Jarno Sundström

Java funktionaalisessa ohjelmoinnissa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

27.10.2021

Tiivistelmä

Tekijä:	Jarno Sundström
Otsikko:	Java funktionaalisessa ohjelmoinnissa
Sivumäärä:	49 sivua
Aika:	27.10.2021
Tutkinto:	insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Lehtori Simo Silander

Tämän työn tavoitteena on käsitellä funktionaalista ohjelmointia Java-ohjelmoijan näkökulmasta. Työssä käsitellään Javan funktionaalisia piirteitä sekä sitä, miten niitä voi hyödyntää. Tavoitteena on myös selvittää, miten nämä piirteet kulkevat rinnakkain olio-ohjelmoinnin yhteydessä. Työn toisena tavoitteena on, että olio-ohjelmoija saa hyvää ymmärrystä, ohjeita sekä informaatiota siihen, kuinka hyödyntää Javan funktionaalisia piirteitä.

Työssä käydään aluksi läpi funktionaalisen ohjelmoinnin tärkeimmät ominaisuudet ja säännöt kuten lambda-lauseke, datan muuttumattomuus, funktiot ensimmäisen luokan kansalaisina sekä rekursio. Myöhemmin työssä käsitellään Java 8:n mukana tulleita funktionaalisia rajapintoja, niiden metodeja sekä sitä, kuinka niitä voidaan hyödyntää sekä teoriassa että esimerkkien avulla. Lisäksi työssä käsitellään streamit ja niiden yhteydessä käytetyt metodit map(), filter() ja reduce().

Viimeisenä toteutetaan ohjelma hyödyntäen työssä läpikäytyjä tekniikoita ja todetaan, mitä etuja funktionaalisesta ohjelmointityylistä on verrattuna perinteiseen olio-ohjelmointiin. Todetaan, että Javalla pystytään toteuttamaan funktionaalista ohjelmointia, vaikka kyseessä ei ole puhtaasti funktionaalinen kieli.

Abstract

Author:	Jarno Sundström
Title:	Java in Functional Programming
Number of Pages:	49 pages
Date:	27 October 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communications Technology
Professional Major:	Software Engineering
Supervisors:	Simo Silander, Senior Lecturer

The purpose of this thesis is to investigate functional programming from the perspective of a Java programmer. The thesis goes through the previously mentioned language, its qualities in functional programming and how to make use of them. Another purpose for the study was to find out how these qualities go hand in hand with object-oriented programming, providing an object-oriented programmer the possibility to acquire a good understanding, guidance and information on how to use functional qualities in Java.

The thesis starts by going through the most important features of functional programming such as the Lambda-statement, immutable data, functions as first-class citizens and recursion. Later the thesis goes through interfaces brought along by Java 8. The thesis demonstrates how these features can be used by going through theory and using examples. Furthermore, the thesis explains streams and its most used methods map(), filter() and reduce().

Lastly, the thesis introduces an implementation of software where functional programming techniques are used. The paper concludes on what benefits functional programming has over the traditional object-oriented programming paradigm. The conclusion is that Java can be used for functional programming even though it is not a purely functional programming language.

Sisällys

1 Johdanto

Tämän opinnäytetyön tavoitteena on saada kattava käsitys siitä, miten Java toimii funktionaalisena ohjelmointikielenä. Tätä aihetta käsitellään olio-ohjelmoijan näkökulmasta.

Olio-ohjelmointi on ohjelmointiparadigma, jossa luodaan olioita esittämään erilaisia asioita, joista ohjelman halutaan koostuvan. Nämä oliot saattavat olla todellisessa maailmassa esiintyviä asioita tai abstraktioita. Olioiden attribuutteja muokataan ja hallinnoidaan metodien sekä funktioiden avulla. Olio-ohjelmoinnin ongelma on siinä, että ohjelmistojen kasvaessa niistä tulee vaikeasti ymmärrettäviä ja alttiita sivuvaikutuksille. Haluamme, että ohjelma, joka käsittelee isoja määriä tietoa, on luotettava eikä muokkaa tietoa prosessoinnin aikana.

Funktionaalinen ohjelmointi on tässä työssä käsiteltävä ohjelmointiparadigma. Perinteisessä olio-ohjelmoinnissa kerromme ohjelmalle, miten tietoa tulee käsitellä, kun taas funktionaalisessa ohjelmoinnissa keskitytään ainoastaan siihen, mitä halutaan tehdä. Ohjelmoijan ei tarvitse enää rakentaa silmukoita ja muokata olioiden sisältämää tietoa vaan käytetään valmiita funktioita tiedon käsittelyyn. Funktiot ketjutetaan oikeassa järjestyksessä halutun lopputuloksen saavuttamiseksi.

Lopuksi työn tavoitteena on selvittää, miten olio-ohjelmoija voisi ottaa funktionaalisen ohjelmoinnin työkaluksi olio-ohjelmoinnin rinnalle hyödyntämisesimerkin avulla.

2 Funktionaalinen ohjelmointi

Funktionaalinen ohjelmointi on deklaratiivinen ohjelmointiparadigma, jota on aika ajoin virheellisesti pidetty olio-ohjelmoinnin vastakohtana. Deklaratiivisuudella tarkoitetaan pyrkimystä ilmaista ohjelman logiikkaa ja lopputuloksia. Imperatiivinen ohjelmointi sen sijaan kuvaa ohjelman etenemistä askel kerrallaan. Imperatiivisiin ohjelmointiparadigmoihin kuuluu muun muassa olio-ohjelmointi. Useimmissa ohjelmistoissa käytetään kuitenkin molempia paradigmoja. Funktionaalinen ohjelmointi tarjoaa selkeitä etuja joissakin tapauksissa ja sitä käytetään monissa teknologioissa. Funktionaalinen ohjelmointi perustuu lisäksi rekursioon eli funktio kutsuu itse itseään, kunnes sille asetettu lopetusehto täyttyy.

Funktionaalinen ohjelmointi perustuu lambda-kalkyylin (tunnetaan myös λ-kalkyylinä). Lambda-kalkyylin kehitti Alonzo Church 1930-luvulla vastatakseen kysymykseen ”voidaanko kaikki yhdellä universaalilla kielellä muodostetut ongelmat ratkaista?”. [1.]

Tietyt teknologiat on suunniteltu alun perin funktionaaliseen ohjelmointiin sopiviksi. Tällaisia ovat esimerkiksi Clojure, Elixir, Elm, F#, Haskell, Idris ja Scala.

Näistä teknologioista Haskell on tätä työtä kirjoitettaessa suosituin isoissa yrityksissä. Näin ollen se on yksi parhaista kielistä opetella, mikäli haluaa työskennellä funktionaalisen ohjelmoinnin parissa.

2.1 Lambda-lauseke

Kuten aiemmin on mainittu, funktionaalinen ohjelmointi perustuu lambda-kalkyyliin. Se on yksinkertainen mutta tehokas matemaattinen systeemi lambda-lausekkeen muokkaamiseksi. Kaikki pohjautuu funktion abstraktioon. Kaikki Javan funk-

tionaalisuus perustuu lambda-lausekkeeseen, sillä se mahdollistaa funktion syöttämisen toisen funktion parametreihin. Lambda-lausekkeesta tulee siis funktion argumentti.

Lambda-lauseke on anonyymi funktio, joka tarjoaa lyhennetyn tavan kirjoittaa funktionaalista koodia, jota käytetään anonyymien metodien luomiseen. Funktiossa käsiteltävät syöteparametrit (input parameters) tulevat vasemmalle puolelle lambda-operaattoria ja funktiolauseke oikealle puolelle.[2]

Lambda-lauseke koostuu seuraavista osista:

- Pilkuin eroteltu lista parametreistä on sulkeiden sisällä
- Lambda ilmaistaan Javassa nuolimerkillä (”->”)
- Body-osio sisältää yhden lausekkeen (expression) tai lauseke-osion (statement block).

Yksinkertaisimmillaan lambda-lauseke näyttää seuraavalta Javassa:

```
parametri -> lauseke
```

Havainnollistetaan lambda-lauseketta yksinkertaisella funktiolla, joka kasvattaa funktioon syötettyä arvoa yhdellä. Esimerkissä on käytössä Java:

```
numero -> numero + 1;
```

Yllä olevasta esimerkistä havaitaan, että argumentti sijoitetaan lambda-merkin vasemmalle puolelle. Oikealle puolelle sijoitetaan body-osio.

2.2 Korkeamman tason funktiot

Korkeamman tason funktio (Higher Order Function) on funktio, joka hyväksyy muita funktioita argumentiksi tai palauttaa funktion. Eric Elliotin mukaan modu-

laarisuus, joka korostuu korkeamman tason funktioissa, on toimivan ja tehokkaan ohjelmoinnin kulmakivi. Yleinen käyttötapa korkeamman tason funktioille on tietorakenteiden käsittely. Kolme käytetyintä korkeamman tason funktiota ovat map, filter ja reduce. Nämä on toteutettu myös ei-funktionaalisissa kielissä kuten Javassa ja Pythonissa. Myöhemmin luvuissa 3.2.6 käsitellään näitä funktioita ja niiden käyttöä.[3]

2.3 Funktiot ensimmäisen luokan kansalaisina

Mikäli ohjelmointikielellä on kyky käsitellä funktiota arvona, syötteinä tai palautusarvona toisesta funktiosta, niin kyseisellä kielellä on ominaisuutena ensimmäisen luokan funktiot. Nämä funktiot tunnetaan ensimmäisen luokan kansalaisina kyseisessä ohjelmointikielessä.[4.]

”::”-operaattori on lyhyempi tapa lambda-lausekkeille kutsua erilaisia metodeja. Käyttäessämme metodiviittauksia (method reference) kohdeviittaus (target reference) asetetaan ensin ”::”-operaattorin eteen ja metodin nimi jälkimmäiseksi. Havainnollistetaan edellä mainittua esimerkillä:

```
public static void main(String[] args) {
        //Luodaan lista ja lisätään siihen kokonaislukuja
        List<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        //Tehdään listasta vuo ja tulostetaan jokainen elementti kon-
soliin
        list.stream().forEach(System.out::println);//1,2,3,4

        //Sama käyttämällä lambda-merkintää
        list.stream().forEach(s -> System.out.println(s));//1,2,3,4
    }
```

Esimerkkikoodi 1. ”::”-operaattoria hyödynnetään lyhentämällä vuon elementtien tulostamiseen vaadittavaa syntaksia.

2.4 Currying-tekniikka

Currying on tekniikka, jolla muutetaan funktio, joka ottaa sisäänsä monta argumenttia monen funktion sarjaksi, joista jokainen ottaa sisäänsä vain yhden argumentin. Kohdistetaan currying-tekniikka seuraavaan funktioon:

```
(x,y) -> z
```

Ennen käsittelyä funktio ottaa argumentiksi x:n sekä y:n.

```
x -> (y -> z)
```

Toimenpiteen jälkeen funktio ottaa argumentikseen ainoastaan x:n ja palauttaa funktion, joka ottaa parametrikseen y:n. Molemmat muodot kertovat meille yhtä paljon, ja kun annamme kaksi argumenttia kumpaan tahansa funktion muotoon, on lopputulos aina sama. Ero on, että currying-tekniikan avulla funktio sallii osittaiset sovellukset. Osittaisilla sovelluksilla tarkoitetaan sitä, että osa argumenteista on valmiiksi kiinnitetty funktioon. Esimerkiksi jos tehdään funktio, jonka käsittelyyn tarvitaan valmiiksi määritelty vakio (esim. arvonlisävero), niin vakio on kiinnitetty funktioon ja lisäksi rinnalle tuodaan itse määritettyjä argumentteja, jotka auttavat vaikkapa tuotteen hinnan määrittelyssä arvonlisäveron kanssa. Havainnollistetaan osittaisia sovelluksia esimerkillä.

```
public class OsittainenSovellus {
    public static <T, U, R> Function<U, R>
    osittainenSovellus(BiFunction<T, U, R> f, T x) {
        return (y) -> f.apply(x, y);
    }

    public static void main(String[] args) {
    double alv = 0.24;
    BiFunction<Double, Double, Double> kerro = (x,y) -> x*y;
    Function<Double,Double> haeAlv = osittainenSovellus(kerro,alv);

    System.out.println("Alv = " + haeAlv.apply(5.0) + " euroa");
}}
```

Esimerkkikoodi 2. Esimerkki osittaisesta sovelluksesta.

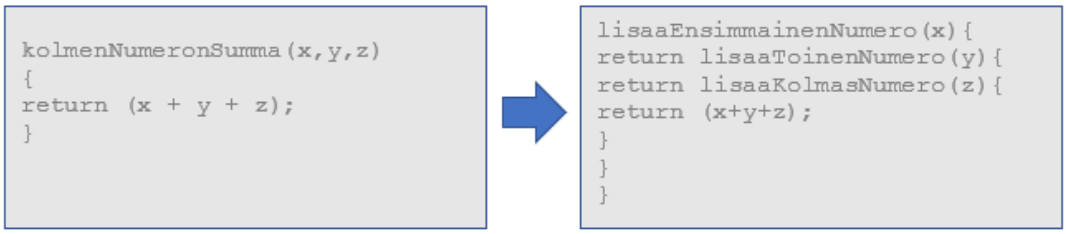
Kuten esimerkkikoodista 2 ilmenee, liukuluku alv kiinnitetään haeAlv-funktioon OsittainenSovellus-luokan osittainenSovellus-BiFunktion avulla. lopuksi käytetään haeAlv-funktiota tulostamaan arvonlisävero haluamastamme summasta.

Havainnollistetaan currying-tekniikkaa esimerkillä, jossa hyödynnetään Function-rajapintaa, jonka syntaksia ja metodeita käsitellään tarkemmin luvussa 3.2.4.

```
import java.util.function.Function;

public class CurryingTekniikka {
    public static void main(String[] args) {

        /*
          Funktio ottaa syötteenä kokonaisluvun ja palauttaa
          funktion,joka tekee saman kuin edellinen funktio. Viimeinen
          palautettava funktio ottaa syötteenään kokonaisluvun
          ja palauttaa kokonaisluvun.
        */
        Function<Integer,
                Function<Integer,
                        Function<Integer, Integer>>>
                laskeYhteen = x -> y -> z -> x + y + z;

        //Apply metodi selitetään luvussa 3.2.4
        System.out.println(laskeYhteen.apply(3).apply(4).apply(5));
    }
}
```

Esimerkkikoodi 3. Määritellään funktio, joka kutsuu toista funktiota, joka kutsuu kolmatta funktiota sarjassa. Tämä funktio olisi matemaattisesti esitettynä muotoa f(x,y,z) = f(x)(y)(z).

Kuten esimerkkikoodista 3 voidaan havaita, sen sijaan että määrittelisimme yhden funktion, joka ottaa kolme kokonaislukua argumenttina, määrittelimme funktiokutsusarjan, joka sisältää kolme erillistä funktiota.[5.] Tätä sekvenssimuutosta kutsutaan Currying-tekniikaksi. Havainnollistetaan tämän lisäksi currying-tekniikkaa esimerkkikuvalla:



Kuva 1. Currying-tekniikalla hajotetaan korkeamman tason funktio pienempiin osiin

2.5 Sivuvaikutukset

Sivuvaikutuksella ohjelmoinnissa tarkoitetaan tilan (state) muuttumista. Yksi funktionaalisen ohjelmoinnin tärkeimmistä perusperiaatteista on sivuvaikutusten välttäminen. Esimerkkejä sivuvaikutuksista ovat

- Muuttuja-arvon muokkaaminen
- datan kirjoittaminen levylle
- käyttöliittymässä olevan painikkeen aktivoiminen tai deaktivoiminen.

Yleinen harhaluulo sivuvaikutuksista on se, että ne ovat väkisin piilossa tai aiheutuvat vain vahingossa. Sivuvaikutuksia voi ilmetä, kun kutsutaan monimutkaista funktiota, joka muokkaa jotakin globaalia muuttujaa. Esimerkkikoodissa 4 havainnollistetaan funktiota, jonka sivuvaikutuksena staattinen luokkamuuttuja saa uuden arvon.

```
public class MuuttujaLuokka{
     public static int x;
}

public class MuuttujanKasittely{

     public int teeKalkulaatio(int x){
     if(x > 100){
          MuuttujaLuokka.x = x + 100;
     }else{
          x*=100;}
     }
          return x;
}}
```

Esimerkkikoodi 4. Esimerkissä luodaan staattinen luokkamuuttuja x, jonka jälkeen MuuttujanKasittely-luokan funktio teeKalkulaatio() saattaa muokata globaalin muuttujan arvoa riippuen funktioon syötetystä arvosta.

2.6 Puhtaat funktiot

Puhtailla funktioilla (pure functions) tarkoitetaan sellaisia funktioita jotka, (1) eivät aiheuta sivuvaikutuksia, (2) sen palautusarvo (return value) ei muutu sivuvaikutuksien myötä, (3) jos sille annetaan samat argumentit, se palauttaa samat arvot jokaisen evaluaation jälkeen. Esimerkkejä puhtaista funktioita ovat matemaattiset funktiot ja sellaiset funktiot, jotka palauttavat esimerkiksi String-muuttujapituuden tai listan pituuden (lista.size() lista.length()). Mikä tahansa funktio, joka käsittelee staattista luokkamuuttujaa, ei voi olla puhdas, sillä kyseisen muuttujan arvo voi muuttua ohjelman ajamisen aikana. Tämän seurauksena funktion palautusarvo voi muuttua evaluaatioiden välillä. Havainnollistetaan puhdasta funktiota esimerkillä:

```
public class Esimerkki{

     public int summa(int x, int y){
          return x + y;
     }
}
```

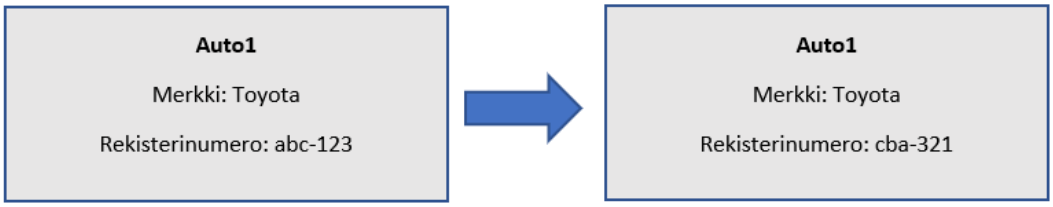
Esimerkkikoodi 5. Summa()-funktio ottaa sisäänsä kaksi kokonaislukua ja palauttaa niiden summan

Kuten esimerkkikoodi 5:stä ilmenee, funktion palautusarvo on riippuvainen ainoastaan siihen syötetyistä arvoista eikä se myöskään muokkaa mitään tilamuuttujia (state variable) funktion ulkopuolella.

2.7 Muuttumattomuus

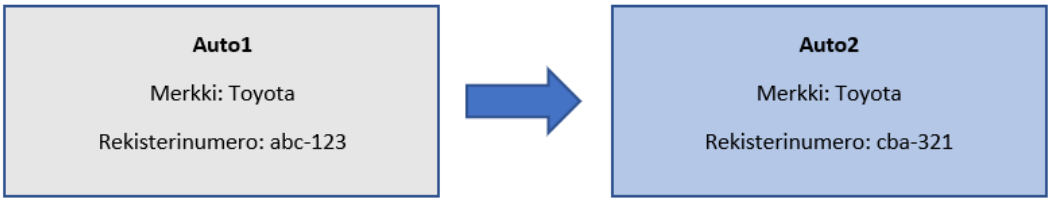
Puhtaasti funktionaaliset ohjelmointikielet kuten esimerkiksi Haskell työskentelevät yleisesti ottaen muuttumattoman (immutable) datan ympärillä. Muuttumattomalla datalla olio-ohjelmoinnissa tarkoitetaan sitä, että kun olio on luotu, se pysyy samassa tilassa kuin se oli luontihetkellään. Joissakin tapauksissa oliota voidaan pitää muuttumattomana, vaikka jokin sen sisäinen attribuutti muuttuisi, mutta olion tila vaikuttaa muuttumattomalta ulkoisesta näkökulmasta. Esimerkiksi String-oliot ovat muuttumattomia, jos ohjelmoija tekee merkkijonoon muutoksen niin String-olion tila ei muutu vaan ohjelma luo uuden String-olion.

Kuvissa 3 ja 4 havainnollistetaan muuttumattomuutta näyttämällä aluksi, kuinka tietoa muokataan olio-ohjelmoinnissa ja sen jälkeen, kuinka sitä muokataan funktionaalisesti.



Kuva 2. Olio-ohjelmoinnissa olion sisältämää dataa voidaan muokata suoraan

Kuvassa 2 ilmenee, kuinka olion tila muuttuu, kun sen sisältämää dataa muokataan. Olion instanssi siis pysyy sen sisältämän rekisterinumeron arvon muokkaamisen jälkeen.



Kuva 3. Funktionaalisessa ohjelmoinnissa luodaan kokonaan uusi olio, joka ottaa vastaan halutut muutokset

Kuten kuvasta 3 ilmenee, funktionaalisessa ohjelmoinnissa tilan muutos käsitellään täysin eri tavalla eli luodaan uusi instanssi oliosta.[6.]

2.8 Rekursio

Yksi isoimmista yllätyksistä funktionaaliseen ohjelmointiin tutustujille on, että silmukoita ei käytetä funktionaalisessa ohjelmoinnissa lainkaan. Olio-ohjelmoinnissa ne ovat työkalu, jota on aina käytetty. Miksi niitä ei siis käytettäisi myös funktionaalisessa ohjelmoinnissa?

Ensimmäiseksi otetaan käsittelyyn se, mihin silmukoita tarvitaan. Silmukoita käytetään vuonohjaukseen. Imperatiivisissa kielissä voimme käskeä silmukkaa suorittamaan tietty tehtävä x kertaa. Silmukoita käytetään myös iteroimiseen. Iteraatiolla tarkoitetaan tietyn tehtävän tai prosessin toistamista, kunnes lopetusehto täyttyy. Tietyissä tilanteissa tiedetään, montako iteraatiota tarvitaan ja välillä iteraatiokertojen tarvetta on mahdotonta määrittää.

Funktionaalisessa ohjelmoinnissa silmukoita ei käytetä, koska funktionaalisessa ohjelmoinnissa käytetään puhtaita funktioita. Puhtaat funktiot eivät määritelmänsä mukaan saa aiheuttaa sivuvaikutuksia. Katsotaan alla olevaa esimerkkiä.

```
public int listanSumma(Arraylist<Integer> lista){

     int summa = 0;

     for(int numero : lista){
          summa += numero;
     }
          return summa;
     }
}
```

Esimerkkikoodi 6. Määritellään funktio, joka ottaa parametrinä sisäänsä ArrayList()-olion. Funktio käyttää for-silmukkaa ja palauttaa lista-alkioiden arvojen summan.

Esimerkissä käytettiin yksinkertaista for-silmukkaa määrittämään listan sisältämien alkioiden summan. Funktio näyttää puhtaalta ja turvalliselta. Valitettavasti funktio ei kuitenkaan ole vapaa sivuvaikutuksilta. Tosiasiassa silmukat ovat riippuvaisia sivuvaikutuksista vuonohjauksen takia. Esimerkissä sivuvaikutuksen aiheuttaja on lausesumma +=, koska se muokkaa tilaa. Voidaan todeta, että vaikka funktio on puhdas, niin silmukka ei ole. Myöskään muuttujat eivät ole puhtaita, sillä ne eivät ole muuttumattomia. Silmukat tarvitsevat toimiakseen muuttujia, joten niitä ei voida sallia puhtaissa funktioissa.

Rekursiolla ohjelmoinnissa tarkoitetaan tekniikkaa, jossa funktio kutsuu itseään. Rekursion avulla mahdollistetaan monimutkaisten ongelmien hajottaminen helpommin ratkaistaviin ongelmiin. Esimerkkikoodissa 7 havainnollistetaan rekursiota yksinkertaisella summausfunktiolla:

```
public class Rekursio{
     public static void main(String[] args) {
       int summa = summaaNumerot(5);
       System.out.println(summa);
     }
     public static int summaaNumerot(int i){
       if(i > 0){
          return i + summaaNumerot(i-1);
          }else{
          return 0;
          }
     }
}
```

Esimerkkikoodi 7. SummaaNumerot()-funktiota kutsuttaessa se lisää syötteen i arvon kaikkien i:tä pienempien numeroiden summaan ja palauttaa lopputuloksen, kunnes i:n arvo on 0, jolloin funktio palauttaa 0:n.

Kuten esimerkkikoodista 7 ilmenee, rekursiivinen kutsu suoritetaan ehtolauseen sisällä, joten silmukoille ei ole tarvetta. Rekursiivinen kutsu tarkoittaa sitä, kun metodi tai funktio kutsuu itseään suorituksensa aikana. Rekursiivinen kutsu voi myös tarkoittaa sitä, että metodi A kutsuu metodia B, joka taas kutsuu metodia A suorituksensa aikana.[7.]

3 Funktionaalinen ohjelmointi Javalla

Java oli alun perin puhtaasti imperatiivinen ohjelmointikieli. Monelle saattaakin tulla yllätyksenä, että funktionaalinen ohjelmointi on tullut Javassa toteuttamiskelpoiseksi vasta Java 8 -julkaisun myötä. Miksi näin yleensä tehtiin? Javan kehittäjät halusivat poistaa aukon bisneslogiikan ja koodin väliltä. Sen sijaan että kerrotaan, miten haluamme koodin toimivan, kerrotaan mitä halutaan tehdä.

Tässä luvussa käydään läpi erilaisia tekniikoita funktionaaliseen ohjelmointiin Javalla sekä tutkitaan sitä, miten se toimii rinnakkain olio-ohjelmoinnin kanssa. Lopuksi saadaan käsitys siitä, miten pystymme kirjoittamaan parempaa koodia hyödyntämällä sekä vanhoja olio-ohjelmointitekniikoita että funktionaalisia tekniikoita.

3.1 Apukirjastot

Java tarjoaa monia vaihtoehtoja funktionaalisen ohjelmoinnin mahdollistamiseksi. Näistä hyviä esimerkkejä ovat:

- http://www.vavr.io on tätä työtä kirjoitettaessa 5 vuotta vanha avoimen lähdekoodin funktionaalinen kirjasto Javalle. Kirjasto tarjoaa dokumentaatioita sekä moduuleita käyttäjille. Vavr tarjoaa myös muuttumattomat kokoelmat (immutable collections) ja niille tarpeelliset funktiot sekä hallintarakenteet (control structures).

- http://www.functionaljava.org on vuonna 2014 rakennettu avoimen lähdekoodin kirjasto, joka tarjoaa funktionaalisuutta Java-ohjelmointiin. Lisäksi kirjasto tarjoaa abstraktioita datarakenteille, muuttumattomille kokoelmille (immutable collections) ja lukuisia muita abstraktioita.

Tässä työssä hyödynnetään yhtä apukirjastoa funktionaalisen ohjelmoinnin sisään ajamiseksi nimeltään java.util.function. Kyseessä on kirjasto, jonka tarkoitus on siis mahdollistaa puhtaasti funktionaalista ohjelmointia Javalla. Kirjasto tarjoaa paljon perus sekä edistyneitä ohjelmointiabstraktioita, joita käytetään yleisesti funktionaalisessa ohjelmoinnissa.

3.2 Funktionaaliset rajapinnat

Java 8:n mukana tulleet funktionaaliset rajapinnat sekä lambda-lauseke auttavat lyhyemmän ja puhtaamman koodin kirjoittamisessa. Mikä tahansa rajapinta, joka sisältää yhden abstraktin metodin, on funktionaalinen rajapinta ja sen toteutusta voidaan käsitellä lambda-lausekkeena. Java 8:n oletusmetodit (default methods) eivät ole abstrakteja, joten edellä mainittu sääntö ei päde niihin. Funktionaalinen rajapinta saattaa sisältää monia oletusmetodeja tai niitä ei välttämättä ole ollenkaan.

Funktionaaliset rajapinnat tarvitaan, koska Javassa välitetään parametreina aina olioita. Näennäisesti parametrina välitettävä toiminnallisuus (funktio) on toteutettu olion sisällä sen ainoassa metodissa. Tämä on se tapa, jolla Javaan saatiin funktionaalisuus mukaan. Esimerkiksi java.awt.event.ActionListener-rajapinta sisältää vain yhden metodin. Tämä tarkoittaa sitä, että se on funktionaalinen rajapinta.[7.] Havainnollistetaan asiaa toteuttamalla actionListener-rajapinta aluksi perinteisellä tavalla, jonka jälkeen seuraavassa esimerkissä hyödynnetään lambda-lauseketta.

```
public static void main(String[] args) {
        JButton painike = new JButton();
        TextField teksti = new TextField();

        painike.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                int numero = 0;
                numero++;
                teksti.setText("Painiketta painettu " +
                numero + " kertaa");
            }
        });

    }
```

Esimerkkikoodi 8 ActionListener-rajapinnan toteutus tehdään normaalisti määrittämällä oletusmetodi actionPerformed.

```
public static void main(String[] args) {
       JButton painike = new JButton();
       TextField teksti = new TextField();

       painike.addActionListener(e -> {
           int numero = 0;
           numero++;
           teksti.setText("Painiketta painettu " +
           numero + " kertaa");
       });
    }
```

Esimerkkikoodi 9. ActionListener rajapinnan toteutus lambaa hyödyntäen. Tämä toteutus on mahdollinen, koska rajapinta sisältää vain yhden metodin ja on siis funktionaalinen rajapinta.

Kuten esimerkkikoodeista 8 ja 9 ilmenee, Java 8:n myötä pystymme hyödyntämään lambda-lauseketta yhden metodin sisältävän rajapinnan (funktionaalinen rajapinta) toteuttamiseen.

Lambda-lauseke on hyödyllinen, kun halutaan esimerkiksi iteroida tai suodattaa dataa. Lambda-lausekkeen ansiosta koodista tulee lyhyttä ja selkeää. Edellä mainitun lisäksi meidän ei tarvitse määrittää funktionaalisen rajapinnan metodia uudelleen sen hyödyntämiseksi. Javassa tärkeää onkin ymmärtää, että lambda-lauseke on funktio. Esimerkkikoodissa 10 demonstroidaan, kuinka lambda-merkintää pystytään käyttämään toteuttamaan funktionaalisia rajapintoja.

```
public class Kone {

    private String merkki;
    private int vVuosi;

    public Kone(String m, int v) {
        this.merkki = m;
        this.vVuosi = v;
    }
    //Määritetään tarvittavat metodit get(), set() ja toString()

    //Määritetään predicate-funktio
    static Predicate<Kone> tarkistaMerkki =
    x -> (x.merkki == "Apple");

    //Määritetään BiFunktio(Kahden argumentin funktio)
    static BiFunction<Kone, Integer, Integer> muokkaaKoneenVvuosi =
    (x,y) -> (x.vVuosi + y);
}
public static void main(String[] args) {
    Kone kone1 = new Kone("Apple", 2011);
    Kone kone2 = new Kone("Asus", 2020);
    System.out.println(tarkistaMerkki.test(kone1)); //true
    System.out.println(tarkistaMerkki.test(kone2)); //false
    System.out.println(muokkaaKoneenVvuosi.apply(kone1,2)); //2013
```

Esimerkkikoodi 10. Määritetään Kone-luokkaan kaksi funktionaalisen rajapinnan toteuttajaa tarkistaMerkki-predicate-funktio sekä muokkaaKoneenVvuosi-BiFunktio. Lopuksi luodaan kaksi Kone-oliota main()-metodissa ja hyödynnetään funktioita. Tässä esimerkissä konsoliin tulostetaan funktion palautusarvoja abstrakteilla oletusmetodeilla test()- sekä apply(). Edellä mainitut käydään läpi luvuissa 3.2.2 ja 3.2.4

Kuten esimerkkikoodista 10 ilmenee, voidaan lambda-lausekkeella toteuttaa funktionaalisia rajapintoja helposti. Rajapinnoista ja niiden sisältämistä metodeista on lisää luvuissa 3.2.2, 3.2.3, 3.2.4 sekä 3.2.5.

3.2.1 Stream

Kuten aiemmin työssä on mainittu, Stream-rajapinta mahdollistaa oliokokoelmien käsittelyn putkitetussa mallissa. Stream alkaa aina lähteestä, josta luodaan stream. Mahdollisia lähteitä ovat esimerkiksi Collections-rajapinnan toteuttaja, kokonaisluvut, liukuluvut ja tiedostorivit (lines of a file). Streamin suorittamat operaatiot ovat joko välitysoperaatioita (intermediate operation) tai lopettavia operaatioita. Havainnollistetaan vuon luomista kokonaisluvuilla sekä liukuluvuilla seuraavassa esimerkkikoodissa:

```
LongStream longVuo = LongStream.of(1,2,3); //1,2,3
DoubleStream doubleVuo = DoubleStream.of(1.2,1.3,1.4); //1.2,1.3,1.4
Stream<Integer> vuo1 = Stream.of(10); //10
Stream<Integer> vuo2 = Stream.of(1,2,3); //1,2,3
IntStream vuo3 = IntStream.range(1,4); //1,2,3,4
```

Esimerkkikoodi 11. Luodaan viisi vuota kaikki omalla tavallaan. Viimeinen vuo luodaan range()-metodilla, joka palauttaa kaikki kokonaisluvut kahden syötteargumentin väliltä mukaan lukien argumentit itse.

Kuten esimerkkikoodista 11 ilmenee, pitää jokaista lukutyyppiä varten tehdä omanlainen vuo. Java 8 sisältää jokaiselle vuotyypille omat metodinsa kyseisten primitiivien käsittelyä varten.

Kuva 4. Stream alkaa lähteestä, käy sille määrätyt operaatiot läpi ja päättyy lopettavaan operaatioon

Välittävät operaatiot kuten filter, map ja sort palauttavat uuden streamin, joten niitä pystyy helposti ketjuttamaan peräkkäin. Lopettavat operaatiot kuten forEach,

collect tai reduce ovat joko void-tyyppisiä tai palauttavat lopputuloksen, joka ei ole stream (kokoelman tai yksittäisen arvon).

Oleellista on ymmärtää, että streamit hyväksyvät operaatioihinsa argumenteiksi funktioita. Esimerkiksi filter() hyväksyy argumentikseen predicate-funktion, sillä predicaten test()-metodi palauttaa true tai false, ja se on filterin käytössä päätarkoitus. Predicate-rajapinta ja sen test()-metodi käydään läpi luvussa 3.2.2. Havainnollistetaan vuon toteutusta seuraavalla esimerkillä:

```
public class Esimerkki{

static Predicate<Kone> tarkistaVuosi = x -> (x.valmistusvuosi < 2019);

public static List<Kone>
 filterKone(List<Kone> koneLista, Predicate<Kone> predicate){
     return koneLista.stream()
            .filter(predicate)
            .collect(Collectors.<Kone>toList());
 }
}
```

Esimerkkikoodi 12. Määritellään predicatefunktio-tarkistaVuosi, joka tarkistaa, onko valmistusvuosi pienempi kuin 2019, jonka jälkeen määritellään metodi filterKone(), joka ottaa argumentikseen sekä listan Kone-olioita sekä predicatefunktion ja palauttaa vuon, joka hyödyntää filter()-metodia sekä collect()-metodia.

```
public static void main(String[] args) {
        FunktioEsimerkki f = new FunktioEsimerkki();

        Kone kone1 = new Kone("Apple", 2021);
        Kone kone2 = new Kone("Apple", 2018);
        Kone kone3 = new Kone("Apple", 2019);
        Kone kone4 = new Kone("Apple", 2001);
        Kone kone5 = new Kone("Apple", 2020);
        Kone kone6 = new Kone("Apple", 2020);
        Kone kone7 = new Kone("Apple", 2014);

        //lista kone-olioista joiden valmistusvuosi oli ennen 2019
        List<Kone> koneList = new ArrayList<Kone>();
        koneList.addAll(Arrays.asList(new    Kone[]{kone1,kone2,kone3,
        kone4,kone5,kone6,kone7}));

        System.out.println(tarkistaVuosi.test(kone1)); //false
        System.out.println(tarkistaVuosi.test(kone4)); //true
        System.out.println(tarkistaVuosi.negate().test(kone1)); //true
```

```
        System.out.println(filterKone(koneList,tarkistaVuosi));

        /*
        Luodaan main()-metodissa uusi stream,
        joka tulostaa jokaisen ehdot täyttävän koneen omalle rivilleen
        */
        koneList.stream()
               .filter(x -> x.valmistusvuosi < 2019)
               .collect(Collectors.toList())
               .forEach(System.out::println);

    }
}
```

Esimerkkikoodi 13. Hyödynnetään aiemmin määriteltyjä funktioita main()-luokassa määrittelemällä Kone-olioista koostuva lista. Lopuksi suoritamme tarvittavat funktiokutsut. Lisäksi luomme uuden vuon, joka toimii täysin samalla tavalla kuin edellinen. Tarkoituksena on demonstroida vuon luontia main()-metodissa.

Kuten esimerkkikoodeista 12 ja 13 ilmenee, pystytään predicate-funktiota käyttämään syötteenä funktionaalisessa ohjelmoinnissa. Funktio käyttäytyy siis kuten muuttuja perinteisessä olio-ohjelmoinnissa.[7.]

3.2.2 Predicate

Matematiikassa predikaatti (predicate) ymmärretään yleisesti totuusarvofunktiona. Java 8 predicate toimii täysin samalla tavalla ja sitä tarvitaan tiedon suodattamisessa erityisesti filter()-metodissa.

Java 8 Predicate-rajapinta sisältää yhden abstraktin metodin test(T t). test() ottaa syötteenä argumentin t ja palauttaa totuusarvon (boolean). Rajapinnan esittely näyttää seuraavalta:

```
@FunctionalInterface
public interface Predicate<T> {
     boolean test(T t);
}
```

Esimerkkikoodi 14. Predikaattifunktion rajapinnan esittely.

Havainnollistetaan test()-metodia määrittelemällä rajapinnan toteuttava luokka, joka sisältää siis pakollisen abstraktin metodin test().

```
import java.util.function.Predicate;

public class PredicateEsimerkki implements Predicate<String> {

    public boolean test(String s) {
        if (s.length() < 10){
            return true;
        }
        else{
            return false;
        }
    }
}
```

Esimerkkikoodi 15. Määritellään luokka PredicateEsimerkki, joka toteuttaa Predicate-rajapinnan

Kuten esimerkkikoodista 15 ilmenee, predikaattifunktion tehtävä on arvioida, onko syötetyn String-olion pituutta vastaava kokonaisluku pienempi kuin 10. Hyödynnetään juuri määrittettyä luokkaa main()-metodissa.

```
import java.util.function.Predicate;

public class Main {

    public static void main(String[] args) {

        Predicate<String> pe = new PredicateEsimerkki();

        System.out.println(pe.test("Tietokone")); //true

        System.out.println(pe.test("Seuraava tietokone")); //false
    }
}
```

Esimerkkikoodi 16. Luodaan PredicateEsimerkki-luokan ilmentymä ja hyödynnetään sinne määritettyä test()-metodia.

Predicate-rajapinta sisältää lisäksi seuraavat oletusmetodit:

- and(Predicate<? super T> other)
- isEqual(object targetRef)
- negate()
- or(Predicate<? super T> other).

and()-metodi ottaa syötteenä predikaattifunktion ja palauttaa yhdistetyn (composed) predikaattifunktion, joka esittää loogista ”AND”-operaattoria tästä funktiosta ja syötteestä. Havainnollistetaan and()-metodia esimerkillä:

```
import java.util.function.Predicate;

public class Esimerkki {

public static void main(String[] args){

Predicate<Integer> vertaaYksi = x -> x > 2;
Predicate<Integer> vertaaKaksi = x -> x < 7;

Predicate testaaPredikaatti = vertaaYksi.and(vertaaKaksi);

int x = 5;
int y = 10;

boolean onkoTosi1 = testaaPredikaatti.test(x); //true
boolean onkoTosi2 = testaaPredikaatti.test(y); //false

     }
}
```

Esimerkkikoodi 17. määritellään kolme predikaattifunktiota, joista ensimmäiset kaksi vertaavat, onko syöte isompi kuin 2 tai pienempi kuin 7. Määritellään kolmas predikaattifunktio and()-metodin avulla, joka ketjuttaa edellä mainitut funktiot yhteen ja käytetään sitä kokonaislukujen x sekä y arvioimiseen.

Kuten esimerkkikoodista 17 ilmenee, voidaan and()-metodia käyttää kahden predikaattifunktion ketjuttamiseen yhdeksi funktioksi. Tässä tapauksessa ketjutettujen funktioiden yhdistelmä olisi testaaPredikaatti-niminen funktio.

isEqual()-metodi on toinen Predicate-rajapinnan oletusmetodeista. Metodi ottaa syötteenä yhden argumentin ja palauttaa predikaatin, joka testaa, ovatko argumentit samanarvoisia Object.equals(Object, Object)-metodin mukaisesti. Esimerkkikoodi 18 havainnollistaa isEqual()-metodin toteutusta.

```
Predicate.isEqual(20).test(30); //false
Predicate.isEqual(20).test(20); //true
```

Esimerkkikoodi 18. Kutsutaan Predicate-funktiota toteuttamaan isEqual()-metodi. test()-metodin avulla verrataan siihen syötettyä arvoa isEqual()-metodin syötteen kanssa.

Kuten esimerkkikoodista 18 ilmenee, voidaan isEqual()-metodia käyttää esimerkiksi kahden kokonaisluvun vertailuun keskenään. isEqual()-metodi sallii syötteenä myös ”null”.

negate()-metodi palauttaa predikaatin, joka edustaa kohdepredikaatin loogista negaatiota. Loogisella negaatiolla tarkoitetaan jonkin asian vastakohtaa. Esimerkiksi matemaattisen lauseen ”2>3” negaatio on ” 2≤3”. Havainnollistetaan negate()-metodia esimerkkikoodissa 19:

```
Predicate<Integer> parillisuustesti = numero -> (numero % 2 == 0);

parillisuustesti.negate().test(8); //false
parillisuustesti.negate().test(9); //true
```

Esimerkkikoodi 19. Määritellään funktio, joka suorittaa syötetyille kokonaisluvuille parillisuustarkistuksen. Tämän jälkeen kutsutaan negate()-metodia tarkistamaan, onko syöte alkuperäisen funktiosäännön negaatio.

Kuten esimerkkikoodista 19 voidaan havaita, negate()-funktio palauttaa kokonaisluvulle 8 arvoksi false, koska testaaNegaatio-funktio itsessään palauttaisi samalle syötteelle true. Sama pätee seuraavan rivin syötteeseen 9 mutta tietysti päinvastoin.

or()-metodi ottaa syötteenä predikaattifunktion ja palauttaa yhdistetyn predikaattifunktion, joka esittää loogista ”OR”-operaattoria tästä funktiosta ja syötteestä. Havainnollistetaan or()-metodia esimerkillä.

```
Predicate<Integer> numeroTesti1 = numero -> (numero == 0);
Predicate<Integer> numeroTesti2 = numero -> (numero > 10);

numeroTesti1.or(numeroTesti2).test(1); //false
numeroTesti1.or(numeroTesti2).test(11); //true
```

Esimerkkikoodi 20. Määritellään kaksi predikaattifunktiota, joiden ehdot ovat erilaiset ja ketjutetaan ne or()-metodilla. Samassa lauseessa testataan, onko syöte yhtä kuin 0 tai suurempi kuin 10.

Esimerkkikoodista 20 nähdään or()-metodin käyttö kahden predikaattifunktion ketjutuksessa. Ensimmäinen syöte ei täytä kumpaakaan ehtoa, joten predikaattifunktio palauttaa false. Jälkimmäisessä tapauksessa toinen ehdoista täyttyy, joten palautusarvo on true.

Mikäli tarvitaan kahden argumentin predicate-funktiota, voidaan käyttää BiPredicate-rajapintaa. Kyseinen rajapinta toimii samoin kuin Predicate, mutta sallii kahden argumentin syöttämisen funktioon yhden sijaan.[9.]

3.2.3 Consumer

Consumer-rajapinta poikkeaa funktionaalisen ohjelmoinnin perusperiaatteista, sillä se ottaa syötteeksi olion ja muokkaa olion arvoja ohjelmoijan määrittämällä tavalla mutta ei palauta mitään. Kuten aiemmin työssä mainittu, funktionaalisessa ohjelmoinnissa vältetään tiedon muokkaamista funktioissa, sillä sen seurauksena syntyy sivuvaikutuksia. Useimmiten consumer-funktioita käytetään juuri olion sisältämän datan muokkaamiseen tai olion arvon tulostamiseksi konsoliin. Consumer-funktio toimii forEach()-metodin syötteenä loistavasti, sillä kyseisen metodin halutaan usein tulostavan jotain konsoliin. Esimerkkikoodissa 21 näytetään esimerkki consumer-funktion hyödyntämisestä vuon yhteydessä:

```
//predicate-funktio vuon suodatinta varten
public static Predicate<Kone> konePredicate =
 x -> x.vVuosi < 2011 && x.vVuosi > 2007;

//consumer-funktio, joka tulostaa konsoliin halutut koneet
public static Consumer<Kone> koneConsumer =
 x -> System.out.println(x.merkki + " Pääsi suodatuksesta läpi");

public static void main(String[] args) {
        Kone kone1 = new Kone("Apple", 2006);
        Kone kone2 = new Kone("Asus", 2018);
        Kone kone3 = new Kone("Apple", 2010);
        Kone kone4 = new Kone("Fujitsu", 2001);
        Kone kone5 = new Kone("Apple", 2009);
        List<Kone> lista = new ArrayList<>();
        lista.add(kone1);
        lista.add(kone2);
        lista.add(kone3);
```

```
        lista.add(kone4);
        lista.add(kone5);

        lista.stream()
                .filter(konePredicate)
                .forEach(koneConsumer); //Apple, Apple
}
```

Esimerkkikoodi 21. Consumer-funktion hyödyntäminen forEach()-metodin syötteenä.

Consumer-rajapinta sisältää yhden abstraktin metodin accept() sekä yhden oletusmetodin andThen(Consumer<? super T> after). accept() on void-tyypin metodi ja suorittaa siis consumer-funktioon määritetyt operaatiot syötteen mukaan. Consumer on hyödyllinen rapapinta, sillä esimerkiksi kokoajan forEach()-metodi hyväksyy sen argumentikseen ja sitä voidaan näin ollen käyttää vuon päätösoperaation argumenttina.

Havainnollistetaan accept()-metodia määrittämällä rajapinnan ilmentymäluokka, joka sisältää abstraktin metodin accept() esimerkkikoodissa 22.

```
import java.util.function.Consumer;

public class ConsumerEsimerkki implements Consumer<String> {

    public void accept(String s) {
        System.out.println(s);
    }
}
```

Esimerkkikoodi 22. Määritellään luokka ConsumerEsimerkki, joka toteuttaa Consumer-rajapinnan.

Kuten esimerkkikoodista 22 ilmenee, consumer-rajapinnan accept()-metodi ottaa tässä tapauksessa argumentiksi String-olion ja tulostaa sen konsoliin. Havainnollistetaan vielä juuri määrittämämme luokan toimintaa main()-metodissa:

```
import java.util.function.Consumer;

public class Main {

    public static void main(String[] args) {

        Consumer ce = new ConsumerEsimerkki();

        ce.accept("Kone");
    }
}
```

Esimerkkikoodi 23. Luodaan ConsumerEsimerkki-luokan ilmentymä ja hyödynnetään sinne määritettyä accept()-metodia.

andThen() on Consumer rajapinnan ainoa oletusmetodi, joka palauttaa yhdistetyn Consumer-funktion, joka suorittaa ensin funktioon määritetyn operaation ja seuraavaksi metodiin syötetyn operaation. Havainnollistetaan andThen()-metodia esimerkkikoodissa 24:

```
import java.util.function.Consumer;

public class Esimerkki{
     public static void main(String[] args){

     Consumer<String> esim1 = x -> System.out.println(x.toUpperCase());
     Consumer<String> esim2 = x -> System.out.println(x.toLowerCase());

     esim1.andThen(esim2).accept("Esimerkki");
     //ESIMERKKI
     //esimerkki
     }
}
```

Esimerkkikoodi 24. Luodaan kaksi Consumer-funktiota ja yhdistetään ne andThen()-metodilla. Lopuksi suoritetaan funktiokutsu accept()-metodin avulla.

Kuten esimerkkikoodista 24 voidaan havaita, andThen()-metodia käytetään kahden consumer-funktion toiminnallisuuden ketjuttamiseen.

Mikäli tarvitaan kahden argumentin Consumer-funktioita, voidaan käyttää BiConsumer-rajapintaa. Kyseinen rajapinta toimii samoin kuin Consumer, mutta sallii kahden argumentin syöttämisen funktioon yhden sijaan. [9]

3.2.4 Function

Function-rajapinta ottaa syötteenä olion ja palauttaa sille määritetyn palautusarvon. Yksi suurimmista käyttötarkoituksista rajapinnan toteutuksille on map()-funktio, jossa syötetty olio muutetaan joksikin muuksi tarpeen mukaan. Usein function-rajapinnan toteuttajat toimivat argumenttina map()-metodille vuon yhteydessä, jossa tietyn tyyppinen vuo muutetaan joksikin muuksi. Esimerkiksi String-olioista koostuva vuo voidaan muuttaa vuoksi, joka koostuu kokonaisluvuista.

Function-rajapinta sisältää yhden abstraktin metodin apply(T t). Metodi toteuttaa kyseisen funktion siihen syötetyn argumentin perusteella. Havainnollistetaan apply()-metodia määrittämällä luokka, joka toteuttaa Function-rajapinnan esimerkillä.

```
import java.util.function.Function;

public class FunctionEsimerkki implements Function<String, Integer> {

    public Integer apply(String s) {
        return s.length();
    }
}
```

Esimerkkikoodi 25. Määritellään luokka FunctionEsimerkki, joka toteuttaa Function-rajapinnan.

Kuten esimerkkikoodista 25 ilmenee, abstrakti metodi apply() ottaa argumentikseen String-olion ja palauttaa kokonaislukuna kyseisen olion pituuden. Seuraavaksi havainnollistetaan luokan toimintaa main()-metodissa:

```
import java.util.function.Function;

public class Main {

    public static void main(String[] args) {

        Function<String, Integer> fe = new FunctionEsimerkki();
        System.out.println(fe.apply("Kone")); //4

    }
}
```

Esimerkkikoodi 26. Luodaan FunctionEsimerkki-luokan ilmentymä ja hyödynnetään sinne määritettyä apply()-metodia.

Kuten edellä on mainittu, funktionaaliset rajapinnat sisältävät sekä abstrakteja että oletusmetodeja. Function-rajapinta sisältää seuraavat oletusmetodit:

- compose(Function<? super V,? extends T> before)
- andThen(Function<? super R,? extends V> after)
- identity().

compose() sekä andThen()-metodien tarkoitus on yhdistää funktioita toisiinsa. Ero näiden metodien välillä on se, missä järjestyksessä ne suorittavat tehtävänsä. Havainnollistetaan kummankin metodin toimintaa esimerkillä.

```
import java.util.function.Function;

public class Esimerkki {

public static void main(String[] args){

Function<Integer, Integer> plusYksi = i -> i + 1;

Function<Integer, Integer> kertaaKaksi = i -> i * 2;

plusYksi.compose(kertaaKaksi).apply(5); //palauttaa 11

plusYksi.andThen(kertaaKaksi).apply(5); //palauttaa 12
}}
```

Esimerkkikoodi 27. Määritellään kaksi funktiota, jotka ottavat sisäänsä kokonaisluvun ja palauttavat kokonaisluvun. plusYksi-funktio palauttaa kokonaisluvun, jonka arvoa on kasvatettu yhdellä, kun taas kertaaKaksi-funktio palauttaa sinne

syötetyn arvon kerrottuna kahdella. Viimeisenä käytetään compose()-, andThen()- sekä apply()-metodeja toteuttamaan funktioita.

Kuten esimerkkikoodista 27 ilmenee, compose()- ja andThen()-metodeja käytetään funktioiden ketjuttamiseen. compose()-metodi toteuttaa ensin syötteenä saamansa funktion, jonka jälkeen se toteuttaa funktion, joka kutsuu compose()-metodia. andThen()-metodi toimii päinvastaisessa järjestyksessä eli ensin toteutetaan kutsujafunktio, jonka jälkeen toteutetaan siihen ketjutettu funktiokutsu argumentin perusteella.

identity()-metodi puolestaan palauttaa funktion, joka palauttaa aina kyseisen funktion syötteen argumentin. Seuraavaksi on esimerkki identity()-metodin esittely rajapinnassa.

```
static <T> Function<T, T> identity() {
          return t -> t;
}
```

Esimerkkikoodi 28. Kuten esimerkkikoodista 28 ilmenee, identity()-metodi palauttaa funktion, joka palauttaa aina syötteensä argumentin.

Yksi identity()-metodin käyttötarkoituksista on sen hyödyntäminen map()-metodin yhteydessä, kun syöte tarvitaan suoraan palautuksena.[8.] Havainnollistetaan identity()-metodia ja sen toimintaa seuraavalla toteutuksella.

```
public static void main(String[] args) {
        Kone kone1 = new Kone("Apple", ”2011”);
        Kone kone2 = new Kone("Asus", ”2020”);
        Kone kone3 = new Kone("Lenovo", ”2012”);
        Kone kone4 = new Kone("HP", ”2021”);

        List<Kone> list = Arrays.asList(kone1,kone2,kone3,kone4);
        Map<String,Kone> map = list.stream()
                .collect(Collectors.toMap
                (Kone::getVvuosi, Function.identity()));
                //Function.identity = function(e -> e)

        System.out.println(map); //{2012=Lenovo, 2011=Apple,
        2021=HP, 2020=Asus}

    }
```

Esimerkkikoodi 29. Luodaan Kone-oliot ja lisätään ne listaan. Tehdään listasta stream, jossa listasta muutetaan map, joka sisältää kaikki koneet. Lopuksi tulostetaan map konsoliin.

Kuten esimerkkikoodista 29 ilmenee, identity()-metodia voidaan käyttää funktion määrittelyyn. Tässä tapauksessa mapin avain on koneen valmistusvuosi ja arvo on koneen merkki.

Mikäli tarvitaan kahden argumentin funktioita, voidaan käyttää BiFunction-rajapintaa. Kyseinen rajapinta toimii samoin kuin Function mutta sallii kahden argumentin syöttämisen funktioon yhden sijaan. [9] Voimme myös itse määritellä kolmen tai useamman argumentin funktion seuraavalla tavalla:

```
@FunctionalInterface
public interface iFunction<T1, T2, T3, R> {
    R apply(T1 t1, T2 t2, T3 t3);

    default <V> iFunction<T1, T2, T3, V> andThen(
            Function<? super R, ? extends V> after) {
            Objects.requireNonNull(after);
            return (T1 t1, T2 t2, T3 t3) ->
            after.apply(apply(t1,t2,t3));
    }
}
```

Esimerkkikoodi 30. Kolmen argumentin funktionaalisen rajapinnan esittely, joka sisältää apply()- sekä andThen()-metodit.

3.2.5 Supplier

Supplier on funktionaalinen rajapinta, joka sisältää ainoastaan yhden metodin get(). Yksi Supplier-rajapinnan mahdollisista käyttötarkoituksista on toimintojen viivästyttäminen (deferred execution). Esimerkkinä Optional-luokka sisältää orElseGet()-metodin, joka toteutuu, jos optional-olio ei sisällä dataa. Lisäksi muun muassa Kokoajan (Collector) forEach()-metodi ottaa supplierin argumentikseen. Kokoajia käydään läpi luvussa 3.2.6. Esimerkki toiminnon viivästyttämisestä myöhemmin tässä luvussa. Supplier-rajapinnan esittely on seuraava:

```
@FunctionalInterface
public interface Supplier<T> {
   T get();
}
```

Koodiesimerkki 31. Supplier-rajapinnan esittely

Rajapinta esittää operaatiota, joka ei ota argumenttia mutta palauttaa ohjelmoijan määrittämän asian (tämä saattaa olla esimerkiksi olio). Kuitenkin koska kyse on funktionaalisesta rajapinnasta, sitä voidaan käyttää lambda-lausekkeen kanssa tai metodiviittauksena. Seuraavaksi on esimerkki toiminnon viivästyttämisestä.

```
//supplier-funktio palauttaa satunnaisen liukuluvun
Supplier<Double> lukuSupplier = () -> Math.random();

//Optional luokkaa hyödyntämällä saamme tyhjää dataa
Optional<Double> valinnainen = Optional.empty();

//Hyödynnetään tyhjää dataa demonstroimaan orElseGet()-metodia
double luku = valinnainen.orElseGet(lukuSupplier); //luku saa arvonsa
supplierista koska valinnainen on tyhjä
```

Esimerkkikoodi 32. Demonstraatio toiminnon viivästyttämisestä. Koska Optional-luokan muuttuja valinnainen ei sisällä dataa saadaan supplier-funktiosta korvike tyhjälle datalle.

Havainnollistetaan supplierin sekä sen sisältämän get()-metodin toimintaa esimerkkikoodissa 33:

```
import java.util.function.Supplier;

public class SupplierEsimerkki {

    static String koneMerkki = "Apple";

    public static void main(String[] args) {

        Supplier<Boolean> totuusSupplier =
        () -> koneMerkki.length() < 10;
        Supplier<Integer> integerSupplier =
        () -> koneMerkki.length() -3;
        Supplier<String> stringSupplier =
        () -> koneMerkki.toUpperCase();

        System.out.println(totuusSupplier.get()); //true
        System.out.println(integerSupplier.get()); //2
        System.out.println(stringSupplier.get()); //APPLE
    }
}
```

Esimerkkikoodi 33. Tehdään kolme erilaista supplier-funktiota ja tulostetaan toteutus konsoliin

Kuten esimerkkikoodista 33 ilmenee, supplier ei vaadi syötettä mutta palauttaa tässä tapauksessa ensin totuusarvon (boolean), sitten kokonaisluvun sekä String-olion. [9.]

3.2.6 Collection

Collection<E> on korkeimman(root) tason rajapinta kokoelmien (collection) hierarkiassa. Kokoelma sisältää olioita, joita kutsutaan kokoelman elementeiksi. JDK (Java Development Kit) ei tarjoa yhtään suoraa implementaatiota tästä rajapinnasta. Sen sijaan se tarjoaa implementaatioita sen alirajapinnoista kuten Set ja List. Tätä periytyvyyttä käytetään tyypillisesti käsittelemään kokoelmia ja manipuloimaan niitä halutulla tavalla. Collection sisältää yhden ylirajapinnan Iterable<E>. Tämä tarkoittaa sitä, että kaikki Collection alirajapinnat, ovat iteroitavissa ja sisältävät tämän seurauksena forEach()-metodin.

forEach()-metodi ottaa parametrikseen consumerin. Tämä tarkoittaa sitä, että se käyttää toteutuksessaan accept()-metodia. forEach() - ei siis palauta mitään vaan

toimii päätösoperaationa streamille. Metodi käy läpi kokoelman jokaisen elementin, kunnes se on suorittanut määritellyt toiminnot jokaiselle kokoelman elementille.

Collector

Collector (kokoaja) redusoi vuota olioksi. Kokoajia tarvitaan, koska vuon pääasiallinen tarkoitus on tuottaa jokin tietorakenne tai arvo. Vuo käy läpi erilaisia operaatioita ja yleensä viimeisenä operaatioista läpi päässeet elementit kerätään kokoelmaan. Esimerkkejä näistä olioista ovat listat, joukot ja mapit. Kokoajan tulos voi myös olla arvo kuten long, String tai kokonaisluku. Collector on rajapinta, jonka esittely on seuraavanlainen:

```
Interface Collector<T,A,R>
```

Rajapinnan parametrit tarkoittavat seuraavaa:

- T – syötteen tyyppi (type) johon redusointi kohdistetaan
- A – Muuttuva (mutable) akkumulaattorin tyyppi eli se johon tuloksia kootaan operaation jälkeen
- R – Redusointiprosessin lopputuloksen tyyppi

Collector-rajapinta koostuu collectors-luokan tehdasmetodeista, joihin on määritelty redusointiprosessin eri vaiheet. Tässä työssä paljon käytetty collect()-metodi on metodirunko (template method), joka määrittelee prosessin eri vaiheet. Ohjelmoijan pitää siis ainoastaan kirjoittaa collect()-metodin parametri. Havainnollistetaan edellä käsiteltyä aihetta esimerkillä:

```
public class CollectorEsimerkki {
    public static void main(String[] args) {
        Stream<String> vuo = Stream.of("A","B","C","D");

        Set<String> joukko = vuo.collect(Collectors.toSet());

        System.out.println(joukko); //[A, B, C, D}
```

```
        }
}
```

Esimerkkikoodi 34. Luodaan vuo, joka sisältää String-olioita, joista muokataan joukko collect-metodirungon avulla. Tässä tapauksessa käytetään collectors-luokan toSet()-metodia, joka palauttaa Collector-rajapinnan toteuttajan, joka luo Set-tietorakenteen.

Kuten esimerkkikoodissa 34 havaitaan, voidaan kokoajaa käyttää vaivattomasti vuon tyypin muokkaamiseen collectors-luokan tehdasmetodien avulla. Ohjelmoija voi myös kirjoittaa omia kokoajia toteuttamalla Collector-rajapinnan. Esimerkkikoodin joukko on tässä tapauksessa siis Set, mutta se voisi olla vaikkapa ArrayList. Redusointiprosessin lopputuloksen tyyppi olisi tässä tapauksessa Set.

Todetaan vielä, miten kokoaja toimii vuon yhteydessä:

- Ensimmäinen vaihe – Supply: Luodaan säiliö (container) elementtien säilyttämiseen vuosta. Tähän tarvitaan supplier-funktio.

- Toinen vaihe - Accumulate: Jokainen elementti vuosta kerätään säiliöön, joka luodaan ensimmäisessä askeleessa.

- Kolmas vaihe – Combine: Valinnainen toimenpide, jota käytetään vain ,jos vuo prosessoidaan rinnakkaisessa ympäristössä. Tämä tarkoittaa sitä, että vuon elementit jaetaan ja käsitellään samanaikaisesti monessa akkumulaattorissa. combine-vaiheen idea on, että rinnakkaiset akkumulaattorit keräävät kaikki elementit akkumulaattoreista samaan säiliöön.

- Neljäs vaihe – Finish: Viimeinen vaihe, joka käynnistyy, kun accumulate (tai mahdollisesti combine) vaiheet on käyty läpi. Tässä vaiheessa muutetaan säiliö halutuksi tyypiksi (result type).

Seuraavassa esimerkkikoodissa määritellään oma kokoaja main()-metodin sisään ja demonstroidaan sen toimintaa:

```
public class Kone {

    String merkki;
    private int vVuosi,sarjanumero;

    public Kone(int s, String m, int v) {
        this.merkki = m;
        this.vVuosi = v;
        this.sarjanumero = s;
    }

    @Override
    public String toString() {
        return merkki;
    }

    public static void main(String[] args) {
        Kone kone1 = new Kone(1,"Apple", 2011);
        Kone kone2 = new Kone(2,"Asus", 2020);
        Kone kone3 = new Kone(3,"Lenovo", 2011);
        Kone kone4 = new Kone(4,"HP", 2020);

        List<Kone> list = Arrays.asList(kone1,kone2,kone3,kone4);

        Collector<Kone, StringJoiner, String> KoneMerkkiCollector =
                Collector.of(
                        () -> new StringJoiner(" | "), //supplier
                       //akkumulaattori
                        (x, y) -> x.add(y.merkki.toUpperCase())
                        (x1, y1) -> x1.merge(y1), //combiner
                        StringJoiner::toString); //finisher

        String merkit = list.stream()
                .collect(KoneMerkkiCollector);

        System.out.println(merkit);// APPLE | ASUS | LENOVO | HP
    }
}
```

Esimerkkikoodi 35. KonemerkkiCollectorin määrittäminen sekä hyödyntäminen

Kuten esimerkkikoodista 35 havaitaan, voi ohjelmoija luoda omia kokoajia ja hyödyntää niitä streamin yhteydessä. Tässä esimerkissä stringJoiner-luokan avulla määritellään, että jokaisen String-elementin väliin halutaan ” | ”-merkki. Akkumulaattorin avulla jokainen String-olion merkki asetetaan isoksi kirjaimeksi, jonka jälkeen combiner yhdistää elementit yhdeksi jonoksi. Viimeisenä finisher muuttaa elementit yhdeksi merkkijonoksi. [7.]

Map

Java 8 Stream-rajapinnan map()-metodi on välitysoperaatio, joka ottaa sisäänsä elementtejä streamin lähteestä ja tuottaa uusia muunneltuja elementtejä streamin jatkamiseen. Yksinkertaisimmillaan map()-metodi muuttaa esimerkiksi streamin String-olioita kokonaisluvuiksi. map on määritelty Stream-rajapintaan seuraavalla tavalla:

```
<R> Stream <R> map(Function<? super T,? extends R>mapper)
```

map()-metodi käyttää mapper-funktiota syötestreamiin (input stream) ja generoi tulostreamin (output stream). Havainnollistetaan map()-metodia esimerkin avulla:

```
    public static void main(String[] args) {
        List<Kone> koneList = new ArrayList<>();
        Kone kone1 = new Kone("Apple", 2011);
        Kone kone2 = new Kone("Lenovo", 2011);
        Kone kone3 = new Kone("HP", 2011);
        Kone kone4 = new Kone("Apple", 2011);
        Kone kone5 = new Kone("Lenovo", 2011);
        Kone kone6 = new Kone("HP", 2011);
        koneList.addAll
       (Arrays.asList(new Kone[]
       {kone1,kone2,kone3,kone4,kone5,kone6}));

        koneList.stream()
                .map(kone -> kone.merkki)//Kone olio
                .collect(Collectors.toList())
                .forEach(System.out::println);
    }
}
```

Esimerkkikoodi 36. Tehdään lista Kone-olioita, joista tehdään stream, jossa map()-metodin sekä toList()-metodin avulla tehdään lista String-olioita.

Kuten esimerkkikoodista 35 havaitaan, voidaan map()-metodilla muuttaa syötestreamista täysin uudenlainen stream. Alun perin stream koostui elementeistä, jotka olivat Kone-olioita ja map()-metodin avulla niistä tehtiin String-olioita. Havaitaan myös, että map()-metodi hyväksyy argumentikseen funktion tai lambda-lausekkeen. [7.]

Filter

Java 8 Stream rajapinnan filter()-metodi on välioperaatio, jota käytetään useimmiten kokoelmien (collections) prosessoimiseen. Filter ottaa argumentikseen predicaten, joka arvioi vuon elementtejä ja päästää läpi vain ne, jotka predicate hyväksyy. Filterin määritys Stream-rajapinnassa näyttää seuraavalta:

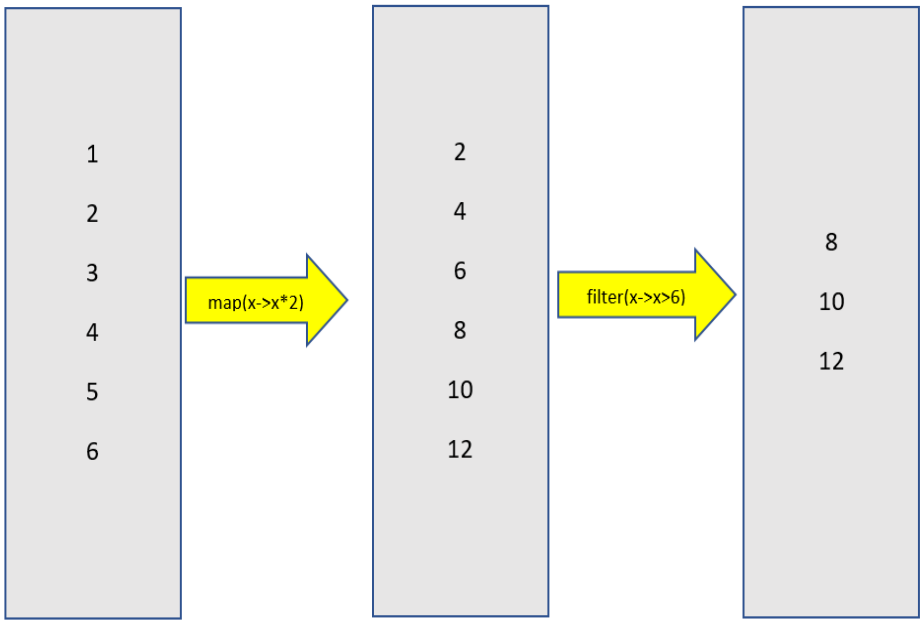
```
Stream<T> filter(Predicate<? super T> predicate)
```

Käytetään seuraavassa esimerkissä filter()-metodia karsimaan elementtejä vuosta:

```
    public static void main(String[] args) {
        List<Kone> koneList1 = new ArrayList<>();
        Kone kone1 = new Kone("Apple", 2016);
        Kone kone2 = new Kone("Lenovo", 2017);
        Kone kone3 = new Kone("HP", 2018);
        Kone kone4 = new Kone("Apple", 2019);
        Kone kone5 = new Kone("Lenovo", 2020);
        Kone kone6 = new Kone("HP", 2021);

        koneList1.addAll
       (Arrays.asList(new  Kone[]
       {kone1,kone2,kone3,kone4,kone5,kone6}));

        Predicate<Kone> suodata =
        x -> x.getMerkki() == "HP" || x.getvVuosi() == 2019;

        List<Kone> koneList2 = koneList1.stream()
                .filter(suodata)
                .collect(Collectors.toList());

        System.out.println(koneList2);

    }

}
```

Esimerkkikoodi 37. Määritellään predicate-funktio suodata, jota hyödynnetään filter()-metodin argumenttina suodattamaan Kone-olioita merkin ja valmistusvuoden perusteella.

Filter-funktio käyttää predikaattia listan jokaiseen alkioon. Vain alkiot, jotka predikaatti palauttaa totuutena (true), palautetaan uudessa listassa. Oleellista on siis havaita, että filter()-metodi hyväksyy parametrikseen predicate-funktion.



Kuva 5. Alkuperäinen lista käsitellään map()-funktiolla, jossa jokainen alkio kerrotaan kahdella ja lisätään uuteen listaan. filter()-funktio ottaa alkioista ne, jotka ovat isompia kuin kuusi ja lisää ne uuteen listaan.

Reduce

Java 8 Stream-rajapinnan reduce()-metodin avulla saadaan yksi elementti streamista määrätyn ehdon perusteella. Kyseessä on siis streamin lopettava operaatio. Palautettu elementti voi olla myös toinen lista. map()- ja filter()-funktioita voidaan ja käytetäänkin usein reduce()-funktion kanssa yhteistyössä. Seuraavassa esimerkissä reduce() saa parametrikseen BiFunktion.

```
public class Esimerkki{

     public static void main(String[] args){

     Integer summaLista = Stream.of(1,2,3,4,5)
     .reduce(100, Integer::sum);
     System.out.println(summaLista);

     summaLista = Stream.of(1,2,3,4,5).reduce(100, (x,y)->x+y);
     System.out.println(summaLista);

     }
}
```

Esimerkkikoodi 38. Luodaan stream-kokonaislukuja, jonka jälkeen käytetään reduce()-funktiota ja saadaan ulos integerien summa plus sata. Alempana sama toistuu lambda-lausekkeen avulla.

Kuten esimerkkikoodista 38 voidaan havaita, reduce() on lopettava operaatio streamissa. Reduktio auttaa ohjelmoijaa tuottamaan yhden ainoan tuloksen elementtisekvenssistä (sequence of elements) kohdistamalla yhdistelmäoperaation (combining operation) vuon elementteihin. [7.]

4 Funktionaalisuuden hyödyntämisesimerkki

Yksi tämän työn tarkoituksista on oppia hyödyntämään funktionaalista ohjelmointia olio-ohjelmoinnin kanssa rinnakkain. Toteutetaan muutaman luokan pankkisovellus, jonka tavoite on toteuttaa funktionaalista ohjelmointia ja olio-ohjelmointia rinta rinnan. Ohjelma toimii havainnollistavana esimerkkinä funktionaalisen ohjelmoinnin ja olio-ohjelmoinnin tekniikoiden yhdistämisestä. Ohjelmalla halutaan simuloida yksinkertaista pankkijärjestelmää, jonka kaikki metodit hyödyntävät funktioita asiakkaiden tilien manipuloimiseen, sekä erilaisten tietojen tarkistamiseen. Toteutus sisältää 5 luokkaa. Työ halutaan pitää mahdollisimman selkeänä, joten sijoitetaan kaikki funktiot omaan luokkaansa nimeltään FunktioToteuttaja. Lisäksi ohjelma sisältää Asiakas- , Tili- , Käsittelijä- sekä Pankki- luokat. Käsittelijä sisältää kaikki pankin toiminnallisuuteen liittyvät metodit. Pankki-luokka sisältää main()-metodin. Aloitetaan määrittämällä Tili-luokka seuraavalla tavalla:

```
public class Tili {

    private int id;
    private double saldo;
    private String tunnus, salasana;


    public Tili(int i, double s, String t, String ll, String ss){
        this.id = i;
        this.saldo = s;
        this.tunnus = t;
        this.salasana = ss;

    }
//get(),set() ja toString metodit
}
```

Esimerkkikoodi 39. Tili-luokan toteutus

Seuraavaksi toteutetaan Asiakas-luokka. Jokaiselle asiakkaalle tehdään olion luontivaiheessa oma tili, joten lisätään Tili-olio asiakkaan konstruktoriin.

```
public class Asiakas {

    private String nimi,luottoLuokitus;
    private int id;
    private double palkka;
    private Tili tili;


    public Asiakas(String n, int i, double p, String ll, Tili t){
        this.nimi = n;
        this.id = i;
        this.tili = t;
        this.palkka = p;
        this.luottoLuokitus = ll;
    }//get(),set() ja toString metodit
}
```

Esimerkkikoodi 40. Asiakas-luokan toteutus.

Palkka-muuttuja on oleellinen, kun asiakas sijoitetaan tässä toteutuksessa tiettyyn luottoryhmään. Seuraavaksi lähdetään määrittämään pankin tarvitsemia funktioita. Kuten aiemmin on mainittu, päädyttiin tässä toteutuksessa tekemään funktioille oma luokka FunktioToteuttaja selkeyden takia. Luokka sisältää lisäksi kaksi metodia, joita hyödynnetään yhdessä funktioiden kanssa ja neljä muuttujaa, jotka ovat vakioita muuttumattomuusperiaatteen toteuttamiseksi. Seuraavaksi esitellään luokan muuttujat.

```
public class FunktioToteuttaja {

   private final double korko = 1.05;

   private final double pieniLaina = 100000;
   private final double keskiLaina = 150000;
   private final double isoLaina = 200000;
```

Esimerkkikoodi 41. FunktioToteuttaja-luokan muuttujat.

Kuten esimerkkikoodista 41 ilmenee, kaikki muuttujat on määritetty final-avainsanalla vakioksi. Funktionaalisessa ohjelmoinnissa vältetään sivuvaikutuksia, ja tämä on Javan tapa toteuttaa vakioita. Näitä arvoja hyödynnetään myöhemmin määriteltävissä funktioissa. Seuraavaksi luodaan DateTimeFormatter-olio, joka tuli myös Java 8:n mukana. Kyseessä on luokka, jonka avulla pystymme jäsentelemään (parse) Päivämäärä-Aika-olioita (date-time objects). Olion avulla pystymme tekemään supplier-funktion, joka tarjoaa aikaleiman (timestamp) pankkitransaktioihin:

```
private static final DateTimeFormatter dtf =
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

Supplier<String> haeTimeStamp =
() -> dtf.format(Local DateTime.now());
```

Esimerkkikoodi 42. Luodaan DateTimeFormatter-olio dtf sekä määritellään haeTimeStamp-funktio.

Lisäksi ohjelma sisältää supplier-funktion jokaiselle konsoliin tulostettavalle merkkijonolle. Useat merkkijonot ovat identtisiä, joten funktionaalisuus lyhentää sekä siistii lopputulosta. Näistä esimerkkejä ovat muun muassa:

```
Supplier<String> supplier1 = () -> "Anna käyttäjätunnuksesi: ";
Supplier<String> supplier2 = () -> "Anna salasanasi";
Supplier<String> supplier3 =
() -> "Käyttäjätunnus ja salasana oikein, tervetuloa ";
Supplier<String> supplier7 = () -> "Tililläsi on nyt ";
Supplier<String> supplier8 = () -> " euroa!";
```

Esimerkkikoodi 43. Supplier-funktioita toistuvien tulosteiden hakemiseksi.

Seuraava vaihe on luoda tarkistajia pankin eri toiminnallisuuksia varten. Tähän toteutukseen tarpeellista on tarkistaa, mihin luottoluokitukseen asiakas kuuluu

sekä erilaisia tunnuslukuja ja salasanoja. Luodaan edellä mainittua varten predicate-funktioita:

```
Predicate<Asiakas> eiLuottoa = x -> x.getPalkka() < 1000;
Predicate<Asiakas> matalaLuotto =
 x -> x.getPalkka() >= 1000 && x.getPalkka() < 2500;
Predicate<Asiakas> keskiLuotto =
x -> x.getPalkka() < 4000 && x.getPalkka() >= 2500;
Predicate<Asiakas> korkeaLuotto =
x -> x.getPalkka() >= 4000;
BiPredicate<String,String> tarkistaTunnus = (x, y) -> x.equals(y);
BiPredicate<Double,Double> vertaaPienempiKuin = (x,y) -> x <= y;
```

Esimerkkikoodi 44. Määritellään predicate-funktiot ohjelmassa esiintyviä metodeja varten.

Kuten esimerkkikoodista 43 ilmenee, predicate-funktiot tarkistavat asiakasolion sisältämän palkkamuuttujan arvoa. BiPredicate-funktiota käytetään tarkistamaan vastaako syöte olemassa olevaa salasanaa.

Määritellään seuraavaksi funktiot, joita pankki tulee käyttämään. Tässä toteutuksessa funktiot auttavat hakemaan listoja asiakkaista sekä yksittäisiä asiakastietoja.

```
Function<List<Asiakas>, List<Asiakas>> haeEiLuottoaLista =
x -> x.stream().filter(eiLuottoa).collect(Collectors.toList());
Function<List<Asiakas>, List<Asiakas>> haeMatalaLuottoLista =
x -> x.stream().filter(matalaLuotto).collect(Collectors.toList());
Function<List<Asiakas>, List<Asiakas>> haeKeskiLuottoLista =
x -> x.stream().filter(keskiLuotto).collect(Collectors.toList());
Function<List<Asiakas>, List<Asiakas>> haeKorkeaLuottoLista =
x -> x.stream().filter(korkeaLuotto).collect(Collectors.toList());
Function<Asiakas, Double> haeSaldo = x -> x.getTili().getSaldo();
Function<Asiakas, String> haeKäyttäjätunnus = x -> x.getTili().getTun-
nus();
Function<Asiakas,String> haeSalasana = x -> x.getTili().getSalasana();
```

Esimerkkikoodi 45. Ohjelmaan määritetyt funktiot. Ensimmäiset neljä funktiota määrittävät vuon asiakaslistasta ja suodattavat asiakkaat listaksi aiemmin määritettyjen predicate-funktioiden perusteella. Kaksi viimeistä funktiota ovat käytännöllisiä koodin lyhentämiseen metodeissa, joita pankki käyttää. Ne hakevat asiakkaan tilin saldon sekä käyttäjätunnukset.

Viimeisenä funktiotyyppinä määritetään pankille tarpeelliset consumer-funktiot. Tässä toteutuksessa niitä käytetään asiakkaan tilin saldon manipuloimiseen funktionaalisesti:

```
Consumer<Asiakas> laskeKorko =
x -> x.getTili().setSaldo(korko*haeSaldo.apply(x));
Consumer<List<Asiakas>> laskeKorkoList = x -> x.forEach(laskeKorko);
BiConsumer<Asiakas,Double> lisaaRahaa =
(x,y) -> x.getTili().setSaldo(x.getTili().getSaldo() + y);
```

Esimerkkikoodi 46. Määritetään ohjelmassa käytetyt consumer-funktiot.

Kuten esimerkkikoodissa 46 ilmenee, laskeKorko-funktion avulla yksittäisen tilin saldo kerrotaan muuttumattomalla liukuluvulla korko. laskeKorkoList-funktio suorittaa saman, mutta kohdistaa toiminnon listaan asiakkaista. lisaaRahaa-funktio lisää nimensä mukaisesti rahaa yksittäiselle tilille. Edellä mainittujen funktioiden lisäksi FunktioToteuttaja-luokka sisältää kaksi apumetodia, jotka hyödyntävät funktioita asiakkaiden luottoluokituksen määrittelemistä varten. Ymmärtääksemme ensimmäistä metodia meidän pitää katsoa siihen sidoksissa olevia funktioita, joita varten luotiin myös oma rajapinta:

```
@FunctionalInterface
interface iConsumer<A,B,C> {
    void accept(A a, B b, C c);
}
```

Esimerkkikoodi 47. iConsumer-rajapinnan määritys. rajapinnassa määritellään accept()-metodi, joka ottaa kolme argumenttia vastaan.

Seuraavaksi määriteltiin funktiot, joita tarvitaan metodin toiminnallisuutta varten, esitellään metodit esimerkkikoodissa 48:

```
    private final String x = "Ei luottoa";
    private final String y = "Matala luotto";
    private final String z = "Keskiluotto";
    private final String w = "Iso luotto";
    Consumer<Asiakas> asetaEiLuottoa = a -> a.setLuottoLuokitus(x);
    Consumer<Asiakas> asetaMatalaLuotto = a -> a.setLuottoLuokitus(y);
    Consumer<Asiakas> asetaKeskiLuotto = a -> a.setLuottoLuokitus(z);
    Consumer<Asiakas> asetaKorkeaLuotto = a -> a.setLuottoLuokitus(w);

    iConsumer<List<Asiakas>, Predicate, Consumer> TriConsumer =
    (x,y,z) -> {
              x.stream()
                      .filter(y)
                      .forEach(z);
    };
```

Esimerkkikoodi 48. Consumer-funktiot, joita metodi käyttää.

Kuten esimerkkikoodista 48 ilmenee, TriConsumer-funktio on iConsumer rajapinnan toteuttaja, joka hyväksyy syötteeksi listan asiakkaista, predikaattifunktion sekä consumer-funktion. Muut consumer-funktiot asettavat Asiakas-olion luottoluokitus-muuttujalle arvon niiden yläpuolella määriteltyjen String-olioiden mukaan. Seuraavassa esimerkkikoodissa esitellään asetaLuottoLuokitus()-metodi:

```
public List<Asiakas> asetaLuottoLuokitus(List<Asiakas> l){
        for(int i =0; i < 4; i++) {
            switch (i){
                case 0:
                    TriConsumer.accept(l,eiLuottoa,asetaEiLuottoa);
                    break;
                case 1:
                    TriConsumer.accept(
                    l,matalaLuotto,asetaMatalaLuotto);
                    break;
                case 2:
                    TriConsumer.accept(
                    l,keskiLuotto,asetaKeskiLuotto);
                    break;
                case 3:
                    TriConsumer.accept(
                    l,korkeaLuotto,asetaKorkeaLuotto);
                    break;
            }
        }
        return l;
    }
```

Esimerkkikoodi 49. asetaLuottoLuokitus()-metodin toteutus.

Kuten esimerkkikoodista 49 havaitaan, asetaLuottoLuokitus()-metodi käy silmukassa kaikki asiakkaat läpi ja asettaa jokaiselle asiakkaalle luottoluokitusarvon.

switch-case-rakenteessa käytetty TriConsumer-funktio määrittää jokaiselle Asiakas-oliolle yhden neljästä mahdollisesta luottoluokituksesta.

Kasittelija-luokka sisältää kaikki asiakkaan käyttämät metodit. Jokainen metodi hyödyntää jollain tavoin FunktionToteuttaja-luokkaa. Käydään esimerkkinä läpi lainan hakuun määritelty metodi haeLainaa:

```
public void haeLaina(Asiakas a){
            double laina = f.haeLuotonMäärä(a);
            System.out.println(f.supplier9.get());
            double lainaAnomus = Double.parseDouble(sc.nextLine());
            if(f.vertaaPienempiKuin.test(lainaAnomus,laina)){
                System.out.println(f.supplier10.get() +
                f.haeTimeStamp.get());
                f.lisaaRahaa.accept(a,lainaAnomus);
                System.out.println(f.supplier7.get() +
                f.haeSaldo.apply(a) + f.supplier8.get());
                }else{
                System.out.println(f.supplier11.get() +
                laina + f.supplier8.get());
                haeLaina(a);
            }
        }
```

Esimerkkikoodi 50. haeLainaa()-metodin toteutus. Metodi ottaa argumentikseen lainaa hakevan asiakkaan. Ohjelma kysyy asiakkaalta, kuinka suuresta summasta on kyse. Tarkistaa, onko kyseinen summa mahdollinen asiakkaan luottoluokituksen perusteella ja lisää lainan tilille, jos ehto toteutuu.

Kuten esimerkkikoodista 50 ilmenee, metodi pystytään toteuttamaan pelkillä funktioilla lyhentäen ja selkeyttäen koodia huomattavasti. haeLaina()-metodin lisäksi asiakkaalle tarjotaan mahdollisuus nostaa rahaa, ja tallettaa rahaa. Kirjaudu()-metodilla asiakas pääsee käsiksi pankin palveluihin. Toteutus on seuraava.

```
public boolean kirjaudu(Asiakas a) {
        String okt = f.haeKäyttäjätunnus.apply(a);
        String oss = f.haeSalasana.apply(a);
        boolean isCorrect = false;
        int yritykset = 0;
        while (yritykset < 3 && !isCorrect) {
            System.out.println(f.supplier1.get());
            String kt = sc.nextLine();
            System.out.println(f.supplier2.get());
            String ss = sc.nextLine();
            if (f.tarkistaTunnus.test(kt,okt) &&
            f.tarkistaTunnus.test(ss,oss)) {
                System.out.println(f.supplier3.get() + a.getNimi());
                isCorrect = true;
                }else{
                System.out.println(f.supplier4.get());
                yritykset++;
        }
}
```

Esimerkkikoodi 51. Kirjaudu()-metodin toteutus

Kuten esimerkkikoodista 51 ilmenee, voidaan FunktioToteuttaja-luokan avulla toteuttaa sisäänkirjautumismetodi siistillä ja lyhyemmällä koodilla kuin normaalisti. Ohjelmalle pystytään nyt luomaan Pankki-luokassa run()-metodi ohjelman käynnistämiseksi.

```
public class Pankki{
    FunctionProvider f = new FunctionProvider();
    Käsittelijä k = new Käsittelijä();
    public void run(Asiakas a){

        System.out.println(f.supplier21.get());
        k.kirjaudu(a);
        k.valitseToiminto(a);

    }
```

Esimerkkikoodi 52. run()-metodin määrittäminen.

main()-metodissa riittää, kun luomme Asiakas-oliot ja lisäämme ne listaan, jotta voimme demonstroida pankin toimintoja. Lisäksi toteutetaan muutama vuo, joiden avulla pystytään hakemaan asiakastietoja funktionaalisesti:

```
public static void main(String[] args) {
        FunctionProvider f = new FunctionProvider();
        Pankki p = new Pankki();
        List<Asiakas> list = new ArrayList<>();

        //Luodaan tilit asiakkaille
        Tili t1 = new Tili(1,1000,"eka", "1234");
        Tili t2 = new Tili(2,2000,"toka", "1234");
        Tili t3 = new Tili(3,3000,"kolmas", "1234");
        Tili t4 = new Tili(4,4000,"neljäs", "1234");
        Tili t5 = new Tili(5,5000,"viides", "1234");

        //Luodaan asiakkaat
        Asiakas a1 = new Asiakas("Kalle K" , 1, 2500,null,t1);
        Asiakas a2 = new Asiakas("Lasse L" , 2, 4600,null,t2);
        Asiakas a3 = new Asiakas("Maija M" , 3, 900,null,t3);
        Asiakas a4 = new Asiakas("Eero E" , 4, 10000,null,t4);
        Asiakas a5 = new Asiakas("Ossi O" , 5, 3400,null,t5);

        //Lisätään asiakkaat listaan
        list.addAll(Arrays.asList(a1,a2,a3,a4,a5));

        //Assetaan jokaiselle asiakkaalle luottoluokitus
        f.asetaLuottoLuokitus(list);

        p.run(a1); //a1 pääsee kirjautumaan tililleen


        //Stream esimerkkejä
        list.stream()
           //funktio joka palauttaa  String-oliota
           .map(f.haeKäyttäjätunnukset)
           //Consumer-funktio String tulostamiseen
           .forEach(f.printtaaTiedot);


         List<Asiakas> list2 = list.stream()
           .filter(f.keskiLuotto) //predicate-funktio suodatukseen
           .collect(Collectors.toList()); //Lisätään asiakkaat listaan
            System.out.println(list2); //[Kalle k, Ossi O]

         //Annetaan uuden listan asiakkaille korkoa tilille
         f.annaKorko.accept(list2);

          Double summa = list.stream()
               //Muutetaan asiakkaat liukuluvuiksi
               .map(x -> x.getTili().getSaldo())
               //Suodatetaan tilit joissa alle 5000 euroa
               .filter(x -> x < 5000)
               //haetaan tilien saldojen summa
               .reduce(0.0, (x,y) -> x+y);

                System.out.println(summa); //10000

}
```

Esimerkkikoodi 53. Toteutuksen main()-metodin toiminta.

Kuten esimerkkikoodista 53 voidaan havaita, ohjelma tarvitsee toimiakseen listan, Pankki-luokan run()-metodia toteuttavan olion sekä FunktionProvider-luokan funktioita toteuttavan olion. Lopuksi ohjelma havainnollistaa stream-toteutuksien toimintaa dokumentoituna.

5 Yhteenveto

Työn tavoitteena oli saada kattava käsitys funktionaalisesta ohjelmoinnista Javalla olio-ohjelmoijan näkökulmasta. Lisäksi työn tavoitteena oli tuoda esiin, miten olio-ohjelmointia ja funktionaalista ohjelmointia voidaan toteuttaa rinnakkain.

Työssä käsiteltiin funktionaalisen ohjelmoinnin termejä, sääntöjä ja perusperiaatteita sekä teorian että esimerkkien avulla. Työssä käytiin kattavasti läpi Java 8:n mukana tulleet funktionaaliset rajapinnat, niiden metodit sekä niiden hyödyntäminen. Tämän lisäksi tutustuttiin kokoajiin ja mahdollisuuksiin hyödyntää niitä funktionaalisessa ohjelmoinnissa vuon yhteydessä.

Funktionaalinen ohjelmointi perustuu puhtaisiin funktioihin, datan muuttumattomuuteen, rekursioon, funktioiden käsittelyyn ensimmäisen luokan kansalaisina sekä funktioiden ketjuttamiseen. Lisäksi funktionaalisessa ohjelmoinnissa listat käydään läpi map-, filter- ja reduce-metodien avulla.

Java ei ole puhtaasti funktionaalinen kieli, vaan se imitoi funktionaalisuutta. Muuttujat eivät saavuta niin sanottua todellista muuttumattomuutta kuten esimerkiksi Haskel-kielessä. Tästä huolimatta Java soveltuu funktionaaliseen ohjelmointiin, koska Java 8 tarjoaa perusperiaatteiden toteuttamiseen tarpeeksi hyvät työkalut. Java tarjoaa myös mahdollisuuden toteuttaa omia funktionaalisia rajapintoja tarpeen mukaan (esimerkiksi kolmen argumentin funktio).

Voidaan sanoa, että Java on nykyisin hybridi funktionaalisen ja olio-ohjelmoinnin välillä. Olio-ohjelmoijan pitäisi tulevaisuudessa hallita myös deklaratiivinen ohjelmointi, sillä se on korvaamaton apuväline isojen datamäärien prosessoimiseen,

koodin selkeyttämiseen, sivuvaikutusten välttämiseen sekä koodin uudelleen-käytettävyyteen.

Lähteet

1 A tutorial introduction to the lambda calculus. Verkkoaineisto. Saatavissa: https://personal.utdallas.edu/~gupta/courses/apl/lambda.pdf. Luettu 27.7.2021.

2 Java Lambda Expressions: Functions as First-Class Citizens, Yogen Rai. Verkkoaineisto. Saatavissa: https://dzone.com/articles/java-lambda-expressions-functions-as-first-class-citizens. Luettu 8.8.2021.

3 Higher order functions (composing software), Eric Elliot. Verkkoaineisto. Saatavissa: https://medium.com/javascript-scene/higher-order-functions-composing-software-5365cf2cbe99. Luettu 20.8.2021.

4 Function as First Class Citizen in Java, Rai Skumar. Verkkoaineisto. Saatavissa: http://geekrai.blogspot.com/2017/01/functional-programming-in-java-function.html. Luettu 1.9.2021.

5 Java Curry: A demonstration of currying in the Java Programming language, Vijay Lakshiminarayanan. Verkkoaineisto. Saatavissa: https://medium.com/galileo-onwards/java-curry-997fb357b47e. Luettu 16.9.2021.

6 Immutability in Java, Dave Nicolette. Verkkoaineisto. Saatavissa: https://www.leadingagile.com/2018/03/immutability-in-java/. Luettu 28.9.2021.

7 Functional programming in Java: Harnessing the power of Java 8 Lambda Expressions, Venkat Subramaniam. Kirja. Saatavissa: http://www.r-5.org/files/books/computers/languages/java/style/Venkat_Subramaniam-Functional_Programming_in_Java-EN.pdf. Luettu 3.10.2021.

8 Java 8 identity function, Satish Varma. Verkkoaineisto. Saatavissa: https://javabydeveloper.com/java-8-identity-function-examples/. Luettu 9.10.2021.

9 Java Platform, Standard Edition (Java SE) 8. Verkkoaineisto. Saatavissa: https://docs.oracle.com/javase/8/. Luettu 13.10.2021.