HOCHSCHULE KONSTANZ
TECHNIK, WIRTSCHAFT UND GESTALTUNG

Verteilte Systeme
Übung

29. März 2017

Aufgabenblatt 1:
Threads, Kollaboration & Kommunikation

Abgabe bis: 03.05.2017

Prof. Dr. Rainer Mueller

SS 2017

Aufgabe 1: Thread-Synchronisierung (Wiederholung zur REKO-Auffrischung)

1

Dieser Aufgabe liegt folgende einfache Thread-Implementierung zugrunde:

```
public class MyThread extends Thread {
    private static final int threadMax=10;
    private static int runCount=0;

    public void run() {
        while(runCount++<100) {
            System.out.println(runCount+": "+Thread.currentThread().getName());
        }
    }

    public static void main(String args[]) {
        for(int i=0;i<threadMax;i++) {
            new MyThread().start();
        }
    }
}
```

a. Starten Sie die Ausführung dieses Programms. Was macht das Programm und was fällt Ihnen auf?

b. Überführen Sie jeden Thread nach der Standardausgabe für 1 s in einen Schlafzustand. Erzeugen Sie für den While-Schleifenrumpf eine separate synchronisierte Methode. Warum erfolgt die Ausgabe der Threads trotz Synchronisierung immer noch nebenläufig (also nicht die Ausgabe und das Schlafen nacheinander Thread für Thread)?

Aufgabe 1: Thread-Synchronisierung

c. Wie können Sie diese Nebenläufigkeit aus Aufgabenteil b. beseitigen, sodass in der Standardausgabe pro Sekunde genau eine Zeile dargestellt wird?

d. Leiten Sie die ursprüngliche Klasse `MyThread` aus Aufgabenteil a. (also ohne synchronisierte Methode aus Aufgabenteil b.) von einer Klasse `MyAccount` ab, die einen einzelnen Integer-Wert als statische Variable mit zugehörigen gettern/settern enthält und selbst von keiner weiteren Klasse abgeleitet ist. Inkremetieren und dekrementieren Sie diesen Integer-Wert in `MyAccount` zufällig in der `run`-Methode von `MyThread`. Dokumentieren Sie die Wert-Veränderungen durch entsprechende Standardausgaben, wie unten angegeben. Verwenden Sie pro Account-Veränderung und zugehöriger Standard-Ausgabe genau einmal `get` und `set`. Beobachten Sie die Konsistenz der Wertveränderung und implementieren Sie ggf. eine konsistente Wertveränderung.

```
Thread-1: 0 + 1 = 2      // <get() am Anfang> + <Inkrement/Dekrement> = <get() nach set()>
Thread-5: 0 + -1 = 2
Thread-6: 2 + -1 = 3
Thread-2: 2 + 1 = 2
Thread-9: 3 + 1 = 1
Thread-0: 3 + 1 = 4
Thread-7: 3 + -1 = 4
```

Aufgabe 2: Threads mit Anfragen und Antworten

2

Das folgende Beispiel realisiert eine verbindungslose nachrichtenorientierte Kommunikation zwischen den Komponenten eines verteilen Systems:

- `PrimeServer.java`: Server-Klasse, die zu einer empfangenen Zahl berechnet, ob diese eine Primzahl ist. Der Server ist nicht nebenläufig realisiert. Diese Klasse verwendet `rm.requestResponse.jar` zur Realisierung der Kommunikation.
- `PrimeClient.java`: Interaktive Client-Klasse, die der Reihe nach eine vorgegebene Anzahl von Zahlen beginnend bei einem vorgegebenen Startwert an den obigen Server übergibt. Das vom Server empfangene Ergebnis des Primzahltest wird in der Console ausgegeben. Diese Klasse verwendet `rm.requestResponse.jar` zur Realisierung der Kommunikation.
- `rm.requestResponse.jar`: Java-Archiv mit einfachen Klassen zur Realisierung einer verbindungslosen nachrichtenbasierten Kommunikation im Sinne von einfachen Request-Response-Paaren
- `rm.requestResponse.javadoc.zip`: Javadoc-Dokumentation zu rm.requestResponse.jar

Zur Verwendung von `PrimeServer.java` und `PrimeClient.java` muss `rm.requestResponse.jar` in den Build Path mit eingebunden werden. In Eclipse erreichen Sie dies beispielsweise über das Kontextmenü (Rechts-Click) des Projekts:

```
Build Path → Add External Archive
```

Hinweis zur ersten Hürde: Starten Sie den Server vor dem Client ☺

Aufgabe 2: Threads mit Anfragen und Antworten

2

a. Bei hohen Startwerten erfolgt teilweise lange keine Rückmeldung des Clients im Sinne einer Ausgabe an den Anwender. Dies hat mindestens drei verschiedene Ursachen. Welche sind das?

b. Erweitern Sie den Client um eine weitere Anwendereingabe, über die der Anwender die Art des Anfragemodus an den Server bestimmen kann, zum Beispiel:

```
Server hostname [localhost] >
Server port [1234] >
Request mode [SYNCHRONIZED] >
Prime search initial value [100000000] >
Prime search count [5] >
```

NEU

c. Erweitern Sie den Client um einen weiteren Abfragemodus, bei dem die Abfrage auf vorliegendes Ergebnis zu einer Primzahlanfrage nicht blockierend in gleichbleibenden kurzen Abständen erfolgt (Polling) und dem Anwender dazu eine Information auf dem Bildschirm ausgegeben wird. Der Anwender kann dabei über die in b. realisierte Eingabemöglichkeit zwischen beiden Modi – der alten blockierenden und der neuen nicht-blockierenden – wählen.

Ausgabe im blockierenden „alten“ Anfragemodus:

```
100000000000000003:                    Während der Anfrage
100000000000000003: prime              Nach der Anfrage
```

...

Aufgabe 2: Threads mit Anfragen und Antworten

2

Ausgabe im nicht blockierenden „neuen“ Anfragemodus:

```
10000000000000003: .......              Während der Anfrage
10000000000000003: .......... prime    Nach der Anfrage
```

d. Erweitern Sie den Client um einen weiteren Abfragemodus, der zwar blockierend aber nebenläufig auf Client-Seite erfolgt und damit aus Anwendersicht dasselbe Verhalten wie der nicht blockierende Anfragemodus aus c. zeigt. Anwenderausgabe und Serveranfrage erfolgen dabei nebenläufig.

   Hinweis: Der Anwender hat am Ende also die Möglichkeit, interaktiv zwischen drei verschiedenen Anfragemodi zu wählen: blockierend, nicht-blockierend, blockierend und nebenläufig.

e. Testen Sie das Szenario mit mehreren nebenläufigen Clients und nennen Sie mögliche Gründe für die Ursachen des gezeigten Verhaltens.

Aufgabe 3: Thread-Serialisierung (Wiederholung zur REKO-Auffrischung)

3

Diese Aufgabe stellt bereits folgende Klassen bereit:

- `SerializeMain`: Hauptprogramm
- `MySerializer`: Serialisierung und Deserialisierung (unvollständig)
- `MySerializableClass`: Serialisierbare Klasse (vollständig, aber anzupassen)
- `MyNonSerializableClass`: Nicht serialisierbare Klasse

Diese Klassen sind wie folgt vorgegeben:

```
import java.io.*;

public class SerializeMain {
    private static MySerializableClass myObject;
    private static MySerializer mySerializer;

    public static void main(String args[]) throws IOException, ClassNotFoundException {
        String command,text="";
        boolean doExit=false;

        myObject=new MySerializableClass();
        mySerializer=new MySerializer(myObject);

        BufferedReader reader=new BufferedReader(new InputStreamReader(System.in ));

        printMenu();
```

Aufgabe 3: Thread-Serialisierung


```
        while(!doExit){
            System.out.print("> ");

            command=reader.readLine();

            switch(command){
                case "t":   System.out.print("text> ");
                            text=reader.readLine();
                            System.out.println("Text: "+text);
                            break;
                case "s":   mySerializer.write(text); break;
                case "d":   System.out.println(mySerializer.read());
                            break;
                case "m":   printMenu(); break;
                case "x":   doExit=true; break;
            }
        }
    }

    private static void printMenu() {
        System.out.println("--------------------");
        System.out.println("se(t) text");
        System.out.println("(s)erialize");
        System.out.println("(d)eserialize");
        System.out.println("print (m)enu");
        System.out.println("\ne(x)it");
        System.out.println("--------------------");
    }
}
```

Aufgabe 3: Thread-Serialisierung

3

```
import java.io.*;

public class MySerializer {
    private MySerializableClass mySerializableClass;

    MySerializer(MySerializableClass serializableClass) {
        mySerializableClass=serializableClass;
    }

    private String readFilename() throws IOException {
        String filename;
        BufferedReader reader=new BufferedReader(new InputStreamReader(System.in ));

        System.out.print("filename> ");
        filename=reader.readLine();

        return filename;
    }

    public void write(String text) throws IOException {
        mySerializableClass.set(text);
        String filename=readFilename();

        // Implementierung erforderlich
        // Serialisiere mySerializableClass in Datei

    }
```

Aufgabe 3: Thread-Serialisierung

3

```
    public String read() throws IOException, ClassNotFoundException {
        String filename=readFilename();

        // Implementierung erforderlich
        // Serialisiere mySerializableClass von Datei

        return mySerializableClass.toString();
    }
}
public class MySerializableClass implements Serializable{
    private static final long serialVersionUID=1;
    private int id;
    private String string;

    MySerializableClass() {
        id=1234;
    }

    public void set(String string) {
        this.string=string;
    }

    public String toString() {
        return "id: "+id+"; string: "+string;
    }
}
```

Aufgabe 3: Thread-Serialisierung

3

```
public class MyNonSerializableClass {
    private int id;

    MyNonSerializableClass() {
        id=5678;
    }

    public String toString() {
        return "id: "+id;
    }
}
```

Aufgabe 3: Thread-Serialisierung

a. Vervollständigen Sie die Implementierung von `write` und `read` in der Klasse `MySerializer`.

b. Fügen ein Objekt der Klasse `MyNonSerializableClass` zu `MySerializableClass` hinzu und berücksichtigen Sie diese Instanzvariable im Konstruktur und in `toString`. Was beobachten Sie bei der Serialisierung?

c. Verwenden Sie Transienz bei der `MyNonSerializableClass`-Instanzvariable. Was beobachten Sie jetzt bei der Serialisierung und Deserialisierung?