Funktionaalisen- ja olioparadigman suorituskykyvertailu Scala-kielessä

LuK-tutkielma
Turun yliopisto
Tulevaisuuden teknologioiden laitos
Labra
2019
Jaakko Paju

Sisältö

1 Johdanto

Funktionaalinen ohjelmointi kasvattaa suosiotaan jatkuvasti. Useisiin yleiskäyttöisiin ja alunperin imperatiivisiin ohjelmointikieliin on lisätty ominaisuuksia funktionaalisesta ohjelmoinnista. Esimerkiksi suositut oliokielet Java, Python ja C++ kaikki mahdollistavat anonyymit- ja korkeamman asteen funktiot. TODO: Tönkkö aloitus, korjaa

Yksi funktionaalisen paradigman keskeisistä käsitteistä on muuttumattomat arvot ja tietorakenteet. Muuttomattomien arvojen käyttäminen lisää kopioimisen tarvetta verrattuna ohjelmiin, joissa käytetään muuttuvia arvoja. Kopioimisen seurauksena muistia pitää varata ja vapauttaa useammin, kuin muuttuvia arvoja käytettäessä. Tämän seurauksena nousee kysymys funktionaalisten ohjelmien suorituskyvystä verrattuna imperatiivisiin ohjelmiin.

Tutkielmassa perehdytään tarkastelemaan millaisia vaikutuksia funktionaalisella paradigmalla on suorituskykyyn verrattuna olio-paradigmaan. Tutkielman suorituskykyvertailut keskittyvät Scala-ohjelmointikieleen, sillä se on suunniteltu tukemaan sekä funktionaalista- että olioparadigmaa, jolloin suorituskyvyn vertailu näiden kahden paradigman välillä on mielekästä ja suoraviivaista. Tarkastelu kohdistuu erityisesti muuttumattomiin kokoelmiin. TODO: Lisää toinen tutkittava aihe Tutkielma on suoritettu perehtymällä aihetta käsittelevään kirjallisuuteen.

Luvussa 2 esitellään molemmat vertailun kohteena olevat paradigmat. Luvussa 3 esitellään Scala-kielen rakenteet ja miten ne tukevat kumpaakin paradigmaa. Luvussa 4 tarkastellaan Scalan standardikirjaston muuttumattomien kokoelmien suorituskykyyn ja ver-

rataan sitä muuttuvien kokoelmien suorituskykyyn. TODO: Varmista että suorituskykyä oikeasti verrataan muuttuviin kokoelmiiin Viimeisenä luvussa 5 esitellään johtopäätökset ja kootaan tutkielman tulokset.

2 Ohjelmointiparadigmat

Ohjelmointiparadigmat luokittelevat ohjelmointikieliä sen perusteella, miten kieli on suunniteltu mallintamaan ongelmia ja minkälaisia mekanismeja kieli tarjoaa näiden ongelmien ratkaisuun. Yhdessä ohjelmointikielessä voi olla vaikutteita useammasta paradigmasta. Tälläistä kieltä kutsutaan *moniparadigmaiseksi* kieleksi. Ohjelmointiparadigmat voi jakaa karkeasti kahteen yläluokkaan: *imperatiivisiin* ja *deklaratiivisiin*. [1, Luku 6]

Imperatiiviset kielet keskittyvät ohjelmointiongelman ratkaisuun määrittelemällä *miten* tietokoneen tulisi ratkaista ongelma. Ohjelmat siis rakentuvat peräkkäisistä käskyistä, jotka muokkaavat ohjelman tilaa ja näin lopulta ratkaisevat ongelman. Imperatiivisista kielistä puhutaan matalamman tason kielinä, sillä ongelman ratkaisu mallinnetaan niissä tietokoneen näkökulmasta. [2, Luku 1]

Deklaratiiviset kielet ratkaisevat ongelman vastaamalla kysymykseen *mitä* tietokoneen tulisi tehdä ongelman ratkaisemiseksi. Käytännössä tämä siis tarkoittaa ongelman ratkaisun kuvailemista ilman toteutuksen yksityiskohtiin uppoutumista. Deklaratiiviset mielletään yleensä korkeamman tason kielinä, sillä ne mallintavat ongelmanratkaisua ohjelmoijan näkökulmasta. [2, Luku 1]

2.1 Olio-paradigma

Olio-ohjelmoinnnin perusajatus on kuvata ongelmaa olioiden avulla, jotka kukin kuvaavat jotakin ongelma-alueen käsitettä. Olioiden tarkoitus on kapseloida kuvaamansa käsitteen tieto ja tila sisäänsä, sekä tarjota operaatioita kapseloidun tiedon muokkaamiseen ja

tarkasteluun. Olio-paradigma on osa imperatiivisia paradigmoja, sillä ongelmanratkaisu tapahtuu peräkkäisillä komennoilla, jotka muuttavat olioiden ja samalla koko ohjelman tilaa kohti ratkaistua ongelmaa. [2, Luku 1] [1, Luku 10]

2.2 Funktionaalinen paradigma

TODO: Vähän tönkösti ilmaistu, pitänee muotoilla uudestaan Nimensä mukaisesti funktionaaliset ohjelmat perustuvat funktioihin. Näillä funktioilla ei ole tilaa, ja ne ovat puhtaita eli eivät aiheuta *sivuvaikutuksia*. Tilattomuus ja puhtaus tarkoittaa että samalla syötteellä funktio palauttaa aina saman arvon, ei heitä poikkeusta eikä muuta ohjelman tilaa. [3, Luku 1] Ohjelmointikieli ilman sivuvaikutuksia olisi käytännössä hyödytön, joten funktionaaliset ohjelmointikielet tarjoavat erilaisia mekanismeja sivuvaikutusten hallintaan. Funktionaalisen ohjelmoinnin katsotaan yleensä kuuluvan deklaratiiviseen paradigmaan. [1, Luku 11]

2.3 Paradigmoille tyypillsiä ominaisuuksia

Lauseke on ohjelmointikielen rakenne jonka suoritus tuottaa arvon. Lauseke voi koostua joko arvosta tai operaattorista jota on sovellettu yhteen tai useampaan lausekkeeseen. Esimerkiksi `3+2` on numeerinen lauseke, jossa on yksi operaattori + jota sovelletaan kahteen numeeriseen arvoon `3` ja `2`. Lausekkeen arvo voidaan sijoittaa muuttujaan, antaa parametriksi funktioon tai sen arvo voidaan palauttaa funktiosta. Lauseke on minkä tahansa ohjelmointikielen perusrakenne, eli lausekkeita on funktionaalisissa- sekä oliokielissä. [1, Luku 6]

Funktionaalisessa- ja olio-paradigmassa molemmissa on muuttujia, mutta niitä käsitellään eri tavoilla. Funktionaalisissa kielissä muuttujiin sijoitettua arvoa ei voi muuttaa alustuksen jälkeen, kun taas imperatiivissa kielissä muuttujan arvoa voi yleensä muuttaa. Sama pätee tietorakenteisiin: imperatiivisssa kielissä niiden muuttaminen alustami-

sen jälkeen on sallittua, funktionaalisissa ei. [3, Luku 3] Funktionaalisissa kielissä funktioita kohdellaan kuin mitä tahansa muitakin arvoja: niitä voi sijoittaa muuttujiin, antaa parametrina toiseen funktioon tai käyttää funktion palautusarvona. Olio-kielissä tämä ei aina ole mahdollista. [1, Luku 6]

Komento on ohjelmointikielen rakenne jonka suoritus ei aina tuota arvoa ja saattaa aiheuttaa sivuvaikutuksia. Komennot saattavat esimerkiksi muuttaa olioiden ja muuttujien tilaa, lukea käyttäjän tuottamaa syötettä tai heittää poikkeuksen. Komentojen suorittamisella saattaa olla sivuvaikutuksia, ja siitä syystä niiden suoritusjärjestys on merkityksellinen. Imperatiivisiin kieliin komennot kuuluvat oleellisesti, mutta funktionaalisiin kieliin eivät.

Imperatiivisissa kielissä iterointia kuvataan silmukoilla, joissa tiettyä osaa ohjelmasta suoritetaan useita kertoja peräkkäin. Silmukkaa suoritetaan ennalta määritelty määrä, tai kunnes silmukan suoritusta säätelevä ehtolause muuttuu epätodeksi. Ehtolauseen arvo määräytyy muuttujan arvon mukaan, jota muutetaan silmukan suorituksen aikana. Funktionaalisissa kielissä muuttujien arvoa ei voi muuttaa, ja siitä syystä silmukat olisivat hyödyttömiä. Silmukoiden sijaan käytetään rekursiota, eli funktio kutsuu itseään kunnes funktioon määritelty perustapaus saavutetaan. [1, Luku 6 ja 11]

3 Scala

TODO: Aivan liian pitkä luku, pitää karsia huolella!

Scala on staattisesti tyypitetty moniparadigmainen käännettävä ohjelmointikieli, joka yhdistää funktionaalisen ohjelmoinnin ja olio-ohjelmoinnin ominaisuuksia. Scala on korkean tason kieli ja sen syntaksi on kompaktia ja eleganttia. Scalan kääntäjä ja tyyppijärjestelmä takaa tyyppiturvallisuuden kääntämisen aikana. Scala on yhteensopiva Javan kanssa, ja se mahdollistaa Java-kirjastojen käyttämisen suoraan Scalasta. Scala-koodi on tarkoitettu käännettäväksi Javan tavukoodiksi, ja sitä ajetaan Javan virtuaalikoneessa (JVM). [4, Introduction] [5, Luku 2]

3.1 Kielen synty

Scalan on kehittänyt Lausannen teknillisen yliopiston professori Martin Odersky. Ennen Scalan kehittämistä Odersky kehitti yhdessä Phil Wadlerin kanssa Pizza-nimisen ohjelmointikielen. Pizzan tarkoituksena oli tuoda funktionaalisen ohjelmoinnin piirteitä Java-maailmaan. Pizza julkaistiin vuonna 1996, vain vuosi Javan julkaisun jälkeen. Vuosina 1997-1998 Odersky ja Wadler kehittivät Sun Microsystemsin pyynnöstä Generic Java-projektin (GJ), jonka tarkoituksena oli tuoda geneerisyys Javaan. Odersky kehitti projektin aikana kääntäjän, joka korvasi Sun Microsystemsin kehittämän Java-kääntäjän versiossa 1.3 vuonna 2000. Lopulta Generic Java-projektin aikana kehitetty geneerisyys liitettiin Javaan versiossa 5 vuonna 2004. [6]

Generic Java-projektin aikana Odersky koki Javan taaksepäin yhteensopivuuden ra-

joittavan liikaa uusien ominaisuuksien kehittämistä. Odersky päätti kehittää täysin uuden ohjelmointikielen, joka olisi Javaa kehittyneempi, mutta silti yhteensopiva Javan kirjastojen ja ajoympäristön kanssa. Tuloksena syntyi Scala,jonka ensimmäinen versio julkaistiin vuonna 2003. [6]

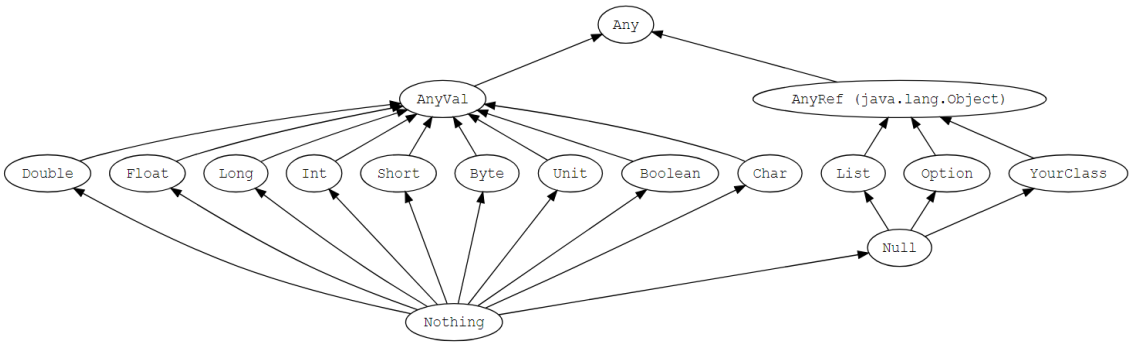
3.2 Muuttujat

Scalassa on tavallisten muuttujien lisäksi vakioita, jotka ovat muuttujia joiden arvoa ei alustuksen jälkeen voi muuttaa. Vakion määritelyyn käytetään avainsanaa **`val`** ja muuttujan määrittelyyn **`var`**. Muuttujan nimen jälkeen määritellään muuttujan tyyppi, joka erotetaan muuttujan nimesta kaksoispisteellä. Esimerkiksi **`val`** `x: Int` **`=`** `1` alustaa kokonaislukuvakion x, jonka arvo on 1. Muuttujan tyyppimäärittelyn voi jättää myös kirjoittamatta, sillä Scala-kääntäjä osaa yleensä päätellä muuttujan tyypin asetetun arvon perusteella. Edellisen esimerkin voi siis halutessaan kirjoittaa muodossa **`val`** `x` **`=`** `1`, ja Scala osaa päätellä muuttujan olevan tyypiltään kokonaisluku. [4, Basics]

Muuttujat ovat myös staattisesti tyypitettyjä, joten muuttujan tyyppi ei voi muuttua ajon aikana. Scalassa myös funktiot ovat arvoja, joten niitä voidaan sijoittaa muuttujiin. Esimerkiksi **`val`** `add1` **`=`** `(x: Int)` **`=>`** `x + 1` luo *funktioliteraalin* ja sijoittaa sen muuttujaan nimeltä add1. Funktioita käsitellään tarkemmin luvussa 3.5. [5, Luku 1]

3.3 Tietotyypit

Scala on puhtaasti oliokieli, ja kaikki muuttujat ovat olioita. Jokainen Scalan tietotyyppi periin luokan `Any` ja luokka `Nothing` on jokaisen tietotyypin alaluokka. Scalassa luokka voi olla arvoluokka tai viiteluokka. Arvoluokat perivät luokan `AnyVal` ja niihin ei voi sijoittaa null-arvoa. Viiteluokat perivät luokan `AnyRef`, joka on tyyppialias Javan luokalle `Object`. Jokaisen viiteluokan alaluokka on `Null`. Tietotyyppien hierarkia on kuvattu



Kuva 3.1: Scalan luokkahierarkia

kuvassa 3.1. [5, Luku 5]

Alkeistietotyyppejä kuvaavat arvoluokat `Byte`, `Short`, `Int`, `Long`, `Char`, `Float`, `Double`, `Boolean` ja `Unit`. Kaikki paitsi viimeinen vastaavat Javan kääreluokkia. `Unit` on erityinen tietotyyppi, joka kuvastaa funktion tyhjää paluuarvoa. Sen voi ajatella vastaavan Javan void-avainsanaa. Merkkijonoja Scalassa kuvaa tietotyyppi `String` joka on tyyppialias Javan merkkijonoa kuvaavalle tietotyypille. [5, Luku 5]

Scalan alkeistietotyypit muutetaan käännösvaiheessa primitiivityypeiksi, esimerkiksi Scalan `Int` käännetään 32-bittiseksi sanaksi, aivan kuten Javan `int`. Tämä takaa yhteensopivuuden Java-kirjastojen kanssa. Se parantaa myös suorituskykyä, sillä primitiiviarvolle ei tarvitse allokoida muistia ajon aikana. [5, Luku 6]

3.4 Kontrollirakenteet

3.4.1 Ehtolauseet

Ehtolauseita kirjoitetaan Scalassa *if/else*-lausekkeiden avulla. Lauseketta voi käyttää imperatiiviseen tyyliin eli jos ehto arvioituu todeksi, suoritetaan if-lohko, muuten else-lohko.

```
val x = true
if (x) {
    println("True")
} else {
    println("False")
}
```

```
// Tulostaa "True"
```

Scalan if-lausekeella on myös arvo, eli if-lauseke palauttaa arvon. Esimerkiksi `val y = if (x >= 0) "positive" else "negative"` asettaa muuttujan `y` arvoksi joko `"positive"` tai `"negative"`, riippuen muuttujan `x` arvosta. Huomattavaa on myös, että if-lauseke voidaan kirjoittaa yhdelle riville ilman aaltosulkeita. [7, Luku 2.1]

3.4.2 Silmukat

Silmukoita Scalassa on kolmenlaisia: **`while`**, **`do`** ja **`for`**. While-silmukka suorittaa sitä seuraavaa lohkoa 0-n kertaa, kunnes annettu ehto arvioituu epätodeksi. Do-silmukka toimii vastaavalla tavalla, mutta sitä suoritetaan 1-n kertaa.

```
val x = false
while (x)
  println("while")

do
  println("do")
    while (x)
```

Yllä oleva esimerkki ei suorita while-silmukkaa kertaakaan, ja do-silmukan kerran. [7, Luku 2.5]

Scalan **`for`**-lauseke on monipuolisempi, kuin monissa muissa kielissä. Sitä voi käyttää perinteiseen tapaan silmukkana, jolloin for-lausekkeen *generaattori* antaa määriteltyjä arvoja yksi kerrallaan muuttujan `i` kautta. Lauseketta voidaan käyttää myös usean generaattorin kanssa ja arvoja voidaan suodattaa lausekkeen sisällä. Kuten if-lauseke, myös for-lauseke voi palauttaa arvon. Seuraavaksi esimerkki kummastakin käyttötavasta.

```
for (i <- 1 to 10)
    println(i)
```

Silmukkana lauseketta voi käyttää yllä olevan esimerkin tapaan tulostamaan kaikki numerot yhdestä kymmeneen.

```
val res = for {
    x <- 1 to 10
    if x % 2 == 0
} yield (x * 2) // Vector(4, 8, 12, 16, 20)
```

Yllä olevassa esimerkissä luodaan kokonaisnumerovektori yhdestä kymmeneen, suodatetaan parilliset arvot, palautetaan jäljelle jääneet arvot kahdella kerrottuna ja asetetaan palautettu vektori muuttujaan `res`. [7, Luku 2.6]

3.5 Metodit ja funktiot

Jokainen operaatio Scalassa on metodikutsu. Myös tavanomaiset aritmeettiset operaattorit on toteutettu metodikutsuina. Esimerkiksi kahden kokonaisluvun yhteenlasku `1 + 2` on lyhyempi versio metodikutsusta `1.+(2)`. Scalassa on mahdollista määritellä funktio kahdella tavalla: olion metodiksi tai funktioliteraaliksi. Metodi voidaan määritellä **`def`**-avainsanalla. [7, Luku 1.4]

3.5.1 Funktioliteraalit

Funktioliteraali on funktio, jota käsitellään kuten mitä tahansa muuttujaa. Yleensä funktioliteraali sijoitetaan muuttujaan, tai annetaan parametriksi toiselle funktiolle. Kuten muillakin muuttujilla, funktioilla ja metodeilla on Scalassa aina tyyppi. Tyypin voi määritellä eksplisiittisesti, mutta monesti Scala osaa päätellä funktion tyypin. Seuraavassa esimerkissä määritellään kummallakin tavoilla funktio, joka kertoo onko parametrina annettu kokonaisluku parillinen. Funktion tyyppi on siis `Int => Boolean`. [5, Luku 8]

```
def isEven1(x: Int): Boolean = x % 2 == 0
val isEven2: (Int => Boolean) = x => x % 2 == 0
```

3.5.2 Korkeamman asteen funktiot

Funktio voi ottaa parametriksi toisen funktiohin ja funktio voi palauttaa funktion. Tälläisiä funktioita kutsutaan *korkeamman asteen funktioiksi*. Korkeamman asteen funktiot ovat yksi funktionaalisen ohjelmoinnin kulmakivistä. Seuraavassa esitellään korkeamman asteen funktio, joka ottaa parametrina kokonaisluvun sekä funktion kokonaisluvusta kokonaislukuu, ja palauttaa kutsuu tätä funktiota parametrin kokonaisluvulle.

```
def mapValue(v: Int, m: Int => Int): Int = m(v)

def double(x: Int): Int = x * 2

mapValue(2, double)
mapValue(2, _ * 3)
```

Kuten yllä olevasta esimerkistä huomataan, annetaan `double`-funktio parametrinä ihan kuten mikä tahansa muukin muuttuja. Viimeisellä rivillä käytetään funktioliteraalia `_ * 3`, joka on lyhyempi muoto funktioliteraalille `x => x * 3`. [7, Luku 12]

3.6 Luokat ja oliot

Scalassa on useita erilaisia luokka- ja oliotyyppejä, sekä tapoja luoda instansseja luokista. Tässä luvussa esitellään pääpiirteittäin erilaiset luokkatyypit.

3.6.1 Class

Luokka voidaan määritellä Scalassa **`class`**-avainsanalla. Luokka voi olla tyhjä, tai se voi sisältää metodeita ja muuttujia. Luokasta luodaan instanssi **`new`**-avainsanalla. Luokan sisältämät kentät voidaan määritellä konstruktorissa heti luokan nimen jälkeen suluissa. Konstruktorissa määritellyt kentät ovat oletuksena **`private val`**. Luokasta saadaan uusi instanssi **`new`**-avainsanan avulla. Ihmistä kuvaava luokka ja instanssi voidaan määritellä seuraavalla tavalla:

```
class Person(val name: String, var age: Int) {
```

```
    def isAdult = age >= 18
}
val p = new Person("Matti", 30)
```

Luokassa Person on kaksi kenttää: julkinen muuttumaton kenttä nimi ja julkinen muutettava ikä. Lisäksi luokassa on metodi, joka kertoo onko henkilö täysi-ikäinen. [4, Classes]

3.6.2 Case class

Scalassa on myös erityisiä case-luokkia, joita käytetään yleensä kuvaamaan muuttumatonta dataa. Case-luokka määritellään kuten tavallinen luokka, mutta konstruktorissa määriteltävät kentät ovat oletuksena `public` **`val`**. Case-luokkaan voidaan määräitellä metodeita samoin kun tavalliseenkin luokkaan. Case-luokan instanssia luotaessa ei tarvitse käyttää **`new`**-avainsanaa. Puhelinmallia kuvaava case-luokka ja instanssi voidaan määritellä seuraavalla tavalla:

```
case class Phone(brand: String, model: String)
val p = Phone("Google", "Pixel4")
```

Scala-kääntäjä lisää case-luokkiin `toString`-, `hashCode`-, `equals`- ja `copy`-metodeita oletustoteutuksilla. Lisäksi kääntäjä lisää *kumppaniolioon* `apply`-tehdasmetodin, jonka avulla uusien instanssien luonti tapahtuu. [5, Luku 15]

Tavallisia luokkia käytetään varsinkin olio-tyylisessä ohjelmoinnissa, kun taas case-luokat ovat käytössä erityisesti funktionaalisessa Scalassa, kun halutaan käyttää muuttumattomia olioita. Tavallisten luokkien ja case-luokkien yhtäsuuruusvertailut eroavat toisistaan merkittävästi. Tavallisia luokkia verrataan viittauksen mukaan, eli osoittaako viittaus samaan olioon JVM:ssä. Case-luokkien yhtäsuuruutta verrataan olion kenttien arvon mukaan. [5, Luku 15]

3.6.3 Singleton-oliot

Useat muut oliokielet tarjoavat mahdollisuuden kirjoittaa staattisia luokkia tai staattisia kenttiä luokkiin. Scalassa tälläistä staattisuutta ei ole, vaan vastaava toiminnallisuus mahdollistetaan erityisien *singleton*-olioiden kautta. Singleton-oliosta on ajon aikana olemassa vain yksi instanssi, ja Scala luo sen automaattisesti. Viittaaminen olioon tapahtuu yksinkertaisesti olion nimellä. [4, Singleton objects]

Singleton-olioita voidaan kutsua myös kumppaniolioiksi jos se on määritelty saman nimisen luokan yhteydessä. Kumppanioliot näkevät ilmentymiensä yksityiset kentät, ja ilmentymät näkevät kumppaniolionsa yksityiset kentät. Luvun 3.6.2 Phone-luokalle voidaan luoda kumppaniolio seuraavalla tavalla:

```
object Phone {
    def iphone11 = new Phone("Apple", "iPhone 11")
}
val iphone = Phone.iphone11
```

Tässä tapauksessa kumppanioliossa on vain yksi metodi, jonka avulla voidaan luoda helposti instansseja. [5, Luku 4]

3.6.4 Piirretyypit

Piirretyyppejä käytetään Scalassa määrittelemään olioiden tarjoamien palveluiden rajapintoja ja mahdollistamaan ohjelmakoodin uudelleenkäytettävyyttä. Uusi piirretyyppi luodaan **`trait`**-avainsanalla. Piirretyyppien on myös sallittua periä toisia piirretyyppejä tai luokkia. Myös singleton-olioihin on mahdollista liittää piirretyyppejä. Piirretyypin voi ajatella olevan Javan rajapintaluokan ja abstraktin luokan yhdistelmä. [5, Luku 6 ja 12]

Piiretyypin jäseniksi voidaan määritellä metodeita tai muuttujia. Jäsenet voivat olla abstrakteja, tai konkreetteja. Abstraktille jäsenelle tulee toteuttavan luokan tarjota toteutus, kun taas konkreettielle jäsenille ei. Yhdessä piireluokassa voi olla molempia abstrakteja ja konkreetteja jäseniä. Tietyissä erityistapauksissa metodi voidaan merkitä abstraktiksi, vaikka sille on annettu toteutus piirretyypissä. [7, Luku 10]

Piirretyyppien määrittelemiä rajapintoja voidaan jaotella karkeasti kahteen luokkaan *ohuisiin* ja *rikkaisiin*. Ohuet rajapinnat määrittelevät vain vähän metodeita, kun taas rikkaat rajapinnat määrittelevät lukuisia metodeita. Ohuet rajapinnat ovat käytännöllisiä luokkaa toteutettaessa, sillä ohjelmoijan tarvitsee tehdä toteutus vain muutamalle metodille, mutta samasta syystä rajapinnan käyttäjälle ohut rajapinta ei ole ideaali. Piirreluokat mahdollistavat Scalassa rikkaiden rajapintojen tekemisen, toteuttamalla vain pienen määrän metodeita. Seuraavassa esimerkki rikkaasta `Ordered`-piirreluokasta, jonka avulla tyypin `T`-luokan instansseille voidaan määritellä suuruusjärjestys.

```
trait Ordered[T] {
    def compare(that: T): Int
    def <(that: T): Boolean = (this compare that) < 0
    def >(that: T): Boolean = (this compare that) > 0
    def <=(that: T): Boolean = (this compare that) <= 0
    def >=(that: T): Boolean = (this compare that) >= 0
}
```

`Ordered`-piirretyyppi siis määrittelee yhden abstraktin metodin `compare`, jonka toteuttamalla saadaan käyttöön loput neljä suurruutta vertailevista metodeista. [5, Luku 12]

3.6.5 Arvoluokat

Tietyissä tilanteissa Scala mahdollistaa erityisten *arvoluokkien* käyttämisen. Arvoluokkia käytetään yleensä kääreinä primitiviarvoille, ja siitä syystä tälläisilla luokilla on vain yksi jäsenmuuttuja. Arvoluokka voi sisältää vain yhden muuttujan ja lukemattoman määrän metodeita. Useimmissa tapauksissa arvoluokka käännetään primitiivarvoksi, jonka luokka pitää sisällään. Arvoluokkien tulee periä `AnyVal`, joka tarkoittaa ettei muuttujaan voi sijoittaa tyhjää arvoa.

Arvoluokat takaavat myös tyyppiturvallisuuden. Esimerkiksi monesti kuvaavia suureita saatetaan tallentaa `Double` -tyyppiseen muuttujaan. Tällöin muuttujan tyyppi ei kuitenkaan kerro mitään pituuden yksiköstä ja on mahdollista sekoittaa eri yksiköiden arvoja keskenään. Seuraavassa esimerkissä ongelma ratkaistaan määrittelemällä yksiköille

omat arvoluokkansa.

```
case class Meter(v: Double) extends AnyVal {
    def +(that: Meter): Meter = Meter(this.v + that.v)
}

case class Inch(v: Double) extends AnyVal {
    def +(that: Inch): Inch = Inch(this.v + that.v)
}

val m1 = Meter(3.1)
val m2 = Meter(3.2)
val i = Inch(2.2)

val res1 = m1 + m2  // Toimii
val res2 = m1 + i   // Virhe!
```

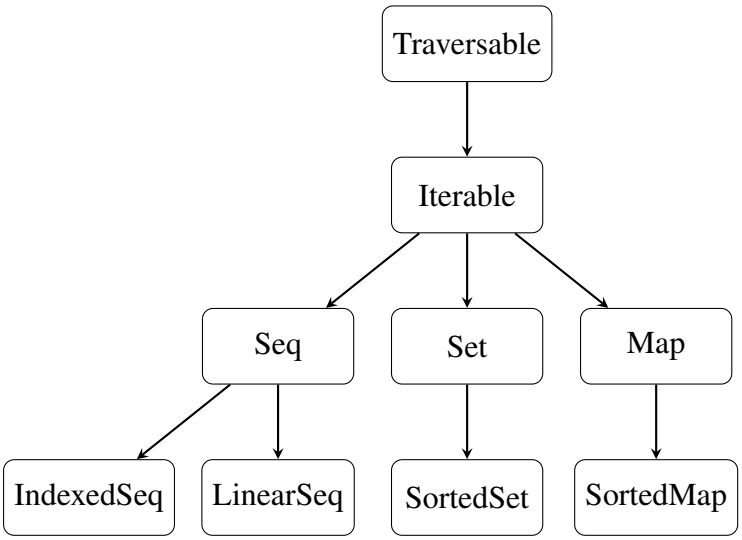
Yllä olevassa esimerkissä määritellään omat arvoluokkansa metrille ja tuumalle. Molemmille luokille määritlelään lisäksi +-metodi kahden instanssin yhteenlaskemista varten. Jos kuitenkin yksiköitä pyritään lisäämään toisiinsa, huomauttaa kääntäjä epäsopivista tyypeistä. [5, Luku 11]

4 Kokoelmat

Tutkimuksen kohteena on Scalan standardikirjaston kokoelmat versiosta 2.8 versioon 2.12. Standardikirjastossa on toteutus useimmille yleisimmille kokoelmaluokille kuten taulukoille, listoille, joukoille(`Set`) ja assosiaatiotauluille(`Map`). Scalan kokoelmaluokat ovat paketissa `scala.collection`. Muuttumattomat kokoelmat ovat alipaketissa `immutable` ja muuttuvat alipaketissa `mutable`. [8]

Muuttumattoman kokoelman sisältämät alkiot eivät voi muuttua alustamisen jälkeen. Tämä tarkoittaa ettei alkioiden lisääminen, poistaminen tai uudelleenjärjestäminen ole mahdollista. Kokoelman muuttamista muistuttavat operaatiot, kuten `map`, `reverse`, `fold` ja kokoelmien yhdistäminen, palauttavat aina uuden kokoelman jättäen alkuperäisen kokoelman ennalleen. Käytännössä kuitenkaan aina ei kaikkia muuttumattoman kokoelman alkioita ei tarvitse kopioida, vaan tietorakenteet pyrkivät käyttämään hyväkseen rakenteellista jakamista parantaakseen suorituskykyä ja optimoidakseen muistinkäyttöä. Muuttuvissa kokoelmissa on nimensä mukaisesti mahdollista vaihtaa alkioiden järjestystä, lisätä ja poistaa alkioita kokoelmasta luomatta uutta kokoelmaa. [8] [5, Luku 22]

Kokoelmaluokat noudattavat piirreluokkien määrittelemää hierarkiaa, joka on paketissa `scala.collection`. Hierarkia on sama sekä muuttumattomille että muuttuville kokoelmille. Ylimpänä hierarkiassa on `Traverable`-piirreluokka, jolla on yksi välitön aliluokka, `Iterable`. `Iterable`:lla on kolme välitöntä aliluokkaa `Seq`, `Set` ja `Map`, jotka edustavat järjestyksellisiä kokoelmia, joukkoja ja assosiaatiotauluja. Kuvassa 4.1 näytetään lisäksi vielä muutama aliluokka. [8]



Kuva 4.1: Kokoelmien hierarkia

4.1 Järjestykselliset kokoelmat

TODO: Etsi parempi ilmaus järjestykselliselle Suorituskykyvertailuun valittiin muuttumattomista kokoelmista `List`, joka tyypillinen rekursiivinen linkitetty rakenne funktionaaliessa ohjelmoinnissa, ja `Vector`, jossa alkion haku indeksin mukaan on tehokkaampaa. Muuttuvista kokoelmista valittiin `Array`, joka on kiinteän kokoinen taulukko, `ArrayBuffer`, joka on dynaamisesti kasvava taulukko ja `ListBuffer`, joka on dynaamisesti kasvava linkitetty rakenne. Kaikki kyseiset muuttuvat tietorakenteet ovat tyypillisiä olio-ohjelmoinnissa käytettyjä tietorakenteita.

`List` on abstrakti luokka, joka kuvaa yhteen suuntaan linkitettyä rekursiivista muuttumatonta listaa, jolla on kaksi konkreettista toteutusta: `::` ja `Nil`. Epätyhjää listaa kuvaavassa luokassa `::`, on kaksi jäsentä: alkio, jota kutsutaan listan *pääksi*, ja *häntä* joka kuvaa listan muita alkioita. Tyhjää listaa ja listan loppumista kuvaa singleton-olio `Nil`. Esimerkiksi luvut 1 ja 2 sisältävä lista voidaan ilmaista näin: `1 :: 2 :: Nil`. Listan `head`-ja `tail`-operaatiot sekä listan alkuun lisääminen tapahtuu vakioajassa. Aikakompleksisuus alkion hakemiselle indeksin perusteella on lineaarinen. [9] [8]

`Vector` on taulukon ominaisuuksia jäljittelevä muuttumaton tietorakenne, joka on

toteutettu puurakenteena, jonka solmut ovat korkeintaan 32-alkioisia taulukoita. Lehtisolmujen taulukot sisältävät vektorin varsinaisia alkioita, ja muut solmut sisältävät alemman tason taulukoita. Jos esimerkiksi alustetaan uusi 70-alkioinen vektori, on se kolmen taulukon puu jossa juurisolmuna olevalla taulukolla on kolme lapsitaulukkoa, joista kahdessa on 32 alkiota ja yhdessä 6 alkiota. Tämä mahdollistaa taulukoiden rakenteellisen jakamisen eri vektori-instanssien kesken, joka vähentää kopioimista ja muistinkäyttöä. Satunnaisten haku-, muokkaus- ja poisto-operaatioiden aikakompleksisuus on log(32, N), eli operaatioiden voidaan ajatella tapahtuvan lähes vakioajassa. [8] [10, Luku 4]

`Array` on Scalassa erityinen kokoelma, joka mahdollistaa primitiivityyppisten-, viittaustyyppisten- ja geneeristen alkioiden tallentamisen. Primitiivityyppisiä alkioita käsitellään ilman alkioiden käärimistä olioiksi. Esimerkiksi kokonaislukuja sisältävä `Array[Int]` käännetyy Javan `int[]`-taulukoksi. Kuten Javassa, taulukon koko ei voi muuttaa alustamisen jälkeen. Verrattuna Javan taulukoihin, `Array` tarjoaa kuitenkin huomattavasti enemmän operaatioita. Haku-ja muokkausoperaatiot tapahtuvat vakioajassa. Häntäoperaatio vie kuitenkin lineaarisen ajan, sillä jokainen alkio, poislukien ensimmäinen, joudutaan kopioimaan uuteen taulukkoon. [8]

`ArrayBuffer` on dynaamisesti kasvava listarakenne, joka käyttää alkioiden tallentamiseen taulukkoa. Tästä johtuen useimmat operaatiot vastaavat aikakompleksisuudeltaan `Array`:ta. Lisäksi `ArrayBuffer` mahdollistaa alkion lisäämisen taulukon alkuun, loppuun sekä keskelle. Alkuun ja keskelle lisäämisen aikakompleksisuus on lineaarinen. Muut operaatiot tapahtuvat pääasiassa vakioajassa, poislukien täyteen taulukkoon lisääminen, joka vie lineaarisen ajan. [8]

`ListBuffer` on dynaamisesti kasvava muuttuva kokoelma joka on toteutettu linkitettynä rakenteena. Poiketen `List`:stä, `ListBuffer` mahdollistaa alkioiden lisäämisen rakenteen alkuun ja loppuun vakioajassa. Satunnaisen alkion haku-ja muokkausoperaatiot, sekä rakenteen keskelle lisääminen ovat aikakompleksisuudeltaan lineaarisia. Myös häntäoperaatio vie lineaarisen ajan, sillä kaikki hännän alkiot täytyy kopioida uuteen tie-

torakenteeseen. [8]

4.2 Joukot ja assosiaatiotaulut

TODO: Tarkastellaanko ollenkaan?

4.3 Suorituskykyvertailut

Tutkielmassa on käytetty kokoelmien suorituskykymittauksia on kahdesta eri lähteestä: Li Haoyi:n[11] ja Toby Hobsonin[12] mittauksista. Haoyin mittauksissa mitataan aikaa, joka operaatioiden suorittamisessa kestää, eli pienempi arvo tarkoittaa parempaa suorituskykyä. Hobsonin mittauksissa tarkastellaan operaatioiden määrää sekunnissa, jolloin suuri luku takoirttaa parempaa suorituskykyä. Hayoin mittauksissa suorituskykyä on mitattu eri kokoisilla kokoelmilla. Kaavioihin on valittu kyseisen operaation kannalta mielekkäät koot kokoelmille.

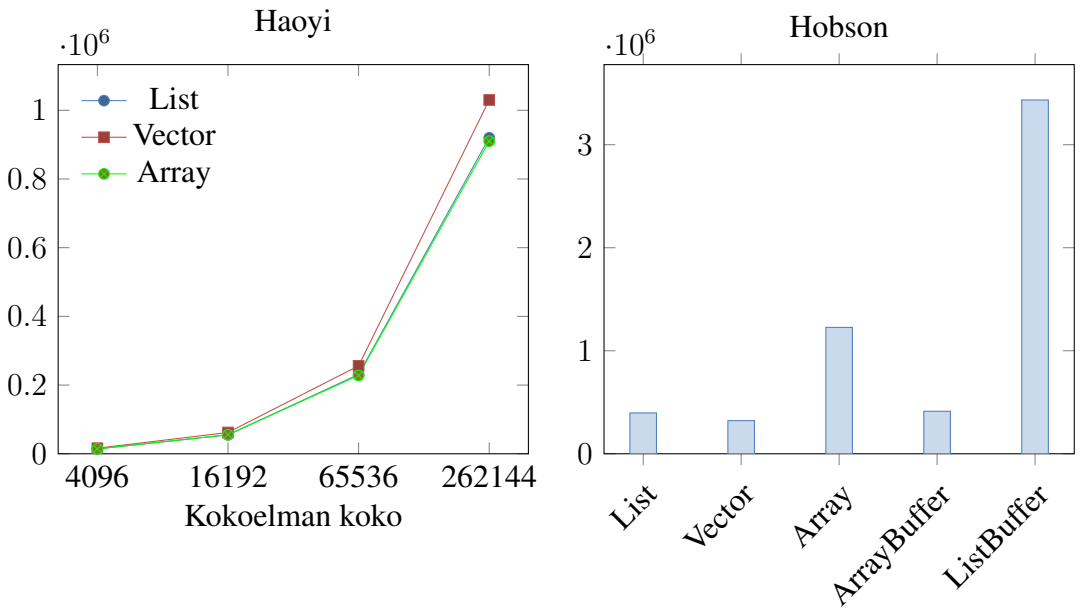
Järjestyksellisten kokoelmien tapauksessa kiinnostuksen kohteena on yleisimpien operaatioiden suorituskyky: kokoelman luominen, iterointi, alkion satunnainen haku sekä lisääminen ja häntäoperaatio.

Imperatiivisissa kielissä iterointia tehdään yleensä silmukoiden, tavallisesti **`for`** ja **`while`**, avulla. Funktionaalinen paradigma ei kuitenkaan tunne silmukoita ja iterointia tehdään tavallisesti korkeamman asteen funktioiden, kuten `map` ja `foreach`, avulla. [5, Luku 2] Seuraavaksi esitellään molemmat tavat tulostamalla listan parilliset luvut kahdella kerrottuna:

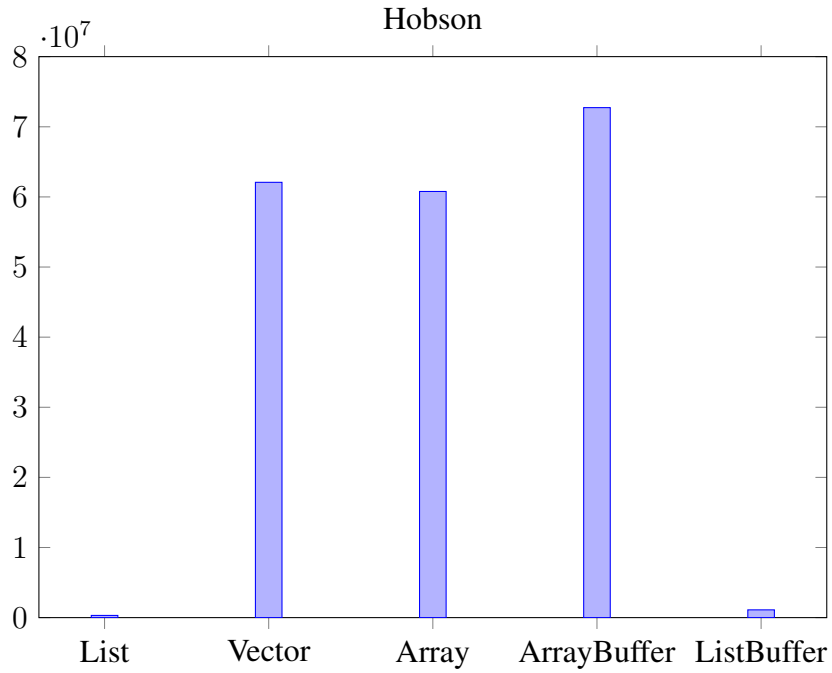
TODO: Laiskat kokoelmat?

```
val list = List.range(1, 11)

// Imperatiivisesti
for(i <- list)
    if (i % 2 == 0) println(i*2)

// Funktionaalisesti
list.filter(_ % 2 == 0).map(_ * 2).foreach(println)
```

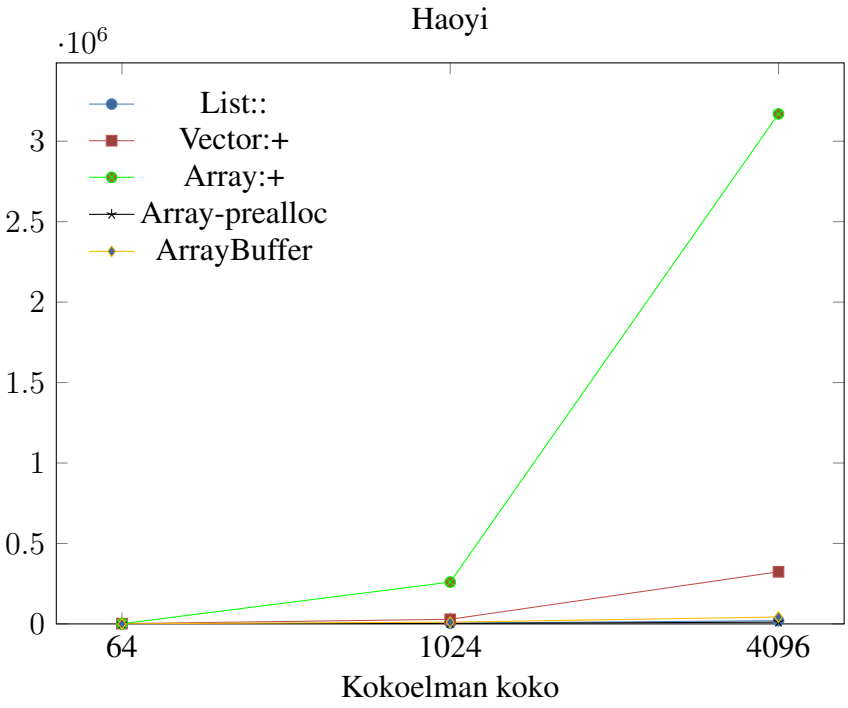
Kuva 4.2: Iterointi imperatiivisesti ja funktionaalisesti



Kuva 4.3: Iterointi

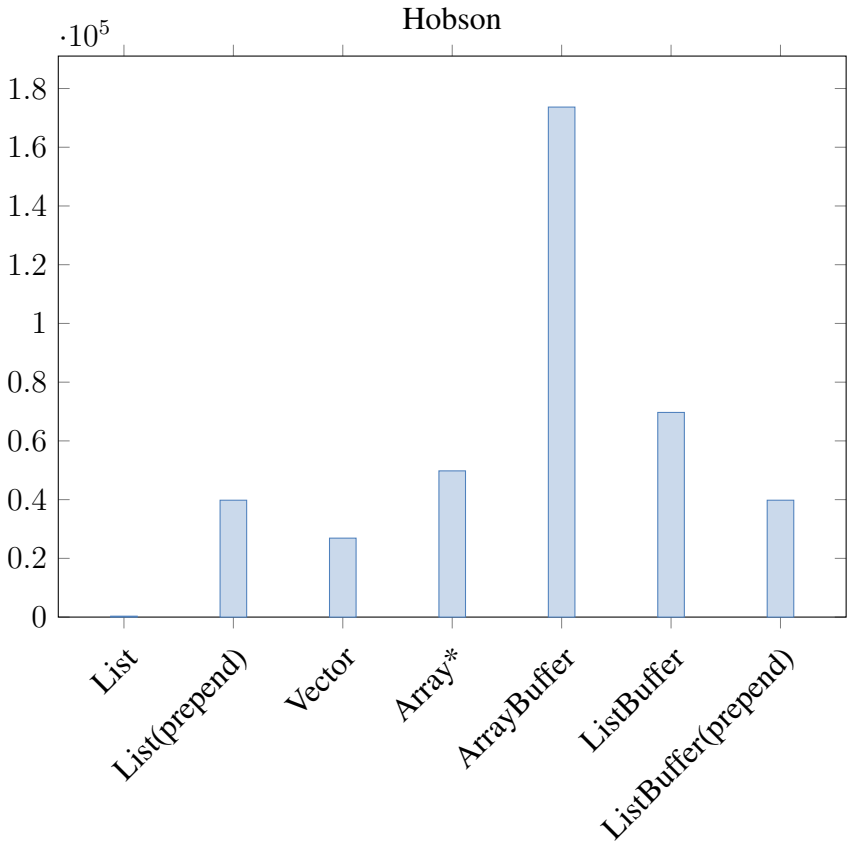


Kuva 4.4: Satunnainen haku

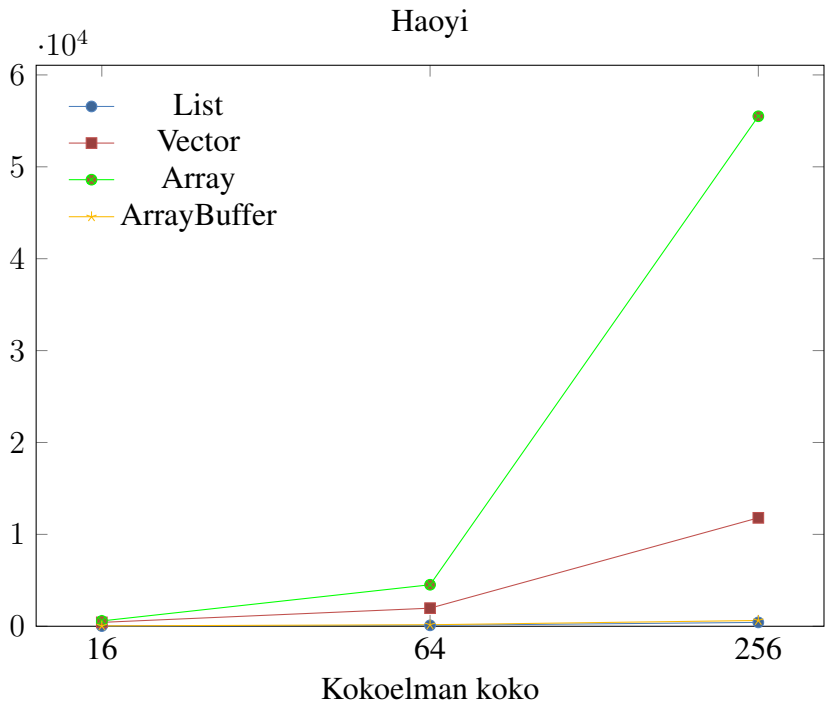


Kuva 4.5: Kokoelman luominen

[draft]



Kuva 4.6: Kokoelman loppuun lisääminen



Kuva 4.7: Häntäoperaatio

5 Yhteenveto

Lähdeluettelo

[1] M. Gabbrielli ja S. Martini, *Programming Languages: Principles and Paradigms*. Springer, 2010.

[2] M. L. Scott, *Programming Language Pragmatics, 4th Edition*. Morgan Kaufmann, 2015.

[3] P. Chiusano ja R. Bjarnason, *Functional Programming in Scala*. Manning, 2015.

[4] (2019). Tour of Scala, url: `https://docs.scala-lang.org/tour/tour-of-scala.html` (viitattu 03. 01. 2020).

[5] M. Odersky, L. Spoon ja B. Venners, *Programming in Scala, 3rd Edition*. Artima, 2016.

[6] (2019). Origins of Scala, url: `https://www.artima.com/scalazine/articles/origins_of_scala.html` (viitattu 03. 01. 2020).

[7] C. Horstmann, *Scala for the Impatient, 2nd Edition*. Addison-Wesley Professional, 2017.

[8] (2017). Scala Collections v2.8-2.12, url: `https://docs.scala-lang.org/overviews/collections/` (viitattu 12. 02. 2020).

[9] (2019). Scala API v2.12, url: `https://www.scala-lang.org/files/archive/api/2.12.x` (viitattu 03. 01. 2020).

[10] V. Theron ja M. Diamant, *Scala High Performance Programming*. Packt, 2016.

[11] L. Hayoi. (2016). Benchmarking Scala Collections, url: http://www.lihaoyi.com/post/BenchmarkingScalaCollections.html (viitattu 18. 02. 2020).

[12] T. Hobson. (2016). Benchmarking Scala Collections, url: https://www.tobyhobson.com/posts/scala/collections-performance/ (viitattu 18. 02. 2020).