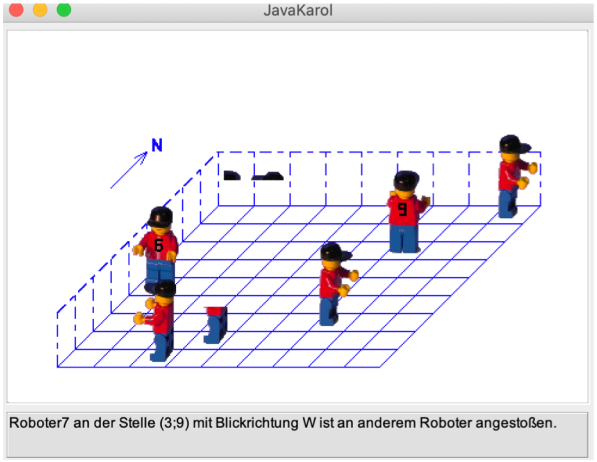
# 2. Kommunikation und Synchronisation von Prozessen

## 2.4 Synchronisation von Prozessen

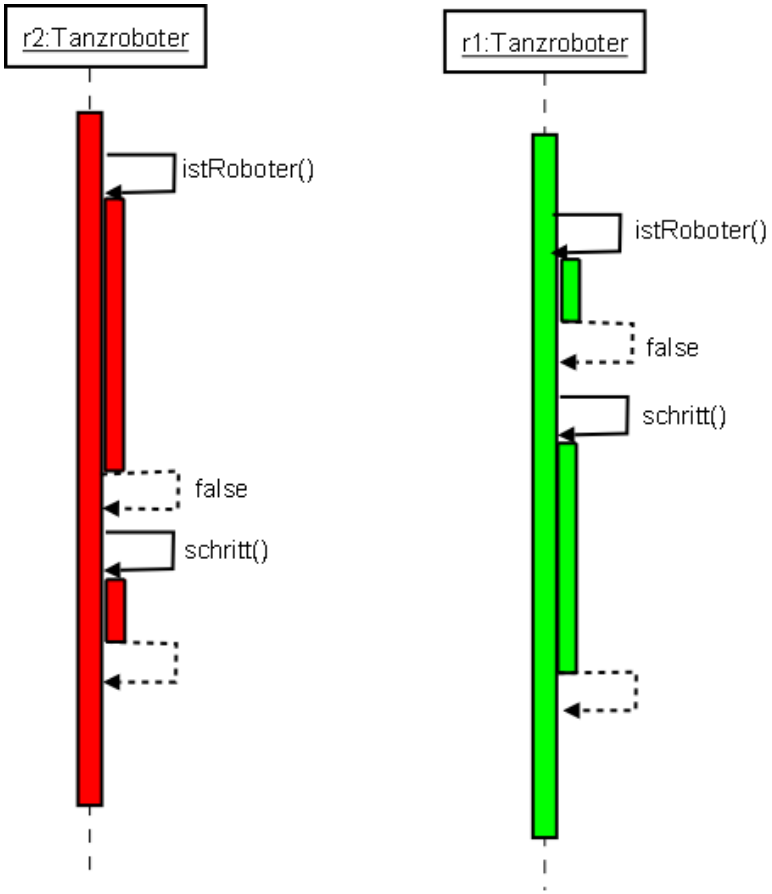
Die Tanzroboter aus dem letzten Kapitel können zusammenstoßen, obwohl sie vor einem Schritt prüfen, ob ein anderer Roboter vor ihnen steht.

Unter welchen Bedingungen kann dies geschehen?



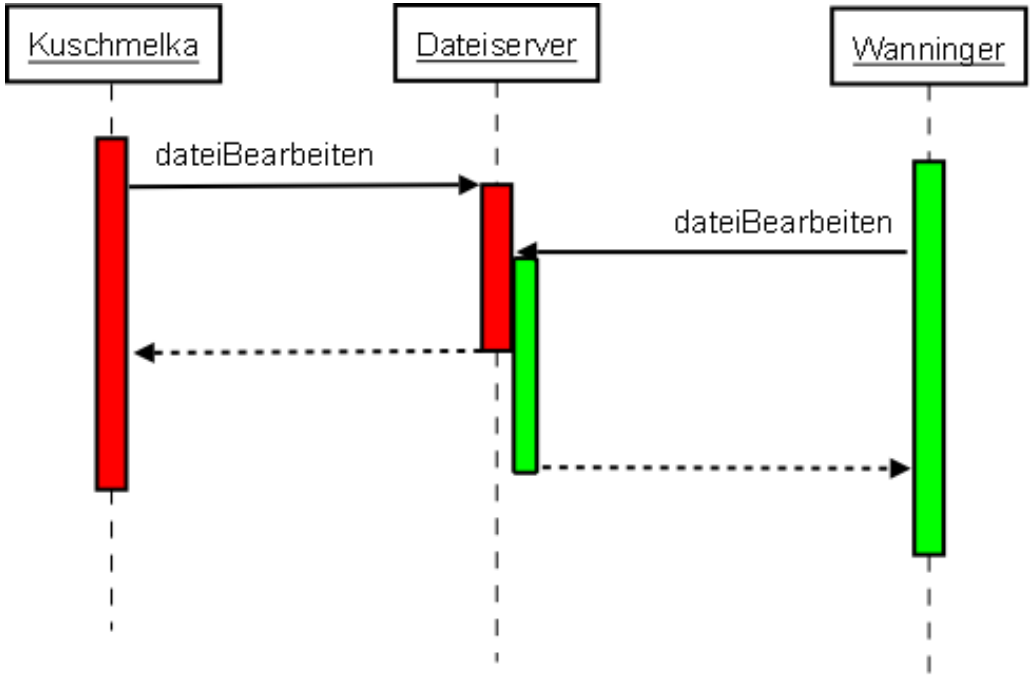

```
Exception in thread "Thread-7" java.lang.RuntimeException: Roboterbewegung nicht moeglich.
        at javakarol.Roboter.Schritt(Roboter.java:257)
        at ROBOTER.Schritt(ROBOTER.java:69)
        at Tanzroboter.vorsichtigerRechtsSchritt(Tanzroboter.java:53)
        at Tanzroboter.tanzen(Tanzroboter.java:71)
        at Roboterthread.run(Roboterthread.java:21)
```

Roboter r1 testet, ob vor ihm niemand steht.
Ein zweiter Roboter r2 läuft aber auf diesen Platz, bevor sich r1 selbst dorhin bewegen kann.



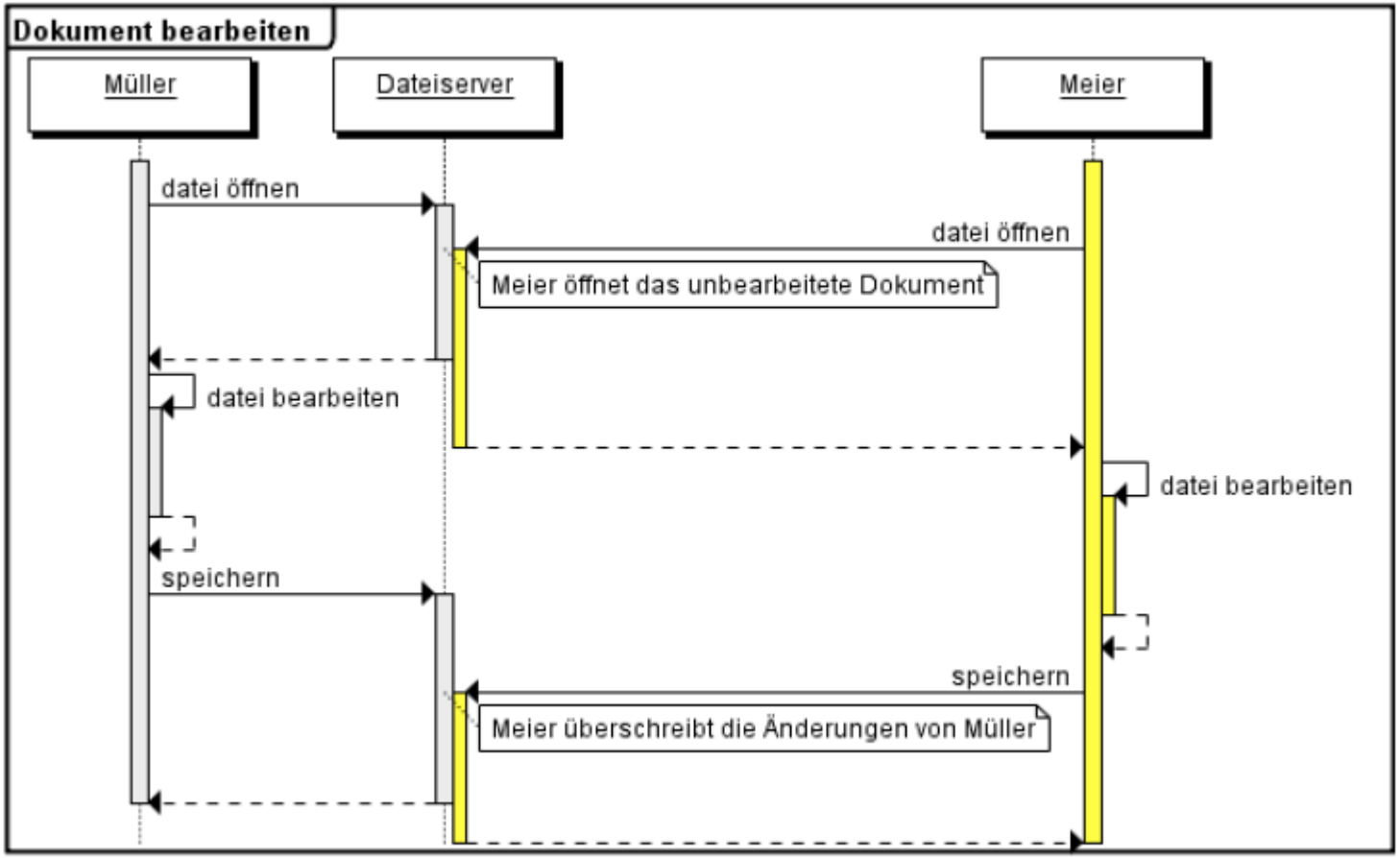
Greifen mehrere Benutzer gleichzeitig auf eine Datei zu und verändern sie, können sich die Prozesse gegenseitig beeinflussen. Man spricht in diesem Fall von **nebenläufigen Prozessen** .

Sequenzdiagramme sind ein geeignetes Hilfsmittel um die **kritischen Abschnitte** herauszufinden. Die Zugriffe müssen dann koordiniert **(synchronisiert)** werden.

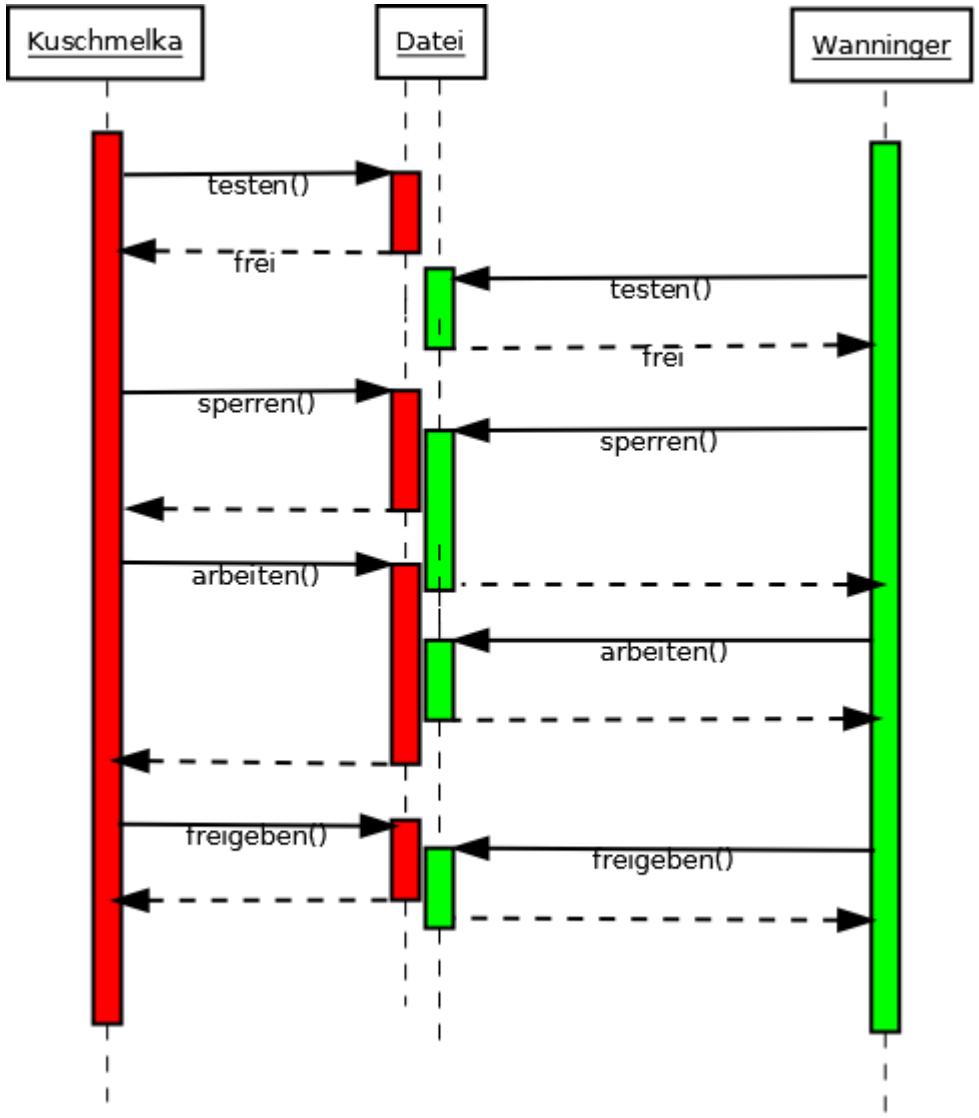



Wanninger überschreibt in diesem Beispiel die Änderungen von Kuschmelka

# Dataillierteres Sequenzdiagramm:

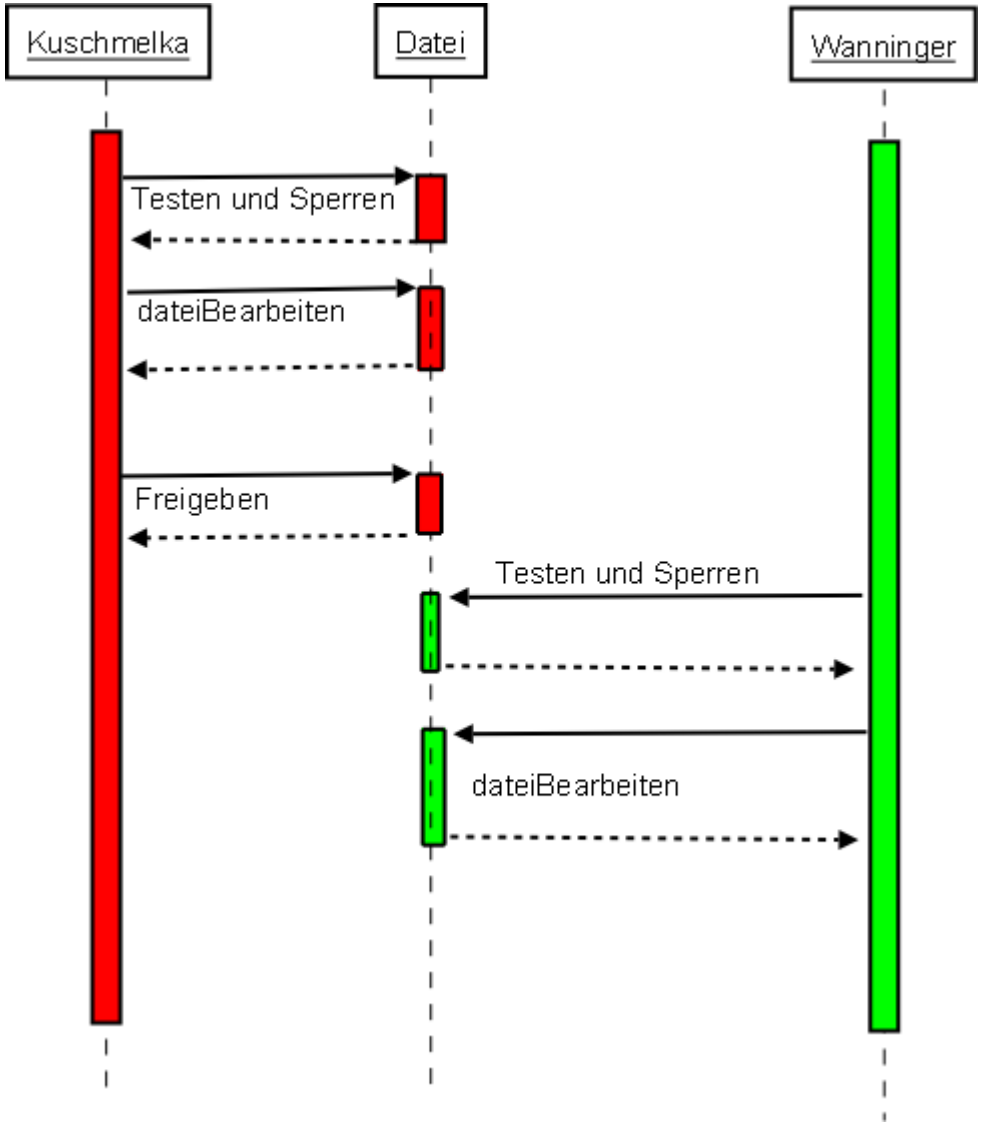


Ein Lösungsversuch könnte eine Art "Absprache" zwischen den Prozessen sein, indem man auf gemeinsame Attribute zugreift, die signalisieren, ob ein Objekt frei ist.
Das Problem ist dadurch aber nicht gelöst, da zwischen Testen und Sperren etwas Zeit vergeht, in der ein anderer Prozess  testet und die Nachricht "frei" erhält.

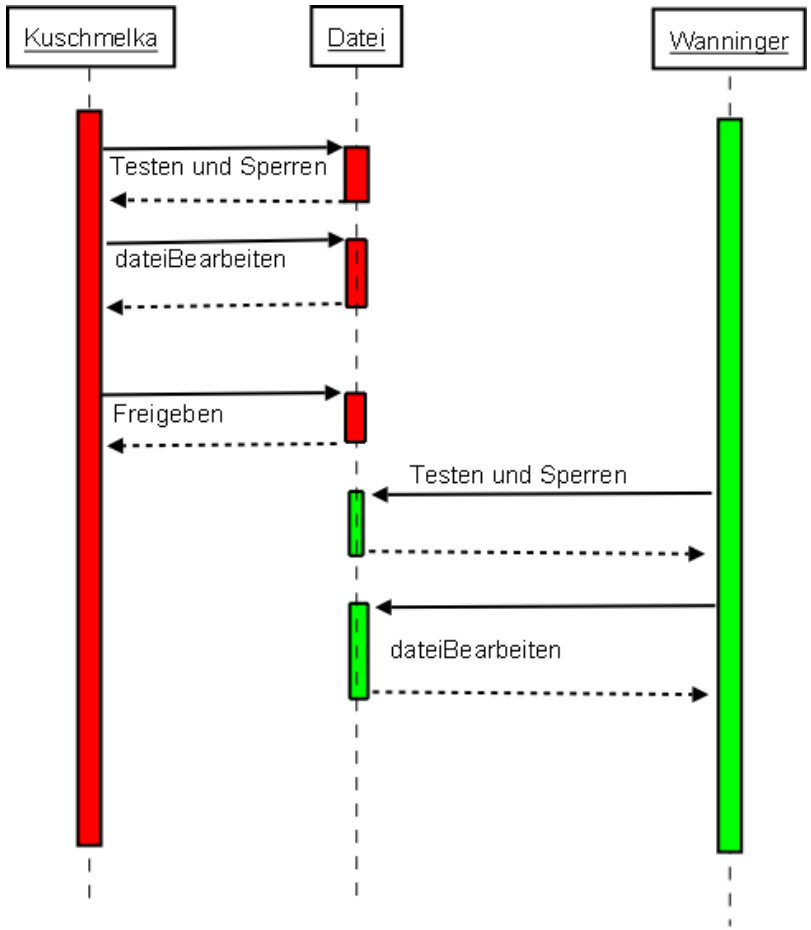


Das Synchronisationsproblem **kann nur auf Betriebssystemebene** mit Hilfe von **Semaphoren** gelöst werden.

Man benötigt dazu eine Methode, welche die Verfügbarkeit testet und gegebenenfalls sofort sperrt sowie eine Methode zum Freigeben.



Ein Konzept, dass intern Semaphore verwendet ist das **Monitorkonzept** .
Diejenigen Methoden eines Objekts, die immer nur von einem Prozess genutzt werden können, werden zu einem Monitor zusammengefasst.
In Java geschieht dies ganz einfach durch den Bezeichner **synchronized** im Methodenkopf.
Den Rest erledigt das Betriessystem.



## Übung

1.
Ändere mithilfe des Monitorkonzepts das Tanzroboterprogramm so ab, dass Zusammenstöße verhindert werden.

2.
Verdeutliche mit Hilfe passender Sequenzdiagramme die Nebenläufigkeit bei
a) gleichzeitigem Einzahlen auf ein Konto
b) gemeinsame Nutzung einer Engstelle von zwei S-Bahnen

# Lösungen

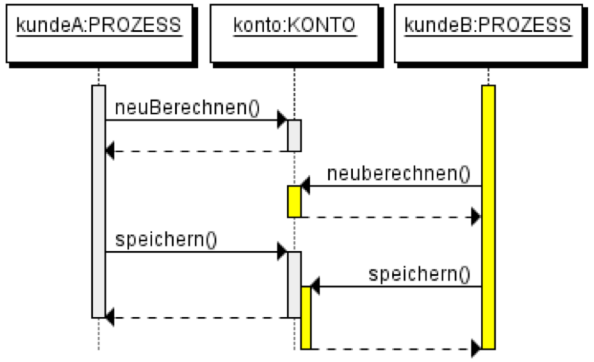
1.

```
public void run(){
    for (int i = 0; i < schritte; i++){
        absichern(taro);

        try{
            Thread.sleep(100);
        }
        catch (Exception e){
        }
    }
}

public synchronized static void absichern (Tanzroboter taro)
{
    taro. tanzen ();
}
```

# Lösungen

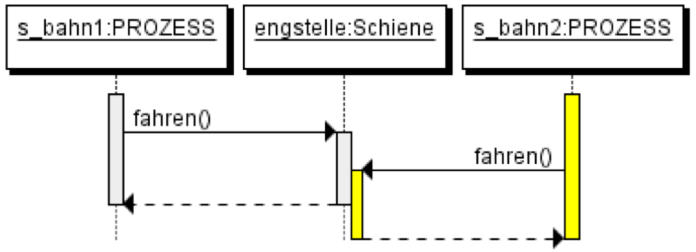
**Die Methode neuBerechnen() erhöht/verringert den Kontostand um einen bestimmten Betrag.
Dieser Betrag wird aber noch nicht abgespeichert.
kundeB rechnet in dem Beispiel mit einem falschen Kontostand.**

## 2.a

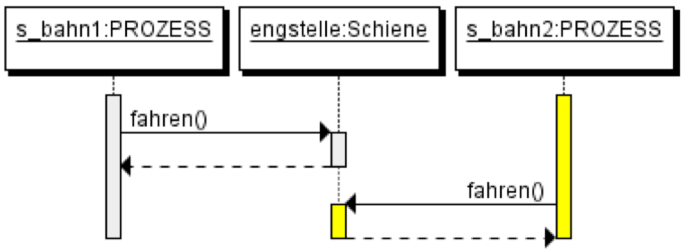



## 2.b

**Die Nebenläufigkeit kann eine Kollision erzeugen**




**Die Synchronisation verhindert eine Kollision.**



# Übung

3.
Untersuche den Quelltext des Programms p08_bahnsimulation auf kritische Abschnitte und ändere das Programm so ab, dass keine Doppelbuchungen mehr auftreten können.

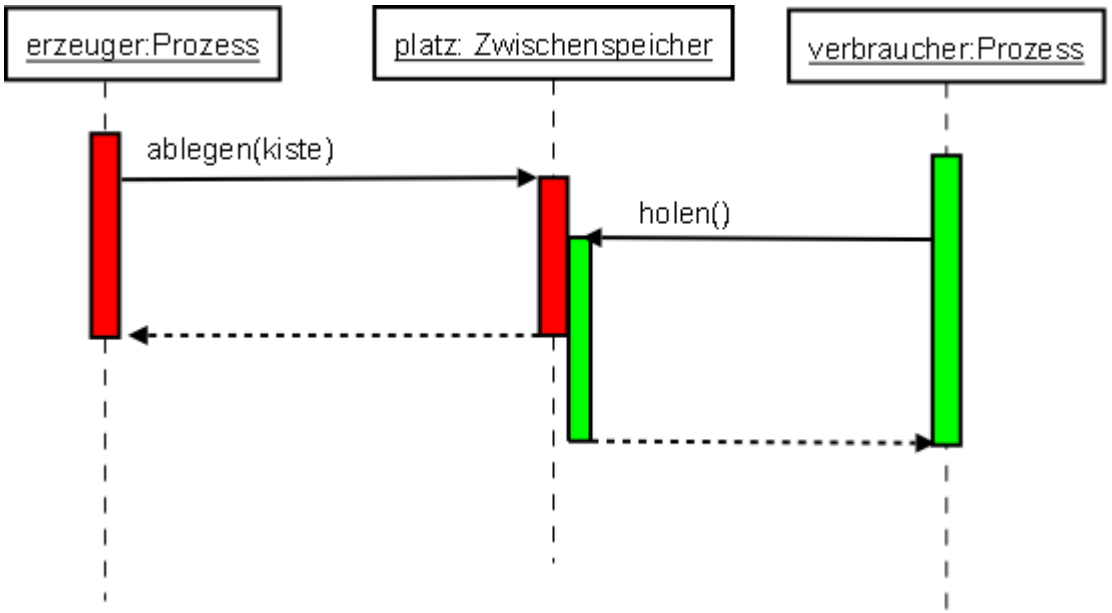
4.
Mache dich zunächst mit dem Server-Client-System p06_platzbuchung vertraut.
Greifen mehrere Clients auf den Server zu, kann es aufgrund der Nebenläufigkeit zu Überbuchungen kommen.
Teste dies mit dem Projekt p07_buchungssimulation.
Ändere das Programm so ab, dass keine Überbuchungen auftreten.

**Erweiterung des Monitorkonzeptes: Das Erzeuger-Verbraucher-Problem**

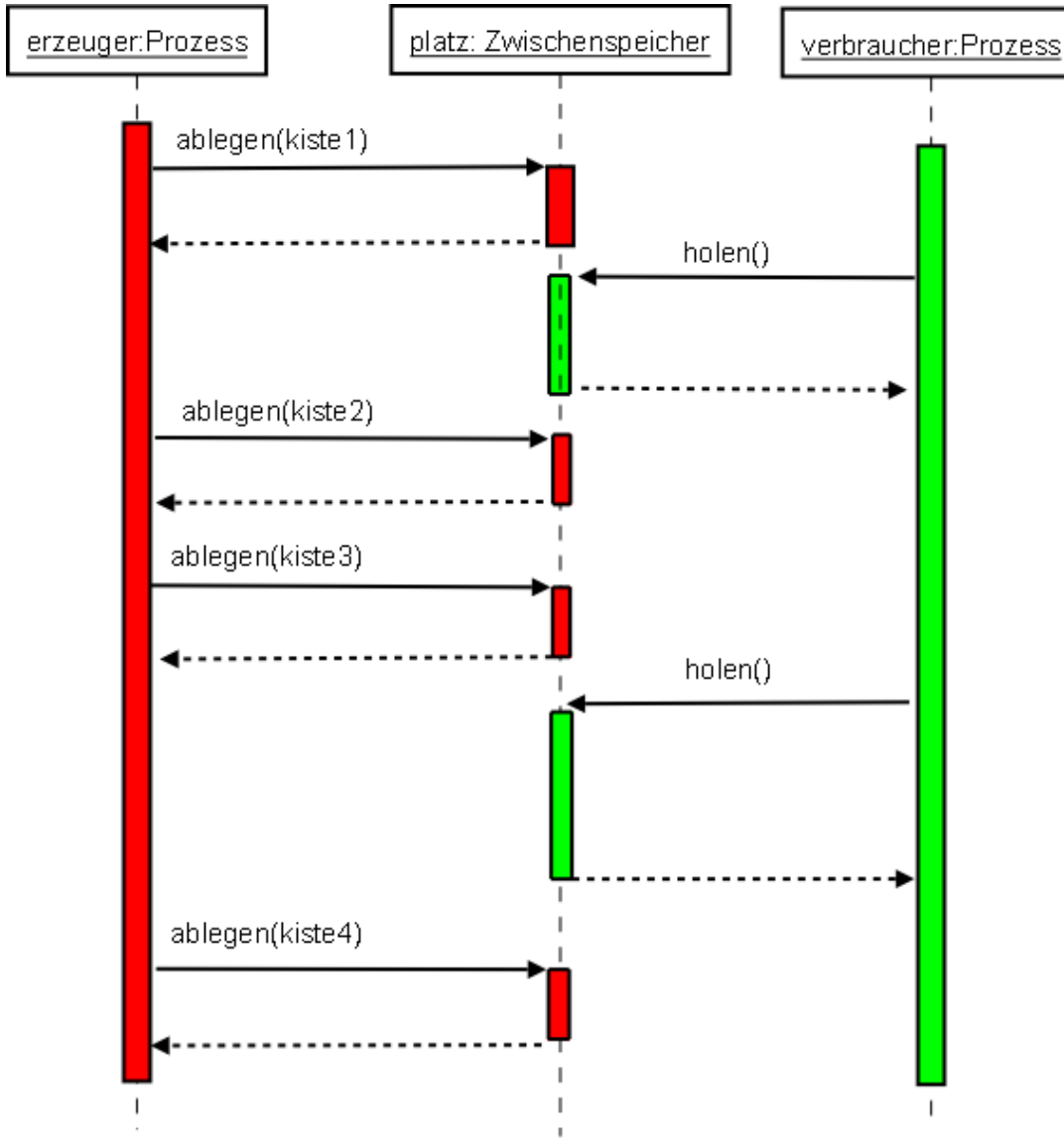
Ein Roboter (Erzeuger) füllt eine Kiste und legt sie auf einem Zwischenspeicherplatz mit begrenzter Kapazität ab.

Die Kiste wird dann von einem zweiten Roboter (Verbraucher) abgeholt.

Auf den Zwischenspeicherplatz darf immer nur ein Roboter zugreifen.



Die Methoden ablegen und holen werden
(in JAVA durch den Modifikator **synchronized** ) als Monitor erklärt.



Probleme können sich ergeben, wenn der Zwischenspeicher begrenzt ist und Erzeuger und Verbraucher mit unterschiedlichen Geschwindigkeiten arbeiten:

Ist der Erzeuger schneller als der Verbraucher, kann es im Speicher zu einer Überlaufsituation kommen, sodass der Erzeuger seine Arbeit unterbrechen muss.

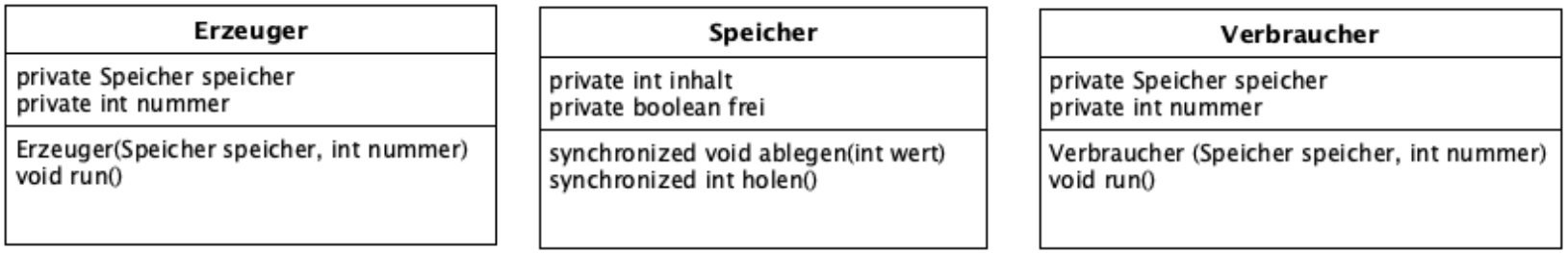
Ist der Verbraucher schneller, kann der Speicher vollständig geleert sein, so dass der Verbraucher eine Wartephase einlegen muss.

Die Prozesse arbeiten effizienter zusammen, wenn der Verbraucher dem Erzeuger eine Nachricht sendet, wenn er ein Objekt aus dem Speicher abgeholt hat.
Dafür gibt es in Java die Methoden **wait ( )** und **notify ( )** .

Mit wait() wird ein Thread blockiert und mit notify() wird er aufgeweckt, sodass für ein Objekt zu einer bestimmten Zeit immer nur ein einziger Thread die zum Monitor gehörenden Methoden ausführen kann.

Die anderen Threads müssen warten und werden in einer Liste verwaltet.

# Klassendiagramm für eine Simulation (Erzeuger und Verbraucher erben von der Klasse Thread)



# Version ohne wait und notify

## Klasse Speicher

| synchronized ablegen (int wert) |
| --- |
| wenn Speicher frei<br>  inhalt = wert<br>  Speicher sperren<br>Ende wenn |

| synchronized int holen () |
| --- |
| wenn Speicher besetzt<br>  Speicher freigeben<br>  return inhalt<br>Sonst<br>  return -1<br>Ende wenn |

# Version ohne wait und notify

## Die Methode run in ERZEUGER und VERBRAUCHER

**ERZEUGER**

wiederhole von i = 1 bis 1000

- Methode ablegen von speicher mit Zählvariable als Übergabeparameter aufrufen
- Textausgabe „Erzeuger1(Nr) abgelegt: Zählvariable“
- Thread schlafen legen (Zufallszahl) (mit try und catch absichern)

**VERBRAUCHER**

int wert deklarieren und 1 setzen
wiederhole von i = 1 bis 1000

- Methode holen von speicher aufrufen und den Rückgabewert an wert übergeben.
- Textausgabe „Verbraucher1(Nr) geholt: wert“
- Thread schlafen legen (Zufallszahl) (mit try und catch absichern)

# Version mit wait und notify

## Klasse Speicher

```
synchronized ablegen (int wert)
solange Speicher nicht frei
        wait
        (mit try und catch absichern)
Ende solange

inhalt = wert
Speicher sperren
notifyAll()
```

```
synchronized int holen ()
solange Speicher frei
        wait
        (mit try und catch absichern)
Ende solange

Speicher freigeben
notifyAll()
return inhalt
```

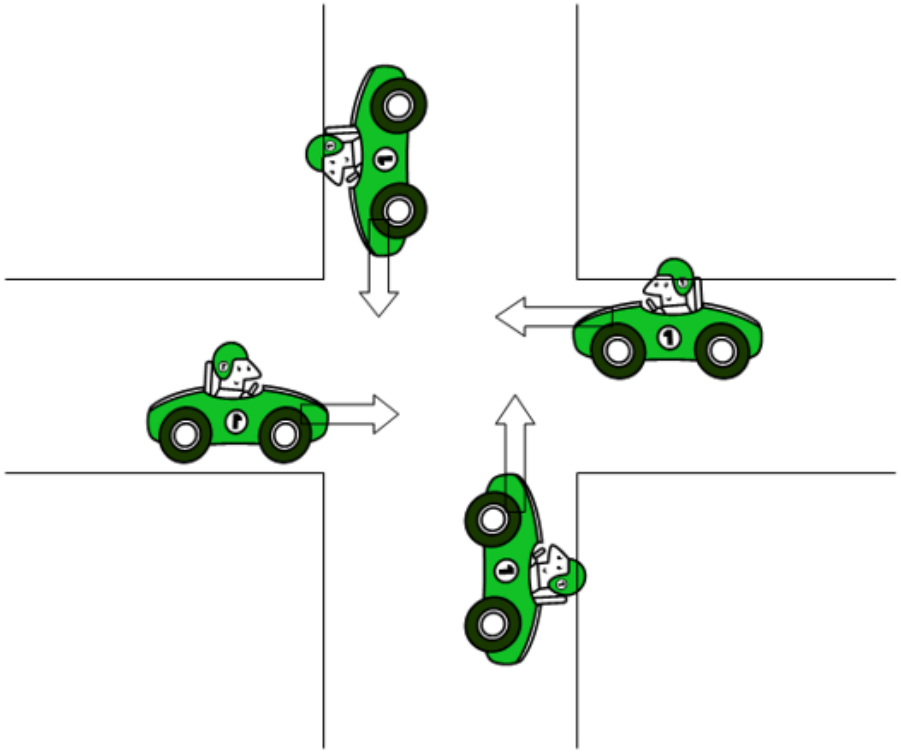
## Deadlock (Verklemmung )

Beim gemeinsamen Zugriff auf Ressourcen kann es zu einem sogenannten Deadlock kommen, wenn alle Prozesse sich gegenseitig sperren, weil sie auf die Freigabe von Ressourcen warten, die von den anderen wartenden Prozessen benutzt werden.

Beispiel

# Deadlock (Verklemmung )

Vier Autos wollen eine Kreuzung ohne Vorfahrtsschilder überqueren und jeder wartet darauf, dass ein anderer ein Signal zum Weiterfahren gibt.



Damit es zu einem Deadlock kommt, müssen folgende Bedingungen erfüllt sein:

## 1. Wechselseitiger Ausschluss

Jede Ressource (man sagt auch Betriebsmittel)wird entweder von genau einem Prozess benutzt oder sie ist verfügbar.

## 2. Anforderung weiterer Ressourcen

Ein Prozess, der bereits eine Ressource nutzt, kann eine weitere anfordern.

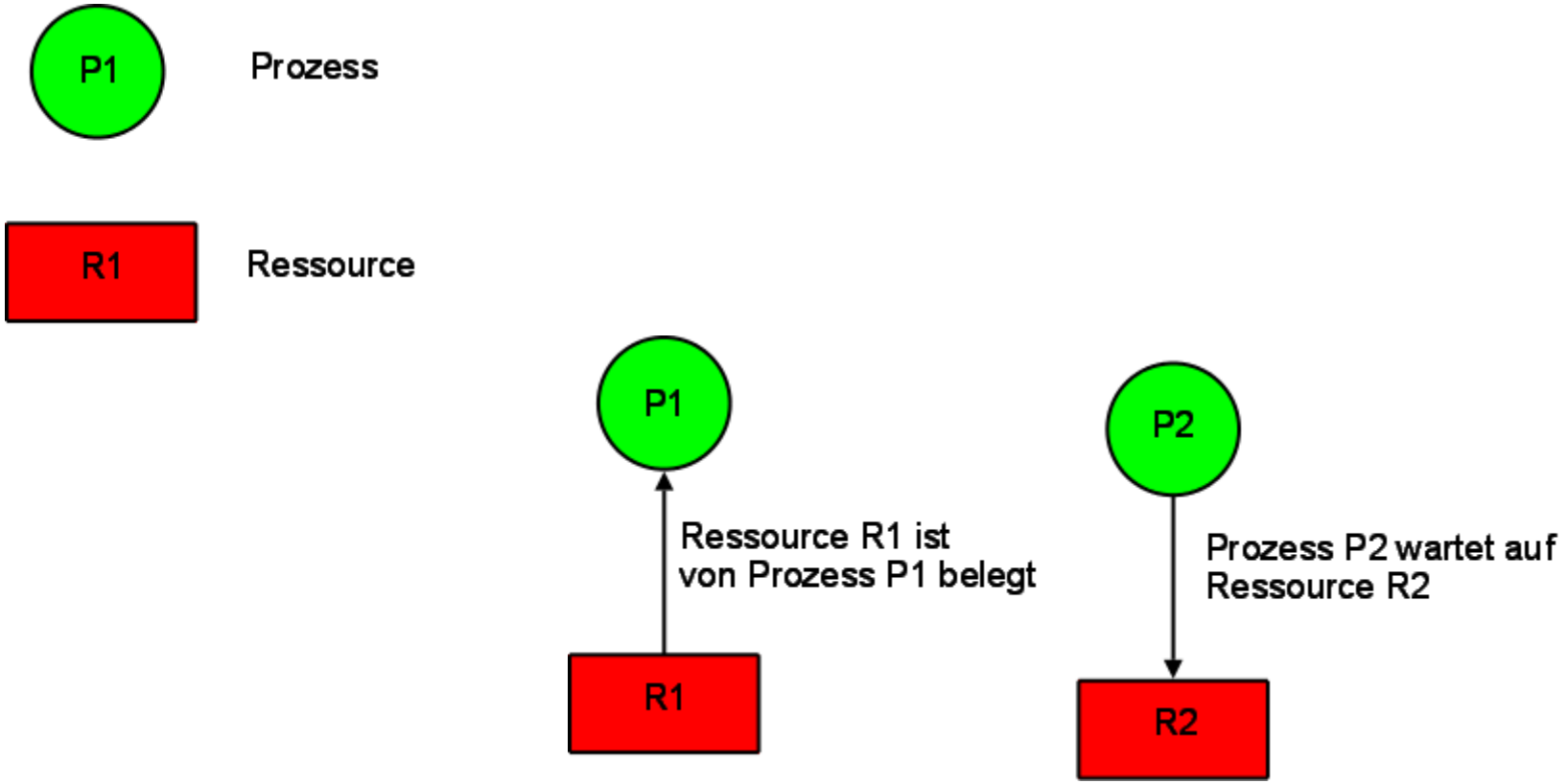
## 3. Ununterbrechbarkeit

Die Nutzung der Ressource kann dem Prozess nicht von außen entzogen werden. Er muss sie selbst freigeben.

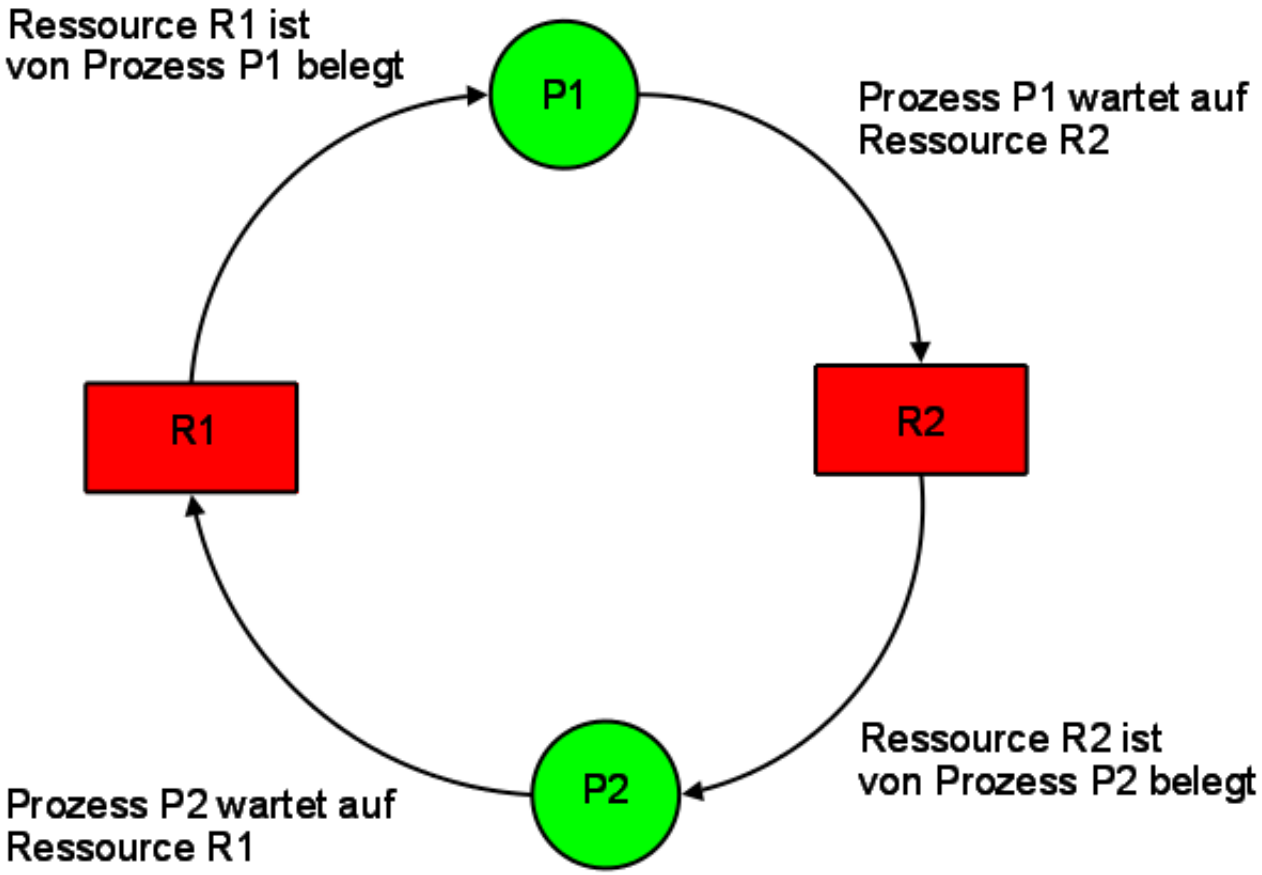
## 4. Zyklische Wartebedingung

Es gibt eine geschlossene Kette (Zyklus) von Prozessen, sodass jeder Prozess eine Ressource anfordert, die vom folgenden Prozess belegt ist.

# Deadlocks kann man sehr gut mit Hilfe sogenannter Betriebsmittelgraphen darstellen:

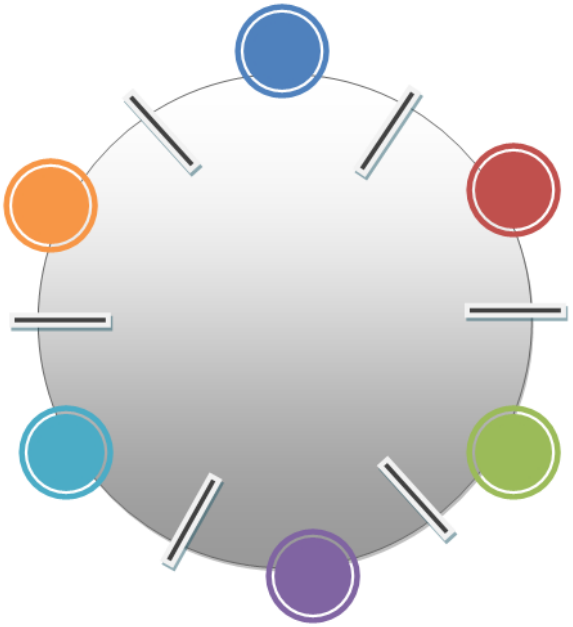


# Deadlocks kann man sehr gut mit Hilfe sogenannter Betriebsmittelgraphen darstellen:



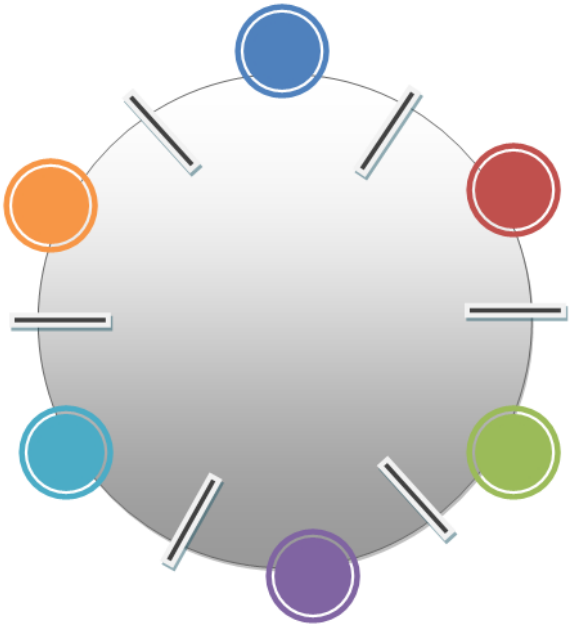
# Das Philosophenproblem

Eine bestimmte Anzahl Philosophen sitzt um einen runden Tisch.
Jeder Philosoph denkt entweder nach oder er hat Hunger und isst.
Zum Essen benötigt er zwei Stäbchen, die er nacheinander in beliebiger Reihenfolge nimmt.
Jeder Philosoph hat einen eigenen Teller und zwischen zwei Philosophen liegt genau ein Stäbchen.



# Das Philosophenproblem

Hat jeder Philosoph das linke Stäbchen genommen, ist ein Deadlock entstanden.
Jeder Philosoph wartet darauf, dass sein rechter Nachbar sein Stäbchen freigibt.



Übung:

Simulation: **p09_SpeisendePhilosophen**
Anspruchsvolle und detaillierte Simulation (Deadlock, Erzeuger und Verbraucher):
**semVisu.jar**

Abituraufgaben:
**Aufgaben_Synchronisation_von_Prozessen.pdf**