Nebenläufige Programmierung II
und
Zusammenfassung und Ausblick

Martin Wirsing

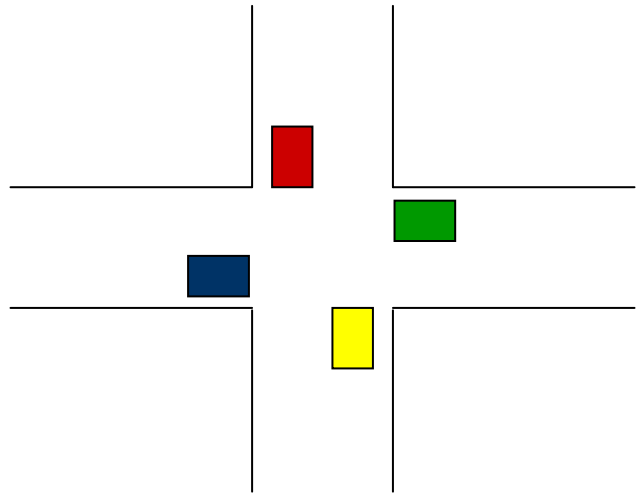
in Zusammenarbeit mit
Moritz Hammer und Axel Rauschmayer

Ziele

- Verklemmungsprobleme als mögliche Folge von Synchronisation verstehen lernen
- Kommunikation zwischen Threads durch aktive und passive Benachrichtigung verstehen lernen
- Das Standardpaket java.util.concurrent und Standardbeispiele für nebenläufige Programme wie z.B. das Erzeuger/Verbraucherproblem kennen lernen
- Wichtige Themen der Vorlesung zusammenfassen
- Einen Einblick in aktuelle Entwicklungen der Programmierung und Softwaretechnik erhalten

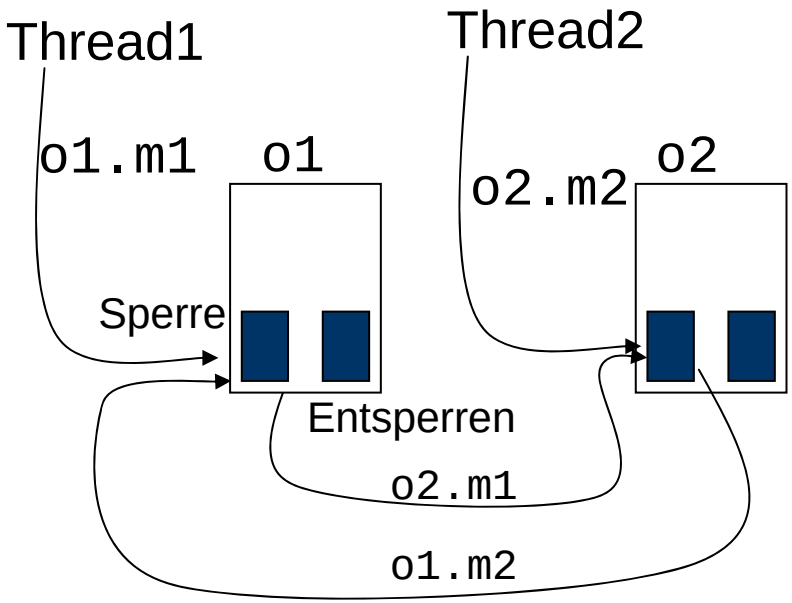
Verklemmung

Beispiel: Straßenkreuzung mit Rechts-vor-Links-Regel, bei der an jeder Straßeneinfahrt ein Auto steht:



Jedes Auto möchte die Straßenkreuzung überqueren, kann aber nicht, da es die Vorfahrt von rechts respektieren muss. Dadurch sind alle vier Autos blockiert.

Verklemmung in der Programmierung



1. Thread 1 beginnt die Ausführung der synchronisierten Methode `m1` bezüglich `o1`.
2. Thread 2 beginnt die Ausführung der synchronisierten Methode `m2` bezüglich `o2`.
3. Während der Ausführung von `o1.m1` wird von Thread 1 versucht mit der synchronisierten Methode `m1` das gesperrte Objekt `o2` aufzurufen.
4. Während der Ausführung von `o2.m2` wird von Thread 2 versucht mit der synchronisierten Methode `m2` das gesperrte Objekt `o1` aufzurufen.

Damit warten Thread 1 und Thread 2 darauf, dass jeweils der andere das gesperrte Objekt verlässt $\Rightarrow$ **Verklemmung!**

Verklemmung („Deadlock“)

Durch Synchronisation können sich **Lebendigkeitsprobleme** ergeben:
Das Programm kann in eine Verklemmung geraten.

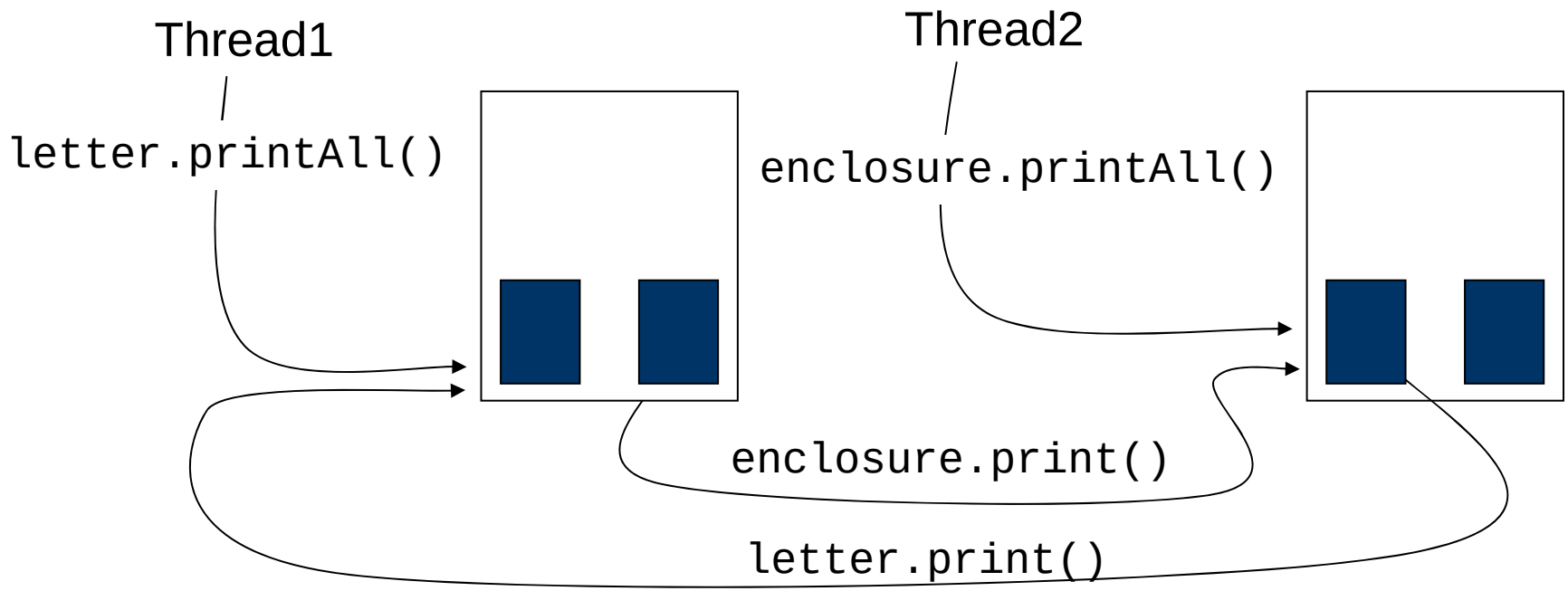
Man spricht von **Verklemmung (Deadlock)**, wenn es mindestens einen Thread eines Programms gibt, der nicht beendet ist, aber nicht weiterarbeiten kann, da die von ihm benötigten Ressourcen nicht freigegeben sind.

Typischerweise benötigt man zu einer Verklemmung mindestens zwei Threads.

Verklemmung in der Programmierung

Beispiel: Eine Dokumentenklasse, bei der Dokumente aus zwei Teilen bestehen, z.B. aus einem Brief und einer Anlage.

Falls nun `letter` ein Dokument mit Anlage `enclosure` und `enclosure` ein Dokument mit Anlage `letter` ist, ergibt sich eine Verklemmung beim Drucken dieser Dokumente mit Hilfe zweier verschiedener Threads z.B. :



Print

```
public class Document
{       Document otherPart;
        public synchronized void print()
        {       System.out.println(„first line");
                ...
                System.out.println(„last line");
        }
        public synchronized void printAll()
        {        otherPart.print();
                 print();
        }
}
```

Verklemmung in der Programmierung

Es gibt keine allgemeine Methode zur Beseitigung von Verklemmungen. Die Lebendigkeit eines Programms muss durch einen guten Entwurf gesichert werden.

Mögliche Vorgehensweisen sind folgende:

1. Top-Down-Methode (Sicherheit zuerst):
Entwurf von Methoden und Klassen mit voller Synchronisation und dann schrittweises Aufheben unnötiger Synchronisation.

2. Bottom-Up-Methode (Lebendigkeit zuerst):
Entwurf der Methoden und Klassen ohne Synchronisation und dann nachträgliches Hinzufügen von Synchronisation durch spezielle Techniken.

Kommunikation zwischen Threads

Synchronisation genügt nicht für die Kommunikation zwischen Threads und kann Verklemmungen bewirken. Wie das folgende Beispiel zeigt braucht man zusätzliche Mechanismen der Benachrichtigung.

```
public synchronized void deposit(double amount) {
  System.out.println("Depositing " + amount);
  double newBalance = balance + amount;
  System.out.println("New balance is: " + newBalance);
  balance = newBalance;
}

public synchronized void withdraw(double amount) {
  while (balance < amount)
   ; /* tue nichts */
  System.out.println("Withdrawing " + amount);
  double newBalance = balance - amount;
  System.out.println("New balance is: " + newBalance);
  balance = newBalance;}}
```

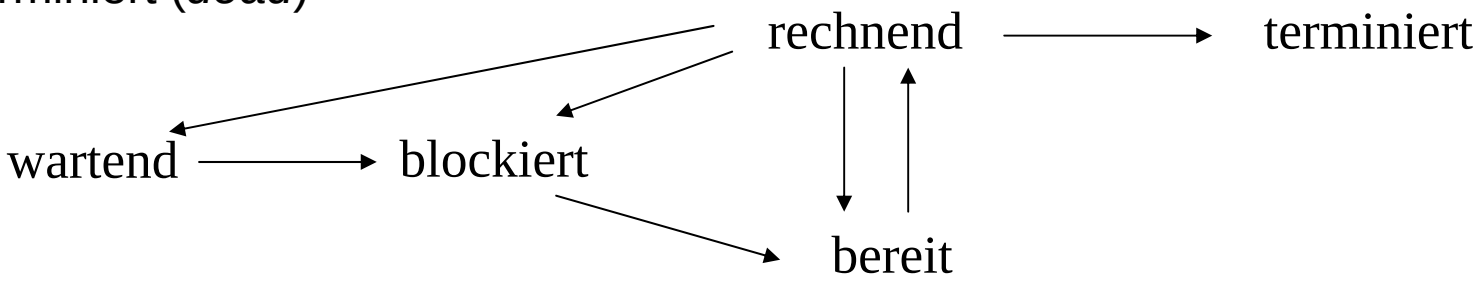
Wird die Programmzeile `/* tue nichts */` erreicht, so verklemmt sich das Gesamtsystem, da ja jeder Thread, der versucht `deposit` aufzurufen, sofort blockiert wird.

Abhilfe durch `wait` und `notifyAll`

- Jedes Objekt (also die Klasse `Object`) bietet die Methoden `wait` und `notifyAll` an.
- Wird `wait()` aufgerufen, so wird der ausführende Thread in den Wartezustand versetzt. Die Sperre des Objekts wird dadurch wieder frei.
- Wird `notifyAll()` aufgerufen, so werden alle Prozesse, die auf das Objekt warten, in den blockierten Zustand versetzt und können so bei nächster Gelegenheit die Sperre erhalten.

Wiederholung Zustandsmodell

- Jeder Thread kann einen von fünf Zuständen innehaben:
  - rechnend (*running*)
  - bereit (*ready*)
  - blockiert (*blocked*)
  - wartend (*waiting*)
  - terminiert (*dead*)



- Jedes Objekt beinhaltet
  - eine eingebaute Boole'sche Variable (für die Sperre),
  - eine Schlange von blockierten Threads und
  - eine Menge von wartenden Threads.

Außerdem gibt es eine **zentrale Schlange von *bereiten* Prozessen**.

Zustandsübergange

- Nach Ablauf einer Zeitscheibe wird ein laufender Thread “bereit” gemacht und der erste “bereite” Zustand “rechnend” gemacht.
- Beim Versuch der Ausführung einer synchronisierten Methode auf einem Objekt, dessen Sperre nicht frei ist, wird der aufrufende Thread in den Zustand “blockiert” versetzt und in die Warteschlange der durch das Objekt blockierten Threads eingereiht.
- Beim Aufruf der Methode `wait` bei einem Objekt wird der aufrufende Thread in den Wartezustand versetzt und in die Menge der wartenden Threads dieses Objekts eingefügt. Der nächste blockierte Thread wird „bereit“.
- Beim Verlassen einer synchronisierten Methode wird der nächste blockierte Thread in der zugehörigen Warteschlange “bereit” gemacht.
- Beim Aufruf von `notifyAll` bei einem Objekt werden alle “wartenden” Threads bei diesem Objekt in den Zustand “blockiert” versetzt und in die Warteschlange des Objektes eingereiht.

Warum werden wartende Threads nicht einfach blockiert? Antwort: Sonst würden sie ständig bereitgestellt und sofort wieder blockiert.

Aktive Benachrichtigung (Active Notification)

Observer Schema:

```
synchronized void doWhenCondition ()
{       while (! condition)
                try {wait();}
                catch (InterruptedException e) {...}
        // Do what is needed when condition is true
}
```

Observable Schema:

```
synchronized void changeCondition ()
{       // ... change some value used in the condition test
        notifyAll();
}
```

Beispiel

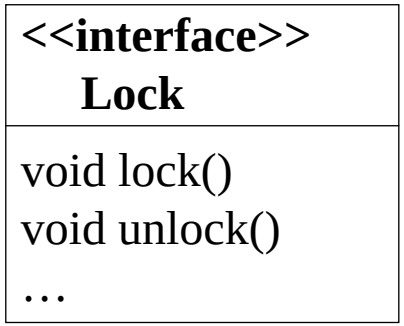
```
public synchronized void withdraw(double amount)
       throws InterruptedException {
  System.out.println("Withdrawing " + amount + " from balance = "
  + balance);
  while (balance < amount)
    try {this.wait();
    } catch (InterruptedException e) {}
  double newBalance = balance - amount;
  System.out.println("New balance is: " + newBalance);
  balance = newBalance;
}
public synchronized void deposit(double amount) {
  System.out.println("Depositing " + amount);
  double newBalance = balance + amount;
  System.out.println("New balance is: " + newBalance);
  balance = newBalance;
  this.notifyAll();
}
```

java.util.concurrent

Das (in Java 1.5) neue Paket **java.util.concurrent** stellt wichtige Hilfsklassen für nebenläufige Programmierung zur Verfügung. Insbesondere:

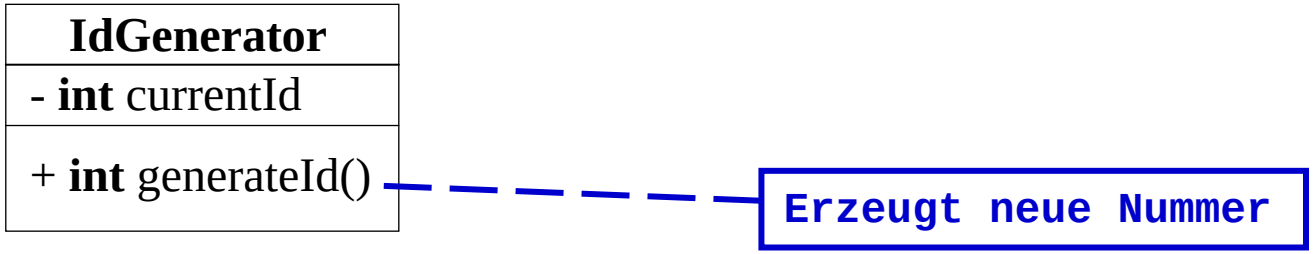
- Unterschiedliche Sperren wie **Locks** und Semaphore
- Synchronisierte Datenstrukturen wie **synchronisierte Schlangen** und Listen und
- Konstrukte zur Organisation mehrerer Threads in „Threadpools“

Lock



- Ein **`Lock (Sperre)`** `l` dient zur Zugangskontrolle einer gemeinsamen Resource und stellt typischerweise einen exklusiven Zugang zu einer Resource zur Verfügung: zu jedem Zeitpunkt kann nur ein einziger Thread die Sperre erwerben und damit auf die Resource zugreifen.
- Mit dem Aufruf **`l.lock()`** wird die Sperre `l` erworben, falls sie frei ist;
  mit **`l.unlock()`** wird sie wieder frei gegeben.
- Ein Lock ist also ein expliziter Monitor und ähnlich zu der impliziten Konstruktion eines Monitors durch `synchronized`. Locks sind aber allgemeiner und können auch über Blockgrenzen hinweg eingesetzt werden.
- Wichtige Implementierungen sind:
  - **`ReentrantLock`**, wobei mehrmaliger Zugriff des „lockbesitzenden“ Threads auf denselben Lock erlaubt ist und
  - **`ReadWriteLock`** mit exklusivem Schreibezugriff, aber mehreren erlaubten Leseoperationen

Beispiel: Generator für eindeutige Nummern



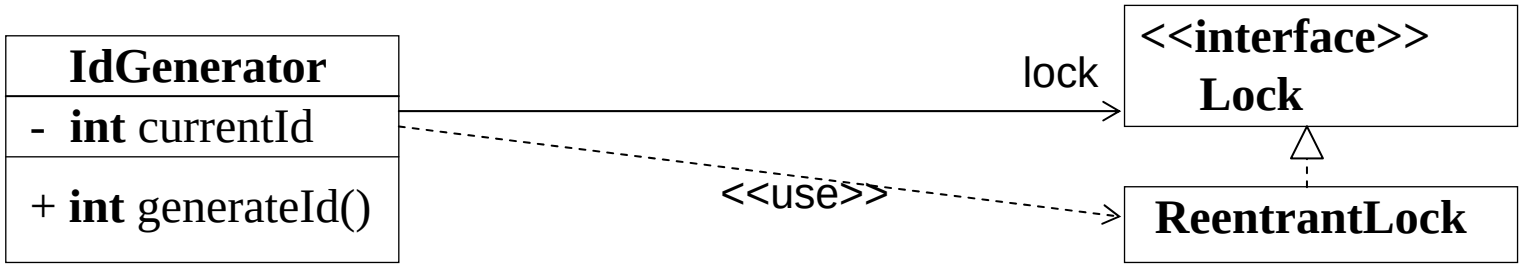
Lösung mit synchronized

```
public class IdGenerator {
  private int currentId;
  public IdGenerator() { this.currentId = 0;}

  public synchronized int generateId1() {
        this.currentId++;
        return this.currentId;
  }
  public int generateId2() {
        synchronized (this) {
            this.currentId++;
            return this.currentId;}
  }
```

synchronized bewirkt, dass diese beiden Anweisungen als atomar behandelt werden; andernfalls könnte es zu Problemen kommen.

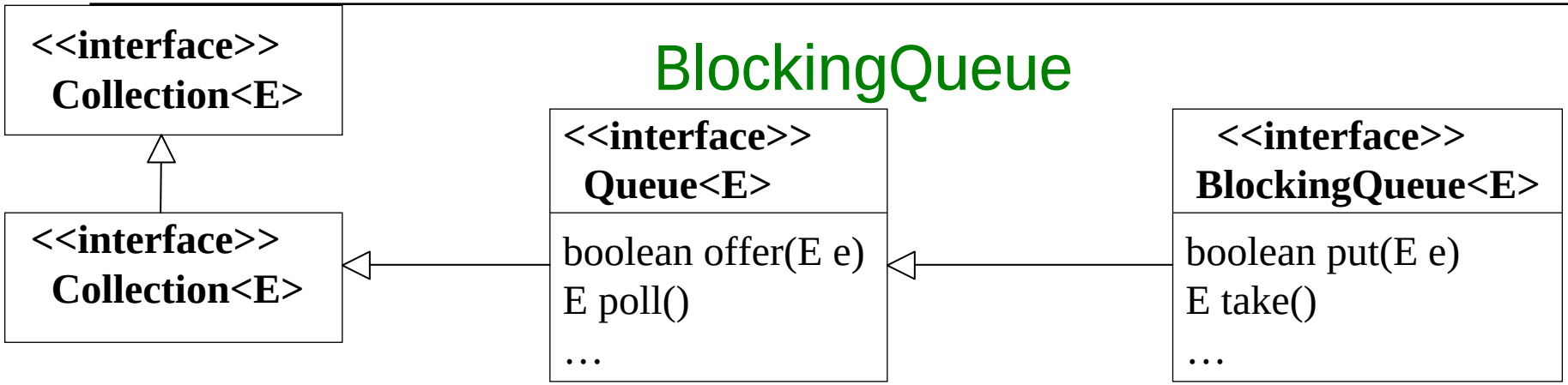
Beispiel: Generator für eindeutige Nummern



Lösung mit Lock

```
import java.util.concurrent.locks.*;
. . .
private Lock lock = new ReentrantLock();
    public int generateId3() {
        this.lock.lock();
        try {
            this.currentId++;
            return this.currentId;
        } finally {  // Vergiss auf keinen Fall, wieder aufzusperren:
            this.lock.unlock();
        }
```

BlockingQueue



- Eine **`Queue (Schlange)`** ist eine listenartige Struktur mit der Operation **`offer`** zum Einfügen am Ende der Schlange und **`poll`** zum Wegnehmen des ersten Elements der Schlange.
- Eine **`BlockingQueue`** ist eine Schlange, deren Operationen „thread-sicher“ sind, d.h. dass Einfügen, Löschen etc. atomar erfolgen.
- Zusätzlich wird angeboten
  - eine Einfüge-Operationen **`put`**, die auf freien Platz wartet, wenn die Schlange voll ist; und
  - eine Löschoperation **`take`**, die wartet, bis die Schlange nichtleer ist.

Wichtige Implementierungen sind:

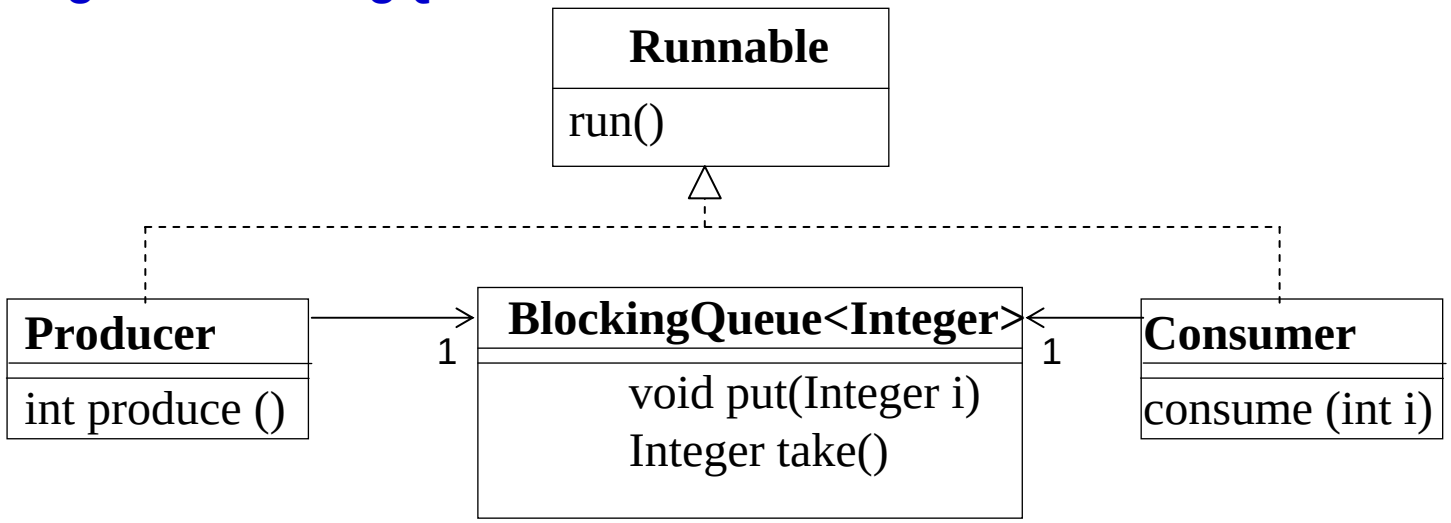
- **`ArrayBlockingQueue<E>`** basierend auf Reihungen
- **`LinkedBlockingQueue<E>`** basierend auf verketteten Listen

Beispiel: Erzeuger/Verbraucher Problem

Beim **Erzeuger/Verbraucher Problem** gibt es eine synchronisierte Schlange, einen Erzeuger und einen Verbraucher.

Der **Erzeuger** produziert unaufhörlich Dinge (hier Zahlen) und schreibt sie in die Schlange; der **Verbraucher** liest unaufhörlich Elemente aus der Schlange.

- **Lösung mit BlockingQueue**



Erzeuger

```
public class Producer implements Runnable {
  int number;
  private final BlockingQueue<Integer> queue;

  Producer(BlockingQueue<Integer> q, int i) {
    queue = q; number = i; }
  public void run() {
    try {
    int i = 0;
    while (true) queue.put(produce(i++));
    } catch (InterruptedException ex) {}
  }
  synchronized Integer produce(int i) {
    System.out.format("Producer #%d put: %d%n", number, i);
    return i;}
}
```

Verbraucher

```
public class Consumer implements Runnable {
  int number;
  private final BlockingQueue<Integer> queue;

  Consumer(BlockingQueue<Integer> q, int i) {
    queue = q; number = i; }
  public void run() {
    try {
      while (true) consume(queue.take());
    } catch (InterruptedException ex) {}
  }
  synchronized Integer consume(int i) {
    System.out.format(„Consumer #%d took: %d%n", number, i);
    return i;}
}
```

Zusammenfassung: Nebenläufigkeit in Java

- Java unterstützt Nebenläufigkeit durch **„leichtgewichtige“ Prozesse**, sogenannte **Threads**, die über die gemeinsame Halde und damit über gemeinsam benutzte Objekte miteinander kommunizieren.
- Nebenläufigkeit wirft **Sicherheits- und Lebendigkeitsprobleme** auf. Gemeinsam benutzte Objekte müssen **synchronisiert** werden. Zuviel Synchronisation kann **Verklemmungen** hervorrufen.
- Das (in Java 1.5) neue Paket **java.util.concurrent** stellt wichtige Hilfsklassen für nebenläufige Programmierung zur Verfügung. Insbesondere sind das **Sperren, synchronisierte Datenstrukturen** und Konstrukte zur **Organisation mehrerer Threads**.

Zusammenfassung: Objektorientierte Programmierung und Software-Entwicklung

- Ein **Objekt** besitzt
  - lokalen Zustand (Attribute)
  - Methoden zur Änderung des Zustandes
- **Modulare Programmierung** durch Bildung von Klassen, Schnittstellen, Kapselung
- Unterstützung von Wiederverwendung und Änderung von Programmen durch **Vererbung und generische Typen**
- Robuste Programmierung durch **selbstdefinierte Ausnahmen**
- **Nebenläufige Programmierung**
- Systematische OO-SW-Entwicklung durch **UML und Entwurfsmuster**

Zusammenfassung: Grundlagen der Informatik

- Parameterübergabemechanismen
- Korrektheit und Spezifikation von Methoden
- Komplexität
- Grundlegende dynamische Datenstrukturen

Ausblick: Neue Entwicklungen in Programmiersprachen

Neue Entwicklungen in Programmiersprachen

- **Dienst-orientierte Programmierung**
  - **Web-Dienste (Web Services)** wichtigste Neuentwicklung der letzten Jahre mit sehr hohen Zuwachsraten in der Industrie
  - **Services** bilden die **Basiskomponenten zur dynamischen Konstruktion von verteilten Systemen => EU-Projekt SENSORIA** (MW)
- **Aspekt-orientierte Progammierung** (AspectJ, HyperJ, ...)
  - „Einweben“ von Aspekten: z.B. Logging, Zusicherungen, Synchronisation beim Einbetten sequentieller in nebenläufige Programme
- **Mobile Systeme (Ambients)**
  - Mobile Computing (mobile Hardware)
  - Mobile Computation (mobiler Code)

Ausblick: Neue Entwicklungen im Software Engineering

- **UML**
  - Extreme Programming & UML
  - Model-driven Architecture
    - Generieren von Code aus den Modellen
    - Modelchecking von Software Entwürfen
  - Sicherheit im Internet => DFG-Projekt (Dr. Knapp)
- **Software-Engineering für mobile Systeme**
  - EU-Projekt Agile Projekt (MW)
- **Web-Engineering**
  - Entwurf von Web-basierten Systemen => DFG-Projekt MAEWA (MW)
  - ICWE 04 in München an der LMU

Ausblick: Lehre

- Programmierpraktikum: Entwicklung eines verteilten Spiels
  - Nebenläufige Programmierung
  - Graphische Benutzeroberfläche
  - Teams mit 4-5 Mitgliedern
- Systempraktikum
- Informatik III
- Methoden des Software-Engineering (5.Semester)
  Requirements Engineering, SW-Architektur, Prozessmodellierung, Testen und Validierung, Web-Engineering
- Grundlagen der Systementwicklung (7.Semester)
- Modelchecking-Praktikum
- Objekt-orientierte SW-Entwicklung

Vielen Dank für Ihre Aufmerksamkeit und Mitarbeit!

Moritz Hammer, Axel Rauschmayer und ich wünschen Ihnen

eine **angenehme**

vorlesungsfreie Zeit!

Auf Wiedersehen!