
Funktionaalisen- ja olioparadigman suorituskykyvertailu Scala-kielessä

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

Sisältö

1	Johdanto	1
2	Ohjelmointiparadigmat	2
2.1	Olio-paradigma	2
2.2	Funktionaalinen paradigma	3
2.3	Paradigmoille tyypillisiä ominaisuuksia	3
3	Scala	5
3.1	Kielen synty	5
3.2	Muuttujat	6
3.3	Tietotyytit	7
3.4	Kontrollirakenteet	8
3.4.1	Ehtolauseet	8
3.4.2	Silmukat	8
3.5	Metodit ja funktiot	9
3.5.1	Funktioliteraalit	9
3.5.2	Korkeamman asteen funktiot	10
3.6	Luokat ja oliot	10
3.6.1	Class	11
3.6.2	Case class	11
3.6.3	Singleton-oliot	12

3.6.4	Piirretyypit	12
3.6.5	Arvoluokat	13

Lähdeluettelo	15
----------------------	-----------

1 Johdanto

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. Tällaista 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-ohjelmoinnin 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

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 deklaratiiiviseen paradigmaan. [1, Luku 11]

2.3 Paradigmoille tyypillisiä 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 parametrikksi 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: imperatiivisissa kielissä niiden muuttaminen alustamisen jälkeen on sallittua, funktionaalisissa ei. [3, Luku 3] Funktionaalisissa kielissä funk-

tioita 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 määriteltä määrää, tai kunnes silmukan suoritusta säätelevä ehtolause muuttuu epätodeksi. Ehtolauseen arvo määräytyy muuttujan arvon mukaan, jota muutetaan silmukan suorituksen aikana. Funktionaalisisissa kielissä muuttujien arvoa ei voi muuttaa, ja siitä syystä silmukat olisivat hyödyttömiä. Silmukoiden sijaan käytetään rekursiota, eli funktio kutsuu itseään kunnes funktioon määriteltä perustapaus saavutetaan. [1, Luku 6 ja 11]

3 Scala

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. Scalan viimeisin vakaa versio 2.13.1 julkaistiin vuoden 2019 syyskuussa ja versio 3.0:n arvioitu julkaisu on vuodenvaihtessa 2020-2021. [4, Introduction] [5]

Tutkielman kieleksi valittiin Scala, koska se on suunniteltu tukemaan funktionaalista ja olioparadigmaa, jolloin suorituskyvyn vertailu näiden kahden paradigman välillä on mielekästä ja suoraviivaista. Scala-koodi on tarkoitettu käännettäväksi Javan tavukoodiksi, ja sitä ajetaan Javan virtuaalikoneessa (JVM). Scalaa voidaan kääntää myös javascriptiksi tai natiiviksi koodiksi. Tässä tutkielmassa keskittytään ainoastaan JVM:ssä suoritettavaan Scalaan. [6, Luku 2] [7] [8]

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-

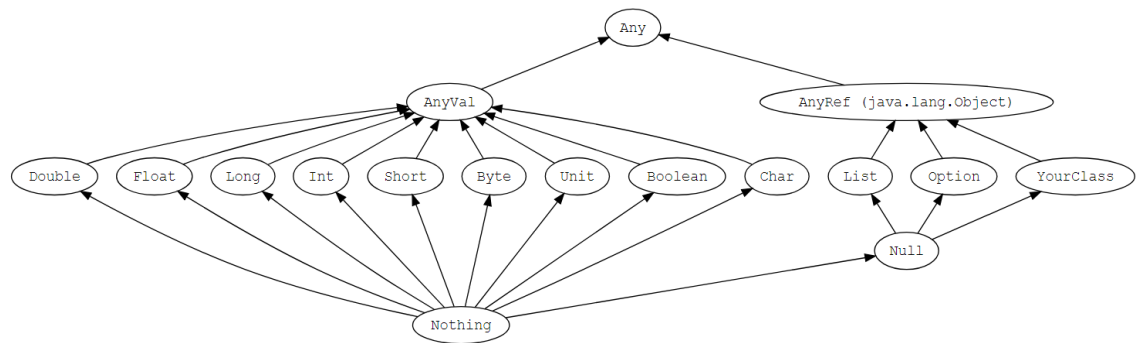
projektin (GJ), jonka tarkoituksena oli tuoda geneerisyys Javaan. Odersky kehitti projektin aikana kääntäjän, joka korvasi Sun Microsystemsin kehittämän Java-kääntäjän versiossa 1.3 vuonna 2000. Lopulta Generic Java-projektin aikana kehitetty geneerisyys liitettiin Javaan versiossa 5 vuonna 2004. [9]

Generic Java-projektin aikana Odersky koki Javan taaksepäin yhteensopivuuden rajoittavan 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. [9]

3.2 Muuttujat

Scalassa on tavallisten muuttujien lisäksi vakioita, jotka ovat muuttujia joiden arvoa ei alustuksen jälkeen voi muuttaa. Vakion määrittelyyn käytetään avainsanaa **val** ja muuttujan määrittelyyn **var**. Muuttujan nimen jälkeen määritellään muuttujan tyyppi, joka erotetaan muuttujan nimestä 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 tyyipitettyjä, 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. [6, Luku 1]



Kuva 3.1: Scalan luokkahierarkia

3.3 Tietotyypit

Scala on puhtaasti oliokieli, ja kaikki muuttujat ovat olioita. Jokainen Scalan tietotyyppi perii 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 tyypialias Javan luokalle `Object`. Jokaisen viiteluokan alaluokka on `Null`. Tietotyyppien hierarkia on kuvattu kuvassa 3.1. [6, Luku 5]

Alkeistietotyyppinä 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 tyypialias Javan merkkijonoa kuvaavalle tietotyyppille. [6, 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. [6, 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-lausekkeella 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. [10, 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. [10, 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, suodattetaan parilliset arvot, palautetaan jäljelle jääneet arvot kahdella kerrottuna ja asetetaan palautettu vektori muuttujaan *res*. [10, 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 funktioliteraalksi. Metodi voidaan määritellä **def**-avainsanalla. [10, 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 muil-

lakin muuttujilla, funktioilla ja metodeilla on Scalassa aina tyyppi. Tyypin voi määritellä eksplisiittisesti, mutta monesti Scala osaa päätellä funktion tyypin. Seuraavassa esimerkissä määritellään kummallakin tavolla funktio, joka kertoo onko parametrina annettu kokonaisluku parillinen. Funktion tyyppi on siis `Int => Boolean`. [6, 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. Tällaisia 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 kokonaislukuun, 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 funktioliteralille `x => x * 3`. [10, 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 su-luissa. 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 muuttumattonta dataa. Case-luokka määritellään kuten tavallinen luokka, mutta konstruktorissa määriteltävät kentät ovat oletuksena **public val**. Case-luokkaan voidaan määritellä 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 *kumppanioliioon* `apply`-tehdasmetodin, jonka avulla uusien instanssien luonti tapahtuu. [6, Luku 15]

Tavallisia luokkia käytetään varsinkin olio-tyylisessä ohjelmoinnissa, kun taas case-luokat ovat käytössä erityisesti funktionaalisessa Scalassa, kun halutaan käyttää muuttu-

mattomia 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. [6, Luku 15]

3.6.3 Singleton-oliot

Useat muut oliokielet tarjoavat mahdollisuuden kirjoittaa staattisia luokkia tai staattisia kenttiä luokkiin. Scalassa tällaista 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. [6, Luku 4]

3.6.4 Piirrettyypit

Piirrettyyppejä käytetään Scalassa määrittelemään olioiden tarjoamien palveluiden rajapintoja ja mahdollistamaan ohjelmakoodin uudelleenkäytettävyyttä. Uusi piirrettyyppi luodaan **trait**-avainsanalla. Piirrettyyppien on myös sallittua periä toisia piirrettyyppejä tai luokkia. Myös singleton-oloihin on mahdollista liittää piirrettyyppejä. Piirrettyypin voi ajatella olevan Javan rajapintaluokan ja abstraktin luokan yhdistelmä. [6, Luku 6 ja 12]

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

Piiretyyppien 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. Piireluokat mahdollistavat Scalassa rikkaiden rajapintojen tekemisen, toteuttamalla vain pienen määrän metodeita. Seuraavassa esimerkki rikkaasta `Ordered`-piireluokasta, jonka avulla tyyppin `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`-piiretystyyppi siis määrittelee yhden abstraktin metodin `compare`, jonka toteuttamalla saadaan käyttöön loput neljä suurruutta vertailevista metodeista. [6, Luku 12]

3.6.5 Arvoluokat

Tietyissä tilanteissa Scala mahdollistaa erityisten *arvoluokkien* käyttämisen. Arvoluokkia käytetään yleensä kääreinä primitiivarvoille, ja siitä syystä tällaisilla 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 määritellään lisäksi `+`-metodi kahden instanssin yhteenlaskemista varten. Jos kuitenkin yksiköitä pyritään lisäämään toisiinsa, huomauttaa kääntäjä epäsopivista tyypeistä. [6, Luku 11]

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] (2019). Scala 2 Roadmap Update: the Road to Scala 3, url: <https://www.scala-lang.org/2019/12/18/road-to-scala-3.html#when-will-scala-30-be-released> (viitattu 03.01.2020).
- [6] M. Odersky, L. Spoon ja B. Venners, *Programming in Scala, 3rd Edition*. Artima, 2016.
- [7] (2019). Scala.js, url: <https://www.scala-js.org/> (viitattu 03.01.2020).
- [8] (2019). Scala Native, url: <https://scala-native.readthedocs.io/en/latest/> (viitattu 03.01.2020).
- [9] (2019). Origins of Scala, url: https://www.artima.com/scalazine/articles/origins_of_scala.html (viitattu 03.01.2020).
- [10] C. Horstmann, *Scala for the Impatient, 2nd Edition*. Addison-Wesley Professional, 2017.