PR Rechnernetze und Netzwerkprogrammierung 621.801/621.802	WS 2024/2025
Institut für Informationstechnologie (ITEC)	Rainer/Timmerer
Übungsblatt 01	

Ü 1.1 Java-I/O Einführung

Machen Sie sich mit den Java-I/O-Klassen vertraut und beantworten Sie folgende Fragen:

a) Erklären Sie die Klassen `OutputStream` und `InputStream`.

b) Was sind `FilterInputStreams` bzw. `FilterOutputStreams`?

c) Erklären Sie die Klassen

1. `BufferedInputStream`,
2. `DataOutputStream`,
3. `CipherInputStream`,
4. `ZipOutputStream` und
5. `PushbackInputStream`

d) Erklären Sie kurz, was `Readers` und `Writers` sind.

Hinweis: https://docs.oracle.com/javase/tutorial/essential/io/

Ü 1.2 Java-I/O Programmierung

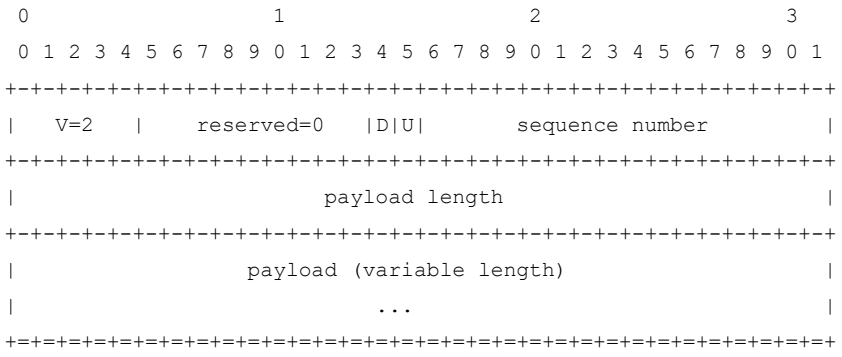
Schreiben Sie ein Java-Programm, das mit Hilfe der I/O Klassen folgende Funktionalität erfüllt. Das Programm soll als Kommandozeilenparameter den Dateinamen einer Textdatei entgegennehmen. Diese Textdatei sei UTF-8 kodiert und verwende Zeilenvorschub (*engl. line feed*, LF) für den Zeilenumbruch (eine Reihe von Testdaten finden Sie im Moodle Kurs in der Zip-Datei `utf8files.zip`). Das Programm soll dabei den Inhalt der Datei nach ISO 8859-1 konvertieren, Wagenrücklauf (*engl. carriage return, CR*) und Zeilenvorschub (CR+LF) als Zeilenumbruch verwenden und den resultierenden Inhalt Zip-komprimiert in eine Ausgabedatei schreiben. Der Name der Ausgabedatei sollte ebenfalls als Kommandozeilenparameter übergeben werden.

Sorgen Sie bei inkorrekten Eingaben für eine entsprechende Fehlerbehandlung.

Ü 1.3 Java-Byte Arrays

Im Gegensatz zu bekannten Netzwerkprotokollen wie HTTP oder POP3 werden ausgetauschten Nachrichten oft nicht in Textform sondern als Sequenz von Feldern unterschiedlichen Typs (z.B. 16-Bit-Integer-Werte, 1-Bit-Flags, etc.) dargestellt. In Java werden solche Nachrichten im Allgemeinen als Byte-Array (`byte []`) repräsentiert. Für die Erstellung/Manipulation solcher Arrays bieten sich Klassen wie z.B. `java.nio.ByteBuffer` (nio steht hier für non-blocking I/O) an.

Gegeben sei ein Nachrichtenformat für ein proprietäres Transportprotokoll:



Die Nachricht besteht also aus einem Versionsfeld V (5 Bit, fixer Wert 2), einer Reihe von reservierten Bits (9 Bit, immer 0), den Flags (jeweils 1 Bit) isData (D) bzw. isUrgent (U), einer 16 Bit Sequenznummer, einem 32 Bit Wert, der die Länge der eigentlichen Nutzdaten (payload length) signalisiert, und den eigentlichen Nutzdaten (payload). Sowohl Sequenznummer und Nutzdatenlänge sollen als positive, ganzzahlige Werte in Network Byte Order übertragen werden.

Schreiben Sie eine Funktion `createMsg`, die aus den übergebenen Parametern eine Nachricht erzeugt:

```
static byte[] createMsg (boolean isData,
                         boolean isUrgent,
                         int sequenceNumber,
                         byte [] payload) throws IllegalArgumentException;
```

Fehlerhafte Parameter (z.B. `sequenceNumber` > (2^{16}-1) oder leere Payload) sollen eine `IllegalArgumentException` verursachen. Testen Sie Ihre Implementierung mit entsprechenden Eingaben.

Ü 1.4 Java-Threads: Programmierung

Hinweis: https://docs.oracle.com/javase/tutorial/essential/concurrency/

Implementieren Sie ein paralleles Programm, das mit Hilfe von Threads eine komplizierte Berechnung mit Daten eines Float-Arrays berechnet. Im System gibt es *n* Threads, die jeweils einen Teil des Arrays (ein *n*-tel, beachten Sie dabei, dass die Länge des Arrays nicht ohne Rest durch n teilbar sein muss) berechnen und das Ergebnis zu einem globalen Endergebnis addieren. Wenn alle Threads mit ihren Berechnungen fertig sind, gibt das Hauptprogramm das Endergebnis am Bildschirm aus. Das Array befindet sich in einer Datei mit folgendem Format: beliebige Bezeichnung (*String/UTF*), gefolgt von einer ID (*int*), gefolgt von der Anzahl der Float-Werte (*int*) und dann die eigentlichen Float-Werte (*float*).

Es existiert bereits die Klasse *TheRealThing*, die die Klasse Threads erweitert. Den Source finden Sie weiter unten. Ergänzen Sie die Klasse *TheRealThing* in der *run*-Methode (und eventuell neuen oder bestehenden Hilfsmethoden, falls notwendig), die die Berechnung durchführt. Denken Sie daran, etwaige Locks so feingranular wie möglich zu setzen! Vervollständigen Sie den Code der *main*-Methode, die die Threads mit dem korrekten *start* und *end* startet.

Testen Sie Ihre Implementierung mit geeigneten Testdaten.

```
public class TheRealThing extends Thread {
	private static float result = -1;
	private String filename;
	private int start;
	private int end;

	/**
	 * Creates a new TheRealThing thread which operates
	 * on the indexes start to end.
	 */
	public TheRealThing(String filename, int start, int end) {
		this.filename = filename;
		this.start = start;
		this.end = end;
	}

	/**
	 * Performs "eine komplizierte Berechnung" on array and
	 * returns the result
	 */
	public float eine_komplizierte_Berechnung(float[] array) {
		// TODO ... erfinden Sie etwas, seien Sie kreativ!
	}

	public void run() {
		// TODO ...
	}

	public static void main(String[] args) {
		String pathToFile = "./myArrayData.dat";
		int numThreads = 12;
		int arraySize = 70;

		// TODO ...
	}
}
```

Ü 1.5 Java Thread Safety

Recherchieren Sie was Thread Safety im Zusammenhang mit Java Klassen bedeutet.

Ü 1.6 Java Thread Safe Class

Schreiben Sie eine Klasse die eine Liste an Integern speichert. Die Klasse soll Thread Safe implementiert werden. Das heißt Sie müssen darauf achten, dass Manipulationen an der internen Repräsentation der Liste geschützt von statten gehen. Testen Sie ihre Implementierung mit mehreren Threads, die dieselbe Instanz der Klasse verwenden um ganze Zahlen zu speichern und zu lesen.

```
public class IntegerList {
	protected int[] data;
	protected int size;
	private static final int DEFAULT_SIZE = 10;

	public IntegerList( ) {
		data = new int[DEFAULT_SIZE];
	}

	public IntegerList(IntegerList toCopy) {
		/**
		 * copy the original
		 */
	}

	/**
	 * TODO:
	 * Implement:
	 * get(int index) - Return the integer stored at index
	 * add(int value) - Add a new value to the integer list
	 * clear() - Deletes all integers
	 * setCapacity(int n) - Reallocates the data array increasing or
decreasing its size to n
	 * toArray() - Returns a copy of the integer list and returns it
	 *
	 */

}
```

Ü 1.7 Java-Threads: Consumer - Producer

Implementieren Sie ein paralleles Programm, welches ein klassisches Consumer Producer Problem löst. Hierzu soll der Producer-Thread Zahlen generieren die dann in einen gemeinsamen „Container“ (Speicher) geschrieben werden (achten Sie dabei auf die nötige Synchronisation von Consumer und Producer). Sobald Zahlen in dem gemeinsamen Speicher zur Verfügung stehen soll zufällig eine Anzahl an vorhandenen (minimal eine Zahl, maximal so viel Zahlen wie in dem gemeinsamen Container vorhanden sind) Zahlen aus dem gemeinsamen Container entnommen und deren Summe berechnet werden. Warten Sie nach dem berechnen der Summe 10ms.
Hinweis: Verwenden Sie die Stack Klasse (java.util.*) als gemeinsamen Speicher.

```
public class Consumer extends Thread {

	private Stack<Integer> stack;

	public Consumer(Stack<Integer> stack)
	{
		this.stack = stack;
	}

	public void run()
	{
		// TODO: implement!
	}

}
public class Producer extends Thread {

	private Stack<Integer> stack;

	public Producer(Stack<Integer> stack)
	{
		this.stack = stack;
	}

	public void run()
	{
		// TODO: implement!
	}
}


public class Main {

	public static void main(String[] args)
	{
		Stack<Integer> sharedStack = new Stack<>();
		// TODO: implement
	}
}
```