EETU JUHANI ASIKAINEN

OHJELMOINTIOPAS LAPPEENRANNAN-

LAHDEN TEKNILLISEN YLIOPISTON LUT:N OLIO-OHJELMOINTIKURSSI

LISÄÄ JULKAISIJA?

PUBLISHED BY LISÄÄ JULKAISIJA?

TUFTE-LATEX.GOOGLECODE.COM

First printing, January 2022

Contents

Oppaasta ja sen käytöstä

Esipuhe

Tämä opas on kirjoitettu osaksi Lappeenrannan-Lahden teknillisen yliopiston LUT:n olio-ohjelmoinnin perusteiden kurssin kurssimateriaaleja. Opas käy läpi koko kurssin teoriasisällön lukuunottamatta Android-API:n käyttöä. Opas on tarkoitettu käytettäväksi osana kurssikokonaisuutta, mutta se on suunniteltu niin, että se tarjoaa laadukkaan perehdytyksen oliopohjaiseen ajatteluun, suunnitteluun ja ohjelmointiin myös itsenäisenä kokonaisuutena.

Oppaan tavoitteena on tarjota opiskelijoille helppokäyttöinen, laadukas ja suomenkielinen resurssi kurssin käsittelemien aihealueiden opiskeluun. Opas olettaa käyttäjän hallitsevan proseduraalisen ohjelmoinnin perusteet, jotka käydään läpi esimerkiksi Lappeenrannan-Lahden teknillisen yliopiston LUT:n "Ohjelmoinnin Perusteet" -kurssilla.

Oppaan rakenne

Opas on jaettu yhteentoista sisältölukuun. Luvut on numeroitu välillä 0-11 niin, että luvun numero vastaa viikkoa, jolla luvun sisältöä käsitellään kurssilla ja luku 0 käsittelee itse opasta ja sen käyttöä. Näin opasta on helppo seurata kurssin edetessä. Viikolla 7 käsitellään vain Androidin API:a, joten oppaassa ei ole lukua 7. Android API:n käsittely on jätetty oppaasta pois, koska API muuttuu ja kehittyy niin nopeasti, että sitä käsittelevät osat todennäköisesti vanhenisivat vuodessa tai kahdessa.

Termistö ja kieli oppaassa

Oppaan pääasiallisena kielenä koodiesimerkkejä lukuunottamatta toimii suomi muun kurssimateriaalin kielivalinnan mukaan. Oppaan käyttämä termistö on tämän vuoksi käännetty Javan avainsanoja lukuunottamatta suomeksi. Olisi kuitenkaan turha kiistää englannin ehdotonta ylivaltaa tietojenkäsittelytieteen pääkielenä. Tämän vuoksi oppaan sanastosta löytyy jokaisen termin selityksen lisäksi termin englanninkielinen vastine.

OPAS PYRKII korostamaan tärkeää termistöä sanaston opettelun helpottamiseksi. Jokainen tärkeä termi on kerätty oppaan sanastoon lyhyen selityksen ja termin englanninkielisen vastineen kera. Sanastosta löytyvät termit on myös korostettu ensimmäisessä esiintymisessään. Sanastotermeillä tämä korostus näkyy kursivoituna kirjoitusasuna ja termin perässä esiintyvänä sulkuihin suljettuna englanninkielisenä vastineena termille. Vastine on niin ikään kursivoitu. Koko korostettu kirjoitusasu termille näyttää siis tältä: *esimerkki (example)*

OPAAN TEORIATEKSTI sisältää sanastosta löytyvien konseptuaalisesti tärkeiden termien lisäksi myös Javan avainsanoja. Nämä on erotettu muusta korostetusta termistöstä alleviivauksilla ja kääntämättömyydellä. Ensimmäisellä esiintymiskerrallaan Javan avainsanat ovat lisäksi kursivoituja. Ensimmäistä kertaa esiintyvä Javan avainsana näyttää tältä: *example* ja uudestaan mainittu tältä: example.

Koodiesimerkit oppaassa

OPAS TARJOAA kattavat koodiesimerkit kaikista siinä käsiteltävistä aihealueista. Koodiesimerkit ovat lyhyitä, toimivia, koodinpätkiä ja ne sijaitsevat heti esitellyn konseptin yhteydessä. Kaikki oppaan koodiesimerkit on koottu yhteen GitHub-repositorioon osoitteessa `https://github.com/EddieTheCubeHead/OhjelmointiopasEsimerkit` kansion src-alta viikoittain lajiteltuina.

KOODIESIMERKIT ON KIRJOITETTU Javan virallisen tyylioppaan suosittelemien ohjelinjojen mukaisesti. Kaikki muuttujat, luokat ja metodit on nimetty englanniksi, mutta koodi on kommentoitu suomeksi. Kommentointi ei noudata hyvää ohjelmointityyliä vaan kommentit ovat koodin opetusmaisen luonteen vuoksi monisanaisempia ja yleisempiä kuin suosituksissa.

Oliopohjainen ajattelu

Abstraktio laadukkaan koodin pohjana

Oliopohjainen ohjelmointi on ohjelmointiparadigma, joka on kehitetty vastaamaan ohjelmistotuotannon peruskysymykseen: kuinka kirjoittaa ymmärrettävää ja ylläpidettävää koodia helposti?

Yksi helpoimmista ja yleisimmistä tavoista selkeyttää koodikantaa on *abstraktio (abstraction)*: yleisten toimintojen ja käskysarjojen eristäminen ennalta määriteltyihin koodipaloihin. Nämä palat tunnetaan nimellä *funktio (function)*. Tällä tavalla jaettuja käskysarjoja voidaan uusiokäyttää ja ohjelman muokkaaminen helpottuu, kun käskysarjan päivittäminen tapahtuu vain yhdessä keskitetyssä paikassa.

Käskysarjat ovat kuitenkin vain puolikas toimivasta ohjelmistosta. Toinen ja aivan yhtä tärkeä puoli on data, jota käskysarjoilla käsitellään. Myös tämän datan abstraktio on mahdollista proseduraalisen ohjelmoinnin puitteissa muokattavilla tietorakenteilla. Tällaisestä hyviä esimerkkejä ovat esimerkiksi Pythonin class-avainsanalla määritellyt luokat, jos niitä käytetään vain datan ryhmittämiseen tai C-kielen *tietue (struct)*.

Nämä abstraktiotyypit ovat kuitenkin erotettuja toisitaan, eivätkä huomioi käskysarjojen ja datan yhteyttä. Lähes poikkeuksetta tiettyjä funktioita kutsutaan koodissa monessa paikkassa parametreinään toistuvasti sama tietorakenne. Eikö siis olisi järkevää yhdistää datan ja sitä käsittelevien funktioiden sijainti koodissa?

Oliopohjaisen ajattelun perusteet

Olio-ohjelmoinnissa ohjelmisto koostuu kokonaan keskenään kommunikoivista yksiköistä. Näille yksiköille vakiintunut kutsumanimi on *olio (object)*. Oliot koostuvat datasta ja sitä manipuloivista käskysarjoista, jotka tunnetaan nimellä *metodi (method)*. Metodi on oliopohjaisen ohjelmoinnin nimi olioon sidotulle funktiolle, termiä funktio ei juuri käytetä.

Ohjelmisto luo kaikki käyttämänsä oliot ohjelmoijan tekemien

muottien pohjalta. Tälläinen muotti, eli *luokka (class)* voi luoda itsestään yleensä rajattoman määrän olioita ajon aikana. Luokan ja olion ero on yksi oliopohjaisen ohjelmoinnin tärkeimmistä käsitteistä ja sen ymmärtäminen alusta lähtien on kriittistä oliopohjaisen ohjelmoinnin opiskelussa.

JOKAINEN OLIO on jonkin luokan *instanssi (instance)*. Saman luokan eri instanssit omistavat identtiset metodit, mutta instansseihin säilötty data on uniikki jokaiselle instansille ja instanssin metodit käsittelevät tätä uniikkia dataa. Luokka määrittää nämä metodit ja kertoo minkä tyyppiset datakentät jokaiselta luokan instanssilta löytyvät. Tästä lähtien opas käyttää termiä "luokka" viitatessaan ohjelmoijan määrittämään olio muottiin eli puhuttaessa suunnitteluperiaatteista ja koodin kirjoittamisesta ja termiä "olio" vain puhuessaan selkeästi koodissa sijaitsevasta luokan ilmentymästä.

TÄLLÄINEN JAOTTELU järjestää koodia proseduraalista koodia luonnollisempiin osiin. Tästä syystä oliopohjaista ohjelmointia esitellessä mainitaan usein sen kyky luoda koodin sisällä järjestelmiä, jotka muistuttavat todellisen maailman rakenteita. Vaikka tämä pitää paikkansa on hyödyllistä ymmärtää alusta lähtien, ettei oliopohjaisen ohjelmoinnin tavoitteena ole luoda yksi yhteen oikean maailman kanssa samoin toimivaa järjestelmää. Sen sijaan oikea oliopohjaisen ohjelmoinnin tavoite on luoda järjestelmä, joka koostuu pienistä tarkasti toisistaan erotelluista osista, jotka kommunikoivat keskenään vain selkeästi määriteltyjä rajapintoja pitkin.

KÄYTÄNNÖSSÄ LAADUKASTA oliopohjaista ohjelmistoa voidaan täten verrata esimerkiksi moderniin modulaarisesti suunniteltuun autoon. Siinä missä auton moottori, pyörät tai verhoilu ovat muokattavissa asiakkaan toiveiden ja budjetin mukaan, on laadukkaan oliopohjaisen ohjelmiston osien, kuten vaikka autorisaatiomoduulin tai tietokantayhteyden laajentaminen ja muokkaaminen mahdollista koskematta muuhun ohjelmistoon.

TÄMÄ KUULOSTAA kunnianhimoiselta ja monimutkaiselta tavoitteelta, joten on tärkeää edetä sopivan pienissä paloissa. Samoin kuin laadukas oliopohjainen ohjelmisto koostuu pienistä paloista, kasataan ohjelmiston tekemiseen vaadittu tieto pieni pala kerrallaan. Tässä kappaleessa esiteltyjä termejä ja käsitteitä tullaan syventämään oppaan tulevissa kappaleissa yksitellen, joten vaikka tiedon määrä voi tuntua nyt kohtuuttomalta ei kannata huolestua liikaa.

Koodin laadun mittaamisesta

JOTTA OHJELMISTOJEN modulaarisuus saadaan toivotulle tasolle on tärkeää miettiä mitä metodeja tai dataa luokka paljastaa muulle ohjelmistolle. Oliopohjaiset kielet sisältävätkin yleensä jonkin keinon rajoit-

taa datakenttien ja metodien näkyvyyttä luokan ulkopuolelle. Tämä näkyvyyden rajaaminen ja tarkkojen vastuiden määrittäminen tunnetaan nimellä *enkapsulaatio (encapsulation)*.

LUOKAN PALJASTAMAT datakentät ja metodit kertovat ulkopuoliselle ohjelmistolle, mitä luokasta luodut oliot voivat tehdä ja mitä niiltä voi odottaa. Tämä muodostaa sopimuksen luokan ja muiden luokan paljastamia metodeja tai datakenttiä käyttävien luokkien välillä. Tätä sopimusta kutsutaan nimellä *rajapinta (interface)*. Rajapinnan abstraktia käsitettä ei pidä sekoittaa Javan samannimiseen *interface*-avainsanaan, joka esitellään myöhemmin oppaan osiossa Rajapinta.

YKSI OLIO-OHJELMOINNIN NYRKKISÄÄNNÖISTÄ on pitää huolta, että jokainen luokka suorittaa yhden ja vain yhden tehtävän. Tämä tarkoittaa luokan rajapinnan pitämistä minimaalisena, eli korkeaa enkapsulaatiota. Tämän lisäksi ohjelmiston peruslaatua voidaan arvioida käsitteillä *koheesio (cohesion)* ja *pariutuminen (coupling)*.

KOHEESIO VOIDAAN ARVIOIDA luokkakohtaisesti luokan metodien ja datakenttien yhteistoiminnan perusteella. Mitä tiiviimmin metodit ja datakentät ovat yhteistyössä ja mitä vähemmän luokassa on metodeja ja datakenttiä, jotka eivät ole vuorovaikutuksissa muun luokan kanssa, sitä korkeampi koheesio luokalla on. Korkea koheesio kulkee siis pitkälti käsi kädessä tiukan enkapsulaation kanssa. Mitä vähemmän luokka paljastaa itsestään muulle ohjelmistolle, sitä tiukemmin sen metodit ja datakentät yleensä komminkoivat keskenään.

PARIUTUMINEN TARKOITTAA ohjelmiston luokkien riippuvuutta toisista ohjelmiston luokista. Siitä puhutaan usein asteikolla *löysä (loose)*-*tiukka (tight)*. Löysästi pariutineessa ohjelmistossa jokainen luokka on riippuvainen vain muutamasta muusta luokasta ja riippuvaisuudet muistuttavat usein enemmän ketjua tai puuta kuin verkkoa.

KOHEESIO JA PARIUTUMINEN kulkevat usein käsi kädessä: Korkea koheesio johtaa löysään pariutumiseen ja matala koheesio tiukkaan pariutumiseen. Enkapsulaatio mukailee yleensä koheesiota niin että korkeasta koheesiosta seuraa korkea enkapsulaatio ja toisin päin. Kaikki kolme ovat tärkeitä mittareita puhuttaessa oliopohjaisen ohjelmiston laadusta. Vaikka oppaan tärkein tehtävä onkin opettaa lukija koodaamaan Javalla, eikä niinkään koodaamaan huipputason koodia, on näiden käsitteiden olemassaolo ja merkitys hyvä tiedostaa. Tämä auttaa ymmärtääkseen oliopohjaisten kielten suunnittelua ja etuja suhteessa muihin ohjelmointikieliin ja saattaa myös parantaa tuotetun koodin laatua huomaamattomasti.

Javan perusteet

Java ohjelmointikielenä

JAVA on staattisesti tyypitetty korkean tason oliopohjainen ohjelmointikieli, joka pyrkii standardisoimaan ohjelmien kirjoittamisen alustasta riippumatta. Java-ohjelmat käännetään bittikoodiksi, joka ajetaan Java-virtuaalikoneella siten, että tismalleen sama koodi voidaan ajaa jokaisella alustalla, jolle virtuaalikone voidaan asentaa.

TÄSTÄ SYYSTÄ Java toimii tismalleen samoin kaikilla käyttöjärjestelmillä. Tämä yhdessä Javan helposti opittavan syntaksin ja oliopohjaiseen suunnitteluun pakottavan luonteen ohella on tehnyt Javasta yhden käytetyimmistä ohjelmointikielistä teollisessa ohjelmoinnissa.

JAVAN SYNTAKSI on saanut paljon vaikutteita C-kielestä. Muun muassa kaarisulkeiden käyttö koodin osien erotteluun ja suurin osa ohjausrakenteiden nimistä on suoraan C-kielestä kopioitua.

Javan syntaksin alkeet

Kommentointi

SUURIN OSA ohjelmointikielistä tukee ainakin yhtä kommentointisyntaksia. Javassa on kaksi pääasiallista tapaa merkata koodissa oleva teksti kommentiksi. Yhden rivin kommentti voidaan aloittaa "//"-merkkiparilla, kun taas monen rivin kommentti suljetaan "/*"- ja "*/"-merkkien väliin. Seuraava esimerkki esittää kommentointisyntaksin vielä tarkemmin. Älä välitä vielä itse koodista, sen tehtävä on vain selventää miten kommenttien sijoittelu vaikuttaa koodin toimintaan.

Koodiesimerkki 1: week2/basicexamples: Kommentointi Javassa

```
// Javassa on kaksi tapaa kommentoida kirjoitettua koodia. Ensimmäinen on aloittaa kommentti
// kahdella kauttaviivalla. Tämä merkitsee kääntäjälle rivin loppuosan olevan kommenttia, jollon
// kääntäjä jättää sen huomiotta Huomioi, että tämä toimii vaikka rivillä olisi ollut aikaisemmin
// koodia, eikä riko toiselle riville jatkuvia rakenteita. Tämän oppaan kommentit käyttävät
// pääasiassa tätä tyyliä. Välin jättäminen kaksoiskauttaviivan ja sitä seuraavan kommentin
// väliin ei ole pakollista, mutta sitä pidetään monessa tyylioppaassa oikeaoppisena tyylinä.
```

```
//Myös tämä on siis toimiva kommentti

package week2.basicexamples; // Pakettimääritelmä toimii vaikka sitä seuraa kommentti

class Commenting { // Luokan määritelmän jälkeen laitettu kommentti toimii myös

    /* Toinen tapa kommentoida on ympäröidä kommentti kauttaviivoilla ja tähdillä tähän tapaan */

    /* Tälläinen kommentti voi ulottua monelle riville.
     * Monesti jatketut kommenttirivit aloitetaan tähdellä.
     * Tämä erottaa kommentin selkeämmin muusta koodista.
     */

    /* Mutta esimerkiksi myös tälläinen kommentti toimii,
    joskin se ei näytä kovin mukavalta ja on hankalampi lukea.
     */

    public static void main(String[] args){

        // /*-merkillä aloitetut kommentit voivat sijaita myös koodirivin alussa tai keskellä.
        // Tällainen asettelu toimii kunhan kommentit eivät katkaise avainsanoja.
        // Tämä ei ole suositeltavaa viimeistellyssä koodissa, mutta saatta olla hyödyllistä
        // kehitysvaiheessa.

        /* Tämä toimii */ System.out.println(/* Tämäkin toimii */"This works");
    }
}
```

Primitiiviset tietotyypit ja String

YKSIÄ TÄRKEIMMISTÄ koodin osista ovat ehdottomasti *muuttujat (variable)*. Ne ovat koodissa määriteltyjä tietokenttiä, jotka kykenevät säilömään joko jonkin luokan instanssin, tai kokonaislukumuodossa esitetyn simppelin datayksikön. Koska Java on *staattisesti tyypitetty kieli (statically typed language)*, täytyy siinä jokaisen muuttujan tietotyyppi määritellä muuttujan määrittelyn yhteydessä. Muuttujan tyyppi voi olla joko jokin käyttäjän määrittelemä luokka, jokin Javan standardikirjaston luokka, tai jokin *primitiivinen tietotyyppi (primitive data type)*.

JAVA SISÄLTÄÄ yhteensä kahdeksan primitiivistiä tietotyyppiä. Nämä määritellään pienellä alkukirjaimella kirjoitetulla tyypin nimellä. Seuraava taulukko ja koodiesimerkki sisältävät kaikki primitiivisten tietotyyppien nimet, arvojoukot, oletusarvot ja määrittelytavan koodissa.

Avainsana	Oletusarvo	Arvojoukko
bool	false	true ja false
byte	0	-2^7:stä $2^7 - 1$:een
short	0	-2^{15}:stä $2^{15} - 1$:een
int	0	-2^{31}:stä $2^{31} - 1$:een
long	0L	-2^{63}:stä $2^{63} - 1$:een
float	0.0f	noin $1.4e-45$:stä noin $3.4e+38$:aan
double	0.0d	noin $4.9e-324d$:stä noin $1.8e+308d$:hen
char	0	0:sta $2^{16} - 1$:een

Koodiesimerkki 2: week2/basicexamples: Primitiiviset tietotyypit Javassa

```
package week2.basicexamples;

// Luokan luonnin notaatio käydään läpi myöhemmin tässä luvussa
class DataTypes {

    // Primitiivisten tietotyyppien luominen

    // boolean määritetäänd "true" tai "false" avainsanalla
    boolean booleanValue = true;
    // byte, short ja int voidaan kaikki määrittää normaaleilla lukuarvoilla
    byte maxByteValue = 127;
    short maxShortValue = 32767;
    int maxIntValue = 2147483647;
    // Huomaa long-tietotyypin määrittämiseen vaadittu L-pääte
    long maxLongValue = 9223372036854775807L;
    // float ja double ilmoitetaan f- ja d-päätteillä
    float floatValue = 1.1f;
    double doubleValue = 3.14159265358979323846264338327950288419716939937510582097494459 2d;
    // char-arvot määritellään ympäröimällä yksittäinen merkki tai unicode-koodi yksittäisillä
    // lainausmerkeillä (')
    char charValue = 'a';
    char unicodedCharValue = '\u0123';
}
```

Näiden tietotyyppien lisäksi Javasta löytyy monta valmiiksi määriteltyä luokkaa datan säilömistä varten. Näistä tärkein kielen opettelua aloitettaessa on ehdottomasti merkkijonoja säilövä *String*-luokka. Tämä luokka mahdollistaa dynaamisten merkkijonojen helpon luomisen ja manipulaation. String-luokka on voimakas työkalu ja sen toimintaan kannattaa kiinnittää huomiota Javan opiskelun alussa.

String-tyyppisen muuttujan arvo kerrotaan ohjelmalle antamalla String-tyypin muuttujalle toivottu merkkijono suljettuna normaalien lainausmerkkien (") sisään. Yksittäiset hipsut (') eivät toimi, koska niitä käytetään char-tyypin muuttujien määrittämiseen.

Koodiesimerkki 3: week2/basicexamples: String-tyypin muuttujan määrittäminen

```
package week2.basicexamples;

class StringUsage {
    // Huomaa, että char-muttuja suljetaan yksittäisiin lainausmerkkeihin (') ja String-muuttuja
    // tuplattuihin launausmerkkeihin (")
    String exampleString = "Example string";
}
```

Toimivan ohjelman perusteet

Oliopohjaisen ohjelman ajaminen: main-metodi

LUOKAT OVAT Javan ydin ja kaiken koodin rakennuspala. Kaikki koodi Javassa on suljettava luokan sisään, niin myös ohjelman ajon aloittava koodi. Tästä syystä jokaisessa Java-ohjelmistossa on yksi erikseen määritelty pääluokka, joka sisältää main-metodin. Tämä metodi määritellään avainsanoilla public static void ja sen *signatuuri (signature)* on main(String[] args). Void-avainsana selitetään myöhemmin tässä kappaleessa osiossa Metodin määritteleminen ja kutsuminen ja static-avainsana kappaleessa Luokkamuuttujat, luokkafunktiot ja static. Toistaiseksi riittää sisällyttää nämä avainsanat annetussa järjestyksessä main-metodin määritelmään. Pääluokan ei tarvitse olla ainoa main-metodin sisältävä luokka, mutta vain yhden ennalta määritellyn luokan main-metodi ajetaan ohjelman käynnistyksen yhteydessä.

Luokan luomisen syntaksi

LUOKKA LUODAAN Javassa *class* avainsanalla. Avainsana tarkistaa edestään myös näkyvyysmääreen (käsitellään alakappaleessa Näkyvyysmääreet). Yleisin tapa luoda luokka on näkyvyysmääreellä public. Koko määritelmä koostuu näkyvyysmääreestä, class -avainsanasta ja luokan nimestä, jota seuraa luokan käyttäytymisen määrittävä koodi hakasulkeisiin suljettuna. Seuraavassa koodiesimerkissä on kommentoitu auki yksinkertaisen pääluokan luonti (huomaa pääluokan nimeämisen vapaus) ja sen sisään "Hello world!" tulostavan main-funktion lisääminen.

Tulostaminen Javassa

TEKSTIN TULOSTAMINEN komentoikkunaan on varmasti tuttua aiemmasta ohjelmointikokemuksesta. Javassa tulostukseen käytetään yleensä standardikirjaston *System*-luokan out-muuttujassa tallennettuna olevaa PrintStream-luokan instanssia, joka edustaa oletustulostevirtaa.

Jos et ymmärtänyt edellistä lausetta, se ei haittaa. Tärkeintä on tietää, että Javassa helpoin tapa tulostaa on kutsua "System.out" - sijainnista sopivaa tulostusmetodia. Opas käyttää suurimmaksi osaksi "println"-metodia, joka ottaa merkkijonon ja tulostaa sen, sekä rivinvaihdon. On huomioitavaa, että println, samoin kuin muutkin System.out:sta löytyvät tulostusmetodit ovat todella monipuolisia ja tukevat monenlaisia tapoja tulostaa toivottua tekstiä. Toistaiseksi nämä ominaisuudet eivät kuitenkaan ole ajankohtaisia ja opas käyttää "System.out.println" metodia tulostukseen.

Koodiesimerkki 4: week2/basicexamples: Yksinkertaisen pääluokan ja main-funktion luominen Javassa

```
package week2.basicexamples;

// public-avainsana kertoo luokan olevan koko muun ohjelmiston nähtävissä
//
// class-avainsana aloittaa luokan määritelmän. Sitä seuraa luokan nimi ja aaltosulkeisiin suljettu
// luokan runko. Pääluokan nimeä ei ole etukäteen määritelty ja onkin suositeltavaa nimetä se
// kuvailevasti, esimerkiksi koko ohjelman nimen mukaisesti.
class HelloWorld {

    // static-avainsana käsitellään oppaassa myöhemmin. Toistaiseksi riittää
    // että tiedostaa main-metodin vaadittavan olevan muotoa "public static void"
    public static void main(String[] args){

        // Kutsu println-funktioon tulostaa annetun merkkijonon ja rivinvaihdon
        System.out.println("Hello world!");
    }
}
```

Metodin määritteleminen ja kutsuminen

METODIN MÄÄRITELMÄ koostuu Javassa metodin paluuarvon tyypistä, metodin nimestä ja sitä seuraavasta sulkuihin suljetusta parametrien määritelmästä ja lopulta kaarisulkuihin suljetusta metodin koodista. Metodin parametrit vaativat pythonista poiketen parametrin nimen määritelmän lisäksi myös parametrin tyypin määrittämistä. Vastaavasti metodia kutsuttaessa on argumenttien, jolla metodia kutsutaan oltava metodin yhteydessä määriteltyä tyyppiä. Jos metodi ei palauta mitään, määritellään sen paluuarvoksi *void*.

METODIEN NIMEÄMISESSÄ suositellaan Javassa vahvasti niin sanottua camel case- nimeämistapaa. Tavassa kaikki nimen sanat kirjoitetaan yhteen niin, että nimi aloitetaan pienellä kirjaimella ja seuraavat sanat aloitetaan isolla alukirjaimella. Täten esimerkiksi "parse user name"-metodin nimi on muotoa "parseUserName". On myös vahvasti suositeltua käyttää verbiä metodin nimessä. Tämä helpottaa metodien ja muuttujien erottamista ja pakottaa myös ajattele-

maan koodin laatua: hyvä metodi tekee vain yhden asian, joten jos nimeäminen simppelissä verbimuodossa ei onnistu, olisi ehkä syytä jakaa metodi useampaan pienempään metodiin. Metodin palautusarvo määritellään *return*-avainsanalla. Avainsana lopettaa metodin ajamisen ja palauttaa avainsanasta välilyönnillä erotetun arvon.

METODIA KUTSUTAAN Javassa pistenotaatiolla metodin omistavasta oliosta. Tämä olio on yleensä tallennettuna muuttujaan jonkin toisen luokan sisällä. Esimerkiksi jos olio on tallennettu muuttujaan "object" ja omistaa metodin "doStuff(int number)", kutsuttaisiin metodi notaatiolla "object.doStuff(322)". Jos metodille on määritelty paluuarvon tyyppi, kutsu palauttaa tämän tyypin muuttujan, jota voidaan käyttää koodissa.

Koodiesimerkki 5: week2/methodexample: Metodin luominen Javassa

```
package week2.methodexample;

public class MethodClass {

    String combineStrings(String first, String second) {
        return first + second;
    }
}
```

Koodiesimerkki 6: week2/methodexample: Metodin kutsuminen Javassa

```
package week2.methodexample;

public class MethodExample {

    public static void main(String[] args) {

        // Luodaan uusi child-olio, josta metodia kutsutaan
        MethodClass child = new MethodClass();

        // Metodia kutsuttaessa on pidettävä huolta, että metodille annetaan oikean tyypin
        // argumentit. Kääntäjä ei suostu kääntämään koodia, jos metodia yritetään kutsua
        // väärän tyyppisillä argumenteilla.
        String combinedString = child.combineStrings("Hello ", "world!");

        // Tulostaa "Hello world!" ja rivinvaihdon.
        System.out.println(combinedString);
    }
}
```

If-lause ja else-lause

OHJELMOINTI ON pohjimmiltaan logiikkaa. Ohjelman ajon muokkaaminen olosuheiden mukaan tekee ohjelmasta älykkään. Yksinkertaisin tapa muokata ohjelman toimintaa on *if*-lause ja sen kanssa toimiva *else*-lause. Lauseet mahdollistavat ennalta määriteltyjen koodin osien ajamisen annetun lauseen totuusarvon mukaan.

IF-LAUSE alkaa if-avainsanalla, jota seuraa suluilla ympäröity lause, joka saa totuusarvon true tai false. Tätä ehtolausetta seuraa kaarisulkuihin ympäröitynä itse if-lauseen runko, eli koodi, joka ajetaan jos ehtolause on tosi.

ELSE-LAUSE on mahdollinen jatko if-lauseelle ja sijaitsee aina if-lauseen ehdollisesti ajettavan koodiosan jälkeen. Lause alkaa else-avainsanalla. Tämän jälkeen on mahdollista kirjoittaa uusi if-lause, tai aloittaa suoraan ehdollisen koodin määrittely. If- else -lauseita voi tarvittaessa ketjuttaa niin monta kuin tarvitaan, joskin pitkät if-else -ketjut ovat yleensä merkki huonosta arkkitehtuurista ohjelmistossa.

SEURAAVA KOODIESIMERKKI esittelee if- ja else-lauseiden käytön perusteet. Huomioi, että normaalisti ehtolauseiden arvoa ei tiedetä ennen ohjelman ajoa, vaan se muuttuu dynaamisesti.

Koodiesimerkki 7: week2/basicexamples: Esimerkki if- ja else-lauseiden käytöstä

```
package week2.basicexamples;

public class IfElse {

    // Tästä ei tarvitse välittää. Ainut tämän rivin tarkoitus on estää IDE:tä, jolla esimerkkikoodi
    // on kirjoitetta huomauttamasta if-lauseista, joiden sisällä olevan ehdon totuusarvo tiedetään
    // koodia kääntäessä.
    @SuppressWarnings("ConstantConditions")
    public static void main(String[] strings) {
        int number = 3;

        // If-lause voi esiintyä yksin, ilman else-lausetta. On huomioitava, että esimerkin vuoksi
        // tässä esimerkissä ehtolauseiden totuusarvo tiedetään jo ennalta, mutta normaalisti
        // totuusarvo muuttuisi ajon aikana.
        if (number < 0) {
            System.out.println("Number is negative.");
        }

        if (number < 3) {
            System.out.println("Number is smaller than 3.");
        // Else-lauseen yhdistäminen uuteen if-lauseeseen tuottaa niin sanotun "else if"-lauseen.
        // Tätä ei esitellä sen tarkemmin oppaassa, koska se ei ole aito avainsana, toisin kun
        // pythonin "elif", vaan ainoastaan else- ja if-lauseiden toimintatavasta johtuva Javan
        // ominaisuus.
        } else if (number < 5) {
```

```
            System.out.println("Number is bigger than 2 but smaller than 5.");
        // Viimeisen else-lauseen ehdollinen koodi ajetaan aina, jos minkään ketjun edellisen
        // lauseen ehdollista koodia ei ajettu.
        } else {
            System.out.println("Number is bigger than 4.");
        }
    }
}
```

Lisää Javan konsepteja

Rakentajat, luokan instanssin luominen ja this

OLIOIDEN HALLINTA on Javalla ohejlmoinnin ytimessä. Koodissa on luotava uusia instansseja sekä Javan standardikirjaston luokista, että käyttäjän itse määrittelemistä luokista. Uusi instanssi luokasta luodaan *new*-avainsanalla. Avainsanaa seuraa luotavan luokan nimi ja luomisen parametrit sulkuihin suljettuna, tai tyhjä pari sulkuja.

LUOKAN INSTANSSIN luonti tyhjänä ja alustamattomana on usein kömpelöä ja vaatisi useita rivejä manuaalista arvojen asettamista jokaisen uuden instanssin luonnin yhteydessä. Instanssien konfiguroimista varten Java tarjoaa *rakentaja (constructor)* -ominaisuuden. Rakentaja on metodi, joka määritellään luokan sisällä erityisellä syntaksilla ja joka palauttaa luokan instanssin. Rakentaja määritellään ilman paluuarvoa ja sen nimi on identtinen luokan nimen kanssa. Rakentajan voi määritellä ottamaan mitä tahansa argumentteja kyseinen tapaus tarvitsee. Rakentajametodi ei tarvitse return-avainsanaa, vaan palauttaa automaattisesti käsittelemänsä instanssin.

RAKENTAJAN YHTEYDESSÄ on tärkeää puhua *this*-avainsanasta. This tarjoaa ohjelmalle mahdollisuuden viitata luokan instanssiin, jolle jokin metodi kuuluu. This toimii vain rakentajassa ja instanssiin sidotussa metodissa, eli konteksteissa, joissa instanssi, johon viitataan on tiedossa. Täten rakentajassa voidaan yhdistämällä this-avainsana ja pistenotaatio viitata helposti rakennettavaan instanssiin.

SEURAAVA KOODIESIMERKKI esittelee rakentajan määrittelemisen ja new-avainsanan käytön rakentajan kanssa. On huomioitavaa, että seuraavassa alakappaleessa käsiteltävä ylikuormittaminen toimii myös rakentajametodeihin, eli luokalla voi olla useampi ylikuormitettu rakentajametodi. Koodiesimerkki kuitenkin käyttää vain yhtä rakentajametodia.

Koodiesimerkki 8: week2/constructorexample: Rakentajan luonti Javassa

```
package week2.constructorexample;
```

```
public class ClassWithConstructor {

    public int intData;
    public String stringData;

    // Rakentajametodi luodaan ilman paluuarvoa tai return-avainsanaa
    public ClassWithConstructor(int intData, String stringData) {
        this.intData = intData;
        this.stringData = stringData;
    }
}
```

Koodiesimerkki 9: week2/constructorexample: new-avainsanan käyttö

```
package week2.constructorexample;

public class ConstructorExample {

    public static void main(String[] args) {
        ClassWithConstructor dataPoint =
                new ClassWithConstructor(1111, "String data");

        System.out.println(dataPoint.intData + ", " + dataPoint.stringData);
    }
}
```

Metodin ylikuormitus

Joskus on toivottavaa tukea useampaa eri yhdistelmää argumenteja metodikutsulle. Pythonissa tämä onnistuu oletusarvoilla, mutta Java ei tue oletusarvoja argumenteille. Sen sijaan Javassa on olemassa konsepti nimeltä *ylikuormitus (overloading)*. Ylikuormittamisessa määritellään koodissa jo käytetty metodinimi uudestaan, erilaisilla argumenteilla. Jos argumenttejä on sama määrä kuin ylikuormitettavassa metodissa, pitää vähintään yhden argumentin tietotyypin poiketa alkuperäisestä metodista. Jos argumenttejä on eri määrä, ei niiden tietotyypeillä ole väliä. Java hakee metodikutsun yhteydessä automaattisesti kutsun signatuuria vastaavan version ylikuormitettua metodia kutsuttaessa. Seuraavassa esimerkissä esitellään ylikuormitetun metodin käyttö luomalla tulostusmetodi, joka ilmoittaa, onko tulostettavan muuttujan tietotyyppi int vai String.

Koodiesimerkki 10: week2/overloadingexample: Tulostajaluokka ylikuormittamisesimerkkiin

```
package week2.overloadingexample;

public class OverloadedPrinter {
```

```
    // Ylikuormittaminen tapahtuu määrittelemällä ensimmäisenä ylikuormitettava metodi normaalisti
    public void print(String printed) {
        System.out.println("String: " + printed);
    }

    // Ylikuormituksella jo olemassa olevaa metodinimeä voidaan käyttää uudestaan, kunhan signatuuri
    // eroaa olemassa olevista signatuureista. Metodin voi ylikuormittaa kuinka monta kertaa
    // tahansa, niin kauan kun jokaisen ylikuormittavan metodin signatuuri on uniikki.
    // (signatuuri = metodin nimi + kaikkien argumenttien tietotyypit)
    public void print(int printed) {
        System.out.println("int: " + printed);
    }
}
```

Koodiesimerkki 11: week2/overloadingexample: Ylikuormittaminen Javassa

```
package week2.overloadingexample;

public class OverloadingExample {

    public static void main(String[] args) {
        OverloadedPrinter printer = new OverloadedPrinter();

        // Kutsuvasta päästä nähden ei ole mahdollista nähdä ylikuormitettua metodia. Metodi
        // kutsutaan kahdesti samalla nimellä, vain annettujen argumenttien tietotyyppi eroaa.
        printer.print("Example string");
        printer.print(17);
    }
}
```

Näkyvyysmääreet

LÄHES KAIKKIEN koodin osien, kuten muuttujan, metodin, interface-luokan tai luokan luominen Javassa tukee kyseisen koodin osan näkyvyyden määrittämistä. Tämä määritelmä on nimeltään *näkyvyysmääre (access modifier)*. Kuten oppaassa aiemmin todettiin, yksi olio-ohjelmoinnin ydinideoista on *enkapsulaatio*, eli koodin osien vastuiden tarkka rajaaminen. Näkyvyysmääreet ovat tärkein työkalu tämän suhteen. Näkyvyysmääre rajoittaa ohjelmiston osan, kuten luokan tai luokan ominaisuuden näkyvyyttä muulle ohjelmalle seuraavan taulukon mukaisesti:

Avainsana	Näkyvyys	Käyttökohteet
private	Vain tämä luokka	Luokkien ominaisuudet
Ei avainsanaa (default)	Kaikki luokat tässä paketissa	Luokat ja luokkien ominaisuudet
protected	Kaikki luokat tässä paketissa ja kaikki tämän luokan perivät luokat	Luokkien ominaisuudet
public	Kaikki luokat kaikkialla	Luokat ja luokkien ominaisuudet

PRIVATE JA PROTECTED ovat siis näkyvyysmääreitä, jotka toimivat vain luokan ominaisuuksien näkyvyyden rajaamisessa, kun taas public ja oletusnäkyvyys toimivat myös luokkien näkyvyyden määrittämisessä. Tämä johtuu siitä, että private ja protected ovat suoraan sidottuja luokkaan, jossa ominaisuus sijaitsee. Protected on lisäksi tiukasti sidoksissa periytymiseen, joka käsitellään myöhemmin oppaan alaluvussa Periytyminen ja Java joten sen toiminnallisuuden ymmärtäminen ei ole vielä ajankohtaista.

TÄHÄN MENNESSÄ oppaassa on vältetty näkyvyysmääreiden käyttöä, eli niitä on käytetty vain main-metodien määrittelemisen yhteydessä. Tällöin Java-kääntäjä asettaa näkyvyydeksi automaattisesti oletusnäkyvyyden (default/package-private), jossa luokka, tai sen ominaisuus näkyy kaikkialle sen määrittelypaketin sisällä, muttei muualle. Tämä ei ole suositeltavaa, vaan jokaisen luokan ja luokkien jokaisen ominaisuuden näkyvyyden tarve pitäisi arvioida erikseen ja asettaa kullekin tiukin mahdollinen näkyvyysmääre. Tämä tyyli paitsi takaa mahdollisimman korkean enkapsulaation, myös tekee ohjelmistosta tehokkaamman, koska vain luokan sisällä näkyvät metodit voidaan optimoida tehokkaammin ohjelman kääntämisen yhteydessä.

SEURAAVA KOODIESIMERKKI sisältää pääluokan ja kaksi apuluokkaa ja esittelee näin public ja private avainsanojen toimivuuden. Avainsana protected on esitelty koodiesimerkissä periytyvyyden yhteydessä kappaleessa Näkyvyysmääre protected.

Koodiesimerkki 12: week2/accessmodifierexample: Ensimmäinen näkyvyysmääre-esimerkin luokka

```
package week2.accessmodifierexample;

public class FirstClass {

    // Tämä merkkijono näkyy, kun luokkaa käytetään muualla koodissa.
    public String publicString = "Public string";

    // Tämä metodi ei näy muualla koodissa.
    private boolean privateMethod() {
        return true;
    }
}
```

Koodiesimerkki 13: week2/accessmodifierexample: Toinen näkyvyysmääre-esimerkin luokka

```
package week2.accessmodifierexample;

public class SecondClass {

    // Tämä merkkijono ei näy, vaikka luokkaa käytettäisiin muualla koodissa.
```

```
    private String privateString = "Private string";

    // Tämä metodi näkyy kaikkialla koodissa.
    public boolean publicMethod() {
        return true;
    }
}
```

Koodiesimerkki 14: week2/accessmodifierexample: Näkyvyysmääre-esimerkin pääluokka

```
package week2.accessmodifierexample;

public class AccessModifierExample {

    public static void main(String[] strings) {

        // Luodaan instanssit molemmista aiemmista luokista. Tämä käydään seuraavassa oppaan
        // alaosiossa läpi tarkemmin.

        // Metodin sisällä määritellyt muuttujat näkyvät aina vain metodin sisälle, eivätkä täten
        // tarvitse näkyvyysmäärettä.
        FirstClass firstChild = new FirstClass();
        SecondClass secondChild = new SecondClass();

        // Koska tässä kysytään julkista metodia ja julkista String-instanssia, tämä toimii.
        if (secondChild.publicMethod()) {
            System.out.println(firstChild.publicString);
        }

        // Pois kommentoitu koodinpätkä, jossa sekä if-portti, että printtaus epäonnistuisivat:
        /*
        if (firstChild.privateMethod()) {
            System.out.println(secondChild.privateString)
        }
         */
    }
}
```

Noutajat ja asettajat

JAVA NEUVOO käyttäjiään alustamaan kaikki muuttujat private näkyvyysmääreellä. Tämä tarkoittaa, että koodatessa virallisten suositusten mukaista Javaa ohjelmoija tuottaa vain luokkia, joiden muuttujat näkyvät ainoastaan kunkin luokan sisällä. Kuitenkin monesti luokan säilömää dataa tarvitaan luokan ulkopuolella. Tämä laajemman näkyvyyden tarve ei ole suunnitteluvirhe ohjelmistossa, vaan yksinkertainen ohjelmoinnin totuus. Tätä varten Javaan on vakiintunut kaksi tärkeää metodityyppiä: *noutaja (getter)* ja *asettaja (setter)*.

NOUTAJA, ELI GETTER on funktio, jonka tehtävä on noutaa luokan sisällä yksityiseksi määritelty muuttuja luokan ulkopuoliseen käyttöön. Noutaja nimetään camelCase-tyylisesti lisäämällä noudettavan muuttujan nimen eteen "get". Tällöin muuttujan "privateVariable" noutajafunktion nimi on "getPrivateVariable". Noutajafunktio ei yleensä ota argumentteja ja palauttaa vain noudettavan muuttujan. Noutajan tehtävä on tarjota rajattu pääsy luokan säilömään dataan luokan ulkopuolelle. Noutajien käyttö mahdollistaa myös esimerkiksi laiskan alustamisen muuttujille, joiden alustaminen on raskasta, muttei aina tarpeellista.

ASETTAJA, ELI SETTER puolestaan on funktio, jonka tehtävä on asettaa luokan ulkopuolelta saatava arvo luokan sisällä yksityiseksi määriteltyyn muuttujaan. Asettaja nimetään noutajan tapaan camelCase-tyylisesti lisäämällä muuttujan nimen eteen "set". Täten edellä mainitun "privateVariable"-muuttujan asettajan nimi on "setPrivateVariable". Asettajafunktio ottaa yleensä asetettavan muuttujan tyyppiä olevan instanssin eikä palauta mitään. Samoin kuin noutajat, asettajat rajoittavat luokan ulkopuolista pääsyä luokan säilömään dataan. Lisäksi asettajat vastaavat usein luokan ulkopuolelta tulevan datan validoinnista ennen sen tallentamista muuttujaan.

NOUTAJIEN JA ASETTAJIEN käytön perusteet on esitetty seuraavassa koodiesimerkissä. Huomaa setPositiveNumber-metodissa toteutettava muuttujan validointi, joka ei olisi mahdollista, jos muuttuja positiveNumber olisi julkinen Javan suositteleman yksityinen muuttuja, noutaja ja asettaja -mallin sijaan.

Koodiesimerkki 15: week2/gettersetterexample: Ensimmäinen noutaja/asettajaesimerkin luokka

```
package week2.gettersetterexample;

public class NumberStorage {

    private int positiveNumber;

    // Suurin osa noutajista näyttää tältä.
    public int getPositiveNumber() {
        return this.positiveNumber;
    }

    // Asettajat mahdollistavat esimerkiksi annettujen arvojen validoinnin pakottamisen.
    public void setPositiveNumber(int positiveNumber) {
        if (positiveNumber < 0) {
            System.out.println("A positive number is required.");
            return;
        }
        this.positiveNumber = positiveNumber;
    }
```

```
}
```

Koodiesimerkki 16: week2/gettersetterexample: Noutaja/asettajaesimerkin pääluokka

```
package week2.gettersetterexample;

public class GetterSetterExample {

    public static void main(String[] args) {
        NumberStorage getterSetter = new NumberStorage();

        // Käytetään asettajafunktiota asettamaan yksityisen positiveNumber-muuttujan arvo
        getterSetter.setPositiveNumber(2021);

        // Noudetaan yksityisen positiveNumber-muuttujan arvo noutajafunktiolla
        System.out.println(getterSetter.getPositiveNumber());
    }
}
```

Paketit ja import

TÄHÄN MENNESSÄ oppaan käyttämät eri tiedostossa sijaitsevat luokat ovat aina olleet osa samaa pakettia, sillä opas jakaa esimerkit paketteihin kappaleen ja aiheen mukaan. Samassa paketissa olevat luokat ovat automaattisesti käytettävissä paketin sisällä, joten luokkia ei ole tarvinnut tuoda tiedoston nimitilaan tiedoston alussa. Aina tarvittavat luokat eivät kuitenkaan sijaitse samassa paketissa. Tätä varten Javassa on *import*-avainsana. Import-avainsanalla listataan tiedoston alussa tiedoston nimitilaan paketin ulkopuolelta tuotavat luokat. Myös Javan standardikirjasto sisältää luokkia, jotka on tuotava nimitilaan import-lauseella. Tästä hyviä esimerkkejä ovat seuraavaksi käytävät Scanner, InputStreamReader ja BufferedReader-luokat.

Syötteen vastaanottamisen perusteet

Käyttäjän syötteen vastaanottaminen Javassa

KOMENTORIVIIN PERUSTUVAN ohjelman tuottaminen vaatii paitsi kykyä tulostaa komentoriville tekstiä, myös kykyä kerätä käyttäjältä syötettä. Tätä varten Javasta löytyy useampi luokka, joiden avulla voidaan kerätä haluttu syöte juuri ohjelmistolle sopivalla tavalla. Näistä tärkeimmät ovat *Scanner*-lukka, *BufferedReader*- ja *InputStreamReader*-luokat. Koska Java monen muun ohjelmointikielen tapaan käsittelee konsolisyötettä ja tiedostosyötettä samoin, kaikkia näitä luokkia voi käyttää myös tiedostojen lukemiseen, käyttäjäsyötteen lukemisen

lisäksi. Huomioi, että syötteen pyytäminen on matalan tason tapahtuma, jonka täyden toiminnallisuuden ymmärtäminen ei ole tarpeellista oppaan käyttämiseksi. Seuraavat kappaleet avaavat koko syötteennoutoprosessin, mutta voit halutessasi hypätä suoraan seuraavaan koodiesimerkkiin, jos haluat vain opiskella tarvittavan syntaksin.

SCANNER-LUOKKA ottaa sisäänsä jonkin luettavan olion ja mahdollistaa sen parsimisen. Käyttäjän syöte tulee oletuksena standardikirjaston System-luokan in-muuttujaan, joten Scanner-luokan rakentaminen System.in-argumentilla on helpoin tapa muodostaa käyttäjältä syötettä keräävä olio. Tältä oliolta voi pyytää monia eri parsintametodeja, kuten seuraavan rivin parsimista, tai seuraavan numeron parsimista. Todennäköisesti yleisin parsintametodi on rivin parsiminen, joka tapahtuu Scanner-olion "nextLine()"-metodilla.

SCANNER, joka on koottu suoraan System.in-streamin päälle on helppo tapa vastaanotta käyttäjäsyötettä, mutta raa'a stream-olion lukeminen on kohtalaisen raskas toimenpide. Java suosittelee puskuroimaan useasti luetut syötteet, olivat ne tiedostoja tai konsolisyötteitä. Tätä varten on olemassa BufferedReader-luokka. Se ottaa rakennettaessa puskuroitavan merkkijonovirran. BufferedReader voidaan antaa suoraan argumenttina Scanner-oliolle luettavaksi syötteeksi.

IKÄVÄ KYLLÄ System.in ei ole merkkijonovirta, vaan tavuvirta. Sitä ei siis voida käyttää raakana BufferedReader-olion rakentamiseen. Tavuvirta voidaan muuttaa merkkijonovirraksi InputStreamReader-luokalla. Rakentamalla InputStreamReader ja käyttämällä System.in-virtaa argumenttina, saadaan merkkijonovirta, joka sisältää käyttäjän syötteen.

SYÖTTEEN NOUTAMINEN on ikävä kyllä Javassa monimutkainen prosessi. Seuraava koodiesimerkki sisältää kuitenkin tarvittavan syntaksin toimivan Scanner-luokan instanssin luomiseen. Kun instanssi on luotu kerran on sen käyttäminen onneksi simppeliä.

Koodiesimerkki 17: week2/basicexamples: Käyttäjän syötteen noutaminen Javassa

```
package week2.basicexamples;

// Huomioi tarvittavien luokkien tuominen nimitilaan import-lauseella. Tämä on pakollista yleensä
// käytettäessä javan standardikirjaston tarjoamia luokkia: String on huomattava poikkeus tähän.
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Scanner;

public class UserInput {

    public static void main(String[] args) {
        // Luodaan InputStreamReader System.in -streamin ympärille muuttamaan System.in tavuvirrasta
```

```
        // merkkijonovirraksi
        InputStreamReader defaultInputReader = new InputStreamReader(System.in);

        // Luodaan BufferedReader äsken luodun merkkijonovirran ympärille puskuroimaan lukuoperaatio
        BufferedReader bufferedReader = new BufferedReader(defaultInputReader);

        // Luodaan Scanner, joka lukee edellä luotua puskuroitua merkkijonovirtaa. Näin luodun
        // Scanner-olion nextLine-metodia voidaan nyt käyttää käyttäjäsyötteen noutamiseen.
        Scanner bufferedScanner = new Scanner(bufferedReader);

        // Käytetään uutta Scanner-oliota käyttäjän nimen kysymiseen.
        System.out.println("Please write your name: ");
        String bufferedUserInput = bufferedScanner.nextLine();
        System.out.println("Hello " + bufferedUserInput);
    }
}
```

Kokoelmarakenteet, toistorakenteet ja lisää oliopohjaista ajattelua

Kokoelmarakenteet

Ohjelmistot käsittelevät yleensä huomattavia määriä dataa. Ei ole mielekästä, eikä yleensä edes mahdollista määritellä kaikille mahdollisille datayksiköille omaa muuttujaansa. Tätä varten ohjelmointikielistä löytyy kokoelmamaisia rakenteita, jotka ovat nimensä mukaisesti yhteen muuttujaan tallennettavia datayksikkökokoelmia. Javan standardikirjasto sisältää useita kokoelmarakenteita erityyppisiin tilanteisiin, mutta opas käy läpi vain yleisimmin käytetyt kokoelmat.

Taulukot

Yksinkertaisin kokoelmarakenteista on *taulukko (array)*, joka löytyy simppeliytensä vuoksi useasta ohjelmointikielestä. Taulukko on tehokas rakenne indeksipohjaisissa hauissa ja lisäyksissä ja sen ylimääräinen muistijalanjälki on pieni verrattuna muihin tietorakenteisiin. Oppaassa on jo näkynyt taulukkomuuttujia, sillä main-funktio ottaa komentoriviargumentit muodossa String[], eli merkkijonoista koostuvana taulukkona. On huomioitava, että Java vaatii kaikkien taulukon alkioiden olevan samaa tyyppiä.

Taulukko alustetaan new-avainsanalla. Avainsanan jälkeen annetaan taulukon alkoiden tietotyyppi, jonka perään kirjoitetaan välittömästi hakasulkujen sisään taulukon koko. Muuttuja, joka sisältää taulukon määritellään puolestaan normaalin muuttujan määrittelyn tapaan, mutta muuttujan tietotyypin perään lisätään tyhjä pari hakasulkuja. Taulukon alkoihin viittaaminen puolestaan tapahtuu kirjoittamalla viitattavan alkion indeksi hakasulkuihin taulukon sisältävän muuttujan perään. Viittaus toimii samoin sekä alkion tallennuksessa, että alkion noutamisessa.

Seuraava esimerkki näyttää esimerkin taulukkojen luomisesta ja tallettamisesta muuttujaan, sekä arvojen tallentamisesta olemassa olevaan taulukkoon ja arvojen noutamisesta olemassa olevasta taulukosta.

Koodiesimerkki 18: week3/arrayexample: Dataluokka käytettäväksi taulukkoesimerkissä

```
package week3.arrayexample;

public class DataPoint {
    public String courseName;
    public int recommendedYear;
}
```

Koodiesimerkki 19: week3/arrayexample: Taulukkojen käyttö Javassa

```
package week3.arrayexample;

public class ArrayExample {

    public static void main(String[] args) {

        // Luodaan taulukko, joka sisältää neljä edellä esitellyn dataluokan instanssia. Taulukon
        // voi luoda primitiivisistä tyypeistä, standardikirjaston luokista tai käyttäjän itse
        // määrittelemistä luokista.
        DataPoint[] dataPoints = new DataPoint[4];

        // Täytetään instanssit esimerkkidatalla. Huomioi, että kuten suurimmassa osassa
        // ohjelmointikielistä, myös Javassa aloitetaan taulukon indeksointi nollasta. Lisäksi
        // huomionarvoista on mahdollisuus viitata olion ominaisuuksiin pistenotaatiolla, mikäli
        // taulukko koostuu olioista.
        dataPoints[0] = new DataPoint();
        dataPoints[0].courseName = "Ohjelmoinnin perusteet";
        dataPoints[0].recommendedYear = 1;

        dataPoints[1] = new DataPoint();
        dataPoints[1].courseName = "Olio-ohjelmointi";
        dataPoints[1].recommendedYear = 1;

        dataPoints[2] = new DataPoint();
        dataPoints[2].courseName = "Käyttöjärjestelmät";
        dataPoints[2].recommendedYear = 2;

        dataPoints[3] = new DataPoint();
        dataPoints[3].courseName = "Web Applications";
        dataPoints[3].recommendedYear = 3;

        // Luetaan data instansseista.
        System.out.println(
                "First entry (index 0): '" + dataPoints[0].courseName
                        + "', year: " + dataPoints[0].recommendedYear + "\r\n" +
                "Second entry (index 1): '" + dataPoints[1].courseName
                        + "', year: " + dataPoints[1].recommendedYear + "\r\n" +
```

```
                "Third entry (index 2): '" + dataPoints[2].courseName
                        + "', year: " + dataPoints[2].recommendedYear + "\r\n" +
                "Fourth entry (index 3): '" + dataPoints[3].courseName
                        + "', year: " + dataPoints[3].recommendedYear
        );
    }
}
```

Listat

Todennäköisesti yleisin tietorakenne datajoukkojen säilömiseen on lista. Javan standardikirjasto sisältää useamman listamaisen implementaation, mutta niistä yleisimmin käytetty ja monikäyttöisin on *ArrayList*. Se muistuttaa pythonin listaa toiminnaltaan, mutta pystyy Javan staattisen tyypityksen vuoksi säilömään vain yhden tyyppisiä muuttujia. Toisaalta on huomioitava, että mikäli ohjelmassa nousee ikinä tarve säilöä usean tyyppisiä muuttujia samaan listaan, on ohjelma luultavasti epäoptimaalisesti suunniteltu. ArrayList ei kykene säilömään primitiivisiä tietotyyppejä.

ArrayList tukee pythonin listan tapaan indeksipohjaista hakua ja lisäystä, alkoiden hakua kriteeripohjaisesti ja iterointia. Toisin kuin taulukko se on kooltaan dynaaminen eikä sen kokoa siksi määritellä sen luonnin yhteydessä. Luonti tapahtuu new-avainsanalla, mutta koska Javan täytyy tietää paitsi listan tyyppi, myös listan säilömien olioiden tyyppi, on luomisen yhteydessä määriteltävä tämäkin. ArrayList on siis *geneerinen luokka* (*generic class*). Geneeristen luokkien luominen ja toimintaperiaate ei ole tämän luvun aiheena, vaan ne käsitellään oppaan luvussa Geneerisyys Tässä kohtaa opasta ArrayListin geneerisyys on huomioitava vain uuden instanssin luonnin yhteydessä, jolloin listan säilömä tyyppi on suljettava pienempi kuin- ja suurempi kuin -merkkien väliin. Seuraava esimerkki näyttää ArrayList-instanssin luomisen, alkioiden lisäämisen, hakemisen ja poistamisen.

Koodiesimerkki 20: week3/basicexamples: ArrayList-luokan käyttö Javassa

```
package week3.basicexamples;

// Huomioi tarve tuoda ArrayList tiedoston nimitilaan import-avainsanalla
import java.util.ArrayList;

public class ArrayListUsage {

    public static void main(String[] args) {
        // Java osaa päätellä ArrayListin sisältämän luokan tyypin, mikäli instanssi asetetaan
        // muuttujaan, jonka täysi tyyppi on määritelty (lista + säilötty tyyppi). Tästä syystä
        // "new ArrayList<>()" ei tarvitse String-tyyppiä <>-merkkien väliin.
        ArrayList<String> entries = new ArrayList<>();
```

```
        // Lisääminen listan loppuun tapahtuu pelkällä add-metodilla
        entries.add("First");
        entries.add("Third");

        // Lisäämisoperaatioon voi lisätä alkuun indeksin, jolloin alkio lisätään kyseiseen
        // indeksiin listan lopun sijaan
        entries.add(1, "Second");

        // Printtaa "First Second Third", koska alkio "Second" lisättiin indeksiin 1
        System.out.println(entries.get(0) + " " + entries.get(1) + " " + entries.get(2));

        // Poistaminen toimii joko määrittelemällä poistettava alkio tai alkion indeksi. Mikäli
        // poistaminen tehdään määrittelemällä alkio, etsii ohjelma ensimmäisen alkion, joka on
        // identtinen annetun alkion kanssa ja poistaa löydetyn alkion.
        entries.remove("Second");
        entries.remove(0);

        // Printtaa "Third", koska alkio "Second" ja listan ensimmäinen alkio poistettiin
        System.out.println(entries.get(0));
    }
}
```

Hajautustaulut

VÄLILLÄ KOKOELMAN sisältämää dataa on tarve yhdistää muuhun dataan. Vaikka olio-ohjelmoinnin hengen mukaisesti datan yhdistäminen tapahtuu luomalla uusi luokka tai muokkaamalla olemassa olevaa luokkaa ei tämä ole aina mahdollista tai järkevää. Erityisesti tilanteissa, joissa dataa haetaan jonkin avaimen avulla ei ole järkevää käyttää listaa, jossa on tämän avaimen omaavia alkioita hitaan ja hankalan hakuoperaation vuoksi. Tämän vuoksi Javassa on olemassa *HashMap*-luokka jonka perustana on *hajautustaulu (hash table)* -tietorakenne.

HASHMAP-LUOKKA säilöö avain-arvopareja ja on verrattavissa pythonin dict-rakenteeseen, mutta vaatii kaikkien avainten olevan samaa tyyppiä ja kaikkien arvojen olevan samaa tyyppiä, eikä takaa arvojen keskeneräisen järjestyksen säilymistä. Se on ArrayList-luokan tapaan geneerinen, mutta toisin kuin ArrayList, HashMap tarvitsee kaksi luokkaa määritelmänsä yhteydessä. HashMap tukee avain-arvo -parin lisäämistä, arvon hakemista ja postamista avaimen perusteella ja iterointia joko avainjoukon, arvojoukon tai avain-arvo -parien yli. HashMap pystyy säilömään identtisiä arvoja, mutta jokaisen avaimen on oltava uniikki. Seuraava esimerkki esittelee HashMap-instanssin luomisen, avain-arvo -parin lisäämisen ja arvon hakemisen sekä poistamisen avaimen perusteella. ArrayListin tapaan HashMap ei tue primitiivisten tietotyyppien säilömistä.

Koodiesimerkki 21: week3/basicexamples: HashMap-luokan käyttö Javassa

```
package week3.basicexamples;

// Huomioi tarve tuoda HashMap tiedoston nimitilaan import-avainsanalla
import java.util.HashMap;

public class HashMapUsage {

    public static void main(String[] args) {

        // ArrayList:in tapaan HashMap:in luonti samalla rivillä muuttujan määrittelyn kanssa ei
        // vaadi geneerisen tyyppimääritelmän toistamista (tässä String, String)
        HashMap<String, String> postalCodes = new HashMap<>();

        // Avain-arvo -pareja lisätään put-metodilla. Huomioi identtisten arvojen mahdollisuus.
        postalCodes.put("02100", "Espoo");
        postalCodes.put("02110", "Espoo");
        postalCodes.put("35850", "Lappeenranta");

        // Hakeminen onnistuu get-metodilla avainpohjaisesti
        // Tämä printtaa "02100: Espoo, 02110: Espoo, 35850: Lappeenranta"
        System.out.println("02100: " + postalCodes.get("02100") +
                ", 02110: " + postalCodes.get("02110") +
                ", 35850: " + postalCodes.get("35850"));

        // Poistaminen tapahtuu remove-metodilla niin ikäin avaimen avulla
        postalCodes.remove("02100");

        // Printtaa "02110: Espoo, 35850: Lappeenranta"
        System.out.println("02110: " + postalCodes.get("02110") +
                ", 35850: " + postalCodes.get("35850"));
    }
}
```

Toistorakenteet

Edellä esitellyiden kokoelmarakenteiden esimerkkikoodista oli ehkä jo havaittavissa toistuvat samankaltaiset kutsut, jotka aiheuttivat rumaa koodia ja pidensivät esimerkkejä turhaan. Kuten missä tahansa itseään kunnioittavassa ohjelmointikielessä, myös Javassa on olemassa toistorakenteita, joilla kokoelmien käsittely ja muut monesti toteutettavat koodin osat saadaan kirjoitettua järkevämpään, luettavampaan ja helpommin muokattavaan muotoon.

For-looppi

FOR-LOOPPI luodaan *for*-avainsanalla. Se poikkeaa pythonin for-loopista ja muistuttaa sen sijaan c:n for-looppia. Pythonin for-looppi tunnetaan Javassa foreach-looppina ja käsitellään myöhemmin osiossa For-each-looppi. For-avainsanan jälkeen for-loopissa seuraa iterointiehto sulkujen sisällä. Iterointiehto jaetaan kolmeen osaan kahdella puolipilkulla. Ensimmäinen osa kertoo, mikä muuttuja toimii iterointiehtona, toinen millä kyseisen muuttujan arvoilla iterointia jatketaan ja kolmas mitä muuttujalle tehdään jokaisen iteraation jälkeen. Yksi yleisimmistä iterointiehdoista on "(int i = 0; i < n; i++)". Auki luettuna tämä tarkoittaa:

1. Määritä iterointimuuttuja "i", joka on tyyppiä "int" joka on alustettu arvoksi 0
2. Iteroi niin kauan, kun i on pienempi kuin tavoiteluku n
3. Jokaisen iteraation jälkeen kasvata i:n arvoa yhdellä

++ on Javan iterointioperaattori joka käydään tarkemmin läpi tämän luvun lopussa. Se kasvattaa muuttujan arvoa yhdellä. Käytännössä koko iterointiehto siis tarkoittaa "Toista loopin rungossa olevaa koodia n kertaa, niin että muuttujaan i on sidottu lukuarvo, joka kertoo mones iteraatio on kyseessä." Muuttujaa i voidaan käyttää loopin ruumiin sisällä.

FOR-LOOPPIA käytetään yleensä lukujoukkojen iteroimiseen esimerkiksi taulukkojen alustuksessa. On huomioitavaa, että kokoelmien iterointi on yleensä helpompaa foreach-loopilla. For-loopin iterointiehtona voi myös toimia jo olemassa oleva muuttuja, iterointiehdoksi ei ole aina pakko luoda uutta muuttujaa. Jos iterointiehdoksi luodaan uusi muuttuja, se näkyy vain loopin sisällä. Seuraava koodiesimerkki esittelee for-loopin luomisen ja käytön perusteet käytännössä.

Koodiesimerkki 22: week3/basicexamples: For-loopin käyttö Javassa

```
package week3.basicexamples;

public class ForLoop {

    public static void main(String[] args) {

        // For-looppi, joka iteroi luvut 0-9 muuttujassa i
        for(int i = 0; i < 10; i++) {

            // Iterointiehtona toimiva muuttuja on käytettävissä loopin sisällä
            // Tulostaa luvut 0-9, jokaisen omalle rivilleen
            System.out.println(i);
        }

        // Seuraava rivi aiheuttaisi virhetilan, koska muuttuja i ei ole saatavilla loopin rungon
        // ulkopuolella:
```

```
        /* System.out.println(i); */
    }
}
```

For-each-looppi

EDELLISESSÄ KAPPALEESSA mainittiin, että kokoelmien yli iterointi on Javassa helpompaa foreach-rakenteella, kuin for-rakenteella. Rakenteet muistuttavat toisiaan siinä mielessä, että molemmat määritellään for-avainsanalla. Foreach-rakenteessa kuitenkin avainsanaa seuraa iteroitavan kokoelman määritelmä sulkuihin suljettuna. Määritelmä koostuu kahdesta puolipisteellä erotetusta osasta. Ensimmäinen osa kertoo, minkä tyyppiseen ja nimiseen muuttujaan iteroitavat arvot sidotaan ja toinen, mistä kokoelmasta muuttujat haetaan. Looppi iteroi koko kokoelman yli, sitoen annettuun muuttujaan jokaisen yksittäisen arvon kokoelmasta. Tämä käyttäytyminen muistuttaa pythonin for-looppia. Seuraava koodiesimerkki näyttää kokoelman yli iteroimisen for-each -loopilla.

Koodiesimerkki 23: week3/basicexamples: For-each -loopin käyttö Javassa

```
package week3.basicexamples;

public class ForEachLoop {

    public static void main(String[] args) {

        // Taulukko voidaan alustaa suoraan tämäntyyppisellä notaatiolla, mikäli taulukon koko
        // sisältö on jo tiedossa
        String[] weekdays = new String[]{"Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
                "Saturday", "Sunday"};

        // Jokainen taulukon alkio sidotaan vuodollaan String-tyypin weekday-muuttujaan
        // Muuttuja on käytettävissä vain loopin sisällä
        // Printtaa viikonpäivät taulukosta, jokaisen omalle rivilleen
        for(String weekday : weekdays) {
            System.out.println(weekday);
        }
    }
}
```

While-looppi

WHILE-LOOPPI on toistorakenne, jossa for-loopin tapaan sidotaan toistoehto johonkin totuusarvoon, yleensä tietyn muuttujan suhteen. Toisin kuin for-loopissa, while loopissa toisto ei tapahdu iterointimaisesti yhden muuttujan suhteen. Sen sijaan looppi luodaan *while*-

avainsanalla, jota seuraa loopin toistoehto sulkuihin suljettuna. Toistoehto voi olla mikä tahansa totuusarvo ja looppia toistetaan niin kauan, kun toistoehto on totta. While-looppi on yleinen ohjausrakenteena ohjelman ytimessä: ohjelmaa tai ohjelman osaa ajetaan niin kauan kuin ennalta määritelty ehto on totta. Seuraava koodiesimerkki esittelee while-loopin käytön.

Koodiesimerkki 24: week3/basicexamples: While-loopin käyttö Javassa

```
package week3.basicexamples;

public class BasicWhileLoop {

    public static void main(String[] args) {

        // While-looppia käytettäessä toistoehdon sisältämät muuttujat on määriteltävä ennen looppia
        int number = 0;

        // Toistoehto olisi yleensä sidottu esimerkiksi käyttäjän syötteeseen tai muuhun
        // ulkopuoliseen etukäteen tuntemattomaan arvoon. For- ja for-each -loopit ovat
        // helppolukuisempia ja helppokäyttöisempiä käsiteltäessä etukäteen tunnettua arvojoukkoa.
        // Printtaa numerot 0-9 omille riveilleen
        while(number < 10) {
            System.out.println(number);
            number++;
        }
    }
}
```

Do-while -looppi

Välillä ilmenee tarve käyttää while-loopin kaltaista toistorakennetta, mutta tarkistaa toistoehdon totuusarvo vasta jokaisen toiston jälkeen. Tätä varten Javasta löytyy do-while -looppi. Looppi aloitetaan *do*-avainsanalla, jota seuraa saman tien loopin runko suljettuna kaarisulkeisiin. Vasta rungon jälkeen määritellään toistoehto käyttämällä while-avainsanaa, jonka jälkeen toistoehto suljetaan sulkeisiin. Looppi käyttäytyy toistoehdon tarkastushetkeä lukuunottamatta tismalleen samoin kuin while-looppi. Yleisimmin tätä looppia käytetään, kun halutaan loopin rungon tulevan ajetuksi ainakin kerran. Seuraava koodiesimerkki esittelee tällaisen tilanteen: toistoehto on koko ohjelman ajan epätosi, mutta loopin runko ajetaan kuitenkin kerran, koska toistoehto tarkastetaan vasta ajon jälkeen. Tämä ei olisi mahdollista while-loopilla.

Koodiesimerkki 25: week3/basicexamples: Do while -loopin käyttö Javassa

```java
package week3.basicexamples;

public class DoWhileLoop {

    public static void main(String[] args) {

        int number = 0;

        // Do-while -loopissa toistoehto tarkastetaan vasta loopin rungon jälkeen. Täten loopin
        // runko ajetaan ainakin kertaalleen, vaikka toistoehto olisi koko ajan epätosi.
        do {
            System.out.println("Inside do while loop.");
            number++;
        } while(number < 0);
    }
}
```

Avainsanat break ja continue

Looppeja käyttäessä on yleensä tarve ohittaa osa loopin rungon ajosta tai keskeyttää loopin ajo ennen kuin toistoehto täyttyy tai iteroitava joukko on käyty kokonaan läpi. Esimerkiksi hakuoperaatiossa on toivottavaa poistua toistorakenteesta heti kun toivottu alkio löytyy. Tätä varten Javassa on kaksi avainsanaa: *break* ja *continue*. break-avainsana keskeyttää loopin toistamisen ja poistuu toistorakenteesta, kun taas continue keskeyttää yhden toiston ja aloittaa seuraavan toiston saman tien. Avainsanat mahdollistavat toistorakenteen tarkemman ohjauksen. Seuraava esimerkki esittelee molempien avainsanojen käytön toistorakenteessa, joka luettelee parillisia lukuja nollasta alkaen maxNumber-muuttujassa määriteltyyn maksimiarvoon asti.

Koodiesimerkki 26: week3/basicexamples: Break- ja continue-avainsanojen käyttö Javassa

```java
package week3.basicexamples;

public class BreakContinue {

    // On huomioitavaa, että tämä luokka ei sisällä hyvää koodia. Parillisten lukujen luettelointi
    // onnistuisi helpoiten yksinkertaisella for loopilla. Luokka on kirjoitettu puhtaasti
    // esimerkiksi demonstroimaan break- ja continue-avainsanojen käyttöä
    public static void main(String[] args) {
        int maxNumber = 20;
        int currentNumber = 0;

        // while(true) luo ikuisen loopin. Ainut tapa päästä loopista pois on break-avainsana
        while(true) {
            currentNumber++;
            // Poistumisehto break-avainsanalla
```

```
            if (currentNumber >= maxNumber) {
                break;
            }

            // Käytetään modulomuuttujaa parillisuuden selvittämiseen ja skipataan loput loopista,
            // mikäli luku ei ole parillinen
            if (currentNumber % 2 == 1) {
                continue;
            }

            // Ohjelma tulostaa luvut 2, 4, 8, 10, 12, 14, 16 ja 18, kaikki omille riveilleen
            System.out.println(currentNumber);
        }
    }
}
```

Javan ajamisen teoriaa

JAVAN AJAMINEN perustuu Javan virtuaalikoneeseen, joka luo ohjelmille oman pienen hiekkalaatikon käyttöjärjestelmän sisällä ja standardoi täten ohjelmat alustasta riippumatta. Tässä kappaleessa kurkataan hieman virtuaalikoneen toimintaan ja käyttöön ohjelmoijan näkökulmasta.

Tulkkaus vastaan kääntäminen

OHJELMOINTIKIELET voidaan jakaa karkeasti kahteen ryhmään sen mukaan, miten niillä kirjoitettu ohjelma ajetaan. Jos kysessä on *tulkattu kieli (intrepeted language)*, ohjelmakoodia luetaan suoraan tekstitiedostosta ajon aikana tarpeen mukaan. Jos kyseessä on taas *kaannetty kieli (compiled language)*, ohjelmakoodi kompiloidaan ennen ajoa. Java on mielenkiintoinen tapaus tässä suhteessa, sillä Javaa ajettaessa ohjelmakoodi kompiloidaan tavukoodiksi, jota Javan virtuaalikone ajaa. Tavukoodin ajotapa riippuu virtuaalikoneesta, mutta yleensä Javan tavukoodin ajotekniikka on *ajonaikainen kaantaminen (Just-In-Time compiling)*.

AJONAIKAINEN KÄÄNTÄMINEN on ohjelman ajamisstrategia, jossa ennalta tuotettua tavukoodia käännetää dynaamisesti ohjelman ajon aikana konekieliseen muotoon. Tämä eroaa ratkaisevasti tulkkaamisesta, tarjoten paremman, lähes kääntämiseen verrattavan suorituskyvyn pienen käynnistysviiveen kustannuksella. Yksi tekniikan eduista on älykkäiden kääntäjien kyky optimoida koodia dynaamisesti ajon aikana. Lisäksi kaikkia tavukooditiedostoja ei tarvitse kääntää uudestaan aina, kun koodia muokataan, joten kehitysvaiheen versioiden iterointi on nopeampaa kuin aidoissa käännetyissä kielissä.

TÄMÄ TARKOITTAA, että Java-ohjelman kääntäminen ja ajamisen aloittaminen vie vähemmän aikaa, kuin c:llä tai muulla käännettävällä kielellä. Ohjelman ajaminen ei kuitenkaan tapahdu saman tien niin kuin tulkatulla kielellä, vaan bittikoodin kompilointi ja ajonaikaisen kääntämisen aloittaminen vie hieman aikaa. Ajonaikaisen kääntämisen suorituskyky on lähellä käännettyjä kieliä, muttei ihan samaa tasoa. Suorituskyky on kuitenkin huomattavasti parempi kuin tulkatuilla kielillä.

Muistinhallinta ja garbage collector

MIKÄLI OLET ohjelmoinut aiemmin vain pythonin kaltaisilla korkeamman tason kielillä, muistinhallinta ei välttämättä ole tuttu asia. Myös Java abstraktoi korkean tason kielenä muistinhallinnan pois käyttäjältä, joten toimivan ohjelman luominen ei vaadi muistinhallinnasta välittämistä. Ohjelmojen optimoinnin kannalta on kuitenkin hyödyllistä ymmärtää muistinhallinnan perusteet.

JAVAN MUISTINHALLINNAN pohjana on niin sanottu *roskankeruu (garbage collection)*. Roskankeruun ideana on ajaa erillinen roskankeruuohjelma tietyin väliajoin. Tämä ohjelma käy läpi muistissa olevat muuttujat ja tarkistaa jokaisen muuttujan kohdalla, onko ohjelmassa aktiivisia viittauksia kyseiseen muuttujaan. Tämän jälkeen ohjelma poistaa kaikki muuttujat, joihin ei ole olemassa aktiivisia viittauksia. Ohjelmien suorituskyvyn parantaminen muistijalanjäljen suhteen onnistuu siis Javassa miettimällä tarkkaan, miten laajalle alueelle muuttuja näkyy: funktion sisään rajoitettu muuttuja on huomattavasti ystävällisempi muistinkäytön kannalta kun luokkaan tallennettu muuttuja.

Lisää Javan konsepteja

TÄSSÄ VAIHEESSA opasta on käyty läpi onnistuneesti suurin osa Javan peruskonsepteista. Reilusti yli kaksikymmentä vuotta vanhassa kielessä on kuitenkin valtava määrä avainsanoja, konsepteja ja syntaksikikkoja. Seuraavat alakappaleet käsittelevät muutamia tärkeimpiä konsepteja Javan käytössä. Moni avainsana jää tässä käsittelemättä, mutta suurin osa ohjelmointikielien oletetusta ominaisuusjoukosta täyttyy seuraavien konseptien avulla.

Valikkorakenne, switch ja case

MIKÄLI OHJELMASSA ilmenee tarve valita useasta rinnakkaisesta vaihtoehdosta pitkä ketju if- ja else-lauseita ei ole helpoin vaihtoehto. Javasta löytyy tällaiseen tilanteeseen erillinen valikkorakenne, joka luodaan *switch*-avainsanalla. Avainsanaa seuraa valikkorakenteen määritelmässä sulkuihin suljettu valintaehto. Valintaehdon jälkeen

seuraa itse valikkorakenteen runko kaarisulkuihin suljettuna. Runko koostuu haaroista, jotka määritellään *case*-avainsanalla. Avainsanaa seuraa valintaehdon arvo, jolla haara ajetaan. Jokainen haara lopetetaan yleensä break-avainsanaan, joka toistorakenteiden tapaan poistuu valikkorakenteesta. Jos break-avainsana jätetään pois haaran lopusta, valikko jatkaa seuraavaan haaraan, sopi valintaehto haaran ehtoon tai ei.

VALIKKORAKENTEEN VIIMEISEKSI haaraksi kirjoitetaan yleensä oletushaara, joka määritellää *default*-avainsanalla. Oletushaara ajetaan, mikäli mitään muuta haaraa ei ajeta, tai mikäli haaraan jatketaan break-avainsanan puutteen vuoksi edellisestä haarasta. Oletushaaraa voi käyttää esimerkiksi virhetilojen korjaamiseen. Seuraava koodiesimerkki esittelee ohjelman, joka muuttaa valintarakenteella kouluarvosanan sanalliseksi arvioksi.

Koodiesimerkki 27: week3/basicexamples: Valikkorakenne, switch- ja case -avainsanojen käyttö Javassa

```
package week3.basicexamples;

public class Switch {

    @SuppressWarnings("ConstantConditions")
    public static void main(String[] args) {
        // grade-muuttujan arvo saataisiin oikeassa ohjelmassa jostain muualta
        int grade = 9;
        String verbalGrade;
        // Valintaehdon ei ole pakko olla näin simppeli, vaan se voi olla monimutkaisempi
        // määritelmä. Esimerkiksi grade * 2 olisi validi valintaehto.
        switch(grade) {
            case 4:
                verbalGrade = "Failed";
                break;
            case 5:
                verbalGrade = "Passable";
                break;
            case 6:
                verbalGrade = "Decent";
                break;
            case 7:
                verbalGrade = "Satisfactory";
                break;
            case 8:
                verbalGrade = "Good";
                break;
            case 9:
                verbalGrade = "Laudable";
                break;
            case 10:
```

```
                verbalGrade = "Excellent";
                break;
            // Oletushaaraa voi käyttää  virhetilanteiden tunnistamiseen ja niihin reagointiin
            default:
                System.out.println("Unrecognized grade! Pleae give a grade from 4 to 10.");
                verbalGrade = "Error: Unrecognized grade";
                // Viimeinen haara (yleensä default) ei tarvitse break-avainsanaa
        }
        // Printtaa tässä tapauksessa "Laudable"
        System.out.println(verbalGrade);
    }
}
```

Luokkamuuttujat, luokkafunktiot ja static

OPPAASSA VIITATTIIN aiemmin jo luokkamuuttujiin/-metodeihin kappaleessa Oliopohjaisen ohjelman ajaminen: main-metodi. Luokkamuuttujat ja -metodit on sidottu ne omistavaan luokkaan luokan instanssin sijasta. Kaikki luokan instanssit voivat käyttää luokkamuuttujia tai kutsua luokkametodeja, mutta luokkametodin sisällä ei voi viitata instanssimuuttujaan, ainoastaan luokkamuuttujiin. Luokkametodeja ja -muuttujia voidaan kutsua myös luokan nimen avulla pistenotaatiolla luokan ulkopuolella. Esimerkiksi tulostamiseen käytetty System.out on oikeastaan luokkamuuttuja out standardikirjaston System-luokassa.

LUOKKAMETODIT JA -MUUTTUJAT luodaan *static*-avainsanalla. Avainsanan sijainti voi olla ennen tai jälkeen näkyvyysmääreen, mutta normalisoitu tyyli on kirjoittaa näkyvyysmääre ennen static-avainsanaa. Main-metodi on siis luokkametodi ohjelman juuriluokassa. Tämä tarkoittaa, ettei Javan tarvitse alustaa juuriluokan instanssia ohjelman ajon aluksi. Seuraava esimerkki esittelee luokkametodin luomisen apuriluokkaan ja kutsumisen apuriluokan ulkopuolelta ilman apuriluokan instanssia.

Koodiesimerkki 28: week3/staticexample: Luokkametodin luominen static-avainsanalla

```
package week3.staticexample;

public class StaticPrinter {

    // Muuttuja voisi olla prettyPrint-metodin sisällä, joten kehitysympäristö ehdottaa tätä.
    // Tämä @SuppressWarnings estää tämän valittamisen. @-merkin sisältävästä rivistä ei siis
    // tarvitse välittää.
    //
    // Luokkamuuttujan luonti
    @SuppressWarnings("FieldCanBeLocal")
    private static String decoratorString = "###";
```

```
    // Luokkametodin luonti. Huomioi, että luokkamuuttujaa voidaan käyttää luokkametodissa, mutta
    // normaalia muuttujaa ei voitaisi.
    public static void prettyPrint(String string) {
        System.out.println(decoratorString + " " + string + " " + decoratorString);
    }
}
```

Koodiesimerkki 29: week3/staticexample: Luokkametodin kutsuminen metodin määrittelevän luokan ulkopuolella

```
package week3.staticexample;

public class StaticExample {

    public static void main(String[] args) {

        // Luokkametodia voidaan kutsua luokan ulkopuolella luokan nimen ja pistenotaation avulla
        // Printtaa "### Example string ###" ja rivinvaihdon
        StaticPrinter.prettyPrint("Example string");
    }
}
```

Tyyppimuunnokset (casting)

STAATTISESTI TYYPITETYSSÄ kielessä voi olla välillä tarvetta vaihtaa tunnetun muuttujan tyyppiä. Esimerkiksi lukuja käsiteltäessä saattaa olla tarvetta vaihtaa luvun tyyppi kokonaisluvusta (int) liukuluvuksi (float) tai toisin päin. Java osaa vaihtaa muuttujan tyypin automaattisesti, mikäli tavoitetietotyyppi on laajempi kuin alkuperäinen tietotyyppi. Käytännössä esimerkiksi int-tyypin muuttuja voidaan aina muuttaa long-tyypin muuttujaksi, koska long-tietotyyppi sisältää aina tarvittavan muistin int-tyypin muuttujan säilömiseen. Kuitenkin välillä ei voida taata datan täydellistä säilymistä. Tällöin Java vaatii ohjelmoijaa määrittelemään tyypinmuunnoksen manuaalisesti. Tämän operaation nimi on *pakotettu tyypinmuunnos (casting)*.

PAKOTETTU TYYPINMUUTOS tapahtuu sulkemalla toivottu tietotyyppi muunnettavan muuttujan eteen sulkuihin. Java yrittää pakottaa muuttujan annettuun tietotyyppiin riippumatta siitä, onko muutos mahdollista, joten pakotettu tyypinmuunnos huonosti harkitussa paikassa voi aiheuttaa ongelmia ohjelman ajovaiheessa. Tyypinmuunnosta käsitellään tarkemmin periytymisen yhteydessä. Seuraava koodiesimerkki esittelee tyyppimuunnoksen perussyntaksin muuttamalla liukulukumuuttujan kokonaisluvuksi.

Koodiesimerkki 30: week3/basicexamples: Tyyppimuunnos primitiivisestä tietotyypistä toiseen Javassa

```java
package week3.basicexamples;

public class Casting {

    public static void main(String[] args) {
        double pi = 3.14159265359;
        // Tyyppimuunnoksen syntaksi
        int intPi = (int) pi;
        // Printtaa: "Pi as int: 3"
        System.out.println("Pi as int: " + intPi);
    }
}
```

Vakiot ja final

MONET OHJELMAT käyttävät vakioarvoja esimerkiksi asetusten tallentamiseen, tai helpottamaan numeraalisten vakioiden lukemista koodissa. Javassa on tätä tarkoitusta varten erillinen avainsana *final*. Final-avainsanalla määriteltyyn muuttujaan asetettua arvoa ei voi enää muokata sen asettamisen jälkeen. Muuttujan voi määritellä final-avainsanalla asettamatta sille arvoa. Tällöin ensimmäinen muuttujaan asetettu arvo jää vakioksi. Tämä on yleistä määrittelemällä vakiomuuttuja luokan ruumiissa ja asettamalla sille arvo rakentajassa. Seuraava koodiesimerkki esittelee final-avainsanan käytön perusteet.

Koodiesimerkki 31: week3/basicexamples: final-avainsanan käyttö Javassa

```java
package week3.basicexamples;

public class FinalUsage {

    // Vakioarvon määrittäminen tapahtuu yleensä määreillä "public static final" Tällöin vakio on
    // nähtävissä kaikkialla koodissa. Luokan sisäisissä vakioissa public-määre korvataan
    // private-määreellä.
    public static final String constantString = "Constant string data";

    public static void main(String[] args) {
        // Seuraava rivi ei kompiloi, koska final-kenttää yritetään muokata
        /* constantString = "Changed string data"; */

        System.out.println(constantString);
    }
}
```

Luetellut tyypit (enum)

Mikäli muuttuja voi saada arvon vain jostain tarkasti rajatusta joukosta, kuten ilmansuunnista, voi ensimmäisenä tulla mieleen kuvata näitä arvoja kokonaisluvuilla tai ennalta määritellyillä merkkijonoilla. Molemmat tavat toimivat, mutta kokonaisluvut hankaloittavat koodin lukemista, koska ohjelmoijan täytyy muistaa mitä arvoa mikäkin luku edustaa. Merkkijonot taas ovat alttiita kirjoitusvirheille ja ovat hitaampia käsitellä ajon aikana. Tätä varten suurimmasta osasta ohjelmointikieliä löytyy *lueteltu tyyppi (enumerated type)* -ominaisuus.

Lueteltu tyyppi luodaan Javassa korvaavalla luokan luomisessa class-avainsana *enum*-avainsanalla. Yksinkertaisimmillaan lueteltu tyyppi sisältää vain kaikkien tyypin mahdollisten arvojen määritelmät. Arvot nimetään yleensä CAPITAL_CASE -nimeämistyylillä, erotetaan toisistaan pilkulla ja erotellaan omille riveilleen. Arvoja voi käyttää joko pistenotaatiolla arvoluokan nimestä tai jos lueteltu tyyppi on määritelty samassa nimitilassa tai tuotu nimitilaan import-avainsanalla pelkällä arvon nimellä. Seuraavat koodiesimerkit esittelevät luetellun tyypin luomisen ja käytön.

Koodiesimerkki 32: week3/enumexample: Lueteltu tyyppi ilmansuunnista

```
package week3.enumexample;

public enum CompassPoint {
    NORTH,
    EAST,
    SOUTH,
    WEST
}
```

Koodiesimerkki 33: week3/enumexample: Lueteltuun tyyppiin viittaaminen

```
package week3.enumexample;

public class EnumExample {

    public static void main(String[] args) {
        // Lueteltuun tyyppiin viittaaminen pistenotaatiolla
        CompassPoint compassPoint = CompassPoint.EAST;

        System.out.println("Enum value assigned was: " + compassPoint);
    }
}
```

Aritmeettiset- ja bittioperaatiot

Java sisältää monia operaatioita lukujen ja tavumuotoisten tietojen käsittelyyn. Seuraava taulukko esittelee Javan perusoperaatiot.

Bittikohtaiset operaatiot Javassa ottavat kaksi kokonaislukutyypin muuttujaa (byte, short, int tai long) ja suorittavat kyseisen operaation luvun binääriesityksen jokaiselle bitille. Esimerkiksi luvut kaksitoista (1100) ja kolme (0011) tuottaisivat bittikohtaisella TAI-operaatiolla tuloksen viisitoista (1111).

Operaatio	Nimi	Toiminta	Esimerkki
+	summa	a + b	Laskee muuttujat a ja b yhteen
-	erotus	a + b	Vähentää muuttujan b muuttujasta a
*	tulo	a * b	Laskee muuttujien a ja b tulon
/	osamäärä	a / b	Laskee muuttujien a ja b osamäärän
%	modulo	a % b	Laskee muuttujan a modulon (jakojäännös) jakajalla b
++	inkrementti	i++	Kasvattaa kokonaislukumuuttujan i arvoa yhdellä
--	dekrementti	i--	Vähentää kokonaislukumuuttujan i arvoa yhdellä
==	on yhtä suuri kuin	a == b	Palauttaa tosi, jos a on yhtä suuri kuin b
!=	on eri suuri kuin	a != b	Palauttaa tosi, jos a on eri suuri kuin b
>	suurmepi kuin	a > b	Palauttaa tosi, jos a on suurempi kuin b
<	suurmepi kuin	a < b	Palauttaa tosi, jos a on pienempi kuin b
>=	suurmepi tai yhtä suuri kuin	a >= b	Palauttaa tosi, jos a suurempi tai yhtä suuri kuin b
<=	pienempi tai yhtä pieni kuin	a <= b	Palauttaa tosi, jos a pienempi tai yhtä pieni kuin b
&	bittikohtainen JA	a & b	Palauttaa bittikohtaisen JA-operaation tuloksen luvuille a ja b
|	bittikohtainen TAI	a | b	Palauttaa bittikohtaisen TAI-operaation tuloksen luvuille a ja b
^	bittikohtainen XTAI	a ^ b	Palauttaa bittikohtaisen XTAI-operaation tuloksen luvuille a ja b
~	Komplementti	~ a	Palauttaa a:n yhden komplementin

Ensimmäinen esimerkkiohjelma: osoitekirja

Opas on nyt esitellyt kaiken tarvittavan Javasta simppelien ohjelmien kirjoittamiseen. Normaalin ohjelman toiminnan, suunnittelun ja käytön esittelemiseksi tämän kappaleen loppu omistetaan toimivalle koodiesimerkille. Koodiesimerkki sisältää osoitekirjaohjelman, johon käyttäjä voi tallentaa kontaktidataa ja josta käyttäjä voi hakea dataa kontaktin nimellä. Tämä esimerkki ei esittele uusia toiminnallisuuksia, joten sen voi halutessaan ohitta ja jatkaa suoraan luvun Tiedostonhallinta ja virheenkäsittely alkuun. Esimerkki esittelee kuitenkin paitsi Javan perusteiden toimintaa laajasti. Myös konkreettisia tapoja lähestyä oliopohjaisen ohjelman suunnittelua ja toteuttamista, joten esimerkkiin tutustumista suositellaan lämpimästi.

Osoitekirjaohjelman toimintaidea ja suunnittelu

Ohjelman suunnittelu on aina syytä aloittaa linjaamalla ohjelman tavoitteet. Tällä kurssilla ei mennä sen tarkemmin projektinhallintaan, ketterään kehitykseen tai muihin metodeihin, joten perustoimintaideoiden listaaminen riittää. Ohjelman perusvaatimukset ovat:

- Ajettavissa komentoikkunassa
- Sisältää ikilooppiin perustuvan tekstipohjaisen käyttöliittymän
- Tallentaa puhelinnumeron nimetylle kontaktille
- Kykenee hakemaan käyttäjän puhelinnumeron nimen perusteella

LISTAN PERUSTEELLA voidaan ruveta miettimään ohjelman rakennetta. Toivottu ikilooppi voi sijaita pääluokassa ohjelman pienen koon ja yksinkertaisen luonteen vuoksi. Pääluokalle ei ole järkevää kasata liikaa vastuita, sillä tämä vaikeuttaa ohjelman laajentamista tulevaisuudessa. Olisi siis järkevää luoda luokka, joka hoitaa osoitekirjan manageroinnin. Pääluokka voi sitten kutsua tätä luokkaa käyttäjän valintojen mukaan.

PÄÄTASOLLA OHJELMAN arkkitehtuuri on siis pääluokka, jossa luodaan AdressBookManager-luokan instanssi ja jonka main-metodi pyörittää ikuista looppia. Tämä manager-instanssi sisältää metodeja, jota pääluokasta voidaan kutsua käyttäjän valinnan mukaan. Manageri tarvitsee ainakin metodit seuraaviin tarkoituksiin:

- Kontaktin luominen: lisää nimi ja kontaktitiedot kontaktilistaan
- Kontaktin haku: Hae kontaktin tiedot listasta nimen perusteella

LISTASTA NÄHDÄÄN, että kontaktien hakeminen tapahtuu aina nimen perusteella. Tämä on tärkeää ottaa huomioon kontakteja säilövää tietorakennetta suunnitellessa. HashMap on rakenne, josta on nopea hakea tietoa avaimen perusteella, joten ContactManager voi säilöä kontaktit HashMap-instanssiin, jossa on avaimena kontaktin nimi. Kontaktidataa varten voitaisiin luoda uusi luokka puhdasta datan säilömistä varten, mutta koska data koostuu kahdesta arvosta, joista toinen säilötään avaimena hajautustauluun, on järkevämpää tallentaa data vain puhelinnumerona avain-arvo -parin arvoksi.

TÄTEN ESIMERKKI koostuu kahdesta luokasta. AddressBook on pääluokka, joka sisältää ohjelman elossa pitävän ikiloopin, joka lukee käyttäjäsyötettä ja kutsuu niiden mukaan metodeja ContactManager-luokasta. ContactManager sisältää metodeja osoitekirjan manipulointiin ja itse osoitekirjan hajautustaulussa, jonka avaimia ovat merkkijonomuotoiset kontaktien nimet ja arvoja merkkijonomuotoiset puhelinnumerot. Puhelinnumeroiden säilöminen kokonaislukuna törmää kahteen ongelmaan. Maakoodien säilöminen ei olisi mahdollista ja nollalla alkavat numerot typistyisivät alusta.

Osoitekirjaohjelman koodi

Koodiesimerkki 34: week3/addressbookexample: Osoitekirjamanageri osoitekirjan hallintaan

```java
package week3.addressbookexample;

import java.util.HashMap;
import java.util.Scanner;

public class ContactManager {

    private final Scanner scanner;
    private final HashMap<String, String> addressBook = new HashMap<>();

    // On helpompaa luoda koko ohjelmalle yksi Scanner-instanssi, kuin huolehtia monesta
    // instanssista. Sen vuoksi ContactManagerin rakentaja ottaa pääluokasta Scannerin
    public ContactManager(Scanner scanner) {
        this.scanner = scanner;
    }

    public void addUser() {
        System.out.print("Please give the name of the contact to add: ");
        String contactName = scanner.nextLine();
        System.out.print("Please give the number of '" + contactName + "': ");
        String number = scanner.nextLine();
        // Virheenhallinta identtisille avaimille
        if (addressBook.containsKey(contactName)) {
            System.out.println("Error: Contact '" + contactName + "' already exists!");
            return;
        }
        addressBook.put(contactName, number);
        System.out.println("Added user '" + contactName + "' with number '" + number + "'");
    }

    public void getUser() {
        System.out.print("Please input the name of the contact you want to find: ");
        String contactName = scanner.nextLine();
        String contactNumber = addressBook.get(contactName);
        // Virheenhallinta puuttuville avaimille
        if (contactNumber == null) {
            System.out.println("Could not find a contact named '" + contactName + "'.");
            return;
        }
        System.out.println("Contact '" + contactName + "', number: " + contactNumber);
    }
}
```

Koodiesimerkki 35: week3/addressbookexample: Pääluokka, joka vastaa ohjelman ajamisesta

```java
package week3.addressbookexample;

import java.io.BufferedReader;
```

```
import java.io.InputStreamReader;
import java.util.Scanner;

public class AddressBook {

    public static void main(String[] args) {
        // Scannerin rakentamisen eristäminen metodiin antaa muokata scanner-implementaatiota
        // tulevaisuudessa ja tekee pääohjelmasta luettavamman
        Scanner bufferedScanner = constructScanner();
        ContactManager contactManager = new ContactManager(bufferedScanner);

        // Eristämällä operaatiovalinta boolean-paluuarvon omaavaan metodiin, voidaan ohjelman
        // päälooppi yksinkertaistaa pelkkään ohjeiden printtaamiseen do - while -loopilla ja
        // aliohjelman ajamiseen toistoehdossa
        do {
            System.out.print("Please choose what you want to do:\r\n" +
                             "  0: Exit program\r\n" +
                             "  1: Add a new contact\r\n" +
                             "  2: Read a contact from the book\r\n" +
                             "Please input your choice: ");
        } while (!performOperation(Integer.parseInt(bufferedScanner.nextLine()), contactManager));
    }

    // Tämä metodi on vastuussa ohjelman logiikasta. Se saa päämetodista numeron ja ContactManager-
    // instanssin ja kutsuu numeron perusteella instanssin metodeja
    static boolean performOperation(int operation, ContactManager contactManager) {
        switch (operation) {
            case 0:
                System.out.println("Terminating the program: Thank you for using it!");
                return true;
            case 1:
                contactManager.addUser();
                return false;
            case 2:
                contactManager.getUser();
                return false;
            default:
                System.out.println("Unknown choice: please give a whole number between 0 and 2");
                return false;
        }
    }

    // Apuohjelma Scannerin luomiseen
    private static Scanner constructScanner() {
        InputStreamReader defaultInputReader = new InputStreamReader(System.in);
        BufferedReader bufferedReader = new BufferedReader(defaultInputReader);
        return new Scanner(bufferedReader);
    }
}
```

Tiedostonhallinta ja virheenkäsittely

Tähän mennessä oppaassa on käyty läpi suurin osa Javan perustoiminnasta ja kerrytetty tarvittava tietotaito simppelien ohjelmistojen luomiseen. Opas ei kuitenkaan ole vielä tarjonnut ratkaisuja pysyvään datan säilömiseen, joten kaikki ohjelmien käsittelemä data kuolee ohjelmien ajon päätyttyä. Tässä kappaleessa tutustutaan yksinkertaisimpaan tapaan säilöä dataa pysyvästi: tiedostonhallintaan.

Linux-maailmasta tuttuun tapaan Javassa tiedostonhallinta muistuttaa vahvasti käyttäjäsyötteen käsittelyä. Tiedostojen lukeminen tapahtuu *FileReader* tai *FileInputStream* -olioilla ja tiedostoon kirjoittaminen *FileWriter* tai *FileOutputStream* -olioilla. Tässä kappaleissa käydään läpi näiden luokkien käyttö. Tätä ennen on kuitenkin opeteltava Javan virheenkäsittelyn syntaksi, sillä virhetilojen hallinta on olennainen osa tiedostojen turvallista käsittelyä.

Virheenkäsittelyn perusteet, try-catch-finally

Java sisältää monipuolisen virhetilajärjestelmän virheenkäsittelyä varten. Virhetilat esitetään Exception-luokasta perittyinä erityyppisinä Exception-instansseina. Esimerkiksi jos ohjelmisto yrittää hakea tietoa muuttujasta, jossa on säilötty vain null-arvo, nostaa Java automaattisesti NullPointerException-virheen.

Virheiden käsittely tapahtuu sulkemalla virhetilan mahdollisesti aiheuttava koodi *try*-avainsanalla määritellyn kaarisulkeisiin suljetun koodin osan sisään. Tätä seuraa virhetilan käsittely, joka määritellään *catch*-avainsanalla, jota seuraa sulkeissa kiinniotettavan virheen määrittely, eli virheluokka, jonka instanssit otetaan kiinni ja muuttujanimi, johon virheluokan ilmentymät kiinnitetään. Tätä määritelmää seuraa itse virhetilan korjaus suljettuna kaarisulkeisiin. Tämä korjausosio ajetaan vain, jos try-osassa tapahtuu sulkeissa määritellyn tyypin virhe. Viimeinen virheenkäsittelyrakenteen osa on *final*-avainsanaa seuraava kaarisulkeisiin suljettu yleinen sulkemiskoodi, joka ajetaan, tapahtui try-rakenteessa virhettä tai ei. Rakenteen ei tarvitse sisältää sekä catch- että finally-osioita, mutta vähintään toisen on oltava try-rakenteen perässä tai koodi ei käänny. Seuraava koodiesmerkki esittelee virheenkäsittelyn perusteet.

Koodiesimerkki 36: week4: try-, catch- ja finally-avainsanojen käyttäminen

```
package week4;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Scanner;

public class BasicTryCatchFinally {

    public static void main(String[] args) {
        // Käytetään taas apuohjelmaa scannerin luomiseen
        Scanner bufferedScanner = constructScanner();
        System.out.print("Please give a number: ");

        // Haetaan parsittava numero merkkijonona: tämä ei voi vielä aiheuttaa virhetilaa
        String receivedString = bufferedScanner.nextLine();

        // Try-osion koodi kannattaa pyrkiä pitämään mahdollisimman lyhyenä, jotta vain toivotut
        // virhetilat jäävät kiinni catch-lauseessa
        try {
            int receivedNumber = Integer.parseInt(receivedString);
            System.out.println("Received number " + receivedNumber);

        // Catch-osiossa kannattaa määritellä kiinni otettavan virheen tyyppi mahdollisimman
        // tarkasti, jotta lause ei nappaisi ylimääräisiä virhetiloja
        } catch (NumberFormatException exception) {
            System.out.println("Could not convert '" + receivedString + "' into a number!");
        } finally {
            // Jaettu siivouskoodi menisi tänne
            System.out.println("Cleaning up...");
        }
        System.out.println("Thank you for using the program!");
    }

    private static Scanner constructScanner() {
        InputStreamReader defaultInputReader = new InputStreamReader(System.in);
        BufferedReader bufferedReader = new BufferedReader(defaultInputReader);
        return new Scanner(bufferedReader);
    }
}
```

FileReader ja FileWriter

FileReader ja FileWriter ovat merkkijonomuotoisten tiedostojen käsittelyyn tarkoitettuja apuriluokkia. FileReader sisältää read-metodin datan lukemiseen ja FileWriter sisältää write- ja append -

metodit datan kirjoittamiseen. Sekä lukeminen, että kirjoittaminen voi epäonnistua ohjelmistosta riippumattomista syistä. Tästä syystä nämä operaatiot on järkevää sulkea edellä esitellyn try-rakenteen sisään.

Syötteen lukemisen tapaan, tiedostojen lukeminen suoraan lukijalla tapahtuu merkki kerrallaan. On siis järkevää sulkea FileReader BufferedReader-instanssin sijaan, kuten kappale Syötteen vastaanottamisen perusteet esittelee. Seuraava esimerkki esittelee tekstimuotoisen datan käsittelemisen tiedostoissa luomalla ensin tiedoston ja tallentamalla merkkijonon tähän tiedostoon ja lukemalla sen jälkeen tallennetun merkkijonon tiedostosta BufferedReader-instanssin sisään kiedotulla FileReader-instanssilla.

Koodiesimerkki 37: week4: Merkkijonodatan käsittely tiedostoissa

```
package week4;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class BasicFileWriterReader {

    public static void main(String[] args) {

        // Suljetaan lukeminen ja kirjoittaminen erillisiin try-catch -rakenteisiin, jotta virheet
        // on helpointa erottaa toisistaan
        try {
            FileWriter fileWriter = new FileWriter("BasicFileWriterReader.txt");
            fileWriter.write("Test");
            fileWriter.close();
        } catch (IOException exception) {
            System.out.println("An exception occurred while writing the file!");
        }

        try {
            // Suljetaan tiedoston lukeminen BufferedReader-luokan sisään, jotta lukuoperaatio
            // voidaan suorittaa rivi kerrallaan, sen sijaan että se pitäisi suorittaa merkki
            // kerrallaan.
            FileReader fileReader = new FileReader("BasicFileWriterReader.txt");
            BufferedReader bufferedReader = new BufferedReader(fileReader);
            String fileContent = bufferedReader.readLine();
            System.out.println(fileContent);
        }  catch (IOException exception) {
            System.out.println("An exception occurred while reading the file!");
        }
    }
```

```
}
```

FileInputStream ja FileOutputStream

TIEDOSTOT EIVÄT aina sisällä merkkimuotoista dataa. Tätä varten on olemassa luokat FileInputStream ja FileOutputStream. Ne käsittelevät tiedostoon suljettua tavumuotoista dataa samaan tapaan kuin InputStreamReader lukee käyttäjäsyötteestä tavumuotoista dataa. Toisin kuin FileReader, ei FileInputStream-luokkaa tarvitse kääriä BufferedReader-luokan sisään. Seuraava koodiesimerkki esittelee tavumuotoisen datan käsittelyn tiedostossa.

Koodiesimerkki 38: week4: Tavumuotoisen datan käsittely tiedostoissa

```
package week4;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Arrays;

public class BasicFileInputOutputStream {

    public static void main(String[] args) {

        // Toteutetaan kirjoittamiselle ja lukemiselle erillinen virheenkäsittely edellisen
        // esimerkin tapaan.
        try {
            FileOutputStream fileOutputStream =
                    new FileOutputStream("BasicFileInputOutputStream.");
            fileOutputStream.write("Test".getBytes());
            fileOutputStream.close();
        } catch (IOException exception) {
            System.out.println("An exception occurred while writing the file!");
        }

        try {
            // Huomioi BufferedReader-luokan käyttämättä jättäminen
            FileInputStream fileInputStream =
                    new FileInputStream("BasicFileInputOutputStream");
            // Tavumuotoisen datan haku tiedostosta tavutaulukkoon
            byte[] fileContent = fileInputStream.readAllBytes();
            // Muutetaan taulukko tekstiksi tulostamista varten. Tämä ei tulosta tekstiä 'Test',
            // vaan kirjainten tavurepresnetaatiot. Tulostaa: '[84, 101, 115, 116]'
            System.out.println(Arrays.toString(fileContent));
        }  catch (IOException exception) {
            System.out.println("An exception occurred while reading the file!");
        }
```

```
    }
}
```

Oliopohjainen suunnittelu

Oppaassa on nyt käyty läpi suurin osa Javan perusominaisuuksista. Kurssin tavoitteena ei ole kuitenkaan opettaa opiskelija ohjelmoimaan Javaa, vaan opettaa opiskelijalle olio-ohjelmoinnin käytänteet. Tämän luvun tavoitteena on avata olio-ohjelmoinnin mahdollistamia, sekä muita sille yleisiä suunnittelumalleja. Tällä tavalla luku pyrkii antamaan alkuoppaan vakaan käytännön pohjan lisäksi laajan teoriapohjan olio-ohjelmointia aloittevalle opiskelijalle.

Aggregaatio ja kompositio

Oliopohjaisen suunnittelun osana joudutaan usein puhumaan luokkien välisistä suhteista, erityisesti, miten luokat viittaavat toisiin luokkiin ja miten näiden luokkien instanssien elinkaaret ohjelmassa eroavat toisistaan. Tätä kuvaavat käsitteet *aggregaatio (aggregation)* ja *kompositio (composition)*. Molemmat ovat omistussuhteita, eli suhteita, joissa luokan A instanssi viittaa luokan B instanssiin "omistaen" tämän.

Aggregaatio

Aggregaatio on luokkien välinen omistussuhde, jossa luokan A instanssi omistaa luokan B instanssin väliaikaisesti. Omistettu instanssi siis luodaan muualla ja ei tuhoudu omistavan instanssin mukana. Tällainen rakenne on yleistä kokoelmapohjaisissa rakenteissa. Esimerkiksi mallinnettaessa linja-autoa oliopohjaisilla käsitteillä, linja-auto -luokan ja matkustaja-luokan välinen suhde olisi aggregaatio: matkustaja on olemassa ennen suhteen luomista ja ei tuhoudu suhteen tuhouduttua.

Kompositio

Kompositio on luokken välinen omistussuhde, jossa luokan A instanssi omistaa luokan B instanssin pysyvästi, niin, että instanssien elinajat on sidottu toisiinsa. Luokan B instanssi luodaan sen omistavan luokan A instanssin mukana ja tuhoutuu tämän instanssin kanssa. Täten luokan B instansseja ei voi olla olemassa erillään

ne omistavasta luokan A instanssista, eikä luokan A instansseja ilman niiden omistamia luokan B instansseja. Tälläinen suhde on yleistä suurien komponenttien erillisessä luokassa esitettyjen osien kanssa. Esimerkiksi aggregaation yhteydessä annetussa linja-auto -esimerkissä linja-auto -luokan ja moottori-luokan välinen suhde on kompositiosuhde. Moottori luodaan linja-auton kanssa ja tuhoutuu kun linja-auto tuhoutuu. Kompositiossa siis omistettu luokka on vain yksi rakennuspalikka omistavan luokan osana.

Periytymisen alkeet

YKSI LAADUKKAAN ohjelmiston tärkeimmistä piirteistä on turhan toiston poistaminen koodista. Tarkastellaan esimerkkinä pankin tilinhallintaa. Pankit tarjoavat monesti erilaisia tilejä eri käyttötarkoituksiin. Käyttötili, säästötili ja yrityksen tili kaikki todennäköisesti tarvitsevat eri tietokenttiä ja metodeja täyden toiminnallisuutensa toteuttamiseen. Kun jokaiselle näistä kolmesta tilityypistä luodaan oma luokkansa huomataan kuitenkin luokkien sisältävän itseään toistavaa koodia. Kaikilla luoduilla luokilla on oltava tietokentät tilinumerolle, saldolle ja omistajan kontaktitiedoille. Lisäksi jokaiselta luokalta löytyy todennäköisesti metodit tilin saldon tarkastamiseen ja panon ja noston suorittamiseen. Turhan toiston eliminoimiseksi tarvitaan siis rakenne, joka antaa luokkien käyttää samaa pohjaa, mutta muokata omaa toiminnallisuuttaan tämän pohjan päällä. Tätä varten oliopohjaiset kielet tarjoavat ominaisuuden, nimeltä *periytyminen (inheritance)*

PERIYTYMISEN PERUSIDEA on mahdollistaa luokkien toiminnallisuuksien uudelleenkäyttö ja laajentaminen. Tämä tapahtuu luomalla niin sanottu *lapsiluokka (child class)* laajennettavasta luokasta. Tällä tavalla laajennetun luokan sanotaan olevan luodun lapsiluokan *kantaluokka (parent class)*.

LAPSILUOKKA PERII kaikki kantaluokan ominaisuudet ja kykenee muokkaamaan niitä tai lisäämään uusia, omia ominaisuuksiaan. Esimerkiksi pankkiohjelmistoa luotaessa voitaisiin kaikki erilaiset tilit toteuttaa luomalla normaalia käyttötiliä kuvaava Account-niminen kantaluokka. Tämän jälkeen säästötili ja yritystili voidaan esittää tästä kantaluokasta periytyvinä lapsiluokkina. Kantaluokka voi sisältää esimerkiksi tietokentät tilinumerolle, tilin saldolle ja tilin omistajan kontaktitiedoille. Näiden lisäksi kantaluokalla voi olla metodeja. Account-luokka esimerkiksi voi sisältää metodin nostoille ja panoille, sekä metodin tilin saldon tarkastamiseen. Lapsiluokat voivat tarpeen vaatiessa ylikirjoittaa näitä metodeja vastaamaan omaa toimintaansa. Esimerkiksi säästötilin nostometodiin voidaan lisätä ehto, jossa sitä voi käyttää vain tietyin väliajoin. Myös uusien kenttien ja metodien lisääminen lapsiluokkaan on mahdollista. Esimerkiksi yritystilille voidaan lisätä tietokenttä tallentamaan yrityksen nimi ja edellä

mainitulle säästötilille voidaan lisätä metodi kerryttämään korkoa.

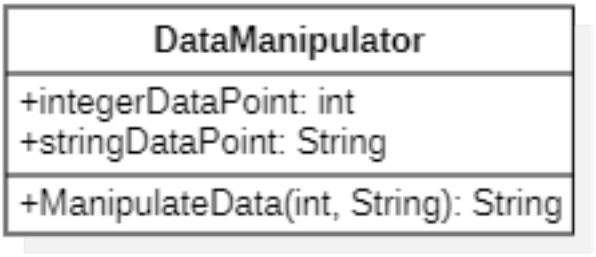
UML

VÄLILLÄ OHJELMISTON suunnittelun yhteydessä ilmenee tarve esittää ohjelmiston luokkien välisiä suhteita tarkemmin ja selkeämmin kuin sanallisesti on mahdollista. Tätä varten on olemassa *UML (Unified Modeling Language)*. Se on ohjelmistotuotantoa varten kehitetty standardi, joka sisältää useita ohjelmistojen suunnittelussa käytettäviä kaavioita. Tässä oppaassa käsitellään näistä vain *luokkakaavio (class diagram)*, sillä se on ehdottomasti tärkein kaaviotyyppi oliopohjaisen ohjelman suunnittelussa. Muita UML-standardissa määriteltyjä kaaviotyyppejä ovat muun muassa oliokaavio, tilakaavio ja sekvenssikaavio.

Luokkakaavio

UML-STANDARDIN TAPA ilmaista luokkien välisiä suhteita ohjelmistossa on luokkakaavio. Kaavio suunniteltiin alun perin tarpeeksi tarkaksi koodin generointiin kaavion pohjalta, mutta nykyään kaavion yleisimmin hyväksytty käyttötapa on suunnitteluvaiheessa ideoiden kommunikointi ohjelmiston suunnittelijoiden välillä.

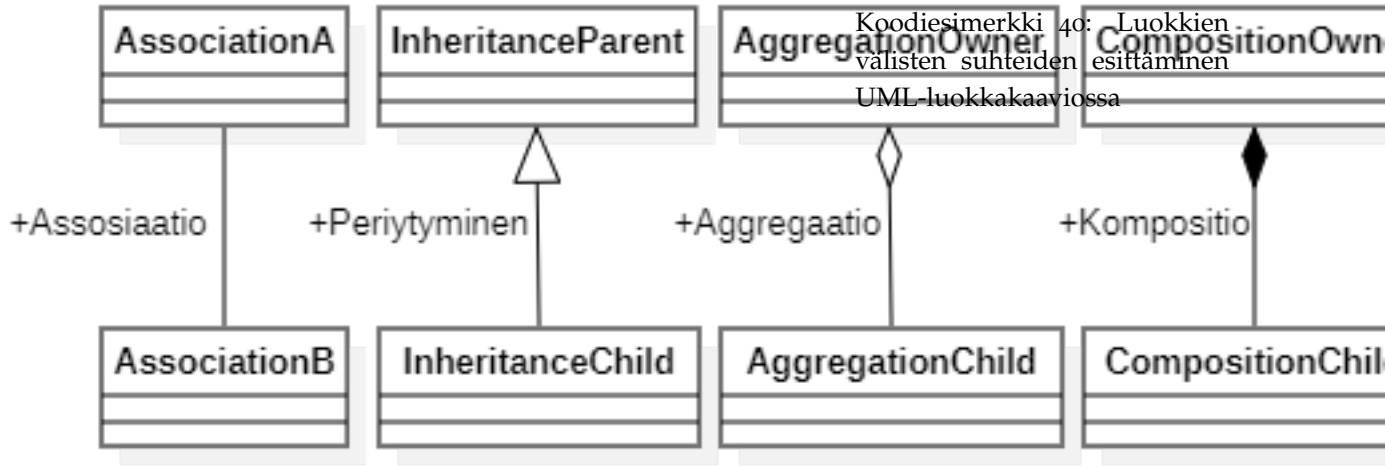
LUOKKAKAAVIO KOOSTUU luokkia kuvaavista suorakulmion muotoisista soluista ja niiden välisistä yhteyksistä. Solu jakautuu yleensä kolmeen osaan. Ylimpänä on solun kuvaaman luokan nimi, sen jälkeen luokan tietokentät ja alimpana luokan metodit. Tietokenttien ja metodien yhteyteen voidaan lisätä tyyppidata, mutta se ei ole pakollista ja riippuu kaavion käyttötarkoituksesta. Seuraava kuva esittää yksinkertaisen luokan kuvauksen luokkakaaviossa. Luokan nimi on DataManipulator, se omistaa int-tyypin tietokentän integerDataPoint ja String-tyypin tietokentän stringDataPoint sekä metodin ManipulateData, joka ottaa int- ja String-tyyppien muuttujat ja palauttaa String-tyypin muuttujan.



Koodiesimerkki 39: Yksittäisen luokan esittäminen UML-luokkakaaviossa

LUOKKAKAAVIOSSA OHJELMISTON erilaisten luokkien esittäminen on kuitenkin vain pieni osa kaavion käyttöä. Tärkein luokkakaavion

käyttötarkoitus on luokkien välisten suhteiden esittäminen ohjelmistossa. Luokkakaavio sisältää näiden suhteiden kuvaamiseen erilaisia yhteystyyppiä, joita luokkien välille voidaan piirtää. Näistä käytetyimmät ovat yleistä assosiaatiota, periytymistä, aggregaatiota ja kompositiota kuvaavat yhteydet. Yhteyden tarkka tyyli riippuu kaavionpiirto-ohjelmasta. Yleensä assosiaatiota kuvaava yhteys on yksinkertainen viiva luokkien välillä, periytymistä kuvaava yhteys on nuoli lapsiluokasta emoluokkaan ja aggregaatiota ja kompositiota kuvaavat yhteydet ovat viivoja, jotka päättyvät vinoneliöön omistussuhteen omistavan osapuolen päässä. Aggregaatiossa vinoneliö on täyttämätön, kompositiossa täytetty. Seuraava kaavio esittelee kaikki neljä yhteystyyppiä.



Koodiesimerkki 40: Luokkien välisten suhteiden esittäminen UML-luokkakaaviossa

Kaavioiden käyttö ohjelmiston suunnittelussa

UML KEHITETTIIN alun perin aikakaudella, jolloin vesiputousmallia pidettiin erinomaisena vaihtoehtona ohjelmistoprojektin projektinhallintamalliksi ja ketterä kehitys oli täysin tuntematon käsite. Tällöin ohjelmiston täydellisen rakenteen mallintamisen UML-kaavioilla ajateltiin olevan järkevä tapa suunnitella ohjelmistoja. Modernissa ohjelmistokehityksessä vesiputousmalli on kuitenkin pudonnut pois suosiosta ja se on korvattu lähinnä ketterän kehityksen innoittamilla projektinhallintametodeilla, kuten Scrum ja XP. Täten myös UML-kaavioiden rooli osana ohjelmitonkehitysprosessia on modernisoitunut. Yleinen vallitseva käsitys alalla on, että kaavioiden ei pitäisi olla yleisessä käytössä, vaan niitä kuuluisi käyttää vain nopeaan ohjelmiston mallintamiseen erityisesti tehtäessä yhteistyötä usean kehittäjän kesken.

Periytyminen ja Java

PERIYTYMINEN KÄSITTEENÄ käytiin läpi viime luvussa, mutta sen soveltaminen käytännön ohjelmistotuotantoon jätettiin käsittelemättä. Tämän luvun tavoitteena on opettaa periytymisen käyttö Javassa ja samalla syventyä periytymisen hyötyihin ja haittoihin, sen oikeaoppisiin käyttökohteisiin ja teoriaan periytymisen takana.

Periytymisen perusteet ja extends

Avainsana extends

JAVASSA PERIYTYMINEN toteutetaan *extends*-avainsanan avulla. Avainsana sijoitetaan lapsiluokan määritelmään luokan nimen perään, niin että toivotun kantaluokan nimi seuraa avainsanaa. Tämän jälkeen luokan määritelmää jatketaan normaalisti. Seuraava koodiesimerkki esittelee yksinkertaisen periytymiseen pohjaavan rakenteen luomisen Javassa.

Koodiesimerkki 41: week6/basicinheritanceexample: Periytymisesimerkin kantaluokka

```
package week6.basicinheritanceexample;

// Raa'an kantaluokan ei tarvitse tietää, että siitä periytyy luokkia. Täten kantaluokan määritelmä
// voi yksinkertaisimillaan näyttää täysin normaalilta luokalta. Tämä tarkoittaa myös, että
// periyttäminen voidaan tehdä luokasta, jota ei ole kirjoitettu alun perin kantaluokaksi
public class Parent {
    private int dataField = 10;

    public int getDataField() {
        return dataField;
    }
}
```

Koodiesimerkki 42: week6/basicinheritanceexample: Periytymisesimerkin lapsiluokka

```
package week6.basicinheritanceexample;
```

```
// Huomioi, että lapsiluokka saa esimerkissä toiminnallisuudet kantaluokalta, vaikka lapsiluokan
// runko ei sisällä koodia.
public class Child extends Parent {
}
```

Koodiesimerkki 43: week6/basicinheritanceexample: Periytymisesimerkin pääluokka

```
package week6.basicinheritanceexample;

public class BasicInheritanceExample {

    public static void main(String[] args) {
        // Koska Child periytyy parent-luokasta, Java näkee Child-instanssin myös Parent-instanssina
        Parent dataProvider = new Child();
        System.out.println("Data field: " + dataProvider.getDataField());
    }
}
```

Kantaluokan toiminnallisuuksien uudelleenmäärittely

Aina ei ole toivottavaa, että lapsiluokka perii kaiken kantaluokan toiminnallisuuden. Tätä varten Java tarjoaa mahdollisuuden uudelleenmäärittellä periytettyjen luokkien perimiä metodeja. Tämä metodin *korvaaminen (overriding)* tapahtuu yksinkertaisesti määrittelemällä lapsiluokassa metodi, jonka signatuuri on tismalleen sama, kuin kantaluokan korvattavan metodin. Tällöin kutsuttaessa lapsiluokan metodia kyseisellä signatuurilla ajetaan lapsiluokassa määritelty versio metodista. Seuraava koodiesimerkki esittelee metodin toiminnan korvaamisen käytännössä.

Koodiesimerkki 44: week6/overridingexample: Korvaamisesimerkin kantaluokka

```
package week6.overridingexample;

public class Parent {
    public String addText(String text) {
        return text + "Parent";
    }
}
```

Koodiesimerkki 45: week6/overridingexample: Korvaamisesimerkin lapsiluokka

```
package week6.overridingexample;

public class Child extends Parent {
    // Metodia korvattaessa on tärkeää huolehtia, että korvaavan metodin signatuuri on identtinen
```

```
    // suhteessa korvattavaan metodiin. Koska signatuuri huomioi vain argumenttien tietotyypit,
    // voi argumenttien nimeä vaihtaa vapaasti.
    //
    // @Override-ei ole pakollinen ylikirjoitettaessa kantaluokan metodia, mutta sen käyttäminen
    // helpottaa luettavuutta ja varmistaa että ohjelmisto ei kirjoitusvirheen vuoksi määrittele
    // uutta metodia, vaan antaa kääntäjävirheen virheellisesti ylikirjoitetusta metodista. Oppaan
    // koodiesimerkit käyttävät aina @Override-attribuuttia metodia ylikirjoitettaessa.
    @Override
    public String addText(String childText) {
        return childText + "Child";
    }
}
```

Koodiesimerkki 46: week6/overridingexample: Korvaamisesimerkin pääluokka

```
package week6.overridingexample;

public class OverridingExample {
    public static void main(String[] strings) {
        Parent parentClass = new Parent();
        Parent childClass = new Child();
        System.out.println(parentClass.addText("Called from "));
        System.out.println(childClass.addText("Called from "));
    }
}
```

Periytyminen ja rakentajat

Näkyvyysmääre protected

AIEMMIN OPPAAN kappaleessa Näkyvyysmääreet käytiin läpi käsite näkyvyysmääre ja avainsanat public ja private. Näiden ja oletusnäkyvyyden lisäksi Javassa on olemassa neljäs näkyvyysmääreavainsana *protected*. Avainsana määrittää muuttujan tai metodin olevan näkyvissä vain luokalle itselleen, sekä kaikille luokan periville luokille. Näkyvyysmääreen private omaava muuttuja tai metodi ei siis näy edes luokan periville luokille.

Periytyminen, rakentajat ja super

LUOTAESSA LAPSILUOKAN instanssia Javassa on aina kutsuttava kantaluokan rakentajaa. Mikäli kummallekaan luokalle ei ole määritelty rakentajaa, kutsuu Java automaattisesti kantaluokan oletusrakentajaa rakentaessaan lapsiluokan instanssia. Mikäli lapsiluokalla ei ole rakentajaa, mutta kantaluokalle halutaan määrittää rakentaja, jota lapsiluokan rakennus käyttää, on tämän rakentajan oltava parametritön.

Mikäli lapsiluokalle luodaan itse määritelty rakentaja, on rakentajan kutsuttava ensimmäisenä komentona jotain emoluokan rakentajaa. Tämän ei tarvitse olla parametritön oletusrakentaja vaan mikä tahansa rakentaja kelpaa.

Päästäkseen käsiksi kantaluokkansa kenttiin, kuten rakentajiin, voi lapsiluokka käyttää *super*-avainsanaa. Avainsana toimii viittauksena kantaluokan instanssiin, jonka pohjalle lapsiluokka on luotu. Käyttämällä avainsanaa metodikutsuna päästään käsiksi kantaluokan rakentajametodin eri ylikuormituksiin ja käyttämällä pistenotaatiota päästään käsiksi kantaluokan muuttujiin ja metodeihin näkyvyysmääreiden salliessa. Seuraavat koodiesimerkit esittelevät rakentajien käytön periytymisen yhteydessä, sekä avainsanat super ja protected

Koodiesimerkki 47: week6/inheritancewithconstructors: Rakentajaton kantaluokka

```
package week6.inheritancewithconstructors;

public class ParentWithoutConstructor {

    protected int x = 1;
    protected int y = 1;

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
}
```

Koodiesimerkki 48: week6/inheritancewithconstructors: Rakentajattoman kantaluokan lapsiluokka, jolle on määritelty rakentaja

```
package week6.inheritancewithconstructors;

public class ChildOfConstructorless extends ParentWithoutConstructor {

    public ChildOfConstructorless(int x, int y) {
        super();
        this.x = x;
        this.y = y;
    }
}
```

Koodiesimerkki 49: week6/inheritancewithconstructors: Kantaluokka, jolla on rakentaja

```java
package week6.inheritancewithconstructors;

public class ParentWithConstructor {

    private int x;
    private int y;

    public ParentWithConstructor(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }
}
```

Koodiesimerkki 50: week6/inheritancewithconstructors: Rakentajallisen kantaluokan lapsiluokka, jolle on määritelty erillinen rakentaja

```java
package week6.inheritancewithconstructors;

public class ChildOfConstructorParent extends ParentWithConstructor {

    public ChildOfConstructorParent(int x, int y) {
        super(x + 2, y + 2);
    }
}
```

Periytymisen oikeaoppinen käyttö

PERIYTYMINEN ON monimutkainen rakenne joka tarjoaa ohjemoijalle voimakkaan työkalun. Tätä työkalua on helppo käyttää väärin mikäli periytymistä lähdetään käyttämään kaikkialla ja siitä yritetään tehdä ratkaisu joka ongelmaan. Periytymiselle on oma paikkansa ohjelmiston suunnittelussa ja se ratkaisee jotkin ongelmat erinomaisesti, mutta on olemassa tiettyjä lainalaisuuksia, jotka on pidettävä mielessä periytymistä hyödynnettäessä. Näin varmistetaan, että ohjelmiston kehittäminen helpottuu periytymisrakenteiden vaikutuksesta, sen sijaan että kehittämisprosessia monimutkaistettaisiin turhaan ilman selkeitä hyötyjä periytymisestä.

Liskovin korvaavuusperiaate

KUTEN KAPPPALEEN Periytyminen, rakentajat ja super koodiesimerkeissä nähtiin, voidaan lapsiluokan olio tallentaa muuttujaan, jonka tyypiksi on määritelty emoluokan olio. Tällöin muuttujaan tallennettua olioita voidaan kohdella kuin emoluokan oliota. Tämä käyttäytyminen mahdollistaa monia hyödyllisiä toiminnallisuuksia, kuten monen eri lapsiluokan instanssien tallentamista samaan ArrayList- instanssiin, kunhan kyseinen ArrayList-instanssi on luotu viitaten kantaluokkaan, eikä lapsiluokkaan.

KUITENKIN TÄLLAINEN luokkien käyttäminen luo velvotteita lapsiluokkien suunnitteluun. Jotta edellä kuvailtu käyttötapa sujuisi ongelmitta, kuuluu lapsiluokkien ja emoluokan toteuttaa *Liskovin korvaavuusperiaate (Liskov Substitution Principle - LSP)*. Periaate on osa viiden periaatteen muodostamaa *SOLID-periaatteet (SOLID-principles)* -ryhmää. Periaatteet on suunniteltu helpottamaan laadukkaan oliopohjaisen koodin tuottamista ja loput periaatteet käsitellään myöhemmin luvussa SOLID-periaatteet.

LISKOVIN KORVAAVUUSPERIAATE määrittelee kantaluokan ja lapsiluokkien suhteen seuraavasti: "Mikä tahansa emoluokan instanssi pitäisi olla korvattavassa millä tahansa lapsiluokan instanssilla ilman että ohjelman toiminta häiriintyy." Käytännössä tämä tarkoittaa, että lapsiluokkien ei pitäisi ylikirjoittaa emoluokkien julkisia kenttiä niin että kentän toiminnallisuus ulkoa tarkasteltaessa muuttuu. Esimerkiksi metodin ylikirjoittaminen toimimattomalla tyngällä rikkoo täten Liskovin korvaavuusperiaatetta.

Periytymisen rakenteita

TÄHÄN MENNESSÄ oppaassa on käsitelty vain suoraa periytymistä, jossa kantaluokka ja lapsiluokka ovat molemmat normaaleja Javan luokkia. Periytyminen on kuitenkin rakenne, jota käytetään monimutkaisten teollisten ohjelmien suunnittelussa ja täten suurimpaan osaan oliopohjaisia kieliä on kehitetty rakenteita periytymiseen pohjaavan arkkitehtuurin siistimiseen. Näistä tärkeimmät ovat *abstrakti luokka (abstract class)* ja *rajapinta (interface)*. Tämä kappale käy läpi näiden rakenteiden käytön koodissa ja niiden käyttötarkoitukset, sekä puhuu hieman rakenteiden eroista.

Abstrakti luokka

ABSTRAKTI LUOKKA on oliopohjainen rakenne, jossa luokan määritelmästä tehdään pelkkä muotti, jonka ainoa tehtävä on toimia pohjana kyseisestä luokasta periytyville lapsiluokille. Tällaisen rakenteen tehtävänä on varmistaa että jokainen luokka, joka periytyy abstraktista kantaluokasta toteuttaa jonkin jaetun toiminnallisuuden ja samalla estää kantaluokan suora käyttäminen koodissa, koska sen ei kuulukaan

tietää miten tämän toiminnallisuuden implementaatio tapahtuu. Luokasta tehdään abstrakti lisäämällä *abstract*-avainsana luokan määritelmään class-avainsanan eteen. Abstraktin luokan ei-abstraktin lapsiluokan nimi on *konkreettinen luokka (concrete class)*.

JOKAISELLA ABSTRAKTILLA luokalla on oltava vähintään yksi *abstrakti metodi (abstract method)*. Tämä metodi määritellään luomalla metodi, joka sisältää pelkän signatuurin ilman runkoa ja lisäämällä metodin määritelmään abstract -avainsana näkyvyysmääreen jälkeen. Abstrakti metoodi on siis ikään kuin musta laatikko. Sen argumentit ja paluuarvo tiedetään, mutta abstrakti kantaluokka, joka määrittelee metodin ei kerro, miten metodeista saadaan paluuarvo vaan jokaisen konkreettisen lapsiluokan on määriteltävä tämä itse.

ABSTRAKTIN LUOKAN käyttö koodissa vaatii yleensä jonkin jaetun toiminnallisuuden tarpeen toteamista ja tämän toiminnallisuuden eristämistä abstraktiin luokkaan. Seuraava koodiesimerkki esittelee abstraktin kantaluokan ja konkreettisen lapsiluokan luomisen. Esimerkissä luodaan "AbstractLogger"-kantaluokka, joka sisältää protected-metodin "formatLogString" ja abstraktin julkisen metodin "logString". Sen jälkeen kantaluokasta tehdään konkreettinen lapsiluokka "CommandLineLogger", joka sisältää määritelmän kantaluokan "logString"-metodille. Huomaa, kuinka lapsiluokka pystyy käyttämään "formatLogString"-metodia, joka on määritelty kantaluokassa.

Koodiesimerkki 51: week6/abstractexample: Abstrakti loggaajapohja

```
package week6.abstractexample;

public abstract class AbstractLogger {

    // Tässä metodissa voitaisiin lisätä esimerkiksi loggaustapahtuman ajankohta, esimerkin vuoksi
    // metodi on yksinkertaistettu
    protected String formatLogString(String logString) {
        return "LOGGED: " + logString;
    }

    // Abstraktista metodista määritellään vain signatuuri
    public abstract void logString(String logString);
}
```

Koodiesimerkki 52: week6/abstractexample: Loggaajaan konkreettinen implementaatio, joka tulostaa saadun tekstidatan komentoriville

```
package week6.abstractexample;

public class CommandLineLogger extends AbstractLogger {

    @Override
```

```
    public void logString(String logString) {
        System.out.println(this.formatLogString(logString));
    }
}
```

Rajapinta

TOINEN TAPA eristää luokkien jakamaa toiminnallisuuttaa koodissa on rajapintaluokka, joka luodaan interface-avainsanalla. Rajapintaluokan tehtävä on luokan nimen mukaisesti määrittää jokin rajapinta jonka muut ohjelmiston luokat voivat sitten ilmoittaa täyttävänsä. Toisin kuin abstrakti luokka, rajapintaluokka voi sisältää ainoastaa julkisia ja staattisia vakiotietokenttiä, eikä rajapintaluokkaa voi käytää kantaluokkana periytymisessä. Tämän sijaan mikä tahansa ohjelmiston luokka voi ilmoittaa, että se *toteuttaa (implement)* rajapintaluokan. Luokan, joka toteuttaa rajapintaluokan on määriteltävä jokainen rajapinnassa määritelty metodi. Näistä määritelmistä jokaisella pitää olla identtinen signatuuri rajapintaluokan vastaavan metodin määritelmän kanssa.

RAJAPINTALUOKKA MÄÄRITELLÄÄN Javassa korvaamalla normaalin luokan määritelmän class-avainsana *interface*-avainsanalla. Rajapintaluokkasta ei voi luoda instansseja, eikä luokalle voi määrittää rakentajaa. Ainoat tietokentät, jotka ovat hyväksyttyjä rajapintaluokassa ovat oletuksena näkyvyydeltään julkisia luokkamuuttujavakioita. Mikäli muuttujan määritelmä rajapintaluokassa ei sisällä avainsanoja public, static ja final lisätään ne määritelmään automaattisesti. Muiden näkyvyysmääreiden käyttäminen estää ohjelmiston rakentamisen.

RAJAPINTALUOKAN KAIKKI metodit ovat automaattisesti abstrakteja metodeja. Myös metodien on oltava rajapintaluokassa julkisia muuttujien tapaan. Java tukee metodin oletustoiminnallisuuden määrittelemistä *default*-avainsanalla, mutta mikäli ohjelmiston kehitysvaiheessa huomataan tälle olevan tarvetta kannattaa miettiä tarkkaan, käytetäänkö rajapintaluokkaa oikein; Avainsana on olemassa lähinnä mahdollistamaan myöhemmät laajennukset yleisesti käytettyihin rajapintoihin eikä siihen kannata tukeutua. Tästä syystä oppaassa ei myöskään käydä avainsanan käyttöä tämän tarkemmin läpi.

SEURAAVA KOODIESIMERKKI esittelee simppelin "TextProvider"-rajapinnan luomisen ja kyseisen rajapintaluokan implementoimisen konkreettiseen luokkaan.

Koodiesimerkki 53: week6/interfaceexample: TextProvider rajapintaluokka

```
package week6.interfaceexample;

public interface TextProvider {
```

```
    // Kaikki rajapintaluokan metodit ovat automaattisesti julkisia ja abstrakteja
    String getText();
}
```

Koodiesimerkki 54: week6/interfaceexample: Luokka, joka implementoi TextProvider-rajapinnan

```
package week6.interfaceexample;

public class FunctionalClass implements TextProvider {

    private String text;

    public FunctionalClass(String text) {
        this.text = text;
    }

    @Override
    public String getText() {
        return text;
    }
}
```

Rajapinta vai abstrakti luokka?

RAJAPINTALUOKKA JA abstrakti luokka muistuttavat toisiaan toiminnallisuuksiltaan ja osin käyttötavoiltaankin. Niille molemmille löytyvät kuitenkin omat erilliset käyttökohteensa ja syvempi ymmärrys luokkatyyppien toiminnallisuuksien eroista auttaa käsittämään nämä erilaiset käyttökohteet.

KOSKA LUOKKIEN *moniperinta (multiple inheritance)* ei ole tuettua Javassa, ei abstrakteja luokkia voi käyttää lisäämään yhdelle luokalle useamman eri luokan toiminnallisuuksia. Tällainen on rajapintaluokkien tehtävä. Sen sijaan abstrakti luokka voi määritellä valmiiksi toiminnallisuuksia ja datakenttiä itselleen, helpottaen lapsiluokkien kirjoittamista. Abstraktin luokan pääkäyttötarkoitus onkin luoda koodiin jokin palvelu, kuten edellä sivuttu logien kirjoittaminen, jonka yleinen toiminnallisuus on tiedossa, mutta toteutus riippuu käytettävästä kirjastosta. Esimerkiksi tässä logiesimerkissä, voitaisiin luoda abstrakti luokka, jolle implementoidaan valmiiksi funktiot logien muotoiluun, mutta jonka loginkirjoitusfunktiot määritellään abstrakteina, jolloin kutakin logikirjastoa käyttävä abstraktin luokan implementaatio määrittelee kirjastoon nojaavan rajapintansa itse.

RAJAPINTALUOKKIEN TEHTÄVÄ puolestaan on yleensä luoda pieniä, selkeitä toiminnallisuussopimuksia koodissa. Mikäli myöhemmin ilmenee tarve lisätä johonkin luokkaan jo valmiiksi määritellyn rajapinnan toiminnallisuudet, voidaan luokan määritelmään lisätä tämä

rajapinta implements-avainsanalla. Esimerkiksi pelinkehityksessä saatetaan tarvita tuhoutuvia objekteja, joita varten voitaisiin luoda "Destructable"-rajapintaluokka jolle on määritelty "destroyObject"-metodi. Koodi voi sitten tarkistaa implementoiko jokin luokka tämä rajapinnan *instanceof*-avainsanalla ja jos luokka implementoi "Destructable"-rajapinnan, koodi voi kutsua "destroyObject"-metodia luokan instanssista tuhoten objektin.

RAJAPINTALUOKAN JA abstraktin luokan käyttötarkoitukset siis eroavat toisistaan hieman. Yleisesti rajapintaluokan tehtävä on määrittää laajemmin käytössä oleva sopimus, joka voidaan implementoida eri kohdissa koodia eri luokille. Tästä hyvä esimerkki on Serialisaatio-kappaleessa käsiteltävä Serializable-rajapintaluokka. Abstrakti luokka taas määrittää yhden toiminnallisuudeltaan rajatun luokkatyypin, mutta antaa tämän toiminnallisuuden toteutusvastuun eteenpäin kantaluokan lapsiluokille. Mahtava esimerkki tästä ovat Minecraft-pelin kuutiot. Jokainen kuutiotyyppi on abstraktin "Block"-luokan konkreettinen lapsiluokka, joka määrittää yksittäisen kuutiotyypin tarkemmat toiminnallisuudet.

OPPAAN KOODIESIMERKEISSÄ tämä käyttötapojen ero näkyy kappaleissa Abstrakti luokka ja Rajapinta. Abstraktin luokan esimerkissä hyödynnetään abstraktin kantaluokan kykyä määritellä konkreettisia metodeja kantaluokkaan: oletettavasti kaikki lapsiluokat hyödyntävät "formatLogString"-metodia. Rajapintaluokkaesimerkissä puolestaan luotu "TextProvider"-rajapinta on vain yksinkertainen rajapinta, jonka mikä tahansa ohjelmiston luokka saattaa toteuttaa tarvittaessa. Abstraktilla luokalla on yksi suunniteltu käyttötarkoitus, jonka toteutustapa määritellään lapsiluokan implementaatiossa, kun taas rajapintaluokan käyttötarkoitus ei ole niin tarkkaan rajattu ja rajapintaa voidaan implementoida useammassa paikassa koodissa.

Serialisaatio

JAVASSA OHJELMISTON omistama data on yleensä tallennettu oliomuodossa, mutta olion tallentaminen tiedostoon tekstidatana ei ole erityisen helppoa. Se vaatisi kaikkien olion kenttien muuttamista tekstiksi manuaalisesti aina oliota tallennettaessa ja tekstimuotoisen datan lukemista ja uuden olion rakentamista luetusta tekstidatasta aina oliota ladattaessa. Tähän tarkoitukseen Javasta löytyy erikseen serialisaatio-niminen tekniikka, joka mahdollistaa olioiden automaattisen muuttamisen tallennettavaan muotoon ja tämän muodon muuttamisen takaisin olioksi.

SERIALISAATIO ON mahdollista vain luokille, jotka implementoivat *Serializable*-rajapinnan. Tämän lisäksi kaikkien luokan muuttujien on myös oltava tietotyyppiä, joka implementoi tämän rajapinnan. Suurimmaksi osaksi tämä koskee vain käyttäjän määrittelemiä lu-

okkia, sillä kaikki primitiiviset tietotyypit ja suurin osa standardikirjaston yleisesti käytetyistä luokista, kuten kokoelmat, implementoivat Serializable-rajapinnan.

JOS LUOKKA ja kaikki luokan muuttujat implementoivat Serializable-rajapinnan, voidaan luokan instanssi muuttaa tekstimuotoon *ObjectOutputStream*-luokan avulla. Luokan julkinen rakentaja ottaa OutputStream-olion, joka edustaa datakuluttajaa, johon serialisoitava objekti kirjoitetaan. Luodusta instanssista voidaan kutsua instanssin writeObject- metodia serialisoitavalla oliolla. Tämä kirjoitaa olion serialisoidun tavudataversion ObjectOutputStream-rakentajassa määriteltyyn OutputStream-olioon.

LUOKAN DESERIALISAATIO tapahtuu *ObjectInputStream*-luokan avulla. Luokan julkinen rakentaja ottaa InputStream-olion, joka edustaa datasyötettä, josta deserialisoitava objekti on luettavissa. Luodusta instassista voidaan kutsua instanssin readObject-metodia ilman argumentteja. Tämä palauttaa deserialisoidun olion luetun tavudatan pohjalta.

SEURAAVA KOODIESIMERKKI esittelee serialisoitavan luokan luomisen, serialisaation tiedostoon ja deserialisaation tiedostosta. Huomioi, että mikäli serialisoituja olioita tallennetaan tiedostoon, on normaalia nimetä tiedosto ".ser"-päätteellä.

Koodiesimerkki 55: week6/serializationexample: Serialisoitava luokka

```
package week6.serializationexample;

import java.io.Serializable;

public class SerializableData implements Serializable {

    // Mikäli muuttujat olisivat käyttäjän määrittelemiä luokkia, olisi pidettävä huolta, että
    // kaikkien muuttujien luokat implementoivat myös Serializable-rajapinnan
    private String stringData;
    private int intData;

    public SerializableData(String stringData, int intData) {
        this.stringData = stringData;
        this.intData = intData;
    }

    public String getDataAsString() {
        return stringData + ", " + intData;
    }
}
```

Koodiesimerkki 56: week6/serializationexample: Serialisaatioesimerkin pääluokka

```java
package week6.serializationexample;

import java.io.*;

public class SerializationExample {

    public static void main(String[] args) {

        SerializableData dataPoint = new SerializableData("Data example", 10);

        // Serialisaatio ja tiedostonkäsittely try-catch -rakenteen sisällä
        try {
            // Huomioi ".ser"-pääte
            FileOutputStream fileOutputStream =
                    new FileOutputStream("serialization_example.ser");
            ObjectOutputStream objectOutputStream = new ObjectOutputStream(fileOutputStream);
            objectOutputStream.writeObject(dataPoint);
            objectOutputStream.close();
            fileOutputStream.close();
        } catch (IOException exception) {
            exception.printStackTrace();
        }

        try {
            FileInputStream fileInputStream =
                    new FileInputStream("serialization_example.ser");
            ObjectInputStream objectInputStream = new ObjectInputStream(fileInputStream);
            SerializableData deSerializedDataPoint = (SerializableData) objectInputStream.readObject();
            System.out.println(deSerializedDataPoint.getDataAsString());
            objectInputStream.close();
            fileInputStream.close();
        } catch (IOException exception) {
            exception.printStackTrace();
        } catch (ClassNotFoundException exception) {
            System.out.println("Could not find the class for deserialization.");
            exception.printStackTrace();
        }
    }
}
```

Singleton-suunnittelumalli

Välillä ohjelmistoa luodessa on tarve luoda luokka, josta on olemassa vain yksi instanssi, jota muu ohjelma voi kutsua tarvittaessa. Tähän käyttötarkoitukseen on kehitetty suunnittelumalli nimeltä *singleton (singleton)*. Suunnittelumallin ideana on tallentaa luokan instanssi yksityiseen luokkamuuttujaan ja tarjota julkinen "getInstance"-luokkametodi, joka joko hakee olemassaolevan instanssin luokkamuuttujasta tai luo uuden instanssin luokkamuuttujaan, mikäli instanssia ei vielä ole olemassa. Seuraava koodiesimerkki esittelee tämän toteutuksen käytännössä.

Koodiesimerkki 57: week8: Singleton-suunnittelumallin toteuttava luokka

```
package week8;

public class SingletonExample {

    private static SingletonExample instance = null;

    // Luomalla luokalle tyhjä yksityinen rakentaja varmistetaan, ettei muu koodi voi luoda
    // luokan instansseja käyttämättä getInstance-metodia
    private SingletonExample() {
    }

    public static SingletonExample getInstance() {
        if (instance == null) {
            instance = new SingletonExample();
        }
        return instance;
    }
}
```

Singleton-suunnittelumallia on kritisoitu, sillä se luo ohjelmistolle jaetun julkisen tilan toimien käytännössä glorifioituna globaalina muuttujana. Mallille on kuitenkin olemassa selkeitä hyödyllisiä käyttökohteita, kuten tietokantayhteyden standardointi. Nämä käyttökohteet ovat yleensä jaettuja palveluita, jotka ovat joko lähes tai täysin tilattomia. Vahvasti tilallisen olion tallentaminen singleton-luokkaan on valtava virheriski, koska olioon pääse käsiksi mistä tahansa kood-

issa ja siten sen tila on erittäin helppo muuttaa epähuomiossa virheelliseksi.

SOLID-periaatteet, käyttäjän määrittelemät virheluokat

SOLID-periaatteet

Tässä vaiheessa opasta oppaan käyttäjä osaa jo toivottavasti tuottaa toimivaa oliopohjaista koodia. Ikävä kyllä toimiva koodi ei tarkoita suoraan laadukasta koodia. Laadukas koodi on uusiokäytettävää, helppolukuista ja laajennettavaa. Yksi laadukkaan koodin tuottamista helpottamaan kehitetty menetelmä ovat *SOLID-periaatteet (SOLID-principles)*. Periaatteet eivät ole kiveen hakattuja, mutta niiden noudattaminen tuottaa yleensä vähintään kohtalaisen laadukasta koodia. Periaatteita on viisi: *yhden vastuun periaate (Single Responsibility Principle)*, *avoin/suljettu-periaate (Open Closed Principle)*, aiemmin käsitelty Liskovin korvaavuusperiaate (kappale Liskovin korvaavuusperiaate), *rajapintojen erottelu -periaate (Interface Segregation Principle)* ja *kaanteisten riippuvuuksien periaate (Dependency Inversion Principle)*. SOLID-lyhenne syntyy periaatteiden englanninkielisten nimien ensimmäisistä kirjaimista. Periaatteiden seuraaminen ei ole pakollista, mutta se yleensä helpottaa koko ohjelmointiprosessia pidemmällä aikavälillä.

Yhden vastuun periaate

Ensimmäinen periaate on yhden vastuun periaate. Se käsittelee luokkien suunnittelua ja käyttöä. Periaatteen mukaan jokaisella luokalla pitäisi olla yksi, ja vain yksi vastuu. Mikäli luokan käyttötarkoitus ei ole kuvailtavissa yhdellä melko yksinkertaisella lauseella, rikkoo luokka todennäköisesti yhden vastuun periaatetta.

Avoin/suljettu-periaate

Toinen SOLID-periaate on avoin/suljettu-periaate. Sen mukaan ohjelmiston koodin tulisi olla avoin laajennuksille, mutta suljettu muutoksille. Käytännössä tämä tarkoittaa, että toteutettujen toiminnallisuuksien koodin tulisi olla suunniteltu olettaen, että sama koodi säilyy ohjelmistossa koko sen elinkaaren läpi. Toisaalta taas koodin tulisi olla helposti laajennettavissa. Periaatteen tavoitteena on rohkaista suunnittelemaan ohjelmistoja niin, että ensisijainen tapa lisätä toiminnallisuuksia ei olisi vanhojen luokkien muokkaaminen, vaan uusien lisääminen.

Normaalisti esimerkiksi switch-rakenteiden käyttö monesti rikkoo avoin/suljettu periaatetta. Mikäli rakenteeseen tarvitsee lisätä uusi haara, täytyy ohjelmoijan käydä muokkaamassa vanhaa koodia, mikä altistaa ohjelmiston virhetiloille, kun kertaalleen toimivaksi todettua koodia muokataan myöhemmin. Oliopohjaisessa ohjelmoinnissa kannustetaan korvaamaan switch rakenteet *strategia (strategy)*-suunnittelumallilla. Seuraava koodiesimerkki havainnollistaa saman rakenteen toteuttamisen enum-switch yhdistelmällä, ja strategia-suunnittelumallilla. Huomaa, kuinka paljon helpompaa jälkimmäisen laajentaminen on, rakenteen sisäistä koodia ei tarvitse muuttaa ollenkaan, vaikka esimerkkiin lisättäisiin väli-ilmansuunnat tai muita suuntia. Jokainen suuntaluokka voisi määritellä hahmon liikkeen omassa movePlayer-metodissaan. Sen sijaan ensimmäisen esimerkin laajentaminen vaatii jo valmiin koodin muokkaamista aina, kun rakenteeseen halutaan lisätä uusi suunta.

Koodiesimerkki 58: week9/openclosedexample: Pelaaja-luokka, jonka koordinaatteja esimerkissä pyritään muuttamaan

```
package week9.openclosedexample;

// Mikäli pelaaja olisi vain yksinkertainen dataluokka, voitaisiin x ja y paljastaa julkisiksi
// muuttujiksi. Koska oletettavasti kuitenkin pelaajan toiminnallisuuksia halutaan laajentaa
// tulevaisuudessa, on esimerkissä kyseiset muuttujat eristetty oikeaoppisen hakija-noutaja-mallin
// taakse. Tämä mahdollistaa esimerkiksi helpomman pelaajan liikkumisen rajaamisen tulevaisuudessa.
public class Player {

    private int x;
    private int y;

    public Player(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getX() {
        return x;
    }

    public void setY(int y) {
        this.y = y;
    }

    public int getY() {
        return y;
    }
```

```
}
```

Koodiesimerkki 59: week3/enumexample: Ilmansuunnat enum-luokalla esiteltynä

```
package week3.enumexample;

public enum CompassPoint {
    NORTH,
    EAST,
    SOUTH,
    WEST
}
```

Koodiesimerkki 60: week9/openclosedexample: Hahmon liikuttaminen switch-rakenteella ja ilmansuuntia esittävällä enum-rakenteella

```
package week9.openclosedexample;

// Huomioi ilmansuuntien uudelleenkäyttö viikolta 3. Koska luokka on eri paketissa, täytyy luokka
// tuoda nimiavaruuteen import-komennolla
import week3.enumexample.CompassPoint;

public class MoveSwitch {

    public void movePlayer(Player player, CompassPoint direction) {
        switch (direction) {
            case NORTH:
                player.setX(player.getX() + 1);
                break;
            case EAST:
                player.setY(player.getY() + 1);
                break;
            case SOUTH:
                player.setX(player.getX() - 1);
                break;
            case WEST:
                player.setY(player.getY() - 1);
                break;
        }
    }
}
```

Koodiesimerkki 61: week9/openclosedexample: Rajapintaluokka hahmon liikkuttamiseen strategia-suunnittelumallilla

```
package week9.openclosedexample;

public interface MovementCommand {
```

```
    void movePlayer(Player player);
}
```

Koodiesimerkki 62: week9/openclosedexample: Liikkumiskomento pohjoiseen

```
package week9.openclosedexample;

public class MoveNorth implements MovementCommand {
    @Override
    public void movePlayer(Player player) {
        player.setX(player.getX() + 1);
    }
}
```

Koodiesimerkki 63: week9/openclosedexample: Liikkumiskomento itään

```
package week9.openclosedexample;

public class MoveEast implements MovementCommand {
    @Override
    public void movePlayer(Player player) {
        player.setY(player.getY() + 1);
    }
}
```

Koodiesimerkki 64: week9/openclosedexample: Liikkumiskomento etelään

```
package week9.openclosedexample;

public class MoveSouth implements MovementCommand {
    @Override
    public void movePlayer(Player player) {
        player.setX(player.getX() - 1);
    }
}
```

Koodiesimerkki 65: week9/openclosedexample: Liikkumiskomento länteen

```
package week9.openclosedexample;

public class MoveWest implements MovementCommand {
    @Override
    public void movePlayer(Player player) {
        player.setY(player.getY() - 1);
    }
}
```

Koodiesimerkki 66: week9/openclosedexample: Hahmon liikuttaminen strategia-suunnittelumallilla

```
package week9.openclosedexample;

public class MoveStrategy {

    public void movePlayer(Player player, MovementCommand command) {
        command.movePlayer(player);
    }
}
```

Rajapintojen erottelu -periaate

SOLID-PERIAATTEISTA neljäs on rajapintojen erottelu -periaate. Se käsittelee rajapintaluokkien käyttöä ja luokkien kokoa. Periaatteen mukaan, jos luokka implementoi rajapintaluokan, täytyisi luokan tarvita kaikkia rajapinnan metodeja. Rajapintaluokkien pitäisi olla siis tarpeeksi hyvin rajattuja, jotta niiden kaikki metodit kuuluvat yhteen toiminnallisuuspakettiin. Periaate kannustaa täten luomaan monta pientä rajapintaa ja implementoimaan niistä tarvittavat yhden valtavan rajapinnan sijaan. Tämä helpottaa jo luotujen rajapintaluokkien uudelleenkäyttöä.

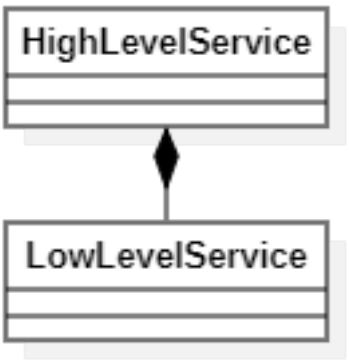
Käänteisten riippuvuuksien periaate

VIIMEINEN SOLID-PERIAATE on kaanteisten riippuvuuksien periaate. Se käsittelee riippuvuuksien suuntaa ohjelmistossa. Periaate perustuu toiveeseen korkean abstraktiotason komponenttien helposta uusiokäytöstä ja niiden vastustuskyvystä muutoksille. Molemmat ominaisuudet vaativat, että kyseiset komponentit eivät ole suoraan riippuvaisia käyttämistään alemman tason komponenteista. Periaate on yleensä esitetty kahtena perusideana:

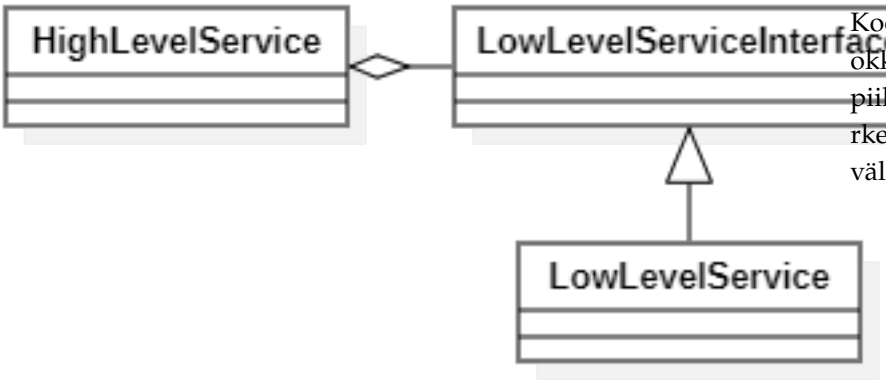
1. Korkean tason komponenttien ei pitäisi olla riippuvaisia matalan tason komponenteista. Molempien pitäisi olla riippuvaisia abstraktioista.
2. Abstraktioiden ei pitäisi olla riippuvaisia yksityiskohdista. Yksityiskohtien pitäisi olla riippuvaisia abstraktioista.

KÄYTÄNNÖSSÄ TÄMÄ tarkoittaa, että seuratakseen käänteisten riippuvuuksien periaatetta korkean tason luokka ei saa luoda itse tarvitsemiaan alemman tason riippuvuuksia, vaan sen on pyydettävä ne muulta ohjelmistolta rakentajassaan. Täten täytetään ensimmäisen säännön ensimmäinen puoli. Säännön toinen puoli puolestaan täyttyy, kun korkean tason komponentti ei pyydä rakentajassaan konkreettista luokkaa riippuvuutena, vaan rajapintaluokkaa, jonka alemman

tason komponentti toteuttaa. Seuraava kuvapari havainnollistaa tämän suorat riippuvuudet rikkovan arkkitehtuurin UML-luokkakaaviona. Huomaa HighLevelService-luokan assosiaationuolen tyypin muuttuminen. Ensimmäisessä kuvassa luokkien välinen suhde on kompositio, kun taas toisessa suhteen tyyppi on aggregaatio. Tämä johtuu teoreettisesta LowLevelServiceInterface-luokan vastaanottamisesta luokan rakentajassa konkreettisen LowLevelService luokan luonnin sijaan. Täten LowLevelServiceInterface luokan elinikä ei enää ole sidottu HighLevelService-luokan elinikään.



Koodiesimerkki 67: Luokkakaavio suorasta riippuvuudesta korkean ja matalan tason luokan välillä



Koodiesimerkki 68: Luokkakaavio rajapintaluokalla piilotetusta riippuvuudesta korkean ja matalan tason luokan välillä

KÄÄNTEISTEN RIIPPUVUUKSIEN periaatteeseen sitoutuminen tuottaa yleensä jonkin verran ylimääräistä koodia riippuvuuskohtien rajapintojen luonnissa. Lisäksi ohjelmistoon on luotava luokka tai luokkia, joiden vastuulla on ohjelmiston riippuvuuksien rakentaminen käynnistyksen yhteydessä, koska korkean tason luokat eivät enää osaa itse rakentaa tarvitsemiaan matalan tason palveluita. Periaate saattaa vaikuttaa turhalta teoreettiselta näpertelyltä ja sitä se todennäköisesti onkin pienemmissä projekteissa. Periaatteella on kuitenkin

valtava jalansija teollisessa koodintuotannossa, eikä tämä ole sattumaa. Käänteisten riippuvuuksien periaatetta noudattavaa koodia on uskomattoman helppo testata, koska jokainen palvelu voidaan irrottaa lopun ohjelmiston kontekstista ja palvelua voidaan testata puhtaasti oman toiminnallisuutensa suhteen. Opas ei siis suosittele kaikkia oppaan lukijoita toteuttamaan omia projektejaan käänneteisten riippuvuuksien periaatteen mukaan. Periaate kannattaa kuitenkin opetella, sillä siirryttäessä työelämään sekä koodikantojen koko että testaamisen merkitys moninkertaistuvat ja silloin käänteisten riippuvuuksien periaate kohoaa todennäköisesti tärkeimmäksi kaikista SOLID-periaatteista.

Käyttäjän määrittelemät virhetilat

Aiemmin oppaan luvussa Virheenkäsittelyn perusteet, try-catch-finally käytiin läpi Javan virhetilanhallintasyntaksi. Luvussa käsiteltiin standardikirjaston toimintojen aiheuttamia virhetiloja, mutta standardikirjasto ei omista yksinoikeutta virhetilojen aiheuttamiseen. Käyttäjä voi myös itse nostaa ja jopa luoda uusia virhetilaluokkia. Virhetilaluokka nostetaan *throw*-avainsanalla ja käyttäjä voi luoda omia virhetilaluokkiaan periyttämällä luokan *Exception*-luokasta.

Käyttäjän määrittelemä virhetilaluokka

Ohjelmistolla voi olla monia syitä käyttää käyttäjän itse määrittelemiä virhetiloja. Perimmäinen syy virhetilalle on aina mahdollistaa jostakin ajonaikaisesta ongelmasta toipuminen. Java tarjoaa virhetilaluokat laitteiston ja standardikirjaston toiminnasta riippuville ongelmille, mutta se ei pysty tarjoamaan ohjelmiston suunnittelusta aiheutuville mahdollisille ongelmille erikseen määriteltyjä virheitä. Esimerkiksi virheellinen käyttäjäsyöte voi aiheuttaa tilanteen, jossa ohjelman ajoa ei voida jatkaa normaalisti. Tätä varten Java tarjoaa käyttäjälle mahdollisuuden luoda omia virhetilaluokkiaan. Tämä tapahtuu yksinkertaisesti periyttämällä toivotun virhetilaluokan Exception-luokasta.

Manuaalinen virhetilan aiheuttaminen

Oman virhetilaluokan luominen ei hyödytä ohjelmistoa, ellei ohjelmointikieli tarjoa työkaluja käyttää luotua virhetilaluokkaa. Tätä varten Java tarjoaa throw-avainsanan. Avainsanaa käytetään lisäämällä avainsanan perään jonkin virhetilan instanssi, tai luomalla uusi virhetilaluokan instanssi new-avainsanalla. Tämä aiheuttaa koodin siirtymisen virhetilanhallintaan, josta aiheutettu virhetila voidaan hallitusti napata ylemmällä tasolla. Tämä mahdollistaa siistin virhetilasta palautumisen tasolla, jossa se on mahdollista.

PELKKÄ VIRHETILAN nosto ei kuitenkaan riitä, mikäli virhetilan palautuminen ei tapahdu saman metodin sisällä. Java vaatii käyttäjää määrittelemään metodit, jotka voivat aiheuttaa käsittelemättömiä virhetiloja *throws*-avainsanalla. Avainsana lisätään metodin määritelmään nimen ja parametrien jälkeen ja avainsanan perään lisätään virhetilaluokka tai -luokat, joita metodi saattaa aiheuttaa. Javan kääntäjä vaatii käyttäjää käsittelemään kaikki virhetilat jota kutsuttu metodi voi aiheuttaa. Tämän vaatimuksen voi kiertää määrittelemällä metodin, joka kutsuu virhetilan aiheuttavaa metodia, myös kyseisen virhetilan aiheuttavaksi. Tämä siirtää vastuun virhetilan käsittelystä yhtä koditasoa ylemmäksi. Yleensä virhetilat kannattaa kuitenkin käsitellä alimmalla tasolla, jolla puhdas palautuminen virhetilasta on mahdollista.

SEURAAVA KOODIESIMERKKI esittelee käyttäjän määrittelemän virhetilaluokan luomisen, sekä virhetilan aiheuttavan metodin määrittelemisen. Virhetila johtuu käyttäjän virheellisestä syötteestä, joten se käsitellään alimalla tasolla, jolla täydellinen toipuminen on mahdollista: käyttäjärajapinnassa.

Koodiesimerkki 69: week9/exceptionexample: Itse määritelty virhetilaluokka

```
package week9.exceptionexample;

public class NoDatabaseEntryException extends Exception {

    public NoDatabaseEntryException(String missingEntry) {
        super("Did not find the entry '" + missingEntry + "' in the database");
    }
}
```

Koodiesimerkki 70: week9/exceptionexample: Yksinkertaistettu tietokantayhteyttä matkiva luokka, joka nostaa luodun virhetilan jos annettua esinettä ei löydy

```
package week9.exceptionexample;

import java.util.HashMap;

public class DatabaseConnection {

    private HashMap<String, String> entries;

    public DatabaseConnection() {
        this.entries = new HashMap<>();
        entries.put("LUT", "Lappeenranta Lahti University of Technology");
    }

    public String getEntry(String key) throws NoDatabaseEntryException {
```

```
        String entry = this.entries.get(key);
        if (entry == null) {
            throw new NoDatabaseEntryException(key);
        }
        return entry;
    }
}
```

Koodiesimerkki 71: week9/exceptionexample: Pääluokka, joka kutsuu luotua valetietokantaa

```
package week9.exceptionexample;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Scanner;

public class ExceptionExample {

    public static void main(String[] args) {
        InputStreamReader defaultInputReader = new InputStreamReader(System.in);
        BufferedReader bufferedReader = new BufferedReader(defaultInputReader);
        Scanner bufferedScanner = new Scanner(bufferedReader);
        DatabaseConnection connection = new DatabaseConnection();

        System.out.println("Please input the entry you want to search: ");
        String key = bufferedScanner.nextLine();
        try {
            String entryData = connection.getEntry(key);
            System.out.println("Entry data: " + entryData);
        } catch(NoDatabaseEntryException exception) {
            System.out.println("Sorry, could not find the entry '" + key + "'");
        }
    }
}
```

Geneerisyys ja iteraattorit

Geneerisyys

Geneerisyys on suunniteltu mahdollistamaan laajempi uudelleenkäyttö luokille ja metodeille. Esimerkiksi ilman geneerisyyttä kaikista kokoelmatyypeistä pitäisi olla erillinen versio jokaikiselle luokalle, jota kokoelmassa säilötään. Oppaassa on sivuttu jo geneerisyyttä kappaleessa Kokoelmarakenteet, mutta konsepti on monimutkainen, joten tarkempi käsittely jätettiin silloin välistä. Tämä kappale antaa yleiskatsauksen tähän hyödylliseen, mutta monimutkaiseen Javan ominaisuuteen.

Geneerisyyden perusteet

Oppaan kappaleessa Listat mainittiin käsite geneerinen luokka. Geneerinen luokka ja *geneerinen metodi (generic method)* ovat yleisiä käsitteitä staattisesti tyypitetyille kielille ja viittaavat luokkaan ja metodiin, jotka määritellään suhteessa johonkin toiseen luokkaan. Tämä geneerinen *tyyppiparametri (type paremeter)* kirjoitetaan yleensä luokan määritelmän yhteydessä yhdellä kirjaimella, joka on toivottua parametria kuvaavan sanan ensimmäinen kirjain; T - Type, E - Element, K -Key, N - Number, V - Value. Monta tyyppiparametria omaavassa luokassa voidaan käyttää joko kirjaimia S, U, V, jne. tai numeroida parametrit (T1, T2, jne.). Tyyppiparametri sijaitsee luokan määritelmässä luokan nimen perässä ja metodin määritelmässä metodin paluuarvotyypin edellä. Molemmissa tapauksissa tyyppiparametri suljetaan pienempi kuin- ja suurempi kuin -merkkien väliin. Mikäli luokka tai metodi on riippuvainen useammasta tyypistä annetaan kaikki tyypit samojen merkkien välissä, pilkulla eroteltuina, metodin parametrien tapaan.

Tyyppiparametri voi olla rajoitettu perintäehdolla, jolloin kaikkien geneeristä luokkaa tai metodia kutsuvien koodin osien on toteutettava tämä ehto luokalle tai metodille tarjoamissaan tyypeissä. Tämä tapahtuu lisäämällä heti rajattavan tyyppiparametrin perään extends-avainsana ja ylärajaluokka. Rajattu tyyppiparametri vaatii, että parametriin annettu tyyppi on joko ylärajaluokan instanssi tai jonkin ylärajaluokasta periytyvän luokan instanssi.

Kuten kappaleessa Listat opittiin, määriteltäessä geneerisen luokan instanssin säilövää muuttujaa täytyy tyyppiparametri antaa samalla notaatiolla kuin luokan määritelmässä. Tätä geneerisen luokan tyyppiparametrin määrittelemistä kutsutaan parametrisoinniksi. Luotaessa instanssia olemassa olevaan parametrisoituun muuttujaan, voidaan Java 7:ssä ja uudemmissa Javan versioissa jättää tyyppiparametrit määrittelemättä: kääntäjä osaa tulkita ne muuttujasta, johon instanssia luodaan. Samaan tapaan kutsuttaessa geneeristä metodia kääntäjä tulkitsee tyyppiparametrit annettujen muuttujien tyypeistä.

Seuraava koodiesimerkki esittelee geneerisen säilöluokan toteuttamisen ja geneerisen metodin toteuttamisen säilöluokalle, sekä luokan instanssin luomisen ja metodin kutsumisen.

Koodiesimerkki 72: week10/genericexample: Geneerinen säiliöluokka, jolla on geneerinen metodi säiliön sisällön vaihtamiseen

```
package week10.genericexample;

// Määritellään luokan olevan geneerinen luokan "T" yli. "T" voi olla mikä tahansa luokka, mutta
// ei primitiivinen tietotyyppi
public class GenericContainer<T> {

    // "T"-parametria voi käyttää normaalin luokan tapaan geneerisen luokan rungon sisällä.
    private T object;

    public GenericContainer(T object) {
        this.object = object;
    }

    public T getObject() {
        return object;
    }

    // Metodi ottaa jonkin T:n tai T:n lapsiluokan olion, vaihtaa sen säiliön olioksi ja palauttaa
    // vanhan olion. "S"-parametri kuvastaa tämän T:n tai T:n lapsiluokan olion tyyppiä.
    //
    // On huomioitavaa, että <S extends T> on tässä tapauksessa turha ja koko tyyppiparametrin
    // käyttö voitaisiin korvata muuttamalla parametrin newObject tyyppi S-tyypistä T-tyypiksi,
    // koska Javan kääntäjä tulkitsee lapsiluokan instanssit myös kantaluokan instanssiksi.
    // Koodi on kirjoitettu tässä muodossa, jotta extends-avainsanan tyyppiparametria rajaava
    // käyttötapa saataisiin esiteltyä.
    public <S extends T> T swap(S newObject) {
        T oldObject = this.object;
        this.object = newObject;
        return oldObject;
    }
}
```

Koodiesimerkki 73: week10/genericexample: Säiliöluokan instanssin luominen, ja vaihtometodin kutsuminen

```java
package week10.genericexample;

import week6.overridingexample.Child;
import week6.overridingexample.Parent;

public class GenericExample {

    public static void main(String[] args) {
        Parent parent = new Parent();
        Child child = new Child();
        // Huomioi tyyppiparametrin puuttuminen new-kutsusta.
        GenericContainer<Parent> container = new GenericContainer<>(parent);
        // swap-metodi ei tarvitse tyyppiparametria, sillä kääntäjä tulkitsee sen tyypin child-
        // muuttujan tyypistä.
        Parent stored = container.swap(child);
        System.out.println(stored.addText("Called from class "));
        System.out.println(container.getObject().addText("Called from class "));
    }
}
```

Geneerisyys ja perintä

genericInheritance

TYYPPIPARAMETRIN KONKREETTINEN määrittäminen on mahdollista myös perinnän yhteydessä. Generisestä kantaluokasta on mahdollisuus luoda parametrisoitu lapsiluokka ja geneerisestä rajapintaluokasta on mahdollista luoda parametrisoitu implementaatio. Tällöin luotu luokka ei enää ole geneerinen, sillä sen tyyppiparametria ei voida määrittää luokkaa luotaessa. Tätä parametrisointia ei ole pakko toteuttaa, periytymisessä voidaan geneerinen parametri myös jättää ennalleen tai sen rajausta voidaan tiukentaa. Rajauksen löysentäminen ei kuitenkaan ole mahdollista. Mikäli kantaluokka on geneerinen tyypin T yli ja T on rajattu olemaan tietyn luokan tai sen lapsiluokkien instanssi, ei voida ilmoitta T:hen käyvän minkä tahansa olion. Tämä rikkoisi kantaluokan toiminnallisuuden ja täten ei seuraa Liskovin korvaavuusperiaatetta.

SEURAAVA KOODIESIMERKKI esittelee geneerisen validointirajapinnan ja kahden rajapinnan implementoivan validaatioluokan luomisen. Toinen luokista on parametrisoitu, mutta toinen on geneerinen alkuperäisen rajapinnan kanssa identtisin rajoituksin (mikä tahansa olio). Huomaa, että parametrisoidun luokan tyyppisen muuttujan luonti ei enää vaadi tyyppiparametria.

Koodiesimerkki 74: week10/genericinheritance: Geneerinen rajapintaluokka validaattoriluokan pohjaksi

```java
package week10.genericinheritance;
```

```
public interface Validator<T> {
    boolean validate(T object);
}
```

Koodiesimerkki 75: week10/genericinheritance: Rajapintaluokan implementaatio, joka on parametrisoitu String-tyypillä

```
package week10.genericinheritance;

public class StringValidator implements Validator<String> {

    @Override
    public boolean validate(String object) {
        return object.equals("Valid");
    }
}
```

Koodiesimerkki 76: week10/genericinheritance: Rajapintaluokan geneerinen implementaatio null-arvojen estämiseksi

```
package week10.genericinheritance;

public class NonNullValidator<T> implements Validator<T> {

    @Override
    public boolean validate(T object) {
        return object != null;
    }
}
```

Koodiesimerkki 77: week10/genericinheritance: Parametrisoidun ja geneerisen validaattoriluokan luonti koodissa

```
package week10.genericinheritance;

public class GenericInheritanceExample {

    public static void main(String[] args) {
        NonNullValidator<Boolean> booleanValidator = new NonNullValidator<>();
        StringValidator stringValidator = new StringValidator();
        Boolean validBoolean = false;
        String invalidString = "Invalid";
        if (booleanValidator.validate(validBoolean)) {
            System.out.println("Valid object");
        } else {
            System.out.println("Invalid object");
        }

        if (stringValidator.validate(invalidString)) {
            System.out.println("Valid string");
```

```
        } else {
            System.out.println("Invalid string");
        }
    }
}
```

Iteraattorit ja kuluttajat

Iteraattorit

KOKOELMIEN KANSSA toimittaessa saattaa ilmetä tarvetta kappaleessa For-each-looppi esiteltyä for-each -looppia tarkempaan oliokohtaiseen iterointiin. Tätä varten Java tarjoaa luokat *Iterator* ja *ListIterator*. Päätös olla tarjoamatta iterointimetodeja suoraan kokoelmaluokasta liittyy kokoelmaluokan tehtävään ja yhden vastuun periaate -SOLID-periaatteen noudattamiseen. Kokoelmaluokan tehtävä on säilöä dataa. Mikäli luokan tehtäviin lisättäisiin iterointikohdan muistaminen, luokalla olisi kaksi tehtävää. Tästä syystä kokoelmaluokat tarjoavat *iterator*-metodin ja List-rajapinnan toteuttavat listat *listIterator*-metodin näiden iterointiluokkien luomiseen. Nimiensä mukaisesti iterator-metodi luo Iterator-instanssin ja listIterator-metodi luo ListIterator-instanssin.

ITERAATTORIEN KÄYTTÖ on melko simppeliä. Iteraattoriolio muistaa indeksin, johon asti kokoelmaa on iteroitu ja mahdollistaa seuraavan instanssin olemassaolon tarkistamisen hasNext-metodilla, joka ei ota argumentteja ja palauttaa totuusarvon seuraavan instanssin olemassaolosta. Seuraava instanssi voidaan noutaa metodilla next, joka ei ota argumentteja ja palauttaa seuraavan instanssin, sekä korottaa iterointi-indeksiä iteraattoriluokan sisällä yhdellä. Mikäli seuraavaa instanssia ei ole, aiheuttaa next-metodi NoSuchElementException-virhetilan. ListIterator omistaa lisäksi hasPrevious ja previous -metodit, jotka toimivat samalla tavalla, mutta iteroivat kokoelmaa takaperin. Sekä Iterator-oliot että ListIterator-oliot sisältävät lisäksi mahdollisuuden ajaa tietyn funktion kaikille iteraattorissa jäljellä oleville alkioille forEachRemaining-metodilla. Metodi ottaa *Consumer*-rajapinnan implementoivan olion ja kutsuu rajapinnassa määriteltyä accept-metodia kaikilla jäljellä olevilla alkioilla. Rajapinnan toiminta käydään tarkemmin läpi seuraavassa oppaan osiossa (kuluttajat). Molemmat iteraattorityypit kykenevät myös poistamaan viimeisimmän noudetun kokoelman alkion remove-metodilla.

NÄIDEN JAETTUJEN toiminnallisuuksien lisäksi ListIterator-rajapinta määrittelee set- ja add-metodit. Metodit ottavat iteraattorin pohjana olevan List-instanssin alkiotyypin olion. Metodi set asettaa viimeisimmän noudetun kokoelman alkion tilalle annetun alkion ja add-metodi lisää annetun alkion listaan viimeisimmän noudetun alkion jälkeiseksi alkioksi. Iteraattorien käyttö koodissa esitellään kuluttajat-osion koodiesimerkin yhteydessä.

kuluttajat

VIIME KAPPALEESSA (Iteraattorit) esiteltiin *kuluttaja (consumer)*-tyyppisen luokan idea. Kuluttaja on metodi, joka ottaa yhden määritellyn tyyppisen olion ja ei palauta mitään, periaatteessa kuluttaen vastaanottamansa instanssin. Java tarjoaa erillisen geneerisen Consumer-rajapintaluokan tällaisten kuluttajien luomiseen. Rajapintaluokka on geneerinen tyyppiparametrin T yli ja määrittelee kaksi metodia; accept(T) ja andThen(Consumer<? super T>). Rajapinnan kuluttajametodi on accept, se ei palauta mitään ja ottaa yhden tyypin T instanssin. Toinen rajapinnan tarjoama metodi on andThen-joka ottaa toisen kuluttajan, jonka pitää olla parametrisoitu samoin tai löysemmin rajoin kuin metodin omaava kuluttajaimplementaatio ja palauttaa yhdistetyn kuluttajan, joka kutsuu molempien kuluttajien accept-metodia vuorotellen.

KOSKA KULUTTAJA-SUUNNITTELUMALLIN ideana on, ettei kuluttava metodi palauta mitään, täytyy kaiken metodin toiminnallisuuden tapahtua muokaten metodin omistavaa luokkaa tai jotain jaettua tilaa. Seuraava koodiesimerkki esittelee tekstikuluttajan luomisen simppelimmän printtauskomennon kirjoittamiseksi. Tässä tapauksessa globaalin metodin toiminta on syötteen antaminen käyttäjälle: koko ohjelmisto jakaa saman terminaalin.

Koodiesimerkki 78: week10/iteratorexample: Kuluttajaluokka, joka tulostaa merkkijonoargumentin

```
package week10.iteratorexample;

import java.util.function.Consumer;

// Luodaan parametrisoitu implementaatio Consumer-rajapinnasta
public class Printer implements Consumer<String> {
    @Override
    public void accept(String string) {
        System.out.println(string);
    }
}
```

Koodiesimerkki 79: week10/iteratorexample: Merkkijonolistan iteraattorin luominen, ensimmäisen alkion kuluttaminen ja loppujen tulostaminen luodulla tulostinluokalla

```
package week10.iteratorexample;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.ListIterator;

public class IteratorExample {
```

```
    public static void main(String[] args) {
        // Valmiiksi täytetyn ArrayList-instanssin luonti onnistuu helposti Arrays.asList-metodilla
        ArrayList<String> strings =
                new ArrayList<>(Arrays.asList("First", "Second", "Third", "Fourth"));
        ListIterator<String> stringsIterator = strings.listIterator();

        // Varmistetaan, että iteraattorissa on jäljellä alkioita hasNext-metodilla ja sen jälkeen
        // kulutetaan ensimmäinen alkio, mikäli alkio on olemassa
        if (stringsIterator.hasNext()) {
            stringsIterator.next();
        }

        // Luodaan kuluttajatulostaja ja tulostetaan listan loput alkiot kuluttajan avulla
        Printer printer = new Printer();
        stringsIterator.forEachRemaining(printer);
    }
}
```

Sisäiset luokat ja anonyymit luokat, kopiointi

Sisäiset ja anonyymit luokat

JAVA TARJOAA koodin selkeyden vuoksi mahdollisuuden määritellä luokan toisen luokan sisällä. Tämän luokan nimi on *sisaluokka (inner class)*. Sisäluokka käyttäytyy kuin mikä tahansa luokan sisällä määritelty kenttä. Se näkee kaikki muuttujat luokasta, jonka sisällä se on määritelty ja se voidaan normaalin luokan sijaan määritellä näkyvyysmääreellä protected tai private. Tarvittaessa sisäluokka voidaan myös luoda static-avainsanalla, jolloin siihen voidaan viitata luomatta sen omistavan luokan instanssia.

SISÄLUOKKIEN KÄYTTÖTARVE perustuu lähinnä puhtaamman koodin tuottamiseen: ne eivät tarjoa valtavaa toiminnallista etua, vaan mahdollistavat koodin jäsentämisen tavalla, joka on joskus luonnollisempi ja luettavampi. Esimerkiksi jossain kompositiorakenteissa vain yksi luokka tarvitsee toista luokkaa. Jos tämä toinen luokka on tarpeeksi pieni, voi olla järkevämpää määritellä tämä luokka sisäluokkana. Lisäksi sisäluokkien uniikki kyky nähdä kaikki ulomman luokan muuttujat ja metodit, mukaan lukien yksityiset muuttujat ja metodit, saattaa auttaa enkapsulaation parantamisessa. Mikäli luokka A paljastaa jonkin metodin tai luo noutajan jollekin muuttujalle vain luokkaa B varten, voi olla syytä muuttaa luokka B luokan A sisäluokaksi. Näin aiemmin julkiset kentät voidaan muuttaa yksityisyksi. Seuraava koodiesimerkki esittelee sisäluokan luomisen ja käytön perusteet.

Koodiesimerkki 80: week11/innerclassexample: Luokka, jolla on sisäluokka

```
package week11.innerclassexample;

public class OuterClass {

    private String message;

    public OuterClass(String message) {
        this.message = message;
    }
```

```
    public String getMessage() {
        return "Inside outer class: " + message;
    }

    public class InnerClass {
        // Sisäluokka näkee ulommaan luokaan muuttujat ja metodit, ellei luokka ole määritelty
        // static-avainsanalla. Tällöin sisäluokka näkee vain static-muuttujat ja metodit.
        public String getMessage() {
            return "Inside inner class: " + message;
        }
    }
}
```

Koodiesimerkki 81: week11/innerclassexample: Sisäluokan instanssin luominen ja käyttö

```
package week11.innerclassexample;

import week11.innerclassexample.OuterClass.InnerClass;

public class InnerClassesExample {

    public static void main(String[] args) {
        OuterClass outerClass = new OuterClass("Example");

        // Sisäluokan luonti pistenotaatiolla kutsutulla new-avainsanalla. Mikäli sisäluokan luokkaa
        // ei olisi tuotu nimitilaan import-avainsanalla tiedoston alussa, voitaisiin luokkaan
        // viitata pistenotaatiolla ulkoisesta luokasta ("OuterClass.InnerClass innerClass = ...")
        InnerClass innerClass = outerClass.new InnerClass();
        // Tulostaa "Inside inner class: Example"
        System.out.println(innerClass.getMessage());
        // Tulostaa "Inside outer class: Example"
        System.out.println(outerClass.getMessage());
    }
}
```

Sisäluokka ei ole sama asia kuin *anonyymi luokka (anonymous class)*. Siinä missä sisäluokka on toiminnallisuuksiltaan täysi luokka, joka vaan sattuu sijaitsemaan toisen luokan sisällä, on anonyymi luokka vain kertakäyttöinen tynkäimplementaatio jollekin rajapintaluokalle. Anonyymi luokka luodaan tallentamalla rajapintaluokan tyyppiseen muuttujaan implementoitavan rajapintaluokan new-avainsanalla luotu parametriton rakentajaakutsu. Tätä kutsua on seurattava anonyymin luokan rungon määritelmä suljettuna aaltosulkeisiin. Runkoon voidaan määritellä muuttujia, mutta luokkaa ei voi nimetä, vaan luotu luokka rakennetaan saman tien ja luokan ainut instanssi sidotaan annettuun muuttujaan. Seuraava koodiesimerkki esittelee kuluttajat-kappaleen koodiesimerkin toteuttamisen luomalla Printer-kuluttajaa vastaava luokka anonyymina luokkana Main-luokassa. Huomaa myös kuinka

anonyymi luokka voi olla määritelty static-avainsanalla tarvittaessa.

Koodiesimerkki 82: week11/anonymousclassexample: Anonyymin luokan luominen ja käyttö

```
package week11.anonymousclassexample;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.ListIterator;
import java.util.function.Consumer;

public class AnonymousClassesExample {

    public static void main(String[] args) {
        ArrayList<String> strings =
                new ArrayList<>(Arrays.asList("First", "Second", "Third", "Fourth"));
        ListIterator<String> stringsIterator = strings.listIterator();

        // Luodaan tulostajakuluttaja anonyymina sisäluokkana
        Consumer<String> printer = new Consumer<>() {
            @Override
            public void accept(String string) {
                System.out.println(string);
            }
        };

        // Tulostaa:
        // "First
        // Second
        // Third
        // Fourth"
        stringsIterator.forEachRemaining(printer);
    }
}
```

Kopionti

Toisin kuin C:n kaltaiset matalan tason kielet, jotka säilövät muuttujat oletuksena arvoina, oliopohjaiset kielet kuten Java säilövät kaikki muuttujat paitsi primitiiviset tietotyypit viittauksina. Tämä tarkoittaa, että annettaessa muuttujaa argumenttina jollekin metodille, Java ei kerro "käytä arvoa 'MyObject'". Tämän sijaan Java kertoo "käytä arvoa, joka löytyy tästä osasta muistia". Tämä helpottaa kielen käyttöä normaalitilanteissa, sillä kieli ei tarvitse C:n kaltaista osoitinsyntaksia antaakseen metodeille viittauksia. Toisaalta muuttujien kopioiminen muuttuu tämän takia hankalammaksi, sillä jos käyttäjä vain tallentaa "MyObject" instanssin uuteen muuttujaan tämä ei kopioi instanssin arvoja. Sen sijaan tällainen tallennus luo uuden viittauksen

samaan "MyObject"-instanssiin. Jos tämän uuden muuttujan tilaa muokataan, myös vanhan muuttujan tila muuttuu. Näin luodun kopion nimi on *viitekopio (reference copy)*.

MIKÄLI HALUTAAN luoda aidosti uusi kopio "MyObject"-instanssista tarvitaan Javassa *Object*-luokan clone-metodia. Jota olio voisi hyödyntää metodia, täytyy sen implementoida *Cloneable*-rajapintaluokka ja ylikirjoitettava Object-luokasta peritty clone-metodi. Tässä vaiheessa on tehtävä päätös onko toivottu kopioinnin tulos *matala kopio (shallow copy)* vai *syva kopio (deep copy)*.

MATALA KOPIO kopioi kaikki luokan muuttujat viittauskopioina, mikäli muuttuja on jonkin luokan instanssi ja suorittaa arvokopion vain jos muuttujan tyyppi on primitiivinen tietotyyppi. Näin kopiointi on melko nopea toimenpide ja uusi ja alkuperäinen instanssi eivät ole suoraan riippuvaisia toisistaan. Kuitenkin luodun uuden instanssin ei-primitiivisen tietotyypin muuttujat viittavaat samoihin olioihin kuin alkuperäisen instanssin vastaavat muuttujat. Täten uuden ja alkuperäisen kopion tilat ovat edelleen jossain määrin riippuvaisia toisistaan. Tämä kopiointityyli on Javan oletuskopiointi ja voidaan täten saavuttaa ylikirjoitetun clone-metodin sisällä kutsumalla super().clone -metodia.

SYVA KOPIO puolestaan kopioi kaikki alkuperäisen luokan muuttujat syväkopioina rekursiivisesti. Tällainen kopiointi kuluttaa huomattavasti enemmän resursseja kuin viitekopion tai matalan kopion luominen, etenkin pitkissä riippuvuusketjuissa. Kopiointityylin etuna on, että se luo kopioidun instanssin, jonka tila on täysin riippumaton alkuperäisen instanssin tilasta. Syvää kopiointia ei ole mahdollista implementoida yhden luokan clone-metodissa. Sen sijaan luokan on manuaalisesti kutsuttava clone-metodissaan kaikkien muuttujiensa clone-metodia, eli myös kaikkien luokan muuttujien on toteutettava Cloneable-rajapinta. Tämä syväkopioiva clone-metodi on sitten implementoitava jokaiseen luokkaan, jonka instanssin luokka omistaa ja niin edelleen täydellisen syväkopion saavuttamiseksi. Kopioinnin voi kuitenkin suorittaa jollain tasolla matalan kopion ja täydellisen syväkopion välillä.

SEURAAVA KOODIESIMERKKI esittelee luokan A, joka tukee matalan kopion luomista itsestään ja luokan, joka sisältää luokan A instanssin ja jonka kopiointimetodi luo matalan kopion instanssin omistamasta luokan A instanssista ja muuten kopio luokan B normaalina matalana kopiona.

Koodiesimerkki 83: week11/cloningexample: Matalan kopion luomista tukeva luokka

```
package week11.cloningexample;

public class DataPoint implements Cloneable {
```

```
    // Tiukka Java-filosofian noudattaminen vaatisi x ja y -kenttien eristämistä
    // private-avainsanalla ja noutajien ja/tai asettajien luomista. Opas ei ota kantaa, onko tämä
    // parempi vai huonompi tapa, mutta puhtaan dataluokan tapauksessa suora julkisen muuttujan
    // luonti vähentää kirjoitettavan koodin määrää jo voi helpottaa luokan käyttöä. Tämä altistaa
    // kuitenkin herkemmälle virheiden kasautumiselle tulevaisuudessa.
    public int x;
    public int y;

    public DataPoint(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public DataPoint clone() {
        // Matalan kopion luonti onnistuu yksinkertaisesti kutsumalla clone-metodia object-luokasta
        // Object-luokan clone-metodi voi aiheuttaa CloneNotSupportedException-vikatilan, joka
        // pitää joko nostaa eteenpäin tai käsitellä.
        try {
            return (DataPoint) super.clone();
        } catch(CloneNotSupportedException ignored) {
            System.out.println("Cloning failed because it was not supported.");
            return null;
        }
    }
}
```

Koodiesimerkki 84: week11/cloningexample: Monimutkaisemman clone-määritelmän omaava luokka

```
package week11.cloningexample;

public class ComplexDataPoint implements Cloneable {

    // Koska luokka on monimutkaisempi kuin DataPoint, on tehty päätös toteuttaa luokalle
    // noutaja-asettaja -suunnittelumalli oikeaoppisesti.
    private DataPoint coordinates;
    private double xVelocity;
    private double yVelocity;

    public ComplexDataPoint(double xVelocity, double yVelocity, DataPoint coordinates) {
        this.coordinates = coordinates;
        this.xVelocity = xVelocity;
        this.yVelocity = yVelocity;
    }

    public DataPoint getCoordinates() {
        return coordinates;
```

```
    }

    public double getXVelocity() {
        return xVelocity;
    }

    public double getYVelocity() {
        return yVelocity;
    }

    @Override
    public ComplexDataPoint clone() {
        // Clone-metodin sisällä voidaan määritellä mukautettua toiminnallisuutta, joka sekoittaa
        // puhtaan matalan kopion ja syvän kopion väliltä.
        try {
            ComplexDataPoint clone = (ComplexDataPoint) super.clone();
            clone.coordinates = this.coordinates.clone();
            return clone;
        } catch(CloneNotSupportedException ignored) {
            System.out.println("Cloning failed because it was not supported.");
            return null;
        }
    }
}
```

Loppusanat

MIKÄLI LUET tätä, olet oletettavasti opiskellut oppaan sisällön. Vilpittömät onnittelut Javan ja oliopohjaisen ohjelmoinnin perusteiden opiskelusta näin pitkälle ja suuret kiitokset mielenkiinnosta opasta kohtaan. Toivon oppaan ja koodiesimerkkien olleen avuksi opiskeluissa ja oppaan koodin laadusta muistuttavan tyylin herättäneen kiinnostusta kirjoittaa toimivan koodin sijaan laadukasta koodia. Lisäksi toivon oppaan antaman tietotaidon olevan hyödyllistä sinulle tulevaisuudessasi. Enemmän tai vähemmän mukavia koodaushetkiä, ja olkoon Google (tai muu valitsemasi hakukone) tukenasi.

Eetu "EddieTheCubeHead" Asikainen

Sanasto

abstrakti luokka (abstract class) Luokka, joka on määritelty abstract-avainsanalla ja jolle on määritelty vähintään yksi abstrakti metodi. Abstraktista luokasta ei voi luoda instanssia, vaan luokkaa voi käyttää vain kantaluokkana luokasta luoduille konkreettisille luokille. Näiden konkreettisten luokkien on implementoitava kaikki luokan abstraktit metodit. 62, 64, *see* kantaluokka, periytyminen, abstract, abstrakti metodi & konkreettinen luokka

abstrakti metodi (abstract method) Metodi, joka on määritelty abstract-avainsanalla ja joka sijaitsee abstraktissa luokassa. Abstraktin metodin määritelmä ei sisällä metodin runkoa, ainoastaan signatuurin. Luokan konkreettisten lapsiluokkien on implementoitava kyseinen metodi. 63, *see* abstrakti luokka, signatuuri & abstract

abstraktio (abstraction) Ohjelmoinnin perustekniikka, jossa ongelman tarkka ratkaisu piilotetaan kutsuttavan koodirakenteen, kuten funktion, tietorakenteen tai luokan taakse. 7

aggregaatio (aggregation) Luokkien suhteita kuvaava termi. Aggregaatiossa luokka A omistaa viittauksen luokan B instanssiin, mutta tämän instanssin luonti ja tuhoutuminen ei ole sidottu luokan A elinkaareen. Esimerkiksi aktiivisia pelaajia säilövän peli-luokan suhde pelaaja-luokkaan on aggregaatio, mikäli pelaajat säilötään peli-luokan sisällä listassa, johon jo olemassa oleva pelaaja lisätään tämän liittyessä peliin ja josta pelaaja poistetaan tämän poistuessa pelistä. 53, 76

ajonaikainen kaantaminen (Just-In-Time compiling, JIT) Ajonaikainen kääntäminen on koodin ajotekniikka, jossa. 36

anonyymi luokka (anonymous class) Toisen luokan sisällä luotu kertakäyttöinen rajapintaluokan implementaatio, joka sisältää vain ylikirjoitetut abstraktit metodit implementoidusta rajapintaluokasta. 90, *see* rajapinta & interface

asettaja (setter) Metodi, jonka tehtävänä on tarjota luokan ulkopuolisille toimijoille mahdollisuus asettaa arvoja luokan yksityiseen muuttujaan. Ottaa yleensä arvoksi muuttujan toivotun arvon, eikä palauta mitään. Nimetään camelCase-tyylillä sijoittamalla muuttujan nimen eteen "set" ("setVariable"). Parantaa enkapsulaatiota ja mahdollistaa esimerkiksi uusien arvojen validoinnin ennen niiden asettamista. 22

avoin/suljettu-periaate (Open/Closed Principle - OCP) SOLID-periaatteisiin kuuluva sääntö, jonka mukaan kirjoitetun koodin on oltava suljettu muutoksilta, mutta avoin laajennukselle. Ohjelmiston osien on säännön mukaan tapa laajentaa toimintaansa muokkaamatta olemassaolevaa toiminnan määrittelevää koodia. Esimerkiksi mikäli ohjelmisto käyttää switch-rakennetta, jonka sisällä suoritetaan logiikkaa saapuvan olion tyypin mukaan, ei rakenne ole laajennettavissa toiminnallisuudeltaan, ilman että rakenteen koodia muokataan. Sen sijaan rakenne, joka on ohjelmoitu vastaanottamaan olio ja kutsumaan oliossa itsessään määriteltyä metodia on laajennettavissa toiminnallisuudeltaan muokkaamatta lähdekoodia. Tämä tapahtuu tarjoamalla rakenteelle erilainen olio, joka toteuttaa rajapinnan, jossa rakenteen kutsuma metodi on määritelty. 71, *see* SOLID-periaatteet

enkapsulaatio (encapsulation) Datan piilottaminen olion sisään niin ettei muu ohjelmisto näe kyseistä dataa. Mitataan asteikolla matala-korkea, niin että korkea enkapsulaatio tarkoittaa pientä määrää julkisia metodeja tai datakenttiä ja matala taas suurta määrää julkisia metodeja ja datakenttiä see. 9, 20

funktio (function) Ohjelmoijan määrittelemä käskysarja, eli koodin osa, joka on rajattu, ottaa tietyn määrän parametreja ja mahdollisesti palauttaa paluuarvon. Tunnetaan olio-ohjelmoinnissa nimellä metodi. 7, *see* parametri & metodi

geneerinen luokka (generic class) Luokan ominaisuus, joka määrittää luokan instanssien täyden tyyppisignatuurin olevan riippuvainen toisesta luokasta. Tämä toinen luokka voidaan määrittää jokaisen luokan instanssin kohdalla erikseen pienempi kuin- ja suurempi kuin -merkkien väliin suljetulla notaatiolla. Esimerkiksi kokoelmat tarvitsevat tietoa sisältämiensä olioiden tyypityksestä. Merkkijonoja sisältävä ArrayList-kokoelma voitaisiin siis luoda seuraavalla notaatiolla:
"new ArrayList<String>()"
Geneerinen luokka voi olla myös geneerinen useamman luokan suhteen, jolloin luokat erotetaan pilkulla tai riippuvainen toisesta geneerisestä luokasta, jolloin geneerisyyden notaatiot kirjoitetaan sisäkkäin:
"new HashMap<String, ArrayList<Int»()". 29, 81, *see* ArrayList & HashMap

geneerinen metodi (generic method) Metodin ominaisuus, joka määrittää metodin signatuurin olevan riippuvainen toisesta luokasta. Tämä toinen luokka voidaan määrittää jokaisen metodikutsun yhteydessä erikseen pienempi kuin- ja suurempi kuin -metkkien suljetulla notaatiolla. Esimerkiksi kokoelmia käsittelevät metodit saattavat kyetä käsittelemään kokoelmia niiden tallentamasta tietotyypistä riippumatta. Tämänlainen metodi voitaisiin luoda no-

taatiolla:
"public <T> void processList(ArrayList<T> list)"
Notaatiossa "T" on geneerinen tyyppiparametri ("Type"). 81

hajautustaulu (hash table) Tietorakenne, joka koostuu avain-arvo -pareista. Jokaista avainta vastaa yksi arvo. Arvot ovat noudettavissa nopeasti avaimen perusteella. Toteutettu Javan standardikirjaston HashMap-luokassa. 30, *see* HashMap

instanssi (instance) Olio, joka on luotu jonkin luokan pohjalta on kyseisen luokan instanssi. 8, *see* olio & luokka

kaannetty kieli (compiled language) Ohjelmointikieli, jossa ohjelmistot kompiloidaan ennen ajamista ja ohjelman ajaminen tapahtuu suorittamalla kompilointiprosessin tuottama tiedosto. Esimerkiksi c ja Rust ovat kompiloituja kieliä. 36

kaanteisten riippuvuuksien periaate (Dependency Inversion Principle - DIP) SOLID-periaatteisiin kuuluva sääntö, joka käsittelee ohjelmiston riippuvuuksien suuntaa. Periaatteen mukaan ohjelmiston korkean tason kompinenttien ei pitäisi olla riippuvaisia matalan tason komponenteista vaan molempien olisi oltava riippuvainen jaetusta abstraktiosta (toteutettu rajapintaluokkana). Samaan tapaan rajapintaluokkien ei tulisi olla riippuvaisia konkreettisista luokista, vaan konkreettisten luokkien tulisi aina olla riippuvaisia rajapintaluokista. 71, 75, *see* SOLID-periaatteet, rajapinta & import

kantaluokka (parent class) Luokka, josta on periytetty vähintään yksi toinen luokka. 54, 62, *see* periytyminen & lapsiluokka

koheesio (cohesion) Ohjelmiston laadun mittaamiseen käytetty käsite. Mittaa luokkien sisäistä yhtenäisyyttä akselilla matala-korkea. Matala koheesio tarkoittaa että luokassa on paljon metodeja jotka eivät keskustele toisten luokan metodien kanssa ja matala että luokan kaikki metodit käyttävät useita muita luokan metodeja. Matala koheesio on toivottavaa, koska luokan tehtävä on tehdä yksi ja vain yksi asia. 9

kompositio (composition) Luokkien suhteita kuvaava termi. Kompositiossa luokka B on sidottu luokan A elinikään niin, että jokainen A:n instanssi sisältää vakiomäärän viittauksia B:n instansseihin (yleensä 1). Kyseiset B:n instanssit luodaan yhdessä ne omistavan A:n instanssin kanssa ja ne tuhotaan yhdessä ne omistavan A:n instanssin kanssa. Kompositiossa siis luokka B on riippuvainen luokasta A olemassaolonsa ajan. 53, 76, 89

konkreettinen luokka (concrete class) Luokka, joka on abstraktin luokan lapsiluokka eikä ole itse abstrakti luokka. Luokan on implementoitava kaikki kantaluokan abstraktit metodit. 63, *see* abstrakti luokka & abstrakti metodi

korvaaminen (overriding) Lapsiluokan perimän metodin toiminnallisuuden uudelleen määrittely. Tapahtuu määrittelemällä lapsiluokassa metodin, jonka signatuuri on identtinen kantaluokan korvattavan metodin kanssa. Tämän metodin toiminnallisuus ajetaan kantaluokan metodin sijaan. 58, *see* periytyminen, signatuuri, kantaluokka & lapsiluokka

kuluttaja (consumer) Metodi, joka ottaa yhden argumentin eikä palauta mitään, tai kyseisen metodin mukaisen rajapinnan tarjoava luokka. Javassa kuluttaja-suunnittelumallille on olemassa erillinen standardikirjastossa määritelty rajapintaluokka Consumer. 86, *see* rajapinta & Consumer

lapsiluokka (child class) Luokka, joka perii toisen luokan. 54, 62, *see* periytyminen & kantaluokka

Liskovin korvaavuusperiaate (Liskov Substitution Principle - LSP) SOLID-periaatteisiin kuuluva sääntö, joka käsittelee kanta- ja lapsiluokkien suhdetta. Liskovin korvaavuussäännön mukaan, lapsiluokan instanssia pitää pystyä käsittelemään kuin kantaluokan instanssia, eli sen on paljastettava vähintään samat metodit ja muuttuvat kuin kantaluokan instanssien. Korvaavuus ei kulje molempiin suuntiin, vaan lapsiluokan instanssit saavat säännön mukaan sisältää metodeja ja muuttujia, joita kantaluokan instansseista ei löydy. 62, 71, 83, *see* SOLID-periaatteet, kantaluokka, lapsiluokka & periytyminen

lueteltu tyyppi (enumerated type) Ohjelmoijan määrittelemä tyyppi, joka koostuu rajatusta arvojoukosta. Jokaisella arvojoukon arvolla on nimi, jolla siihen viitataan koodissa. Ohjelma käsittelee kuitenkin arvoja kokonaislukuina, nopeuttaen ajamista. Javassa lueteltu tyyppi luodaan enum-avainsanalla. 42, *see* enum

luokka (class) Ohjelmoijan kirjoittama muotti, jonka pohjalta ohjelmisto luo olioita. Voi sisältää metodeja ja datakenttiä mutta yleensä näiden käyttämiseksi vaaditaan olion luontia. 8, *see* olio, metodi & instanssi

luokkakaavio (class diagram) UML-standardin kaavio, joka kuvaa luokkien välisiä suhteita ohjelmistossa. Koostuu luokkia esittävistä yleensä neliön muotosista soluista. Solut sisältävät luokkien tietokentät ja metodit. Luokkia voidaan yhdistää toisiinsa erilaisilla nuolilla osoittamaan niiden välisiä suhteita kuten periytymistä, kompositiota ja aggregaatiota. 55, *see* periytyminen, kompositio & aggregaatio

luokkametodi (class method) Metodi, joka on luokan instanssin sijaan sidottu itse luokkaan. Ei voida käyttää instanssiin sidottuja muuttujia, mutta voi käyttää luokkamuuttujia. Määritellään yleensä static-avainsanalla ja kutsutaan luokan nimen pistenotaatiolla. *see* luokkamuuttuja & static

luokkamuuttuja (class variable) Muuttuja, joka on luokan instanssin sijaan sidottu itse luokkaan. Muokattavissa ja tarkasteltavissa jokaisesta luokan instanssista jaetusti. Määritellään yleensä static-avainsanalla. *see* static

matala kopio (shallow copy) Kopio, joka on tehty luomalla uusi kopioitavan luokan instanssi ja kopioimalla kaikki muuttujat viitekopioina uuden instanssin muuttujiin. Näin luodun uuden olion tila ei ole suoraan riippuvainen vanhan olion tilasta, mutta niillä on yhteysiä viittauskopioita, joten vanhan olion omistaman olion tilan muutos voi vaikuttaa vastaavan olion tilaan uudessa oliossa ja toisin päin. 92, *see* & syva kopio

metodi (method) Luokkaan sidottu käskysarja, joka suorittaa ottamiensa parametrien ja luokan omien datakenttien perusteella jonkin tietyn toiminnon. 7, *see* parametri, funktio & luokka

moniperinta (multiple inheritance) Periytymisen tekniikka, jossa yhdellä lapsiluokalla on useampi kantaluokka. Suora moniperintä ei ole tuettua javassa, mutta useamman rajapintasopimuksen implementoiti yhteen luokkaan onnistuu rajapintaluokilla. 65, *see* periytyminen, lapsiluokka, kantaluokka, rajapinta, &

muuttuja (variable) koodissa määritelty tietokenttä, joka sisältää jonkin ohjelman käyttämän arvon. 12

noutaja (getter) Metodi, jonka tehtävänä on tarjota rajoitettu saatavuus johonkin luokan yksityiseen muuttujan. Ei yleensä ota argumentteja ja palauttaa luokan omistaman toivotun muuttujan. Nimetään camelCase-tyylillä sijoittamalla muuttujan nimen eteen "get" ("getVariable"). Parantaa enkapsulaatiota ja mahdollistaa esimerkiksi laiskan alustuksen muuttujille. 22

näkyvyysmääre (access modifier) Muuttujan näkyvyyden määrittävä avainsana. 20, 59, *see* muuttuja, public, protected & private

olio (object) Luokan instanssi. Yksittäinen koodissa luotu toimija, joka sisältää datakenttiä ja metodeita. Luokka, jonka pohjalta olio luodaan määrittää olion käytettävissä olevat metodit ja siihen tallennetut datatyypit, mutta vain olio pääsee käsiksi omiin metodeihin ja datakenttiinsä. 7, *see* luokka, metodi & instanssi

pakotettu tyypinmuunnos (casting) Operaatio, jossa muuttujan tyyppi yritetään manuaalisesti muuttaa joksikin toiseksi tyypiksi. Voi epäonnistua. Tapahtuu kirjoittamalla toivottu tietotyyppi muutettavan muuttujan nimen eteen sulkuihin. Esimerkiksi double-tyypin muuttuja "doubleNumber" voitaisiin säilöä int-tyypin muuttujaan "intNumber" pakotetun tyyppimuunnoksen avulla seuraavalla koodinpätkällä:
"int intNumber = (int)doubleNumber". 40

parametri (parameter) Arvo, jonka funktio tai metodi ottaa muulta koodilta vastaan. *see* funktio & metodi

pariutuminen (coupling) Ohjelman laadun mittaamiseen käytetty käsite. Mittaa luokkien keskenäisten riippuvuuksien määrää akselilla löysä-tiukka. Löysässä pariutumisessa ohjelmiston luokkien väliset riippuvuudet ovat harvassa, jolloin ohjelmiston muokkaaminen on helppoa. Tiukassa pariutumisessa puolestaan jokaisella ohjelmiston luokalla on riippuvuus moneen muuhun ohjelmiston luokkaan, jolloin ohjelmison muokkaus hankaloituu ja täten voidaan katsoa ohjelmiston laadun laskevan. 9

periytyminen (inheritance) Olio-ohjelmoinnin konsepti, jossa luokat voivat periä toisen luokan ominaisuudet. Luokkaa, joka perii toisen luokan kutsutaan lapsiluokaksi ja luokkaa, jolla on lapsiluokka kantaluokaksi. Lapsiluokka sisältää automaattisesti kaikki kantaluokan kentät, eli muuttujat ja metodit, joiden näkyvyysmääre on protected tai laajempi. Kenttiä voidaan tarvittaessa ylikirjoittaa. 54, 57, *see* lapsiluokka, kantaluokka, , protected & extends

primitiivinen tietotyyppi (primitive data type) Tietotyyppi, jonka ohjelmointikieli kykenee säilömään suoraan muistipaikkaan raakana numeerisena datana. Ainoat tietotyypit Javassa, jotka eivät ole jonkin luokan instansseja. 12, *see* luokka & instanssi

rajapinta (interface) Yleisessä käytössä termi ohjelmistossa toteutuvalle sopimukselle, jonka jokin metodi, luokka tmv. toteuttaa tai kahden ohjelmiston osan välinen taso. Javan yhteydessä luokka, joka luodaan interface-avainsanalla ja joka sisältää vain luokan implementoimien metodien signatuurit. Rajapintaluokka ei yleensä sisällä ollenkaan toiminnallista koodia. 9, 62, 64, *see* interface

rajapintojen erottelu -periaate (Interface Segregation Principle - ISP) SOLID-periaatteisiin kuuluva sääntö, joka käsittelee rajapintojen toiminnallisuuksien rajaamista. Periaatteen mukaan rajapinnan implementoivan luokan ei pitäisi joutua määrittelemään metodeja, joita se ei tarvitse. Käytännössä siis jokainen rajapinta pitäisi suunnitella tarpeeksi pieneksi, jotta kaikki sen implementoivat luokat käyttävät kaikkia sen metodeja see. 71, 75

rakentaja (constructor) Luokkametodi, joka määritellään ja kutsutaan luokan nimellä ja nimensä mukaisesti rakentaa ja palauttaa uuden luokan instanssin. 18, *see* luokkametodi, instanssi & new

roskankeruu (garbage collection) Muistinhallinnan tekniikka, jossa ohjelma tietyin väliajoin ajaa aliohjelman, joka käy läpi kaikki muuttujat muistissa, tarkistaa onko niihin olemassa viittauksia ja poistaa muuttujat, joilla ei ole enää aktiivista viittausta. 37

serialisaatio (serialization) Erityisesti olio-ohjelmoinnissa yleinen ohjelmointikielten ominaisuus, joka mahdollistaa olioiden esittämisen

tekstimuotoisena datana. Tällä tavalla esitettyjä olioita on helpompi käsitellä esimerkiksi tiedostojenhallinnan yhteydessä tai verkkoliikenteessä. 66

signatuuri (signature) Metodin määritelmä, joka sisältää sekä metodin nimen, että sen ottamien argumenttien tyypit. 14, 58, 64

singleton (singleton) Singleton viittaa sekä singleton-suunnittelumalliin ja suunnittelumallin toteuttavaan luokkaan. Suunnittelumallissa luokka säilyttää itsetään yhtä instanssia yksityisessä luokkamuuttujassa ja sisältää julkisen getInstance-metodin, joka joko luo uuden instanssin, mikäli luokasta ei ole vielä olemassa instanssia, tai noutaa olemassa olevan instanssin. Luokkaa, joka toteuttaa tämä suunnittelumallin sanotaan myös singletoniksi. Suunnittelumallia on kritisoitu, koska se luo ohjelmalle jaetun globaalin tilan ja on hankala testata, mutta sillä on meriittinsä esimerkiksi loggaamisessa. 69, *see* luokkamuuttuja

sisaluokka (inner class) Toisen luokan rungon sisällä määritelty luokka. Luokka näkee kaikki ulomman luokan muuttujat. Toisin kuin normaali luokka, sisäluokka voidaan määritellä myös protected tai private näkyvyysmääreellä, jolloin vain luokka, jonka rungossa sisäluokka on määritelty ja mahdollisesti luokan lapsiluokat näkevät sisäluokan. 89, *see* protected, private & näkyvyysmääre

SOLID-periaatteet (SOLID-principles) Viisi oliopohjaisen ohjelmoinnin käyttöön kehitettyä periaatetta, joiden tavoitteena on auttaa laadukkaan, siistin, helposti laajennettavan ja helposti luettavan oliopohjaisen ohjelmiston tuottamisessa. Periaatteet ovat "yhden vastuun periaate" (Single Responsibility Principle - SRP), "avoin/suljettu-periaate" (Open/Closed Principle - OCP), "Liskovin korvaavuus-periaate" (Liskov Substitution Principle - LSP), "rajapintojen erottelu -periaate" (Interface Segregation Principle - ISP) ja "käänteisten riippuvuuksien periaate" (Dependency Injection Principle - DIP). 62, 71, *see* yhden vastuun periaate, , , &

staattisesti tyypitetty kieli (statically typed language) Ohjelmointikieli, joka tietää jokaisen koodissa esiintyvän muuttujan tietotyypin koko ajan. 12

strategia (strategy) Suunnittelumalli, jossa jokin abstrakti kantaluokka tai rajapintaluokka määrittelee jonkin metodin, jonka useampi lapsiluokka toteuttaa. Tätä metodia kutsutaan sitten koodissa sen määrittelemän abstraktin kantaluokan tai rajapintaluokan tyyppiseksi määritellystä muuttujasta. Tällöin muuttujan tyyppi määrittelee, miten metodi käyttäytyy, sillä jokainen abstraktin kantaluokan konkreettinen lapsiluokka, tai rajapintaluokan implementoiva luokka on määritellyt itse oman tapansa toteuttaa metodi. Käytännössä kaikki abstraktit metodit, niin abstrakteissa luokissa kuin rajapintaluokissa täyttävät strategia-suunnittelumallin määritelmät. 72, 73, 75, *see* abstrakti luokka, rajapinta & abstrakti metodi

syva kopio (deep copy) Kopio, joka on tehty luomalla uusi kopioitavan luokan instanssi ja kopiomalla kaikki muuttujat rekursiivisesti syvinä kopioina uuden instanssin muuttujiin. Riippuen riippuvuusketjun pituudesta, voi tällainen kopiointi olla raskasta, mutta tuloksena on kopio, jonka tila on aidosti riippumaton alkuperäisen olion tilasta. 92, *see* & matala kopio

taulukko (array) Tietorakenne, jossa yksittäiset alkiot on säilötty vierekkäisiin muistipaikkoihin. Vie vähän muistitilaa ja mahdollistaa nopeat haut ja lisäykset jos alkion indeksi on tiedossa etukäteen. Yleensä taulukon koko on määriteltävä sen luomisen yhteydessä, tämä on totta myös Javassa. Javassa taulukko luodaan new-avainsanalla lisäämällä taulukon säilömän tietotyypin perään hakasulkeet, joiden sisään suljetaan taulukon koon määrittelevä numero. Taulukkotyyppinen muuttuja määritellään lisäämällä muuttujan tietotyypin määritelmän perään hakasulkeet. Tällöin esimerkiksi int-tyypin kymmenen alkion taulukko, joka on säilötty "intArray"-muuttujaan määriteltäisiin seuraavasti:
"int[] intArray = new int[10]". 27

tietue (struct) Vanhahko muokattava tietotyyppi, yleinen esimerkiksi C-kielessä. Käyttäjä voi määritellä tietueen sisältämään mitä tahansa vakiokokoisia datakenttiä. Luokkien edeltäjä. 7

toteuttaa (implement) Rajapinnan toteuttaminen tarkoittaa, että jokin luokka tai metodi toimii kuten jokin rajapintamääritelmä sanoo sen toimivan. Javassa puhutaan rajapintaluokan toteuttamisesta käytettäessä implements avainsanaa luokan määritelmässä. Tällöin rajapinnan toteuttavan luokan voi olettaa toimivan kuten rajapintaluokka määritää, eli luokkaa voidaan käsitellä kuten toteutetun rajapintaluokan instanssia. 64, *see* rajapinta, luokka & implements

tulkattu kieli (intrepeted language) Ohjelmointikieli, joka ajetaan lukemalla kooditiedostot reaaliajassa. Esimerkiksi Python ja JavaScript ovat tulkattavia kieliä. 36

tyyppiparametri (type parameter) Geneerisen luokan tai metodin tyypin määrittävä parametri. Annetaan pienmpi kuin- ja suurempi kuin -metkkien välissä. 81, *see* geneerinen luokka &

UML (Unified Modelin Language) Mallinnustekniikka ohjelmistojen rakenteen esittelyyn. UML-standardi sisältää monia erilaisia kaavioita, mutta olio-ohjelmoinnin kannalta niistä tärkein on luokkakaavio. Muita tärkeitä kaavioita ovat oliokaavio, tilakaavio ja sekvenssikaavio. 55, 56, 76, *see* luokkakaavio

viitekopio (reference copy) Kopio, joka on tehty luomalla uusi viittaus kopioitavaan olioon. Viittauskopion ja alkuperäisen olion tilat ovat sidottuja toisiinsa: mikäli toista muokataan, toinen muuttuu. 92, *see* matala kopio & syva kopio

yhden vastuun periaate (Single Responsibility Principle - SRP) SOLID-periaatteisiin kuuluva sääntö, joka käsittelee koodin jakamista luokkiin. Periaatteen mukaan jokaisella luokalla pitäisi olla yksi ja vain yksi vastuu. Esimerkiksi String-luokan vastuu on vain merkkijonotyyppisen datan säilöminen ja sen vastuisiin ei kuulu esimerkiksi datan esittäminen. 71, 85, *see* SOLID-periaatteet

ylikuormitus (overloading) Kahden samannimisen metodin määrittäminen eri argumenteilla. Argumenttejä voi olla sama määrä, mutta eri tyypeillä, tai eri määrä. Java hakee metodikutsun yhteydessä automaattisesti kutsuttua signatuuria vastaavan version ylikuormitetusta metodista. 19

Javan avainsanat

abstract Avainsana, joka luo abstraktin luokan tai abstraktin metodin. 63, *see* abstrakti luokka & abstrakti metodi

ArrayList Javan standardikirjaston luokka, joka toteuttaa List-rajapinnan. Geneerinen säilötyn luokan suhteen, eli voi säilöä minkä tahansa luokan olioita, kunhan säilötyt oliot ovat kaikki saman luokan instansseja. Ei voi säilöä primitiivisiä tietotyyppejä. Luodaan new-avainsanalla normaalin geneerisen luokan tapaan. Merkkijonoja säilövä ArrayList-instanssi voidaan siis luoda seuraavalla koodinpätkällä:
"new ArrayList<String>();". 29, 30, 62, *see* List, geneerinen luokka & new

bool Primitiivinen tietotyyppi, joka sisältää totuusarvon (true tai false). 13, *see* primitiivinen tietotyyppi

break Avainsana, joka lopettaa toistorakenteen toiston kokonaan tai poistuu switch-rakenteesta. Käytettävissä kaikissa loopeissa ja switch-valintarakenteessa. 35, 38, *see* for, while, do & switch

BufferedReader Javan standardikirjaston luokka puskuroitua merkkijonostream-olion lukemista varten. Rakentaja ottaa luettavan stream-olion ja vapaaehtoisena argumenttina puskurin koon. Käytetään isompien syötteiden lukemiseen, koska syötteen kääriminen BufferedReader-olioon luettaessa vähentää turhia lukuoperaatioita. Scanner-luokka on suositellumpi esimerkiksi lyhyen käyttäjäsyötteen lukemiseen. 24, 25, 49, 50, *see* Scanner

byte Primitiivinen tietotyyppi, joka sisältää tavun kokoisen merkillisen kokonaisluvun. 13, 43, *see* primitiivinen tietotyyppi

case Avainsana, joka määrittää yksittäisen haaran switch-avainsanalla luodun valikkorakenteen sisällä. Avainsanaa seuraa välilyönnillä erotettu haaran arvo, jota verrataan switch-avainsanan määrittämään valintaehtoon. Haarat, joiden arvo toteutuu valintaehdossa ajetaan järjestyksessä. On normaalia päättää haara break-avainsanaan, joka lopettaa koko rakenteen ajamisen, ellei tavoitteena ole ajaa useampaa haaraa peräkkäin. Oletushaara, joka ajetaan, mikäli mikään haara ei vastaa valintaehtoa voidaan määrittää default-avainsanalla. 38, *see* switch, break & default

catch Avainsana, joka määrittää virheenkäsittelyrakenteen virhetilankorjausosion Try-avainsanalla aloitetussa virheenkäsittelyrakenteessa. Avainsanaa seuraa sulkuihin suljettuna määritelmä muuttujalle, johon mahdollinen kiinni otettu virhe tallennetaan. Määritelmä määrittää myös kiinni otettavan virheen tyypin. Jos virhe ei peri määriteltyä virhetyyppiä catch-rakenne ei nappaa virhettä. Yhtä try-avainsanalla määriteltyä rakennetta kohden voi olla useampi catch-rakenne, kunhan ne nappaavat erityyppisiä virheitä. 47, 48, *see* try & finally

char Primitiivinen tietotyyppi, joka sisältää kahden tavun kokoisen unicode-koodatun merkin esitettynä merkittömänä kokonaislukuna. 13, *see* primitiivinen tietotyyppi

class Avainsana, joka aloittaa luokan määritelmän. 14, 42, 63, 64

Cloneable Standardikirjaston rajapintaluokka, joka määrittää implementoivan olion olevan kloonattavissa clone-metodilla. Rajapinnan implementoivan luokan on ylikirjoitettava Object-luokan määrittelemä clone-metodi. 92, *see* rajapinta & Object

Consumer Standardikirjaston rajapintaluokka, joka määrittelee niin sanotun kuluttajan. Kuluttaja on geneerinen olio, joka on parametrisoitu tyypillä T ja jolla on yksi metodi, joka ei palauta mitään ja ottaa argumentiksi yhden tyypin T instanssin. Tämän metodin nimi rajapinnassa on accept. Rajapinta tarjoaa lisäksi andThen-metodin, joka ottaa toisen kuluttajan, jonka täytyy myös olla parametrisoitu tyypillä T. Metodi yhdistää kuluttajien kutsut ja palauttaa kutsut yhdistävän kuluttajainstanssin, joka kutsuu vuorotellen kummankin kuluttajan accept-metodit. 85, 86, *see* kuluttaja, interface, geneerinen luokka & tyyppiparametri

continue Avainsana, joka lopettaa toistorakenteen toiston ja aloittaa seuraavan toiston saman tien. Käytettävissä kaikissa loopeissa. 35, *see* for, while & do

default Avainsana, joka määrittää oletushaaran switch-avainsanalla luodussa valikkorakenteessa. Oletushaara ajetaan, jos valikkorakenteen valintaehto ei täsmää mitääm muuta rakenteessa määriteltyä haaraa. 38, 64, *see* switch

do Avainsana, joka luo do-while -loopin. Avainsanaa seuraa loopin runko kaarisulkuihin suljettuna. Rungon jälkee määritellään loopin toistoehto while-avainsanalla, niin, että avainsana kirjoitetaan rungon perään ja toistoehto suljetaan avainsanan jälkeen sulkuihin. Toisin kuin while-looppi, joka tarkistaa toistoehdon ennen jokaista toistoa, do-while tarkistaa toistoehdon jokaisen toiston jälkeen. Tukee continue- ja break-avainsanoja. 34, *see* while, continue & break

double Primitiivinen tietotyyppi, joka sisältää 64 bitin kokoisen liukumaesitetyn desimaaliluvun. 13, *see* primitiivinen tietotyyppi

else Avainsana, joka voi seurata if-lauseella määriteltyä ehdollista koodin osaa. Else-lausetta seuraa kaarisulkuihin suljettu koodin osa. Tämä koodin osa ajetaan vain, jos else-lausetta edeltänyttä ehdollista koodin osaa ei ajettu. 17, 37, *see* if

enum Avainsana, joka määrittää luetellun tyypin. Korvaa class-avainsanan luokan määrittelyssä. Luetellun tyypin määritelmä tarvitsee runkoonsa ainakin tyypin arvojoukon. Yleensä tämän arvojoukon jäsenet kirjoitetaan CONSTANT_CASE-tyylillä. 42, 73, *see* lueteltu tyyppi & class

Exception Standardikirjaston luokka, joka toimii kaikkien korjattavissa olevien virhetilojen kantaluokkana. 77

extends Avainsana, joka määrittää luokan kantaluokan periytymisessä. Sijoitetaan luokan nimen määritelmän perään niin, että avainsanaa seuraa kantaluokan nimi. Esimerkiksi luotaessa luokkaa "ChildClass", joka halutaan periyttää kantaluokasta "ParentClass", kirjoitettaisiin luokan määritelmä seuraavasti:
"public class ChildClass extends ParentClass..."
Lisäksi avainsana toimii geneerisen luokan tai metodin tyyppiparametrin rajaamisessa syntaksilla
"<T extends UpperBoundClass>"
Tällöin kaikkien tyyppien T pitää olla joko UpperBoundClass luokan, tai siitä periytyvien luokkien instansseja. 57, 81, *see* periytyminen, lapsiluokka, kantaluokka & tyyppiparametri

FileInputStream Standardikirjaston luokka, joka sisältää tiedostoa edustavan syötteen tavuina esitettynä virtana. Omistaa read-metodin syötteen lukemiseen. InputStream-luokan konkreettinen lapsiluokka. 47, 50

FileOutputStream Standardikirjaston luokka, joka sisältää tiedostoon kirjoitettavaa dataa edustavan tavuina esitetyn virran. Omistaa write-metodin kirjoittamiseen. OutputStream-luokan konkreettinen lapsiluokka. 47, 50, *see* OutputStream & lapsiluokka

FileReader Standardikirjaston luokka, joka lukee merkkijonomuotoista dataa rakentajassa määritellystä tiedostosta. Perii InputStreamReader- ja Reader -luokat ja sisältää niistä perityn read-metodin lukuoperaatioita varten. 47–50

FileWriter Standardikirjaston luokka, joka kirjoittaa merkkijonomuotoista dataa rakentajassa määriteltyyn tiedostoon. Perii OutputStreamWriter- ja Writer -luokat ja sisältää niistä perityt write- ja append -metodit kirjoitusoperaatioita varten. 47, 48

final Avainsana, joka määrittää muuttujan olevan vakio, tai luokan olevan mahdoton periyttää. Voi sijaita ennen tai jälkeen näkyvyysmääreen ja mahdollisen static-avainsanan. Normaalisti viimeinen näistä kolmesta. 41, 47, 64

finally Avainsana, joka määrittää virheenkäsittelyrakenteen jaetun lopetusosion. Finally-osio on aina try-avainsanalla aloitetun virheenkäsittelyrakenteen viimeinen osio ja se ajetaan riippumatta siitä, aiheuttiko try-osion koodi virhetilan vai ei. Finally-osio sisältää yleensä koodia, joka varmistaa ohjelmiston olevan jatkamiseen kelpaavassa tilassa virheenkäsittelyn jälkeen. 47, 48

float Primitiivinen tietotyyppi, joka sisältää 32 bitin kokoisen liukumaesitetyn desimaaliluvun. 13, 40, *see* primitiivinen tietotyyppi

for Avainsana, joka alustaa for-loopin tai foreach-loopin. Avainsanaa seuraa loopin toistoehto sulkuihin suljettuna. Toistoehto määrittää miten kauan looppia ajetaan. For-loopissa ehto on kolmiosainen: ensimmäinen osa kertoo mikä muuttuja määrittää toistoehdon, toinen millä kyseisen muuttujan arvoilla toistoa jatketaan ja kolmas mitä muuttujalle tehdään jokaisen toiston jälkeen. Foreach-loopissa toistoehto määrittää kokoelman, jonka yli looppi iteroi. Kummassakin looppityypissä toistoehdon jälkeen seuraa loopin runko suljettuna kaarisulkuihin. Molemmat loopit tukevat continue- ja break-avainsanoja. 32, 33, *see* continue & break

HashMap Javan standardikirjaston luokka, joka toteuttaa hajautuskartta-tietorakenteen. Geneerinen luokka, joka ottaa erikseen avaimen ja arvon tietotyypin. Osaa säilöä myös null-arvoja avaimeen tai arvoon. Luodaan new-avainsanalla normaalin geneerisen luokan tapaan. HashMap, jossa on merkkijonoavaimet ja kokonaislukuarvot voidaan siis luoda seuraavalla koodinpätkällä:
"new HashMap<String, int>"
Huomioi, että avaimen tyyppi annetaan ensimmäisenä. 30, 44, *see* geneerinen luokka & hajautustaulu

if Avainsana, jota seuraa normaaleihin sulkuihin suljettu totuusarvo ja kaarisulkuihin suljettu koodin osa. Tämä koodin osa ajetaan vain jos annettu totuusarvo on tosi. Voidaan yhdistää else-lauseeseen. 17, 37, *see* else

implements Avainsana, joka määrittää luokan implementoivan jonkin rajapinnan. Tällöin kyseiseen luokkaan on määriteltävä metodit, jotka vastaavat kyseisessä rajapinnassa määriteltyjä metodeja signatuuriltaan. Esimerkiksi luokka "PasswordLogin", joka implementoi "Authenticator"-rajapinnan määritellään seuraavasti:
"class PasswordLogin implements Authenticator...". 66, *see* class & interface

import Avainsana, joka määrittää luokan tuotavaksi tiedoston nimitilaan. 24, 42

InputStream Abstrakti kantaluokka, joka lukee tavudataa lapsiluokassa määritellystä lähteestä. 67, *see* abstrakti luokka & FileInputStream

InputStreamReader Javan standardikirjaston luokka tavujonon merkkijonoksi muuttamista varten. Rakentaja ottaa muutettavan streamolion ja vapaaehtoisesti merkkisetin. 24, 25, 50

instanceof Avainsana, joka tarkistaa onko olio annetun luokan instanssi. Käyttö tapahtuu muodossa "olio instanceof luokka". 66

int Primitiivinen tietotyyppi, joka sisältää 32 bitin kokoisen merkillisen kokonaisluvun. 13, 40, 43, *see* primitiivinen tietotyyppi

interface Avainsana, joka aloittaa rajapintaluokan määritelmän Javassa normaalin class- avainsanan sijaan. Rajapintaluokka määrittää signatuurit metodeille, jotka kaikkien rajapinnan implementoivien luokkien on toteutettava. Näin luotu rajapintaluokka ei voi sisältää datakenttiä eikä sitä voi käyttää olion muottina. Sen sijaan toinen luokka voi implementoida rajapinnan jolloin implementoiva luokka lupaa muulle ohjelmistolle tarjoavansa rajapinnan määrittelemät funktiot osana toiminnallisuuttaan. 9, 64, *see* rajapinta & implements

Iterator Standardikirjaston rajapintaluokka, joka mahdollistaa iteroinnin jonkin kokoelman yli. Luodaan kutsumalla iterator-metodia iteroitavasta kokoelmasta. Tarjoaa metodit next, hasNext, remove ja forEachRemaining. 85, *see* iterator, ListIterator & listIterator

iterator Standardikirjaton kokoelmien omistama metodi, joka luo Iterator-olion kyseisestä kokoelmasta. 85, *see* Iterator

List Javan standardikirjaston rajapinta, jonka instanssit ovat järjestettyjä kokoelmia. Kaikki List-rajapinnan toteuttavat luokat tukevat iteraattorin luontia, instanssien vertailua ja indeksipohjaista hakua ja lisäystä. Yleisimmin käytetty List-rajapinnan toteuttava luokka on ArrayList, mutta esimerkiksi myös LinkedList ja ArrayQueue toteuttavat rajapinnan. 85, *see* ArrayList, interface & rajapinta

ListIterator Standardikirjaston rajapintaluokka, joka mahdollistaa iteroinnin jonkin List-rajapinnan implementoivan kokoelman yli. Rajapinnan implementoivan luokan instanssi saadaan kutsumalla listIterator metodia iteroitavasta List-instanssista. Rajapinta tarjoaa metodit, next, hasNext, nextIndex, previous, hasPrevious, previousIndex, remove, add(E element) ja set(E element). 85, *see* listIterator, Iterator & iterator

listIterator Standardikirjaston List-rajapintaluokan olioiden omistama metodi, joka luo ListIterator-olion kyseisestä List-instanssista. 85, *see* ListIterator & List

long Primitiivinen tietotyyppi, joka sisältää 64 bitin kokoisen merkillisen kokonaisluvun. 13, 40, 43, *see* primitiivinen tietotyyppi

main Varattu metodinimi metodille, jonka Java ajaa ensimmäisenä ajaessaan ohjelmistoa. Ohjelmistossa voi olla useampi-main nimi-

nen metodi, mutta vain määritellyn juuriluokan main-metodi ajetaan. Metodin täytyy olla muotoa public static void ja ottaa yksi taulukko String-luokan instansseja. 14, 27, *see* metodi & taulukko

new Avainsana, joka luo uuden instanssin luokasta. Avainsanaa seuraa aina toivotun luokan nimi, jonka perässä on sulkuihin suljettuna toivotut rakentajalle annettavat parametrit, metodikutsun tapaan. 18, 19, 27, 29, 77, 90, *see* rakentaja

Object Standardikirjaston luokka, josta kaikki muut Javan oliot periytyvät. Sisältää fundamentaalisia, kaikille olioille saatavilla olevia metodeja, kuten hashCode, jota käytetään HashMap-luokkaa luotaessa ja equals(Object), jota käytetään vertailussa. 92, *see* HashMap

ObjectInputStream Javan standardikirjaston luokka, jota käytetään Serializable-rajapinnan implementoivien luokkien deserialisaatioon. Luokan rakentaja ottaa vastaan InputStream-luokan konkreettisen lapsiluokan. Instanssista voidaan kutsua readObject-metodia, joka palauttaa annetusta InputStream-oliosta luetusta tavudatasta deserialisoidun olion. 67, *see* InputStream, Serializable & serialisaatio

ObjectOutputStream Javan standardikirjaston luokka, jota käytetään Serializable-rajapinnan implementoivien luokkien serialisaatioon. Luokan rakentaja ottaa vastaan OutputStream-luokan konkreettisen lapsiluokan. Instanssista voidaan kutsua writeObject-metodia serialisoitavalla oliolla, jolloin luokka kirjoittaa annetun olion serialisoidun tekstimuodon annettuun OutputStream-olioon. 67, *see* OutputStream, Serializable & serialisaatio

OutputStream Abstrakti kantaluokka, joka vastaanottaa tavudataa ja syöttää sen datankuluttajaan, joka määritellään lapsiluokassa. 67, *see* abstrakti luokka & FileOutputStream

private Näkyvyysmääre, joka määrittää ominaisuuden olevan käytettävissä vain ominaisuuden omistaman luokan sisällä. 20–22, 59, 89

protected Näkyvyysmääre, joka määrittää ominaisuuden olevan käytettävissä ominaisuuden omistavan luokan sisältävässä packagessa ja kaikissa ominaisuuden omistavan luokan perivissä luokissa. 20, 21, 59, 60, 63, 89

public Näkyvyysmääre, joka määrittää ominaisuuden olevan käytettävissä kaikkialla ohjelmistossa. 14, 20, 21, 59, 64

return Avainsana, joka aiheuttaa metodista poistumisen ja palauttaa avainsanaa seuraavan arvon. Esimerkiksi "return 4", poistuisi metodista saman tien, palauttaen kokonaislukuarvon 4. 16, 18

Scanner Javan standardikirjaston luokka halutun lähteen lukemiseen ja parsimiseen. Rakentaja ottaa parsittavan stream-olion. Sisältää metodeja eri tietotyyppien parsimiseen annetusta stream-oliosta. 24, 25

Serializable Rajapintaluokka, joka määrittelee luokan olevan serialisoitavissa. Jotta luokka voi implementoida Serializable-rajapinnan täytyy kaikkien luokan muuttujien olla myös serialisoitavia. Suurin osa standardikirjaston luokista, sekä kaikki primitiiviset tietotyypit ovat serialisoitavia, joten tämä rajoitus koskee lähinnä muuttujia, joiden tietotyyppi on jokin käyttäjän määrittelemä luokka, tai jotka ovat geneerisiä käyttäjän määrittelemän luokan yli. 66, 67, *see* primitiivinen tietotyyppi &

short Primitiivinen tietotyyppi, joka sisältää kahden tavun kokoisen merkillisen kokonaisluvun. 13, 43, *see* primitiivinen tietotyyppi

static Staattinen metodi tai muuttuja näkyy kaikille sen omistavan luokan instansseille jaetusti. Tunnetaan luokkamuuttujana tai luokkametodina. 14, 39, 64, 89, 91

String Javan standardikirjaston merkkijonoimplementaatioluokka. Suositellaan käytettäväksi merkkijonojen säilömiseen koodissa. Pystyy säilömään dynaamisen merkkijonon, jonka alustus- tai maksimikokoa ei tarvitse määrittää erikseen. Lisäksi sisältää lukuisia merkkijonon käsittelyä ja muokkausta helpottavia metodeja. 13, 14

super Avainsana, joka viittaa lapsiluokan kantaluokkaan instanssiin lapsiluokan instanssin sisällä. Voidaan käyttää myös kantaluokan rakentajaan viittaamiseen, mikäli avainsanaa käytetään funktiokutsuna ("super()" tai "super(args)"). 60, *see* periytyminen, lapsiluokka, kantaluokka & rakentaja

switch Avainsana, joka aloittaa valikkorakenteen. Avainsanaa seuraa valintaehto sulkuihin suljettuna. Valikkorakenne koostuu case-avainsanalla määritellyistä haaroista ja mahdollisesta default-avainsanalla määritellystä oletushaarasta. Koko valikkorakenne suljetaan kaarisulkuihin. Valikkoehto lasketaan rakenteen alussa ja kaikki haarat, joiden arvo vastaa laskettua ehtoa ajetaan järjestyksessä. Valikkorakenteesta voi poistua break-avainsanalla, mutta se ei tue continue-avainsanaa. On normaalia päättää jokainen haara break-avainsanalla, jotta haaroja ei ajettaisi vahingossa useampaa, ellei useamman haaran ajaminen ole tarkoituksenmukaista. 37, 38, 72, 73, *see* case, default & break

System Javan standardikirjaston luokka, joka sisältää luokkametodeja ja luokkamuuttuja systeemirajapintojen, kuten ympäristömuuttujien, tulosteen ja syötteen käyttöön. 14, 15, 25, 39, *see* luokkamuuttuja & luokkametodi

this Avainsana, joka viittaa luokan instanssiin luokan omistaman instanssimetodin tai luokan rakentajan sisällä. Ei toimi luokkametodeissa, koska niissä puuttuu instanssi, johon viitata. 18

throw Avainsana, joka aiheuttaa virhetilan. Käytetään antamalla nostettava virhetilainstanssi avainsana jälkeen. Esimerkiksi "throw new NullPointerException("oops")" aiheuttaa NullPointerException-luokan virhetilan, jonka viestiksi on määritelty "oops". 77, *see* try, catch, finally & throws

throws Avainsana, joka ilmoittaa metodin määritelmässä metodin voivan aiheuttaa jonkin virhetilan. Avainsana sijaitsee metodin nimen ja parametrien määritelmän jälkeen ja virhetilaluokka tai luokat listataan avainsanan jälkeen. Esimerkiksi "public void doStuff(String arg) throws NullPointerException" luo metodin nimeltä doStuff, joka ilmoittaa voivansa aiheuttaa NullPointerException-luokan virhetilan. 78, *see* try, catch, finally, throw & metodi

try Avainsana, joka määrittää virheenkäsittelyrakenteen osion, joka sisältää mahdollisen virhetilan aiheuttavan koodin. Try-osioita täytyy seurata joko yksi tai useampi catch-osio, finally-osio tai molemmat. 47–49, *see* catch & finally

void Muuttujan paluuarvotyyppi muuttujalle, joka ei palauta mitään. 14, 15

while Avainsana, joka luo while-loopin. Avainsanaa seuraa loopin toistoehto sulkuihin suljettuna totuusarvona. Tämän jälkeen annetaan loopin runko kaarisulkeisiin suljettuna. Runkoa toistetaan niin kauan, kunnes annettu totuusarvo on epätosi. Tukee continue- ja break-avainsanoja. Määrittää toistoehdon do-loopissa. 33, 34, *see* continue, break & do