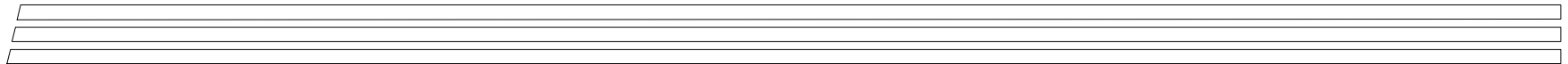
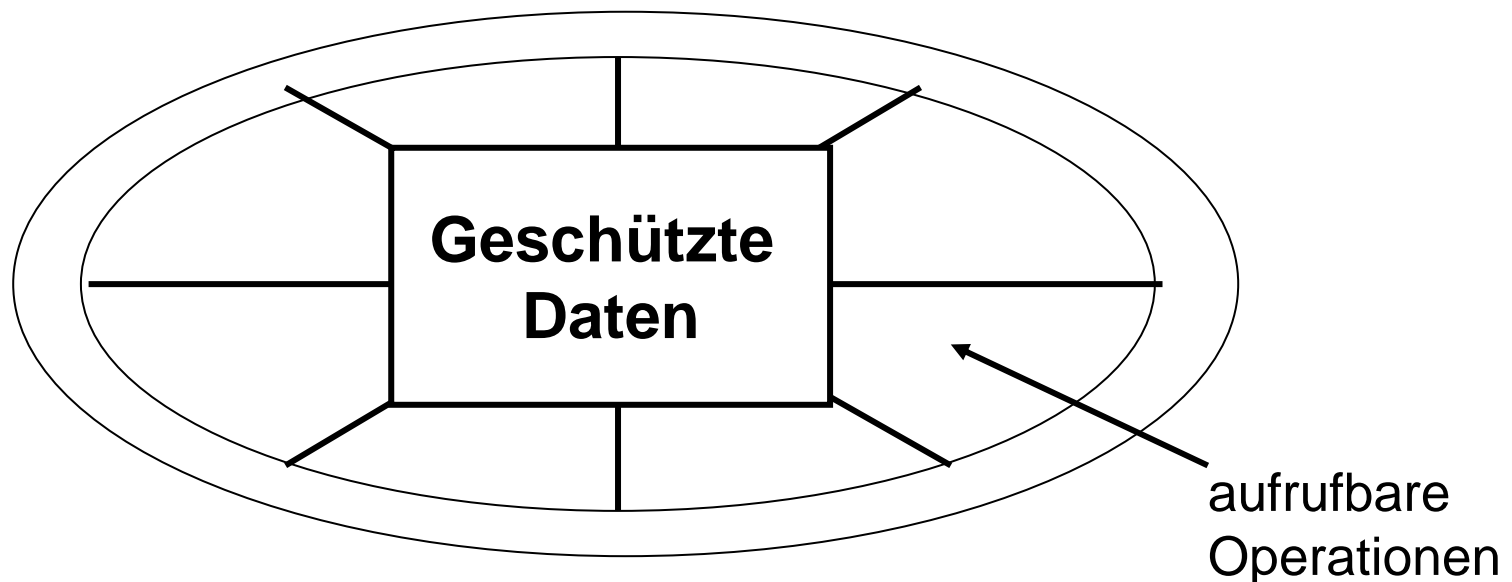


# *Das Monitorkonzept*

*(nach Hoare/Brinch-Hansen 1974)*

Nur ein Prozess bzw. Thread kann zu einem bestimmten Zeitpunkt im Monitor aktiv sein

=> gegenseitiger Ausschluss, mutual exclusion.



## *Beispiel: Zählermonitor (Pseudosyntax)*

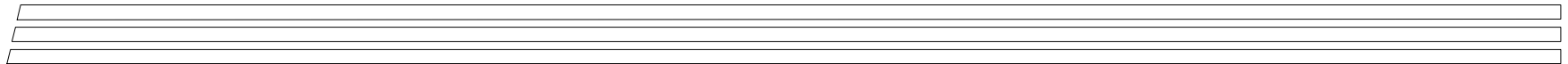
```
monitor counter {  
    int count = 0  
    procedure reset ( )      { count = 0 }  
    procedure increment ( ) { count += 1 }  
    procedure decrement ( ) { count -= 1 }  
}
```

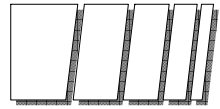
**Implementierung mittels mutex-Semaphor:**

**Jeder Prozedurrumpf wird in**

**P(mutex) \_ V(mutex)**

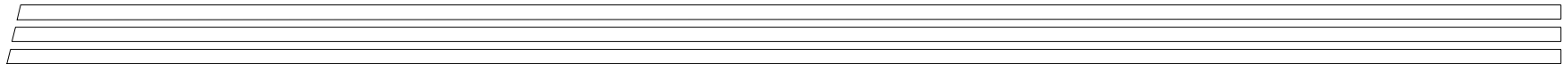
**eingeschlossen.**





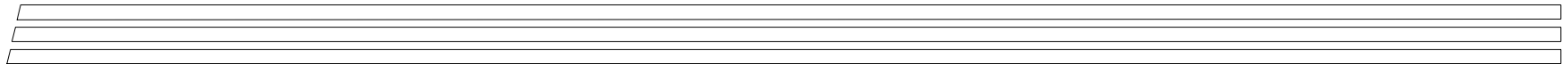
## *Einseitige Synchronisation in Monitoren*

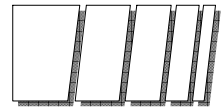
- Für einseitige (bedingte) Synchronisationen innerhalb von Monitoren ist eine Erweiterung des Basiskonzeptes nötig.
- Bedingungsvariablen (condition variables) sind Monitor-interne Synchronisationsvariablen mit den Operationen
  - wait (cond) „Warte auf Erfülltsein von Bedingung cond“  
Der ausführende Prozess wird suspendiert und in eine Warteschlange für cond eingereiht.  
Der Monitor wird freigegeben.
  - signal(cond) „Signalisiere, dass cond gilt.“  
Der ausführende Prozess reaktiviert den „ältesten“ Prozess in der Warteschlange zu cond.  
Es gibt verschiedene Signalisierungsmethoden, die festlegen, welcher Prozess nach einem signal im Monitor aktiv ist.



# *Signalisierungsmethoden*

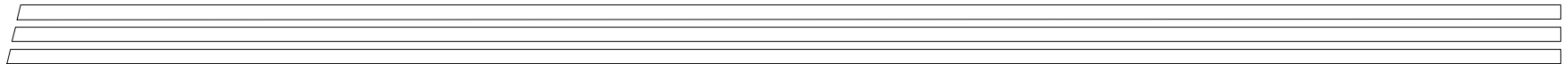
- signal and continue (SC)      -> Java  
Der signalisierende Prozess bleibt aktiv.  
Der reaktivierte Prozess muss sich neu um den Monitorzugang bewerben.  
Die Gültigkeit der signalisierten Bedingung muss erneut geprüft werden.
- signal and exit (SX)              -> Concurrent Pascal  
Ein signal ist nur am Ende von Monitorprozeduren erlaubt.  
Der reaktivierte Prozess erhält sofort Zugang zum Monitor.  
Die Gültigkeit der signalisierten Bedingung ist garantiert.
- signal and wait (SW)              -> Modula  
Der signalisierende Prozess muss auf erneuten Monitorzugang warten. Der reaktivierte Prozess erhält den Monitorzugang.  
Die Gültigkeit der signalisierten Bedingung ist garantiert.





## *Monitor-Simulation eines Semaphors*

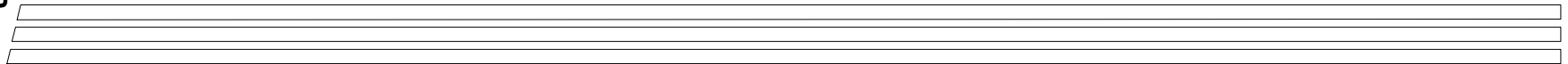
```
monitor Semaphore {
    int s = 0
    condvar positiv
    procedure Psem () {
        while (s == 0) { wait(positiv) }
        s -= 1
    }
    procedure Vsem () {
        s += 1
        signal (positiv)
    }
}
```

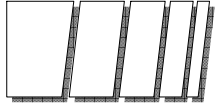




## *Monitor für beschränkten Puffer*

```
monitor Bounded_Buffer {
  typeT buf[n]
  int front = 0; rear = 0; count = 0
  ## Invariante: rear = (front + count) mod n
  condvar not_full, not_empty
  procedure enter (typeT data) {
    while (count == n) { wait(not_full) }
    buf[rear] = data; count += 1; rear = (rear+1) mod n
    signal (not_empty)
  }
  procedure get (typeT &result) {
    while (count == 0) { wait(not_empty) }
    result = buf [front]; count--; front = (front+1) mod n
    signal (not_full)
  }
}
```





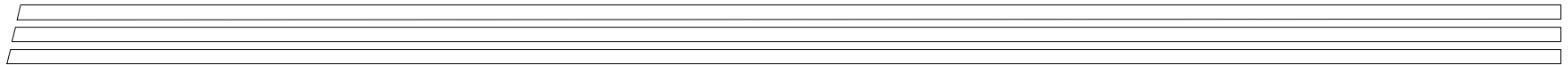
# *Exkurs: Thread-Synchronisation in Java*

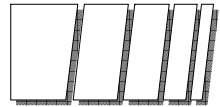
**Threads**

**Wechselseitiger Ausschluss**

**Bedingte Synchronisation**

**Deadlock**



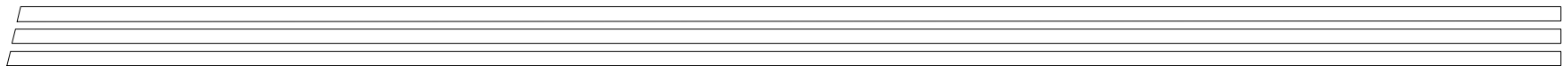


## *Die Klasse Thread*

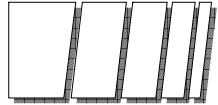
- Die Klasse Thread gehört zur Standardbibliothek von Java.
- Zur Generierung eines weiteren Kontrollflusses muss zunächst ein Objekt dieser Klasse erzeugt werden:

```
Thread worker = new Thread ( );
```

- Dieses kann dann konfiguriert werden (Setzen von initialer Priorität, Namen etc.) und anschließend zum Ablauf gebracht werden.
- Durch Aufruf der Methode start wird ein auf den Daten im Thread-Objekt basierender Kontrollfluß initiiert und durch Aufruf der Methode run aktiviert.
- Die Methode run muss für erweiterte Thread-Klassen neu definiert werden. Die voreingestellte Implementierung ist wirkungslos.







## *Beispiel: Die Klasse PingPong*

```
class PingPong extends Thread {
    String word; int delay;
    PingPong (String whatToSay, int delayTime) {
        word = whatToSay;    delay = delayTime;
    }
    public void run() {
        try { for (;;) {
            System.out.print(word+`` ``);
            sleep(delay);
        }
        } catch (InterruptedException e) { return; }
    }
    public static void main (String[] args) {
        new PingPong (``ping``,33).start();
        new PingPong (``PONG``,100).start();
    }
}
```

---

---

---

---



## *Das Interface Runnable*

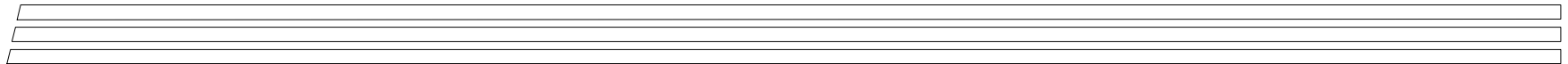
- **Alternativ können Threads durch Implementierung des Interfaces Runnable programmiert werden:**

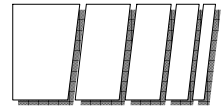
```
class Simple implements Runnable {  
    public void run() { ... }  
}
```

- **Threaderzeugung**

```
Runnable s = new Simple();  
new Thread(s).start();
```

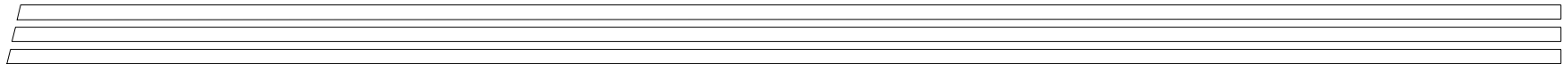
- **Vorteil: Die Klasse Simple kann auch andere Klassen erweitern, was als Erweiterung der Thread-Klasse nicht möglich wäre.**



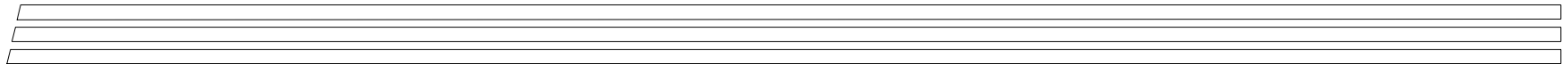
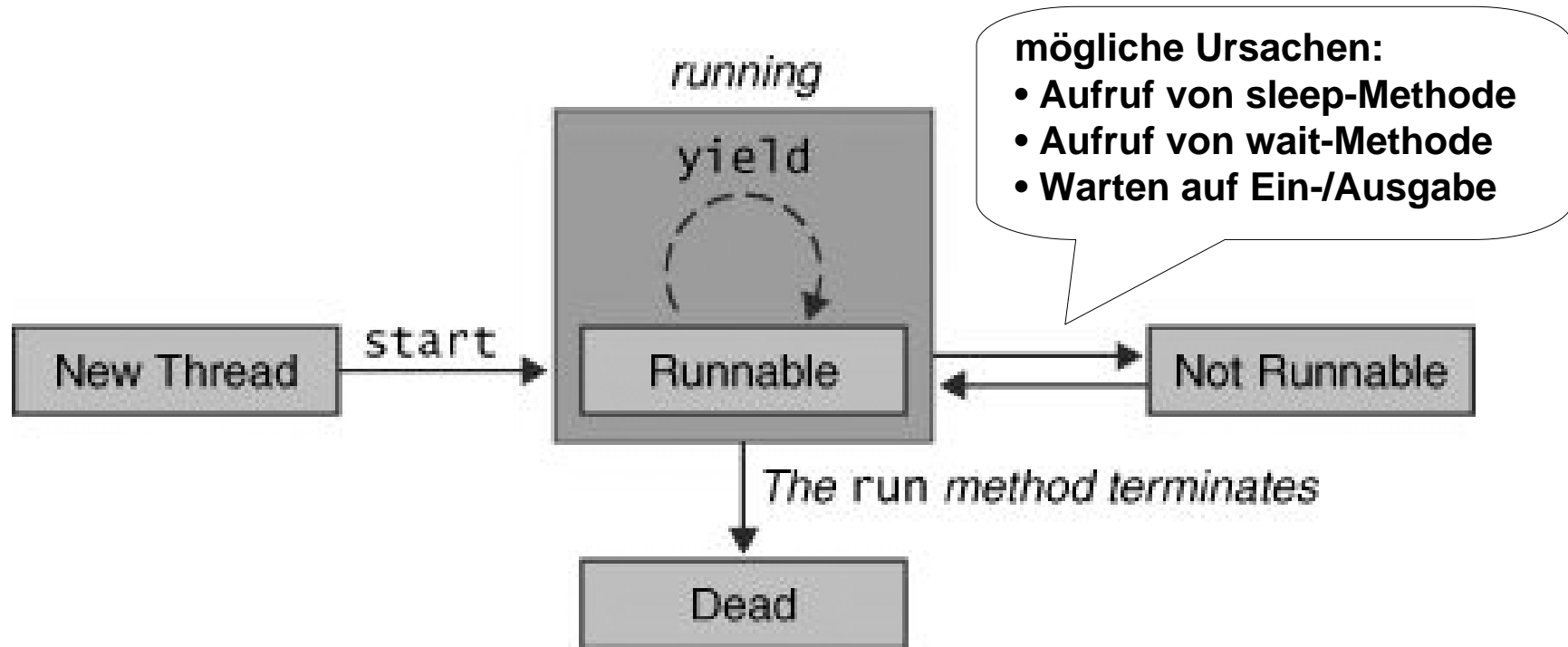


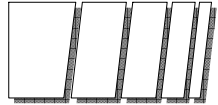
## *Threaderzeugung in 4 Schritten*

- 1. Definition einer neuen Klasse**
  - als Erweiterung der Thread-Klasse oder
  - als Implementierung des Runnable-Interfaces
- 2. Definition einer Methode run in der neuen Klasse**
  - Festlegung des Thread-Anweisungsteils
- 3. Erzeugung einer Klasseninstanz mit der new-Anweisung**
- 4. Starten des Threads mit der Methode start.**



# Thread-Life Cycle





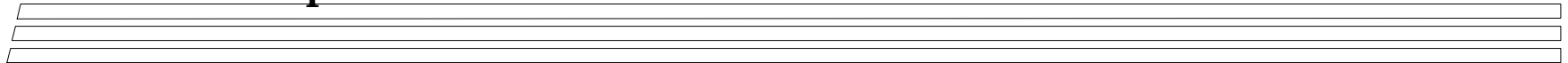
## *Wechselseitiger Ausschluss*

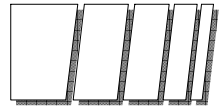
mit `synchronized` deklarierten Methoden:

Wird eine solche Methode aufgerufen, wird eine Sperre gesetzt und es kann keine andere als „`synchronized`“ deklarierte Methode ausgeführt werden, solange die Sperre gesetzt ist.

```
Bsp: class Account { private double balance;
    public Account (double initialDeposit)
        {   balance = initialDeposit;   }
    public synchronized double getBalance ()
        {   return balance;   }
    public synchronized void deposit (double amount)
        {   balance += amount;   }
}
```

Eine Klasse, deren extern aufrufbare Klassenmethoden alle mit `synchronized` versehen sind, entspricht somit weitgehend dem **Monitorkonzept**.





## Wechselseitiger Ausschluss

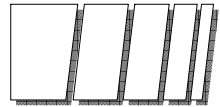
mit dem `synchronized` Konstrukt:

```
synchronized (<expr>)  
    <statement>
```

`expr` wird zu einem Objekt (oder Array) ausgewertet, das für einen weiteren Zugriff geschlossen werden soll. Falls das Schloss offen und damit der Eintritt erlaubt ist, wird `statement` ausgewertet.

Bsp: Alle Elemente eines Feldes sollen nicht negativ werden.

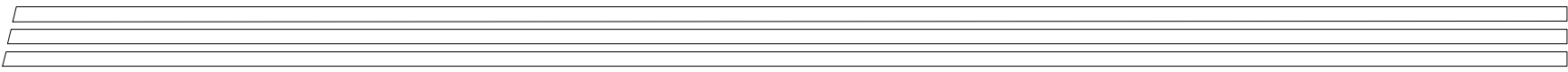
```
public static void abs (int[] values) {  
    synchronized(values) {  
        for (int i=0; i<values.length; i++) {  
            if (values[i] < 0)  
                values[i] = -values[i];  
        }  
    }  
}
```

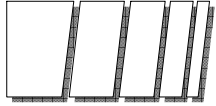


## *Beispiel: Producer-Consumer*

```
public class Producer/Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;
    public Producer/Consumer (CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(number, i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}
```

```
public void run() {
    int value = 0;
    for (int i = 0; i < 10; i++) {
        value = cubbyhole.get(number);
    }
}
```

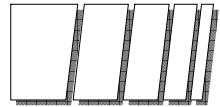




```
public class CubbyHole {  
    private int contents;  
    private boolean available = false;  
    public synchronized int get (int who) {  
        if (available == true) {  
            available = false;  
            System.out.println("Consumer " + who + " got: " + contents);  
            return contents;  
        }  
    }  
    public synchronized void put (int who, int value) {  
        if (available == false) {  
            contents = value;  
            available = true;  
            System.out.println("Producer " + who + " put: " + contents);  
        }  
    }  
}
```

## *Ein-Platz-Puffer - 1. Ansatz -*



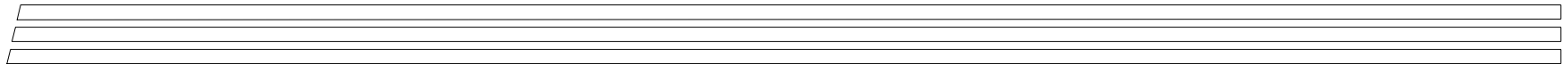


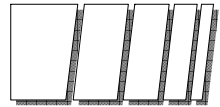
## *Bedingte Synchronisation*

- **Monitorkonzept nach Lampson/Redell (1980)**
- **Die Klasse Object enthält Methoden**
  - `wait` zum Blockieren von Threads
  - `notify` zum Reaktivieren des am längsten wartenden Threads
  - `notifyall` zum Reaktivieren aller wartenden Threads

die von allen Klassen ererbt werden.

- **Es gibt keine Bindung an eine Bedingungsvariable, jedoch dürfen `wait` und `notify` nur in `synchronized`-Klassenmethoden vorkommen.**
- **Im Unterschied zu dem im Monitorkonzept von Hoare verwendeten `signal` führt `notify` nicht zur unmittelbaren Ausführung des reaktivierten Threads, sondern lediglich zu seiner Einreihung in die Warteschlange der auf die Ausführung einer synchronisierten Methode wartenden Prozesse.**
- **`notify` ist gegenüber `signal` weniger fehleranfällig und erlaubt eine einfachere Warteschlangenverwaltung.**





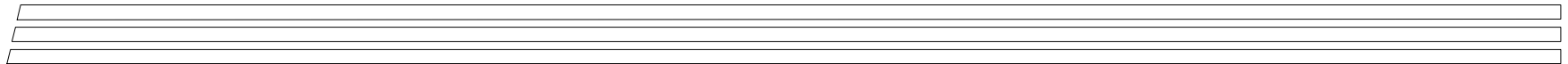
## *Standardschemata*

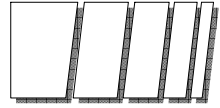
- **Bedingung abwarten:**

```
synchronized doWhenCondition () {  
    while (!condition)  
        wait();  
    ... „Aktion bei erfüllter Bedingung“ ...  
}
```

- **Bedingung signalisieren:**

```
synchronized changeCondition () {  
    ... „Änderung von Werten, Bedingung erfüllen“ ...  
    notify();  
}
```





```
public class CubbyHole {
```

```
    private int contents;
```

```
    private boolean available = false;
```

```
    public synchronized int get (int who) {
```

```
        while (available == false) { try { wait(); } catch (InterruptedException e) { }  
        }
```

```
        available = false;
```

```
        System.out.println("Consumer " + who + " got: " + contents);
```

```
        notifyAll();
```

```
        return contents;
```

```
    }
```

```
    public synchronized void put (int who, int value) {
```

```
        while (available == true) { try { wait(); } catch (InterruptedException e) { }  
        }
```

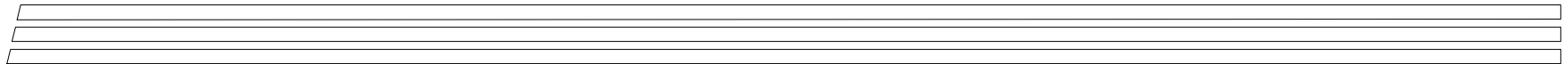
```
        contents = value; available = true;
```

```
        System.out.println("Producer " + who + " put: " + contents);
```

```
        notifyAll();
```

```
    }
```

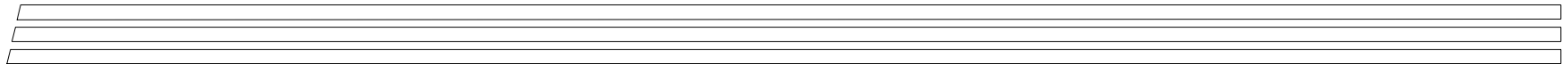
```
}
```

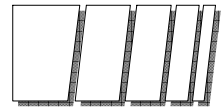


## *Ein-Platz-Puffer*

# *Deadlocks*

- **Synchronisierte Methoden eines Objektes können beliebige andere Methoden aufrufen (synchronisierte und nicht synchronisierte, desselben oder anderer Objekte)**
- **mögliche Ursachen für Deadlocks**
  - synchronisierte Methoden rufen sich gegenseitig auf
  - Threads benötigen Locks auf mehrere Objekte
  - irregulär terminierende Threads
- **Java hat keine Mechanismen, um Deadlocks zu verhindern oder aufzulösen.**





## *Deadlocks - ein konstruiertes Beispiel*

```
class Dead { // erzeugt zwei Threads
  public static void main (String[] args) {
    final Object resource1 = new Object();
    final Object resource2 = new Object();
    Thread t1 = new Thread(
      new Runnable() {
        public void run() {
          synchronized(resource1) {
            compute(1);
          }
          synchronized(resource2) {
            compute(1);
          }
        }
      }
    );
  }
};
```

```
Thread t2 = new Thread(
  new Runnable() {
    public void run() {
      synchronized(resource2) {
        System.out.println();
      }
      synchronized(resource1) {
        compute(2);
      }
    }
  }
);
t1.setPriority(6);    t1.start();
t2.setPriority(10);   t2.start();
};
static void compute(int i){
  System.out.println("Thread"+ i);
  for (int k=0; k<= 2000; k++)
    System.out.print(k); }
} // end Dead
```

