HSLU Hochschule Luzern

Algorithmen & Datenstrukturen (AD)

Übung: Weiterführende Konzepte (N3)

Themen: Semaphore, Java Thread Pool Executors, Future- und Callable-Schnittstelle, Callables und Executors, Atomic-Variablen, Thread-sichere Container, Blocking Queues
Zeitbedarf: ca. 240min.

Roger Diehl, Version 1.0 (FS 2025)

1 Suche nach grossen Primzahlen[1] (ca. 60')

1.1 Lernziele

- Sie können `Callable`, `Future` und `ExecutorService`, sowie Thread Pool Executoren anwenden.

1.2 Grundlagen

Diese Aufgabe basiert auf dem AD Input N31 (Callables und Executors) und dem OOP Input O13_IP_Datenströme. Zudem erhalten Sie die Code Vorlage `ch.hslu.ad.exercise.n3.prime`.

Die feste Länge der primitiven Datentypen `int` und `long` für Ganzzahlwerte reicht für diverse numerische Berechnungen nicht aus. Besonders wünschenswert sind beliebig grosse Zahlen in der Kryptografie. Für solche Anwendungen gibt es im `math`-Paket die Klasse `BigInteger` für Ganzzahlen.

Der Konstruktor `BigInteger(int numbits, Random rnd)` liefert eine Zufallszahl aus dem Wertebereich 0 bis $2^{numBits-1}$. Alle Werte sind gleich wahrscheinlich.

Die Methode `boolean isProbablePrime(int certainty)` gibt an, ob das `BigInteger` Objekt mit der Wahrscheinlichkeit `certainty` eine Primzahl ist. Die Methode ist blockierend. Je grösser `certainty` ist, desto mehr Zeit nimmt die Prüfung in Anspruch.

Hinter dieser Methode steckt der Miller-Rabin Test, eine Weiterentwicklung des Fermat Tests. Der Miller-Rabin Test ist ein probabilistisches Verfahren. Es ist effizient, aber es liefert die richtige Antwort nur in 99,999… % aller Fälle. Die Anzahl der Neunen hängt von der Anzahl der Iterationen (`certainty`) des Miller-Rabin-Tests ab.

`https://de.wikipedia.org/wiki/Miller-Rabin-Test`

1.3 Aufgabe

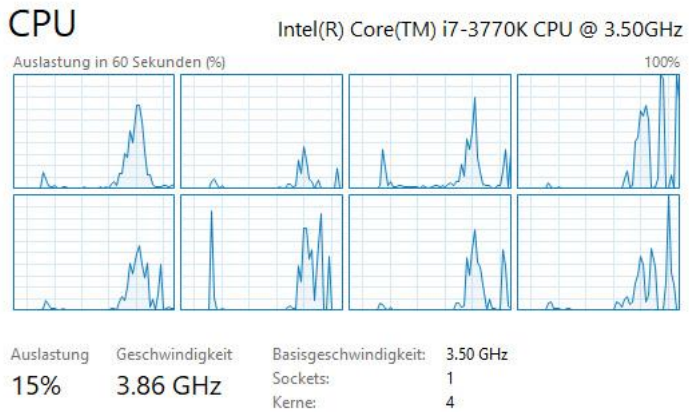
Erzeugen Sie `BigInteger` Objekte, die mindesten 1024 Binärstellen besitzen. Die `BigInteger` Objekte sollen auf Prim getestet werden und zwar mit der höchst möglichen Genauigkeit. Die `BigInteger` Objekte, die Primzahlen sind, sollen von einem Print-Task entgegengenommen und ausgegeben werden.

[1] Primzahlen stellen die Basis für verschiedene Verschlüsselungsverfahren dar. Das Prinzip des RSA Verfahrens zum Beispiel besteht darin, dass man leicht aus zwei geheim zuhaltenden grossen Primzahlen `p` und `q` das öffentlich bekanntzugebende Produkt `n=p*q` berechnen, aber nur sehr schwer die öffentlich bekannte Zahl `n` wieder in ihre beiden geheimen Primfaktoren `p` und `q` zerlegen kann. Sein Reiz besteht darin, dass der erforderliche Aufwand zur "Faktorisierung" bisher nicht bekannt ist. Man nimmt an, dass eine Zahl `n` von mindestens 1024 Binärstellen mit heutiger Rechner Technologie nicht "faktorisiert" werden kann.

Ziel ist es innert möglichst kurzer Zeit 100 grosse Primzahlen zu finden. Sequentiell kann man dies so tun:

```
int n = 1;
while (n <= 100) {
    BigInteger bi = new BigInteger(1024, new Random());
    if (bi.isProbablePrime(Integer.MAX_VALUE)) {
        LOG.info("{} : {}...", n, bi.toString().substring(0, 20));
        n++;
    }
}
```

Die CPU-Auslastung bei diesem Algorithmus ist jedoch minimal. Es wird gerade einmal im Schnitt einen Kern oder eine Threading Pipeline (falls die CPU Hyper-Threading anbietet) eingesetzt…



…und es dauert seine Zeit, bis man 100 grosse Primzahlen gefunden hat.

```
...
2025-03-23 09:01:35.114 [main] INFO  - 98 : 17866322638002439494...
2025-03-23 09:01:35.154 [main] INFO  - 99 : 72805543917242644978...
2025-03-23 09:01:35.306 [main] INFO  - 100 : 10745402069996513416...
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  25.369 s
Finished at: 2025-03-23T09:01:35+01:00
------------------------------------------------------------------------
```

Parallelisieren Sie den Algorithmus!

- Überlegen Sie sich, wie Sie den Algorithmus nebenläufig ausführen können.
- Überlegen Sie sich, welche Information die Applikation benötigt, damit sie weiss, wann die 100 grossen Primzahlen erreicht sind.
- Bedenken Sie, nicht jede zufällige generierte `BigInteger` Zahl ist eine Primzahl.

1.4 Reflektion

Reflektieren Sie die Aufgabe (hilft auch bei einer eventuellen Präsentation) und beantworten Sie sich die folgenden Fragen:

- Wie viele Threads lassen Sie laufen?
- Wie lange dauert es jetzt?
- Was passiert, wenn die Anzahl Threads verdoppeln, vervierfachen, verzehnfachen?
- Können Sie die Applikation noch schneller machen?

2 Speed Count (ca. 45')

2.1 Lernziele

- Sie können Thread-sichere Zugriffe auf einzelnen Variablen von elementaren Datentypen nachvollziehen, modifizieren und erstellen.
- Sie können einen Performance Vergleich mit Thread-sicheren Zugriffen auf Variablen machen.
- Sie können Thread Pool Executoren anwenden.

2.2 Grundlagen

Wir wollen den Performance-Gewinn bei Atomic-Variablen „sehen“. Deshalb basiert diese Aufgabe auf dem AD Input N31 (Atomic Variablen und Future- und Callable-Schnittstelle) und dem AD Input N22 (Java Thread Pool Executors). Für die Performance Messungen beachten Sie den Zusatzinput Z01 aus D2. Benutzen Sie die Code Vorlage `ch.hslu.ad.exercise.n3.count`.

2.3 Aufgabe

Im AD Input N31 ist ein Thread-sicherer Atomic Zähler vorgestellt worden. Diesen Zähler sollen Sie einem Synchronized Zähler gegenüberstellen und den Performance Unterschied messen. Um die Zeitmessung nachvollziehbar zu gestalten, sind folgende Klassen, bzw. Algorithmen vorgegeben.

- Gegeben ist die gemeinsame Schnittstelle `Count` für die beiden Thread-sichere Zähler Klassen. Die Zähler Klassen sollen die Schnittstelle `Count` implementieren.
- Gegeben ist ein Task, der ein Zähler Objekt Hoch- und Runterzählen kann. Dies macht die Klasse `CountTask`.
- Gegeben ist eine Klasse, welche die Thread-sicheren Zähler ausführt. Dies macht die Klasse `SpeedCount`, aber ohne Zeitmessungen.

Die Zeitmessung der beiden Zähler muss folgende Eigenschaften aufweisen.

- Die Zeitmessung muss nachvollziehbar und vor allem reproduzierbar sein.
- Um die Performance auszuweisen, muss man die Zeit eines Durchlaufes möglichst genau messen. Eine einzelne Messung wird immer eine etwas andere Zeit ergeben. Machen Sie deshalb **mindestens je fünf (besser mehr) Messungen**, wobei die erste Messung nicht mitgezählt werden darf.
- Erstellen Sie eine Tabelle mit den Messdaten und ermitteln Sie die durchschnittliche Laufzeit.
- Die Zeitmessung soll nach getaner Arbeit sauber beenden. Ein `System.exit` ist nicht erlaubt.

Nutzen Sie die Möglichkeiten, die Ihnen das Paket `java.util` und `java.util.concurrent` bietet.

2.4 Reflektion

Reflektieren Sie die Aufgabe (hilft auch bei einer eventuellen Präsentation) und beantworten Sie sich die folgenden Fragen:

- Was können Sie über die Performance der beiden Thread-sicheren Zähler aussagen?
- Was stellen Sie bei den Messresultaten fest?
- Wie erklären Sie sich die Messresultate?
- Welche Genauigkeit bezüglich gemessener Zeit erreicht Ihre Messung?

3 Bankgeschäfte (ca. 60')

3.1 Lernziele

- Sie können Thread-sichere Zugriffe auf einzelnen Variablen von elementaren Datentypen nachvollziehen, modifizieren und erstellen.
- Sie können Thread Pool Executoren anwenden.

3.2 Grundlagen

Diese Aufgabe basiert auf dem AD Input N31 (Atomic Variablen) und der Aufgabe 2 aus N1. Für die Performance Messungen beachten Sie den Zusatzinput Z01 aus D2. Sie erhalten die Code Vorlage `ch.hslu.ad.exercise.n3.bank`.

3.3 Aufgabe

Gegeben ist die gleiche Bankkonto Klasse, wie aus der Aufgabensammlung N1.

```java
/**
 * Einfaches Bankkonto, das nur den Kontostand beinhaltet.
 */
public final class BankAccount {

    private int balance;

    /**
     * Erzeugt ein Bankkonto mit einem Anfangssaldo.
     * @param balance Anfangssaldo
     */
    public BankAccount(final int balance) {
        this.balance = balance;
    }

    /**
     * Erzeugt ein Bankkonto mit Kontostand Null.
     */
    public BankAccount() {
        this(0);
    }

    /**
     * Gibt den aktuellen Kontostand zurück.
     * @return Kontostand.
     */
    public int getBalance() {
        return this.balance;
    }

    /**
     * Addiert zum bestehen Kontostand einen Betrag hinzu.
     * @param amount Einzuzahlender Betrag
     */
    public void deposite(final int amount) {
        this.balance += amount;
    }
```

```
    /**
     * Überweist einen Betrag vom aktuellen Bankkonto an ein Ziel-Bankkonto.
     * @param target Bankkonto auf welches der Betrag überwiesen wird.
     * @param amount zu überweisender Betrag.
     */
    public void transfer(final BankAccount target, final int amount) {
        this.balance -= amount;
        target.deposite(amount);
    }
}
```

a) Mit Instanzen dieser Klasse sollen Sie Überweisungen tätigen. Wie schon in N1 sollen tausende von Überweisung pro Sekunde stattfinden. Auch sind Zugriffe denkbar, die gleichzeitig auf ein Konto gemacht werden. Diesen Sachverhalt wollen wir simulieren. Gegeben ist folgendes Szenario (`ch.hslu.ad.exercise.n3.bank`):

- Es gibt eine Liste von Quell Konten und eine Liste von Ziel Konten.
- Ein Betrag wird von den Quell Konten an die Ziel Konten überwiesen und wieder zurück.
- Die Überweisung von der Quelle zum Ziel und zurück wird von einem Bankauftrag (Klasse `AccountTask` – Klasse aus N1) mit Hilfe eines Threads gemacht.
- **Neu:** Verwenden Sie einen Thread Pool Executor für diese Aufgabe. Es darf keine Anweisung `new Thread` mehr in Ihrem Code stehen!

b) Korrigieren Sie die Schwachstelle der Bankkonto Klasse mit Hilfe von `Atomic` Instanzen.

c) Messen Sie die Performance der Bankgeschäfte der Aufgaben aus N1 und N3. Nehmen Sie dazu einen grossen Betrag (grösser als 100’000) für die Überweisung. Zudem soll die Liste der Konten umfangreich (grösser als 5) sein. Machen Sie mindestens je fünf (besser mehr) Messungen, wobei die erste Messung nicht mitgezählt werden darf. Erstellen Sie eine Tabelle mit den Messdaten und ermitteln Sie die jeweilige durchschnittliche Laufzeit.

3.4 Reflektion

Reflektieren Sie die Aufgabe (hilft auch bei einer eventuellen Präsentation) und beantworten Sie sich die folgenden Fragen:

- Was können Sie über die Performance der beiden Varianten Bankgeschäfte der Aufgabensammlung N1 und N3 aussagen?
- Was stellen Sie bei den Messresultaten fest?
- Welche Genauigkeit bezüglich gemessener Zeit erreicht Ihre Messung?

4 Container Thread-sicher machen (ca. 75')

4.1 Lernziele

- Sie kennen die Wrapper Klassen um herkömmliche Datenstrukturen Thread-sicher zu gestalten.
- Sie können `Callable`, `Future` und `ExecutorService`, sowie Thread Pool Executoren anwenden.

4.2 Grundlagen

Diese Aufgabe basiert auf dem AD Input N31 (Callables und Executors, sowie Thread-sichere Container). Zudem erhalten Sie die Code Vorlage `ch.hslu.ad.exercise.n3.conclist`.

Für diese Aufgabe gibt es mehrere Produzenten, die Integer-Zahlen in der folgenden Art produzieren und die Summe der produzierten Zahlen zurückgeben.

```
private final List<Integer> list;
//...
long sum = 0;
for (int i = 1; i <= maxRange; i++) {
    sum += i;
    list.add(i);
}
return sum;
```

Und es gibt einen Konsumenten, der die Integer-Zahlen ausliest und die Summe der gelesenen Zahlen zurückgibt.

```
private final List<Integer> list;
//...
long sum = 0;
Iterator<Integer> iterable = list.iterator();
while (iterable.hasNext()) {
    sum += iterable.next();
}
return sum;
```

4.3 Aufgabe

a) Gegeben ist die Klasse `DemoConcurrentList` bei der die Produzenten und Konsumenten jeweils auf eine gemeinsame Liste (`LinkedList`) zugreifen lassen. Dabei werden alle Summen der Produzenten ausgelesen, zu einem Total addiert und verglichen mit der Konsumentensumme.

Reflektion: Was stellen Sie fest?

b) Machen Sie aus der gemeinsamen Liste eine synchronisierte Liste.

Reflektion: Was stellen Sie jetzt fest?

c) Erstellen Sie eine Demonstration mit einer **Blocking Queue** und vergleichen Sie, welche Thread-sichere Container schneller, d.h. effizienter sind.