisäisillä luokilla voidaan järjestää koodia ja piilottaa olioiden suorituksia, mutta sisäiset luokat tarjoavat myös enemmän ominaisuuksia. Sisäisellä luokalla on myös mahdollisuus päästä käsiksi ulkoisen luokan, eli kietovan luokan toiminnallisuuksiin ilman erityisiä vaatimuksia.

Esimerkki 8.2. Sisäisen ja ulkoisen luokan kommunikointi

```
interface Selector {
    boolean end();
    Object current();
    void next();
}

public class Sequence {

    private Object[] items;
    private int next = 0;
    public Sequence(int size) { items = new Object[size]; }
    public void add(Object x) {
      if(next < items.length) {
        items[next++] = x;
      }
    }
    private class SequenceSelector implements Selector {
      private int i = 0;
      public boolean end() { return i == items.length; }
      public Object current() { return items[i]; }
      public void next() { if(i < items.length) i++; }
    }
    public Selector selector() {
      return new SequenceSelector();
    }
```

```
    public static void main(String[] args) {
        Sequence s = new Sequence(10);
        for(int i = 0; i < 10; i++) {
            s.add(Integer.toString(i));
        }
        Selector selector = s.selector();
        while(!selector.end()) {
            System.out.print(selector.current() + " ");
            selector.next();
        }
        System.out.println();
    }
}
```

Tuloste

```
run:
0 1 2 3 4 5 6 7 8 9
BUILD SUCCESSFUL (total time: 0 seconds)
```

Esimerkin Sequence-luokka on vain taulukon kietova luokka, joka tarjoaa lisäämistä varten toiminnallisuuden taulukolle. SequenceSelector-luokka taas tarjoaa mahdollisuuden iteroida taulukko läpi askel kerrallaan (kuten Javan Enumeration- tai Iterator-luokat) sekä mahdollistaa viimeisen alkion ja käytössä olevan alkion havaitsemisen. Esimerkissä luodaan uusi Sequence-olio, johon lisätään alkioita ja nämä alkiot sitten iteroidaan läpi sisäisen SequenceSelector()-olion avulla. Esimerkissä on syytä huomata, että SequenceSelector käyttää iteroimiseen items-muuttujaa, joka on Sequence-luokan yksityinen muuttuja, eli sisäisen luokka voi käsitellä ulkoisen luokan muuttujia aivan kuin se omistaisi ne.

9. Java ja internet

“*Java and Internet: Java is strongly associated with the Internet. Internet users can use Java to create applet programs and run them locally using a "Java-enabled browser" such as HotJava. They can also use a Java-enabled browser to download an applet located on a computer anywhere in the Internet and run it on his local computer. In fact, Java applets have made the Internet a true extension of the storage system of the local computer.*” - Smartclass.co
[URL: http://www.smartclass.co/2011/10/java-internetii-java-and-www.html]

Java on ohjelmointikielistä eniten käytetty Internetin ohessa, sillä kukapa tietokonetta käyttävä ei olisi kuullut uutisia, missä tietoturvariskeistä syytetään tietokoneen Javaa. Tämä ei todellakaan viittaa kielen heikkouksiin, vaan siihen, miten Javaa on mahdollista upottaa verkkosivuille appletteina ja miten niitä on mahdollistakäyttää hyväksi sekä laillisesti että laittomasti. Oppaan tässä luvussa ei käydä kuitenkaan läpi appletin rakentamista, vaan käydään hieman läpi Javan mahdollisuuksia luoda yhteys Internetissä sijaitseviin sivustoihin. Lähdetään liikenteeseen aivan perusteista eli yhteyden muodostamisesta, johon liittyy vain sivun lähdekoodin hakeminen ja sen tulostus.

Esimerkki 9.1. URL-luokka ja html

```
import java.net.*;
import java.io.*;

public class ConnectToInternet {

   public static void main(String[] args) throws Exception {
      URL url = new URL("https://noppa.lut.fi/");
     BufferedReader in = new BufferedReader(
                 new InputStreamReader(url.openStream()));

      String inputLine;
      while((inputLine = in.readLine()) != null) {
         System.out.println(inputLine);
      }
      in.close();
   }
}
```

Tuloste

```
run:
<HTML>
<HEAD>
<meta http-equiv="refresh" content="0; URL=https://noppa.lut.fi/noppa">
</HEAD>
<BODY>
</BODY>
```

```
</HTML>

BUILD SUCCESSFUL (total time: 1 second)
```

Tämähän ei ole itsessään kovinkaan näppärä työkalu, mutta internetistä on mahdollista ladata useita erilaisia tietopankkeja, tavallisesti XML-muotoisena (Extensible Markup Language). Internetissä vapaassa jaossa olevaa tietoa kutsutaan myös nimellä avoin data, sillä se on jonkin tahon vapaaseen jakoon keräämää dataa, jota kuka tahansa voi käyttää. Avoimesta datasta puhutaan enemmän oppaan seuraavassa osassa. Seuraavassa esimerkissä käytämme Open Weather Map-palvelua, josta on mahdollista hakea säädataa pelkästään URL:n avulla.

Esimerkki 9.2. Datan hakeminen verkosta

```
import java.io.*;
import java.net.*;

public class WeatherData {

   public static void main(String[] args) throws Exception {
     URL url = new URL("http://api."
           + "openweathermap.org/data/2.5/weather?q=lappeenranta&mode=xml");
      BufferedReader in = new BufferedReader(
         new InputStreamReader(url.openStream()));
      String inputLine;

      while((inputLine = in.readLine()) != null) {
         System.out.println(inputLine);
      }
      in.close();
   }
}
```

Tuloste

```
run:
<?xml version="1.0" encoding="utf-8"?>
<current>
  <city id="648900" name="Lappeenranta">
    <coord lon="28.19" lat="61.06"/>
    <country>Finland</country>
    <sun rise="2014-04-30T02:00:24" set="2014-04-30T18:08:25"/>
  </city>
  <temperature value="282.15" min="282.15" max="282.15" unit="kelvin"/>
  <humidity value="31" unit="%"/>
  <pressure value="1005" unit="hPa"/>
  <wind>
    <speed value="4.6" name="Gentle Breeze"/>
    <direction value="320" code="NW" name="Northwest"/>
  </wind>
  <clouds value="0" name="sky is clear"/>
  <precipitation mode="no"/>
```

```
  <weather number="800" value="Sky is Clear" icon="01d"/>
  <lastupdate value="2014-04-30T11:20:00"/>
</current>

BUILD SUCCESSFUL (total time: 0 seconds)
```

Ja dataahan sieltä tuli. Tämä kerätty data on nyt merkkijono, eli String-olio, mutta siitä tulisi rakentaa XML-muotoinen dokumentti, jotta voisimme etsiä ja muotoilla tietoa omiin tarpeisiimme. Olisi myös huomattavasti miellyttävämpää, jos tätä olisi mahdollista parsia jotenkin mukavammin luettavaan muotoon. Se tosin ei ole aivan niin yksinkertaista kuin voisi toivoa. Lisätäänpä ohjelmaan jotakin toiminnallisuutta.

Esimerkki 9.3. Säädatan keruu

```
import java.io.*;
import java.net.*;
import java.util.Map;

public class WeatherData {

   private String gatherData() throws Exception {
      URL url = new URL("http://api."
            + "openweathermap.org/data/2.5/weather?q=lappeenranta&mode=xml");
      BufferedReader in = new BufferedReader(
         new InputStreamReader(url.openStream()));
      String inputLine, total="";

      while((inputLine = in.readLine()) != null) {
         total += inputLine+"\n";
      }
      in.close();
      return total;
   }

   private void printData(Map m) {
      System.out.println("Current weather in Lappeenranta:");
      System.out.println("Total weather: " + m.get("Weather"));
      System.out.println("Temperature: " + m.get("Temperature") + " C");
      System.out.println("Humidity: " + m.get("Humidity") + "%");
      System.out.println("Wind: " + m.get("Wind"));
      System.out.println("Clouds: " + m.get("Clouds"));
   }

   public static void main(String[] args) throws Exception {
      WeatherData data = new WeatherData();
      WeatherXML xml = new WeatherXML(data.gatherData());
      data.printData(xml.getMap());
   }
}
```

Tuloste

```
run:
```

```
Current weather in Lappeenranta:
Total weather: light rain
Temperature: 1.0 C
Humidity: 82%
Wind: Moderate breeze
Clouds: overcast clouds
BUILD SUCCESSFUL (total time: 0 seconds)
```

Olipas helppoa, tosin luokasta näyttäisi puuttuvan muutamia toiminnallisuuksia. Esimerkissä on käytetty laajempaa taustaluokkaa, jonka avulla data ensin muutetaan XML-muotoon ja kerätään sekä parsitaan kyseisestä dokumentista. Luokassa luodaan Document-olio ja Map-olio, joiden avulla data muotoillaan oikeaan muotoon ja tiedot tallennetaan helposti käytettävään muotoon. Map-tietorakenteesta keskustelimme syvällisemmin jo oppaan alkupuolella, joten lukijan on hyvä kerrata sen toiminta. Document-oliolla viitataan olioon, joka on XML-muotoinen ja antaa ohjelmoijalle kyvyn parsia eli muotoilla sen sisältämää tietoa valmiiksi rakennetuilla metodeilla.

Esimerkki 9.4. Säädatan parsiminen

```
import java.io.IOException;
import java.io.StringReader;
import java.util.HashMap;
import java.util.Map;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

import org.xml.sax.InputSource;
import org.xml.sax.SAXException;

public class WeatherXML {

   private Document doc;
   private Map m;

   public WeatherXML(String total) {
      try {
         DocumentBuilderFactory dbFactory =
                        DocumentBuilderFactory.newInstance();
         DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
         doc = dBuilder.parse(new InputSource(new StringReader(total)));
         doc.getDocumentElement().normalize();
      } catch (ParserConfigurationException e) {
         System.err.println("Caught ParserConfigurationException!");
      } catch (IOException e) {
         System.err.println("Caught IOException!");
```

```
        } catch (SAXException e) {
            System.err.println("Caught SAXException!");
        }
    }

    public Map getMap() {
        m = new HashMap();
        getCurrentData();
        return m;
    }

    protected void getCurrentData() {
        NodeList nodes = doc.getElementsByTagName("current");
        for(int i = 0; i < nodes.getLength(); i++) {
            Node node = nodes.item(i);
            Element e = (Element) node;
            m.put("Weather", getValue("weather", e, "value"));
            m.put("Temperature", K2C(getValue("temperature", e, "value")));
            m.put("Humidity", getValue("humidity", e, "value"));
            m.put("Wind", getValue("speed", e, "name"));
            m.put("Clouds", getValue("clouds", e, "name"));
        }
    }

    private double K2C(String k) {
        return Double.parseDouble(k) - 273.15;
    }

    private String getValue(String tag, Element e, String attr) {
        return ((Element)
                (e.getElementsByTagName(tag).item(0))).getAttribute(attr);
    }
}
```

Kuten esimerkissä voidaan huomata, jo pienen askareen suorittamiseen tarvitaan Javassa useampia paketteja suorittamaan vaadittuja tehtäviä. Esimerkki käyttää hyödykseen jo aikaisemmin oppaassa esiteltyä Map-tietorakennetta, johon on mahdollista tallentaa avain/arvo-pareja niin, että avaimella päästään aina käsiksi siihen liittyvään arvoon. Esimerkissä Webistä ladattu String-muotoinen data muutetaan XML-muotoon, josta dataa voidaan lukea W3C:n tarjoamien metodien avulla. Tämä karkaa jo laajuudessaan hieman tämän oppaan asian ulkopuolelle ja lukijan ei ole välttämätöntä ymmärtää eksaktisti, miksi oheisia muutoksia tehdään, mutta ne tulevat kyllä vastaan tulevaisuudessa.

Huomioitavaa XML-muotoisessa datassa on se, että se tavallisesti käyttää jotakin standardoitua päivämäärän muotoa, jolloin päivämäärän muokkaamiseen tarvitaan omat työkalut. Seuraavassa lyhyt esimerkki, jossa on käytetty ISO Local Date and Time mukaista muunnosta.

Esimerkki 9.5. Päivämäärän tulostaminen

```
public static void main(String[] args) {
    String input = "2014-06-16T10:35:00";
    try {
        DateTimeFormatter formatter = DateTimeFormatter.ISO_LOCAL_DATE_TIME;
        LocalDate date = LocalDate.parse(input, formatter);
        LocalTime time = LocalTime.parse(input, formatter);
        System.out.println(date);
        System.out.println(time);
    }
    catch (DateTimeParseException exc) {
        System.out.println("Päivämäärää ei voitu muokata!");
    }
}
```

Tuloste

```
2014-06-16
10:35
```

Lisää päivämäärämuunnoksista voi lukea Oraclen tarjoamasta materiaalista:

http://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html

10. Lähteet

Teemu Saarelainen, Java-ohjelmointi-kurssin tietovirrat-luento, 2006,
URL: www2.kyamk.fi/~atesa/java/Java2006_Streams.pdf

Pilone D, Pitman N, *UML 2.0 in a nutshell*, 2005, O'Reilly Media

Eckel B., *Thinking in Java*, 4th edition, 2008, Prentice Hall

11. Liite 1: Asennusopas

Oletuksena kurssilla käytetään Ubuntu 12.04 LTS tai 14.04 LTS ympäristöjä

Valitettavasti Ubuntu ei tarjoa pakettivalikoimassaan oikeita versioita työkaluista, joten joudumme asentamaan ne käsin. Yliopiston Linux-luokassa on asennettuna tarvittavat ohjelmistot. Seuraavassa lyhyet ohjeet mitä tarvitsee asentaa.

Java 8u5 ja NetBeans 8:

http://www.oracle.com/technetwork/java/javase/downloads/jdk-netbeans-jsp-142931.html

Asennusohje Windowsille, Macille ja Linuxille löytyy osoitteesta:

http://www.oracle.com/technetwork/java/javase/downloads/install-jdk6-22nb691-177131.html

Huomaa, että paketti lataa sekä Netbeans 8:n ja uusimman Javan (tässä tapauksessa 8), eli jos koneellasi on jo uusin mahdollinen Java, voit ladata ainoastaan Netbeans 8:n sen omilta sivuilta: https://netbeans.org/downloads/

JavaFX Scene Builder 2.0:

http://www.oracle.com/technetwork/java/javase/downloads/sb2download-2177776.html

Asennusohje:

http://docs.oracle.com/javase/8/scene-builder-2/installation-guide/index.html

Vastoin kuin Javan ja Netbeansin asennuksessa, Scene Builder tarjotaan valmiina .deb-pakettina, joka on mahdollista asentaa suoraan Ubuntun pakettimanagerin avulla.

UML-piirto-ohjelmistoja:

Umbrello: Löytyy pakettienhallinnasta

yEd (jolla esimerkit on tehty): http://www.yworks.com/en/products_yed_download.html

Dia: Löytyy pakettienhallinnasta

yEd on mahdollista asentaa samalla tavoin kuin Netbeans eli komentoriviltä. Lataa siis käyttöjärjestelmällesi sopiva versio ja navigoi terminaalin avulla kansioon, johon latasit tiedoston. Tämän jälkeen muuta kansiossa olevan asennustiedoston oikeudet niin, että voit suorittaa tiedoston (chmod +x). Kun sinulla on ajo-oikeus tiedostoon, voit ajaa sen käyttämällä sen käskyllä ./<tiedoston nimi>.