Ville-Pekka Honkanen

OHJELMISTOKEHITTÄJÄN KEINOJA JAVA-KOODIN TEHOSTAMISEEN JA KEINOJEN HYÖDYLLISYYS

Tietojärjestelmätieteen
kandidaatintutkielma
11.05.2010

Jyväskylän Yliopisto
Tietojenkäsittelytieteiden laitos
Jyväskylä

TIIVISTELMÄ

Honkanen, Juho Ville-Pekka
Ohjelmistokehittäjän keinoja Java-koodin tehostamiseen ja keinojen hyödyllisyys / Ville-Pekka Honkanen
Jyväskylä: Jyväskylän yliopisto, 2010.
37 s.
Kandidaatintutkielma

Tässä tutkielmassa käydään lävitse keinoja, joilla voidaan parantaa Java-koodin suorituskykyä. Tutkielma on toteutettu kirjallisuuskatsauksena ja sen tarkoituksena on tutkia perinteisiä ”koodioptimointeja” ja selvitetään onko niistä enää nykypäivänä hyötyä. Lisäksi käsitellään keinoja, joilla todella voi parantaa ohjelman suorituskykyä ja missä tilanteissa näitä voidaan käyttää. Läpi käydään myös koodaamistapoja, joita tulisi välttää, esimerkiksi lopetusmetodien käyttö.

Tutkielmassa selviää, että useat aiemmin järkevinä pidetyt optimoinnit ovat nykyisin useissa tilanteissa suhteellisen turhia ja niitä tulisi jopa välttää. Tällaisina voi pitää ainakin `final`-määreen käyttöä luokkiin tai metodeihin liitettynä.

Toisaalta jotkin aiemminkin hyvin toimineet koodin tehostuskeinot ovat nykyisinkin toimivia. Tällaisia ovat erityisesti merkkijonojen käsittelyyn sopiva `StringBuilder`-luokka ja olioiden kierrättäminen. Toimivissakin tehostuskeinoissa on huomioitava, että ne sopivat vain tiettyihin tapauksiin.

Lopputuloksena koko koodintehostamisesta on se, että koodia ei tulisi tehdä vain suorituskyvyn vuoksi, vaan tulisi pyrkiä tekemään hyvää koodia ja jättää optimoiminen JVM:lle. Tällöin usein myös suorituskyky seuraa mukana. Jos kuitenkaan suorituskyky ei riitä, voidaan miettiä, mitä tulisi koodissa optimoida.

AVAINSANAT: java, optimointi, koodin tehostaminen, interning, object pooling

Tutkielman ohjaaja:

Pertti Hirvonen

Jyväskylän Yliopisto, Tietojenkäsittelytieteiden laitos

Tutkielman tarkastaja:

Jorma Kyppö

Jyväskylän Yliopisto, Tietojenkäsittelytieteiden laitos

SISÄLLYS

1 JOHDANTO

Ohjelmiston suorituskyky on aina yksi ohjelmiston toivotuista laadullisista vaatimuksista, ainakin jossakin muodossa. Tämän takia ohjelmistokehittäjän on loogista pyrkiä etsimään keinoja nopeuttaa tuottamiaan ohjelmia. Tietyissä tapauksissa huonosti optimoitu koodi saattaa jopa halvaannuttaa ohjelman käytettävyyden.

Java on esimerkki varsin abstraktista ja korkealla tasolla olevasta ohjelmointikielestä, jota on perinteisesti syytetty huonosta suorituskyvystä. Nykyisin nopeusero esimerkiksi C-kieleen on pienentynyt, mutta silti ohjelmoija saattaa epähuomiossaan luoda ohjelmakoodia, jonka suorituskyky on alhainen, koska hän ei ole ottanut huomioon Javan erityispiirteitä. Tilanteesta riippuen tämä suorituskyvyn alentuma saattaa olla käytännössä täysin merkityksetön tai toisaalta jopa kriittinen ohjelmiston toiminnalle.

Tässä tutkielmassa paneudutaan siihen, kuinka ohjelmistokehittäjä pystyy parantamaan ohjelmistonsa suorituskykyä ja missä tilanteissa erilaisia tapoja voidaan käyttää. Niinpä tätä tutkielmaa voi käyttää ohjeena, kuinka mahdollisissa ongelmatilanteissa tietyt suorituskykyongelmat voidaan kiertää. Toisaalta tutkielman voi myös nähdä ohjelmoijalle yleishyödyllisenä katsauksena Javan sisäiseen toimintaan.

Tutkielmassa tullaan käsittelemään ohjelmakoodiin tehtäviä muutoksia ja huomioitavia seikkoja, joilla voidaan vaikuttaa ohjelman suorituskykyyn. Vastaavasti käsitellään myös seikkoja, joita ohjelmakoodissa tulisi välttää. Osa esitellyistä keinoista ovat osin hyödyttömiä tai jopa haitallisia, mutta ne on otettu mukaan tutkielmaan, koska ne esitellään usein vanhemmassa perus-

kirjallisuudessa hyvinä keinoina. Yksi tämän tutkielman tarkoituksista onkin hälventää koodin tehostamiseen liittyviä myyttejä.

Tutkielmassa käsitellään erityisesti Javaan sopivia keinoja, joten mitään yleispäteviä algoritmien kehittämisideoita ei tässä tutkielmassa tulla käymään läpi. Lisäksi on huomattava, että tutkielma on tarkoitettu lähinnä ohjelmistokehittäjille, joten keinoja, joita loppukäyttäjä voi hyödyntää, ei käydä lävitse. Yksi tällainen keino olisi esimerkiksi Javan virtuaalikoneen asetusten muuttaminen.

Tutkielmassa tullaan esittelemään useita koodiesimerkkejä, joista osa on selkeästi virheellisiä hyvän ohjelmointitavan tai suorituskykynsä puolesta. Nämä koodiesimerkit on erikseen merkitty ”virheellinen esimerkki”-tekstillä.

Luvussa 2 tullaan käsittelemään tutkielmassa käytettäviä pääkäsitteitä lyhyesti. Tämän jälkeen varsinaiset tekstiluvut ovat luvuissa 3-5. Luvussa 3 käsitellään olioiden luomiseen ja tuhoamiseen liittyviä seikkoja. Luvussa 4 tutustutaan merkkijonoihin liittyviin suorituskyvyllisiin seikkoihin ja lopulta luvussa 5 tutustutaan sekalaisiin keinoihin tehostaa Java-koodia.

2 TUTKIELMAAN LIITTYVIÄ PÄÄKÄSITTEITÄ

Tässä luvussa käydään lävitse käsitteitä, joita tutkielmassa myöhemmin tullaan käyttämään, mutta joita ei ole ollut mielekästä selittää samassa kohdassa varsinaisen tekstin kanssa. Helpommin selitettävät ja enemmän varsinaiseen lukuunsa liittyvät käsitteet löytyvät tutkielmasta kohdista, joissa niitä käytetään.

Koska tässä tutkielmassa käsitellään tapoja, joilla voidaan parantaa suorituskykyä, on paikallaan käydä lävitse, mitä ohjelman suorituskyvyllä tarkoitetaan. Tässä tutkielmassa tullaan suorituskyvyllä tarkoittamaan kolmea asiaa. Ensimmäiseksi, suorituskykynä nähdään se, kuinka nopeasti ohjelmisto suorittaa tietyn asian ja mikä on ohjelman vasteaika käyttäjän toimille. Toiseksi suorituskyvyllä tarkoitetaan ohjelmiston käynnistymisnopeutta. Ja kolmanneksi sillä tarkoitetaan sitä, miten paljon ohjelmisto käyttää tietokoneen keskusmuistia. (Wilson & Kesselman 2000)

Inline-optimointi tarkoittaa esimerkiksi Javan virtuaalikoneen tapaa optimoida Java-koodia. Siinä käännettäessä ohjelmaa kääntäjä sijoittaa kutsuttavan metodin koodin kutsuvaan metodiin. Näin säästytään metodikutsulta, jotka vievät aina prosessoriaikaa. Esimerkissä 1 nähdään alkuperäinen koodi ja sen jälkeen esitetään Esimerkissä 2 inline-optimoitu koodi.

Esimerkki 1: inline-optimoitava ohjelmakoodi (Stoodley, Ma & Lut 2007)

```
int foo() {
     int x=2, y=3;
     return bar(x,y);
}
final int bar(int a, int b) {
     return a+b;
}
```

Kun virtuaalikone optimoi Esimerkissä 1 esitetyn koodin, se huomaa kutsuttavan `bar()`-metodin olevan määritetty `final`-määreellä, joka tarkoittaa, että sen on oltava `foo()`:sta kutsuttu metodi. Tämä johtuu siitä, että `foo()`-metodia ei ole voitu ylikirjoittaa missään aliluokassa. Tällöin JVM muuntaa koodin optimoidumpaan muotoon joka on nähtävissä Esimerkissä 2. Näin toimittaessa jätetään välistä kokonaan yksi metodikutsu ja ohjelma nopeutuu. (Stoodley, Ma & Lut 2007)

Esimerkki 2: inline-optimoitu ohjelmakoodi (Stoodley, Ma & Lut 2007)

```
int foo() {
     int x=2, y=3;
     return x+y;
}
```

Tutkielmassa sanalla virtuaalikone tai lyhenteellä JVM viitataan Sun Microsystemsin/Oraclen valmistamaan Java Virtual Machineen (JVM) nimeltä Hotspot, jos tekstissä ei muuta mainita. On olemassa muidenkin valmistajien

virtuaalikoneita Javalle, muun muassa IBM J9 ja Kaffe, mutta tässä tutkielmassa käsitellään vain Hotspot-virtuaalikonetta.

JVM on ohjelmisto, joka tulkitsee tai kääntää Javan tavukoodia ja siten suorittaa sitä. Jokaiselle alustalle, jolla koodia on tarkoitus ajaa, on oma virtuaalikoneensa. Tämä mahdollistaa sen, että samaa tavukoodia on ainakin teoriassa mahdollista ajaa useilla eri alustoilla ilman muutoksia koodiin. (Vesterholm & Kyppö 2008)

Nykyaikaiset virtuaalikoneet kykenevät tekemään useita erilaisia optimointeja joilla ne voivat nopeuttaa koodia. Tällainen on esimerkiksi ajonaikainen kääntäminen (Just-in-Time compilation), jolloin tavukoodia ei tulkata, vaan käännetään natiiviksi konekieleksi samaan aikaan kun sitä suoritetaan. Hotspot virtuaalikone osaa myös etsiä koodista niin sanotusti ”kuumia kohtia” (engl. hotspots), joita suoritetaan usein tai toistuvasti, ja keskittää optimoinnit juuri näihin (Sun Microsystems 2006). Lisäksi virtuaalikoneilla on monia muita tapoja parantaa tehokkuuttaan, mutta niiden käsittely ei kuulu tämän tutkielman sisältöön.

3 OLIOIDEN LUOMISEEN JA TUHOAMISEEN LIITTYVÄT TEHOKKUUSSEIKAT

Perinteisesti Javan olioiden luomista ja tuhoamista on pidetty hitaana toimenpiteenä, koska ne ovat sisäisesti suhteellisen vaativia operaatioita. Nykyisin ero muihin ohjelmointikieliin on pienentynyt ja olioiden luomista tai roskienkeruuta ei voida enää pitää niin kriittisenä tai hitaana toimenpiteenä kuin aiemmin (Krill 2008).

Varsinkin usein toistettuna olioiden luominen on silti nykypäivänäkin ajallisesti varsin raskas operaatio. Sen lisäksi oliot kuluttavat dynaamista muistia. Olioiden luominen on todella monimutkainen operaatio, mutta rajusti yksinkertaistettuna olion luomiseen kuuluvat seuraavat vaiheet: muistitilan varaus, sekä olion luokkaviitteen että metadatan alustus ja lopulta olion luokassa määritellyt alustustoimet (Sakkinen 2007). Vastaavasti oliota tuhottaessa samat toimet tehdään toisinpäin. Näin ollen voidaan todeta, että tilanteissa, joissa olioita luodaan esimerkiksi silmukassa, voi olioiden luominen haitata ohjelman suorituskykyä. Tällaisessä tilanteessa voidaan esimerkiksi miettiä, olisiko mahdollista käyttää primitiivityyppejä olioiden sijaan (Bloch 2008).

Tämän luvun keinoissa tullaan käsittelemään pääosin sitä, kuinka välttää turhien olioiden luominen. Lisäksi käsitellään muita olioiden luontiin ja tuhoamiseen liittyviä seikkoja. Aliluvussa 1 käydään lävitse olioiden kierrättämistä, aliluvussa 2 käsitellään olioiden varastointia ja lopulta aliluvussa 3 tutustutaan olioiden lopetusmetodien ongelmiin.

3.1 Olioiden luominen ja kierrättäminen

Kuten aiemmin on todettu, on uusien olioiden luominen suorituskyvyn kannalta varsin kallis operaatio ja täten sitä tulisi mahdollisuuksien mukaan välttää. Tietenkään aina tämä ei ole mahdollista, mutta kokonaan uuden olion luomisen sijaan voidaan usein olio niin sanotusti kierrättää. (Shirazi 2003)

Olion kierrättämisellä tarkoitetaan sitä, että olio palautetaan alkutilaansa, jolloin olio periaatteessa vastaa uutta. Usein tällainen operaatio on nopeampi kuin kokonaan uuden olion luominen. (Klemm 1999) Esimerkiksi luokkaan voidaan luoda metodi, jolla olion tiedot pyyhitään tai oliota voidaan käyttää sillä tavoin, että se tulee aina ennen jokaista käyttökertaa tyhjennetyksi. Esimerkissä 3 esitellään kyseisestä ongelmaa ja sen jälkeen Esimerkissä 4 sen korjattua vastinetta. Viallinen kohta on koodiesimerkissä korostettu alleviivauksella.

Esimerkki 3: ei olion kierrättämistä (sovellettu Klemm 1999)

```
class Lotto {
     private Random randGen = new Random();
     public int[] arvoLottoNumerot() {
          int[] lottoNumerot = new int[7];
          for (int i=0; i<7; i++) {
               lottoNumerot[i]= randGen.nextInt(39)+1;
          }
          return lottoNumerot;
     }
}
```

Esimerkissä 3 on metodi, jolla arvotaan lottonumeroita. Joka kerta, kun metodi suoritetaan, luodaan taulukko. Ajatellaan esimerkiksi, että on ohjelma, joka luo lottorivejä ja laskee niistä todennäköisyyksiä. Jos ohjelma luo esimerkiksi miljoona lottoriviä, luodaan samalla miljoona taulukkoa. Tämä on selvästi turhaa vaivaa, kun voitaisiin luoda vain yksi taulukko, jota kierrättää. Esimerkissä 4 on korjattu esimerkki Esimerkistä 3. Varsinainen korjauskohta on koodiesimerkissä korostettu alleviivauksella.

Esimerkki 4: olion kierrättäminen (sovellettu Klemm 1999)

```
class Lotto {

     private Random randGen = new Random();
     private int[] lottoNumerot = new int[7];
     public int[] arvoLottoNumerot() {
          for (int i=0; i<7; i++) {
               lottoNumerot[i]=randGen.nextInt(39)+1;
          }
          return lottoNumerot;
     }
}
```

Korjatussa luokassa luodaan vain kerran `lottoNumerot`-taulukko ja kierrätetään sitä uudestaan ja uudestaan. Tässä tapauksessa oliota ei tarvitse millään tavalla tyhjentää välissä, koska uudet arvot automaattisesti korvaavat vanhat arvot.

Kierrätyksellä voidaan myös tarkoittaa sitä, että on turhaa luoda uutta oliota, jos voidaan käyttää vanhaa ja tiedetään, ettei sitä tulla muuttamaan. Tällöin sen

sijaan, että olio luotaisiin aina uudestaan, voitaisiin se määritellä vakioksi. (Bloch 2008)

Olioiden kierrätystä voidaan pitää varsin yksinkertaisena seikkana, jonka pitäisi olla jokaisen enemmän ohjelmoineen hallussa. Periaatteessa olioiden kierrätyksessä ei ole haittapuolia, jos koodia ei aleta väkisin pakottamaan siihen esimerkiksi muuttamalla arvokeskeisiä (engl. immutable) luokkia muuttujakeskeisiksi (engl. mutable). Idealtaan olioiden kierrätys liittyy hieman myös olioiden varastointiin, jota käsitellään aliluvussa 2.

3.2 Olioiden varastointi (Object pooling)

Olioiden varastoinnilla tarkoitetaan sitä, että olioita luodaan etukäteen ja pidetään tallessa sekä käytön aikana tallennetaan talteen, jolloin niitä voidaan käyttää uudestaan ja uudestaan. Tällöin säästytään toistuvalta olioiden luonnilta ja tuhoamiselta. Esimerkiksi olioiden varastoinnissa voidaan tehdä usein käytettävät oliot jo ennalta valmiiksi. (Kircher & Jain 2002) Koska olioiden luonti ja tuhoaminen ovat raskaita tapahtumia, tällöin luonnollisestikin säästetään resursseja. Oliovaraston voi siis nähdä eräänlaisena puskurina (engl. buffer) tai välimuistina (engl. cache).

Javan tapauksessa olioiden luonti ja roskienkeruu olivat aikoinaan raskaampia operaatioita, jolloin myös olioiden varastointi oli kätevä tapa välttää näitä ja parantaa suorituskykyä. Nykyisin kuitenkin olioiden varastointi sopii harvoihin tapauksiin, koska JVM on ajan myötä kehittynyt ja olioiden varastoinnin toteuttaminen on aina kohtuullisen hankalaa (Goetz 2004) .

Hankaluudet olioiden varastoinnissa johtuvat useista asioista. Esimerkiksi, jos

käytetään säikeitä ja oliovarasto on jaettu säikeiden kesken, saattaa tämä osoittautua synkronisoinnin pullonkaulaksi. Olioiden varastointi myös pakottaa tuhoamaan käytetyt ja turhat oliot eksplisiittisesti, jolloin ongelmaksi saattaa tulla tarpeellisten viitteiden tuhoaminen, tarpeettomien jääminen paikoilleen ja viitteiden meneminen sekaisin. Lisäksi oliovaraston koko pitää olla lähes täydellinen, muutoin suorituskyky kärsii. (Goetz 2004) Tietenkin on myös huomattava, että oliovaraston pelkkä toteuttaminen vaatii tarkkaa suunnittelua ja ohjelmointia, joihin kulutettu aika on luonnollisesti pois muusta työajasta.

Lopuksi olioiden varastoinnista on sanottava, että Clickin (2003) mukaan nykyisin olioiden luonti ja tuhoaminen eivät ole niin kalliita operaatioita, että oliovarastojen käyttö olisi käytännöllistä useimmissa tapauksissa. Hänen testeissään todetaan, että olioiden kierrätyksestä tällä tavoin on hyötyä vain, jos käytettävät oliot ovat todella raskaita luoda. Koska Click jo vuonna 2003 tuli tähän lopputulokseen, voidaan olettaa kehityksen jatkuneen samaan suuntaan. Joten olioiden varastoinnin voi katsoa olevan perusteltua vain, jos käsitellään raskaita olioita.

3.3 Olioiden lopetusmetodit

Olion lopetusmetodilla (engl. finalizer) eli purkajalla tarkoitetaan sen `finalize()`-metodia, joka suoritetaan automaattisesti, kun viitteet olioon ovat hävinneet ja oliota oltaisiin tuhoamassa. Lopetusmetodia käytetään muun muassa vapauttamaan resursseja (esimerkiksi tiedostot), joita ei enää tarvita (Wikla 2003). Lopetusmetodin käytön ongelmallisuus on siinä, että sen suoritusaika on ohjelmoijan kannalta epädeterministinen, ja ohjelman suorituksen päättyessä olioita voi jäädä kokonaan purkamatta (Sakkinen 2007).

Goetzin (2004) mukaan lopetusmetodien käyttöä tulisi välttää, koska `finalize()`-metodi hidastaa olion luomista sekä tuhoamista. Tämä johtuu siitä, että luonnin aikana `finalize()`-metodilla varustetut oliot joudutaan rekisteröimään roskienkerääjälle ja niiden luominen tapahtuu erilaisin ja hitaammin toimenpitein kuin tavallisten olioiden luominen.

Vastaavasti lopetusmetodilla varustettujen olioiden tuhoaminen on myös hitaampaa, koska niiden tuhoaminen vaatii vähintään kaksi roskienkeruusykliä, ja lisäksi roskienkerääjä joutuu vielä erikseen kutsumaan kyseisen olion `finalize()`-metodia. Tästä johtuen tällaisten olioiden tuhoaminen on hitaampaa, koska roskienkerääjä joutuu käyttämään enemmän muistia näiden olioiden pitämiseen muistissa, sen sijaan, että kyseiset oliot tuhoutuisivat nopeasti ja poistuisivat viemästä muistitilaa. (Goetz 2004)

Samoihin tuloksiin on päätynyt myös Bloch, joka lisäksi toteaa omissa testeissään `finalize()`-metodilla varustetun olion luomisen ja tuhoamisen olevan jopa useita satoja kertoja hitaampaa kuin normaalin olion (Bloch 2008, 28). Hän suositteleekin lopetusmetodien sijaan käyttämään tavallisia metodeja, joilla voidaan tuhota tai vapauttaa olion sisältämät resurssit. Tämän jälkeen tulisi vain kehottaa käyttäjiä käyttämään tätä metodia.

Koska lopetusmetodit ovat hitaita ja epävarmoja käyttää, suosittelevat Goetz (2004) ja Bloch (2008) välttämään niitä kokonaan tai ainakin käyttämään niitä säästeliäästi, jos niiden käyttö on pakollista. Onkin mielenkiintoista, miksi kyseisestä ominaisuudesta ylipäätään kerrotaan kritiikittä useimmissa Javan peruskirjoissa. Varsinkin, kun sen käyttö rajoittuu erikoistapauksiin ja kyseinen ominaisuus on nähtävissä yleisimmissä tilanteissa epädeterministisyytensä takia epäilyttävänä ja suorituskyvynsä puolesta jopa haitallisena.

3.4 Olioiden laiska alustaminen

Laiska alustus (engl. lazy initialization) tarkoittaa sitä, että alustettavaa oliota ei alusteta välittömästi, vaan tämä tehdään vasta silloin, kun olion arvoa tarvitaan (Bloch 2008). Loogisesti ajateltuna tämä on loistava idea, koska mahdollisesti ohjelmistossa saattaa olla muuttujia, joita ei koskaan tulla käyttämään ja joiden alustus varmuuden vuoksi saattaisi olla raskas ja turha operaatio.

Esimerkissä 5 esitetään malli tavallisesta alustamisesta. Tätä esimerkkiä seuraa Esimerkki 6, jossa kuvataan, kuinka sama tehtäisiin laiskasti alustamalla. Esimerkeissä on `Frame`-luokka, joka käyttää `MessageBox`-oliota virheiden näyttämiseen. Erityisesti huomioitavat kohdat on korostettu alleviivauksella.

Esimerkki 5: tavallinen alustaminen (Bishop & Warren 1999)

```
public class MyFrame extends Frame
{
     private MessageBox mb_ = new MessageBox();
     private void showMessage (String message) {
          // Asetetaan näytettävä viesti
          mb_.setMessage( message );
          mb_.pack();
          // Näytetään MessageBox
          mb_.show();
     }
}
```

Huomioitavaa Esimerkissä 5 on, että kun `MyFrame`-luokka luodaan, alustetaan myös muuttuja `mb_` samalla. Esimerkissä 6 näytetään, kuinka sama asia tehdään laiskalla alustamisella. Tällöin muuttuja `mb_` alustetaan vain, jos sitä tarvitaan.

Esimerkki 6: laiska alustaminen (Bishop & Warren 1999)

```
public final class MyFrame extends Frame
{
     private MessageBox mb_ ; // Ei vielä alustettu

     private void showMessage (String message) {
          if(mb_ == null) //Ensimmäinen kutsu metodiin
               mb_=new MessageBox(); // Laiska alustus

          // Asetetaan näytettävä viesti
          mb_.setMessage( message );
          mb_.pack();
          // Näytetään MessageBox
          mb_.show();
     }
}
```

Kuitenkin laiska alustaminen on vaarallista etenkin säikeitä käytettäessä, jolloin pitää huolehtia muuttujan synkronoinnista eri säikeiden kesken. Lisäksi laiskan alustamisen ongelmana on se, että vaikka se vähentää luokan tai muuttujan alustusaikaa, se hidastaa sen jälkeistä käyttöä. (Bloch 2008) Tämä on loogista, koska ensimmäisen käyttökerran jälkeen aina metodia käytettäessä on tarkistus, onko olion viite `null`. Lisäksi laiska alustaminen synkronoinnin kanssa saa yksinkertaisenkin alustustoimenpiteen näyttämään monimutkaiselta ja epäselvältä.

Blochin (2008) kanta onkin, että laiskaa alustamista ei oletusarvoisesti tulisi käyttää. Mutta toisaalta hänen mielestään sillä pystytään parantamaan suorituskykyä tilanteissa, joissa olioiden toistuvista alustamisista ei kätevästi päästä eroon muulla tavoin. Samaten hän toteaa laiskan alustamisen sopivan tilan-

teisiin, joissa käsitellään raskaita olioita, joita ei välttämättä koskaan oikeasti käytetä, jolloin niiden alustaminenkin on turhaa.

4 MERKKIJONOIHIN LIITTYVÄT TEHOKKUUSSEIKAT

Tässä luvussa käsitellään Javan merkkijonoihin ja niiden suorituskykyyn liittyviä keinoja. Ensimmäisessä aliluvussa tullaan käsittelemään yleisesti Javan merkkijonojen luonnetta ja perusasioita niiden suorituskyvyssä. Aliluvussa 2 käsitellään Javan muuttujakeskeisiä ja arvokeskeisiä merkkijonoja eli `String`- ja `StringBuilder`-luokkia. Lopulta viimeisessä aliluvussa käsitellään merkkijonojen varastointia (engl. String interning).

4.1 Yleistä merkkijonoista

Javan merkkijonot eli `Stringit`, ovat periaatteessa eräänlaisia kääreolioita, jotka sisältävät `char`-tyyppisen taulukon ja metodeita, joilla käsitellä tätä. Käytännössä Java on `Stringien` osalta varsin optimoitu, mutta välillä saattaa olla tehokkaampaa toteuttaa jokin merkkijonoihin liittyvä tehtävä puhtaasti `char`-taulukkona (Shirazi 2003). Jotkin `String`-operaatiot myös sisältävät sisäisesti paljon uusien olioiden luomista, jotka saattavat olla hitaita, mutta eivät näy ohjelmoijalle mitenkään (Kluge 1997).

Niinpä esimerkiksi jokin erikoisempi algoritmi, joka vaatii paljon merkkijonon käsittelyä saattaisi olla tehokkaampi toteuttaa `char`-taulukolla kuin `Stringinä`. Esimerkkinä voisi olla pitkä merkkijono, jota käydään lävitse indeksi kerrallaan ja etsitään tiettyjä merkkejä. Tällöin saattaisi olla tehokkaampaa käyttää taulukon läpikäymiseen soveltuvia perustoimenpiteitä `String`-luokan `charAt()`-metodin sijaan.

On kuitenkin huomattava, että tällaiset käyttökohteet ovat usein vain poikkeustapauksia, joita harvemmin tulee vastaan. On luonnollisesti turhaa ja mahdolli-

sesti koodin selkeydelle sekä suorituskyvylle vaarallista, jos jokainen `String`-olio muutetaan `char`-taulukoksi perustoimenpiteitä varten.

4.2 Javan muuttujakeskeiset ja arvokeskeiset merkkijonot

Koska `String`-luokka on tyypiltään arvokeskeinen, eli sen metodit eivät muuta olion tilaa, monen `Stringin` liittäminen on kohtuullisen hidasta. Tämä johtuu siitä, että joka kerta kun `String`-olio liitetään yhteen toisen kanssa, luodaan uusi `String`-olio, johon nämä edelliset `Stringit` on yhdistetty. Tämä muodostuu varsinaiseksi ongelmaksi vasta silloin, kun suoritetaan useita merkkijonojen liittämisiä nopealla tahdilla eli esimerkiksi silmukassa. (Vesterholm & Kyppö 2008) Niinpä tästä ei tarvitse välittää suoritettaessa yksittäisten `Stringien` liittämisiä.

Ongelman voi kiertää `StringBuilder`-luokalla, joka on `String`-luokan muuttujakeskeinen vastine. Muuttujakeskeisyys tarkoittaa, että metodit voivat muuttaa olion sisäistä tilaa (Vesterholm & Kyppö 2008). `StringBuilder`, kuten muutkaan muuttujakeskeiset luokat, ei kuitenkaan ole yhtä turvallinen käyttää kuin vastaava arvokeskeinen luokka (Bloch 2008, 73). Kuitenkin suorituskykykriittisissä sovelluksissa sen käyttöä voitaneen pitää hyväksyttävänä.

Esimerkeissä 7 ja 8 esitellään aiheesta esimerkki, jossa kuvataan kuinka `String`-merkkijonojen liittäminen yhteen voidaan korvata `StringBuilderilla`. Ensimmäisessä esimerkissä, eli Esimerkissä 7, kuvataan kuinka `Stringin` lisäys suoritettaisiin normaalilla tavalla liittämällä kaksi `Stringiä` yhteen.

Esimerkki 7: `String`-merkkijonojen liittäminen (Bloch 2008, 227)

Virheellinen esimerkki!

```
public String statement() {
     String result = "";
     for (int i = 0; i < numItems(); i++)
          result += lineForItem(i); // Liittäminen
     return result;
}
```

Esimerkissä 7 käydään silmukassa jotakin indeksoitua tietorakennetta lävitse indeksistä nolla `numItems()`-metodin palauttamaan arvoon asti. Samalla joka kierroksella liitetään `result`-merkkijonoon tietorakenteesta noudettu muuttuja, joka tässä tapauksessa on luultavasti toinen `String`. Lopulta jokainen tietorakenteen alkio on liitetty `result`-muuttujaan.

Esimerkissä 8 kuvataan kuinka, Esimerkin 7 merkkijonojen liittäminen voidaan korvata `StringBuilderin append()`-metodilla. Tämä on huomattavasti tehokkaampi tapa toteuttaa kyseinen algoritmi, jos läpikäytäviä ja liitettäviä merkkijonoja on huomattavia määriä.

Esimerkki 8: merkkijonojen liittäminen `StringBuilderilla`

```
public String statement() {
     StringBuilder b = new StringBuilder(numItems() *
               LINE_WIDTH);
     for (int i = 0; i < numItems(); i++)
          b.append(lineForItem(i));
     return b.toString();
}
```

Esimerkissä 8 luodaan `StringBuilder`-olio nimeltä `b`, jonka pituus määritetään rakentajassa etukäteen. `LINE_WIDTH` kuvaa tietorakenteen sisältämien merkkijonojen maksimipituutta ja kun se kerrotaan `numItems()`-metodin palauttamalla arvolla, saadaan palautettavan merkkijonon maksimipituus. Tämän jälkeen tietorakenne käydään lävitse kuten aiemmassa esimerkissä, mutta merkkijonojen liittämiseen käytetään `StringBuilder`-luokan `append()`-metodia.

Kuten Esimerkissä 8 tehtiin, on `StringBuilderiä` luotaessa hyvä antaa sen kapasiteetin pituus rakentajassa. Lisäksi tämä kapasiteetti on syytä asettaa sopivaksi, jotta `StringBuilderin` kapasiteetin kokoa ei jouduta muuttamaan dynaamisesti, koska se heikentää suorituskykyä (Holmes 2010). Oletusrakentajalla luodun `StringBuilder`-olion kapasiteetti on vain 16 merkkiä. (JavaAPI 2009) Lisäksi `StringBuilderia` käytettäessä tulisi pysyä `StringBuilder`-olion käyttämisessä niin kauan, kunnes `String`-oliota oikeasti tarvitaan. Eli turhaa merkkijonon vaihtelua `StringBuilderista Stringiksi` tai toisinpäin tulisi välttää. (Holmes 2010)

Tämän merkkijonoihin liittyvän tehostuskeinon tulisi olla jokaisen Java-ohjelmoijan työkalupakissa, koska se on selkeästi tietyissä tilanteissa tehokas tapa parantaa ohjelmiston suorituskykyä. Täytyy kuitenkin muistaa, että tavallisten `Stringien` käyttö on kuitenkin turvallisempaa ja niiden käytön pitäisi olla vakiotoimintatapana useimmissa normaaleissa tilanteissa.

4.3 Merkkijonojen varastointi String-luokan intern()-metodilla

`Stringeistä` on myös huomattava, että Javassa voidaan hyödyntää niin sanottua merkkijonojen varastointia. Tämä tarkoittaa sitä, että `String`-olion

`intern()`-metodia kutsuttaessa kyseinen `String`-olio sijoitetaan `String`-luokan ylläpitämään merkkijonovarastoon (engl. String pool). Samalla `intern()`-metodi palauttaa viittauksen tähän olioon. Jokainen merkkijono on varastossa vain kertaalleen, ja kutsuttaessa merkkijonoa, joka on jo varastossa, saadaan sen jaettu viite. (van der Linden 2004)

Kaikki `String`-literaalit ja merkkijonovakiot lisätään automaattisesti merkkijonovarastoon, jolloin varmistuu, ettei kahta samanlaista `String`-literaalia ole olemassa kerrallaan (Gosling & ym. 2005, 28-29). Oman `Stringin` lisääminen varastoon `intern()`-käskyllä vie hieman suoritusaikaa, joten sen käyttö on suositeltavaa vain, jos ohjelmassa on merkkijonojen vertailuja paljon (van der Linden 2004).

Varsinainen hyöty tällaisesta varastoinnista on se, että `Stringejä`, jotka `intern()`-metodi palauttaa, voidaan vertailla yksinkertaisella ”==” -vertailulla. Tämähän ei normaalissa tilanteessa toimi merkkijonojen vertailussa, koska kyseinen operaatio vertailee vain `String`-olioiden viitteitä, jotka luonnollisesti ovat erilaiset, vaikka merkkijonojen sisältö olisi sama. Tämä johtuu siitä, että vaikka merkkijonot ovat samat, ne ovat kahden eri olion sisällä.

Normaalisti merkkijonojen vertailuun käytettäisiin `String`–luokan `equals()`-metodia. Kuitenkin, jos käytämme `intern()`-metodia hakeaksemme merkkijonoillemme viitteet merkkijonovarastosta, voidaan näitä merkkijonoja vertailla ”==”-operaattorilla, koska jokainen merkkijono on varastossa vain kertaalleen. Jos molempien `Stringien intern()`-metodi palautti saman viitteen, myös ”==”-operaattori paljastaa merkkijonojen olevan samat. (van der Linden 2004)

Tällainen tarkastus ”==”-vertailulla on paljon nopeampaa kuin `equals()`-

metodin käyttö. Samalla saatetaan säästää hieman muistia, sillä olioiden määrä on pienempi, koska tietty merkkijono on vain yhdessä `String`-oliossa. Kuitenkin `intern()`–metodikutsun hitauden takia tällainen vertailu sopii vain tilanteisiin, joissa tehdään paljon vertailuja merkkijonojen kesken ja kuitenkin luodaan niitä verrattain vähän. (van der Linden 2004)

Neto (2009) on testannut edellä mainittua merkkijonojen internointia ja hän toteaa, että vaikka `intern()`-metodin hyödyntäminen onkin nopeampaa kuin `equals()`-metodin käyttäminen, ei nopeusero ole niin suurta, että siitä olisi todellista hyötyä. Hänen mukaansa tämä johtuu siitä, että `equals()`-metodin kutsu ei maksa paljoa suoritinaikaa ja sen kaltaiseen pieneen metodiin on helppo soveltaa esimerkiksi inline-optimointia. Lisäksi hän toteaa, että Javan alkuaikoina merkkijonovarastoon vietyjen `Stringien` voitiin katsoa kestävän ohjelman koko suorituksen ajan. Mutta hänen mukaansa roskienkerääjä tuhoaa nykyisin jopa merkkijonovarastossakin olevia olioita, jos näihin ei ole viitteitä.

Neto myös kritisoi oletettua muistinsäästöä, koska internoitu merkkijono siirretään Permanent Generation -alueelle, joka on JVM:n alue. Tämä alue on tarkoitettu erityisesti järjestelmän luomille olioille. Tämän alueen muisti on rajallinen ja usein pienempi kuin kekomuisti, jolloin se saattaa tulla useista internoinneista täyteen ja kaataa ohjelman. (Neto 2009) Toisaalta Kohler (2009) vastaa Netolle, että `String`-oliot tyypillisesti vievät Java-ohjelman muistista noin 20 – 50 %, joten internointi on kätevä ja tehokas tapa poistaa turhat merkkijonokopiot ja säästää muistia.

Esimerkiksi, jos ladataan muistiin iso taulukkotiedosto voi esimerkiksi 20 000 taulukkorivin lataaminen viedä paljon muistia, jos jokaisesta tekstialkiosta luodaan `String`-olio. Usein tietokannoissa ja taulukoissa on myös paljon

samoja merkkijonoja alkioina. Kohler (2009) käyttää esimerkkinä kaupungin nimeä, joka todennäköisesti henkilötietokannassa on usealla henkilöllä sama. Tällaisessa tilanteessa merkkijonojen varastoinnilla `intern()`-metodilla säästetään paljon muistia, kun identtiset merkkijonot hävitetään.

Tästäkin aiheesta on useita mielipiteitä, mutta on luultavasti parasta olla tavallisissa ohjelmissa käyttämättä turhaan merkkijonojen varastointia, koska se itseasiassa saattaa hidastaa ohjelmaa tai aiheuttaa muita, edellä mainittuja ongelmia. Toisaalta, jos ohjelmassa käsitellään runsaasti merkkijonoja, ohjelman muistista suurin osa kuluu merkkijono-olioihin tai ohjelmistossa suoritetaan paljon merkkijonojen vertailua, on merkkijonojen varastointi mahdollisesti yksi mahdollisuus parantaa ohjelmiston suorituskykyä.

5 SEKALAISIA KEINOJA JAVA-KOODIN TEHOSTAMISEKSI

Tässä luvussa tullaan käymään lävitse Java-koodille mahdollisesti suoritettavia optimointeja, jotka eivät teemoiltaan sovi muihin lukuihin. On huomattava, että osa lukujen sijoittelusta on keinotekoista, koska osittain keinot saattavat liittyä esimerkiksi olioiden luomiseen tai tuhoamiseen. Tässä luvussa tullaan käsittelemään perustietotyyppejä ja käärintää sekä `final`-määrettä ja poikkeuksia.

5.1 Perustietotyypit ja käärintä

Java-maailmassa on kahdenlaisia tyyppejä, primitiivityyppejä ja viitetyyppejä. Aiempiin lukeutuvat sellaiset tyypit, kuten `int`, `double` ja `char`. Jälkimmäisiin lukeutuvat muun muassa `String` ja `List <E>`. (Bloch 2008)

Joskus Java-ohjelmissa tulee tarve muuttaa primitiivityyppi viitetyypiksi. Tällainen tilanne syntyy esimerkiksi, jos metodi vaatii parametrikseen `Object`-tyyppisen olion, mutta käytössämme oleva arvo on primitiivityyppinen. Esimerkiksi useat Javan API-luokkien metodeista ovat sellaisia, jotka vaativat parametrikseen `Object`-olion.

Tämän vuoksi Javassa on viitetyyppeihin kuuluvia niin sanottuja kääreluokkia, jotka sisältävät primitiivityypin olion sisällä. Tällöin kyseistä primitiivityyppiä voidaan käyttää siellä, mihin primitiivityyppiä ei normaalisti voisi käyttää.

Java osaa versiosta 1.5 eteenpäin automaattisesti vaihtaa tarvittaessa primitiivityypistä kääreluokkaolioksi. Tätä kutsutaan käärinnäksi (engl. boxing tai autoboxing), joka tosin vaatii luonnollisesti aina olion luomista, mikä saattaa jossain tilanteissa haitata suorituskykyä. Vastaavasti Java osaa tehdä saman toi-

sinpäin eli purkaa kääreluokkaolion sisällön primitiivityyppiin. Tätä toimintoa kutsutaan purkamiseksi (engl. unboxing). Kirjallisuudessa tämä Javan ominaisuus tunnetaan yleisesti englanninkielisellä nimellä autoboxing. Esimerkeissä 9 ja 10 selvennetään automaattisen käärinnän ideaa.

Esimerkki 9: automaattinen käärintä (van der Linden 2004)

```
int i = 27;

Integer myInt = i;  // käärintä
```

Esimerkin 9 koodin ongelmana on, että vaikka se saattaa nopeasti katsottuna näyttää tavanomaiselta sijoitukselta, se tarkoittaa samaa kuin alla näkyvä Esimerkki 10. Eli vaikka asiaa ei eksplisiittisesti näytetä, on kyseessä uuden olion luonti.

Esimerkki 10: automaattinen käärintä (van der Linden 2004)

```
Integer myInt = new Integer(i);
```

Esimerkissä 11 on esiteltynä toinen suorituskykyyn vaikuttava käärintäesimerkki. Esimerkkiä ajettaessa nähdään, että ohjelma toimii, mutta sen suorituskyky on heikko jatkuvan käärinnän ja purkamisen takia.

Esimerkissä 11 on metodi nimeltä `asList()`, joka palauttaa `AbstractList<Integer>` -tyyppisen olion. Tämä olio on toteutettu sisäluokkana, joka sisältää `size()`-, `get()`- ja `set()`-metodit. Metodi `get()` palauttaa halutun indeksin osoittaman arvon taulukosta, joka annettiin `asList()`-metodin parametrina. Koska taulukon alkiot ovat primitiivityyppiä `int` ja `get()`-metodi palauttaa viitetyyppiä `Integer` olevan arvon, tapahtuu tässä kohdassa käärintä.

Purkamista esiintyy `set()`-metodissa sen toiseksi viimeisellä rivillä. Tässä yritetään sijoittaa viitetyyppistä `Integer`–muuttujaa nimeltä `val` taulukkoon, jonka alkiot ovat primitiivityyppiä `int`. Tällöin tehdään automaattinen muunnos `Integer`-oliosta `int`-primitiivityypiksi. Selvyyden vuoksi koodiesimerkissä kohdat, joihin autoboxing vaikuttaa, on merkitty alleviivauksella.

Esimerkki 11: automaattisen käärinnän hidastama ohjelmakoodi (Bloch 2008, 95)

Virheellinen esimerkki!

```
public static List<Integer> asList(final int[] a) {

    return new AbstractList<Integer>() {
        // int → Integer (käärintä)
        public Integer get(int i) { return a[i]; }
        public Integer set(int i, Integer val) {
            // int → Integer (käärintä)
            Integer oldVal = a[i];
            // Integer → int (purkaminen)
            a[i] = val;
            return oldVal;
        }
        public int size() { return a.length; }
    };
}
```

Käärimisen käyttäminen vaatii aina olion luomista, joka on suhteellisen hidas tapahtuma. Niinpä kääreolioiden sijaan tulisikin aina käyttää perustietotyyppejä, jos kääreolioiden käyttöön ei ole mitään pakottavaa syytä. Tällainen syy voi olla esimerkiksi kääreolion käyttö alkiona tai avaimena kokoelmissa (Bloch 2008).

5.2 final-määre

Määrittelemällä luokka `final`-määreellä, ei luokasta voida enää periyttää aliluokkia. Metodin määritteleminen `final`-määreellä aiheuttaa vastaavasti sen, ettei metodia voi ylikirjoittaa mahdollisissa aliluokissa.

Tällainen muuttumattomuus mahdollistaa joitakin optimointeja. Virtuaalikone voi käyttää staattista sidontaa ja inline-lavennusta (Vesterholm & Kyppö 2008). Esimerkissä 12 esitetään, kuinka luokka määritellään lopulliseksi `final`-määrellä.

Esimerkki 12: luokan määritteleminen `final`-määreellä

```
public final class Aika { /* ... */ }
```

Toisaalta tällaista `final`-määreen käyttöä kritisoidaan, koska sen hyödyt ovat vähäiset ja luokkien ja metodien määrittely `final`-määreellä haittaa näiden uudelleenkäytettävyyttä. Lisäksi tällainen tapa määritellä luokkia ja metodeja `final`-määreellä haittaa vakavasti olioperusteista ohjelmointia (Goetz 2002). Tämä johtuu siitä, että jos luokkia ja metodeita systemaattisesti määritellään `final`-määreellä pelkästään suorituskyvyn vuoksi, aiheutetaan samalla se, että esimerkiksi aliluokkien luominen on mahdotonta. Tämä taas on omalla tavallaan ristiriidassa Javan olioperusteisen ideologian kanssa. Lisäksi Goetzin (2002) mukaan luulo, että virtuaalikone tekisi inline-optimointeja `final`-määreellä on harhaanjohtava ja suorastaan väärä.

Koska tämä aihe on hieman epäselvä, vaatisi tämä asia lisätutkimusta siitä, onko `final`-määreen käytöllä nykyään mitään järkevää syytä suorituskyvyn takia. Luultavasti kuitenkaan sen käytöstä ei ole niin suurta hyötyä, että sen

käyttö kannattaisi pelkästään suorituskyvyn vuoksi, vaan `final`-määreen käyttö tulisi jättää sinne, minne se koodin järkevyyden kannalta kuuluisikin.

5.3 Poikkeusten käytöstä ohjausrakenteena

Javan poikkeukset ovat luku sinällään ja on monta tapaa käyttää niitä väärin. Tässä tutkielmassa paneudutaan vain niiden väärinkäytön suorituskyvylliseen puoleen.

Bloch (2008) toteaa osuvasti, että poikkeuksia tulisi käyttää vain poikkeustilanteissa, ei osana tavallisia ohjelman ohjausrakenteita. Hän tarjoaa myös esimerkin poikkeuksien käytöstä ohjausrakenteena, joka on nähtävissä Esimerkissä 13.

Esimerkki 13: poikkeusten virheellinen käyttäminen ohjausrakenteena (Bloch 2008, 241)

Virheellinen esimerkki!

```
try {
      int i = 0;
      while(true)
            range[i++].climb();
} catch(ArrayIndexOutOfBoundsException e) {
}
```

Esimerkin 13 ohjelma saattaa näyttää erikoiselta, mutta sen idea on yksinkertaisesti taulukon läpikäyminen. Kun saavutaan taulukon loppuun, ohjelma heittää `ArrayIndexOutOfBoundsException` -poikkeuksen ja silmukan suoritus päättyy. Luonnollisempi ja oikea tapa toteuttaa sama olisi Esimerkissä 14 esitetyllä tavalla.

Esimerkki 14: esimerkin 13 korjattu versio (Bloch 2008)

```
int i = 0;
while (i < range.length)
     range[i++].climb();
```

Esimerkin 13 kaltaista ohjelmakoodia on joskus saatettu käyttää, koska varsinkin Javan alkuaikoina kuviteltiin, että algoritmi nopeutuu, kun jätetään pois indeksintarkistus (kohta `i < range.length`). Tämä siksi, että JVM kuitenkin koko ajan tarkistaa, ollaanko menty indeksin rajan yli ja pitääkö nyt heittää poikkeus. Ajateltiin, että indeksin tarkistus on turhaa rasitetta, sillä samaan aikaan JVM tarkistaa kuitenkin indeksin. Kuitenkin tällainen poikkeusten väärinkäyttö estää JVM:n omien optimointien toimimisen ja on nykyisten virtuaalikoneiden kanssa hitaampaa. (Click 2003)

Bloch (2008) toteaakin omissa testeissään poikkeuksella toteutetun ohjelmakoodin (Esimerkki 13) olleen huomattavasti hitaampi. Syiksi hän luettelee kolme seikkaa. Ensinnäkin poikkeuksia käytetään vain harvoissa kohtia koodia, eikä Javan kehittäjillä siksi ole ollut niiden suorituskyvyn optimointiin niin suuria paineita. Toiseksi `try-catch` -osa saattaa myös hänen mukaansa estää virtuaalikonetta suorittamasta tiettyjä optimointeja. Viimeiseksi hän toteaa, että Esimerkin 14 tapauksessa moderni JVM ei välttämättä edes testaa indeksiarvoja jokaisella silmukan suorituskerralla.

Lisäksi on helppo huomata, että poikkeusten väärinkäyttö tällä tavoin tekee koodista hyvin sekavaa ja lisäksi aiheuttaa mahdollisuuden, että koodi ei toimi toivotulla tavalla. Näin ollen poikkeukset tulisi jättää sinne, missä niitä oikeasti tarvitaan, eikä yrittää käyttää niitä keinotekoisina ohjausrakenteina.

6 YHTEENVETO

Aloittaessani aiheen tutkinnan ajattelin, että löytyisi paljonkin keinoja, joilla Java-koodia voisi tilanteessa kuin tilanteessa nopeuttaa ja tehostaa. Kuitenkin tutkielman edetessä on tullut selväksi, että tällaisia keinoja on todella harvassa ja lopuissa keinoissa on aina omat haittansa, jonka vuoksi ne eivät sovellu kuin tiettyihin tapauksiin.

Järkevin tapa on ohjelmoida hyvää, selkeää Java-koodia välittämättä erityisesti suorituskyvystä ja jättää koodin optimoiminen pääosin JVM:n vastuulle. Keinot, jotka voivat tehostaa suorituskykyä, mutta eivät ole hyvän koodaustavan mukaisia, saattavat menettää suorituskykyhyötynsä uusien JVM-versioiden mukana. Näin on tapahtunut aiemminkin ja varmasti tulee tapahtumaan tulevaisuudessakin.

Jos kuitenkin ohjelman suorituskyky on liian alhainen, tulisi ohjelmaa tutkia esimerkiksi niin sanotuilla profiler-ohjelmistoilla ja etsiä ohjelman alueet mitkä vaatisivat optimointia. Tämän jälkeen voidaan soveltaa esimerkiksi tässä tutkielmassa mainittuja neuvoja tarvittavissa kohdissa. Lisäksi, jos kokemuksella tiedetään, että ohjelma tulee normaalilla tavalla toteutettuna olemaan liian raskas, silloin näitä keinoja voidaan käyttää mahdollisuuksien mukaan jo etukäteen.

Java-koodia voidaan optimoida usealla tavalla, mutta on epäselvää, onko kaikista keinoista mitään käytännön hyötyä. Osa optimoinneista voi olla epäkäytännöllisiä ja jopa haitata ohjelmiston myöhempää kehitystyötä. Tällaisesta esimerkkinä on luokkien määritteleminen `final`-määreellä.

Suurimmat hyödyt optimoinnista saavutetaan hyvin tarkasti määritellyissä erikoistapauksissa ja tarve optimointiin herää vasta sitten, kun ohjelmisto ei toimi tarpeeksi nopeasti. Tällaisia tapauksia luultavasti ovat esimerkiksi merkkijonojen käsittely, joiden runsaiden liittämisten yhteydessä saattaa olla parempi käyttää `StringBuilderia` normaalin `String`-luokan sijaan.

Toisaalta osa näistä keinoista pitäisi olla jokaisen Java-ohjelmoijan perustietoa. Näillä toimintamalleilla ei ole juuri mitään negatiivisia seurauksia ja ne parhaimmillaankin vain selventävät ja nopeuttavat koodia. Tällaisia optimointeja ovat esimerkiksi turhien olioiden luomisen välttäminen, niiden kierrättäminen ja olioiden välttäminen, jos voidaan käyttää perustietotyyppejä.

Kuten Goetz (2003) toteaa, tulisi optimoinnit säästää sinne, missä niitä tarvitaan ja käyttää optimointeja, joista on selvää hyötyä. Ongelmana vain on, että kirjallisuudesta ei juurikaan löydy kattavia ja ajanmukaisia testejä optimointien oikeasta suorituskyvystä ja toimivuudesta. Ei varsinkaan sellaisia, joissa optimointeja olisi testattu useita.

Jatkotutkimuksen kannalta olisikin mielenkiintoista tutkia, miten tehokkaita nämä, osittain antiikkisetkin keinot ovat nykypäivänä. Todennäköisesti tulenkin jatkamaan tästä aiheesta pro gradu -tutkielmassa, jolloin empiirisesti tutkin optimointien tehokkuutta ja mielekkyyttä. Erityisesti `final`-määreen käyttöön ja merkkijonojen internointiin suhtaudutaan kirjallisuudessa ristiriitaisesti, joten näiden tutkiminen voisi olla hyvinkin mielenkiintoista.

Lopuksi yhdyn siihen, mitä Bloch (2008, 236) toteaa kirjassaan ja mikä tutkielmaa kirjoittaessani on selventynyt itsellenikin:

> To summarize, do not strive to write fast programs—strive to write

> good ones; speed will follow. [...] When you've finished building the system, measure its performance. If it's fast enough, you're done. If not, locate the source of the problems with the aid of a profiler, and go to work optimizing the relevant parts of the system.

LÄHTEET

Bishop P. & Warren N., 1999. Java Tip 67: Lazy instantiation [online]. JavaWorld [viitattu 27.4.2010]. Saatavilla www-osoitteessa: <http://www.javaworld.com/javaworld/javatips/jw-javatip67.html>

Bloch J., 2008. Effective Java (2. painos). Upper Saddle River, NJ, USA: Prentice Hall PTR.

Bulka D., 2000. Server-side programming techniques. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Click C., 2003. Performance myths exposed [online]. JavaOne 2003 Conference [viitattu 28.4.2010]. Saatavilla www-osoitteessa: <http://www.ujug.org/stuff/1522_performance_myths_exposed.pdf>

Goetz B., 2002. Java theory and practice: Is that your final answer? [online]. IBM developerWorks Java Technical Library [viitattu 5.4.2010]. Saatavilla www-osoitteessa: <http://www.ibm.com/developerworks/java/library/j-jtp1029.html>

Goetz B., 2003. Java theory and practice: Urban performance legends [online]. IBM developerWorks Java Technical Library [viitattu 2.4.2010]. Saatavilla www-osoitteessa: <http://www.ibm.com/developerworks/java/library/j-jtp04223.html>

Goetz B., 2004. Java theory and practice: Garbage collection and performance [online]. IBM developerWorks Java Technical Library [viitattu 5.4.2010]. Saatavilla www-osoitteessa: <http://www.ibm.com/developerworks/java/library/j-jtp01274.html>

Gosling J., Joy B., Steele G. & Bracha G., 2005. Java language specification (3. painos). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Holmes D., 2010. Minimize garbage generation: GC is your friend, not your servant [online]. David Holmes' WebBlog [viitattu 30.4.2010]. Saatavilla www-osoitteessa: <http://blogs.sun.com/dholmes/entry/minimize_garbage_generation>

JavaAPI, 2009. Java Platform, Standard Edition 6 API Specification [online]. Sun Microsystems, Inc. Saatavilla www-osoitteessa:

<http://java.sun.com/javase/6/docs/api/>

Kircher M. & Jain P., 2002. Pooling. Munich, Germany: Siemens AG.

Klemm R., 1999. Practical guidelines for boosting Java server performance. Anonymous (toim.) JAVA '99: Proceedings of the ACM 1999 Conference on Java Grande, San Francisco, California, United States. 25-34.

Kluge K., 1997. The experts talk: Thirteen great ways to increase Java performance [online]. Sun Developer Network [viitattu 5.4.2010]. Saatavilla www-osoitteessa: <http://java.sun.com/developer/technicalArticles/Programming/Performance/>

Kohler M., 2009. Is java.lang.String.intern() really evil? [online]. Java Performance Blog [viitattu 28.4.2010]. Saatavilla www-osoitteessa: <http://kohlerm.blogspot.com/2009/01/is-javalangstringintern-really-evil.html>

Krill P., 2008. Java performance improvements touted [online]. InfoWorld [viitattu 28.4.2010]. Saatavilla www-osoitteessa: <http://www.infoworld.com/d/developer-world/java-performance-improvements-touted-608>

van der Linden P., 2004. Just Java 2: J2SE 1.5 Edition (6. painos). Upper Saddle River, NJ, USA: Prentice Hall PTR.

Neto D., 2009. Busting java.lang.String.intern() myths [online]. Code Instructions Blog [viitattu 28.4.2010]. Saatavilla www-osoitteessa: <http://www.codeinstructions.com/2009/01/busting-javalangstringintern-myths.html>

Sakkinen M., 2007. Olio-ohjelmointi. Jyväskylä: Jyväskylän yliopisto, tietojenkäsittelytieteiden laitos.

Shirazi J., 2002. Java performance tuning. Sebastopol, CA, USA: O'Reilly & Associates, Inc.

Stoodley M., Ma K. & Lut M., 2007. Real-time Java, part 2: Comparing compilation techniques [online]. IBM developerWorks Java Technical Library [viitattu 4.4.2010]. Saatavilla www-osoitteessa: <http://www.ibm.com/developerworks/java/library/j-rtj2/index.html#N10153>

Sun Microsystems, 2006. The Java HotSpot Performance Engine Architecture [online]. Sun Microsystems, Inc [viitattu 8.5.2010]. Saatavilla www-osoitteessa: <http://java.sun.com/products/hotspot/whitepaper.html>

Vesterholm M. & Kyppö J., 2008. Java-ohjelmointi (7. painos). Helsinki: Talentum.

Wikla A., 2003. Ohjelmoinnin perusteet Java-kielellä. Espoo: OtaData.

Wilson S. & Kesselman J., 2000. Java platform performance: Strategies and tactics. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.