

EETU JUHANI ASIKAINEN

# OHJELMOINTIOPAS<sub>LAPPEENRANNAN-</sub>

LAHDEN TEKNILLISEN YLIOPISTON LUT:N OLIO-OHJELMOINTIKURSSI

LISÄÄ JULKAISIJA?

Copyright © 2021 Eetu Juhani Asikainen

PUBLISHED BY LISÄÄ JULKAISIJA?

TUFTE-LATEX.GOOGLECODE.COM

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at <http://www.apache.org/licenses/LICENSE-2.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

*First printing, October 2021*

# Contents

<i>Oppaasta ja sen käytöstä</i>	5
<i>Oliopohjainen ajattelu</i>	9
<i>Javan perusteet</i>	13
<i>Kokoelmarakenteet, toistorakenteet ja lisää oliopohjaista ajattelua</i>	29
<i>Sanasto</i>	33
<i>Javan avainsanat</i>	37



# *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 proceduraalisen ohjelmoinnin perusteet, jotka käydään läpi esimerkiksi Lappeenrannan-Lahden teknillisen yliopiston LUT:n "Ohjelmoinnin Perusteet" -kurssilla.

## *Oppaan rakenne*

OPAS ON JAETTU kahteentoista sisältöluukuun ja kahteentoista bonusluukuun. Sisältöluvut käsittelevät kurssilla käsiteltäviä sisältöjä ja sisältävät kaiken kurssin sisältöön kuuluvan materiaalin. Sisältöluvut 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.

JOKAISTA SISÄLTÖLUKUA vastaa oppaan toisella puoliskolla yksi bonusluku. Bonusluvut tarjoavat laajempia ja syvempiä katsauksia Javaan ja oliopohjaiseen ohjelmointiin. Ne eivät aina syvennä suoraan vastaavan sisältöluvun materiaaliin liittyviä aiheita, mutta myös niiden järjestys on suunniteltu niin, että käyttäjä voi opiskella yhden luvun viikossa.

BONUSLUVUT ON TARKOITETTU hyödyllisiksi ja hauskoiksi lisälukemistoiksi. Opas on suunniteltu niin, että bonusluvut eivät sisällä kurssin oppimistavoitteisiin luokiteltua sisältöä. Niiden lukeminen on siis täysin vapaaehtoista.

### *Termistö ja kieli oppaassa*

OPPAAN PÄÄASIAALLISENA kielenä koodiesimerkkejä lukuunottamatta toimii suomi muun kurssimateriaalin kielivalinnan mukaan. Oppaan käyttämä termistö on tämän vuoksi käännetty Javan avainsanoja lukuunottamatta suomeksi. Olisi kuitenkin 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 opetteluun 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 kirjoitusasuuna ja termin perässä esiintyvänä sulkuihin suljettuna englanninkielisenä vastineena termille. Vastine on niin ikään kursivoitu. Koko korostettu kirjoitusasu termille näyttää siis tältä: *esimerkki (example)*

OPPAAN 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 *\*tänne\**.

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 monisanaisempaa ja yleisempää kuin suosituksissa.





# Oliopohjainen ajattelu

## *Abstraktio laadukkaan koodin pohjana*

OLIPOHJAINEN 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 kutsusarjojen eristäminen ennalta määriteltuihin koodipaloihin. Nämä palat tunnetaan nimellä *funktio* (*function*). Tällä tavalla jaettuja kutsusarjoja voidaan uusiokäyttää ja ohjelman muokkaaminen helpottuu, kun kutsusarjan 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. Tällaisesta hyviä esimerkkejä ovat esimerkiksi Pythonin class-avainsanalla määritellyt luokat, jos niitä käytetään vain datan ryhmittämiseen tai C-kielen *tietue* (*struct*).

NÄMÄ ABSTRAKTIOITYYPIT ovat kuitenkin erotettuja toisistaan, eivätkä huomioi käskysarjojen ja datan yhteyttä. Lähes poikkeuksetta tiettyjä funktioita kutsutaan koodissa monessa paikassa parametreinä toistuvasti sama tietorakenne. Eikö siis olisi järkevää yhdistää datan ja sitä käsittelevien funktioiden sijainti koodissa?

## *Oliopohjaisen ajattelun perusteet*

OLIO-OHJELMOINNIN OHJELMISTO koostuu kokonaan keskenään kommunikoivista yksiköistä. Näille yksiköille vakiintunut kutsuma-

nimi on *olio* (*object*). Oliot koostuvat datasta ja sitä manipuloivista kääntäjäkoosteista, jotka tunnetaan nimellä *metodi* (*method*). Se on oliopohjaisen ohjelmoinnin nimi olioon sidotulle funktiolle, termiä *funktio* ei juuri käytetä.

OHJELMISTO LUO kaikki käyttämänsä oliot ohjelmoijan tekemien muuttien pohjalta. Tällainen muotti, eli *luokka* (*class*) voi luoda itsensä 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 instanssille 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" viitattaessa 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 autorisaatiomodulin 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 esiteltäviä 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 rajoittaa 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 samankaltaiseen "interface"-avainsanaan, joka esitellään myöhemmin.

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 ARVIODA 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 TARKOITTAÄ ohjelmiston luokkien riippuvuutta toisista ohjelmiston luokista. Siitä puhutaan usein asteikolla *löysä* (*loose*)-*tiukka* (*tight*). Löysästi pariutuneessa 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.

```

// Javassa on kaksi tapaa kommentoida kirjoitettua koodia. Ensimmäinen on aloittaa kommentti
// kahdella kauttaviivalla. Tämä merkitsee kääntäjälle rivin loppuosan olevan kommenttia, jolloin
// 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; // Pakettimääritelmä toimii vaikka sitä seuraa kommentti

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

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

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

    /* Mutta esimerkiksi myös tällainen 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");
    }
}

```

Listing 1: Kommentointi Javassa

## 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 simppelein 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 primitiivistä 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
<u><a href="#">bool</a></u>	false	true ja false
<u><a href="#">byte</a></u>	0	$-2^7$ :stä $2^7 - 1$ :een
<u><a href="#">short</a></u>	0	$-2^{15}$ :stä $2^{15} - 1$ :een
<u><a href="#">int</a></u>	0	$-2^{31}$ :stä $2^{31} - 1$ :een
<u><a href="#">long</a></u>	0L	$-2^{63}$ :stä $2^{63} - 1$ :een
<u><a href="#">float</a></u>	0.0f	noin $1.4e - 45$ :stä noin $3.4e + 38$ :aan
<u><a href="#">double</a></u>	0.0d	noin $4.9e - 324d$ :stä noin $1.8e + 308d$ :hen
<u><a href="#">char</a></u>	0	0:sta $2^{16} - 1$ :een

NÄIDEN TIETOTYYPPIEN lisäksi Javasta löytyy monta valmiiksi määriteltyä luokkaa datan säilömistä varten. Näistä tärkein kielin opettelua aloitettaessa on ehdottomasti merkkijonoja säilövä *String*-luokka. Tämä luokka mahdollistaa dynaamisten merkkijonon 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.

```

package week2;

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

    // Primitiivisten tietotyyppien luominen

    // boolean määritetään "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-tietotyyppin määrittämiseen vaadittu L-pääte
    long maxLongValue = 9223372036854775807L;
    // float ja double ilmoitetaan f- ja d-päätteillä
    float floatValue = 1.1f;
    double doubleValue = 3.141592653589793238462643383279502884197169399375105820974944592d;
    // char-arvot määritellään ympäröimällä yksittäinen merkki tai unicode-koodi yksittäisillä
    // lainausmerkeillä (')
    char charValue = 'a';
    char unicodedCharValue = '\u0123';
}

```

Listing 2: Primitiiviset tietotyypit Javassa

```

package week2;

class BasicString {

    public static void main(String[] args){
        String exampleString = "Example string";
        System.out.println(exampleString);
    }
}

```

Listing 3: String-tyypin muuttujan määrittäminen



## 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 kappaleessa ja static-avainsana kappaleessa `*x*`. 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 käynnistyttyä yhteydessä.

### Luokan luomisen syntaksi

LUOKKA LUODAAN Javassa luokat luodaan `class` avainsanalla. Avainsana tarkistaa edestään myös näkyvyysmääreen (käsitellään seuraavassa alakappaleessa). 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 TULOASTAMINEN 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 on käyttää suurimmaksi osaksi "`println`"-metodia, joka ottaa merkkijonon ja tulostaa sen, sekä rivinvaihdon. On huomioitavaa, että `println`, samoin kuin muutkin `System.out`:sta löytyvät tulostusmenetodit ovat todella monipuolisia ja tukevat monenlaisia tapoja tulostaa toivottua tekstiä. Toistaiseksi

```

package week2;

// public-avainsana kertoo luokan olevan koko muun ohjelmiston nähtävissä
//
// class-avainsana aloittaa luokan määritelmän. Sitä seuraa luokan nimi ja hakasulkeisiin suljettu
// luokan runko. Pääluokan nimeä ei ole etukäteen määriteltä 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!");
    }
}

```

Listing 4: Yksinkertaisen pääluokan ja main-funktion luominen  
Javassa

nämä ominaisuudet eivät kuitenkaan ole ajankohtaisia ja opas käyttää "[System.out.println](#)" metodia tulostukseen.

### *Metodin määrittäminen 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 määriteltä tyyppejä.

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 ajattelemaan koodin laatua: hyvä metodi tekee vain yhden asian, joten jos nimeäminen

```

package week2;

public class BasicMethodChild {

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

```

Listing 5: Metodien luominen Javassa

simppeleissä verbimuodossa ei onnistu, olisi ehkä syytä jakaa metodi useampaan pienempään metodiin.

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 notaa tiolla "object.doStuff(322)". Jos metodille on määritelty paluuarvon tyyppi, kutsu palauttaa tämän tyyppin muuttujan, jota voidaan käyttää koodissa.

### *If-lause ja else-lause*

OHJELMOINTI ON pohjimmiltaan logiikkaa. Ohjelman ajon muokkaaminen olosuhteiden 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 ehtolauseetta 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

```

package week2;

public class BasicMethod {

    public static void main(String[] args) {

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

        // Metodia kutsuttaessa on pidettävä huolta, että metodille annetaan oikean tyyppin
        // 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);
    }
}

```

Listing 6: Metodin kutsuminen Javassa

perusteet. Huomioi, että normaalisti ehtolauseiden arvoa ei tiedetä ennen ohjelman ajoa, vaan se muuttuu dynaamisesti.

## *Javan ominaispiirteitä*

### *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:

```

package week2;

public class BasicIfElse {

    // Tästä ei tarvitse välittää. Ainut tämän rivin tarkoitus on estää IDE:tä, jolla esimerkkikoodi
    // on kirjoitettu 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.");
        }
    }
}

```

Listing 7: Esimerkki if- ja else-lauseiden käytöstä

Avainsana	Näkyvyys	Käyttökohteet
<a href="#"><i>private</i></a>	Vain tämä luokka	Luokkien ominaisuudet
Ei avainsanaa (default)	Kaikki luokat tässä paketissa	Luokat ja luokkien ominaisuudet
<a href="#"><i>protected</i></a>	Kaikki luokat tässä paketissa ja kaikki tämän luokan perivät luokat	Luokkien ominaisuudet
<a href="#"><i>public</i></a>	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 luvussa \*x\*, joten sen toiminnallisuuden ymmärtäminen ei ole vielä ajankohtaista.

TÄHÄN MENNESSÄ oppaassa on vältetty näkyvyysmääreiden käyttöä, eli niitä on käytetty vain main-metodien määrittämisen yhteydessä. Tällöin Java-kääntäjä asettaa näkyvyydeksi automaattisesti oletusnäkyvyyden (default), 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 \*x\*.

### *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 säilömiä dataa tarvitaan luokan ulkopuolella. Tämä ei ole suunnitteluvirhe, vaan yksinkertainen ohjelmoinnin totuus. Tätä varten Javaan on vakiintunut kaksi tärkeää metodityyppiä: *noutaja* (getter) ja *asettaja* (setter).

```
package week2;

public class AccessModifierExampleFirstChild {

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

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

Listing 8: Ensimmäinen näkyvyysmääre-esimerkin luokka

```
package week2;

public class AccessModifierExampleSecondChild {

    // Tämä merkkijono ei näy, vaikka luokkaa käytettäisiin muualla koodissa.
    private String privateString = "Private string";

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

Listing 9: Toinen näkyvyysmääre-esimerkin luokka

```

package week2;

public class AccessModifierExampleMain {

    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 näkyvyyttä.
        AccessModifierExampleFirstChild firstChild = new AccessModifierExampleFirstChild();
        AccessModifierExampleSecondChild secondChild = new AccessModifierExampleSecondChild();

        // 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)
        }
        */
    }
}

```

Listing 10: Näkyvyyttä-esityksen pääluokka



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 säilömiseen 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 säilömiseen 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 suosittelman yksityinen muuttuja, noutaja ja asettaja -mallin sijaan.

## *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, kuten moni muu ohjelmointikieli, käsittelee konsolisyytettä ja tiedostosyytettä 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

```

package week2;

public class GetterSetterChild {

    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;
    }
}

```

Listing 11: Ensimmäinen noutaja/asettajaesimerkin luokka

```

package week2;

public class GetterSetterMain {

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

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

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

```

Listing 12: Noutaja/asettajaesimerkin pääluokka

koko syötteen noutoprosessin, 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 rakentamisen `System.in`-argumentilla on helpoin tapa muodostaa käyttäjältä syötettä keräävä olio. Tältä oliolta voi pyytää monia eri parsintametoja, 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 vastaanottaa käyttäjäsyötettä, mutta raaka 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 syötteenä.

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 syöteen.

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 simppeä.

```

package week2;

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

public class BasicUserInput {

    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);
    }
}

```

Listing 13: Käyttäjän syötteen noutaminen Javassa

# *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 simppeilytensä 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 alkioden 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 tietotyyppin 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,

```
package week2;

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

Listing 14: Dataluokka käytettäväksi taulukkoesimerkissä

että alkion noutamisessa.

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

### *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

```

package week2;

public class BasicArray {

    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.
        BasicArrayDataPoint[] dataPoints = new BasicArrayDataPoint[4];

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

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

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

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

        // Luetaan data instansseista. Huomio plus-merkillä toteutettu tekstin yhdistäminen
        // tulostuslauseessa. Javassa on olemassa sivistyneempiä tapoja formatoida tulostetta,
        // mutta niitä käsitellään myöhemmin. Tämän tavan huonot puolet ovat ilmeisiä: se tuottaa
        // rumaa, ja hankalasti luettavaa koodia
        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
        );
    }
}

```

Listing 15: Taulukkojen käyttö Javassa

*Hajautustaulut*

*Toistorakenteet*

*For-looppi*

*While-looppi*

*Foreach-looppi*

*Javan ajamisen teoriaa*

*Tulkkaus vastaan kompilointi*

*Muistinhallinta ja garbage collector*

*Lisää Javan konsepteja*

*Luokkamuuttujat, luokkafunktiot ja static*

*Tyypimuunnokset (casting)*

*Vakiot ja final*

*Luetellut tyypit (enum)*

*Aritmeettiset- ja bittiooperaatiot*



# Sanasto

*abstraktio* (abstraction) Ohjelmoinnin perustekniikka, jossa ongelman tarkka ratkaisu piilotetaan kutsuttavan koodirakenteen, kuten funktion, tietorakenteen tai luokan taakse. [9](#)

*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](#)

*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. [11](#), [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. [9](#), [10](#), see [parametri](#) & [metodi](#)

*instanssi* (instance) Olio, joka on luotu jonkin luokan pohjalta on kyseisen luokan instanssi. [10](#), see [olio](#) & [luokka](#)

*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. [11](#)

*luokka* (class) Ohjelmoijan kirjoittama muotti, jonka pohjalta ohjelmisto luo olioita. Voi sisältää metodeja ja datakenttiä mutta yleensä näiden käyttämiseksi vaaditaan olion luontia. [10](#), *see* [olio](#), [metodi](#) & [instanssi](#)

*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](#)

*metodi* (method) Luokkaan sidottu käskysarja, joka suorittaa ottamiensa parametrien ja luokan omien datakenttien perusteella jonkin tietyn toiminnon. [10](#), *see* [parametri](#), [funktio](#) & [luokka](#)

*muuttuja* (variable) koodissa määritelty tietokenttä, joka sisältää jonkin ohjelman käyttämän arvon. [15](#)

*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" ("get-Variable"). Parantaa enkapsulaatiota ja mahdollistaa esimerkiksi laiskan alustuksen muuttujille. [22](#)

*näkyvyysmääre* (access modifier) Muuttujan näkyvyyden määrittävä avainsana. [20](#), *see* [muuttuja](#)

*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. [10](#), *see* [luokka](#), [metodi](#) & [instanssi](#)

*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 ohjelmiston muokkaus hankaloituu ja täten voidaan katsoa ohjelmiston laadun laskevan. [11](#)

*primitiivinen tietotyyppi* (primitive data type) Tietotyyppi, jonka ohjelmointikieli kykenee säilömään suoraan muistipaikkaan raakana numeerisena datana. Ainoat tietotyypit Javassa, jotka eivät ole jonkin luokan instansseja. [15](#), *see* [luokka](#) & [instanssi](#)

*rajapinta* (interface) Yleisessä käytössä termi ohjelmistossa toteutuvalla sopimuksella, jonka jokin metodi, luokka tmv. toteuttaa tai kahden ohjelmiston osan välinen taso. Javan yhteydessä luokka, joka luodaan interface-avainsanalla ja joka sisältää vain luokan implementoitujen metodien signatuurit. Rajapintaluokka ei yleensä sisällä ollenkaan toiminnallista koodia. [11](#), *see* [interface](#)

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

*signatuuri* (signature) Metodin määritelmä, joka sisältää sekä metodin nimen, että sen ottamien argumenttien tyypit. [17](#)

*staattisesti tyypitetty kieli* (statically typed language) Ohjelmointikieli, joka tietää jokaisen koodissa esiintyvän muuttujan tietotyypin koko ajan. [15](#)

*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 määrittellään 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]". [29](#)

*tietue* (struct) Vanhahko muokattava tietotyyppi, yleinen esimerkiksi C-kielessä. Käyttäjä voi määritellä tietueen sisältämään mitä tahansa vakiokokoisia datakenttiä. Luokkien edeltäjä. [9](#)

# *Javan avainsanat*

*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 tietotyypppejä. Luodaan new-avainsanalla sulkemalla normaalin geneerisen luokan määritelmän tapaan säilöttävä luokka pienempi kuin- ja suurempi kuin -merkkien väliin. Tällöin merkkijonoja säilövä ArrayList-instanssi voidaan luoda seuraavalla koodinpätkällä:

"new ArrayList<String>()". 30, see [List](#), & new

*bool* Primitiivinen tietotyyppi, joka sisältää totuusarvon (true tai false). 15, see [primitiivinen tietotyyppi](#)

*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. 25, 27, see [Scanner](#)

*byte* Primitiivinen tietotyyppi, joka sisältää tavun kokoisen merkillisen kokonaisluvun. 15, see [primitiivinen tietotyyppi](#)

*char* Primitiivinen tietotyyppi, joka sisältää kahden tavun kokoisen unicode-koodatun merkin esitettynä merkittömänä kokonaislukuna. 15, see [primitiivinen tietotyyppi](#)

*class* Avainsana, joka aloittaa luokan määritelmän.. 17

*double* Primitiivinen tietotyyppi, joka sisältää 64 bitin kokoisen liukumaesitetyn desimaaliluvun. 15, 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. [19](#), *see* [if](#)

*float* Primitiivinen tietotyyppi, joka sisältää 32 bitin kokoisen liukumaesitetyn desimaaliluvun. [15](#), *see* [primitiivinen tietotyyppi](#)

*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. [19](#), *see* [else](#)

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

"class PasswordLogin implements Authenticator...". *see* [class](#) & [interface](#)

*InputStreamReader* Javan standardikirjaston luokka tavujonon merkkijonoksi muuttamista varten. Rakentaja ottaa muutettavan streamolion ja vapaaehtoisesti merkkisetin. [25](#), [27](#)

*int* Primitiivinen tietotyyppi, joka sisältää 32 bitin kokoisen merkillisen kokonaisluvun. [15](#), *see* [primitiivinen tietotyyppi](#)

*List* Javan standardikirjaston rajapinta, jonka instanssit ovat järjestettyjä kokoelmia. Kaikki List-rajapinnan toteuttavat luokat tukevat iterointia, indeksipohjaista hakua ja lisäystä ja vertailua. Yleisimmin käytetty List-rajapinnan toteuttava luokka on ArrayList, mutta esimerkiksi myös LinkedList ja ArrayQueue toteuttavat rajapinnan. *see* [ArrayList](#), [interface](#) & [rajapinta](#)

*long* Primitiivinen tietotyyppi, joka sisältää 64 bitin kokoisen merkillisen kokonaisluvun. [15](#), *see* [primitiivinen tietotyyppi](#)

*main* Varattu metodinimi metodille, jonka Java ajaa ensimmäisenä ajaessaan ohjelmistoa. Ohjelmistossa voi olla useampi-main niminen metodi, mutta vain määritellyn juuriluokan main-metodi ajetaan. Metodin täytyy olla muotoa public static void ja ottaa yksi taulukko String-luokan instansseja. [17](#), [29](#), *see* [metodi](#) & [taulukko](#)

*new* Avainsana, joka luo uuden instanssin luokasta. Avainsanaa seuraa aina toivotun luokan nimi, jonka perässä on sulkuihin sul-

jettuna toivotut rakentajalle annettavat parametrit, metodikutsun tapaan. [29](#), *see* [rakentaja](#)

*private* Näkyvyysmääre, joka määrittää ominaisuuden olevan käytettävissä vain ominaisuuden omistaman luokan sisällä. [22](#)

*protected* Näkyvyysmääre, joka määrittää ominaisuuden olevan käytettävissä ominaisuuden omistavan luokan sisältävässä *package*ssa ja kaikissa ominaisuuden omistavan luokan perivissä luokissa. [22](#)

*public* Näkyvyysmääre, joka määrittää ominaisuuden olevan käytettävissä kaikkialla ohjelmistossa. [17](#), [22](#)

*Scanner* Javan standardikirjaston luokka halutun lähteen lukemiseen ja parsimiseen. Rakentaja ottaa parsittavan *stream*-olion. Sisältää metodeja eri tietotyyppien parsimiseen annetusta *stream*-oliosta. [25](#), [27](#)

*short* Primitiivinen tietotyyppi, joka sisältää kahden tavun kokoisen merkillisen kokonaisluvun. [15](#), *see* [primitiivinen tietotyyppi](#)

*static* Staattinen metodi tai muuttuja näkyy kaikille sen omistavan luokan instansseille jaetusti. Tunnetaan luokkamuuttujana tai luokkametodina. [17](#)

*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. [15](#), [17](#)

*System* Javan standardikirjaston luokka, joka sisältää luokkametodeja ja luokkamuuttuja systeimirajapintojen, kuten ympäristömuuttujien, tulosteen ja syötteen käyttöön. [17](#), [18](#), [27](#), *see* [luokkamuuttuja](#) & [luokkametodi](#)

*void* Muuttujan paluuarvotyyppi muuttujalle, joka ei palauta mitään. [17](#)