

- a) Programmieren Sie eine Pipe mit `PipedInputStream` und `PipedOutputStream` aus dem `java.io` Paket. Es sollen von einem Thread A ein Feld mit `INHALT_GROESSE` Zahlen vom Typ `byte` in die Pipe eingeschrieben werden. Dazu soll die Methode
- ```
void write(byte[] b, int off, int len)
```
- verwendet werden. Ein Thread B soll in einer Endlosschleife jeweils 1ms schlafen und dann ein Byte mit der Methode
- ```
int read()
```
- auslesen und auf dem Bildschirm ausgeben.
- I) Schreiben Sie in Thread A ein Feld mit 1024 byte Zahlen in die Pipe und lassen Sie die Pipe gleichzeitig durch Thread B auslesen.
 - II) Schliessen Sie in dem Thread A den Stream mit `close()`, nachdem die 1024 byte Zahlen geschrieben wurden. Wie verändert sich das Programmverhalten? Was hat es mit der -1 auf sich. Was ist ein Stromschlusszeichen? Entfernen Sie jetzt den `close()` Befehl wieder.
 - III) Was passiert, wenn `INHALT_GROESSE` kleiner gleich 1024 ist und Thread B nicht gestartet wird?
 - IV) Was passiert, wenn `INHALT_GROESSE` größer als 1024 ist und Thread B nicht gestartet wird?
 - V) Gehen Daten verloren, wenn `INHALT_GROESSE` größer als 1024 ist und die obige `write` Methode aufgerufen wird (Thread B arbeitet wie oben beschrieben)?
 - VI) Was passiert, wenn `INHALT_GROESSE` mindestens 1026 ist und Thread B gestartet wird und keine Schleife, sondern nur einen einzigen `read` Befehl enthält?

Begründen Sie Ihre Antworten.

Füllen Sie folgende Tabelle aus:

Programm version	Läuft das Programm fehlerfrei zuende, wird eine Exception geworfen oder verklemmt das Programm?	Wenn eine Exception geworfen wird, welche und mit welchem Text?	Wie ist das Verhalten des Programms jeweils zu erklären?
I)			
II)			
III)			
IV)			
VI)			

- b) `pop()` soll die Methode sein, die einer Queue einen neuen Wert entnimmt. Wenn die Queue leer ist, gerät der Thread in den wait Zustand. Warum wird sobald der Thread in `pop` in den waiting Status gerät die Schleife nie verlassen, selbst wenn ein Interrupt gesetzt wird?

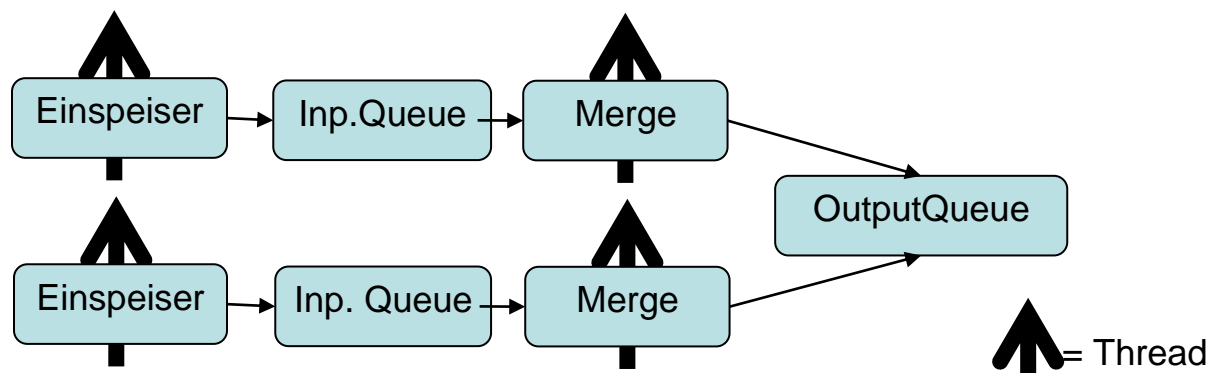
```
while (!isInterrupted()) {  
    try {  
        int zahl = 0;  
        zahl = input.pop();  
        output.push(zahl);  
    } catch (InterruptedException e) {};  
}
```

c) Programmieren Sie folgende **Verarbeitungskette für Integer Zahlen**.

Funktionsweise

In der folgenden Abbildung ist eine Verarbeitungskette für Integer Zahlen dargestellt. Am Anfang der Verarbeitungskette stehen zwei **Einspeiser** Objekte die jeweils in einem eigenen Thread laufen. Sie haben folgende Aufgabe:

- Das obere Einspeiser Objekt produziert fünf Mal die Zahl eins und speist sie in die obere Input Queue ein.
- Das untere Einspeiser Objekt produziert fünf Mal die Zahl null und speist sie in die untere InputQueue ein.
- ACHTUNG: alle vier Threads, die in dieser Grafik als Pfeile dargestellt sind, müssen gleichzeitig gestartet werden. Wenn die Merge Threads noch nichts in den InputQueues finden, müssen sie warten!



Dann folgt in der Verarbeitungskette jeweils oben und unten ein Merge Objekt. Diese beiden Merge Objekt laufen jeweils in einem Thread. Die run Methoden der Merge Objekte funktionieren folgendermaßen:

- Eine Schleife wird ausgeführt. In einem Schleifendurchgang holt sich das Merge Objekt eine int Zahl aus der zugehörigen InputQueue (Methode get der Input Queue). Dann wird diese int Zahl in das OutputQueue Objekt geschrieben (Methode put der OutputQueue).

Wenn das Programm fertig gelaufen ist, sind in der OutputQueue 5 Nullen und 5 Einsen in zufälliger Reihenfolge eingetragen.

Folgende Randbedingungen müssen eingehalten werden:

Einspeiser

- Wenn die Queue voll ist, soll der Einspeiser warten. Zwischen jeder Produktion einer Zahl soll zufällig zwischen 0 und 2 Sekunden geschlafen werden. Jede produzierte Zahl soll angezeigt werden

InputQueue, OutputQueue

- Die beiden InputQueues und die OutputQueue sollen alle Objekte derselben Klasse Queue sein. Queue soll wie der in der Vorlesung beschriebene Ringpuffer (N Buffer) implementiert sein.

- Falls eine der InputQueues leer sein sollte, darf der Prozess nicht bei dem Versuch hängen bleiben, Daten aus den Input Queues herauszunehmen.
- Außerdem soll die Verschmelzung der Datenströme in folgendem Sinne beliebig erfolgen: die Einordnung der Daten in die OutputQueue soll in zufälliger Ordnung passieren. Die Erzeugung und das Mergen sollen parallel passieren, d.h. **nicht** erst alle Zahlen erzeugen und in die Input Queues schreiben und dann erst mergen.

Merge

- Die zum einem Merge Objekt gehörige InputQueue kann auch mal leer sein (weil z.B. der Einspeiser zu langsam Zahlen einspeist). Dann soll das Merge Objekt warten, bis wieder Zahlen verfügbar sind.
- Die Queues bzw. die Merge Objekte sollen nicht wissen, wie viel Zahlen die Einspeiser einspeisen werden. Um die Merge Objekte zu beenden, muss man ihnen daher einen Interrupt schicken, irgendwann, nachdem die Einspeiser ihr Werk getan haben.
- Nach dem ein Merge Objekt einen Interrupt erhalten hat (siehe Beschreibung main-Methode) soll das Merge Objekt auf jeden Fall noch die InputQueue ganz auslesen, bevor es sich beendet.

main Methode

- Zum Schluss soll der Inhalt des Output Queue angezeigt werden. Es müssen zehn Ziffern sein, davon 5 Einsen und 5 Nullen in zufälliger Reihenfolge.
- Es sollen alle fünf Threads (2 Einspeiser Threads, 2 Merger Threads, 1 main Thread) korrekt beendet werden (Interrupt benutzen! Vergessen Sie nicht, nach dem Interrupt die Input Queue ganz auszulesen). Mit dem Ende des main Thread soll auch das Java Programm enden. Das Programm muss selbstverständlich auch bei wiederholtem Durchlauf korrekt funktionieren.

Im Folgenden finden Sie Vorlagen für die Klassen Queue, Merge und Einspeiser:

```
class Queue {
    Queue(int groesse, String name) {...}
    int get() {...}
    void put(int zahl) {...}
    boolean isEmpty() {...}
    void print() {...} // Inhalt der Queue auf Bildschirm
}

class Merge extends Thread {
    Merge(Queue input, Queue output) {...}
    public void run() {...}
    // in main Threads starten und beenden, join() benutzen!
    public static void main(String[] args) {...}
}
```

```
class Einspeiser extends Thread {  
    // kennwert ist die Zahl, die dieser Einspeiser  
    // einspeisen soll  
    Einspeiser(Queue zielQueue, int kennWert) {...}  
    public void run() {...}  
}
```