

# Iterator

## ITERAATTORIN TUTKIMISTA

Kalle Mustonen | Suunnittelumallit | 30.04.2019

## 1. Johdanto

Tässä raportissa käydään läpi suunnittelumallit kurssilla annettu iteraattori tehtävä, jossa oli tarkoitus tutkia miten iteraattori käyttäytyy säikeiden kanssa erilaisissa tilanteissa.

Annetut tilanteet olivat:

- a) Tutki kuinka Javan iteraattori käyttäytyy, jos yritetään iteroida kokoelmaa kahdella säikeellä yhtä aikaa, kun molemmilla on oma iteraattori.
- b) entä, jos säikeet käyttävät samaa iteraattoria vuorotellen?
- c) Kuinka käy, jos kokoelmaan tehdään muutoksia iteroinnin läpikäynnin aikana.
- d) Etsi jotain muuta testattavaa.

Loin jokaisesta tehtävänannosta oman luokan, jotka toteuttavat tietorakenteen, sitä läpi käyvän iteraattorin ja tarvittavat säikeet.

## 2. Tehtävä a)

Loin TwoIterators nimisen luokan, joka sisälsi listarakenteen, johon generoitiin konstruktorissa käyttäjän haluaman määrän Dataolioita, jotka sisälsivät kokonaislukumuuttujan. Luokka sisälsi myös sisäluokan IteratorThread, jolla oli oma iteraattori. Tämän säikeen tarkoituksena oli iteroida tämä jaettu datalista läpi, kunnes lista oli iteroitu loppuun. Kuva 1 on kuva TwoIterators luokan koodista.

```

public class TwoIterators {

    private List<Data> dataList;

    public TwoIterators(int amount) throws InterruptedException {
        dataList = new ArrayList<>();
        for (int i = 0; i < amount; i++) {
            dataList.add(new Data(i + 1));
        }
        IteratorThread it1 = new IteratorThread();
        IteratorThread it2 = new IteratorThread();
        it1.start();
        it2.start();
        it1.join();
        it2.join();
    }

    private class IteratorThread extends Thread {

        private Iterator it;

        private IteratorThread() {
            it = dataList.iterator();
        }

        @Override
        public void run() {
            while (it.hasNext()) {
                try {
                    System.out.println(this.getName() + ", Data: " + it.next());
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

```

Kuva 1 TwoIterators luokka

Konstruktorissa luodaan lista, täytetään lista, luodaan kaksi säiettä ja sitten nämä säikeet käynnistetään.

## 2.1. HAVAINTOJA

Koodia ajettaessa ei ilmennyt mitään poikkeuksia, mutta koska molemmilla säikeillä oli omat iteraattorit, niin molemmat säikeet kävivät listan alusta loppuun tulostaen jokaisen alkion tiedot. Kuva 2 on kuva säikeiden tulostuksesta.

```

Määrä?
4
Thread-0, Data: 1
Thread-1, Data: 1
Thread-0, Data: 2
Thread-1, Data: 2
Thread-0, Data: 3
Thread-1, Data: 3
Thread-0, Data: 4

```

Kuva 2 Säikeiden tulostus.

### 3. Tehtävä b)

Tähän tehtävään loin luokan nimeltään SharedIterator, joka sisälsi myös oman listarakenteen ja säikeen sisäluokkana. Erona edellisen tehtävän luokkaan oli se, että kaikki säikeet käyttivät samaa iteraattoria synchronized lohkon sisällä. Kuva 3 on kuva SharedIterator luokan koodista.

```
public class SharedIterator {

    private List<Data> sharedDataList;
    private Iterator sharedIterator;

    public SharedIterator(int amount) throws InterruptedException {
        this.sharedDataList = new ArrayList<>();
        for (int i = 0; i < amount; i++) {
            sharedDataList.add(new Data(i + 1));
        }
        this.sharedIterator = sharedDataList.iterator();
        Sit sit1 = new Sit();
        Sit sit2 = new Sit();
        sit1.start();
        sit2.start();
        sit1.join();
        sit2.join();
    }

    public synchronized void iterateList(Sit sit) {
        try {
            System.out.println(sit.getName() + " iterating.");
            if (sharedIterator.hasNext()) {
                System.out.println(sharedIterator.next());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

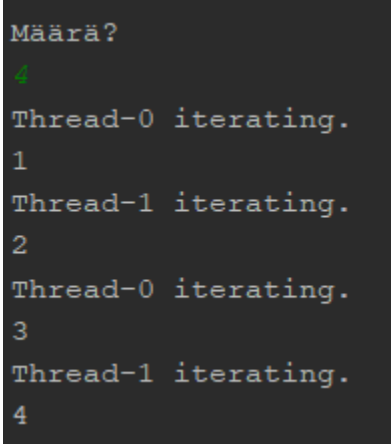
    private class Sit extends Thread {
        @Override
        public void run() {
            while (sharedIterator.hasNext()) {
                try {
                    iterateList(sit: this);
                    Thread.sleep(millis: 500);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Kuva 3 SharedIterator luokka

Samalla tavalla kuin edellisessä tehtävässä, konstruktorissa luodaan lista, joka sitten täytetään ja kaksi säiettä luodaan, jotka sitten myös käynnistetään. Säikeet sitten vuorotellen käyttävät samaa iteraattoria ja iteroivat sillä saman listan läpi.

### 3.1. HAVAINTOJA

Tässäkään ohjelmassa ei syntynyt mitään poikkeuksia ajon aikana. Testituloksista voidaan havaita kuinka säikeet yhdessä käyvät samaa listaa läpi tulostavat jokaisen alkion tiedot. Kuva 4 on kuva säikeiden tulostuksesta.

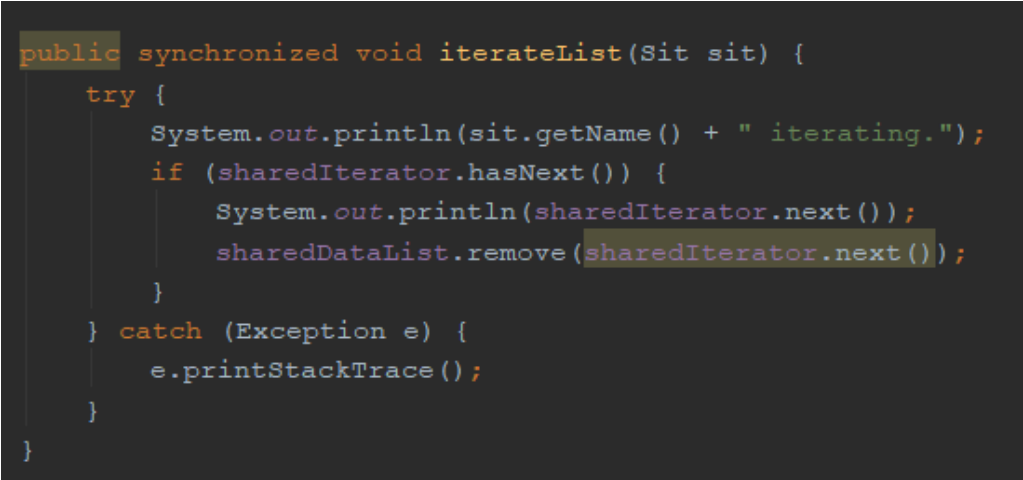


```
Määrä?  
4  
Thread-0 iterating.  
1  
Thread-1 iterating.  
2  
Thread-0 iterating.  
3  
Thread-1 iterating.  
4
```

Kuva 4 Säikeiden tulostukset

## 4. Tehtävä c)

Tähän tehtävään loin uuden luokan nimeltään ModifyWhileIterating. Luokka on hyvin samanlainen edellisen tehtävän luokan kanssa. Ainoa ero on se, että iteroinnin aikana listasta yritetään poistaa Dataolio. Kuva 5 on kuva ModifyWhileIterating luokan iterateList metodin koodista.



```
public synchronized void iterateList(Sit sit) {  
    try {  
        System.out.println(sit.getName() + " iterating.");  
        if (sharedIterator.hasNext()) {  
            System.out.println(sharedIterator.next());  
            sharedDataList.remove(sharedIterator.next());  
        }  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

Kuva 5 iterateList metodi

iterateList metodissa kutsutaan listan remove metodia, joka yrittää poistaa iteroinnin kohdalle olevan olion listasta. Tämä johtaa ajon aikaiseen poikkeukseen.

#### 4.1. HAVAINTOJA

Iteroinnin aikana listasta ei voida poistaa alkioita, koska tämä aiheuttaa poikkeuksen nimeltään: `ConcurrentModificationException`. Kuva 6 on kuva ajonaikaisesta tulostuksesta.

```
Määra?
1
Thread-0 iterating.
1
Thread-1 iterating.
java.util.ConcurrentModificationException
    at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1042)
    at java.base/java.util.ArrayList$Itr.next(ArrayList.java:996)
    at ModifyWhileIterating.iterateList(ModifyWhileIterating.java:28)
    at ModifyWhileIterating$Sit.run(ModifyWhileIterating.java:41)
Thread-0 iterating.
java.util.ConcurrentModificationException
    at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1042)
    at java.base/java.util.ArrayList$Itr.next(ArrayList.java:996)
    at ModifyWhileIterating.iterateList(ModifyWhileIterating.java:28)
    at ModifyWhileIterating$Sit.run(ModifyWhileIterating.java:41)
Thread-1 iterating.
java.util.ConcurrentModificationException
    at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1042)
    at java.base/java.util.ArrayList$Itr.next(ArrayList.java:996)
    at ModifyWhileIterating.iterateList(ModifyWhileIterating.java:28)
    at ModifyWhileIterating$Sit.run(ModifyWhileIterating.java:41)
```

*Kuva 6 Säikeiden virheilmoitus*

Javassa on kuitenkin mahdollisia ratkaisuja, jos halutaan poistaa listasta iteroinnin aikana. Esim. käyttämällä `removeIf` funktiota tai tietovirtojen avulla.

## 5. Tehtävä d)

Tähän tehtävään sai itse keksiä jotain muuta testattavaa iteraattorilla. Minua kiinnosti miten iteraattori iteroi Settiä ja sen konkreettista toteutusta HashSettiä.

Loin uuden luokan `MessingWithIterators`, missä luon `ArrayListin` ja `HashSetin` ja näille omat iteraattorit. Täytän molempiin listoihin saman verran `Dataolioita` samassa järjestyksessä ja samoilla arvoilla. Tämän jälkeen iteroin molemmat tietorakenteet läpi samanaikaisesti ja tulostan niiden sen hetkisen arvon. Kuva 7 on kuva `MessingWithIterators` luokan koodista.

```

public class MessingWithIterators {

    private Set<Data> dataSet;
    private List<Data> dataList;
    private Iterator setIterator;
    private Iterator listIterator;

    public MessingWithIterators(int amount) {
        this.dataSet = new HashSet<>();
        this.dataList = new ArrayList<>();
        for (int i = 0; i < amount; i++) {
            dataSet.add(new Data(i));
            dataList.add(new Data(i));
        }
        this.setIterator = dataSet.iterator();
        this.listIterator = dataList.iterator();
        System.out.println(setIterator.hasNext());
        while (setIterator.hasNext() && listIterator.hasNext()) {
            System.out.println("Set: " + setIterator.next().toString() + ", List: " + listIterator.next().toString());
        }
    }

}

```

*Kuva 7 MessingWithIterators luokka*

Set eroaa List tietorakenteesta kahdella keskeisellä erolla. List säilyttää lisäysjärjestyksen ja sallii duplikaatteja, kun taas Set ei säilytä lisäysjärjestystä, eikä salli duplikaatteja.

### 5.1. HAVAINTOJA

Koska Set on järjestämätön tietorakenne, niin voidaan hyvin olettaa, että näiden kahden iteraattorin tulostus eroaa toisistaan. Kuva 8 on kuva iteraattoreiden tulostuksesta.

```

Set: 11, List: 0
Set: 18, List: 1
Set: 2, List: 2
Set: 4, List: 3
Set: 7, List: 4
Set: 6, List: 5
Set: 3, List: 6
Set: 19, List: 7
Set: 9, List: 8
Set: 10, List: 9
Set: 12, List: 10
Set: 17, List: 11
Set: 15, List: 12
Set: 1, List: 13
Set: 16, List: 14
Set: 0, List: 15
Set: 14, List: 16
Set: 5, List: 17
Set: 13, List: 18
Set: 8, List: 19

```

*Kuva 8 Set ja List Iteraattoreiden tulostus*

Kuten arvata saattaa, niin Set on järjestänyt siihen syötetyt Dataoliot itsenäisesti HashSetin mukaisesti ja List on säilyttänyt syöttöjärjestyksen.