Java 8 ja Stream API

Torikka, Ilja

2016 Laurea

Laurea-ammattikorkeakoulu

Java 8 ja Stream API

Torikka Ilja
Tietojenkäsittelyn koulutusohjelma
Opinnäytetyö
Huhtikuu, 2016

Laurea-ammattikorkeakoulu **Tiivistelmä**
Tietojenkäsittelyn koulutusohjelma

Torikka Ilja

Java 8 ja Stream API

Vuosi 2016 sivut 23

Tämän opinnäytetyön tarkoitus on esitellä yksityiskohtaisemmin Java 8:n mukanaan tuomaa Stream API:a. Uusi java.util.stream-paketti lisättiin Javan standardikirjastoon, joka mahdollistaa erilaisten operaatioiden suorittamisen kokoelmissa. Käyn työssä läpi miten, miten kokoelmia voi suodattaa ja supistaa Stream API:n avulla.

Käytän lähdekoodeja esimerkkeinä, kun näytän uutta tapaa käydä läpi kokoelmia. Streamit on suunniteltu toimimaan yhdessä lambda-lausekkeiden kanssa, joita käytän paljon esimerkeissä. Lambda-lausekkeiden käyttö on olennainen osa Stream API:n käyttöä.

Stream API:n tuoma muutos Javaan on niin suuri, että se itsestään vaatii jo uuden ohjelmointiparadigman opiskelua. Lopputuloksena on kuitenkin lyhyempi, loogisempi ja helpommin ylläpidettävä lähdekoodi.

Asiasanat: java, stream, lambda

Laurea University of Applied Sciences **Abstract**
Degree Programme in Business Information Technology

Torikka Ilja

Java 8 and Stream API

Year 2015 pages 23

The idea of this thesis is to explain and to clarify Stream API . New java.util.stream package was added to the Java standard library, which allows you to do different kind of operations in the collections. I 'll give you examples how you can filter and reduce collections by using the Stream API.

I use the source code examples, when I show new ways to go through the collections. Stream is designed to work with lambda expressions which I use a lot in examples. Using lambdas is an essential part of the Stream API usage.

The changes which Stream API brings to Java is so large that it requires a self-study for a new programming paradigm. However, the end result is a shorter, more logical and easier to maintain source code.

Asiasanat: java, stream, lambda

Sisällys

1. Johdanto

Opinnäytetyön sisältö tarkoitettu henkilöille joilla on jo perustaidot Java SE ohjelmoinnista. En käy läpi Javan perusteita tai olio-ohjelmoinnin alkeita olenkaan, eli jos et aikaisemmin ole ohjelmoinut Javalla, niin et välttämättä saa kovinkaan paljon irti esimerkki koodeista.

Sen sijaan jos jo olet aikaisemmin tutustunut Javaan ja ehkäpä jopa Java SE:n 8 versioon, niin näillä lukijoille pyrin tällä opinnäytetyöllä omalta osaltani selventämään Stream API:ta.

Lähes jokainen sivu sisältää lähdekoodi esimerkkejä, joiden avulla näytän Stream API:n ja lambda-lausekkeiden käyttöä. Ilman lähdekoodien käyttöä näiden kahden Java kieleen tulleen lisäominaisuuksien selvittäminen ei olisi järkevää. Kuten jo aikaisemmin mainitsin lambda-lausekkeiden ja Stream API:n käyttö vaatii uuden ohjelmointi-paradigman sisäistämistä.

Lähes jokaisen ohjelmointikielen parissa tulet olemaan tekemisissä kokoelmien kanssa. Niiden avulla tallennat, muokkaat ja haet dataa. Koodasit sitten pientä tai suurempaa ohjelmaa tulet käyttämään kokoelmia (List,Set). Vaikka puhutaankin näin olennaisesta asiasta niin esimerkiksi Javalla saattaa perusoperaatiot kokoelmien kanssa käydä yllättävän turhauttavaksi. Ennen Stream API:ta käytiin kokoelmia läpi niin sanotusti ulkoisella iteroinnilla esim. for loopin avulla. Tämä niin sanottu imperatiivinen eli käskevä tapa ohjelmoida luo usein turhan paljon koodia jopa yksinkertaiseen toimenpiteeseen. Mitä enemmän lähdekoodia tulee, sitä vaikeaselkoisempaa on sen luettavuus, ylläpito ja testaus.

Sen sijaan että kerrotaan miten lopputulos halutaan niin kerrottaan mitä halutaan lopputulokseksi ja annetaan Javan tehdä lopputyö. Tiedon hakeminen kokoelmista Stream API:ta käyttäen muistuttaakin enemmän tiedon hakua tietokannoista SQL-kielellä. Siirrytään ulkoisesta iteroinnista, sisäiseen iterointiin. Käskevästä tavasta siirrytään selittävään tapaan.

Stream API:n avulla on mahdollista toteuttaa tiedon haku kokoelmista joko peräkkäin tai rinnakkain. Varsinkin rinnakkaiset operaatiot mahdollistavat nopeamman lopputuloksen, kun työskennellään suurien data määrien kanssa. Rinnakkaiset operaatiot mahdollistavat myös käyttämään täyden hyödyn moniydinprosessoreista tietokoneissa. Rinnakkaisuuden käyttö on kuitenkin tehty todella helpoksi Stream API:ssa. Tätä varten ei tarvitse kirjoittaa monimutkaista koodia säikeitä(thread) varten.

2. Mikä on “stream”

Stream on eräänlainen putki(stream pipeline), joka luodaan jostain lähteestä esim. kokoelma tai I/O kanava. Tällä datavirtaa sisältävällä putkella on aina alku ja loppu. Tätä kyseistä datavirtaa voidaan muokata halutun kaltaiseksi lopputulokseksi välioperaatioilla (intermediate operation) ja sen jälkeen koota lopputulos yhteen pääteoperaatiolla (terminal operation). Pääteoperaatio pysäyttää aina datavirran ja palauttaa halutun lopputuloksen. Stream ei vaikuta millään tavalla alkuperäisen lähteen sisältöön.

Kokoelman ja Streamin erona on se, että kokoelman kaikki sisältö on laskettava ja laitettava muistiin, ennen kuin se voidaan lisätä kokoelmaan. Stream ei säilytä tietoja missään tietorakenteessa, kuten esim. kokoelmat. Streamin lähteenä on tietorakenne, jota se voi sitten muokata halutulla tavalla.

Kuten aiemmin sanoin lähteestä tulevaa datavirtaa voidaan muokata välioperaatioilla. Välioperaatio palauttaa aina uuden “streamin”, joten välioperaatioita voidaan linkittää toisiinsa. Välioperaatiot ovat niin sanotusti laiskoja (lazy), eli ne eivät suorita mitään laskelmia ennen kuin datavirralle annetaan pääteoperaatio, jotka ovat päinvastaisesti ahneita(eager).

Kokoelmien kautta luotava stream, käyttäen stream() tai parallelstream() metodia.

```
List<Integer> numerot = Arrays.asList(1,2,3,4,5,6);
numerot.stream().filter( n -> n % 2 == 0)
        .forEach( System.out::print ); // tulostaa luvut 246
```

Listaan voidaan tulostaa myös suoraan ilman streamiä.

```
numerot.forEach( System.out::print ); // tulostaa luvut 123456
```

Arraysta luotu stream, käyttäen Arrays.stream(Object[]).

```
int[] numerot = {1,2,3,4,5,6};
int summa = Arrays.stream( numerot ).sum(); // tulostaa luvun 21
```

Stream luokan kautta luodut staattiset metodit kuten esim. IntStream.range(int,int).

```
IntStream.range(1, 4)
         .forEach( System.out::print ); // tulostaa luvut 123
```

Tiedostosta voidaan hakea rivejä BufferedReader.lines() metodin kautta.

```
String tiedosto = "c://numerot.txt";
try (Stream<String> stream = Files.lines(Paths.get( tiedosto )) ) {
stream.forEach( System.out::println );
}

catch (IOException e) {
e.printStackTrace();}
```

Satunnaisia numeroita käyttäen Random.ints() metodia.

```
Random rand = new Random();
IntStream is = rand.ints(0, 100 );
is.distinct()
  .filter( num -> num % 2 == 0)
  .limit(5)
  .forEach( System.out::println );
// tulostaa viisi eri parillista numeroa väliltä 0-99
```

3 Välioperaatiot

3.1 Stream.filter()

Voit käyttää filter() metodia datan suodattamiseen. Parametriksi filter() metodille annetaan Predicate-olio, tämän jälkeen lambda-lauseketta käyttäen syötetään haluttu suodatus. Predicate rajapinta sisältää metodin nimeltään test(), joka ottaa vastaan yhden parametrin ja palauttaa boolean arvon. Eli, kun lambda-

lauseke syötetään filter() metodin sisään toteuttaa se Predicate.test() metodin. Kyseinen metodi on määritely seuraavanlaisesti boolean test(T t).

```
List<String> oppilaat = new ArrayList<>();
oppilaat.add("Anna");
oppilaat.add("Heikki");
oppilaat.add("Antti");
oppilaat.add("Laura");
oppilaat.stream()
        .filter(nimi -> nimi.startsWith( "A" ))
        .forEach( System.out::println ); // Tulostaa nimet Anna ja Antti
```

3.2 Stream.map()

Jos haluat muuttaa lähteestä saatua datavirtaa, kuten esim. numeroiden arvoa tai kirjainten kokoa jne., voit käyttää map() metodia. Parametriksi annat map() metodille Function-olion, joka palautta muunnetun datavirran. Kuten aikaisemmin kerroin alkuperäisen lähteen arvoihin streamit eivät vaikuta millään tavoin. Voit myös palauttaa tiettyä datatyypin käyttämällä esim. mapTodouble()- ja mapToInt() metodia.

```
List<Integer> numerot = new ArrayList<>();
numerot.add(1);
numerot.add(2);
numerot.add(3);
numerot.add(4);
numerot.stream().map( num -> num * 2 ).forEach( System.out::print ); // 2468
```

Ylempi koodi hakee datavirrasta numerot ja kertoo ne kahdella. Alempi koodi suodattaa ensiksi pois parilliset numerot ja sen jälkeen muuttaa luvut liukuluvuiksi(double) käyttämällä mapToDouble() metodia. Lopuksi jaetaan jäljellä olevat luvut kahdella.

```
List<Integer> numerot = new ArrayList<>();
numerot.add(1);
numerot.add(2);
numerot.add(3);
numerot.add(4);
numerot.stream() .filter( num -> num % 2 != 0 )
        .mapToDouble( num -> num )
        .map( num -> num/ 2 ) .forEach( System.out::print );
```

3.3 Stream.flatMap()

Voita halutessa muuttaa kahdesta eri lähteestä tehdyn streamin, yhdeksi peräkkäiseksi streamiksi flatMap() metodin avulla. Alla olevassa koodissa tehdään ensiksi kahdesta listasta, kaksi eri datavirtaa Stream.of() metodia käyttäen. Sen jälkeen muutetaan nämä datavirrat yhdeksi kokonaiseksi streamiksi flatMap() metodilla.

```
List<String> nimet1 = Arrays.asList( "Toni", "Tarja" );
List<String>nimet2 = Arrays.asList( "Jussi", "Laura" );
Stream.of(nimet1, nimet2)
        .flatMap(nimi -> nimi.stream() )
        .forEach(nimi -> System.out.print(nimi + " ") );
```

Tehdäänpä sama käyttäen flatMap() metodin sijaan, pelkkää map() metodia. Alla oleva koodi tulostaa [Toni, Tarja] [Jussi, Laura].

```
List<String> nimet1 = Arrays.asList( "Toni","Tarja" );
List<String>nimet2 = Arrays.asList( "Jussi","Laura" );
Stream.of(nimet1, nimet2)
        .map(nimi -> nimi)
        .forEach(nimi -> System.out.print(nimi) );
```

3.4 Stream.sorted()

Kun haluat tulostaa listan sisältävien objektien tietoja tietyssä järjestyksessä, voit käyttää sorted() metodia. Tässä on yksi tapa tulostaa työntekijöiden palkat suuruus järjestyksessä käyttäen sorted() metodin sisällä lambda-lauseketta.

```
List<Tyontekija> tt = new ArrayList<>();
tt.add(new TyonTekija("Jussi J", 15000.00) );
tt.add(new TyonTekija("Heikki H", 35000.00) );
tt.add(new TyonTekija("Toni T", 25000.00) );
tt.stream()
  .sorted( (t1, t2) -> Double.compare( t2.getPalkka(), t1.getPalkka() ) )
  .forEach( System.out::println ); // Tulostus järjestys on Heikki,Toni,Jussi
```

Tulostetaan työntekijät etunimen perusteella käyttämällä samaa sorted() metodia.

```
List<Tyontekija> tt = new ArrayList<>();
tt.add(new TyonTekija("Jussi J", 15000.00) );
tt.add(new TyonTekija("Heikki H", 35000.00) );
tt.add(new TyonTekija("Toni T", 25000.00) );
tt.stream() .sorted( Comparator.comparing( Tyontekija::getNimi ) )
                .forEach( System.out::println ); // Tulostus järjestys on Heik-
ki,Jussi,Toni
```

4. Pääteoperaatiot

4.1 Reduce-operaatiot

Käytä reduce operaatioita, kun sinulla on esim. kokoelma arvoja ja haluat palauttaa näistä yhden arvon. Voit käyttää suoraan reduce() metodia tai sitten muita tiettyyn tarkoitukseen tehtyjä reduce-operaatioita, joita ovat average(), sum(), min() ja max(), jotka käyttävät IntStreamiä ja DoubleStremiä. Metodin elementtien lukumäärän palauttamiseen on count() metodi, joka palauttaa long arvon.
Käytetään ensimmäiseksi suoraan reduce() metodia. Aluksi metodille annetaan alkuarvo ja sen jälkeen tehdään haluttu toimenpide datavirrassa oleville arvoille.

```
double palkat = lista.stream() // lista on ArrayList<Tyontekija>.
                        .map( Tyontekija::getPalkka )
                        .reduce(0.0, ( p1,p2 ) -> p1 + p2);
                        System.out.print( palkat );
```

Sama reduce-operaatio suoritettuna sum() metodilla, sitä ennen muutetaan datavirtaa mapToDouble() metodilla.

```
double palkat = lista.stream() // lista on ArrayList<Tyontekija>
                        .mapToDouble( Tyontekija::getPalkka )
                        .sum();  System.out.print(palkat);
```

Lasketaanpa palkkojen keskiarvo. Sen saa käyttämällä average() metodia, jonka jälkeen pyydetään vielä getAsDouble() metodia. Syy tähän on se, että average() metodi palauttaa OptionalDouble arvon, joka varautuu siihen, että datavirta on tyhjä.

```
double palkat = lista.stream()
                        .mapToDouble( Tyontekija::getPalkka )
                         .average()
                         .getAsDouble();
```

Ilman getAsDouble() metodia koodi näyttäisi seuraavalta.

```
OptionaDouble palkat = lista.stream()
                                .mapToDouble( Tyontekija::getPalkka )
                                .average(); // OptionalDouble[keskiarvon];
```

Haetaan suurin palkka ja pienin palkka käyttäen max()- ja min() metodeita.

```
Tyontekija palkat1 = lista.stream()
                            .max((p1, p2)->Double.compare( p1.getP(), p2.getP())
                             .get(); // tulostaa henkilön, kenellä on isoin palkka
System.out.println( palkat1 );
```

Etsitään pienin palkka, mutta hiukan eri tavalla.

```
final Comparator<Tyontekija>palkat2=(p1,p2) -> Doub-
le.compare(p1.getPalkka(),p2.getPalkka());
Tyontekija minPalkka = lista.stream()
                                .min( palkat2 )
                                .get(); System.out.println( minPalkka );
```

Työntekijöiden määrän laskeminen count() metodia käyttäen.

```
long luku = lista.stream().count();
System.out.println( luku );
```

4.2 Stream.collect()

Tämä pääteoperaation avulla on mahdollista muuttaa lähteestä saatu datavirta kokoelmaksi(List,Set,Map). Tätä varten on olemassa Collectors luokka, jota käytetään collect() metodin sisällä.

```
int[] num = {3,2,4,5,1,5};
Set<Integer> numerot = Arrays.stream(num)
                                    .boxed()
                                    .collect( Collectors.toSet() ); //[1,2,3,4,5]
                                    System.out.println(numerot);
```

Yllä olevassa koodissa arrayn datavirtaa käytetään, kun luodaan Set kokoelma. Alempana tehdään lista, joka sisältää vain "a" merkit.

```
List<String> kirjain = new ArrayList<>();

kirjain.add("a");
kirjain.add("d");
kirjain.add("b");
kirjain.add("a");
List a = kirjain.stream()
            .filter( k -> k.startsWith("a") ).collect( Collectors.toList() ); //[a, a]
```

Kolmannessa koodissa kerrotaan täsmälleen, että halutaan luoda TreeSet, käyttäen toCollection() metodia.

```
Set puu = kirjain.stream().collect(Collectors.toCollection( TreeSet::new) );
System.out.println( puu );
```

4.3 Stream.forEach()

Jokainen Java-ohjelmoija on kirjoittanut perinteisen for loopin satoja kertoja. Tämä kyseinen koodi on olennainen osa ohjelmoinnin perusteita ja ollut Javan mukana jo ensimmäisestä versiosta alkaen.

Javan 1.5 versio toi mukanaan advanced for loopin, joka sai näyttämään vanhan tutun perinteisen for loopin vanhanaikaiselta ja virhe herkältä, kun käytiin läpi vaikka kokoelmia ja arrayta. Nämä molemmat tavat ovat kuitenkin imperatiivisia tapoja eli käskeviä tapoja käydä esim. kokoelmaa läpi. Käytämme näissä loopeissa niin sanottua ulkoista iterointia, jossa kerromme miten ja mitä haluamme lopputulokseksi. jälkimmäinen advanced for loop tekee sen Iterator rajapinnan hasNext() ja next() metodeilla.

Javan 1.8 versio toi Stream API:n forEach() metodin, jonka avulla voimme käyttää sisäistä iterointia. Näin voimme keskittyä pelkästään siihen mitä haluamme lopputulokseksi ja antaa Java kirjastojen tehdä lopputyö.

```
List<String> kaupungit = Arrays.asList("Kerava","Helsinki","Turku","Lahti");
```

Perinteinen for loop.

```
for(int i = 0; i < kaupungit.size(); i++){
System.out.println( kaupungit.get(i) );}
```

Java 1.5 version mukana tullut advanced for loop.

```
for(String kaupunki : kaupungit){
System.out.println(kaupunki);}
```

Java 1.8 versio, joka toi mahdollisuuden käyttä Stream API:ta ja lambda-lausekkeita.

```
kaupungit.forEach( System.out::println );
```

4.4 Stream.findFirst() ja findAny()

Short-circuit metodin käyttäminen pääteoperaationa tarkoittaa sitä, että kaikkea datavirran sisältöä ei tarvitse välttämättä käydä läpi, ennen kuin virta lopetetaan. Näytän esimerkit findFirst() ja findAny() metodeilla. Molemmissa tapauksissa virta lopetetaan, kun ehto toteutuu. Toki findAny() metodi ei välttämättä palauta ensimmäistä kohdetta.

```
List<String> nimet = Arrays.asList("Antti","Satu","Sami","Heikki");
Stream<String> nimi = nimet.stream();
Optional<String> eka = nimi.filter( n -> n.startsWith("S")).findFirst();
if(eka.isPresent() ){
System.out.println( eka.get() );}
```

4.5 Stream.allMatch(), anyMatch() ja noneMatch()

Nämä short-circuit metodit ottavat vastaan Predicate-olion ja palauttavat boolean arvon.

```
Stream<Integer> numerot = Stream.of(1,2,3,4,5);
boolean ehto = numerot.allMatch( n -> n < 6 );// ehto saa arvokseen true.
```

4.6 parallelStream()

Nykypäivänä lähes kaikki tietokoneet alkavat olemaan prosessoreiden suhteen moniytimisiä, kun prosessorit ovat moniytimisiä niin tämän avulla voidaan tehdä rinnakkaisohjelmointia tehokkaasti. Ideana on, että ohjelma jakaa tehtäviä eri ytimille ja ne suorittavat tehtäviä samanaikaisesti.
Java 8:ssa voit käyttää Collection.parallelStream()- tai BaseStream.parallel() metodeita rinnakkaisuuden saavuttamiseen. Molemmat metodit käyttävät Java 7 version mukana tullutta Fork/Join rajapintaa siihen, että voidaan jakaa operaation suoritus eri prosessorien ytimille.
Tuettavat tietorakenteet rinnakkaisuuden suorittamiseen ovat IntStream, DoubleStream, LongStream sekä Stream<T>. Jos työskentelet Stream-kokoelmien kanssa ja haluat varmistaa onko kokoelmaa mahdollista ajaa rinnakkain, niin voit tehdä sen isParallel() metodin avulla.Rinnakkaisuuden suorittamiseen ja hallintaan Java 8:ssa

päästään hyvin pienillä muutoksilla koodi tasolla. Kannattaa kuitenkin muistaa, että useissa tapauksissa peräkkäin suoritus on kuitenkin riittävä nopeuden suhteen.

Tapaukset missä on syytä pohtia rinnakkaisuuden käyttöä ovat silloin, kun kokoelmassa on erittäin paljon käsiteltäviä elementtejä tai yksittäisen elementin käsittely vie paljon aikaa.

Myös tilanteet missä jostain muusta syystä kokoelman käsittelyssä ilmenee suorituskyky ongelmia, niin kannattaa kokeilla auttaako parallelStream() suorituskykyä.Näytän seuraavassa esimerkissä parillisten numeroiden tulostuksen väliltä 1-12, käyttämällä stream() ja parallelStream() metodeita. Käyttämällä stream() metodia tuloksena on numerosarja 2 4 6 8 10 12, kun käytän parallelStream() metodia on tuloksena samat numerot, mutta järjestys on eri. Syy on se, että parallelStream() suorittaa "filtteroinnin" rinnakkain erillisinä sub-stremeinä ja tulostaa numerot sen

mukaan milloin kukin on valmis. Luonnollisesti näin pienen numero määrän suodattaminen on nopeampaa ja järkevämpää peräkkäin. Mielenkiintoista on se, että saan aina ensimmäiseksi numeroksi 8 ja sen jälkeen numeroiden järjestys vaihtelee per tulostus. Ilmeisesti numerot jaetaan kahteen eri "sub-streamiin" jotka ovat 1-6 ja 7-12.

```
List<Integer> numerot = Arrays.asList(1,2,3,4,5,6,7,8,9,10,11,12);
numerot.stream()
        .filter( n -> n % 2 == 0 )
        .forEach( System.out::print ); //tulostaa 2 4 6 8 10 12
numerot.parallelStream()
        .filter( n -> n % 2 == 0 )
        .forEach( System.out::print ); //tulostaa esim. 8 2 12 4 6 10
```

5 Tiedostojen käsittely

Files luokka esiteltiin ensimmäisen kerran Java 7 version mukana, joka oli osa Java NIO API:ta. Testataan Java 8 version Stream API:ta tiedostojen käsittelyssä.

```
String tiedosto = "numerot.txt"; // tiedostoon on kirjoitettu kolmelle riville: 10.0
20.0 ja 30.0
try (Stream<String> virta = Files.lines( Paths.get(tiedosto) )) {
String lue = virta.collect( Collectors.joining(", ") );
System.out.print(lue); // tulostaa 10.0, 20.0, 30.0
catch (IOException e) {e.printStackTrace();}
```

Edellinen koodi lukee tiedoston rivi riviltä ja lisää "," merkin ja välilyönnin jos kyseessä ei ole tiedoston viimeinen rivi. Alhaalla oleva koodi lukee tiedoston rivit ja jos rivi sisältää merkin "1", niin se tulostetaan.

```
String tiedosto = "numerot.txt"; // tiedostoon on kirjoitettu kolmelle riville: 10.0
20.0 ja 30.0
try( Stream<String> virta = Files.lines( Paths.get(tiedosto) )) {
virta.filter( line -> line.contains( "1" ) ).forEach( System.out::println ); }
catch (IOException e) {
e.printStackTrace();}
```

6. Miten lambdat muuttavat koodia

6.1 ActionListener

Ensimmäinen koodi on ilman lambda-lauseketta.

```
JButton button = new JButton("Tulosta");
button.addActionListener( new ActionListener() {
public void actionPerformed( ActionEvent evt ) {
System.out.println( "Terve!!" );
}});
```

Koska ActionListener rajapinta määrittelee vain yhden abstraktin metodin, niin silloin se on funktionaalinen rajapinta, joka taas mahdollistaa lambda-lausekkeen käytön. Tehdäänpä sama koodi lambda-lausekkeen kanssa.

```
JButton button = new JButton("Tulosta");
button.addActionListener( e -> System.out.println( "Terve" ) );
```

6.2 Runnable

Ilman lambda-lauseketta:

```
Runnable r = new Runnable() {
@Override
public void run() {
System.out.println( "Juokse" );
} };
```

Lambda-lausekkeen kanssa:

```
Runnable r = () -> System.out.println("Juokse");
```

6.3 Thread

Ilman lambda-lauseketta:

```
Thread t = new Thread(new Runnable() {
public void run () {System.out.println("Huomenta!");} } );
t.start();
```

Lambda-lausekkeen kanssa:

```
Thread t = new Thread(() -> System.out.println("Huomenta!") );
t.start();
```

7. Funktionaaliset rajapinnat

Javan 1.8 version API sisältää jo valmiiksi rakennettuja funktionaalisia rajapintoja. Aikaisemmista versioista tuttuja rajapintoja, kuten esim. Comparator ja Runnable. Molempiin on lisätty @FunctionalInterface annotaatio, jotta lambda-lausekkeiden käyttö olisi mahdollista. Jos rajapintaan lisätään kyseinen annotaatio, niin sen pitää täyttää funktionaalisen rajapinnan ehdot.
Funktionaalisessa rajapinnassa saa olla vain yksi abstrakti metodi, muuten rajapintaa ei pysty kääntämään. Java 1.8 version mukana tulleita oletusmetodeita saa rajapinnassa olla vaikka kuinka paljon mm. Comparator rajapinta sisältää yli kymmenen oletusmetodia.

Java 1.8:saan lisättiin myös uusia valmiiksi tehtyjä funktionaalisia rajapintoja, joista esittelen seuraavaksi osan. Voit myös tehdä itse funktionaalisia rajapintoja, lisäät rajapintaan vain @FunctionalInterface annotaation.

7.1 Predicate

Predicate-rajapinta sisältää yhden abstraktin test() metodin, joka palauttaa boolean arvon. Rajapinnassa on myös kaksi oletusmetodia nimeltään negate() ja or(), sekä staattinen isEqual().

```
String a = "abc";
String b = "abcd";
Predicate<String> pituus = s -> s.length() > 3;
boolean c = pituus.test(a); //false
boolean d = pituus.test(b); //true
```

7.2 Consumer

Consumer rajapinta sisältää accept() metodin, joka ottaa vastaan yhden argumentin, mutta ei palauta mitään(void).

```
Consumer<Oppilaat> tervehdi = (Oppilas o) -> System.out.println("Terve " + o.eNimi);
lista.forEach( tervehdi ); // lista on ArrayList, joka sisältää Oppilas luokan objektit.
```

7.3 Supplier

Supplier rajapinta toimii juuri päinvastoin kuin Consumer, se ei ota vastaan argumenttia ja palauttaa tuloksen. Supplier rajapinta sisältää get() metodin.

```
Supplier<Integer> tulos = () -> (int) ( Math.random()*10 );
System.out.println( tulos.get() ); // tulostaa luvun väliltä 1-9;
```

7.4 Function

Function<T,R> rajapinnan tarkoitus on ottaa vastaan argumentti tyyppiä T ja palauttaa sitten tulos haluttuna tyyppinä R. Tähän toimenpiteeseen käytetään rajapinna apply() metodia.

```
Function< String,Double > jako = n -> Double.valueOf(n);
System.out.println( jako.apply( "5" )/2 ); // tulostaa 2.5
```

7.5 UnaryOperator

Jos tiedetään, että argumentiksi annettava arvo pysyy samana, kun se tulostetaan. Voidaan käyttää UnaryOperatoria, Functionin sijaan.

```
UnaryOperator< String > koko = s -> s.toUpperCase();
System.out.println( koko.apply( "isot kirjaimet" )); // tulostaa ISOT KIRJAIMET
```

7.6 BiFunction

Tämän rajapinnan avulla on mahdollista käsitellä kahta argumenttia ja tulostaa niiden perusteella haluttu lopputulos.

```
CD cd1 = new CD(20,"CD1");
CD cd2 = new CD(30,"CD2");
BiFunction<CD,CD,Integer> hinta = (c1,c2) -> {
return c1.getPrice()+c2.getPrice(); };
int hinnat = hinta.apply(cd1, cd2);
System.out.println(hinnat);
```

8 Unit Testing

Lambda-lausekkeen yksikkötestauksessa on omat haasteensa, koska niillä ei ole nimeä ja tämän takia niiden kutsuminen testikoodissa on mahdotonta. Tähän ongelmaan on kuitenkin kaksi lähestymistapaa joiden avulla voit testata lambda-lausekkeita.

Ensimmäinen tapa on katsoa lambda-lauseketta siitä näkökulmasta, että se on pelkästään lähdekoodi metodin sisällä. Tässä ratkaisussa testaan metodia, eikä lambda-lauseketta sen sisällä.

```
public static List<String> isotKirjaimet(List<String> sanat) {

    return sanat.stream()
                    .map(string -> string.toUpperCase())
                    .collect(Collectors.toList());}
```

Varsinaisessa testikoodissa kutsutaan isotKirjaimet() metodia ja testataan metodin toimintaa, eikä varsinaisesti sen sisällä olevaa koodia.

```
@Test
public void sanat() {

    List<String> data = Arrays.asList("a", "b", "laurea");
    List<String> tulos = isotKirjaimet(data);
    assertEquals(asList("A", "B", "LAUREA"), tulos);

}
```

Toinen tapa lähestyä ongelmaa on se että, lambda-lausekkeen sijaan käytetään metodiviittausta. Alla olevassa koodissa tehdään erillinen metodi, joka muuttaa ensimmäisen kirjaimen isoksi. Tätä kyseistä metodia kutsutaan sitten metodiviittauksella map() välioperaation sisällä erillisessä metodissa.

```
public static String ekaIsoksi(String arvo) {

   char ekaChar = arvo.charAt(0);
   ekaChar = Character.toUpperCase(ekaChar);
   return ekaChar + arvo.substring(1);}
public static List<String> ekaKirjainIsoksi(List<String> sanat) {

   return sanat.stream().map(LuokanNimi::ekaIsoksi)
                                .collect(Collectors.toList());}
```

Tämän jälkeen testataan normaalisti metodi, jota kutsutaan metodiviittauksella.

```
@Test
public void twoLetterStringConvertedToUppercase() {
  String arvo = "laurea";
  String tulos = LuokanNimi.ekaIsoksi(arvo);
  assertEquals("Laurea", tulos);}
```

9 Stream API yhteenveto

Perusajatuksena Stream API:lle on virta(stream), joka on jono elementtejä tietystä lähteestä(source). Lähde on itsenäinen kokonaisuus, joka säilyttää itsellään näitä elementtejä. Näitä lähteitä voivat olla esim. kokoelmat(List,Set,Map), jotka säilyttävät itsellään elementtejä, kun lähteestä on luotu virta, niin tätä virtaa voidaan käydä läpi peräkkäin tai rinnakkain. Virtaan voidaan kohdistaa välioperaatioita(intermediate operation) tai pääteoperaatioita(terminal operation). Välioperaatio palauttaa uuden virran, kun taas pääteoperaatio lopettaa virran.

Operaatiot muodostavat yhtenäisen putken, joka alkaa lähteestä. Tämän jälkeen putkeen lisätään haluttaessa välioperaatioita tai lopetetaan suoraan pääteoperaatioon.

Jos verrataan esimerkiksi kokoelmien käsittelyä, niin funktionaalisella tavalla koodia syntyy vähemmän ja sen luettavuus helpottuu huomattavasti. Nämä seikat yleensä vähentävät virheitä sekä helpottavat ylläpitoa. Kumpi tapa on lopulta tuottavampi, riippuu siitä minkälaista sovellusta ollaan tekemässä. Nopeuden suhteen ei myöskään voi aina suoraan sanoa kumpi tapa on nopeampi tai tehokkaampi. Joka tapauksessa lambda-lausekkeet ja Stream API on hyviä lisäyksiä Java-kieleen.

Lähteet

Kirjat:

Peltomäki, J. 2014. Pieni Java 8 -kirja. Helsinki: BoD - Books on Demand

Warburton, R. 2014. Java 8 Lambdas. Sebastopol: O'Reilly

Sähköiset Lähteet:

Oracle Corporation. 2015. Stream. Viitattu 21.12.2015. https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

Oracle Corporation. 2015. Functional interface. Viitattu 21.12.2015. https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html