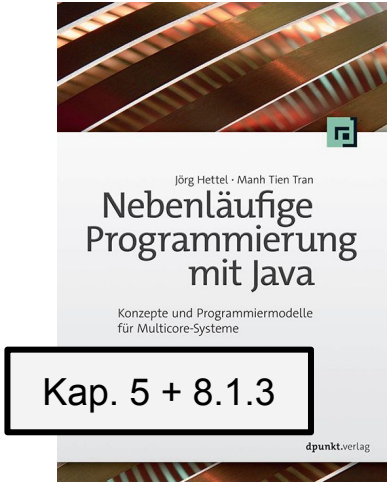
# Parallele Programmierung

Steuerung von Threads

# Überblick

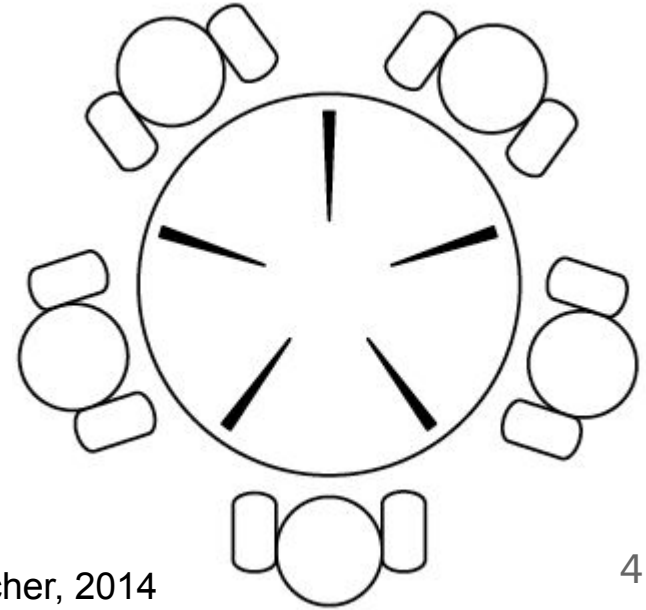
- Wiederholung
- Steuerung von Threads
  - mit synchronized
  - ***Übung:*** *Dinierende Philosophen*
  - mit synchronized und Signalisieren (`notifyAll()`)
  - ***Übung:*** *Ringspeicher*
  - mit mehreren `Condition`-Objekten und Signalisieren (`signalAll()`)
  - ***Übung:*** *Ringspeicher (mit `Condition`)*
- Pflichtaufgabe: *Dinierende Philosophen* (mit `Condition`)



# Steuerung von Threads mit synchronized

# pp03.01-SynchPhilosopher (15 Minuten)

- Warum ist es keine Lösung jeden Zugriff auf Variablen zu synchronisieren?
  - Effizienz (keine Parallelisierung mehr möglich)
  - Verklemmungsmöglichkeit (“Deadlock”)
- Beispiel dinierende Philosophen
  - denken
  - linkes Essstäbchen greifen
  - rechtes Essstäbchen greifen
  - essen
  - rechtes Essstäbchen zurücklegen
  - linkes Essstäbchen zurücklegen
  - von vorne…
- Greifen eines Essstäbchen kritischer Abschnitt
  - Essstäbchen exklusive Ressource



Bildquelle: Butcher, 2014

# Deadlock bein dinierenden Philosophen

Wenn **alle** Threads auf die Locks in **derselben numerischen Reihenfolge** zugreifen (“acquire”), dann kommt es **nicht zu einem Deadlock**.

```
synchronized (left) {
    synchronized (right) {
        //…
    }
}
```

Die meisten Philosophen folgen diesem Schema: `left` und `right` sind Chopstick-Objekte. Philosoph *i* und Philosoph *i+1* teilen ein `Chopstick`-Objekt. Für Philosoph *i* ist das das rechte, auf das zuletzt zugegriffen wird, für Philosoph *i+1* ist das das linke, auf das zuerst zugegriffen wird.

Nur im Fall des letzten bzw. 0. Philosoph wird diese Reihenfolge gebrochen, daher kann es zum Deadlock kommen.

*To my mind, what makes multithreaded programming difficult is* ***not that writing it is hard****, but that* ***testing it is hard****. It's not the pitfalls that you can fall into; it's the fact that* ***you don't necessarily know*** *whether you've fallen into one of them.*

Paul Butcher (2014). *Seven Concurrency Models in Seven Weeks. When Threads Unravel.* The Pragrmatic Programmers, S. 45

# Deadlock, Starvation, Livelock

***Deadlock* (Verklemmung):**

- “Eine Menge von Prozessen befindet sich in einem Deadlock-Zustand, wenn jeder Prozess aus der Menge auf ein Ereignis wartet, das nur ein anderer Prozess aus der Menge auslösen kann.”

**Verhungern (*starvation*):**

- Ein Prozess erhält nie Zugriff auf eine Ressource, die von mehreren Prozessen beansprucht wird.

***Livelock* (Spezialfall von Verhungern):**

- Zustand, in dem mindestens ein Prozess verhungert, weil die Kontrolle zwischen anderen Prozessen hin und her geht, ohne dass verhungernde Prozesse die Kontrolle jemals mehr erlangen können.

# Prioritätsinversion (Prioritätsumkehr)

**Sachverhalt**

- Prozess mit niedriger Priorität hat Betriebsmittel akquiriert.
- Prozess mit hoher Priorität möchte dasselbe Betriebsmittel und wird daher blockiert.
- Ein lange laufender Prozess mit mittlerer Priorität kannibalisiert Rechenzeit, so dass der niederpriorisierte Prozess nicht mehr drankommt.
- **Der hochpriorisierte Thread verhungert.**

**Lösung (eine von mehreren möglichen): Prioritätsvererbung**

- Die Bedeutung im Java-Umfeld ist unklar, da kein spezifisches Scheduling Verfahren für JVM vorgegeben ist.

**Beispiel:** Pathfinder (NASA Mars-Fahrzeug, 1996)

- gemeinsame Ressource: Middleware zur Datenspeicherung
- Prozess niederer Priorität: Geo/Met Datensammlung
- Prozess hoher Priorität: Überwachung der Middleware (Watchdog)

# Steuerung von Threads mit Bedingungs-variablen und Signalen

# Monitor (wait, notify, notifyAll)

Manchmal ist es erforderlich Threads **über Ereignisse zu steuern**.

Dazu dient ein Vermittlerobjekt. Es muss lediglich von `Object` erben. Die beteiligten Threads kommunizieren über den Vermittler miteinander.

Ein `Thread` muss im Besitz des Locks beim Vermittler sein, um die Methoden `wait()`, `notify()` und `notifyAll()` aufrufen zu dürfen, sonst wird eine `IllegalMonitorStateException` geworfen.

Diese Methoden können somit nur innerhalb von `synchronized`-Blöcken bzw. -Methoden verwendet werden.

Alle wartenden Threads sind in der ***condition queue*** des Locks, auch ***condition variable*** genannt.

# Monitor: wait

`obj.wait()`:

- Der aufrufende Thread `t` muss das Schloss des vermittelnden Lock-Objekts `obj` geschlossen haben
  - `synchronized(obj){... obj.wait(); ...}`
- Der aufrufende Thread `t` trägt sich in die Warteliste des vermittelnden Lock-Objekts `obj` ein.
- Der aufrufende Thread `t` gibt die Sperre auf `obj` wieder frei.
- Er wechselt in den **`Waiting`** -Zustand und bleibt so lange darin, bis ihn ein anderer Thread durch `obj.notify()` oder `obj.notifyAll()` weckt oder er ein `t.interrupt()` erhält.

# Monitor: notifyAll

`notifyAll()`:

- Der aufrufende Thread `t` muss das Schloss des vermittelnden Lock-Objekts `obj` geschlossen haben.
  - `synchronized(obj){... obj.notifyAll(); ...}`
- Der Scheduler weckt alle auf `obj` wartenden Threads auf (neuer Zustand **`Running`**).
- Ein wieder aktivierter Thread schließt zunächst das Schloss `obj` wieder, bevor er mit den Anweisungen nach `wait()` fortfährt. Daher kann nur einer der wartenden Thread fortfahren, obwohl alle aufgeweckt wurden.

# Monitor: notify

`notify()`:

- Der aufrufende Thread `t` muss das Schloss des vermittelnden Lock-Objekts `obj` geschlossen haben.
  - `synchronized(obj){... obj.notify(); ...}`
- Der Scheduler wählt einen auf `obj` wartenden Thread aus und weckt ihn (neuer Zustand **`Running`**).
- Ein wieder aktivierter Thread schließt zunächst das Schloss `obj` wieder, bevor er mit den Anweisungen nach `wait()` fortfährt.

Folie mit Anmerkungen

# Benutzungsmuster

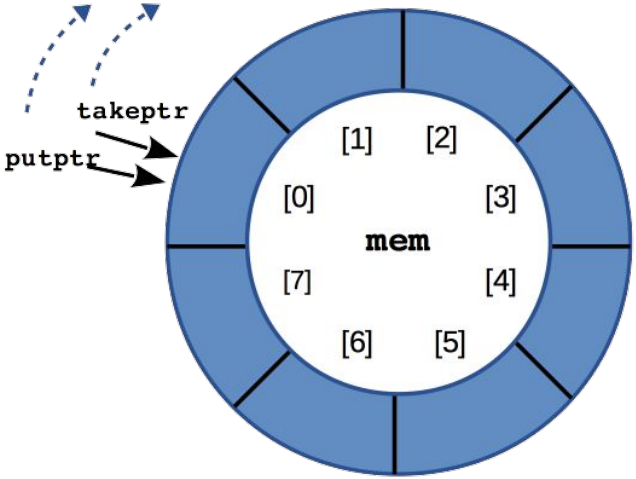
```
synchronized(obj) {
    while (!«Bedingung ist true») {
        obj.wait();
    }
    «Benutzung geteilter Ressourcen»
    obj.notifyAll();
}
```

falls in der Zwischenzeit die Bedingung wieder eingetreten ist oder der Thread “fehlerhaft” geweckt wurde; hier `if` wäre ein Fehler!

Falls mehrere Threads auf unterschiedliche Bedingungen an `obj` warten, muss zum Signalisieren immer `notifyAll()` benutzt werden! Andernfalls kann es bei einem Interrupt dazu kommen, dass die Wirkung von `notify()` verlorengeht (falls ein Thread geweckt wird, der gerade auf eine noch nicht realisierte Bedingung wartet). Ein Deadlock wäre dadurch möglich.

# Ringpuffer: Start

Folie mit Anmerkungen



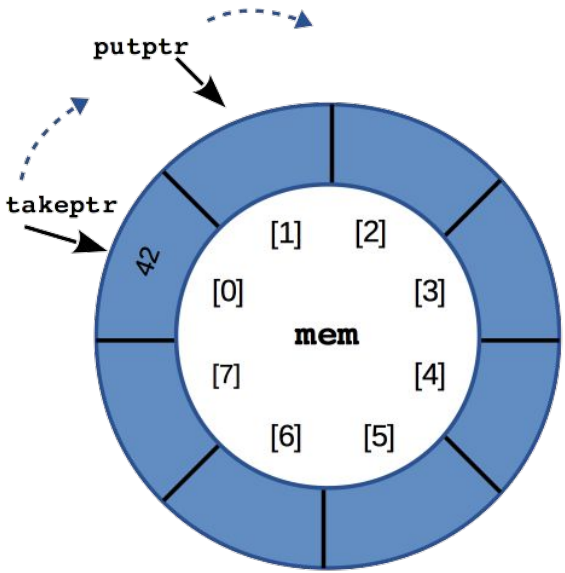

```
mem.length   == 8 (Kapazität von mem)
count        == 0 (Anzahl aktuelle Elem.)
takeptr      == 0 (Index nächstes put())
putptr       == 0 (Index nächstes take())
```

**solange**
**“leer” =>**
**`take()`-Operationen müssen warten**

Bildquelle: Hettel & Tran, 2016 (abgewandelt)

# Ringpuffer: put(42)

Folie mit Anmerkungen

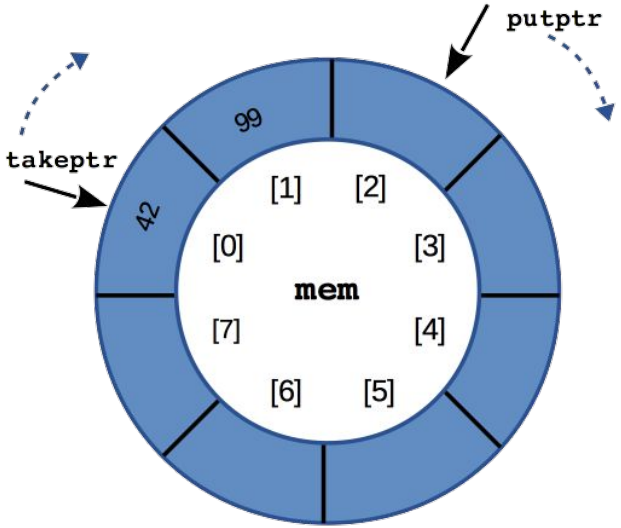



```
mem.length   == 8  (Kapazität von mem)
count        == 1  (Anzahl aktuelle Elem.)
takeptr      == 0  (Index nächstes put())
putptr       == 1  (Index nächstes take())
```

Bildquelle: Hettel & Tran, 2016 (abgewandelt)

# Ringpuffer: put(99)

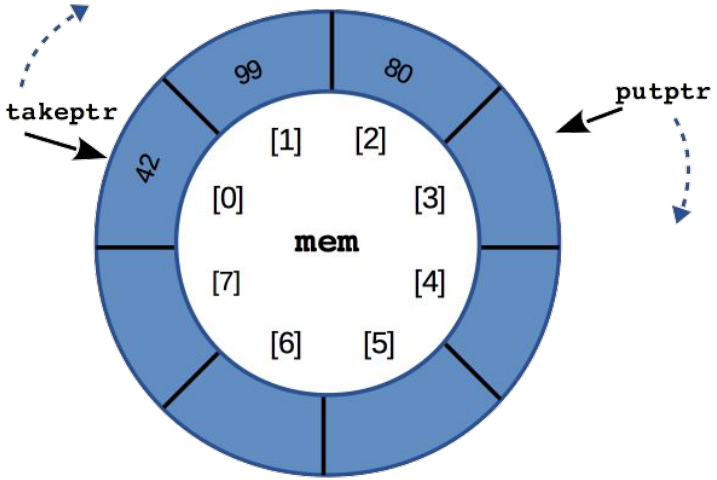
Folie mit Anmerkungen




```
mem.length   == 8  (Kapazität von mem)
count        == 2  (Anzahl aktuelle Elem.)
takeptr      == 0  (Index nächstes put())
putptr       == 2  (Index nächstes take())
```

Bildquelle: Hettel & Tran, 2016 (abgewandelt)

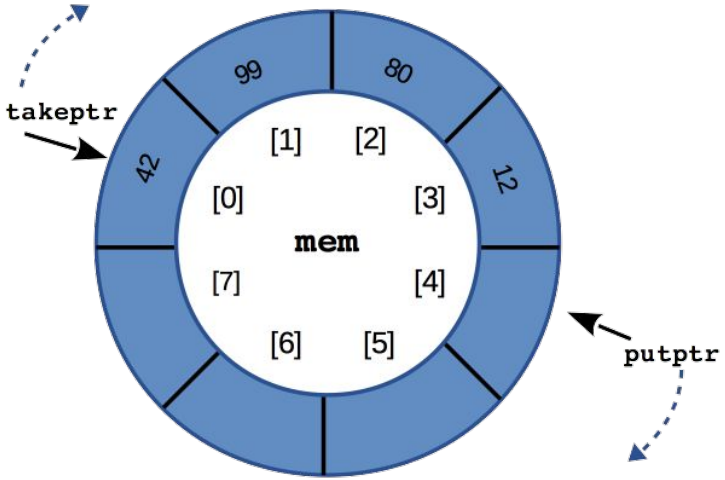
# Ringpuffer: put(80)




```
mem.length   == 8  (Kapazität von mem)
count        == 3  (Anzahl aktuelle Elem.)
takeptr      == 0  (Index nächstes put())
putptr       == 3  (Index nächstes take())
```

Bildquelle: Hettel & Tran, 2016 (abgewandelt)

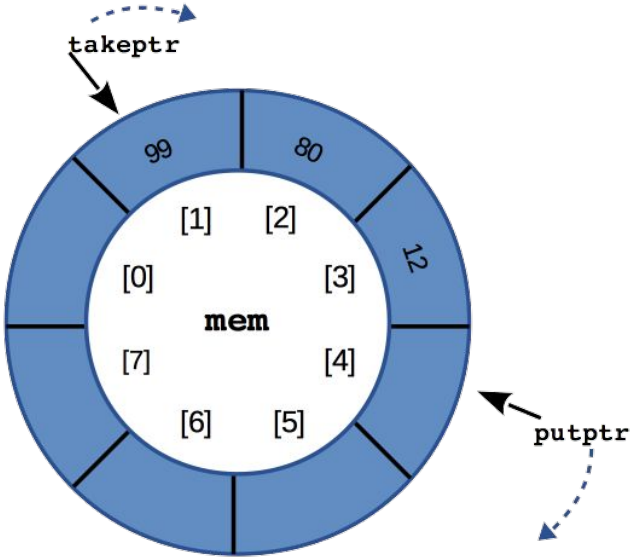
# Ringpuffer: put(12)




```
mem.length   == 8  (Kapazität von mem)
count        == 4  (Anzahl aktuelle Elem.)
takeptr      == 0  (Index nächstes put())
putptr       == 4  (Index nächstes take())
```

Bildquelle: Hettel & Tran, 2016 (abgewandelt)

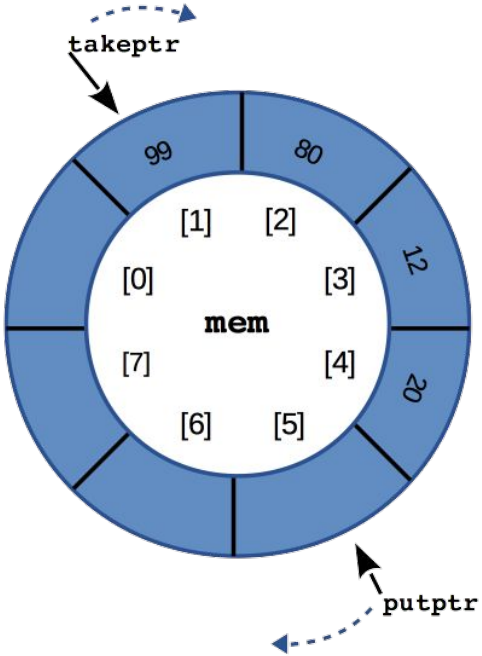
# Ringpuffer: take() => 42




```
mem.length   == 8  (Kapazität von mem)
count        == 3  (Anzahl aktuelle Elem.)
takeptr      == 1  (Index nächstes put())
putptr       == 4  (Index nächstes take())
```

Bildquelle: Hettel & Tran, 2016 (abgewandelt)

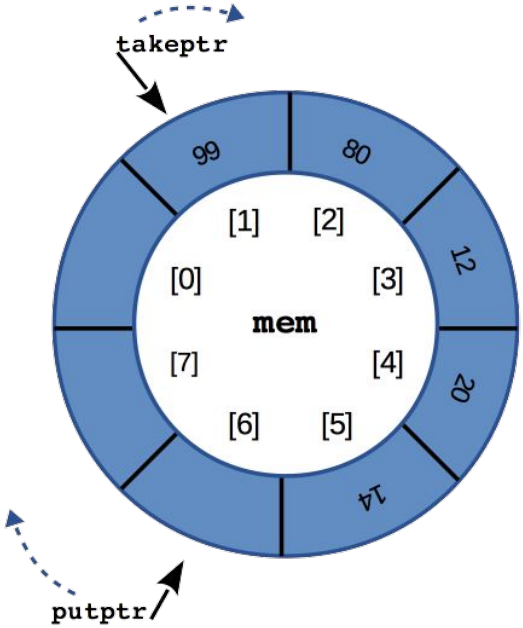
# Ringpuffer: put(20)




```
mem.length   == 8  (Kapazität von mem)
count        == 4  (Anzahl aktuelle Elem.)
takeptr      == 1  (Index nächstes put())
putptr       == 5  (Index nächstes take())
```

Bildquelle: Hettel & Tran, 2016 (abgewandelt)

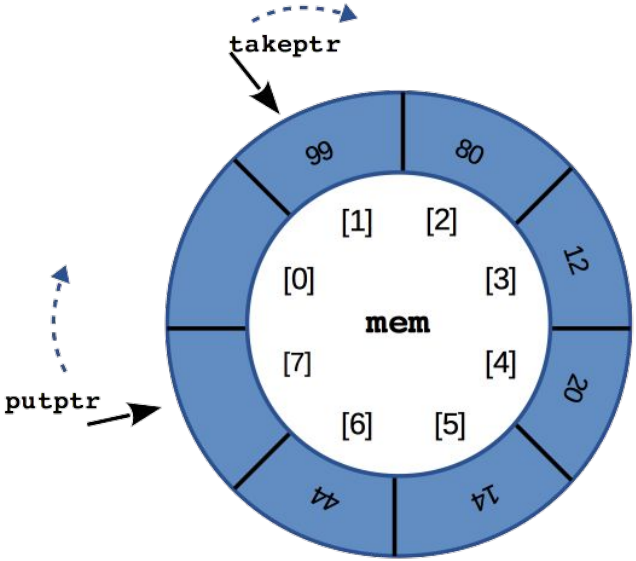
# Ringpuffer: put(14)




```
mem.length   == 8  (Kapazität von mem)
count        == 5  (Anzahl aktuelle Elem.)
takeptr      == 1  (Index nächstes put())
putptr       == 6  (Index nächstes take())
```

Bildquelle: Hettel & Tran, 2016 (abgewandelt)

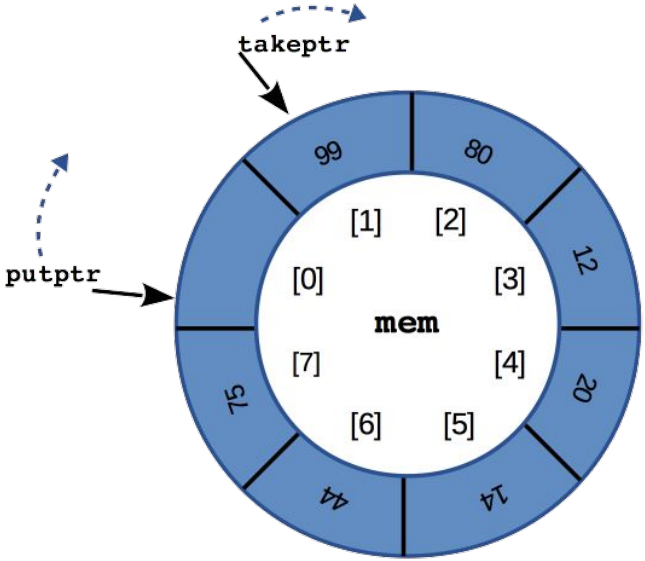
# Ringpuffer: put(44)




```
mem.length   == 8  (Kapazität von mem)
count        == 6  (Anzahl aktuelle Elem.)
takeptr      == 1  (Index nächstes put())
putptr       == 7  (Index nächstes take())
```

Bildquelle: Hettel & Tran, 2016 (abgewandelt)

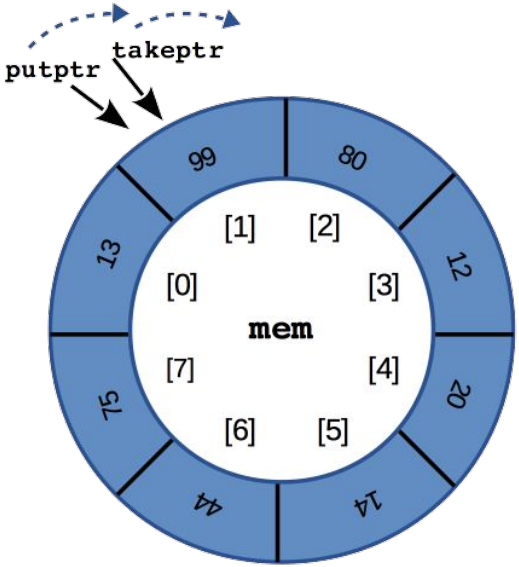
# Ringpuffer: put(75)




```
mem.length   == 8  (Kapazität von mem)
count        == 7  (Anzahl aktuelle Elem.)
takeptr      == 1  (Index nächstes put())
putptr       == 0  (Index nächstes take())
```

Bildquelle: Hettel & Tran, 2016 (abgewandelt)

# Ringpuffer: put(13)

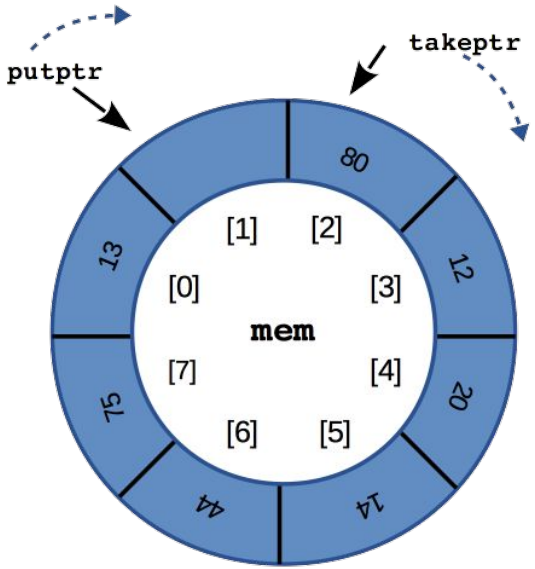



```
mem.length   == 8  (Kapazität von mem)
count        == 8  (Anzahl aktuelle Elem.)
takeptr      == 1  (Index nächstes put())
putptr       == 1  (Index nächstes take())
```

**solange**
**“voll” =>**
**`put()`-Operationen müssen warten**

Bildquelle: Hettel & Tran, 2016 (abgewandelt)

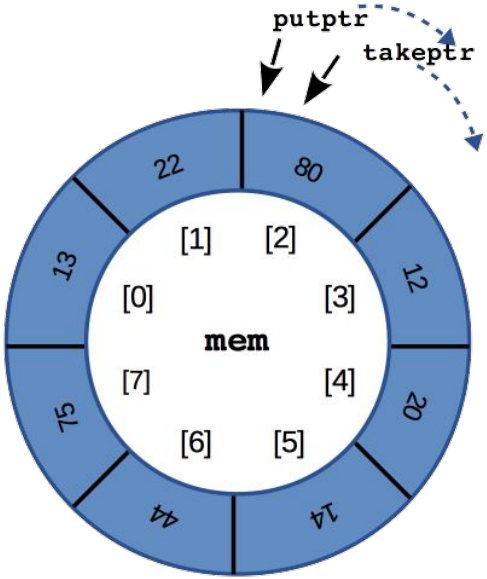
# Ringpuffer: take() => 99




```
mem.length   == 8  (Kapazität von mem)
count        == 7  (Anzahl aktuelle Elem.)
takeptr      == 2  (Index nächstes put())
putptr       == 1  (Index nächstes take())
```

Bildquelle: Hettel & Tran, 2016 (abgewandelt)

# Ringpuffer: put(22)

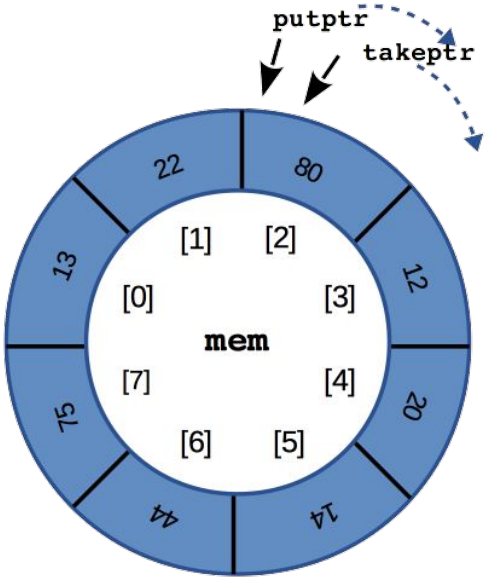



```
mem.length   == 8  (Kapazität von mem)
count        == 8  (Anzahl aktuelle Elem.)
takeptr      == 2  (Index nächstes put())
putptr       == 2  (Index nächstes take())
```

**solange**
**“voll” =>**
**`put()`-Operationen müssen warten**

Bildquelle: Hettel & Tran, 2016 (abgewandelt)

# Ringpuffer:




```
mem.length   == 8  (Kapazität von mem)
count        == 8  (Anzahl aktuelle Elem.)
takeptr      == 2  (Index nächstes put())
putptr       == 2  (Index nächstes take())
```

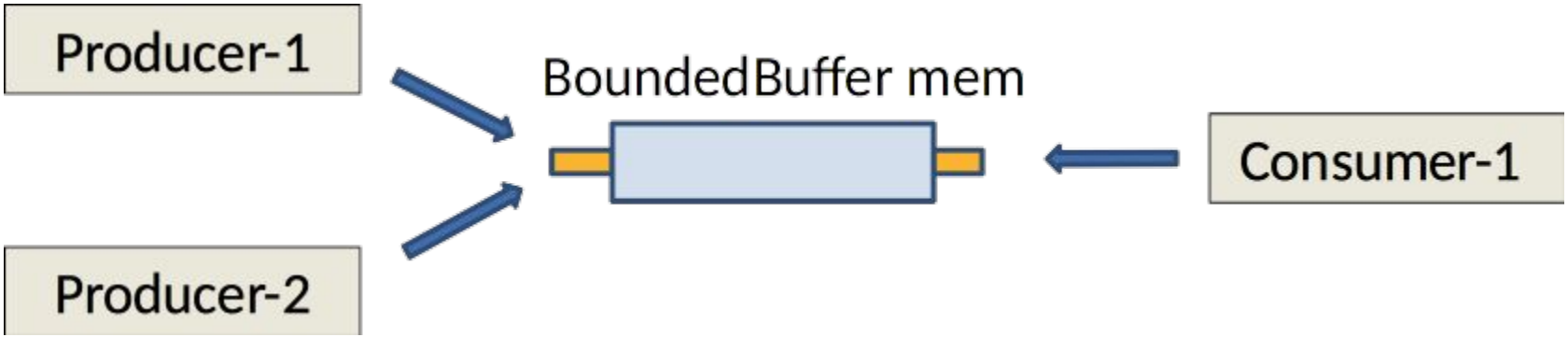
**Bedingungen:**

1) **“leer”, wenn**
   `count == 0`
2) **“voll”, wenn**
   `count == mem.length`

Bildquelle: Hettel & Tran, 2016 (abgewandelt)

# Ringpuffer: pp03.02-BoundedQueueWaitNotify (20 Minuten)

Folie mit Anmerkungen




Bildquelle: Hettel & Tran, 2016 (abgewandelt)

# Steuerung von Threads mit Condition

# Bedingungsvariablen

Eine Bedingungsvariable ist immer implizit bei `synchronized` vorhanden, auf der `wait()`/`notify()` ausgeführt werden kann.

Es gibt Anwendungen mit **mehreren voneinander getrennten Bedingungen** (z.B. “leer” und “voll” bei Producer/Consumer). Wenn immer alle Threads mit `notifyAll()` aufgeweckt werden, kann es sein, dass Verarbeitungskapazität vergeudet wird. Mit `Condition` können **mehrere “Warteräume”** etabliert werden.

Deshalb ist es seit JDK 1.5 möglich mehrere Bedingungsvariablen (Interface **`java.util.concurrent.locks.Condition`**) in einem “Monitor” (Abschnitt mit gegenseitigem Ausschluss) zu verwenden.

Statt `synchronized` muss das Interface **`java.util.concurrent.locks.Lock`** benutzt werden, um den Monitor zu begrenzen.

Folie mit Anmerkungen

# Benutzungsmuster

```
import java.util.concurrent.locks.ReentrantLock;


ReentrantLock lock = new ReentrantLock();

lock.lock();
try {


    «Benutzung geteilter Ressourcen»
} finally {
    lock.unlock();
}
```

Damit sichergestellt ist, dass der Lock auch im Fehlerfall wieder freigegeben wird.

Folie mit Anmerkungen

# Benutzungsmuster

```
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;

ReentrantLock lock = new ReentrantLock();
Condition condition = lock.newCondition();
lock.lock();
try {
    while (!«Bedingung ist true») {
        condition.await();
    }
    «Benutzung geteilter Ressourcen»
} finally {
    lock.unlock();
}
```

falls in der Zwischenzeit die Bedingung wieder eingetreten ist oder der Thread “fehlerhaft” geweckt wurde; hier **if** wäre ein Fehler!

außer die Bedingungen sind komplett getrennt: Alle Threads, die in einem Warteraum warten, wären in der Lage ihre Arbeit wieder aufzunehmen, wenn dessen Bedingung eingetroffen ist.

# Benutzungsmuster

Wenn die wartenden Threads nun “bedingungsgenau” in getrennten Warteräumen (für mehrere Bedingungsvariablen) verweilen, kann `signal()` statt `signalAll()` zum Aufwecken benutzt werden, da sichergestellt ist, dass ein zufällig ausgewählter aufgeweckter Thread eine Situation vorfindet, in der die Bedingung, auf die gewartet wurde, erfüllt ist.

Würden mehrere Threads auf unterschiedliche inhaltliche Bedingungen in demselben Warteraum warten, könnte es sein, dass ein Thread geweckt wird, der auf eine Bedingung wartet, die noch nicht eingetreten ist – er ruft daher `wait()` für sich auf. Damit wären alle Threads dann im Zustand **`Waiting`** und es gäbe keine Möglichkeit, dass einer andere wecken könnte.

# java.util.concurrent.locks. Condition

Folie mit Anmerkungen

```
public interface Condition {
    public void await()                    // analog wait()
        throws InterruptedException;
    public void awaitUninterruptibly();
    public long awaitNanos(long nanos)
        throws InterruptedException;
    public boolean await(long time, TimeUnit unit)
        throws InterruptedException;
    public boolean awaitUntil(Date deadline)
        throws InterruptedException;
    public void signal();                  // analog notify()
    public void signalAll();               // analog notifyAll()
}
```

# pp03.03-BoundedQueueAwaitSignal BoundedBuffer (20 Minuten)

```
public class BoundedBuffer<T> {

    final Lock lock = new ReentrantLock();
    final Condition notFull = this.lock.newCondition();
    final Condition notEmpty = this.lock.newCondition();
    final Object[] items = new Object[8];
    int putptr, takeptr, count;
```

# Pflichtaufgabe (Gruppenarbeit): Abgabe bis 16.11.2020, 09:15 Uhr

Bauen Sie die (synchronisierten) dinierenden Philosophen so um, dass Bedingungsvariablen benutzt werden:

- Jeder Philosoph ist ein Thread und besitzt einen Warteraum, in dem die Nachbarphilosophen darauf warten, dass er seine Stäbchen freigibt.
- Jeder Philosoph hat je eine Referenz auf seinen linken und rechten Nachbarn.
  - `setLeft(IPhilosopher left); setRight(IPhilosopher right);`
- Der Tisch wird durch ein `ReentrantLock`-Objekt (Interface `Lock`) repräsentiert. Alle Philosophen müssen den Tisch **verwenden**, wenn sie beginnen zu essen. Eine Referenz auf den Tisch kann mit dem Setter übergeben werden:
  - `setTable(Lock table);`
- Möchte ein Philosoph beginnen zu essen, prüft er, *ob sein linker Nachbar oder sein rechter Nachbar isst*; er **erwartet** den Moment, in dem beide nicht essen. Dann beginnt er zu essen.
- Wenn ein Philosoph denkt, isst er nicht und **signalisiert** jeweils seinem linken Nachbarn und seinem rechten Nachbarn, dass er nicht isst.