

Woche 11

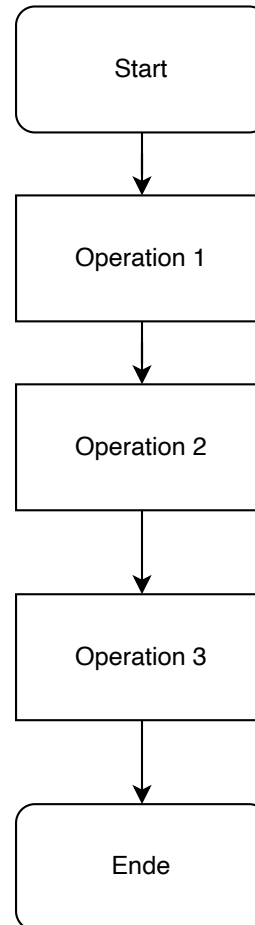
Threads und Concurrency I

Vorwort

- Diese Woche schauen wir uns viel Code an
- Damit ihr euch das entspannt auf eurem Laptop anschauen könnt, ist der gesamte Code in der Aufgabe W11P00 - Threads Playground enthalten
- Bitte cloned und öffnet den Code bei euch lokal während des Tutoriums!

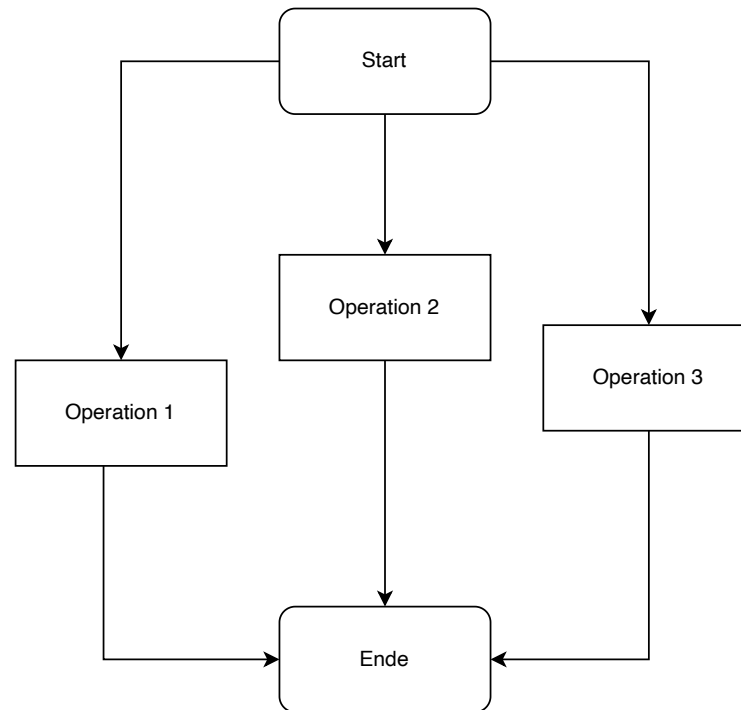
Unsere Programme bisher...

- Programme laufen sequentiell alle Programm-Schritte ab.

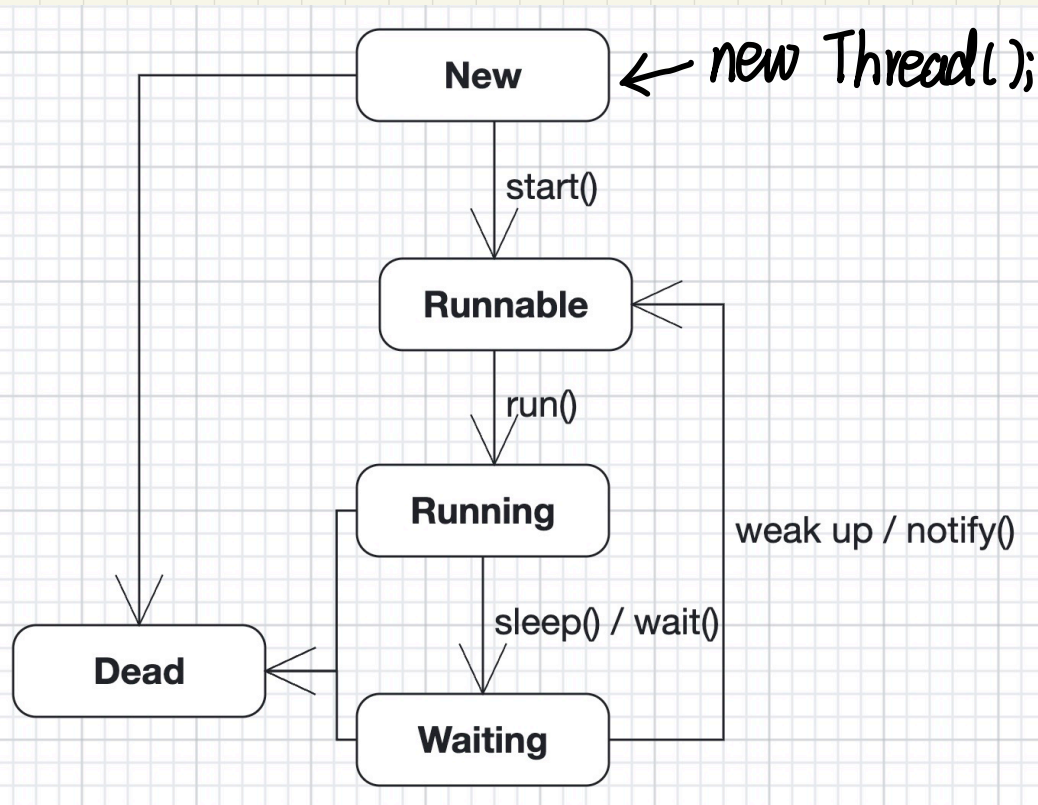


Threads

- Threads erlauben die quasi-parallele Ausführung von Programmen
- Kann den Code schneller machen
- Operationen können in beliebiger Reihenfolge ablaufen!
- Erhöht Komplexität des Programms und ist nicht trivial in den meisten Fällen!



Thread Life Cycle:



1. New (Neu)

- `new Thread ()`;
- Der Thread wird erstellt, aber noch nicht gestartet.

2. Runnable (Lauffähig)

- `start()`-Methode wurde aufgerufen,
- Der Thread ist bereit, ausgeführt zu werden
- Wichtig: der Thread wird hier nicht unbedingt gerade ausgeführt, sondern darauf wartet, vom Scheduler ausgewählt zu werden

3. Running (Laufend)

- `run()`-Methode aufgerufen wurde Der Thread führt die Anweisungen seines Codes aus.
- Der Thread beansprucht einen Prozessor für sich und führt Code aus.

4. Blocked/Waiting (Blockiert/Wartend)

- Der Thread kann in einen Blockierungs- oder Wartezustand wechseln, wenn
- Thread auf eine Ressource wartet (z.B. Datei-I/O, Netzwerkzugriff)
- oder Thread explizit pausiert wurde (z.B. durch `wait()` oder `sleep()`).
- Um wieder in den Zustand Runnable zurückzukehren, muss er entweder aufwachen (z. B. durch ein Timeout von `sleep()` oder `notify()` bei `wait()`).

5. Terminated/Dead (Beendet)

- die `run()`-Methode ist beendet oder thread wurde explizit beendet
- z. B. durch `interrupt()` in einigen Programmiersprachen.
- Der Thread ist nicht mehr ausführbar.

Thread erstellen.

① Runnable Interface

```
Class Newthread implements Runnable {  
    // Konstruktor  
    public Newthread(String name){...}  
    @Override  
    public void run(){  
        ....  
    }  
}
```

- run() Methode überschreiben : Wie implementiert der Thread konkret.

Initialisierung

```
public static void main (String args[]) {  
    Newthread nt1 = new Newthread("thread1");  
    Thread t1 = new Thread ( nt1 );  
}
```

② Thread abstrakte Klasse

```
public class Greeter extends Thread {  
    private final String name;  
  
    public Greeter(String name) {  
        this.name = name;  
    }  
  
    @Override  
    public void run() {  
        System.out.println("Hello from " + name);  
    }  
}
```

- run() Methode überschreiben : Wie implementiert der Thread konkret.

Initialisierung

```
public static void main (String args[]) {  
    Greeter t1 = new Greeter ("thread1");  
}
```

Vereinfachung :

Simple Threads kann man einfach über den Konstruktor erstellen

Thread hello = new Thread() -> System.out.println("Hello")

run() wird direkt überschrieben.

Threads

- Simple Threads kann man einfach über den Konstruktor erstellen

```
1 Thread hello = new Thread(() -> System.out.println("Hello"));
```


Threads

- Simple Threads kann man einfach über den Konstruktor erstellen

```
1 Thread hello = new Thread(() -> System.out.println("Hello"))
```

- Der Konstruktor erwartet eine Instanz der Klasse Runnable

```
1 @FunctionalInterface
2 public interface Runnable {
3     /**
4      * Runs this operation.
5      */
6     void run();
7 }
```

Threads

- Threads, die komplexere Logik erfordern oder einen internen Zustand verwalten müssen, sollten von der Klasse Thread erben und die Methode **run** überschreiben

```
1 public class Greeter extends Thread {  
2     private final String name;  
3  
4     public Greeter(String name) {  
5         this.name = name;  
6     }  
7  
8     @Override  
9     public void run() {  
10        System.out.println("Hello from " + name);  
11    }  
12 }
```

Threads

- Threads, die komplexere Logik erfordern oder einen internen Zustand verwalten müssen, sollten von der Klasse `Thread` erben und die Methode `run` überschreiben
- Alternativ kann man auch das Interface `Runnable` implementieren, falls Vererbung nicht möglich ist

```
1 public class Greeter extends Thread {  
2     private final String name;  
3  
4     public Greeter(String name) {  
5         this.name = name;  
6     }  
7  
8     @Override  
9     public void run() {  
10         System.out.println("Hello from " + name);  
11     }  
12 }
```

Threads: start()

- Erstellte Threads können wir über **start** starten
- Die JVM führt die **run** Methode des Threads bzw. der Runnable in einem neuen Thread parallel aus

```
1 Thread counter = new Thread(() -> {  
2     for (int i = 0; i < 100_000; i++) {  
3         System.out.println(i);  
4     }  
5 });  
6  
7 counter.start();
```


Threads: join()

- Mit der Methode **join** kann man warten, bis ein Thread fertig ist
- Threads können während ihrer Ausführung unterbrochen werden, was eine **InterruptedException** zur Folge hat

```
1 Thread counter = new Thread(() -> {  
2     for (int i = 0; i < 100_000; i++) {  
3         System.out.println(i);  
4     }  
5 });  
6  
7 counter.start();  
8  
9 counter.join();
```

Threads: join()

- Mit der Methode **join** kann man warten, bis ein Thread fertig ist
- Threads können während ihrer Ausführung unterbrochen werden, was eine **InterruptedException** zur Folge hat

```
1 Thread counter = new Thread(() -> {
2     for (int i = 0; i < 100_000; i++) {
3         System.out.println(i);
4     }
5 });
6
7 counter.start();
8
9 counter.join();  Unhandled exception: java.lang.InterruptedException
```

Threads: join()

- Mit der Methode **join** kann man warten, bis ein Thread fertig ist
- Threads können während ihrer Ausführung unterbrochen werden, was eine **InterruptedException** zur Folge hat

```
1 Thread counter = new Thread(() -> {
2     for (int i = 0; i < 100_000; i++) {
3         System.out.println(i);
4     }
5 });
6
7 counter.start();
8
9 try {
10     counter.join();
11 } catch (InterruptedException ex) {
12     throw new RuntimeException(ex);
13 }
```

Threads: interrupt()

- Threads können manuell von außen unterbrochen werden

```
1 Thread anotherCounter = new Thread(() -> {  
2     for (int i = 0; i < 100_000; i++) {  
3         System.out.println(i);  
4     }  
5 });  
6  
7 anotherCounter.start();  
8  
9 anotherCounter.interrupt();
```


Threads: sleep()

- Über **sleep** kann man Threads für eine bestimmte Zeit nichts tun lassen

```
1 Thread sleeper = new Thread(() -> {  
2     try {  
3         Thread.sleep(1000);  
4     } catch (InterruptedException ex) {  
5         System.out.println("I was sleeping and got interrupted");  
6     }  
7 });
```

Threads: `currentThread()`

- Mit `currentThread` kommt man an den derzeitigen Thread
- Threads heißen normalerweise “Thread-x”, wobei x die fortlaufende numerische ID von Threads ist
- Der Main-Thread der immer implizit gestartet wird, heißt “main”

```
1 Thread myNameIs = new Thread(() -> {  
2     String name = Thread.currentThread().getName();  
3     System.out.println("My name is " + name);  
4 });  
5  
6 myNameIs.start();  
7 myNameIs.join();  
8  
9 System.out.println(Thread.currentThread().getName());
```

Beispiel-Ausgabe:

```
1 Thread-4  
2 main
```

Threads

Bearbeite nun die Aufgabe W11P01 - Parallele Summierung!

Synchronisation

```
1 private class Counter extends Thread {  
2     private static int count = 0;  
3  
4     @Override  
5     public void run() {  
6         for (int i = 0; i < 1000; i++)  
7             count++;  
8     }  
9 }  
10  
11 public int getCount() {  
12     return count;  
13 }  
14 }
```

```
1 var c1 = new Counter();  
2 var c2 = new Counter();  
3  
4 c1.start();  
5 c2.start();  
6  
7 c1.join();  
8 c2.join();  
9  
10 // Ausgabe ...?  
11 println(counter1.getCount());
```

Synchronisation

```
1 private class Counter extends Thread {  
2     private static int count = 0;  
3  
4     @Override  
5     public void run() {  
6         for (int i = 0; i < 1000; i++)  
7             count++;  
8     }  
9 }  
10  
11 public int getCount() {  
12     return count;  
13 }  
14 }
```

```
1 var c1 = new Counter();  
2 var c2 = new Counter();  
3  
4 c1.start();  
5 c2.start();  
6  
7 c1.join();  
8 c2.join();  
9  
10 // Ausgabe ...?  
11 println(counter1.getCount());
```

Beispiel-Ausgabe:

```
1 1789
```

Synchronisation

- Race Condition: Unvorhersehbares Verhalten durch gleichzeitigen Zugriff mehrerer Threads auf gemeinsame Ressourcen
→ **Wir müssen sicherstellen, dass mehrere Threads nicht gleichzeitig die gleiche Ressource benutzen!**

Synchronisation Wegs

① synchronized keyword:

- in Methodeskopf.

(Kapselung) synchronized (return type) name() { ... }

- in Methodeskörper:

```
synchronized (lockObjekt) {  
    :  
}
```

lockObjekt: Objekt von Synchronisation:
gemeinsam Resource oder
this

- ein Thread tritt in Methode/Block ein

↓

andere Threads blockiert

↓ bis

der aktuelle Thread verlässt den Methode/Block.

- synchronized in Kopf \equiv synchronized (this) { ... }

- gültig für statische Methode:

static synchronized (return type) name() { ... } \equiv synchronized (Classname.class) { ... }

Synchronisation

- **synchronized** verhindert parallelen Zugriff auf das gleiche Objekt im kritischen Abschnitt
- **lock** ist ein Monitor, der dafür sorgt, dass nur ein Thread auf den count zugreifen kann

```
1 public class SynchronizedCounter extends Thread {  
2     private static int count = 0;  
3     private static final Object lock = new Object();  
4  
5     @Override  
6     public void run() {  
7         for (int i = 0; i < 1000; i++) {  
8             synchronized (lock) {  
9                 count++;  
10            }  
11        }  
12    }  
13 }
```


Synchronisation

- **synchronized** verhindert parallelen Zugriff auf das gleiche Objekt im kritischen Abschnitt
- **lock** ist ein Monitor, der dafür sorgt, dass nur ein Thread auf den count zugreifen kann

```
1 public class SynchronizedCounter extends Thread {  
2     private static int count = 0;  
3     private static final Object lock = new Object();  
4  
5     @Override  
6     public void run() {  
7         for (int i = 0; i < 1000; i++) {  
8             synchronized (lock) {  
9                 count++;  
10            }  
11        }  
12    }  
13 }
```

Ausgabe:

```
1 2000 🎉
```

Synchronisation

- Falls count ein Object ist, können wir auch gleich darauf synchronisieren und brauchen kein weiteres Object
- Siehe W11P00 für den restlichen Code

```
1 public class SynchronizedCounter2 extends Thread {  
2     private static final Count count = new Count();  
3  
4     @Override  
5     public void run() {  
6         for (int i = 0; i < 1000; i++) {  
7             synchronized (count) {  
8                 count.increment();  
9             }  
10        }  
11    }  
12 }
```

Ausgabe:

```
1 2000 🎉
```

Synchronisation

- Methoden können auch **synchronized** sein

```
1 public synchronized void someMethod() {  
2     // ... code ...  
3 }
```

- ...das ist dann äquivalent zu einem inneren **synchronized** Block mit **this**

```
1 public void someMethod() {  
2     synchronized (this) {  
3         // ... code ...  
4     }  
5 }
```

Synchronisation

- Static Methoden können auch **synchronized** sein

```
1 public static synchronized void someMethod() {  
2     // ... code ...  
3 }
```

- ...das ist dann äquivalent zu einem inneren **synchronized** Block mit der Klasse!

```
1 public static void someMethod() {  
2     synchronized (SomeClass.class) {  
3         // ... code ...  
4     }  
5 }
```

- Welches Problem könnte hier auftreten?

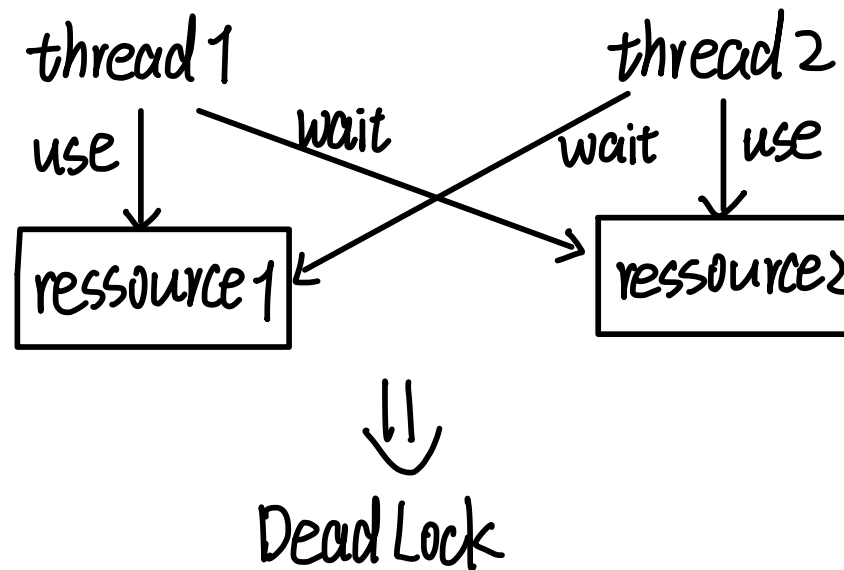
Synchronisation

- Öffnet die Klasse **Deadlock** in der Aufgabe W11P00 - Threads Playground
- Was ist die Ausgabe der main Methode?

Deadlock

Situation, in der zwei oder mehr Threads dauerhaft blockiert.

Grund: jeder auf eine Ressource wartet, die von einem anderen Thread gehalten wird, wodurch keine Fortschritte möglich sind.



Synchronisation

- Öffnet die Klasse **Deadlock** in der Aufgabe W11P00 - Threads Playground
- Was ist die Ausgabe der main Methode?

Beispiel-Ausgabe:

```
1 Thread 1: Locked Resource1  
2 Thread 2: Locked Resource2
```

- ...aber die Methode terminiert nicht, wieso?

Synchronisation

- Deadlock: Threads blockieren sich gegenseitig durch zyklisches Warten auf Ressourcen
→ **Wir dürfen nicht einfach wahllos irgendwie synchronisieren!**

Threads: So viele Probleme...

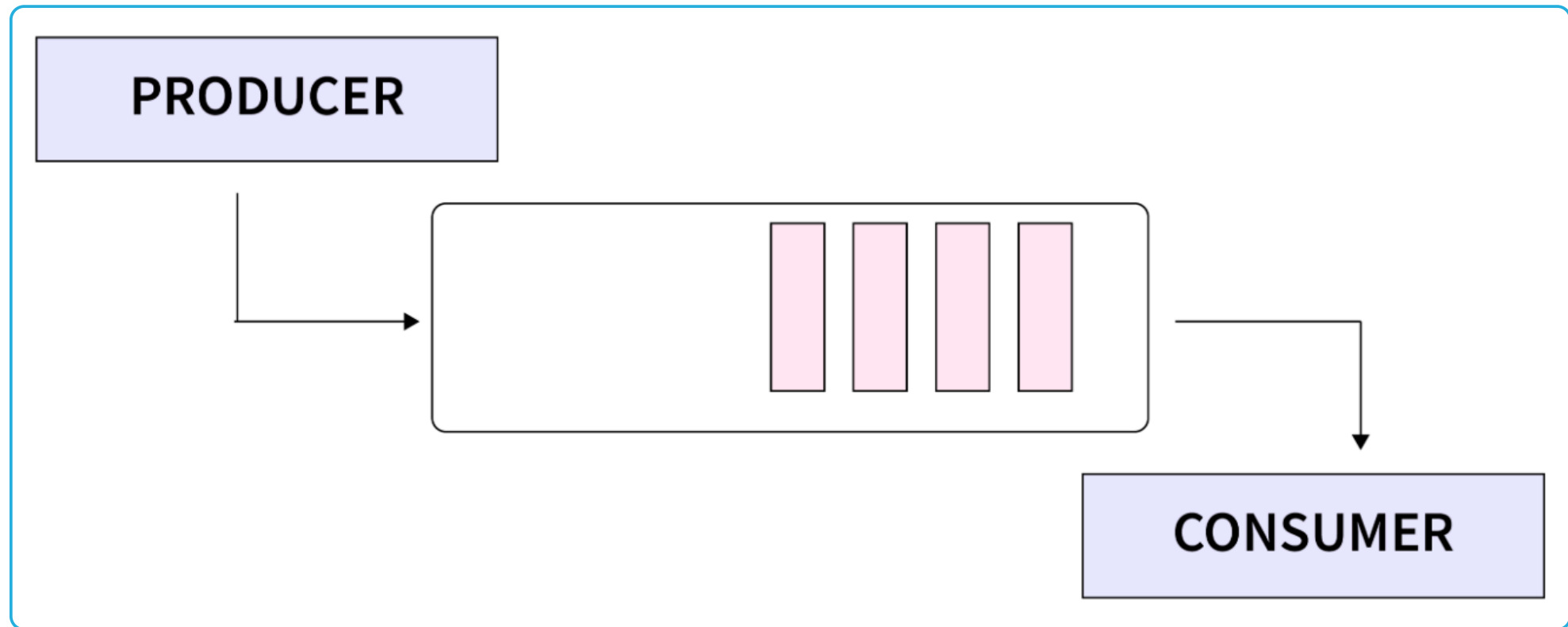
- Race Condition: Unvorhersehbares Verhalten durch gleichzeitigen Zugriff mehrerer Threads auf gemeinsame Ressourcen
→ **Wir müssen sicherstellen, dass mehrere Threads nicht gleichzeitig die gleiche Ressource benutzen!**
- Deadlock: Threads blockieren sich gegenseitig durch zyklisches Warten auf Ressourcen
→ **Wir dürfen nicht einfach wahllos irgendwie synchronisieren!**
- Und viele, viele mehr... → Mehr dazu in fortführenden Modulen!

Threads

Bearbeite nun die Aufgabe W11P02 - Geschäftspartner!

Producer-Consumer Problem

- Klassisches und häufiges Problem in der parallelen Programmierung
- 1-N Produzenten und 1-M Konsumenten teilen sich einen Buffer als gemeinsame Ressource



Synchronisation mit `wait()` und `notify()`

- Öffnet die Klasse **ProducerConsumerBuffer** in der Aufgabe W11P00 - Threads Playground, wie wird das Problem dort gelöst?
- Über **wait** wartet der Thread auf die gemeinsame Ressource bis es informiert wird, dass es weitermachen kann
- Über **notify** wird dem wartenden Threads bescheid gegeben aufzuwachen
- Über **notifyAll** könnte man allen wartenden Threads bescheid geben aufzuwachen

Threads

Bearbeite nun die Aufgabe W11P03 - Klausurkorrektur!

Klasse Semaphore

- begrenzt die Anzahl der Threads, die gleichzeitig auf eine Ressource
- funktioniert wie ein Zähler für die verfügbaren Permits:
 - `acquire()` : ein Thread auf die Ressource zugreift, die Anzahl der verfügbaren Permits reduziert wird.
 - `release()` : ein Thread die Ressource freigibt, die Anzahl der verfügbaren Permits erhöht wird.

Synchronisationsmechanismen mit der Java Standardbibliothek

- Man braucht für verschiedene Use-Cases verschiedene Synchronisationsmechanismen
- Die Java Standardbibliothek bietet dabei schon eine Menge an...
 - `ReentrantLock`
 - `Condition`
 - `Semaphore`
 - ...Und viele mehr.
- Könnten diese Woche bereits hilfreich sein, aber mehr dazu nächste Woche! :)