Aleksandr Pasharin

Jaettujen resurssien turvallinen käyttö yhtäaikaisuusohjelmoinnissa - lähestymistavat ja työkalut

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikan tutkinto-ohjelma

Insinöörityö

13.4.2023

Tiivistelmä

Tekijä:	Aleksandr Pasharin
Otsikko:	Jaettujen resurssien turvallinen käyttö yhtäaikaisuusohjelmoinnissa - lähestymistavat ja työkalut
Sivumäärä:	60 sivua
Aika:	13.4.2023
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikan tutkinto-ohjelma
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Lehtori Simo Silander

Tässä lopputyössä tutkimme yhteisten jaettujen resurssien käyttöön yhtäaikaisohjelmoinnissa liittyviä ongelmia. Aloitamme selvittämällä, mistä ongelmat tarkasti ottaen johtuvat. Opimme, että taustalla on toisaalta kilpailutilanteen mahdollisuus ja toisaalta myös datan näkyvyyteen liittyvät realiteetit nykyisissä moniydintietokoneissa.

Tämän jälkeen siirrymme ratkaisujen pariin. Otamme kolme konkreettista ohjelmointikieltä: Java, Clojure ja Rust ja tarkastelemme joitakin tyypillisiä työkaluja, joita nämä tarjoavat ongelmaan. Vertaamme näitä työkaluja ja ylipäätään näiden kielten erilaisia lähestymistapoja ongelmaan. Opimme, että mitä enemmän kieli rajoittaa sen käyttäjän mahdollisuuksia, sitä turvallisempaa yhtäaikaisohjelmointi tällä kielellä on.

Avainsanat: yhtäaikaisuus, jaetut resurssit, kilpailutilanne, datan näkyvyys, Java, Clojure, Rust, säie, lukko, atomariset muuttujat, volatile, referenssimuuttuja, STM transaktio, persistentti tietorakenne, muuttumattomuus

Abstract

Author:	Aleksandr Pasharin
Title:	Safe Usage of Shared Resourses in Concurrent Programming
Number of Pages:	60 pages
Date:	13 April 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Software Engineering
Supervisors:	Simo Silander, Senior Lecturer

The thesis investigates the problem of safe usage of shared resources in a concurrent context. First, the exact nature of the problem was identified, which boiled down to the possibility of race conditions as well as the fact that different CPUs might see shared resources in different states.

After the problems were identified, some common solutions the programming languages Java, Clojure and Rust provide were gone through and compared. The outcome of the study was that the more limitations and constrains a language forces on its user, the safer and more robust concurrent programming turns out to be.

Keywords: concurrency, shared resources, Java, Clojure, Rust, thread, lock, atomic variables, volatile, reference variable, STM transaction, persistent data structure, immutability

Sisällys

1 Johdanto

Karkeasti sanottuna *yhtäaikais-* tai *rinnakkaisohjelmoinnilla* tarkoitetaan ohjelmallisia keinoja saada tietokone suorittamaan eri komentoja ainakin näennäisesti “samanaikaisesti“ (sen käyttäjän näkökulmasta). Suomenkielisissä kirjallisuudessa sekä ammattilisissä piireissä molempia termejä käytetään usein sekoittaen niiden merkitystä keskenään ja niitä saatetaan helposti pitää synonyymeina.

Jos ollaan kuitenkin formaalisti tarkkoja, **yhtäaikaisuus** (engl. **concurrency**) ja **rinnakkaisuus** (engl. **parallelism**) eivät ole sama asia. Rinnakkaisuus tietojenkäsittelytieteessä tarkoittaa tietyn tietokoneen suoritinarkkitehtuurin rinnakkaisuuden aidon tuen käyttöä. Käytännössä tämä tarkoittaa sitä, että tietokoneella on käytössä ainakin kaksi *suoritinta* (engl. *CPU*, *central processing unit*), jotka pystyvät suorittamaan komentoja toisistaan täysin riippumattomasti ja kirjaimellisesti samanaikaisesti. Rinnakkaisuus ei siis ole (ainakaan yleensä) ohjelman ominaisuus, vaan liittyy myös konkreettiseen fyysiseen ympäristöön, jossa tämä ohjelma ajetaan. [1, chapter 1, section “Concurrent or Parallel“.]

Sen sijaan kun puhutaan yhtäaikaisuudesta, tarkoitetaan sitä, että ohjelmassa määritellään osia (“aliohjelmia“), jotka voidaan ainakin *semanttisessa mielessä* pitää toisistaan jokseenkin *erillisinä*. Tällöin nämä erilliset osat voidaan ainakin periaatteessa ajaa “samanaikaisesti“. [1, chapter 1, section “Concurrent or Parallel“.]

Näin määriteltynä yhtäaikaisuus on ohjelman sisäinen, looginen ominaisuus, joka liittyy sen rakenteeseen, ei siihen miten se oikeasti suoritetaan tietyllä tietokoneella. Yhtäaikaisuutta käytettävän ohjelman pitäisi pystyä imaisemaan koodin muodossa, mitkä erilliset ohjelman osat on tarkoitus ajaa samanaikaisesti ohjelman käyttäjän näkökulmasta. On selvää, että tämä onnistuu vain jos käytetty ohjelmointikieli tarjoaa työkaluja siihen.

Tässä työssä keskitymme nimenomaan *yhtäaikaisohjelmointiin*, ei rinnakkaisuuteen. Lisäksi valitsemme tietyn suppean joukon ohjelmointikieliä - **Java**, **Clojure** ja **Rust** - ja tutkimme juuri näiden kielten näkökulmasta yhtä yhtäaikaisuuteen liittyvää haastetta eli **yhteisten resurssien koordinoitua käyttöä**. Ensin identifioimme, mitä konkrettisia teknisiä ongelmia tulee vastaan, kun yritämme käyttää samanaikaisesti yhteisiä resursseja. Osoittautuu, että ongelmia on oleellisesti tasan kaksi: kilpailutilanteet ja datan näkyvyys. Sen jälkeen käymme läpi, mitä työkaluja valitsemamme kolme kieltä tarjoavat näiden ratkaisemiseen, miten nämä ratkaisut eroavat toisistaan käytännössä sekä mitä hyviä ja huonoja puolia jokaisen kielen lähestymistavalla on.

Kaikissa yllä mainituissa kielissä yhtäaikaisuuden käsittely perustuu niin sanottujen **säikeiden** käyttöön. Oletamme, että *säikeen* (engl. *thread*) käsite on tuttu lukijalle ainakin käytännön Javalla kirjoitetun koodin näkökulmasta. Muistetaan tässä vaiheessa vain, että säikeen voidaan ajatella olevan juuri se abstraktio, joiden avulla tietokone saadaan ajamaan “samanaikaisesti“ yllä mainitut ohjelman “erilliset tehtävät“.

Oletamme myös, että kielten Java, Rust ja Clojure perussyntaksi ja keskeiset ohjelmointiperiaatteet ovat lukijalle tuttuja. Tarvittaessa kirjallisuudesta tai internetistä helposti löytyy materiaalia, jonka avulla voidaan tutustua näiden kielten perusteisiin.

2 Ongelmat

2.1 Kilpailutilanne

Pyritään ensin ymmärtämään, mistä yhteisten resurssien käyttämisen ongelmassa on kyse - erityisesti miksi se todellakin on vaikea haaste, joka vaatii epätriviaaleja ratkaisuja.

Melkein jokainen ohjelma käyttää joitakin resursseja, esimerkiksi tietokoneen muistia, tietokoneessa sijaitsevia tiedostoja, tietokantayhteyksiä ja niin edelleen.

Tällöin jokaisella tämän ohjelman käyttävällä säikeellä on myös pääsy näihin resursseihin. Ongelman juuri piilee siinä, että eri säikeiden koodi ajetaan **samanaikaisesti**, jolloin säikeet myös pääsevät yhteisten resurssien käsiksi samanaikaisesti. Seurauksena ei ole mitään takeita siitä, missä järjestyksessä säikeet *manipuloivat* näitä resursseja.

Jos kaikki säikeet vain **lukevat** yhteisestä resurssista muuttamatta sen tilaa, niiden yhteinen toiminta on täysin turvallista. Ongelmat alkavat, kun ainakin yksi säie haluaa **muokata** näitä resursseja.

Koodiesimerkissä 1 esitetään Javalla kirjoitettua yksinkertaista laskuriluokaa, joka ei ole *säieturvallinen* (eli joka ei välttämättä toimi tarkoituksenmukaisesti, kun sitä käytetään ainakin kahdessa eri säikeessä samanaikaisesti).

```
class ThreadUnsafeCounter {
    private int count = 0;

    public void incr() {
        count++;
    }

    public int getCount(){
        return count;
    }
}
```

Koodiesimerkki 1: Ei-säieturvallinen luokka

Tarkastellaan seuraavaa koodia, jossa kaksi säiettä käyttävät tämän luokan instanssia samanaikaisesti:

```
1  ThreadUnsafeCounter c = new ThreadUnsafeCounter();
2
3  class CountingThread extends Thread {
4      public void run() {
5          for (int i = 0; i < 10000; i++) {
6              c.incr(); // yhteisen resurssin muokkaaminen tapahtuu
                   tässä
7          }
8      }
9  }
10
11 Thread A = new CountingThread();
```

```
12 Thread B = new CountingThread();
13
14 A.start(); B.start(); A.join(); B.join();
15
16 System.out.println(c.getCount()); // voisi luulla, että tämä rivi
       printtaa luvun 20000
```

Koodiesimerkki 2: Ei-säieturvallisen luokan käyttö yhtäaikaiskontekstissa

Yhteinen resurssi, jonka säikeet `A` ja `B` jakavat esimerkissä 2, on luokan `ThreadUnsafeCounter` eräs instanssi (koodissa muuttuja `c`). Rivillä 6 kumpikin säie pääsee muokkaamaan tämän instanssin sisäistä tilaa (sen metodin `incr` välityksellä).

Kun tämä ohjelma ajetaan, sen näkyviin printattu luku tulee olemaan joka ajokerralla hieman erilainen, mutta se tuskin tulee olemaan 20000, kuten voisi luulla (ja olisi tarkoituksenmukaista). Esimerkiksi kolme satunnaista peräkkäistä ohjelman ajoa, jotka suoritin, kun kirjoitin tämän kappaleen, tuottivat lopputuloksen `count` muuttujan arvoiksi 10945, 11365 ja 10380. Melkein puolet lisäyksistä näyttävät “hävinneen“.

Syy tähän on yksinkertainen. Metodi `incr` sisältää tasan yhden rivin - operaation `count++`, mutta tämä operaatio ei ole **atominen**. Kun se suoritetaan, todellisuudessa suorittimen tasolla tapahtuvat seuraavat erilliset alkeisoperaatiot ([2, s. 6]):

1. Muuttujan `count` arvo *“luetaan“* eli **kopioidaan väliaikaiseen muuttujaan**.
2. Tämän väliaikaisen muuttujan arvo kasvatetaan yhdellä.
3. Näin saatu luku kirjoitetaan takaisin muistiin muuttujan `count` uudeksi arvoksi.

Kun kaksi säiettä kukin suorittavat operaatiota `count++` suurin piirtein samaan aikaan, nämä erilliset komennot voivat tapahtua missä tahansa järjestyksessä toisiinsa nähden. Esimerkiksi voi tapahtua näin:

1. Säie A lukee muuttujan `count` arvon, joka sattuu olemaan tällä hetkellä luku 5.

2. Seuraavaksi säie B myös lukee saman arvon eli 5, koska A ei ole vielä ehtinyt suorittaa lisäystä loppuun.
3. Säie A laskee seuraavan arvon eli 6.
4. Säie A päivittää muuttujan `count` arvoksi 6.
5. Säie B laskee seuraavan arvon. Se käyttää aikaisemmin lukemaansa arvoa 5, joten sekin saa lopputuloksena myös 6. Vaikka muuttujan `count` arvo on jo päivittynyt, B ei tarkistaa tätä enää uudestaan, vaan operoi arvolla 5, joka on jo tavallaan “vanhentunut“.
6. Säie B päivittää muuttujan `count` arvoksi 6.
7. Lopputuloksena yhteisen muuttujan `count` arvo on kasvanut vain yhdellä, vaikka suoritettiin 2 erillistä lisäysoperaatiota.

Tilannetta, jossa ohjelman suorituksen lopputulos riippuu siitä, missä järjestyksessä eri säikeet suorittavat operaatiot, sanotaan **kilpailutilanteeksi** (engl. **race condition**). Jos tämä vaikuttaa yhteisen datan arvoon, kuten esimerkissä 1, puhutaan **datakilpailusta** (engl. **data race**). Kilpailutilanteen mahdollisuus on suurimpia syitä, miksi yhteisten resurssien käyttö monesta säikeestä käsin vaatii epätriviaaleja työkaluja. [2, s. 20, 342.]

Esimerkissä 1 yhteinen resurssi muokattiin kahdessa eri säikeessä. Jos vain yksi säie muokkaa resurssia ja toinen vain lukee tätä, ongelma ei ole kenties niin paha, mutta kuten yllä esitetyn skenaarion kohdasta 5 voidaan helposti päätellä, silloinkin lukeva säie voi lukea ja esittää käyttäjälle arvon, joka on jo *vanhentunut* (engl. *stale date*). Lisäksi, koska jokaisella suorittimella on oma työmuistinsa (engl. *cache*), eri säikeet lukevat saman muuttujan arvo mahdollisesti jopa eri paikoista. Tästä ilmiöstä puhumme tarkemmin seuraavassa aliluvussa “Datan näkyvyys“ .

Tilanteesta riippuen vanhentuneen datan käyttö voi olla hyväksyttävissä, mutta voi myöskin johtaa virheisiin esimerkiksi, jos tämän vanhentuneen arvon perusteella ohjelmassa tehdään seuraavaksi joitakin loogisia päättelyä ja valintoja.

2.2 Datan näkyvyys

Kilpailutilanne ei ole ainoa syy, miksi monisäikeisessä kontekstissa jokin säie voi lukea muuttujan vanhan tai jopa suorastaan väärän arvon. Tarkastellaan seuraavaa jälleen Javalla kirjoitettua esimerkkiä.

```
1  public class DataVisibilityProblems  {
2      static boolean readyToReadValue = false;
3      static long value = 0;
4
5      static Thread t1 = new Thread(() -> {
6          value = Long.MAX_VALUE;
7          readyToReadValue = true;
8      });
9
10     static Thread t2 = new Thread(() -> {
11         while (!readyToReadValue) {}
12         System.out.println(value);
13     });
14
15     public static void main(String[] args) throws
           InterruptedException {
16         t1.start(); t2.start();
17         t1.join(); t2.join();
18     }
19 }
```

Koodiesimerkki 3: Näkyvyysongelmat

Ensi näkemältä voisi ajatella, että on olemassa vain yksi tapa, jolla tämä koodi voi toimia (jos mahdollista `InterruptedException` poikkeusta ei oteta huomioon). Säie `t1` asettaa muuttujan `value` arvoksi 9223372036854775807 (eli `Long.MAX_VALUE`) ja *sen jälkeen* asettaa muuttujan `readyToReadValue` arvoksi `true`. Toinen säie tarkistaa tätä arvoa jatkuvasti silmukassa, joten edes jos hetkellisesti syntyykin kilpailutilanne, se ei haittaa. Aiemmin tai myöhemmin silmukassa luetaan `readyToReadValue`-muuttujan arvoksi `true`, jolloin silmukasta poistutaan. Muuttujan `value` arvo päivitettiin *ennen* kuin `readyToReadValue`-muuttujan arvoksi asetettiin `true`. Tästä seuraa, että kun silmukasta poistutaan, muuttujan `value` arvo on 9223372036854775807 (eli `Long.MAX_VALUE`), ja juuri tämä arvo printataan.

Kuitenkin kun tämä koodi ajetaan, on myös täysin mahdollista, että sen sijaan tapahtuu yksi seuraavista skenaarioista:

1. Ohjelma printtaa `0`, eli muuttujan `value` vanhan, alkuperäisen arvon.
2. Ohjelman suoritus ei päädy koskaan, koska säikeen `t2` sisällä ei koskaan poistuta silmukasta.
3. Ohjelma printtaa jotain muuta kuin `0` tai `9223372036854775807`.

Miten ihmeessä nämä ovat mahdollisia?

Kun kävimme läpi, mitä tapahtuu ohjelman suorituksen aikana, me teimme muutaman intuitiivisesti järkevältä tuntuvan oletuksen, jotka eivät kuitenkaan välttämättä pidä paikkaansa. Yksi näistä oletuksista oli seuraava: rivi 6 (`value = Long.MAX_VALUE`) suoritetaan aina ennen riviä 7 (`readyToReadValue = true`), koska koodissa yksi tulee toisen jälkeen. Näin ei kuitenkaan tarvitse olla.

Javan kääntäjä, JVM virtuaalikone tai tietokoneen suorittimet saavat nimittäin täysin luvallisesti vaihtaa koodin komentojen järjestystä **optimisaatiosyystä**, jos tämä ei vaikuta ohjelman semantiikkaan **yksittäisessä säikeessä** [2, s. 33-35, s. 337-338]. Säikeen `t1` näkökulmasta rivien 6 ja 7 komennot ovat toisistaan täysin riippumattomia, joten kun ohjelma ajetaan, ne voidaan todellisuudessa suorittaa missä tahansa järjestyksessä. Ohjelmoijalla ei ole tällöin mitään tapaa tietää, että näin on käynyt, tai estää tämä tapahtumasta. Tämä ilmiö ei ole myöskään mikään Javan erikoisuus, vaan käytännössä kaikki nykyiset mainstream-ohjelmointikielet toimivat samalla tavalla, mukaan lukien Rust ja Clojure. Vaikka eivät toimisikaan, raudan tasolla nykytietokoneilla on myös lupa toimia näin. Tämän ilmiön syyt ovat monimuotoiset eikä niitä ole mahdollista tarkastella tarkemmin tämän työn puitteissa. Lyhyesti sanottuna ohjelmointikielten kääntäjät sekä tietokoneiden suorittimet toimivat tällä tavalla, koska se nopeuttaa huomattavasti ohjelmien suoritusaikaa. Kyse on siis tärkeistä optimisaatioista, joista ei voi tinkiä, koska muuten tietokoneet toimisivat paljon hitaammin. [3, subchapter 22.5; 4, subchapter 2.2.3; 5, chapter 3.]

Näin ollen periaatteessa on täysin mahdollista, että ensin suoritetaan komento `readyToReadValue = true` ja tämän jälkeen seuraavaksi säie `t2` ehtii heti

huomata tämän muutoksen, poistua silmukasta ja ojentaa printtausmetodi muuttujan `value` alkuperäinen arvo eli `0`.

Itse asiassa on myös täysin mahdollista, että säie `t1` ehtii välissä päivittää muuttujan `value` arvon, mutta tämä muutos ei näy säikeelle `t2` ajoissa, ja se silti lukee tämän muuttujan arvoksi `0`. Tämä johtuu siitä, että eri tietokoneen suorittimet saattavat operoida omilla muuttujien “kopioilla“, jotka sijaitsevat niiden omassa *työmuistissa* (engl. *cache*) [2, s. 34, s. 338]. Tällöin yhden säikeen tekemät muutokset eivät välttämättä näy sen jälkeen toiselle säikeelle - ei ainakaan heti ja mahdollisesti ei edes koskaan!

Tämä selittää, miten skenaario 2 on mahdollinen. Jos säie `t2` käyttää omaa cachattua kopiota muuttujasta `readyToReadValue`, se ei välttämättä edes koskaan huomaa, että sen arvo on vaihtunut säikeen `t1` toimesta. Jos näin käy, säikkeen `t2` näkökulmasta muuttujan `readyToReadValue` arvo on aina `false`, ja säikeen suorittama koodi jää ikuiseen silmukkaan. Että näin ei kävisi, tarvitaan sellaisia työkaluja, jotka varmistaisivat, että tietyillä hetkillä säikeiden työmuistin cache invalidoidaan ja yhteisten muuttujien arvot *synkronoidaan* säikeiden kesken.

Jäljellä on vielä mahdollisuus 3: ohjelma voi myös printtaa jotain muuta kuin arvot `0` tai `9223372036854775807`, vaikka nämä ovat ainoat sen koodissa eksplisiittisesti vastaavaan muuttujaan asetetut arvot. Tämä johtuu siitä, että Javassa 64 bitin kokoisten primitiivimuuttujan (eli `long` ja `double`) arvon päivittämisen *ei tarvitse olla atomaarinen operaatio*. Kielen standardin mukaan on sallittua, että tällaisen muuttujan päivitys hoidetaan osissa päivittämällä muuttujasta ensin puolet (32 bittiä) ja seuraavaksi toiset 32 bittiä. Tällöin voi käydä niin, että joku säie käy lukemassa muuttujan arvo kesken tätä päivitystä. Silloin se näkee sen tilassa, jossa tämä päivitys on “kesken“ - puolet biteistä tulevat uudesta arvosta ja puolet vanhasta. [2, s. 36.]

3 Ratkaisut

3.1 Lukot

Esimerkissä 2 datakilpailu johtui oleellisesti siitä, että eri säikeet saattavat suorittaa koodiriviä `count++` samanaikaisesti sekaisin kukin omalla tahdillaan. Jos olisi mahdollista jotenkin sopia, että vain yksi säie kerralla saa suorittaa lisäysoperaation ++, kilpailutilannetta ei olisi tapahtunut.

Lukko on työkalu, jonka avulla voidaan varmistaa, että vain yksi säie suorittaa tietty sarja operaatioita. (Eksluksiivinen) lukko on mikä tahansa objekti, jonka voi “lukita“ tai “avata“. Kun lukko lukitaan, kaikki koodi, joka suoritetaan samassa säikeessä tämän jälkeen, on “suojattu“ tällä lukolla, kunnes lukko avataan. Suojaaminen tarkoittaa tässä yhteydessä tasan sitä, että vain yksi säie kerrallaan voi suorittaa tätä koodia.

Jos säie haluaa suorittaa lukolla suojattua koodiosiota, sen täytyy ensin *saada tämä lukko haltuunsa*. Vain yksi säie kerrallaan saa “omistaa“ lukon. Jos lukko on jo toisen säikeen omistuksessa, mikä tahansa muu säie, joka haluaa myös saada tämän lukon haltuunsa, joutuu odottamaan, kunnes lukko avataan.

Näin ollen, kun säie siirtyy suorittamaan lukolla suojattua koodia, tapahtuu tasan yksi seuraavista vaihtoehdoista:

1. Lukko on vapaa. Tällöin säie voi heti ottaa se haltuunsa. Jos tämä onnistuu, säie siirtyy koodissa eteenpäin normaalisti.
2. Lukko on toisen säikeen omistuksessa. Silloin säie **pysäyttää** suorituksensa ja **odottaa**, kunnes lukko vapautuu.

Jos lukko on varattu ja ainakin kaksi eri säiettä odottavat samanaikaisesti sen vapauttamista, vain yksi niistä saa sen haltuunsa, kun se vapautuu; muut joutuvat jatkamaan odotusta.

Voidaan ajatella, että lukolla suojatusta koodiosiosta tulee ikään kuin **atominen** muiden säikeiden näkökulmasta. Tämä riittää välttämään kilpailutilannetta lukolla

suojatussa koodikokonaisuudesta.

Lukon määritelmästä seuraa, että pätee itse asiassa vahvempi väite: kun säie omistaa jonkun lukon, vain tämä säie voi suorittaa **minkä tahansa** tämän lukon suojaamaa koodia.

3.1.1 Lukot Javassa

Yksinkertaisin tapa lukita mielivaltainen koodipätkä Javassa on käyttää kielen tarjoamaa `synchronized`-syntaksia:

```
synchronized (lockObject) {
    // tähän laitetaan koodi, johon halutaan päästää vain yksi säie
        kerrallaan
}
```

Koodiesimerkki 4: Javan synchronized-syntaksi

Tässä `lockObject` voi itse asiassa olla mikä tahansa Java-objekti. Tarkemmin sanottuna Javassa jokaisella oliolla on sisäänrakennettu niin sanottu *sisäinen lukko* (engl. *insintric lock*)[1]. Juuri tätä lukkoa JVM käyttää `synchronized`-lohkon yhteydessä. [2, s. 25.]

`synchronized`-syntaksia käytettäessä lukon lukitseminen ja sen vapauttaminen jäävät eksplisiittiseksi - Java pitää niistä itse huolta taustalla. Lukko vapautuu *automaattisesti* aina, kun `synchronized`-lohkosta poistutaan - myös silloin kun tämän lohkon sisällä oli heitetty poikkeus. [2, s. 25.]

Jos lukolla halutaan suojata jonkun olion metodin koko koodia, ja lukko-objektiksi kelpaa tämä objekti itse, riittää merkitä metodi `synchronized`-avainsanalla. [2, s. 25.]

Edellisen valossa yksinkertaisin tapa estää datakilpailun mahdollisus esimerkissä 2 olisi tehdä seuraavat muutokset esimerkin 1 luokassa `ThreadUnsafeCounter` (muutokset on merkitty lihavoidulla punaisella fontilla):

[1] Siitä käytetään myös nimitystä *monitori* (engl. *monitor*)

```
class ThreadSaferCounter {
    private int count = 0;

    public synchronized void incr() {
        count++;
    }

    public int getCount(){
        return count;
    }
}
```

Koodiesimerkki 5: Datakilpailusta vapaa laskuriluokka

Jos nyt esimerkin 2 koodissa vaihdetaan luokan `ThreadUnsafeCounter` instanssi luokan `ThreadSaferCounter` instanssiksi ja näin saatu ohjelma ajetaan, kentän `count` arvoksi saadaan ohjelman lopussa aina 20000.

Metodiin `getCount` implementaatiossa on kuitenkin edelleenkin jäänyt vanhentuneen datan mahdollisuus, koska sen koodia ei ole suojattu lukolla. Meidän esimerkissä se ei tuo mitään ongelmia, mutta jos `ThreadSaferCounter`-luokkaa käytetään eri tavalla toisessa kontekstissa, voi esimerkiksi käydä seuraavasti:

- Säie A alkaa `incr`-metodin suorittamista ja lukee `count`-muuttujan arvoksi luku 5.
- Ennen kun A ehtii kuitenkin päivittää `count`-muuttujan arvoa, säie B alkaa suorittamaan metodia `getCount` ja lukee `count`-muuttujan arvoksi saman luvun 5.
- Säie A tulee väliin ja suorittaa metodi `incr` loppuun - `count`-muuttujan arvo on nyt 6.
- Säie B suorittaa metodi `getCount` loppuun - se palauttaa “vanhan arvon“ 5, jonka se on ehtinyt lukea jo aikaisemmin.

Näin ollen täysin säieturvallinen versio `Counter`-luokasta saadaan suojaamalla **samalla** lukolla myös lukeva metodi `getCount`:

```
class ThreadSafeCounter {
    private int count = 0;

    public synchronized void incr() {
        count++;
    }

    public synchronized int getCount(){
```

```
        return count;
    }
}
```

Koodiesimerkki 6: Täysin säieturvallinen laskuriluokka

Nyt kumpikin metodi on suojattu **samalla lukolla** (joka on tämän metodin olion sisäinen lukko). Jos säie A on suorittamassa esim. metodia `incr`, toinen säie B ei voi aloittaa edes metodin `getCount` suorittamista. Se joutuisi ensin odottamaan, kunnes säie A on suorittanut metodin `incr` suorittamista loppuun, minkä jälkeen `getCount`-metodin kutsu palauttaisi tämän hetken oikean arvon muuttujalle `count`. Seuraava `incr`-metodin kutsu taas voisi alkaa vasta, kun metodin `getCount` kutsu on palautunut ja näin poispäin. Seurauksena koodi on täysin säieturvallinen ja korrekti: muuttujan `count` lukeminen tuottaa aina ajankohtaisen arvon, ja päivittäminen tapahtuu aina atomisesti.

Aliluvussa “Datan näkyvyys“ olemme oppineet, että yleisesti ottaen edellisessä kappaleessa esitetty väite “muuttujan lukeminen tuottaa aina ajankohtaisen arvon“ ei vältämättä pidä paikkaansa. Jos yksi säie on tehnyt muutoksen yhteiseen resurssin, toinen säie ei vältämättä näe sitä heti. Onneksi Javan sisäisillä lukoilla on ominaisuuksia, jotka automaattisesti takaavat korrektin datan näkyvyyden. Nimittäin Javan spesifikaatio sisältää niin sanotun **muistimallin** (engl. **memory model**), joka antaa tiettyjä tarkkoja takeita siitä, mitä voidaan olettaa ohjelmassa, joka käyttää eri säikeitä. Esimerkiksi Javan muistimalli vaatii, että kun sisäinen lukko vapautetaan, kaikki lukolla suojatussa koodiosiossa tapahtuneet yhteisten muuttujien muutokset näkyvät muille säikeille **viimeistään silloin, kun ne saavat saman lukon haltuun**. [2, s. 36-37.]

Näin ollen esimerkin 6 säieturvallisen laskuriluokan kohdalla Javan muistimalli takaa korrektin datan näkyvyyden säikeiden kesken. Koska luokan kaikki metodit on suojattu samalla lukolla, mikä tahansa niitä kutsuva säie näkee ajoissa kaikki aikaisemmin muuttujan `count` tapahtuneet päivitykset riippumatta siitä, missä säikeessä ne on sovellettu. Tämä implementaatio on vapaa sekä datakilpailuista, että kilpailutilanteista, myös niistä, jotka liittyvät vain lukemiseen, ei päivittämiseen.

Implisiittiset lukot

Sisäiset monitorit ovat “eksplisiittisiä lukkoja“ siinä mielessä, että niitä käyttäessä lukitukseen liittyvä semantiikka on piilotettu ohjelmoijalta, ja JVM pitää itse huolta siitä, että lukko lukitaan / vapautetaan kun `synchronized`-lohkoon astutaan / siitä poistutaan. [2, s. 25.]

Joskus kuitenkin sisäiset monitorit eivät riitä ja tarvitaan lukkoja, joilla on ominaisuuksia, joita sisäisillä monitoreilla ei ole. Tästä syystä Javan standardikirjasto tarjoaa myös `Lock`-rajapinnan, joka abstrahoi “eksplisiittisen“ lukon ominaisuuksia, sekä valmiita luokkia, jotka toteuttavat tämän rajapinnan. Näistä yksinkertaisin ja käytetyin lienee `ReentrantLock`. Tämä implementaatio myös takaa korrektin datan näkyvyyden säikeiden kesken samalla tavalla kuin sisäiset monitorit. [2, s. 277-278.]

Sisäisiin monitoreihin verrattuna `Lock`-rajapinta tarjoaa monipuolisempia ja joustavampia työkaluja. Esimerkiksi lukituksen (rajapinnan metodin `lock` kutsu) ja lukon vapauttamisen (rajapinnan metodin `unlock` kutsu) ei tarvitse tapahtua samassa näkyvyysalueessa. Toinen ero liittyy siihen, että sisäisen monitorin tarjoamaa lukitusta ei voi mitenkään perua lukituksen jälkeen. `Lock` rajapinta tarjoaa sen sijaan seuraavia työkaluja, jotka mahdollistavat tämän.

- Metodi `void lockInterruptibly()` heittää poikkeuksen, jos kutsuva säie keskeytetään (esimerkiksi kutsumalla `Thread.interrupt`) sillä aikaa, kun lukon vapauttamista odotetaan.
- Metodi `boolean tryLock()` luopuu heti yrityksestä saada lukko haltuun, jos se on jo toisen säikeen hallussa. Tällöin se palauttaa `false`. Jos lukitus on onnistunut, metodi palauttaa `true`.
- Edellisen metodin ylikuormitettu versio `boolean tryLock(long time, TimeUnit unit)` odottaa lukon vapauttamista vain tietyn ajan verran. Tavallisen lukon kohdalla on aina mahdollista, että se vapautuu liian myöhään - tai jopa ei koskaan. Tämän metodin avulla voidaan välttää ainakin tällaisia ikäviä tilanteita. [2, s. 279-282.]

Toinen hyödyllinen lukitusta tarjoava rajapinta Javassa on `ReadWriteLock`. Tämän instanssi ei itse ole lukko, vaan se tarjoaa kaksi erillistä lukkoa, jotka ovat `ReadLock`, jonka tarkoitus on tarjota lukitusta datan **lukemista varten**, ja

`WriteLock`, joka tarjoaa lukitusta datan **kirjoittamista varten**. Näiden idea on seuraava. Palautetaan mieleen edellä tarkasteltu säieturvallisen laskurin esimerkki 6. Koska emme halunneet, että `getCount`-metodin avulla tapahtuvat datan lukemiset tuottaisivat vanhentunutta arvoa, olimme suojanneet sen luokan instanssin sisäisellä lukolla. Kuvitellaan kuitenkin tilanne, jossa monta säiettä haluaa vain lukea dataa eli tarkemmin sanottuna päästää suorittamaan `getCount`-metodia silloin kun mikään muu säie ei yritä päivittää tätä arvoa. On selvä, että tämä olisi periaatteessa täysin turvallista. Sisäinen monitori tai `ReentrantLock` eivät kuitenkaan sallisi tätä. Kumpikin päästäisi vain yhden säikeen kerrallaan tämän lukevan metodin sisään. Tässä tapauksessa tämä rajoitus ei todennäköisesti hidasta suoritusta kovin huomattavasti käytännössä, sillä yhden kokonaisluvun lukeminen on nopeata, mutta jossakin toisessa yhteydessä tämä lähestymistapaa voisi vähentää sovelluksen suorituskykyä turhaan ilman mitään hyvää syytä. `ReadLock` sen sijaan ei ole eksklusiivinen (eli ei edes ole lukko tämän kappaleen alussa annetun määritelmän mukaan), sillä monta säiettä voivat omistaa sen samanaikaisesti. Kuitenkin `WriteLock` on aito lukko: sen saa omistaa vain yksi säie kerrallaan. Tarkoitus on, että `ReadLock` suojaa paikkoja koodissa, jossa dataa luetaan, kun taas `WriteLock` suojaa paikkoja, joissa tämä sama dataa muokataan, mutta itse kieli tietysti ei voi tarkistaa, että näitä käytetään “oikein“. Tämä on täysin ohjelmoijan vastuulla. Jotta rajapinnan implementaatiosta olisi hyötyä, kummankin lukon pitäisi myös taata korrektia datan näkyvyyttä. Implementaatio saa itse päättää, mitä tapahtuu, jos jokin säie haluaa saada haltuunsa `WriteLock`-lukkoa saman aikaan, kun `ReadLock` on muiden säikeiden omistuksessa - tai toisinpäin. Javan standarikirjasto tarjoaa rajapinnalle implementaation `ReentrantReadWriteLock`, joka ei salli minkään säikeen omistaa `ReadLock`-lukkoa, kun `WriteLock` on jonkun säikeen omistuksessa. Tämä strategia takaa, että datan lukemiset eivät voi koskaan tuottaa jo vanhentunutta dataa. [2, s. 286-289.]

Luokka `ReentrantReadWriteLock` (tai mikä tahansa muu järkevä implementaatio rajapinnalle `ReadWriteLock`) on hyödyllinen esimerkiksi tilanteissa, joissa yhteisiä resursseja luetaan usein eri säikeissä, mutta päivitetään suhteellisen harvoin. Tällaisessa tapauksessa perinteinen lukko hidastaisi lukevia säikeitä turhaan.

3.1.2 Lukkojen heikkoudet ja haasteet

Lukot tarjoavat periaatteessa toimivan ratkaisun yhteisten resurssien käyttöön liittyviin ongelmiin, mutta tuovat mukanaan myös omia ongelmia ja haasteita.

Kenties ensimmäisenä mieleen tuleva lukkojen huono puoli on seuraava. Lukon vapauttamista odottava säie pysäyttää toimintansa tekemättä mitään muuta odotusaikana. Tämä voi vaikuttaa negatiivisesti sovelluksen suorituskykyyn, ja pahimmillaan johtaa siihen, että yhtäaikaisuutta ikään kuin hyödyntävä sovellus ei olekaan käytännössä edes nopeampi kuin vastaava ohjelma, jossa samaa työtä tehdään yhdessä säikeessä - tai on jopa sitä hitaampi! Lyhyesti sanottuna, jos säikeet viettävät paljon aikaa odotustilassa, haitat saattavat mitätöidä yhtäaikaisuuden tuomat hyödyt.

Tämä havainto johtaa seuraavaan lukkojen käytön “nyrkkisääntöön“: on pyrittävä siihen, että lukkoja pidetään suljettuna niin lyhyen ajan kuin on mahdollista. Koodin tasolla se tarkoittaa sitä, että lukolla pitää suojata mahdollisimman “pienet“ koodialueet, tarkemmin sanottuna sellaiset, joiden suorittaminen ei kestä pitkään.

Edellisestä, ja muistakin syistä, säikeiden kesken jaettujen isojen kokoelmien (esimerkiksi taulukkojen tai listojen) säieturvallinen, mutta tehokas käyttö on haasteellista lukkoja käyttäessä. Esimerkiksi jos koko kokoelma suojellaan lukolla iteroinnin aikana, muut säikeet saattavat joutua odottamaan pitkään, kunnes lukko luovutetaan. Yksi tapa taistella tätä ongelma vastaan on tehdä ensin kokoelmasta kopio ja käyttää sitä, mutta ison kokoelman kopioiminen saattaa viedä liikaa aikaa ja muistia. Lisäksi jos kopio ei ole tarpeeksi “syvä“ ja jakaa sisältöään alkuperäisen kokoelman kanssa, tämän muuttaminen toisessa säikeessä saattaa vaikuttaa kopioon ei-halutulla tavalla. Kopion sisältämä dataa myös riskeeraa vanheta nopeasti. Toisessa lähestymistavassa kokoelmasta lukitaan vain pieni osa kerralla, kun sitä pitkin edetään, esimerkiksi pitkässä listassa lukitaan aina kerralla vain tarkasteltava alkio ja sitä seuraava alkio. Tällöin kuitenkin toinen säie voi muuttaa kokoelmaa samaan aikaan, kun se

iteroidaan, mikä Javassa voi helposti johtaa `ConcurrentModificationException`-poikkeuksen syntyyn. [2, s. 82-83; 1, chapter 2, day 2, section “Hand-over-Hand Locking“.]

Lukitukseen liittyy aina yhteisen datan synkronoinnin tarve, koska haluamme taata korrektia yhteisen datan näkyvyyttä säikeiden kesken. Tämä kuluttaa lisää resurssia ja lisäksi pienentää mahdollisuuksia hyödyntää nopeata välimuistia.

Säieturvallista koodia kirjoittaessa joudutaan usein suojamaan lukolla myös sellaisia koodin osia, joissa yhteisiä resursseja ei muokata, vaan ainoastaan luetaan. Seurauksena säikeet, jotka vain lukevat dataa, hidastavat toistensa suoritusta, mahdollisesti turhaan, sillä yhteisen datan lukeminen on usein turvallista. Tähän ongelmaan nähtiin edellä jo yksi mahdollinen ratkaisu - rajapinnan `ReadWriteLock` kaltainen työkalu.

Lukkiuma

Tarkastellaan seuraavaa yksinkertaista esimerkkiä.

```
Object l1 = new Object();
Object l2 = new Object();

Thread t1 = new Thread(() -> {
    try {
        synchronized (l1) {
           Thread.sleep(1000);

           synchronized (l2) {
               // tässä ohjelman suorituksen on tarkoitus jatkua
            }
        }
    } catch (InterruptedException e){ }
});

Thread t2 = new Thread(() -> {
    try {
        synchronized (l2) {
            Thread.sleep(1000);

            synchronized (l1) {
                // tässä ohjelman suorituksen on tarkoitus jatkua
            }
```

```
        }
    } catch (InterruptedException e){ }
});

t1.start(); t2.start();
t1.join();  t2.join();
```

Koodiesimerkki 7: Lukkiuma

Kun tämä ohjelma ajetaan, melko varmasti tapahtuu seuraavaa. Säie `t1` varaa lukon `l1` ja säie `t2` varaa lukon `l2`. Tämän jälkeen kumpikin säie nukkuu noin sekunnin. Kun `t1` herää, se haluaa varata lukon `l2`, mutta se on säikeen `t2` omistuksessa, joten `t1` jää odottamaan lukon vapauttumista. Vastaavasti kun `t2` herää, se haluaa varata lukon `l1`, mutta se on säikeen `t1` omistuksessa, joten `t2` jää myöskin odottamaan. Nyt kumpikin säie odottaa lukkoa, joka ei koskaan enää vapaudu. Ohjelman suoritus ei koskaan pääty eikä eteenpäin päästä.

Esimerkissä 7 syntyvää tilannetta sanotaan **lukkiumaksi** (engl. **deadlock**). Lukkiuma on yksi suurimpia ongelmia, mitä lukkojen käyttöön liittyy käytännössä, sillä isossa ja monimutkaisessa projektissa on vaikeata seurata kaikkia lukkojen mahdollisia käyttöjä ja estää mahdollista lukkiuman syntyä.

Lukkiuman esiintyminen edellyttää, että eräässä sovelluksen lukkojen tilaa kuvaavan graafiin ilmestyy sykli, jolloin säie `t1` odottaa lukkoa, jonka omistaa säie `t2`, joka puolestaan odottaa lukkoa, jonka omistaa säie `t1` ja niin edelleen, kunnes lopuksi säie `tN` odottaa lukkoa, jonka omistaa syklin ensimmäinen säie `t1`. Näin ollen yksi suosittu tapa tehdä lukkiuma mahdottomaksi on sopia jokin tarkka järjestys lukkojen joukossa ja lukita ne aina tämän järjestyksen mukaan. [1, chapter 2, day 1, section Multiple Locks; 2, s. 205-206.]

Toinen tunnettu strategia on lisätä järjestelmään koodi, joka jatkuvasti monitoroi taustalla tätä sovelluksen lukkoriippuvuuksien graafia, etsii siinä syklit ja poistaa ne (esimerkiksi lopettamalla yhden syklin prosesseista). Tätä lähestymistapaa sovelletaan monen tietokannan implementaatioissa. [2, s. 206.]

Livelock

Tarkastellaan seuraavaa esimerkkiä, jossa kaksi liiankin kohteliasta puolisoa yrittävät syödä illallista yhdessä. Heillä on kuitenkin vain yksi lusikka ja kumpikin puoliso luovuttaa lusikan toiselle ennen kuin suostuu syömään itse.

```
public class PoliteDinner {
    static class Spoon {
        private Person owner;
        public Spoon(Person d) { owner = d; }
        public synchronized void setOwner(Person d) { owner = d; }
    }

    static class Person {
        private String name;
        private boolean isHungry;

        public Person(String n) { name = n; isHungry = true; }
        public boolean isHungry() { return isHungry; }

        public void eatWith(Spoon spoon, Person spouse) {
            while (isHungry) {
                // jos puoliso on nälkäinen, anna sille lusikka
                if (spouse.isHungry()) {
                    spoon.setOwner(spouse);
                } else {
                    // muuten voi vihdoinkin siirtymään eteenpäin ja
                        syödä
                    isHungry = false;
                }
            }
        }
    }

    public static void main(String[] args) {
        Person husband = new Person("Bob");
        Person wife = new Person("Alice");

        Spoon s = new Spoon(husband);

        new Thread(new Runnable() {
            public void run() { husband.eatWith(s, wife); }
        }).start();

        new Thread(new Runnable() {
            public void run() { wife.eatWith(s, husband); }
        }).start();
    }
}
```

Koodiesimerkki 8: Livelock

Kun tämä ohjelma ajetaan, se pyörii ikuisesti ja kumpikin puoliso jää nälkäiseksi. Kaksi säiettä vain vaihtavat jatkuvasti yhteisen resurssin (eli lusikan) omistuksen. Vaikka tilanne on jumissa ja kumpikin säie ei pääse eteenpäin kohti tavoitetta (päästä syömään), kyseessä ei ole lukkiuma, sillä molemmat säikeet tekevät jatkuvasti työtä eivätkä odota passiivisesti resurssin vapauttamista. Mitään hyödyllistä ne eivät kuitenkaan ehdi tehdä, koska kaikki aikaa kuuluu kohteliaisiin yrityksiin päästä toinen säie eteenpäin. Tällaisesta tilanteesta käytetään nimitystä **livelock**. [2, s. 219.]

Livelock ei esiinny käytännössä niin usein kuin lukkiuma, mutta on silti todellinen uhka aina, kun eri säikeet käsittelevät yhteisiä resursseja.

Säikeen nälkiintyminen

Säikeen nälkiintyminen (engl. **thread starvation**) tapahtuu, kun säie ei koskaan saa haltuunsa haluamiaan resursseja, koska ne ovat suojattuja lukolla ja tämä on jatkuvasti muiden säikeiden käytössä. [2, s. 218.]

Nälkiintymiseen auttaa esimerkiksi niin sanottujen “reilujen“ lukkojen käyttö. Reilun lukon implementaatio varmistaa, että säikeet pääsevät resurssien luo suurinpiirtein yhtä usein, eli resurssien käyttöä jaetaan säikeiden kesken “reilusti“. [2, s. 283-284.]

Reiluuden ylläpitäminen vaatii kuitenkin lisää resursseja, joten reilut lukot toimivat hieman hitaammin kuin tavalliset, epäreilut lukot [2, s. 283-284].

Yleisemmin eri säikeille voidaan asettaa eri prioriteetteja, jolloin korkean prioriteetin säikeellä on etuoikeus päästä seuraavaksi varaamaan lukko matalan prioriteetin säikeeseen nähden. Tällöin kuitenkin voi tapahtua niin sanottu *prioriteettien inversio*. Täksi kutsutaan tilannetta, jossa matalan prioriteetin säie pitää lukittuna resurssia, jonka vapauttamista odottaa korkeamman prioriteetin säie. Lisäksi eri käyttöjärjestelmät tulkitsevat säikeiden prioriteetteja eri tavalla,

jolloin ohjelma saattaa käyttäytyä eri koneissa eri tavalla. [2, s. 218, 320.]

Lukituspolitiikan manuaalinen ylläpito

Suurimmassa osassa kielistä, jotka tarjoavat lukkoja työkaluna, niiden käyttö edellyttää ohjelmoijalta äärimmäistä tarkkuutta. Se on haastavaa, virhealtista, eikä kieli yleensä tarjoa mitään automaattisia turvamekanismeja, jotka varmistaisivat, että lukkoja on sovellettu koodissa oikein ja turvallisesti. Esimerkiksi Javan kääntäjä ei tiedä, mitä koodia tai dataa pitää suojata lukolla ja onko tämä suojaaminen tehty oikein, joten se ei tarkista sitä. Usein koodissa on tietty resurssi, jonka kanssa pitää aina käyttää lukitusta, mutta koska kielessä ei ole mekanismeja tämän turvaamiseksi, on helppoa käyttää väärä lukkoa tai jopa unohtaa käyttää sitä. Tästä seuraa, että käytännössä joudutaan turvautumaan sopimuksiin, jotka pitää dokumentoida, seurata ja päivittää projektin kehittyessä. On selvä, että tämä lähestymistapa ei ole kovin luotettava. Mitä isompi projekti, sitä vaikeampaa tällaisen “lukituspolitiikan“ manuaalinen ylläpito on, sekä sitä enemmän virheitä livahtaa ohjelmistotuotteeseen.

Seuraavassa luvussa tutustumme lukitukseen kielessä Rust. Tämä toimii periaatteessa samalla tavalla kuin muissakin kielissä, mutta sisältää kielen pakottamia automaattisia turvamekanismeja, jotka helpottavat ainakin “manuaalisen lukituspolitiikan“ ylläpito-ongelman kanssa.

3.1.3 Yhteisten resurssien jakaminen ja lukot Rustissa

Rust on suhteellisen nuori ohjelmointikieli[2], joka on hengiltään lähempänä kieliä **C** ja **C++**, sillä se ei käytä automaattista roskienkerääjä ja on tarkoitettu lähinnä järjestelmäohjelmointiin ja muissa samantyyppisissä asianyhteyksissä, joissa kielen äärimmäinen tehokkuus ja nopeus ovat erittäin tärkeitä. Automaattisen roskienkeräyksen sijaan Rust hallinnoi muistin vapauttamista sekä turvallista käyttöä tarkkojen “arvojen omistukseen“ liittyvien sääntöjen avulla. Juuri nämä Rustille ominaiset säännöt tekevät tästä kielestä ainutlaatuisen. [6, chapter

[2]Kielen kehitys alkoi vuonna 2006 ja ensimmäinen stabili versio julkaistiin vuonna 2015

“Getting Started with Rust“, subchapter “What is Rust and why should you care“.]

Perinteisesti kaikissa imperaattivisissa ohjelmointikielissä kaikki arvot on vapaasti sijoitettavissa muuttujin, kunhan muuttujan tyyppi vastaa arvon tyyppiä. Erityisesti toisen muuttujan sisältämä arvo voidaan aina sijoittaa toiseen muuttujan, jolloin ohjelmassa on (ainakin) kaksi eri muuttujaa, jotka sisältävät tämän arvon (tai viittaavat siihen, kuten Javassa olioiden tapauksessa). Rustissa tämä ei yleisesti ottaen ole näin. Sen sijaan jokaisella arvolla on tyypillisesti **tasan yksi omistaja** eli muuttuja, joka “omistaa“ tämän arvon. Kun yhden muuttujan omistama arvo sijoitetaan toiseen muuttujaan, tapahtuu yksi seuraavista. Jos arvo on tarpeeksi “yksinkertainen“ (tarkemmin sanottu niin sanottu `Copy`-arvo, esimerkiksi jokin primitiivinen arvo, kuten luku tai boolean), sijoittaminen toisen muuttujan tekee arvosta uuden täydellisen **kopion**. Jos se on taas monimutkaisempi Java-olion tapainen arvo, ensimmäinen muuttuja “‘luovuttaa“‘ arvonsa omistuksen toiselle muuttujalle, menettää oman arvonsa eikä sitä voi enää käyttää, ainakaan kunnes siihen sijoitetaan toinen arvo. Molemmissa tapauksessa lopputulos on sama. Jokaista muistissa sijaitsevaa arvoa vastaa ohjelmassa tasan yksi muuttuja, joka omistaa sen. Kun tämän muuttujan näkyvyysalue loppuu, kieli automaattisesti vapauttaa kaiken muistin, jonka muuttuja omisti. Yhden omistajan sääntö on tässä kontekstissa tärkeä, sillä saman muistialueen kaksinkertainen vapauttaminen olisi vakava virhe, joka saattaa jopa kaataa ohjelman. [7, chapter 4.]

```
fn example_function() {
    let counter = 0;
    let counter2 = counter; // koska 0 on Copy arvo, siitä tehdään
    riippumaton kopio ja muuttuja counter on edelleenkin
    käytettävissä
    println!({}, counter); // ok, printtaa nollan
    let important_numbers = vec![1, 2, 3]; // kolmen luvun vektori
    println!({}, important_numbers[0]); // printataan vektorin
    ensimmäinen alkio
    let important_numbers_2 = important_numbers; // vektori ei ole
    Copy, joten sen omistajuus on nyt siirtynyt uuteen muuttujaan
    println!({}, important_numbers_2[0]); // ok
    // println!({}, important_numbers[0]);  // olisi kääntöaikainen
    virhe, important_numbers muuttuja on luovuttanut arvonsa, joten
    vektoriin ei pääse enää käsiksi sen kautta
```

```
} // kun funktiosta poistutaan muuttujan important_numbers_2 omistama
    muisti (vektori sekä sen sisältämät arvot) vapautuu
```

Koodiesimerkki 9: Omistuksen siirto ja muistin vapauttaminen Rustissa

Tällaiset säännöt auttavat vältämään monia virheitä ja ongelmia, erityisesti liittyen muistin varaamiseen ja vapauttamiseen, mutta ne eivät ole monissa tilanteissa kovin käytännöllisiä. Esimerkiksi näiden sääntöjen mukaan jos jokin ei `Copy`-arvo arvo välitetään funktion kutsun parametrina, sen omistus siirtyy tälle funktiolle, joten kutsuva puoli ei voi enää käyttää tätä arvoa (paitsi jos kyseinen funktio palauttaa sen takaisin paluuarvona). Usein tämä ei ole kovin järkevää - esimerkiksi funktion tehtävänä voi olla vain printata jotakin tai laskea jokin apuarvo käyttäen hyväksi jonkun olion sisältämiä arvoja. Tällaisia käyttötapauksia varten Rust tarjoaa **referensseja**. Nämä toimivat kuten **C**-kielestä tutut *osoittimet* (engl.*pointer*) paitsi, että referenssi on aina validi eli osoittaa muistiin, jossa todellakin sijaitsee tällä hetkellä ohjelmassa olemassa oleva arvo (jos kääntäjä ei pysty vakuuttumaan siitä, se ei suostu kääntämään ohjelmaa). Referenssin käyttö **ei** siirrä vastaavan arvon omistusta vaan ainoastaan “lainaa“ tämä arvo. [7, chapter 5.]

Rustin referenssit jakautuvat kahteen tyyppiin: “jaetut“ referenssit ja “eksklusiiviset“. Erityisiä tarkasti määriteltyjä poikkeuksia lukuun ottamatta arvon **päivittäminen** on mahdollista ainoastaan *eksklusiivisen* referenssin kautta; *jaettu* referenssi antaa mahdollisuuden vain *lukea* arvo. Lisäksi millä tahansa hetkellä ohjelmassa voi olla joko **tasan yksi eksklusiivinen** referenssi arvoon tai mielivaltainen määrä jaettuja referensseja samaan arvoon, mutta ei molempia samanaikaisesti. Jaettu referenssi muuttujaan `x` merkitään `&x` ja eksklusiivista merkitään `&mut x`. Lyhenne “mut“ tässä tulee sanasta “mutable“ ja viittaa siihen, että (yleensä) vain eksklusiivinen referenssi mahdollistaa arvon päivittämistä eli mutatointia. [7, chapter 5.]

Tarkastellaan nyt yhteisten jaettujen resurrsien käsittelyä monisäikeisessä ohjelmassa Rustin näkökulmasta. Omistussäännöistä seuraa, että oletusarvoisesti kielen muuttujat eivät yleensä voi edustaa tällaisia yhteisiä

resursseja erityisesti, jos ne pitää päivittää. Tämä johtuu monista syistä. Yksi liittyy siihen, että jokaista säiettä kielessä edustaa erityinen olio, joka yleensä **omistaa** kaikki arvot, joita se käyttää [8, chapter 16.1]. Tällöin mikään muu säie ei enää voi käyttää samoja arvoja, sillä tyypillisesti vain yksi omistaja on sallittu. Referenssit saattavat auttaa tässä, mutta jos yhteisiä resursseja pitää päivittää, tätä voi tehdä vain yksi tietty säie, sillä vain yksi aktiivinen eksklusiivinen referenssi arvoon on sallittu. Lisäksi tällöin muut säikeet eivät pääse edes lukemaan samaa dataa, sillä muut referenssit eivät ole sallittuja niin kauan, kuin yksi eksklusiivinen referenssi on aktiivinen.

Toinen, syvällisempi syy on siinä, että yleisesti ottaen referenssin vastaanottavalla säikeellä ei ole tarpeeksi takeita siitä, että tämä referenssin takana oleva arvo “elää“ niin kauan kuin se tarvitsee sen. Jos referenssi viittaa arvoon, joka on luotu toisessa säikeessä, se ei vaan “‘elää“‘ kauemmin kuin tämän arvon omistajan näkyvyysalue. Koska säikeet ovat riippumattomia ja jokaisen säikeen sisältämän koodin ajo voi kestää mielivaltaisen pitkään, tyypillisesti ei ole mitään takeita siitä, että toisen säikeen omistuksessa oleva arvo elää tarpeeksi kauan. Jos kääntäjä ei voi helposti vakuuttua siitä (ja yleensä se ei voi), se hylkää ohjelman. [8, chapter 16.1.]

Tässä vaiheessa saattaa kuulostaa siltä, että mahdollisuudet yhtäaikaisohjelmointiin Rustissa ovat niin rajoitettuja, että se on käytännössä mahdotonta. Tämä ei kuitenkaan ole näin. Päinvastoin kielestä löytyy erinomainen tuki yhtäaikaisuudelle, mutta kielen “tavalliset“ tietotyypit sekä referenssit niihin ovat tarkoituksella rajoitettu yllä tarkastetuilla säännöillä siitä syystä, että Rust ei haluakaan, että niitä jaettaisiin suoraan säikeiden kesken, sillä tällöin voidaan helposti saada aikaan tässä vaiheessa jo hyvin tunnettuja ongelmia kuten kilpailutilanteet tai epäkorrekti datan näkyvyys. Sen sijaan Rust tahtoo, että käytämme tiettyjä erityisiä datatyyppejä, jotka on valmiiksi suunniteltu sillä tavalla, että niiden käyttäminen on säieturvallista. Rajoittamalla meidän vapautta Rust suojaa meitä virheiltä itse, jolloin yhtäaikaisuutta käyttävää koodia on helpompaa kirjoittaa sekä vakuuttua sen oikeudellisuudesta.

Palautetaan mieleen ei-säieturvallisen laskurin käyttöön liittyviä esimerkkejä 1 ja 2. Näillä havainnollistimme datakilpailutilanteen, joka syntyy, kun kaksi säiettä pääsevät muokkaamaan samaa muuttujaa vapaasti. Nämä esimerkit muodostavat osan täysin validia Java-ohjelmaa, jonka Javan kääntäjä huoletta kääntää ajettavaksi ohjelmaksi, vaikka se tulee olemaan todella huono, rikkinäinen ohjelma. Rust-kielellä taas samanlaista ohjelmaa ei edes pystyy kirjoittamaan niin, että kielen kääntäjä suostui kääntämään sen! Tämä johtuu siitä, että ohjelmassa on yksinkertainen kokonaislukumuuttuja, joka yritetään vapaasti jakaa kahden säikeen välissä niin, että molemmat pääsevät vapaasti muokkaamaan sen arvoa - eikä tämä vaan onnistu Rustissa.

Javassa tämä ohjelma ei toiminut oikein, ja Rust kieltää sen kokonaan. Javassa yksi tapa korjata se on suojata yhteisen muuttujan päivityksiä lukolla (esimerkki 5). Myös Rust tarjoaa tähän lukitukseen perustuvan ratkaisun, mutta siinä on mielenkiintoisia piirteitä, jotka eroavat Javan lähestymistavasta.

Tältä esimerkkiin 5 perustuva, eli lukitusta käyttävä, toimiva ratkaisu voisi näyttää Rustissa:

```
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Mutex::new(0);

    thread::scope(|s| {
        for _ in 0..2 {
            s.spawn(|| {
                for _ in 0..10000 {
                    let mut counter_guard = counter.lock().unwrap();
                    *counter_guard += 1;
                }
            });
        }
    });

    let result = counter.into_inner().unwrap();
    println!("{}", result); // tämä printtaa aina 20000
}
```

Koodiesimerkki 10: Säieturvallinen laskuri Rustilla

Esimerkin 10 koodissa tapahtuu seuraavaa. Rivillä 5 määritellään `Mutex`-tyyppinen muuttuja, jota voidaan ajatella edustavan **lukolla suojattua kokonaislukua**. Rivillä 7-16 käynnistetään kaksi uutta säiettä, jotka molemmat suorittavat samaa koodia. Rust tarjoaa muutaman tavan luoda ja käynnistää uudet säikeet, tässä käytämme funktiota `thread::scope` [5, chapter 1, subsection “Scoped Threads“]. Tämä funktio antaa mahdollisuuden määritellä säikeet, joiden suorituksen loppumista pääsäie odottaa automaattisesti (jolloin `join`-funktiota ei tarvitse kutsua eksplisiittisesti). Funktio `thread::scope` ottaa argumenttina anonyymisen funktion, jonka ainoa parametri, yllä `s`, on **scope**-olio, joka ikään kuin edustaa `thread::scope` kutsuvan koodin ulkopuolista näkyvyysaluetta. Tämä olio osaa tarvittaessa ottaa halttuunsa omistusoikeuden tämän näkyvyysalueen muuttujia kohti - tai ottaa referenssejä niihin riippuen käyttötapauksesta. Tässä tapauksessa tällainen ulkopuolinen saman näkyvyysalueen muuttuja on juuri meidän laskuri `counter`. Rivillä 9 tämän olion metodilla `spawn` käynnistetään uusi säie. Sen suorittama koodi on tämän metodin argumenttina annettu anonyymifunktio. Jokaisen säikeen sisällä laskurin arvo kasvatetaan yhdellä 10000 kertaa. Kasvattaminen tehdään ei-atomisesti, tavallisella operaattorilla +=. Kilpailutilanetta ei kuitenkaan tapahdu, sillä ensin lukko lukitaan rivillä 11. Tämä toimii kuten yleensä - jos lukko on toisen säikeen omistuksessa, odotetaan kunnes se vapautuu. Lukitusoperaatio rivillä 11 palauttaa muuttujan `counter_guard`, jonka tyyppi on `MutexGuard` [5, chapter 1, subsection “Locking: Mutexes and RwLocks - Rust’s Mutex“]. Tämä muuttujan avulla lukitetussa koodilohkossa voidaan päästää lukolla suojatun arvon (meidän laskuri) käsiksi. Juuri tämä tapahtuu rivillä 12, sillä `*counter_guard` on suora referenssi laskurin arvoon [5, chapter 1, subsection “Locking: Mutexes and RwLocks - Rust’s Mutex“]. Muuttuja `counter_guard` vastaa myös lukon avaamisesta - lukko vapautuu automaattisesti, kun tämä muuttuja poistuu näkyvyysalueesta, mikä tapahtuu rivillä 13 [5, chapter 1, subsection “Locking: Mutexes and RwLocks - Rust’s Mutex“]. Siksi koodista ei löydy eksplisiittista lukon avaamisoperaatiota.

Rivillä 16 funktiosta `thread::scope` poistutaan vasta kun kumpikin sen sisällä käynnistetty säie on lopettanut tehtävänsä [5, chapter 1, subsection “Scoped

Threads“]. Rivillä 18 `Mutex`-muutujan sisältä kaivataan laskurin nykyinen arvo funktiolla `into_inner` [5, chapter 1, subsection “Locking: Mutexes and RwLocks - Rust’s Mutex“]. Tämä itse asiassa ottaa `Mutex`-muuttujan omistuksen haltuunsa, joten sitä ei voi enää käyttää tämän kutsun jälkeen. Lopputuloksena ohjelma printtaa aina 20000 eli toimii korrektisti ilman datakilpailuja.

Periaatteen tasolla tämä ohjelma on strukturoitu kuten vastaava aikaisempi tarkasteltu esimerkki 5. On kuitenkin ainakin yksi hyvin oleellinen ero. Javassa lukko ja data, jonka se suojaa, ovat toisistaan täysin riippumattomia olioita. Kieli itse ei anna työkaluja ilmaista niiden välistä suhdetta, eikä tarkista sitä. Tämä jää täysin ohjelmoijan vastuulle, ja reaalielämän koodissa joudutaan asia dokumentoimaan erikseen, sekä muistaa ylläpitää tällaista dokumentaatiota ajan tasolla. Rustissa taas `Mutex` on lukko, joka samalla inkapsuloi sisäänsä dataa, jonka se suojaa. Datan päivittää, tai edes pelkästään lukeminen, voi tapahtua vain tämän lukko-objektin julkisen APIn kautta, joka on valmiiksi säieturvallinen. Esimerkiksi metodi `lock` palauttaa `MutexGuard`-olion, jonka kautta voidaan suoraan päivittää dataa. Tämä metodi kuitenkin takaa, että ennen kuin tämä olio palautetaan, vastaava lukko lukitaan, jolloin mikään muu säie pääsee saman datan käsiksi kunnes lukko vapautetaan [5, chapter 1, subsection “Locking: Mutexes and RwLocks - Rust’s Mutex“]. Tämän jälkeen voidaan vapaasti ja turvallisesti käsitellä lukon sisällä inkapsuloitua dataa tämän `MutexGuard`-olion avulla. Tämä johtuu siitä, että niin kauan kuin tämä olio on olemassa, lukko säilyy lukittuna. Lukko vapautetaan vasta kun `MutexGuard`-muuttuja on poistunut näkyvyysalueesta eikä ole enää käytettävissä. Tämän jälkeen data on taas turvallisesti “piilossa“ lukon sisällä ja siihen ei pääse käsiksi kunnes lukko on taas lukittu. Tämä on hyvin erillaista kuin Javassa, jossa mikään ei estää käsittelemään virheellisesti suoraan sellaista dataa, joka olisi pitänyt suojata ensin lukolla.

Voidaan huomata myös seuraavaa: koodiesimerkissä 10 pystymme jakamaan yhteisen `Mutex`-muuttujan kaikkien ohjelman säikkeiden kesken - kumpikin funktion `thread::scope` käynistetty säie, sekä pääsäie itse pääsevät “vapaasti“ (koodin näkökulmasta) lukea ja päivittää `Mutex`-muuttujaan sisäistää tilaa, joka on

itse kokonaislukulaskuri. Tämä on esimerkki tilanteesta, jossa Rust tarkoituksella lieventää muuten ankaria omistussääntöjään. Tämä on kielen näkökulmasta ok, sillä lukko on rakennettu täysin säieturvalliseksi eikä sen sisäistä tilaa pysty korruptoimaan korrektissa ohjelmassa, jonka kielen kääntäjä suostuu kääntämään. Tässä tapauksessa muuttujaa ei edes tarvitse merkitä `mut`-avainsanalla, vaikka sen sisäinen tila mutatoidaan.

Mainittakoon vielä, että `Mutex` pitää myös automaattisesti itse huolta korrektista datan näkyvyydestä säikeiden kesken.

Kun puhuimme lukoista Javan kontekstissa, käytimme usein ilmaisua “lukolla suojattu *koodi*“. Rustin lähestymistapa seuraa periaatetta “suojaa lukolla *dataa*, ei koodia“. Tällä on ilmiselviä hyviä puolia. Rustissa eri säikeiden kesken jaettuja resursseja on **pakko** inkapsuloida `Mutex`in kaltaiseen “konteineriin“, joka suojaa sen yhtäikaisessa käytössä automaattisesti ohjelmoijan puolesta itse. Lukituspolitiikasta ei myöskään tarvitse pitää erillistä dokumentaatiota, se on “dokumentoitu“ koodissa.

Yhteenveto - Rust vs. Java

Lukot ja muut työkalut, joiden avulla Rustissa voidaan käsitellä jaettuja resursseja yhtäaikaisskenaariossa, ovat turvallisempia kuin Javassa ja muissa samantyyppisissä kielissä. Tämä johtuu siitä, että kun kieli suunniteltiin 2000-luvulla, klassiset ongelmat, jotka liittyvät tähän aiheeseen, olivat jo hyvin tiedossa, joten Rustin luojat tietoisesti halusivat tehdä kielestä mahdollisimman säieturvallisen. Siinä missä vastaava Java-ohjelma sisältää virheitä, mutta voidaan ajaa reaalielämässä, samanlainen ohjelma Rustilla ei yksinkertaisesti edes menisi kielen kääntäjän läpi.

Rustin lähestymistavan hyödyt koskevat kuitenkin lähinnä sellaisia asioita kuten datakilpailujen mahdottomuus sekä lukituspolitiikan ylläpito kielen tasolla. Jos tarkastellaan muita lukkoihin liittyviä ongelmia, jotka mainitsimme edellisessä alaluvussa, voidaan todeta, että Rustissa ne ovat yhtä paljon ongelmallisia kuten Javassa sekä muissa vastaavissa kielissä, eikä Rust tarjoaa mitään uutta tai

ylimääräistä niiden ratkaisemiseksi. Erityisesti siis lukkiuma, livelock tai lukon vapauttamiseen liittyvän odotuksen tuoma performanssi-isku ovat Rustissa edelleenkin samanlaisia ongelmia, missä ne ovat muissakin kielissä.

Onneksi lukkojen lisäksi kaikissa meidän tarkastelemissa kielissä on olemassa muitakia työkaluja, jotka auttavat jakamaan turvallisesti resuresseja eri säikeiden kesken. Lisäksi näillä työkaluilla ei ole samantyyppisiä ongelmia kuin lukoilla, kuten esimerksiksi lukkiuma. Kyseessä on *atomiset muuttujat* ja niihin tutustumme seuraavaksi.

3.2 Atomiset muuttujat

Luvussa 3.1 motivoimme lukon käsitettä esmerkin 1 avulla. Siinä esimerkissä kilpailutilanne johtui oleellisesti siitä, että operaatio ++ *ei ole atominen*. Lukitus ei ole kuitenkaan ainoa ratkaisu, jonka Java, Rust tai muut vastaavat kielet tarjoavat tähän tarkoitukseen.

Itse asiassa kenties jopa luonnollisempi ratkaisu seuraa melkein triviaalisti ongelman kuvauksesta. Jos kerran operaatio ei ole atominen, tehdään siitä sellainen!

Tällaisia epätriviaaleja atomisia operaatioita tarjoavia muuttujia (tai niitä vastaavia tietorakenteita ja tyyppejä) sanotaan **atomisiksi**.

3.2.1 Atomiset muuttujat Javassa

Javan standardikirjasto sisältää monta atomisia operaatioita tarjoavia luokkia, esimerkiksi `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, `AtomicReference` [9]. Nämä vastaavat “tavallisia“ Javan tyyppejä - esimerkiksi primitiivisiä `int`, `long`, `boolean`-tyyppejä, tai jopa oliotyyppejä (`AtomicReference`), mutta tarjoavat samalla atomisia ja *ei-blokkaavia*, lukkovapaita versiota operaatioista, jotka eivät tavallisesti ole sellaisia.

Esimerkiksi käyttämällä `AtomicInteger`-luokkaa voidaan korjata edellä tarkasteltu ei-säieturvallisen laskurinesimerkki 1 seuraavalla tavalla:

```
AtomicInteger count = new AtomicInteger(0);

class CountingThread extends Thread {
    public void run() {
        for (int i = 0; i < 10000; i++) {
            count.incrementAndGet();
        }
    }
}

Thread newThread1 = new CountingThread();
Thread newThread2 = new CountingThread();

newThread1.start(); newThread2.start();
newThread1.join(); newThread2.join();

System.out.println(count.get()); // Printtaa aina 20000.
```

Koodiesimerkki 11: Atomisen muuttujan yhtäaikainen käyttö Javassa

Luokan `AtomicInteger` metodi `incrementAndGet` on funktionaalisesti ekvivalentti operaation ++ kanssa (postfix muodossa). Se kasvattaa arvoa yhdellä ja palauttaa uuden arvon. Ainoa ero on siinä, että kaikki tämä tapahtuu atomisesti, joten toinen säie ei voi tulla väliin tämän operaation kesken. Luokka `AtomicInteger` pitää myös huolta korrektista näkyvyydestä, jolloin kaikki säikeet näkevät olion aina ajankohtaisessa tilassa. Toisin sanoen tämä implementaatio on yhtä säieturvallinen kuin lukkoja käyttävä ratkaisu 5. Lisäksi, koska siinä ei käytetä lukitusta, tämä lähestymistapa on vapaa sellaisista ongelmista kuin säikeen turha blokkaus lukkon vapauttamista odottaessa, lukkiuma, livelock jne.

3.2.2 Atomiset muuttujat Clojuressa

Clojure-kielessä atomisia muuttujia sanotaan yksinkertaisesti “**atomeiksi**“. Clojuren atomit ovat yksi esimerkki kielen niin sanotuista “*referenssityyppeista*“, joita tarkastelemme tarkemmin myöhemmin alaluvussa Muuttumaattomuus ja referenssimuuttujat. [1, chapter 4, day 1.]

Uusi atomi luodaan Clojuressa funktiolla `atom` ja sen arvo päivitetään funktiolla `swap!`. Atomin sisältämä arvo saadaan funktiolla `deref` tai referenssioperaatorilla `@`. [1, chapter 4, day 1.]

```
(def my-atom (atom 0)) ;; luodaan uusi "my-atom" niminen muuttuja,
    jonka alkuarvo on 0
(deref my-atom) ;; atomin arvo tällä hetkellä on 0
(swap! my-atom inc) ;; funktio inc kasvattaa numero yhdellä
@my-atom ;; nyt arvo on 1, koska (inc 0) on 1
(swap! my-atom + 3)
@my-atom ;; nyt arvo on 4
```

Koodiesimerkki 12: Clojuren atomin luominen ja käyttö

Kuten esimerkeistä nähdään, funktiolle `swap!` annetaan argumentteina atomi ja funktio, joka sovelletaan atomin nykyiseen arvoon. Funktion palauttamasta arvosta tulee atomin seuraava arvo. Jos funktio ottaa enemmän kuin yhden argumentin, loput parametrit voidaan antaa `swap!`-funktiolle lisäargumenttina [1, chapter 4, day 1]. Esimerkiksi yllä rivillä `(swap! my-atom + 3)` atomiin nykyiseen arvoon `1` sovelletaan funktio + eli yhteenlasku, jolle annetaan myös toinen argumentti `3`. Lopputuloksena atomin uudeksi arvoksi tulee yksi plus kolme eli neljä.

Koska Clojure on dynaaminen kieli, jossa muuttujalla ei tarvitse olla pysyvää tyyppiä, Clojuren atomi voi sisältää mielivaltaisen Clojure-arvon, jolla voi olla mikä tahansa tyyppi (ja tyyppi voi vaihtua ohjelman ajon aikana). Tämä on ainakin syntaksisesti erilaista kuin Javassa, jossa atomisella muuttujalla on tietynlainen tyyppi, joka ei saa muuttua ohjelman ajoaikana. Muuten atomit Clojuressa ja Javan atomiset muuttujat ovat melko samanlaisia. Yleisemmin atomisia muuttujia löytyy hyvin monesta nykyohjelmointikielestä, muun muassa myös Rustissa, ja tyypillisesti ne toimivat kaikissa kielessä suurin piirtein samalla tavalla [5, chapter 2].

3.2.3 Atomiset muuttujat ja datan näkyvyys

Javan atomisilla muuttujalla on hyvin vahvoja datan näkyvyyteen liittyviä ominaisuuksia. Nimittäin Javassa aina kun atominen muuttuja päivitetään, myös

kaikkien *muidenkin muuttujien muutokset*, jotka suoritetaan koodissa *ennen tätä päivitystä* samassa säikkeessä, näkyvät korrektisti myös jokaiselle säikeelle, kun se *lukee* saman atomisen muuttujan arvon seuraavan kerran [9]. Tässä “muut muuttujat“ tarkoitavat siis ihan mitä vaan yhteisiä muuttujia, niiden ei tarvitse olla atomisia tai olla lukolla suojattuna.

Clojure on hostattu kieli, joka käyttää taustalla Javan virtuaalikonetta JVM, joten todellisuudessa Clojuren atomit on rakennettu vastaavien Javan työkalujen päälle [10]. Erityisesti niillä on datan näkyvyyden suhteen samanlaisia ominaisuuksia kuin vastaavilla Javan atomisilla muuttujilla.

Rustissa atomiset muuttujat tarjovat monipuolisempia mahdollisuuksia Javaan ja Clojureen verrattuna. Niiden operaatiot voidaan hienosäätää lisäparametrilla, jonka tyyppiä sanotaan “muistijärjestykseksi“ ja jonka arvo vaikuttaa mahdollisesti yhteisen datan näkyvyyteen eri säikkeiden kesken. Käyttämällä tiettyjä arvoja muistijärjestysparametria on mahdollista saada aikaan yhtä vahvoja näkyvyysseurauksia kuin Javassa tai Clojuressa, mutta jos niitä ei tarvita, voidaan käyttää näkyvyyden suhteen heikompia, mutta suorituksen kannalta tehokkaampia muistijärjestysarvoja. Tätä monimutkaista mekanismia Rust on itse asiassa lainannut C++-muistimallilta. [5, chapter 3.]

Näin ollen, kaikissa kolmessa kielessä, joita tarkastelemme, atomisten muuttujien avulla voidaan ratkaistaa myös datan näkyvyyteen liittyviä ongelmia.

3.2.4 Miten atomiset muuttajat toimivat

On tärkeätä tiedostaa, että atomiset muuttujat eivät ole mitään “syntakstista sokeria“ lukkojen päällä, vaan ne on tyypillisesti implementoitu ilman lukitusta käyttämällä suoraan hyväksi tietokonesuorittimien tarjoamia atomisia operaatioita . Itse asiassa pätee päinvastainen suhde - hyvin usein lukot implementoidaan käyttämällä muun muassa tietokoneen ytimen tarjoamia matalatasoisia atomisia operaatioita. Tämä johtuu pohjimmiltaan siitä, että operaation “tarkista onko lukko vapaa ja jos on, ota se halttuun“ täytyy olla atominen, jotta lukko toimisi oikein. [2,

s. 319, 321–324.]

Koska atomiset operaatiot ovat lukkovapaita, ne eivät ole **blokkaavia**. Tämä tarkoittaa sitä, että säikeen käsittelemät atomiset resurssit eivät ole “lukittuja“ eli kun yksi säie suorittaa niille jonkun atomisen operaation, muutkin säikeet voivat lukea tai jopa yrittää päivittää samoja muuttujia. Toisin sanoen muut säikkeet eivät joudu silloin odotustilaan, kuten olisi käynyt lukkoja käyttäessä. [2, s. 321.]

On selvä, että jos kaksi säiettä yrittävät päivittää saman atomisen muuttujan arvon “samanaikaisesti“, kyseessä on konflikti - juuri sama konflikti, joka on tämän työn pääaiheena. Tässä vaiheessa tunnemme vain yhden tavan ratkaista tämä konflikti, eli lukitus. Lukitus voidaan ajatella olevan “pessimistinen“ lähestymistapa - oletetaan pahinta ja lukitaan resurssi operaation ajaksi kaiken varalta riippumatta siitä, yrittävätkö muut säikkeet oikeasti päästää samaan aikaan siihen käsiksi.

Atomisten muuttujien lähetystapa on sen sijaan “optimistinen“. Ei välitetä muiden säikkeiden olemassaolosta ja yritetään päivittää yhteinen resurssi joka tapauksessa. Lopussa, ennen kuin tämä päivitys astuu todellakin voimaan, tarkistetaan, onko toinen säie mahdollisesti ehtinyt jo vaihtaa sen sillä aikaa. Jos näin on käynyt, hylätään tämä yritys ja kokeillaan päivitys uudestaan alusta.

Teknisemmin ilmaistuna konflikti ratkaistaan niin sanotulla “*compare and set*“-strategialla. Tämä toimii seuraavasti. Kun atomin päivityspyynnön käsittelyä aloitetaan, ensin “pannaan merkille“ muuttujan nykyinen arvo operaation alussa. Uuden arvon laskemisen jälkeen suoritetaan niin sanottu *compare and set* **atominen** operaatio, jonka nykyiset tietokonesuorittimet tarjoavat raudan tasolla. Tämä operaatio ensin tarkistaa, onko atomisen muuttujan arvo edelleenkin sama kuin operaation alussa. Jos on, kaikki on hyvin, konfliktia ei tullut ja päivitysoperaatio voidaan suorittaa loppuun onnistuneesti. Jos taas huomataan, että arvo on muuttunut, tämä tarkoittaa sitä, että jokin toinen säie on ehtinyt vaihtaa atomin arvo sen jälkeen, kun tämä päivitysoperaatio on alkanut. Tällöin palataan alkuun ja kokeillaan koko päivitysoperaatio uudestaan - muuttujan

uudella arvolla. Tämä jatkuu niin kauan, kunnes operaatio onnistuu. [2, s. 321-324; 1, chapter 4, day 1, section “Retries“.]

Korostetaan vielä, että itse “*compare and set*“-operaatio on atominen, joten mikään muu säie ei voi ehtiä vaihtaa atomin arvoa sen jälkeen, kun muuttujan arvo on luettu sekä verrattu sen alkuperäisen arvoon, ja ennen kuin uusi arvo mahdollisesti asetetaan. Jotta tämä toimisi, tietokoneen raudan pitää tarjota tällainen atominen operaatio. Tämä päde käytännössä kaikkiin nykyaikaisiin tietokoneisiin. [2, s. 321.]

Edellisestä seuraa, että atomisen muuttujan päivitys joutuu kutsumaan päivitysfunktiota mahdollisesti monta kertaa, kunnes lopulta onnistuu. Tästä johtuen atomisen muuttujan päivitysfunktion täytyy olla **vapaa sivuvaikutuksista**. Clojuressa tämä rajoitus voidaan kiertää liittämällä atomiin niin sanottu `watcher`-funktio. Tämä kutsutaan tasan kerran sen jälkeen kun atomin tila on onnistuneesti muuttunut. [1, chapter 4, day 1.]

Koska atomisten muuttujien operaatiot eivät ole blokkaavia, niiden käyttö on monissa tilanteissa tehokkampaa ja johtaa nopeampiin ohjelmiin kuin lukkoja käyttäessä. Toinen hyvin puoli niissä on siinä, että lukituksen poissaollessa sellaiset ikävät ongelmat kuten esimerkiksi lukkiuma eivät voi tapahtua. Live-lockin tai säikkeen nälkiintymisen tapaiset tilanteet voi kuitenkin edelleenkin tapahtua, joskin ovat käytännössä hyvin harvinaisia. Esimerkiksi periaatteessa atominen operaatio voi joutua ikuiseen silmukkaan, jos jokainen eritys päivittää muuttujan arvo epäonnistuu. [2, s. 322, 323.]

Aidosti lukko-vapaiden atomisten operaatioiden joukko on kuitenkin melko suppea, sillä sen rajoittaa luonnollinen tietokoneen suorittimien atomisten primittivisten operaatioiden tarjonta. Yleisesti ottaen suorittimet osaavat natiivisti suorittaa tällaisia operaatioita vain suhteellisen pienikokoisille muuttujille, joiden arvo mahtuu rekisteriin (eli niin sanotun “konesanan“ kokoinen). Toisin sanoen lukko-vapaat atomiset operaatiot eivät voi olla mielivaltaisen isoja. Silloin kun monimutkaisemasta operaatiosta tai muutaman operaation jonosta halutaan

tehdä “atominen“, ei vältämättä ole muita vaihtoehtoja kuin lukkojen käyttö. Tästä syystä atomiset muuttujat eivät tarjoaa täysivaltaista korvausta lukoille. Reaalielämässä tarvitaan molempia työkaluja.

3.3 Volatile-muuttujat

Oikein käytettynä sekä lukot että atomiset muuttujat auttavat välttämään sekä kilpailutilanteita että takamaan datan korrektia näkyvyyttä. Java (sekä Clojure) tarjoaa yhteisen datan synkronointiin vielä yhden työkalun, niin sanottuja `volatile`-muuttujia. Nämä ovat sekä lukkoja että atomisia muuttujia heikompia. `volatile`-muuttujat eivät pysty estämään datakilpailuja, ainoastaan takaamaan korrektia datan näkyvyyttä eri säikeiden kesken.

Muuttujalle, jotka on merkitty `volatile`-avainsanalla, Javan muistimalli tekee samoja vahvoja datan näkyvyyten littyvia lupauksia kuin atomisille muuttujille. Tarkemmin sanottuna aina kun `volatile`-muuttuja päivitetään, kaikki *muidenkin muuttujien muutokset*, jotka sijaitsevat koodissa *ennen tätä päivitystä* ja on suoritettu samassa säikeessä, näkyvät myös jokaiselle säikeelle kun se *lukee* tämän `volatile`-muuttujan arvon seuraavan kerran. Lisäksi myös `long` ja `double`-muuttujien päivitykset ovat Javassa aina atomisia, jos kyseessä on `volatile`-muuttuja. Lyhyeesti sanottuna Javassa `volatile`-muuttujilla on samat ominaisuudet kuin atomisilla muuttujilla, paitsi että ne eivät tarjoaa mitään erityisiä atomisia operaatioita. [2, s. 36-38.]

Edellisestä seuraa, että datan näkyvyyden suhteen `volatile`-muuttujan (tai atomisen muuttujan) päivittämisellä on sama efekti kuin lukon vapauttamisella, ja vastaavasti sen lukemisella on sama efekti kuin lukon halttuunottamisella. Lisäksi kun `volatile`- tai atomisia muuttujia käytetään, voidaan olettaa, että tiettyjen koodin rivien järjestystä ei muuteta, kun koodia ajetaan (tai ainakin ohjelman suoritus aina näyttää siltä).

Ajemmin esimerkissä 3 esitetty `DataVisibilityProblems`-luokka voidaan korjata yhdellä pienellä muutoksella - tehdään muuttujasta `readyToReadValue` `volatile`:

```
public class NoDataVisibilityProblems  {
    static volatile boolean readyToReadValue = false;
    static long value = 0;

    static Thread t1 = new Thread(() -> {
        value = Long.MAX_VALUE;
        readyToReadValue = true;
    });

    static Thread t2 = new Thread(() -> {
        while (!readyToReadValue) {}
        System.out.println(value);
    });

    public static void main(String[] args) throws
        InterruptedException {
        t1.start(); t2.start();
        t1.join(); t2.join();
    }
}
```

Koodiesimerkki 13: Volatile-muuttujan sovellus Javassa

Nyt kun muuttuja on `volatile` ja säie `t1` asettaa sen arvoksi `true`, säie `t2` heti näkee tämän muutoksen, kun se lukee `while`-silmukassa sen arvon seuraavan kerran. Tällöin silmukka päättyy. Lisäksi, koska koodissa rivi `value = Long.MAX_VALUE` sijaitsee ennen riviä, jossa `volatile`-muuttuja päivitetään, sen vaikutus näkyy säikeelle `t2`, kun se poistuu silmukasta. Seurauksena tämä säie printtaa seuraavaksi oikean, päivitetyn arvon muuttujalle `value`. Tässä ratkaisussa ei ole käytetty yhtäkään lukkoa, eikä edes yhtäkään atomista muuttujia ja silti se on nyt täysin säieturvallinen. Tämä johtuu siitä, että datan näkyvyys oli tämän esimerkin koodin ainoa potentiaalinen ongelma, joten siinä ei ole tarvetta atomisille operaatioille.

`volatile`-muuttujat sopivat erityisen hyvin tilanteisiin, jossa vain yksi säie päivittää muuttujan arvoa, mutta mielivaltainen määrä muita säikeitä lukee muuttujan arvon edellyttäen, että tämän muuttujan arvon päivityksen ei tarvitse olla atominen.

Clojure tarjoaa `volatile!`-funktion, jonka avulla voidaan konstruoida muuttujia,

jotka käyttäytyvät kuten Javan `volatile`-muuttujat. Koska Clojure on hostattu kieli, joka toimii Javan virtuaalikoneen päälle, verhon takana nämä eivät oikeastaan olekaan mitään muuta kuin Javan `volatile`-muuttujat. [11, chapter 5, subchapter “Low-level concurrency“.]

3.4 Muuttumattomuus ja Clojuren referenssimuuttujat

3.4.1 Muuttumattomuus

Tässä vaiheessa olemme oppineet seuraavan tosiasian: oleellinen syy siihen, miksi yhteisten resurssien käyttäminen monisäikeisessä kontekstissa on vaikeata, on näiden resurssien **päivittämiseen** liittyvät haasteet. Epätriviaaleja työkaluja tarvitaan siitä syystä, että ilman sellaisia työkaluja joko resursseja **muutetaan** eri säikeissä mahdollisesti hallitsemattomasti (kts. luku Kilpailutilanne), tai siitä, että toisen säikeen tekemät muutokset eivät välttämättä näy toiselle säikeelle ajoissa (kts. luku Datan näkyvyys).

Voidaan siis todeta, että “pahan alkujuuri“ tässä tapauksessa on juuri resurssien sisältämien arvojen **suora muuttaminen**. Tästä väistämättä seuraa hyvin yksinkertainen oivallus - mitä vähemmän ohjelman käsittelemä data muuttuu, mitä enemmän aikaa se vie “read-only“-tilassa, sitä helpompaa on käyttää hyväksi sekä ylläpitää yhtäaikaisuutta. Erityisesti, jos jokin yhteisessä käytössä oleva data *ei muutu lainkaan*, sen käyttö yhtäaikaisessa kontekstissa on *käytännössä ongelmatonta*.

Dataa, joka ei koskaan muutu sen jälkeen, kun se luodaan, sanotaan **muuttumattomaksi** (engl. **immutable**). Imperatiivisessa ohjelmoinnissa muuttumatonta dataa käytetään tyypillisesti hyvin vähän, sen sijaan idiomaattisesti ohjelman toiminta perustuu pitkälti muuttujien päivittämiseksi “paikalla“ (engl. *in place*). Sellaisessa ohjelmointiparadigmassa on hyvin arkipäivästä, että vanhan arvon päälle kirjoitetaan uusi arvo suoraan samaan muistiosoitteeseen tuhoamalla vanha arvo ilman mitään mahdollisuuksia palauttaa sitä tarvittaessa. Nykyiset mainstream-olio-ohjelmointikielet, tyypillisenä

esimerkkinä Java, ovat perineet imperatiivista kielistä tämän lähestymistavan. Se on tehokas (ohjelman käyttämien resurssien mielessä) ja usein se vastaa hyvin ohjelmoijan luonnollista intuitiota siitä, mitä tapahtuu.

Olemme kuitenkin nähneet monta esimerkkiä siitä, että kaikessa luonnollisuudessaan tämä ajattelutapa osoittautuu ongelmalliseksi erityisesti silloin, kun siirrytään yhtäaikaisohjelmointiin. Päinvastoin, kun jokin “muuttuja“ itse asiassa onkin todellisuudessa muuttumaton, sen käyttö on *täysin säieturvallista*.

Java, sekä monet muut samantyyppiset kielet, tarjoavat jonkin verran työkaluja, joiden avulla voidaan luoda ja ylläpitää muuttumattomia oliota - esimerkiksi `final`-muuttujan käsitteen. Näitä pitää osata kuitenkin käyttää oikein. Esimerkiksi jos pääsy `final`-muuttujaan ei ole täysin *inkapsuloitu* luokan sisälle, mikään ei estä ulkopuolista koodia muuttamasta sen sisältöä. Tässä on yksinkertainen esimerkki tämäntyyppisestä ongelmasta.

```
class NotReallyImmutable {
    private final List<String> importantConstants =
        Arrays.asList("Java", "Clojure", "Rust");

    public List<String> getImportantConstants() {
        return this.importantConstants;
    }
}
```

Vaikka kenttä `importantConstants` on muuttumaton luokkansa sisäisen koodin näkökulmasta, metodin `getImportantConstants` avulla kuka tahansa tämän luokan käyttäjä (ja erityisesti mikä tahansa säie) voi saada referenssin tämän kentän arvoon `List`-oliona ja muuttaa sen sisältöä tämän luokan APIn avulla:

```
NotReallyImmutable obj = new NotReallyImmutable();
List<String> constants = obj.getImportantConstants();
constants.add("C++");
```

Jos kenttä `importantConstants` halutaan oikeasti muuttumattomaksi, viite siihen ei pitäisi koskaan paljastaa julkisissa metodeissa ja kentissä, joita luokka tarjoaa.

Tämä on taas helpompaa sanoa kuin tehdä, koska Java ei tarjoa mitään mekanismeja tämän automatisoinniksi. Käytännössä kehittäjän pitää vain olla äärimmäisen tarkkaa ja noudattaa tarkasti tätä sopimusta. Ongelma on pohjimmiltaan täysin samanlainen kuin sopimuksiin perustavaan lukituspolitiikan kohdalla.

3.4.2 Persistentit tietorakenteeet

Yksi syy siihen, miksi Javassa on vaikeaa soveltaa ja ylläpitää muuttumattomuutta korrektisti, piilee siinä, että sen tarjoamat perustietorakenteet, esim. taulukot, listat, hajautustaulut jne., eivät yleisesti ottaen ole muuttumattomia.

Niin sanotut **funktionaaliset ohjelmointikielet** sen sijaan tyypillisesti rakentavat koko paradigmansa muun muassa **muuttumattomien objektien ja tietorakenteiden varaan**. Esimerkiksi Clojure on funktionaalinen kieli, ja **kaikki** siihen sisäänrakennetut perustietorakanteet, esimerkiksi vektorit, listat, hajautustaulut sekä joukot ovat valmiiksi **muuttumattomia**. Aina kun tällaisen kokoelman sisältöä halutaan “muuttaa“ (esimerkiksi lisätä siihen uusi alkio), suoraan muokkaamisen sijaan tietorakenteesta tehdään **uusi versio**, uusi **muuttumaton** kokoelma, johon on sovellettu halutut “muutokset“. Vanha versio kokoelmasta jää tällöin elämään muuttumattomana sellaiseen tilaan, missä se on aina ollut. [1, chapter 4, day 1.]

Javan standardikirjastosta löytyvät itse asiassa niin sanotut `CopyOnWrite`-kokoelmat, jotka toimivat samalla logiikalla. Tällainen kokoelma pitää `private`-muuttujana viitettä kokoelman tämänhetkiseen sisältöön. Tämän viiteen takana olevaa oliota ei koskaan muokata suoraan, vaan kokoelman “‘päivittäminen“‘ hoidetaan tekemällä sisällöstä täysin uusi **kopio** ja vaihtamalla sisäinen viite sisältöön vanhasta arvosta uuteen. Tällöin ulkopuolisen havaitsijan näkökulmasta tietorakenne näyttää muuttuvan, mutta koska se tehdään käyttämällä muuttumattomia oliota, tietorakenteen käyttö on säieturvallista. [2, s. 86-87.]

Ilmiselvästi tämä strategia ei ole kuitenkaan kovin tehokas käytännössä. Kokoelman sisällön täydellinen kopiointi joka kerta, kun se halutaan muuttaa, on hidasta ja käyttää paljon muistia. Mitä isompi kokoelma on, siitä kallimaksi kopionti tulee. Tästä syystä Javan `CopyOnWrite`-kokoelmilla on suhteellisen suppea käyttökelpoisuus - niitä kannattaa käyttää ainoastaan tilanteissa, joissa kokoelma päivitetään harvoin (ja lisäksi se pysyy suhteellisen pienenä).

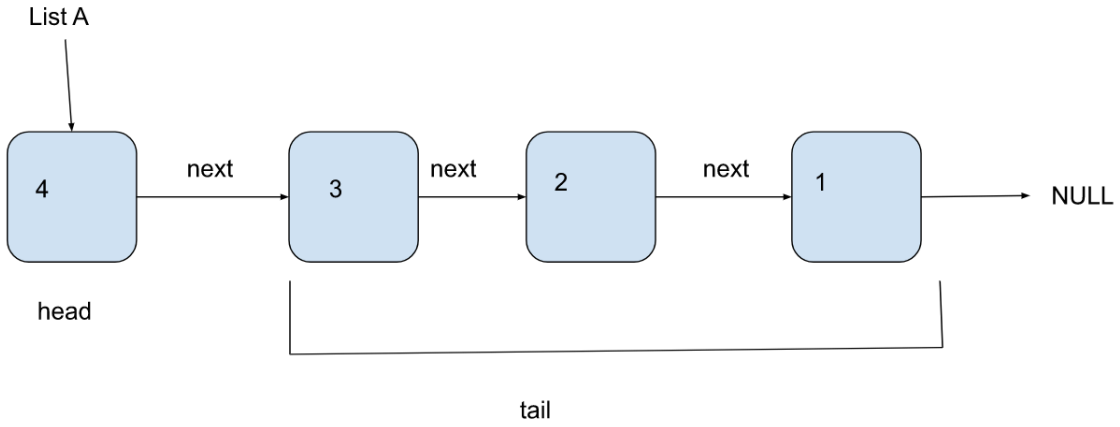
Syy siihen, miksi Javan `CopyOnWrite`-kokoelmat eivät kykene kilpailemaan täysin perinteisten, muuttuvien tietorakenteiden kanssa on ironisesti siinä, että ne on implementoitu juuri näiden perinteisten tietorakenteiden avulla. Tällaiset tietorakenteet toimivat tehokkaasti, jos niitä muokataan“in-place“, mutta niiden suora, “naiivi“ kopioiminen on usein epäkäytännöllistä.

Funktionaaliset ohjelmointikielet, mukaan lukien Clojure, käyttävät samanlaista kopiontistrategiaa, mutta se toimii Javan `CopyOnWrite`-kokoelmia tehokkaammin, koska taustalla tämä perustuu niin sanottuihin **persistenteihin** (engl. **persistent**) muuttumattomiin tietorakenteisiin.

Persisentti tietorakenne on tietorakenne, joka säilyttää sekä ylläpitää oman “historiansa“ sillä tavalla, että jokainen sen “versio“ on täysin muuttumaton. Lisäksi se tekee sen suhteellisen tehokkaasti. Oleellinen ero Javan `CopyOnWrite`-kokoelmiin näissä on siinä, että persistentissa tietorakenteessa uusi versio ei ole suoraan naiivi täysi kopio vanhasta, vaan sen sijaan **jakaa fiksusti osan omasta rakenteesta** sen kanssa. [1, chapter 4, day 1.]

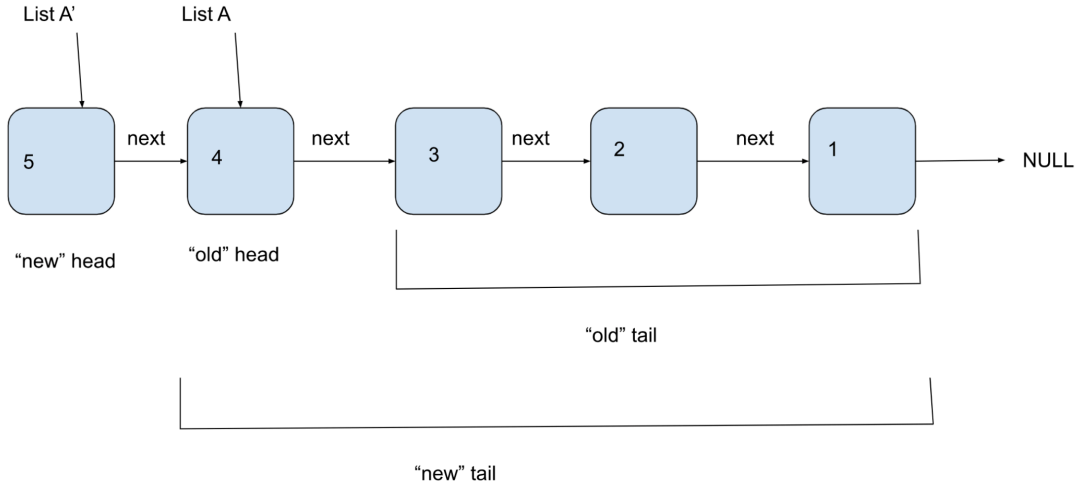
Yksinkertaisin tietorakenne, joka havainnollistaa, miten persisentit struktuurit toimivat käytännössä, on (yhteen suuntaan) *linkitetty lista* (engl. *singely linked list*). Oletamme, että tämä struktuuri on tuttu lukijalle tietorakenteiden teoriasta.

Seuraavassa kuvassa on esitetty eräs neljän alkion linketetty lista A, jonka alkiot säilyttävät kokonaislukuja 1-4.



Kuva 1: Linkitetty lista

Uuden alkion lisääminen listan alkuun on helppoa ja tehokasta. Luodaan uusi solmu, josta tulee uuden listan pää ja asetetaan sen `next`-linkkiä osoittamaan alkuperäisen listan päähän.



Kuva 2: Uuden alkion lisääminen listan alkuun

Kuvassa 2 esitetään tilanne, jossa listaan A “lisätään“ uusi solmu, jonka arvo on 5. Kun näin tehdään, lista A ei itse asiassa muutu miksikään. Se on edelleenkin sama neljän alkion lista ja mikä tahansa koodi, jolla on “linkki“ listaan A (eli sen päähän, solmuun jonka arvo on 4), voi edelleenkin käyttää sitä huoletta kuten ennenkin. Mitään olemassa olevaa listaa ei ole muokattu - “uuden solmun

lisääminen“ on tässä tapauksessa vain kielikuva. Todellisuudessa vanhan listan muokkaamisen sijaan olemme luoneet *uuden listan*. Sen nimi kuvassa 2 on A’. Tällä listalla on eri pää kuin solmulla A, mutta muuten niillä on samat alkiot. Vaikka listat A ja A’ ovat eri listoja, *ne jakavat suurimman osan solmuistaan*.

Tärkeä oletus, jonka teemme taustalla tässä yhteydessä, on se, että kumpikin lista A ja A’ pidetään *muuttumattomana*. Kumpikin luodaan kerran ja sen jälkeen niiden struktuuri sekä sisältö eivät saa koskaan muuttua. Tämä tarkoittaa myös sitä, että listan jokaisen solmun arvoa sekä `next`-linkkä **ei saa muuttaa**. Ovela sisäisen struktuurin turvallinen jakaminen eri tietorakenteiden kesken onnistuu ainoastaan tällä ehdolla. Jos meillä olisi lupa muuttaa vaikkapa listan A’ solmujen arvoja, se tarkoittaisi siitä, että listan A’ linkin omistaja voi myös vapaasti muokata listan A sisältöä (ja toisinpäin) - sotkemalla näin mahdollisesti asioita listan A käyttäjälle - joka ei välttämättä edes ole tietoinen siitä, että on olemassa myös lista A’, eikä kenties ole edes kiinnostunut tietämään tätä.

Juuri tämäntyyppistä “taikatemppua“ persistentit tietorakenteet käyttävät. Jos tietorakenteen sisällön suoraa muuttamista kielletään, ei ole tarpeellista luoda esimerkiksi täydellistä kopiota listan “hännästä“, koska se voidaan vain jakaa toisen listan kanssa sellaisenaan. Kun näin tehdään, säästetään sekä aikaa että muistia. Uuden alkion lisääminen yllä esitettyyn persistenttiin linkitettyyn listaan on aina nopea O(1)-operaatio riippumatta listan koosta (kunhan sovitaan, että uusi alkio lisätään listaan alkuun). Lisäksi, koska listat A ja A’ ovat muuttumattomia objekteja, eri säikeet voivat vapaasti käyttää niitä ilman mitään tarvetta synkronointiin tai pelkoa siitä, että kilpailutilanne syntyy.

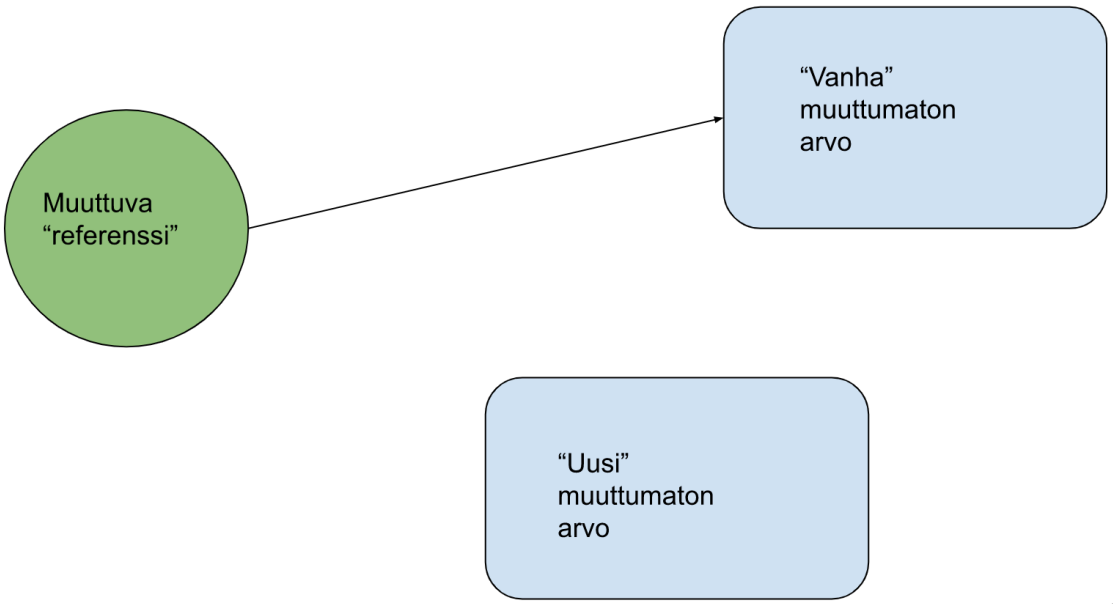
Kaikki Clojuren tarjoamat kieleen sisäänrakennetut tietorakenteet (lista, vektori, hajautustaulu, joukko) ovat *persistenttejä* sekä muuttumattomia [1, chapter 4, day 1]. Ne on implementoitu sillä tavalla, että niiden käyttö on käytännössä ainakin melkein yhtä tehokasta kuin vastaavien perinteisten, muuttuvien tietorakenteiden käyttö. Koska ne ovat muuttumattomia, ne ovat automaattisesti sekä **säie-**, että **iterointiturvallisia**. Jälkimmäinen ominaisuus tarkoittaa tässä sitä, että kun yksi säie iteroi persistentin kokoelman läpi ja toinen tekee siitä uuden version, mitään

ongelmia ei synny. Javan tapaisissa kielissä, kun kokoelma muutetaan kesken sen iteraation, seuraava iteroinnin kierros tyypillisesti kaatuu ja heittää `ConcurrentModificationException` tai vastaavan poikkeuksen [2, s.82]. Funktionaalisessa kielessä sen sijaan kokoelmat ovat muuttumattomia, joten korkeintaan pahinta mitä voi tapahtua tällaisessa tapauksessa, että iteroiva säie jatkaa rauhassa iterointia tietorakenteet edellisen (“vanhentuneen“) version läpi, mutta usein tämä on hyväksyttävissä. Itse asiassa Clojuren ja muiden funktionaalisten kielten filosofiaan kuuluu periaate, jonka mukaan jatkuvasti muuttuva sovelluksen maailma ei pitäisi edes yrittää pysäyttää sen tarkastelemiseksi. Sen sijaan jos sovelluksen tila halutaan tarkastella, siitä otetaan tämän hetken muuttumaton staattinen “snapshot“ ja tutkitaan se rauhassa blokkaamatta sovelluksen etenemistä [12, 20:00-22:00]. Tätä voidaan verrata tilanteeseen, jossa jalkapallopelissä katsoja ottaa valokuvan kentästä ja sen pelaajista. Silloin hän ei tietenkään pyydä laittamaan tätä varten peliä hetkellisesti “tauolle“. Tällaisen filosofian mukaan mahdollisesti vanhentunut data ei välttämättä ole ongelma, vaan saattaa hyvinkin kuulua prosessiin luonnollisella tavalla.

3.4.3 Referenssimuuttujien idea

Muuttumattomuus on hyödyllinen yhtäaikaisohjelmoinnissa, ja persistentit funktionaaliset tietorakenteet tarjoavat käytännössä toimivan kehyksen, jolla sen voi soveltaa tehokkaasti. On kuitenkin selvä, että reaalielämän ohjelmissa ei voi aina pärjätä ainoastaan muuttumattomilla arvoilla. Monet ongelmat ovat luonnostaan sellaisia, että niihin kuuluu yhteinen muuttuva tila. Miten ylläpidetään tällainen muuttuva tila, jos käytössä on vain muuttumattomat tietorakenteet? Miten tehdään koordinoidut muutokset, joita varten Javassa ja muissa samanlaisissa kielissä turvaudutaan tyypillisesti lukkoihin?

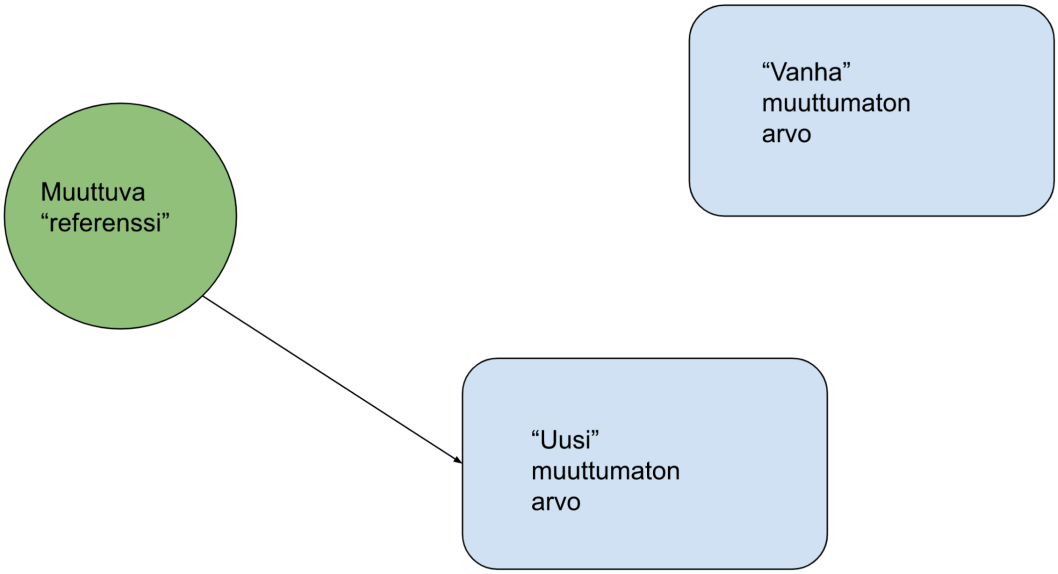
Vastaus ensimmäiseen kysymykseen on seuraava. Lisätään “indirection“-astetta - sallitaan **muuttuvat** “referenssimuuttujat“, joiden ainoana tehtävänä on referoida muuttumattomia objekteja.



Kuva 3: Referenssimuuttuja - alkuperäinen tila

Tällainen referenssimuuttuja on muuten “dummy proxy“, eli ainoa data, jonka se sisältää on linkki toiseen objektiin. Tämä objekti puolestaan on aidosti muuttumaton. Linkki tähän objektiin, jonka referenssimuuttuja sisältää on ainoa, mitä sallitaan muuttuvan.

Aina kun tämän linkin arvo halutaan muuttaa, muuttumattomasta arvosta linkin takana luodaan *uusi versio* ja sen jälkeen yksinkertaisesti vaihdetaan referenssimuuttujan sisältämää linkin arvoa ja osoitetaan se viittaamaan tähän uuteen versioon. Linkin arvon vaihto täytyy tehdä *atomisesti*, jotta vältämme tuttuja yhteisen muuttavan datan päivittämiseen liittyviä ongelmia. Koska linkin arvo on osoittimen kokoinen, se on tarpeeksi pieni atomiselle muuttujalle (kts. aliluku Miten atomiset muuttajat toimivat), joten lisäksi tämä atominen operaatio onnistuu yleensä ilman blokkavaa lukitusta.



Kuva 4: Referenssimuuttuja - päivittynyt tila

Tähän mennessä tämä ei ole vielä varsinaisesti mitään uutta. Onhan Javassakin oikeastaan kaikki oliot “langan päässä“ linkin takana, muuttumattomia tai ei. Javassakin on mahdollista implementoida täsmälleen samaa skenaariota käyttämällä atomisia muuttujia.

Oleellinen erona Javan ja Clojuren välillä on tässä yhteydessä se, että Javassa on **mahdollista** valita tämäntyyppinen tapa käsitellä yhteistä tilaa, mutta ei ole mitenkään **pakko**. Lisäksi, jos näin päätetään tehdä, synkronoidusta, atomaarisesta muuttujan arvon vaihdosta on pidettävä itse manuaalisesti huolta (jolloin voi helposti tehdä virheen). Clojure taas **pakottaa** käyttämään sen muuttuvia referenssimuuttujia hyvin tarkasti määritellyillä tavalla ja semantiikalla. Nämä on jo valmiiksi implementoitu sillä tavalla, että niiden käyttö on säieturvallista ja ohjelmoijan ei tarvitse itse kantaa huolta siitä. Jos Clojuressa haluaa käyttää muuttuvia arvoja, niiden on pakko toimia tällä tavalla. Kielestä löytyy sekä muuttuvia, että muuttumattomia palikoita, muuta kummallakin luokalla on omat, hyvin tarkat pelisäännöt sekä rajoitukset. [12, 36:00-36:30.]

Jos ollaan tarkkoja ja täysin rehellisiä, edellisessa kappaleessa esitetyt väiteet ei vät ole sataprosenttisesti tosia. Todellisuudessa Clojure on niin sanottu *hostattu* kieli, jonka implementaatio on rakennettu Javan virtuaalikoneen JVM päälle [13, chapter 4]. Sellaisenaan se tarjoaa sitten myös suoraan mahdollisuuden kutsua koodissaan Javan koodia ja toisinpäin. Näin ollen on täysin mahdollista hylätä Clojuren tarjoamat persistentit muuttumat kokoelmat sekä referenssimuuttujat koodata sen sijaan käytännössä Javaa, mutta Clojuren syntaksilla. Tämä sisältää mahdollisuuden käyttää suoraan Javan muuttuvia oliota, lukkoja ja niin poispäin. Lisäksi Clojuressa referenssimuuttujan linkin takana ei tarvitse olla muuttumaton olio - se voi olla esimerkiksi toinen referenssimuuttuja, tai mielivaltainen Javan objekti. Kieli periaatteessa sallii nämä tapaukset, mutta se on vahvasti ei-suositeltu.

Jos nämä poikkeukset unohdetaan hetkeksi, ainoa tapa käyttää yhtäaikaisuutta sekä aidosti muuttuvia arvoja Clojuressa on käyttää kielen siihen tarjoamia valmiita työkaluja, joiden mahdollisuuksia on rajattu tarkoituksella ja joiden implementaatiossa on otettu huomioon tärkeät yksityiskohdat, jotka takaavat säieturvallisuutta suoraan “‘out of the box“‘. Lisäksi nämä on rakennettu sillä tavalla, että seuraavat ominaisuudet pätevät.

- Ohjelmoijan ei tarvitse itse luoda ja ylläpitää säikeitä - Clojure luo ja ylläpitää ne itse “verhon takana“.
- Referenssimuttujan arvon vaihto voidaan tehdä atomisesti ilman blokkavaa lukitusta. Näin ollen myös esimerkiksi lukkiuma ei ole mahdollinen.
- Jokaisella kielen tarjoamalla referenssimuuttujan tyypillä on tarkasti määritelty semantiikka siitä, miten sitä käytetään sekä mitä ominaisuuksia sillä on. Tästä semantiikasta ei ole mahdollista poiketa.
- Clojuren referenssimuuttujat pitävät itse ohjelmoijan puolesta huolta korrektista datan näkyvyydestä eri säikeiden kesken.
- Työkalut ovat kuitenkin monipuolisia ja tarjoavat tarpeeksi mahdollisuuksia käytännön tarpeisiin. Jos mahdollisuutta käyttää suoraan Javan oliota ei oteta huomioon, nämä työkalut ovat kielen ainoa tapa saada aikaan ohjelmassa muuttavaa tilaa - kaikki muu on muuttumatonta.
- Referenssimuuttujat ovat hyvin yksinkertaisia, “dummy“-olioita.

> Käytännössä ainoata, mitä sellainen muuttuja osaa, on osoittaa muuttumattomaan resurssiin sekä vaihtaa tämän sisältämä referenssi toiseen muuttumattomaan resurssiin *säieturvallisella tavalla*.

Toisin sanoen Clojuressa kieli pitää itse huolta asioista, jotka Javan tyyppisessä kielessä jäävät ohjelmoijan vastuulle - ja kyse on juuri sellaisista asioista, jotka ovat tunnetusti vaikeita ja virhealttiita.

Clojure tarjoaa tasan neljä eri referenssimuuttujien typpiä: `atom`, `ref`, `agent` ja `var` [14, chapter 10.1]. Atomehin olemme tutustuneet jo aliluvussa Atomiset muuttujat. Seuraavaksi käymme läpi `ref` sekä `agent` muuttujiin. `var`-muuttujia emme käsittele tässä työssä, sillä ne eivät ole niin hyödyllisiä yhtäaikaisuuden näkökulmasta [14, chapter 10.1].

Jokainen Clojuren referenssimuuttuja luodaan tietyllä funktiolla, jolle annetaan parametrina sen alkuarvo. Muuttujan sisältämä tämänhetkinen arvo saadaan funktiolla `deref` tai macrolla `@`. Erityyppisillä referenssimuuttujilla on kullakin oma semantiikkansa sekä käyttötarkoitus. Eri referenssimuuttujatyypit eroavat siinä, onko niiden sisältämän linkin päivitysoperaatio *synkroninen* vai *asynkroninen*, sekä sillä tarjoaako se eri referenssimuuttujien *koordinoitua* eli samanaikaista päivitystä. [14, chapter 10.1.]

3.4.4 Ref-muuttujat ja transaktiot

Atomimuuttujat (joihin olemme tutustuneet jo aliluvussa Atomiset muuttujat) ovat kenties yksinkertaisimpia referenssimuuttujia Clojuressa. Ne tarjoavat minimaalisia mahdollisuuksia, mitä referenssimuuttujat ylipäätän voivat tarjota. Atomin päivitysoperaatio `swap!` on synkroninen, eli päivitys sovelletaan heti, ennen kuin siirretään koodissa eteenpäin. Lisäksi jokainen atomi on täysin riippumattoman muista atomeista, joten atomien API ei tarjoa kahden tai useamman atomin *koordinoitua*, samanaikaista päivitystä. [14, chapter 10.1.]

Esimerkiksi seuraava tapa siirtää rahat yhdestä tililtä toiselle ei ole säieturvallinen:

```
(def account-1 (atom 200))
(def account-2 (atom 100))
```

```
(swap! account-1 - 100)
(swap! account-2 + 100)
```

Koodiesimerkki 14: Tilisiirto atomeilla - tämä ei toimi

Ongelma on siinä, että kahden `swap!`-funktion kutsun välissä muut säikeet voivat operoida kummallakin tilillä, silloin kun rahan siirto on "'kesken"'. Tarkoituksena olisi kuitenkin, että mikään säie ei pystyisi näkemään nämä tilit "epäkonstistentissä" tilassa - esimerkiksi kun yhdestä tilistä on jo vähennetty siirron summa, mutta se ei ole vielä ilmestynyt toiselle tilille. Rahansiirron pitäisi olla **atominen operaatio kokonaisuudessaan**. Atomit eivät kuitenkaan takaa tätä, ainoastaan sen että, jokainen `swap!`-operaatio on erikseen atominen.

Javassa tällaisessa tilanteessa tyypillisesti turvaudutaan lukitukseen, jolla suojellaan paitsi koko siirto, että myös kummankin tilin tilan lukeminen. Tällöin todennäköisesti joudutaan käyttämään kaksi lukkoa - yksi lukko jokaista tiliä kohti. Lisäksi niille täytyy sopia joku universaali lukitusjärjesys, koska missä on kaksi lukkoa, joista toinen lukitaan toiseen jälkeen, siinä on todellinen lukkiuman vaara.

Clojuressa sen sijaan tällaisia eri objektien koordinointeja muutoksia varten on olemassa `ref`-muuttujat ja niin sanottu **software transactional memory** (STM jatkossa).

`Ref`-referenssimuuttuja luodaan `ref`-funktiolla ja sen arvo päivitetään funktioilla `alter` tai `ref-set`. [1, chapter 3, day 2.]

Määritellään edellisen esimerkin tilit `ref` muuttujiksi.

```
(def account-1-ref (ref 200))
(def account-2-ref (ref 100))
```

Koodiesimerkki 15: Tilit referenssimuuttujina

Kokeillaan nyt suorittaa rahansiirron ensimmäinen vaihe:

```
(alter account-1-ref - 100) ;; java.lang.IllegalStateException: No
    transaction running
```

Koodiesimerkki 16: Yritys päivittää tili suoraan

Tämä ei toiminut - Clojure heitti `IllegalStateException` poikkeuksen. Syy tähän on yksinkertainen - `ref`-muuttujan arvoa saa päivittää ainoastaan niin sanotun *STM-transaktion sisällä*. [1, chapter 3, day 2.]

Clojuren STM-transaktiot toimivat suurin piirtein samalla tavalla kuin tietokantojen maailmasta tutut transaktiot. STM-transaktio on

- **Atominen**. Kaikki `ref`-muuttujien transaktion sisällä tapahtuneet muutokset tulevat voimaan atomisesti “samaan aikaan“ transaktion ulkopuolella katsottuna. Transaktion tekemä muuttujien päivitystyö joko tapahtuu kokonaisuudessaan tai ei tapahdu lainkaan.
- **Konsistentti**. `ref`-muuttujalle voidaan asettaa *validaatio*, joka sallii sille vain tietynlaiset arvot. Jos transaktion sisällä yhdenkin `ref`-muuttujan validointi epäonnistuu, koko transaktio epäonnistuu ja mikään sen tekemää muutosta `ref`-muuttujiin ei sovelleta.
- **Eristynyt**. Minkään transaktion sisällä ei voida nähdä muissa samaan aikaan käynnissä olevissa transaktioissa tapahtuvia muutoksia (silloin kun ne ovat vielä kesken).

Nämä ovat tietokantojen teoriasta tutun **ACID** lyhynteen **A**tomicity, **C**onsistency ja **I**solation. Viimeistä, eli **Pysyvyyttä** (**D**urability) STM ei tarjoa, sillä `ref`-muuttujat ja niiden arvot ovat olemassa vain ohjelman ajoaikana. [1, chapter 3, day 2.]

Transaktio suoritetaan `dosync`-funktiolla [1, chapter 3, day 2]. Tässä on toimiva esimerkki miten yllä tarkasteltu rahan siirto voidaan toteuttaa turvallisesti STM-transaktiossa:

```
(dosync
    (alter account-1-ref - 100)
    (alter account-2-ref + 100)
)
```

Koodiesimerkki 17: Pävitetään tilit STM-transaktiossa

Tämä toimii täsmälleen kuten toivotaan. Kummankin tilin rahan määrä muuttuu samanaikaisesti kaikkien muiden säikeiden näkökulmasta. Pannaan erityisesti merkille, kuinka helppoa, kätevää ja samalla turvallista tällaisen koodin kirjoittaminen on. Ohjelmoijan vastuulle jää käytännössä vain kääriä transaktioon sisään päivitykset, jotka halutaan astuvan voimaan yhdessä, “samanaikaisesti“. Lukkoja ja niiden käyttöön liittyviä ongelmia ei edes tarvitse miettiä. Kieli pitää itse

huolta siitä, että transaktiomuistin ja `ref`-muuttujien lupaukset lunastetaan. Itse asiassa opimme seuraavaksi, että STM-transaktioissa ei käytetä lukitusta.

Miten Clojuren STM-transaktiot toimivat?

Ylätasolla transaktiossa tapahtuu seuraavaa. Aina kun transaktion sisällä kutsutaan `alter` tai `ref-set` jollekin `ref`-muuttujalle, transaktio ikään kuin luo “oman paikallisen kopionsa“ tämän muuttujan nykyisestä arvosta ja operoi sitä vastaan omassa eristetyssä “hiekkalaatikossaan“, johon ainoastaan sillä on pääsy. Se myös muistaa tämän muuttujan alkuperäisen arvon, josta se on ottanut “kopionsa“. Jos missään vaiheessa transaktion edetessä se “huomaa“, että tämän muuttujan arvo onkin jo muuttunut (koska joku muu transaktio on ehtinyt päivittää se onnistuneesti), tämä transaktio *keskeytetään ja aloitetaan uudestaan alusta*. Jos taas transaktion lopussa tällaisia konfliktiovia muutoksia ei ole havaittu, kaikki transaktiossa tekemät `ref`-muuttujien muutokset sovelletaan atomisesti (muiden transaktioiden näkökulmasta) muuttujien todellisiin arvoihin. [14, chapter 10.1.2.]

Näin ollen, samalla tavalla kuten atomisten muuttujien kohdalla kohdalla, `ref`-muuttujan päivitysyrityksiä joudutaan suorittamaan transaktiossa mahdollisesti monta kertaa, ennen kun (ja jos) onnistutaan. Erityisesti myös transaktion sisällä ei pitäisi ajaa koodia, jolla on mahdollisesti sivuvaikutuksia (kohta opimme kuitenkin, että *agentti*-referenssimuuttujat itse asiassa mahdollistavat tämän tarvittaessa). Strategia, jonka transaktiot käyttävät muistuttaa tavan, jolla atomiset muuttujat päivittävät arvonsa (kts. aliluku Miten atomiset muuttajat toimivat)). Siinä, missä atomiset muuttujat tarjoavat tavan suorittaa alkeellisia atomisia päivityksiä, transaktiot voidaan ajatella olevan hengeltään vastaava tapa suorittaa “atomisesti“ mielivaltaisen pitkiä operaatioita, jotka koostuvat monesta toimipiteestä. Toisin sanoen ne tarjovat samaa funktionaalisuutta kuten lukot, mutta atomien muuttujien tapaisella strategialla, ilman blokkausta. Lisäksi STM-transaktioita käytettäessä `ref`-muuttujien arvoja pelkästään lukevat säikeet eivät joudu odottamaan. Näiden ominaisuuksien ansiosta transaktioilla on käytännössä myös hyvä suorituskyky siitäkin huolimatta, että niiden koodia joudutaan ajamaan mahdollisesti monta kertaa, kunnes

onnistutaan.

Commute-funktio

Yleensä datan eheys monisäieympäristössä vaatii yllä esitettyä ankaraa strategia, jonka mukaan transaktio ajetaan aina uudestaan alusta, jos huomataan, että yksikin transaktiossa osallistuva `ref` on jo ehtinyt muuttua tämän transaktion ulkopuolella.

Joskus on kuitenkin tilanteita, joissa transaktion jatkaminen on täysin turvallinen, vaikka jotain sen `ref`-muuttujaa päivitetään samanaikaisesti toisessa transaktiossa. Esimerkiksi kuvitellaan tilanne, jossa `ref`-muuttujan arvo on yksinkertainen kokonaislukulaskuri, ja jokaisen transaktion sisällä sen arvoa kasvatetaan yhdellä. On selvä, että tämä operaatio tuottaisi täsmälleen saman tuloksen, vaikka emme tarkistaisi transaktion sisällä, onko tämän laskurin arvo mahdollisesti muuttunut jo kesken sen suoritusta. Syy tähän on siinä, että kaikki kasvattamisoperaatiot “‘kommutoivat“‘ keskenään, eli ei ole merkitystä, missä järjestyksessä ne sovelletaan.

Tällaisia tilanteita varten Clojure tarjoaa `commute`-funktion. Se toimii täsmälleen samalla tavalla kuin `alter`, mutta siitä käytettäessä transaktio ei tarkista, onko vastaavan `ref`-muuttujan arvo muuttunut. Tällöin järjestys, jossa muuttujaan sovelletaan päivitykset ei ole enää deterministinen, joten `commute` soveltuu vain tilanteissa, joissa eri transaktioiden tekemät muutokset kommutoivat keskenään, eli niiden sovellusjärjestyksellä ei ole merkitystä lopputuloksen kannalta (funktion nimi tuleekin matemaattisesta termistä “kommutatiivisuus“). [14, chapter 10.2.2.]

Funktio `commute` on yleisesti ottaen tehokkaampi kuin `alter`, koska se ei koskaan triggeröi transaktion aloitusta alusta, mutta siihen pitää suhtautua varauksella ja käyttää ainoastaan, kun se varmasti sopii tilanteeseen eli silloin, kun päivitykset varmasti tiedetään kommutoivan keskenään.

Lukeminen transaktiossa

Transaktioiden käyttötarkoitus on tarjota turvallisia, keskenään koordinoituja **päivityksiä** referenssimuuttujin. Jos päivitysyrityksessä tapahtuu yksikin konflikti, koko operaatio hylätään ja kokeillaan uudestaan. Sen sijaan jos jonkun `ref`-muuttujan arvo pelkästään **luetaan** transaktion sisällä, STM-mekanismi ei seuraa sen muutoksia. Tätä ilmiötä sanotaan "*kirjoittamisen vinouttumaksi*" (engl "*write skew*") - `ref`-muuttujat ja STM-transaaktiot kohtelevat lukemista ja päivittämistä eri tavalla [14, chapter 10.2.]. Lukkojen maailmassa tämä vastaisi tilannetta, jossa lukolla suojellaan vain datan päivitykset, mutta ei lukemiset. Tämä voi olla ok, mutta voi myöskin olla ongelmana, sillä jossakin tilanteessa voisimme haluta "lukita" `ref`-muuttujan arvon transaktion aikana eli vaatia, että sen arvo pysyy samana kunnes transaaktio loppuu onnistuneesti.

Ratkaisuksi tähän Clojure tarjoaa `ensure`-funktion. Kutsu `(ensure ref-val)` transaktion sisällä palauttaa `ref`-muuttujan `ref-val` nykyisen arvon, mutta samalla "suojaa"sen muutoksilta transaktion ulkopuolelta. Suojaaminen ei tarkoita tässä sitä, että se ei oikeasti saa muuttua tästä alkaen, kunnes transaktio loppuu, vaan yksinkertaisesti sitä, että jos se muuttuukin, transaktio keskeytetään ja aloitetaan uudestaan. Toisin sanoen funktion `ensure` semantiikka on sama kuin funktiolla `alter`, mutta lukemisen eikä kirjoittamisen suhteen. [14, chapter 10.2.]

3.4.5 Agentit

Atomit ja `ref`-muuttujat päivittävät arvonsa *synkronisesti* eli kun päivitysfunktio on palautunut / transaktio on onnistuneesti ajettu loppuun, päivitys on jo sovellettu.

Joskus on tilanteita, joissa on järkevämpää tai tehokkaampaa sallia *asynkroniset* päivitykset. Esimerkiksi kun ohjelman suorituksen aikana halutaan pitää lokihistoria, tai datasta halutaan aina välillä ottaa varmuuskopio, se voidaan tehdä erillisestä säikeessä omana prosessina, joka saa olla jopa hieman "myöhässä". Tällaisia käyttötapauksia varten Clojure tarjoaa **agentteja**.

Agentti luodaan funktiolla `agent` ja päivitetään funktiolla `send` [14, chapter 10.3]:

```
(def my-agent (agent 0))
```

```
(send my-agent inc) ;; kun tämä päivitys on (joskus tulevaisuudessa)
    käsitelty, agentin arvoksi tulee 1
```

Koodiesimerkki 18: Clojuren agentti

Päinvastoin kuin esimerkiksi atomin päivitysfunktio `swap!`, funktion `send` kutsu ei siis välttämättä päivitä agentin sisältämää arvoa heti. Sen sijaan tämän funktion kutsu *lähettää agentille ikään kuin päivityspyynnön*, ja palaa sen jälkeen heti, jolloin agentilla saattaa hyvinkin olla vielä vanha arvo jokin aikaa `send`-kutsun jälkeenkin. Nämä päivityspyynnöt laitetaan agentin sisäisesti ylläpitävään jonoon, ja niitä käsitellään yksi kerralla asynkronisesti *erillisessä säikeessä*, samassa järjestyksessä, jossa ne saapuivat. Nämä säikeet kuuluvat Clojuren tätä varten ylläpitämään säievarastoon, jolla on optimaalisella tavalla valittu kiinteä koko (jälleen kerran kieli pitää tässä itse huolta implementaation yksityiskohdista) ja jonka kaikki agentit jakavat keskenään. [14, chapter 10.3.]

Tästä seuraa, että funktio `send` soveltuu huonosti tilanteeseen, jossa päivityspyyntö sisältää blokkavan operaation, sillä tällöin voidaan vahingossa estää muidenkin agenttien etenemistä. Tästä syystä agenteille on olemassa toinen päivitysfunktio `send-off`. Se toimii kuten `send`, paitsi että sen lähettämää päivityspyyntöä käsitellään omassa erillisessä, agentille varatussa säikeessä, jota muut agentit eivät käytä. [14, chapter 10.3.]

Tarvittaessa funktion `await` avulla voidaan odottaa, kunnes agentiin kaikki tämän funktion kutsuvasta säikeestä lähetetyt päivityspyynnöt on käsitelty. [14, chapter 10.5.]

Agentit ja transaktiot

Tässä vaiheessa voidaan paljastaa, miksi Clojuressa on tapana nimetä jotkut funktiot, kuten `swap!`, niin, että funktion nimen lopussa on huutomerkki. Tällä tavalla Clojuressa merkitään funktiot, jotka eivät ole **transaktioturvallisia**. Funktiot, joilla on joku sivuvaikutus, eivät yleensä ole transaktioturvallisia, esimerkiksi jos funktio `swap!` kutsutaan transaktion sisällä, se päivittää atomin arvon joka kerta, kun transaktio keskeytetään ja yritetään ajaa alusta, mikä

yleensä ei ole sitä, mitä halutaan. [1, chapter 4, day 2.]

Agentin päivitysfunktion `send` nimi sen sijaan ei sisällä huutomerkkiä. Tämä viittaa siihen, että `send` on transaktioturvallinen, vaikka tavallaan sisältää sivuvaikutuksen. Kun `send` kutsutaan STM-transaktion sisällä, päivityspyyntöä ei lähetetä heti, vain korkeintaan kerran, kun ja jos transaktio lopulta onnistuu. Tämä ominaisuus tarjoaa myös tavan suorittaa muitakin “sivuvaikutuksia“ transaktion osana. Ne pitää vain liittää johonkin agenttiin. [1, chapter 4, day 2.]

Esimerkiksi lisätään rahansiirtoesimerkkiin kuitin lähettäminen rahaa siirtävälle tilin omistajalle.

Oletetaan yksinkertaisuuden vuoksi, että meillä on jo käytössä Javan luokka `ReceiptSender`, jonka instanssi osaa lähettää kuitin metodinsa `send(from, to, amount)` avulla.

Ensin määritellään agentti, jonka tehtävä tulee olemaan kuitin lähettäminen minkä tahansa siirron yhteydessä, sekä funktio, joka lähettää tälle pyynnön lähettää kuitti:

```
(def transaction-receipt-sender (agent (new ReceiptSender)))

(defn send-receipt [from to amount]
    (send-off transaction-receipt-sender (fn[sender] (. sender send
        from to amount)))
)
```

Koodiesimerkki 19: Kuitin lähetysfunktio

Lisätään rahansiirtotransaaktion loppuun kuitin lähetyspyyntö:

```
(dosync
    (alter account-1-ref - 100)
    (alter account-2-ref + 100)
    (send-receipt account-1-ref account-2-ref 100)
)
```

Koodiesimerkki 20: Lähetetään kuitti transaktion yhteydessä

Vaikka tämä transaktio yritetäänkin suorittaa monta kertaa ennen kuin se onnistuu, Clojure takaa, että kuitti lähetetään korkeintaan kerran, jos ja kun

rahansiirto on vihdoinkin onnistuneesti tapahtunut. Tämä ei johdu siitä, että esimerkissämme kuitti lähetetään viimeisenä transaktion askeltena. Samoin olisi käynyt jos tämä koodinrivi sijoitetaan vaikkapa transaktion alkuun. Kuitin lähetyksestä vastaava agentti laittaa kaikki nämä lähetyspyynnöt jonoon ja käsittelee ne yksi kerrallaan omassa säikeessä, taustaprosessina.

3.4.6 Clojure vs. Java

Clojuren paradigma suosii “staattista“ näkemystä ohjelman tilasta sekä erottaa selkeästi muuttujan “identiteetin“ sen tämänhetkisestä arvosta. Tämä tarkoittaa sitä, että Clojuressa muuttuja, jolla on nimi, ja tämän muuttujan arvo mielletään selkeästi eri asioina. Muuttuja, “identiteetti“, ajatellaan olevan ajassa elävä diskreetti *sarja* eri arvoja, jonka se saa ohjelman ajoaikana, prosessi eikä arvo. Imperatiivissa kielissä tämä ero on yleensä hämärtynyt, ja muuttuja on sama asia kuin tietyssä muistialueella tällä hetkellä sijaitseva bittijono. Kun tämä arvo ylikirjoitetaan, vanha arvo häviää ikuisesti. Clojuressa sen sijaan arvo on yleensä muuttumaton ja sen “nimi“, muuttuja, joka siihen viittaa, on vain hetkellinen referenssi siihen tiettynä hetkenä. Todellinen arvo on “staattinen“, se ei koskaan muutu miksikään. [1, chapter 4, day 1.]

Staattisen maailmankuvan filosofia korostuu Clojuressa myös STM-transaktioissa. Transaktion alussa sen `ref`-muuttujista otetaan hetkellinen “snapshot“ ja operoidaan sitä vastaan.

Staattisen, muuttumattoman kuvan kanssa on paljon helpompaa työskennellä kuin jatkuvasti muuttuvan kanssa, erityisesti kun ohjelmassa on käynnissä monta erillistä prosessia eri säikeissä.

Tämän luvun lopuksi tehdään vielä lista konkreettisista Javan ja Clojuren eroista, mitä tulee tässä työssä käsiteltävään aiheeseen.

- Clojure suosii muuttumattomia objekteja, joiden käyttö on säieturvallista määritelmän mukaan.
- Muuttuvien objektien päivitys onnistuu puhtaassa Clojuressa vain

käyttämällä tarkasti määriteltyjä työkaluja, joilla on valmiiksi säieturvallisia piirteitä, jotka ei tarvitse koodata itse.

- Clojure suosii lukkovapaita, ei-blokkavia ratkaisuja. Tämä vähentää huomattavasti lukkiuman vaaraa, sekä potentiaalisesti parantaa sovelluksen suorituskykyä.
- Clojure pitää itse huolta säievarastosta, ohjelmoijan ei yleensä tarvitse ylläpitää säikeitä itse.
- Jos kuitenkin kaipaa tarkempaa matalatasoista kontrollia, Java saattaa olla parempi työkalu. Onneksi, koska Clojuressa Java-yhteentoimivuus sisältyy kieleen, tämäkin voidaan tehdä Clojuresta käsin, ei tarvitse vaihtaa kieltä.
- Clojure keskittyy saamaan **päivitykset** monisäikeisessä ohjelmassa toimimaan oikein, muuttuvaan datan turvalliseen lukemiseen on kiinnitetty kielessä vähemmän huomiota. Tämä voi johtaa vanhentuneen datan ilmiöön, mikä voi olla virhe jossakin tilanteessa.
- Javasta löytyy atomiset muuttujat, joilla on periaatteessa sama funktionaalisuus kuten Clojuren atomeilla, joten niiden kohdalla Clojure ei varsinaisesti tarjoaa mitään uutta. Sen sijaan STM-transaktioille tai agenteille Javassa ei löydy suoraa tukea.

3.4.7 Clojure vs. Rust

Clojuren referenssimuuttujat ja Rustin lukot muistuttavat toisiaan siinä mielessä, että kummassakin eri säikkeiden kesken jaettu resurssi inkapsuloidaan säieturvallisen objektin sisään. Tällöin resurssin päivittäminen onnistuu vain tämän objektin julkisen APIn kautta. Löytyy kuitenkin myös tärkeä ero. Clojuressa on helpompaa “unohtaa“ piilottaa resurssi referenssimuuttujaan ja silti jakaa se eri säikkeiden kesken, varsinkin jos käyttää kielen Java-interopia. Rust on tässä mielessä paljon ankarampi, sillä siinä resurssi ei voida jakaa eri säikkeiden kesken noin vaan, vaan se on pakko kääriä johonkin lukko-tapaiseen objektiin, joka on säieturvallinen.

Kuten Java, Rust ei myöskään tarjoaa suoraa tukea STM-transaktioille tai agenteille.

4 Yhteenveto

Olemme tutustuneet jaettujen resurssien käyttämiseen liittyvistä haasteista yhtäaikaisohjelmoinnissa kolmen konkreettisen ohjelmointikielen näkökulmasta. Tässä vaiheessa voidaan todeta, että yksi selkeä ero Javan ja kahden muun kielen välillä liittyy siihen kuinka paljon vapautta ja omaa vastuuta ohjelmoijalle annetaan. Yhtäaikaisuuden mielessä Java edustaa “perinteistä“ lähestymistapaa, jossa kieli antaa vain suhteellisen matalatasoisia primitiivejä ja loput on täysin ohjelmoijan vastuulla. Tämä antaa ohjelmoijalle paljon vapautta sekä mahdollisuuden tarkkaan kontrolliin, mutta ei suojaa virheiltä käytännössä lainkaan. Resursseja voi jakaa säikeiden väliin täysin vapaasti, vaikka se ei olisi säieturvallista ja altistaa selvästi kilpailutilanteelle. Javan kielen kääntäjä ei tarkista tällaisia asioita lainkaan, jolloin on täysin mahdollista luoda oikeasti ajettavaa ohjelmaa täynnä pahoja bugeja.

Vuosien varrella on opittu käytännön kokemuksen myötä, kuinka vaikea yhtäaikaisohjelmointi on ja mitä kaikkia salakavalia virheitä voi helposti syntyä yrityksissä ohjelmoida se “oikein“. Samalla kun on kehitetty uusia työkaluja, joilla voidaan välttää nämä virheet, on myös oivallettu, että on mahdollista rajoittaa ohjelmoijan mahdollisuuksia suoraan ohjelmointikielessä - tarkoituksena vähentää virheiden syntyä ja helpottaa ohjelmoijan työtä. Siksi monet Javaa nuoremmat ohjelmointikielet, mukaan lukien juuri Clojure ja Rust, pyrkivät tietoisesti tehdä kielen käytöstä turvallisempaa - tietoisesti rajoittamalla mitä kieli sallii. Tämä pätee monissa aspektissa, myös mitä tulee yhtäaikaisuuteen, jonka korrekti koodaaminen on tunnetusti erityisesti vaikeata ja virhealtista.

Rust mainostaa itseään turvallisena versiona C++ kielestä. Sen hyvin ankarat omistusäännöt, jotka rajoittavat ohjelmoijan mahdollisuuksia huomattavasti, on keksitty takamaan, että muistin käsittely on turvallista, korrektia ja tehokasta, vaikka automaattista roskienkerääjää ei käytetä. Vaikka näiden ensisijainen tarkoitus ei liity suoraan yhtäaikaisuuteen, osoittautuu, että ne riittävät myös tekemän siitä turvallisen automaattisesti ainakin tietyssä mielessä. Esimerkiksi

koodi, jossa eri säikeet saisivat hallitsemattomasti päivittää samoja resurssia ei yksinkertaisesti ole korrektia Rust-koodia, ja kielen kääntäjä ei suostu edes kääntämään sitä. Rust rajoittaa mitä olioita voi ylipäätän jakaa eri säikeiden väliin ja tämä on jälleen esimerkki tietoisesta rajoituksesta, jonka tarkoitus on auttaa välttämään virheitä. Toinen tärkeä oivallus, jonka perinteiset kielet jäivät aikoinaan tekemättä, mutta Rust on soveltanut, on lukolla suojatun datan ja itse lukon yhdistäminen samaksi jaottomaksi objektiksi, jota ei pysty vahingossakaan käyttämään “väärin“.

Clojure on myös esimerkki suhteellisen modernista kielestä, joka on yrittänyt omalla tavallaan välttää sen esivanhempien virheitä ja tuottaa lisää turvallisuutta rajoittamalla kielenkäyttäjän mahdollisuuksia tahallaan. Funktionaalisena kielenä yksi sen valttikortteista yhtäaikaisuudessa on datan muuttumattomuus oletusarvoisesti. Kuten omistussäännöt Rustissa, myös muuttumattomuus oli alun perin nostettu funktionaalisissa kielissä tärkeäksi periaatteeksi syistä, jotka eivät suoraan liittyneet yhtäaikaisuuteen. Myöhemmin kuitenkin selvisi, että muuttumattomuus yksinkertaistaa myös yhtäaikaisohjelmointia ja auttaa välttämään siihen liittyviä virheitä. Syy tähän on yksinkertainen - käytännössä kaikki yhteisten resurssien samanaikaiseen käyttöön haasteet liittyvät niiden päivityksiin. Kielessä kuten Java kaikki objektit ovat oletusarvoisesti muuttuvia ja muuttumattomuus on jotain, mitä ohjelmoijan pitää tarkasti järjestää itse (ja kielestä saa siihen aika vähän tukea). Clojuressa asianlaita on päinvastainen - kaikki arvot ovat oletusarvoisesti muuttumattomia, ja jos haluaa muuttuvia arvoja, pitää valita muutamasta tarkasti määritellystä vaihtoehdoista, joita kieli tarjoaa tähän tarkoitukseen. Tämä on esimerkki rajoituksesta - rajoitetaan tarkoituksella käyttäjän mahdollisuuksia vapaasti päivittää arvot. Tietotyypit, jotka on muuttuvia, eivät myöskään ole mielivaltaisia. Niitä on vain neljä ja kaikki ne ovat niin sanottuja “referenssimuuttujia“, joiden sisäinen struktuuri on pohjimmiltaan samanlainen - ne edustavat muuttuvaa viitettä muuttumattomaan arvoon, ja ainoa minkä voi muuttaa on itse tämä viite. Jokaisella referenssimuuttujatyypillä on tarkka semantiikka ja korkeatasoinen API, joka tarjoaa ainoan tavan päivittää muuttujan arvo. Tämä päivitysoperaatio on automaattisesti säieturvallinen. Näin ollen, paitsi, että itse muuttamista rajoitetaan, myös silloin kun se sallitaan, tämä

edelleenkin sisältää paljon rajoituksia ja ennalta määriteltyä funktionaalisuutta, jota ei voi muuttaa.

Yleisesti ottaen erilaiset ohjelmointikielet, paradigmat ja muut ohjelmistotyökalut eroavat toisistaan muun muassa siinä, kuinka paljon ne rajoittavat käyttäjän mahdollisuuksia. Mitä enemmän rajoituksia, sitä turvallisempaa työkalun käyttäminen on ja sitä pienempi virheiden mahdollisuus on. Toisaalta mitä vähemmän rajoituksia, siitä enemmän valinnanvapautta ja kontrollia. Kyseessä on tietynlainen “trade-off“ ja kummassakin ääripäässä on omat hyödyt ja haitat. Toisaalta kokemus on osoittanut, että yhtäaikaisuuden korrekti käsittely on todella vaikeata ja todella virhealtista, erityisesti reaalielämän oikean hyvin kompleksisen teollisen koodin kohdalla. Tällaisessa tilanteessa on kenties viisaampaa valita selvästi se rajoittava, mutta turvallisempi työkalu.

Lähteet

1 Butcher, Paul. 2014. Seven Concurrency Models in Seven Weeks. Verkkoaineisto. <https://learning.oreilly.com/library/view/seven-concurrency-models/9781941222737/>. Luettu 16. 02. 2023.

2 Goetz, Brian. 2006. Java Concurrency in Practice. Addison-Wesley Professional.

3 Charpentier, Michel. 2022. Functional and Concurrent Programming: Core Concepts and Features. Verkkoaineisto. <https://learning.oreilly.com/library/view/functional-and-concurrent/9780137466696/>. Luettu 16. 02. 2023.

4 Kukunas, Jim. 2015. Power and Performance. Verkkoaineisto. <https://learning.oreilly.com/library/view/power-and-performance/9780128007266/>. Luettu 16. 02. 2023.

5 Bos, Mara. 2023. Rust Atomics and Locks. Verkkoaineisto. <https://learning.oreilly.com/library/view/rust-atomics-and/9781098119430/>. Luettu 16. 02. 2023.

6 Sharma, Rahul; Kaihlavirta, Vesa & Matzinger, Claus. 2019. The Complete Rust Programming Reference Guide. Verkkoaineisto. <https://learning.oreilly.com/library/view/the-complete-rust/9781838828103/>. Luettu 16. 02. 2023.

7 Blandy, Jim; Orendorff, Jason & F.S.Tindall, Leonora. 2021. Programming Rust, 2nd Edition. Verkkoaineisto. <https://learning.oreilly.com/library/view/programming-rust-2nd/9781492052586/>. Luettu 16. 02. 2023.

8 Klabnik, Steve & Nichols, Carol. 2022. The Rust Programming Language. Verkkoaineisto. <https://doc.rust-lang.org/stable/book/>. Luettu 16. 02. 2023.

9 Oracle Java documentation: Package java.util.concurrent.atomic 2023. Verkkoaineisto. <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html>. Luettu 16. 02. 2023.

10 Clojure guides: Concurrency and Parallelism in Clojure 2012. Verkkoaineisto. <https://clojure-doc.org/articles/language/concurrency_and_parallelism/>. Luettu 16. 02. 2023.

11 Kumar, Shantanu. 2015. Clojure High Performance Programming - Second Edition. Verkkoaineisto.

<https://learning.oreilly.com/library/view/clojure-high-performance/9781785283642/>. Luettu 16. 02. 2023.

12 Hickey, Rich. 2019. Persistent Data Structures and Managed References. Video. <https://www.youtube.com/watch?v=toD45DtVCFM>. Luettu 16. 02. 2023.

13 Meier, Carin. 2015. Living Clojure. Verkkoaineisto. <https://learning.oreilly.com/library/view/living-clojure/9781491909270/>. Luettu 16. 02. 2023.

14 Fogus, Michael & Houser, Chris. 2014. The Joy of Clojure, Second Edition. Verkkoaineisto. <https://learning.oreilly.com/library/view/the-joy-of/9781617291418/>. Luettu 16. 02. 2023.