Parallele Programmierung

Monitor Synchronisation

Vorlesung 2

Prof. Dr. Luc Bläser

Letzte Vorlesung - Quiz

```
Thread.currentThread().join()
```

Was passiert?

Thread Synchronisation

- Ohne Vorkehrungen laufen Threads beliebig verzahnt oder parallel
- Oft muss Nebenläufigkeit der Threads aber beschränkt werden

Synchronisation = Einschränkung der Nebenläufigkeit

Inhalt Heute

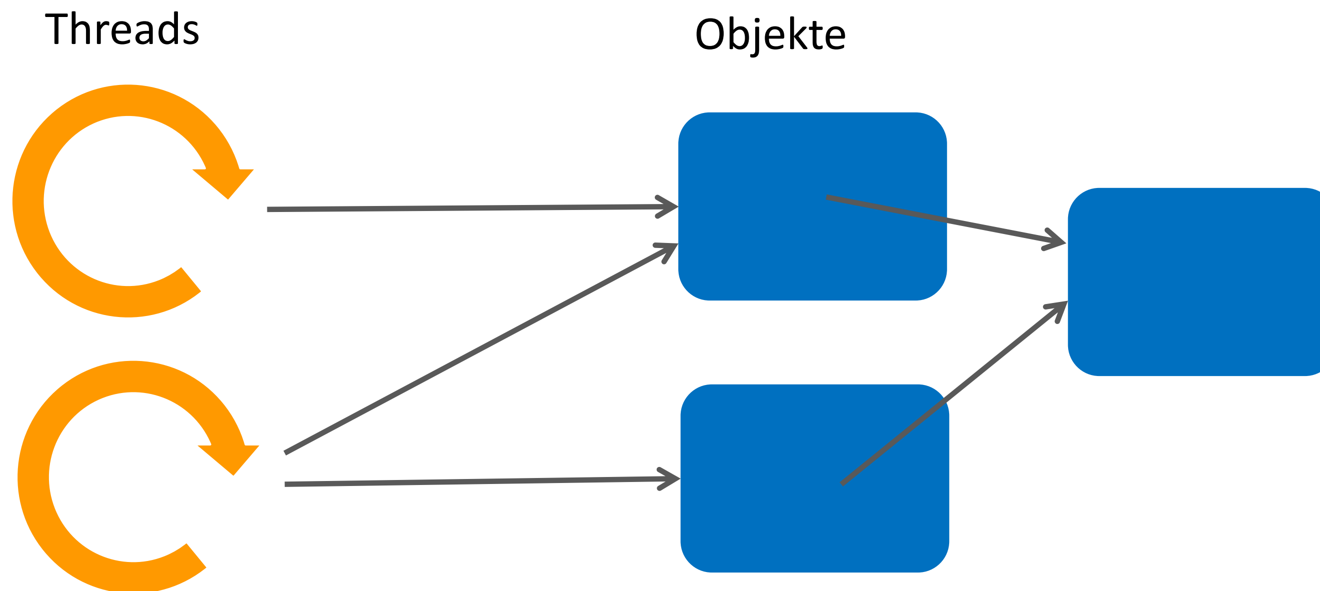
- Thread Synchronisation
  - Gemeinsame Ressourcen
  - Kritische Abschnitte
  - Gegenseitiger Ausschluss
  - Java Monitor Konzept
  - Wait & Signal
  - Fallstricke beim Java Monitor

Lernziele

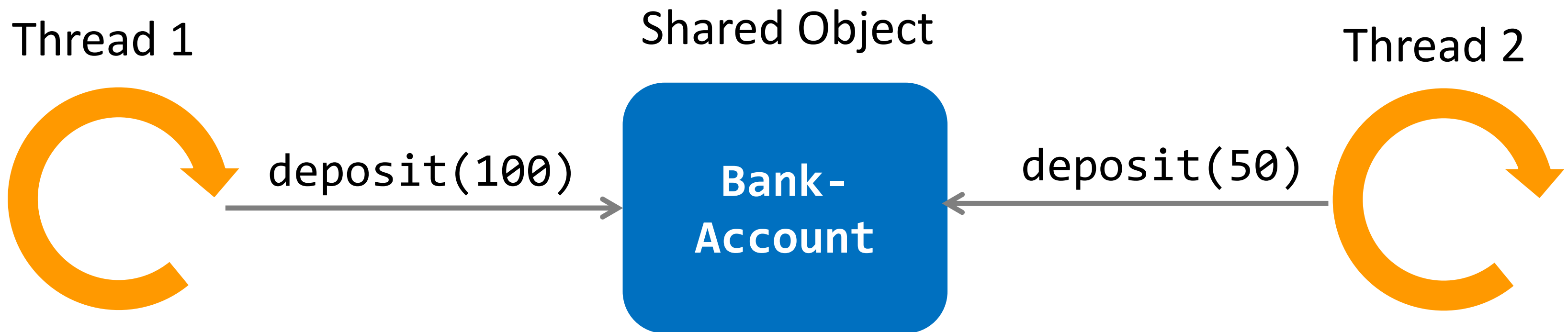
- Notwendigkeit bestimmter Synchronisation von Threads erkennen
- Das Monitorkonzept und seine Implementierung in Java verstehen
- Den Monitor zur Synchronisation in Java einsetzen können

Gemeinsame Ressourcen

- Threads teilen sich Adressraum und Heap



Race Condition Beispiel




```
class BankAccount {
   private int balance = 0;

   public void deposit(int amount) {
     this.balance += amount;
   }
}
```

Instruktionsebene

```
read balance => reg
reg = reg + amount
write reg => balance
```

Race Condition Szenario

Thread 1	Balance	Thread 2
read balance => reg (reg == 0)	0	
	0	read balance => reg (reg == 0)
reg = reg + amount (reg == 100)	0	
	0	reg = reg + amount (reg == 50)
write reg => balance	100	
	50	write reg => balance

Erwartet: 150

Kritischer Abschnitt

- deposit() ist kritischer Abschnitt (Critical Section)
  - Nur von einem Thread zur gleichen Zeit ausführbar
  - Brauche gegenseitigen Ausschluss (Mutual Exclusion)

```
public void deposit(int amount) {
  enter critical section
  this.balance += amount;
  exit critical section
}
```

Wie implementiert man den gegenseitigen Ausschluss?

Naiver Ansatz

```
class BankAccount {
  private int balance = 0;
  private boolean locked = false;

  public void deposit(int amount) {
    while (locked) { } // busy waiting loop
    locked = true;     // enter critical section
    this.balance += amount;
    locked = false;    // exit critical section
  }
}
```

Was ist hier falsch?

Naiver Ansatz - inkorrekt

```
public void deposit(int amount) {
  while (locked) { }
  locked = true;
  this.balance += amount;
  locked = false;
}
```

Schlaufe und Zuweisung sind nicht atomar

Thread 1	locked	Thread 2
Read locked in loop (locked == false)	false	Read locked in loop (locked == false)
Write locked = true	true	Write locked = true
Critical section		Critical section

Kein gegenseitiger Ausschluss

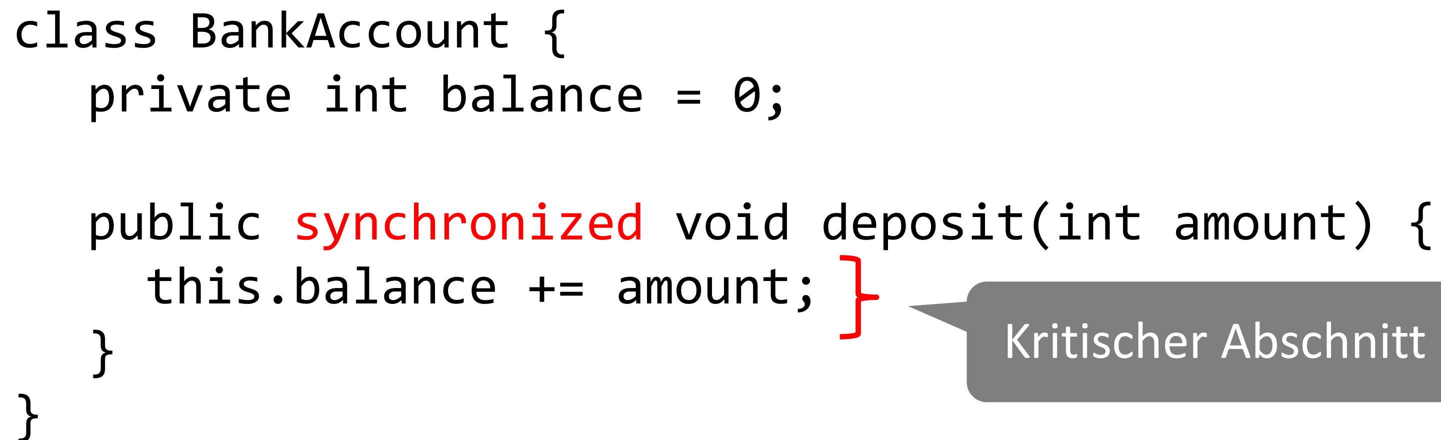
Gegenseitiger Ausschluss

- Korrekte Implementierung ist nicht trivial
  - Algorithmen wie Dekker, Peterson, Lamport’s Bakery
  - Atomare Compare-and-Set (CAS) Instruktionen
- Muss auch Weak Memory Consistency beachten
  - Sehe Änderungen anderer CPUs umgeordnet oder nicht
  - Brauche Memory Fences (z.B. volatile Keyword)
- Effizienzfragen
  - Busy Waiting zu teuer für lange Wartezeiten oder 1 CPU
  - Warteschlange teuer für sehr kurzes Warten bei Multi-CPU

=> Vorgefertigte Synchronisationsmechanismen

Java Synchronized Methoden

- synchronized Modifier für Methoden
  - Body der Methode ist ein kritischer Abschnitt
  - Wird unter gegenseitigem Ausschluss ausgeführt

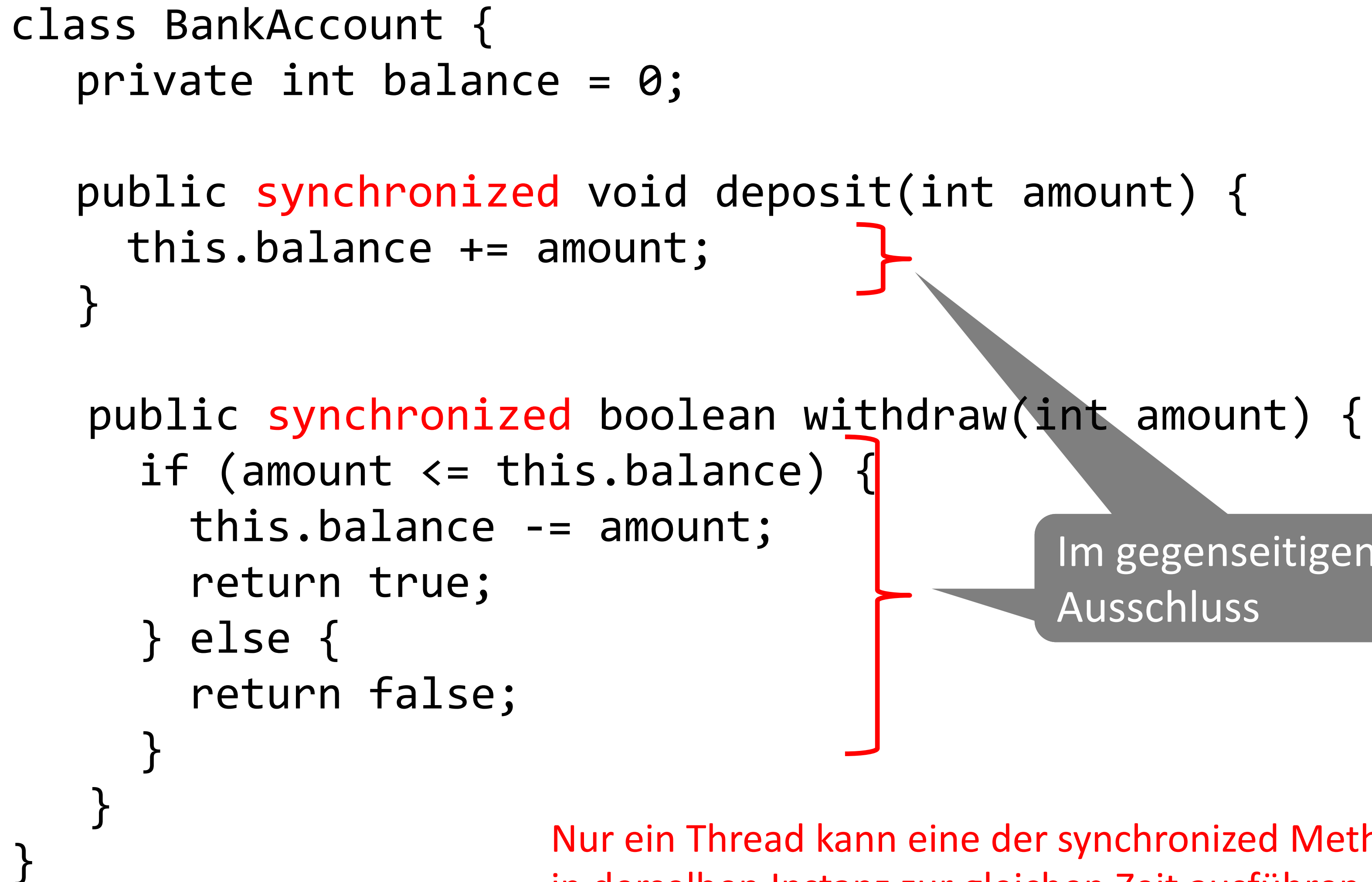


```
class BankAccount {
   private int balance = 0;

   public synchronized void deposit(int amount) {
     this.balance += amount;
   }
}
```

Funktionsweise von Synchronized

- Jedes Objekt hat einen Lock (Monitor-Lock)
  - Maximal ein Thread kann denselben Lock haben
- synchronized Block belegt den Lock des Objektes
  - Bei Eintritt wird Lock besetzt - sonst warten, bis frei
  - Bei Austritt wird Lock wieder freigegeben

Beispiel: Monitor-Lock



Synchronized Statement Block

- `synchronized(object) { statements }`
  – Explizite Angabe, auf welcher Instanz gelockt wird

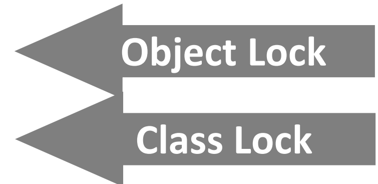
```
class BankAccount {
   private int balance = 0;

   public void deposit(int amount) {
      synchronized(this) {
        this.balance += amount;
      }
      System.out.println("Deposit done");
   }
}
```

Monitor-Lock auf this

Kritischer Abschnitt

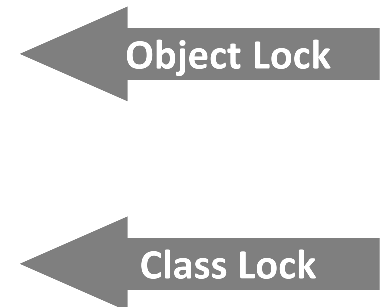
Äquivalente Verwendungen

```
public class Test {
   synchronized void f() { ... }
   static synchronized void g() { ... }
}
```



≡

```
public class Test {
  void f() {
    synchronized(this) { ... }
  }
  static void g() {
    synchronized(Test.class) { ... }
  }
}
```



Exit aus Synchronized Block

- Lock wird bei jedem Exit freigegeben
  - Ende des Blocks
  - Return Statement
  - Unbehandelte Exception

Naiver Ansatz: Warten auf Bedingung

```
class BankAccount {
  private int balance = 0;

  public synchronized void withdraw(int amount)
     throws InterruptedException {
    while (amount > this.balance) {
      Thread.sleep(1);
    }
    this.balance -= amount;
  }

  public synchronized void deposit(int amount) {
    this.balance += amount;
  }
}
```

Keine Lösung!

Wieso funktioniert das nicht?

Naiver Ansatz - inkorrekt

- sleep() und yield() geben Monitor-Lock nicht frei
  - Anderer Thread kann Bedingung gar nie erfüllen
- Pollen in Zeitabständen ineffizient

=> Wait & Signal Mechanismus des Monitors

Monitor Konzept

- P. Brinch Hansen, C. A. R. Hoare, 1973/1974

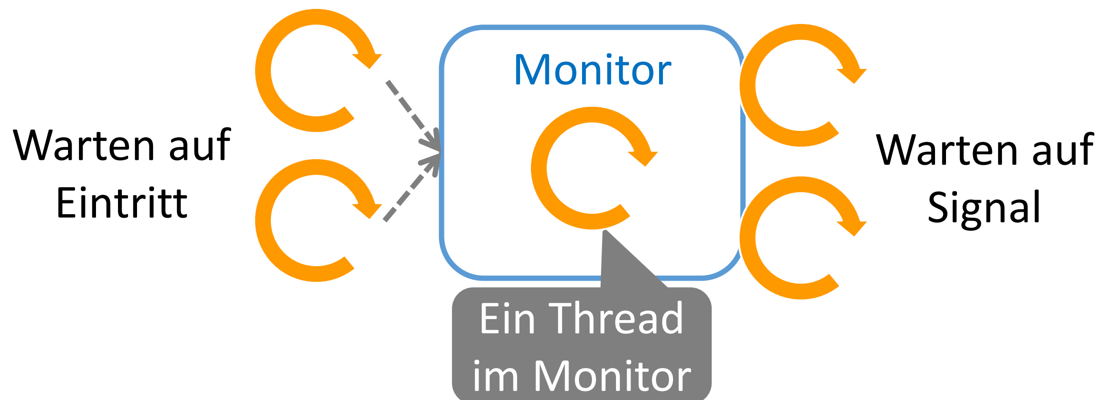
©wikipedia.org

Monitor

Objekt mit internem gegenseitigen Ausschluss und Wait & Signal Mechanismus

Monitor

- Objekt mit internem gegenseitigen Ausschluss
  - Nur ein Thread operiert zur gleichen Zeit im Monitor
  - Alle äusseren Methoden synchronized
- Wait & Signal Mechanismus
  - Threads können im Monitor auf Bedingung warten
  - Threads können Wartende aufwecken (signalisieren)



Monitor Beispiel

```
class BankAccount {
  private int balance = 0;

  public synchronized void withdraw(int amount)
                    throws InterruptedException {
    while (amount > balance) {
      wait();
    }
    balance -= amount;
  }

  public synchronized void deposit(int amount) {
    balance += amount;
    notifyAll();
  }
}
```

Warte auf Bedingung

Wecke alle im Monitor wartenden Threads

Wieso funktioniert das?

- `wait()` gibt Monitor-Lock temporär frei
  - Damit anderer Thread Bedingung im Monitor erfüllen kann

```
public synchronized void withdraw(int amount)
                    throws InterruptedException {
  while (amount > this.balance) {
    wait();
  }
  this.balance -= amount;
}
```

1. In Warteraum gehen
2. Monitor freigeben
3. (Inaktiv, bis zum Wecksignal)
4. Monitor neu beziehen

Wecksignal

- Signalisieren einer Bedingung im Monitor
  - `notify()` weckt einen beliebigen wartenden Thread im Monitor
  - `notifyAll()` weckt alle im Monitor wartende Threads
  - Kein Effekt, falls kein Thread wartet

```
public synchronized void deposit(int amount) {
  this.balance += amount;
  notifyAll();
}
```

1. Weckt alle Threads in wait()
2. Behält Monitor und macht weiter

Analyse des Beispiels

```
class BankAccount {
  private int balance = 0;

  public synchronized void withdraw(int amount)
                           throws InterruptedException {
    while (amount > balance) {
      wait();
    }
    balance -= amount;
  }

  public synchronized void deposit(int amount) {
    balance += amount;
    notifyAll();
  }
}
```

Wieso wird hier notifyAll() verwendet?
Wieso ist das Warten in einer Schlaufe?

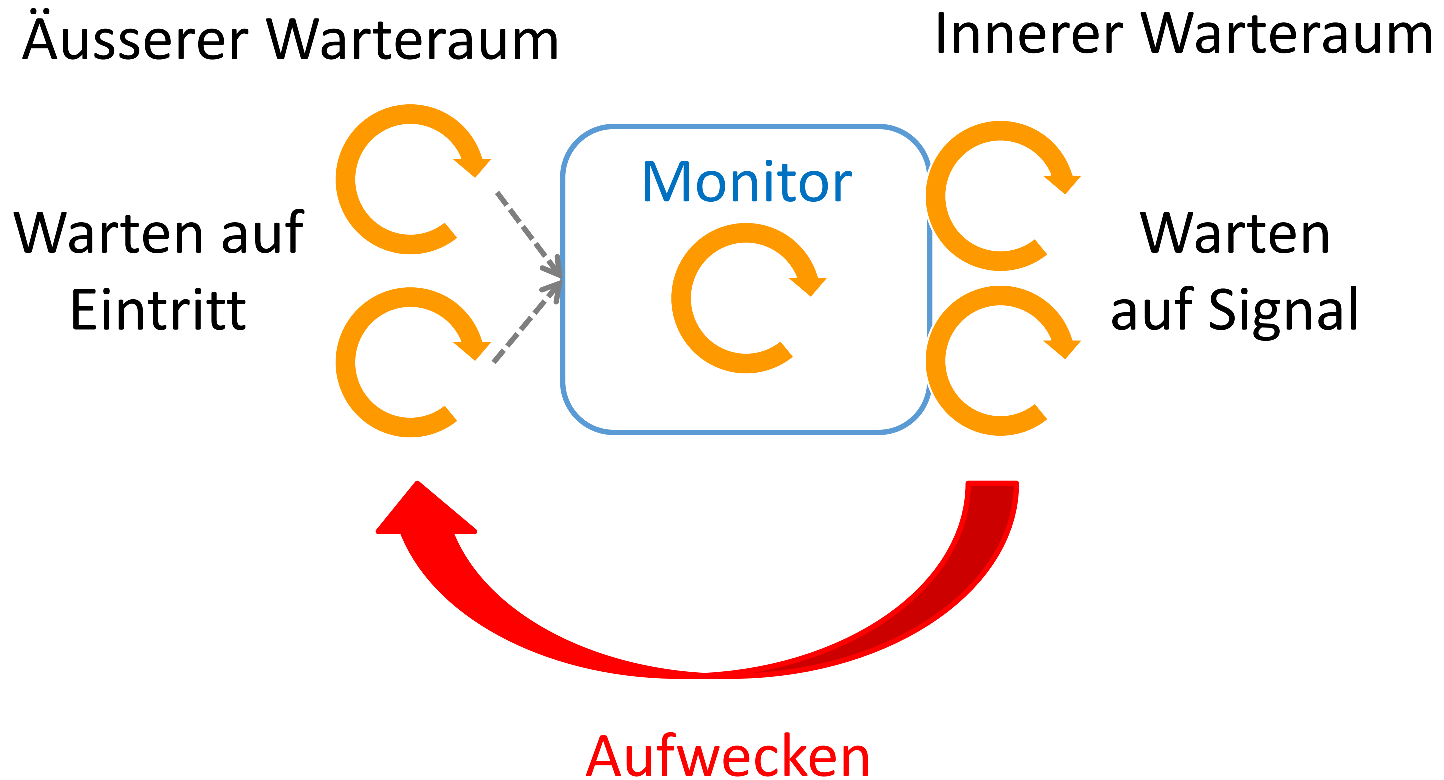
Pauschales Wait & Signal

- Keine Unterscheidung zwischen verschiedenen semantischen Bedingungen
  - Wartende müssen selber schauen, ob sie ein Signal interessiert (ihre Bedingung erfüllt ist)

Gründe zum Aufwachen aus wait()

- notifyAll(), notify()
- InterruptedException
- Spurious Wakeup
  - Fälschliches Aufwecken in vereinzelten Betriebssystemen (z.B. in POSIX Thread API spezifiziert)
  - Schlechtes Design: OS Implementierung vereinfacht, dafür Benutzung kompliziert

Java Monitor Funktionsweise



Signal and Continue

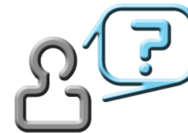
- Signalisierender Thread behält Monitor
  - Nach notify()/notifyAll() läuft er im Monitor weiter
- Aufgeweckter Thread kommt nicht direkt in Monitor
  - Signalisierender Thread ist ja noch drin
  - Kompromiss: Kommt in äusseren Warteraum

Aufgeweckter Thread muss neu um Monitor-Eintritt kämpfen wie alle anderen eintrittswilligen Threads

Java Monitor: Typische Fallen

Falsch

```
class BoundedBuffer<T> {
  private Queue<T> queue = new LinkedList<>();
  private int limit = 1; // or initialize in constructor

  public synchronized void put(T item) throws InterruptedException {
    if (queue.size() == limit) {
      wait();   // await non-full
    }
    queue.add(item);
    notify();   // signal non-empty
  }

  public synchronized T get() throws InterruptedException {
    if (queue.size() == 0) {
      wait();   // await non-empty
    }
    var item = queue.remove();
    notify();   // signal non-full
    return item;
  }
}
```



Was macht dieser Code?
Sehen Sie die Fehler?

Monitor Falle 1: Wait mit If

- Aufgeweckter Thread muss neu um Monitor-Eintritt kämpfen
  - Anderer Thread kann vorher in den Monitor und seine Bedingung invalidieren
- Eventuell auch Spurious Wakeup

Lösung: Wartebedingung in Schlaufe testen

```
while (!condition) {
  wait();
}
```

Java Monitor: Zweiter Versuch

```
class BoundedBuffer<T> {
  private Queue<T> queue = new LinkedList<>();
  private int limit = 1; // or initialize in constructor

  public synchronized void put(T item) throws InterruptedException {
    while (queue.size() == limit) {
      wait();     // await non-full
    }
    queue.add(item);
    notify();  // signal non-empty
  }

  public synchronized T get() throws InterruptedException {
    while (queue.size() == 0) {
      wait();     // await non-empty
    }
    var item = queue.remove();
    notify();  // signal non-full
    return item;
  }
}
```

Immer noch falsch

Monitor Falle 2: Single Notify

- Mehrere semantische Wartebedingungen
  - «nicht leer», «nicht voll»
- notify() weckt beliebigen Thread im Warteraum auf
  - Aufgeweckter Thread wartet evtl. auf andere Bedingung
  - Es kommt dann kein Thread dran, der weitermachen kann

Lösung: notifyAll() verwenden

```
notifyAll();
```

Java Monitor: Korrekte Version

```
class BoundedBuffer<T> {
  private Queue<T> queue = new LinkedList<>();
  private int limit = 1; // or initialize in constructor

  public synchronized void put(T item) throws InterruptedException {
    while (queue.size() == limit) {
      wait();       // await non-full
    }
    queue.add(item);
    notifyAll();    // signal non-empty
  }

  public synchronized T get() throws InterruptedException {
    while (queue.size() == 0) {
      wait();       // await non-empty
    }
    var item = queue.remove();
    notifyAll();    // signal non-full
    return item;
  }
}
```

Rückblick: Lernziele

- Notwendigkeit bestimmter Synchronisation von Threads erkennen
- Das Monitorkonzept und seine Implementierung in Java verstehen
- Den Monitor zur Synchronisation in Java einsetzen können

Weitere Lektüre (fakultativ)

- B. Goetz et al. Java Concurrency In Practice, Addison-Wesley, 2006.
  - 2: Thread Safety
- D. Lea. Concurrent Programming in Java: Design Principles and Patterns, 2nd Edition, Addison-Wesley, 2003.
  - 3.2.2 Monitor Mechanism
- Historisches Paper bei Interesse
  - C. A. R. Hoare. 1974. Monitors: An Operating System Structuring Concept. *Commun. ACM* 17(10): 549-557.

Anhang: Monitor-Fallstricke

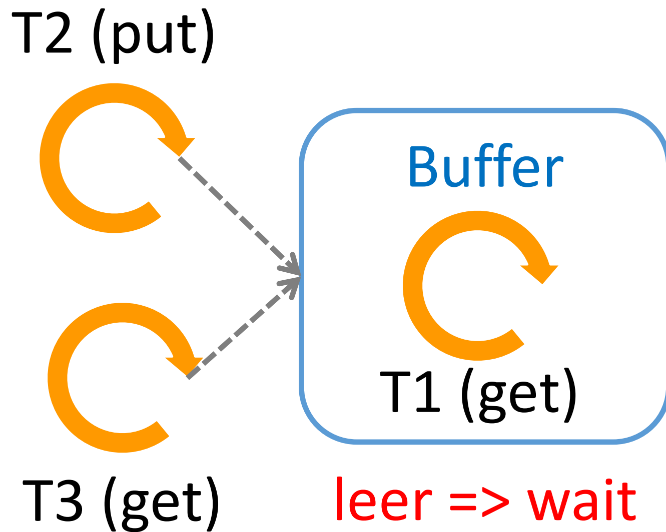
Szenarien zum Nachlesen

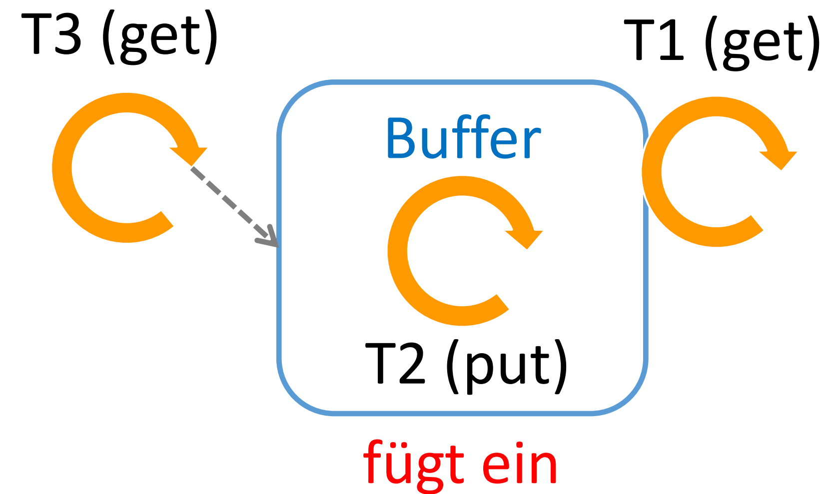
Monitor Falle 1: Überholproblem

Aktion	size()	Innerer Warteraum	Äusserer Warteraum
T1 ruft get() auf => tritt in Monitor	0		
T2 ruft put() auf => wartet auf Eintritt	0		T2
T3 ruft get() auf => wartet auf Eintritt			T2, T3
T1 wait()	0	T1	T2, T3
T2 tritt in Monitor => fügt Element ein => weckt T1	1		T3, T1
T2 verlässt Monitor	1		T3, T1
T3 tritt in Monitor => entfernt Element => verlässt Monitor	0		T1
T1 erhält Monitor Lock nach wait()	0		

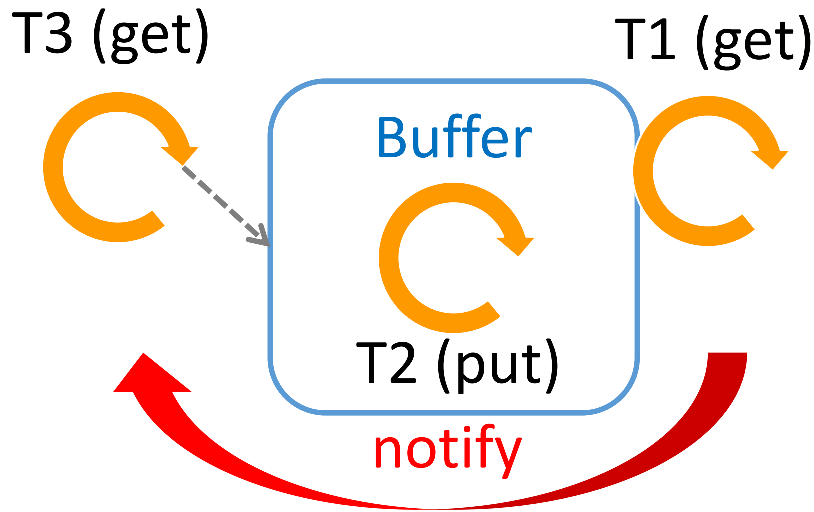
Bedingung für T1 verletzt
(T3 hat Buffer zuvor geleert)

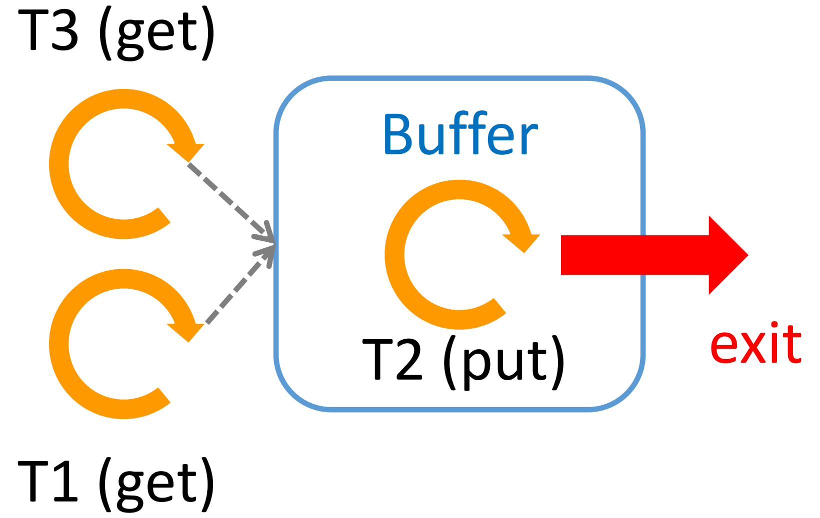
Monitor Falle 1: Szenario



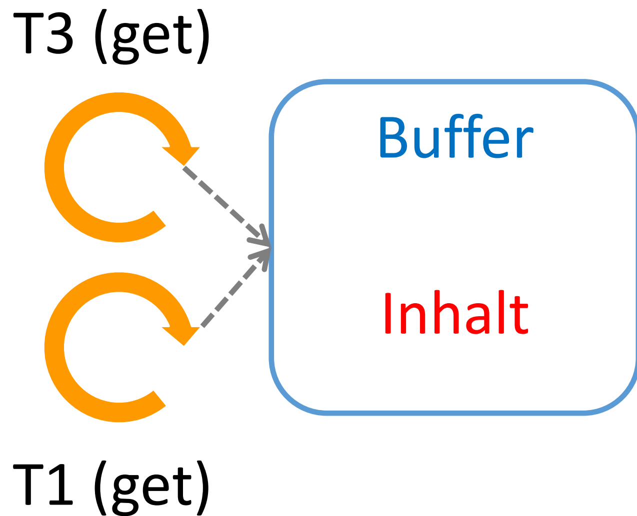


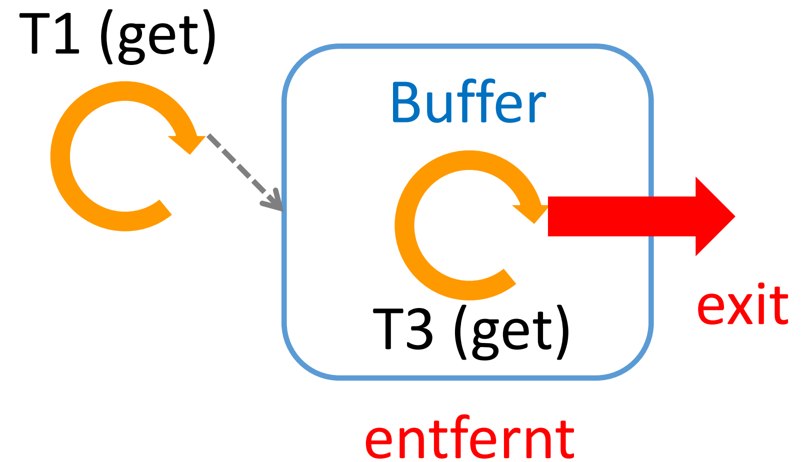
Monitor Falle 1: Szenario





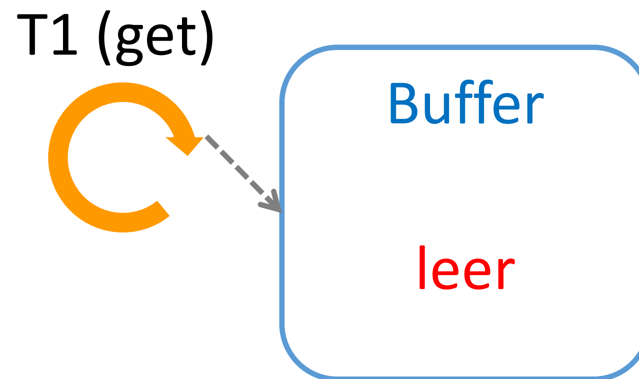
Monitor Falle 1: Szenario





T3 hat T1 überholt

Monitor Falle 1: Szenario



T1 trifft nach wait() wieder leeren Buffer an

=> Muss Bedingung nach wait() neu prüfen

Monitor Falle 2: Falscher Notify

Aktion	size()	Innerer Warteraum	Äusserer Warteraum
T1 ruft get() auf => wait()	0	T1	
T2 ruft get() auf => wait()	0	T1, T2	
T3 ruft put() auf => fügt Element ein	1	T1, T2	
T4 ruft put() auf => wartet auf Eintritt	1	T1, T2	T4
T3 weckt T1 => verlässt Monitor	1	T2	T4, T1
T4 tritt ein => wait()	1	**T2 get(), T4 put()**	T1
T1 erhält Monitor Lock nach Wakeup in get() => nimmt Element raus => Weckt T2	0	T4	T2
T2 erhält Monitor Lock => wait()	0	T4 put(), T2 get()	

Thread 4 wartet, obwohl Bedingung wahr ist => Kein Fortschritt mehr