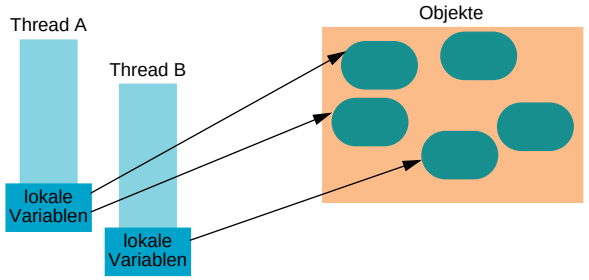
C Threads

- Referenz:
  - D. Lea. *Concurrent Programming in Java - Design Principles and Patterns.* The Java Series. Addison-Wesley 1999.
  - Scott Oaks; Henry Wong - *Java Threads, 3rd Edition* - 2004
    `http://proquestcombo.safaribooksonline.com/0596007825`
  - John Ousterhout: *Why Threads Are A Bad Idea (for most purposes).*

C.1 Was ist ein Thread?

- Aktivitätsträger mit eigenem Ausführungskontext:
  - Instruktionszähler, Register und Stack
- Alle Threads laufen im gleichen Adressbereich



C.2 Vorteile / Nachteile

- Vorteile:
  - Ausführen paralleler Algorithmen auf einem Multiprozessorrechner
  - durch das Warten auf langsame Geräte (z.B. Netzwerk, Benutzer) wird nicht das gesamte Programm blockiert
- Nachteile:
  - komplexe Semantik
  - Fehlersuche schwierig

C.3 Thread Erzeugung: Möglichkeit 1

(1) Eine Unterklasse von **`java.lang.Thread`** erstellen.
  - Threads können zur besseren Fehlersuche benannt werden.

(2) Dabei die **`run()`**-Methode überschreiben.
(3) Eine Instanz der Klasse erzeugen.
(4) An dieser Instanz die Methode **`start()`** aufrufen.

- Beispiel:

```
class Test extends Thread {
    public Test(String name) {
        super(name);
    }
    public void run() {
        System.out.println("Test");
    }
}
```

C.4 Thread Erzeugung: Möglichkeit 2

(1) Das Interface **`java.lang.Runnable`** implementieren.
Dabei muss eine **`run()`**-Methode erstellt werden.
(2) Ein Objekt instantiieren, welches das Interface **`Runnable`** implementiert.
(3) Eine neue Instanz von **`Thread`** erzeugen, dem Konstruktor dabei das **`Runnable`**-Objekt mitgeben.
(4) Am neuen Thread-Objekt die **`start()`**-Methode aufrufen.

- Beispiel:

```
class Test implements Runnable {
    public void run() {
        System.out.println("Test");
    }
}

Test test = new Test();
Thread thread = new Thread(test);
thread.start();
```


C.5 Die Methode `sleep`

- Ein Thread hat die Methode **`sleep(long n)`** um für **`n`** Millisekunden zu "schlafen".
- Der Thread kann jedoch verdrängt worden sein nachdem er aus dem **`sleep()`** zurückkehrt.

C.6 Die Methode `join`

- Ein Thread kann auf die Beendigung eines anderen Threads warten:

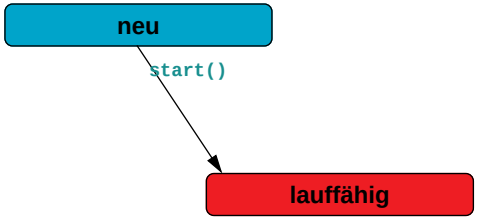
```
workerThread = new Thread(worker);
...
workerThread.join();
worker.result();
```


C.7 Daemon-Threads vs. User Threads

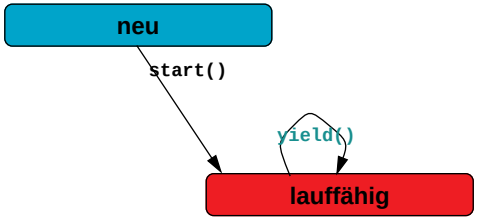
- Daemon-Threads werden für Hintergrundaktivitäten genutzt
- Sie sollen nicht für die Hauptaufgabe eines Programmes verwendet werden
- Sobald alle *nicht-daemon* Threads beendet sind, ist auch das Programm beendet.
- Woran erkennt man, ob ein Thread ein Daemon-Thread sein soll?
  - Wenn man keine Bedingung für die Beendigung des Threads angeben kann.
- Wichtige Methoden der Klasse **`Thread`**:
  - **`setDaemon(boolean switch)`**: Ein- oder Ausschalten der Daemon-Eigenschaft (nur vor dem Aufruf von **`start()`**).
  - **`boolean isDaemon()`**: Prüft ob ein Thread ein Daemon ist.

C.8 Zustände von Threads

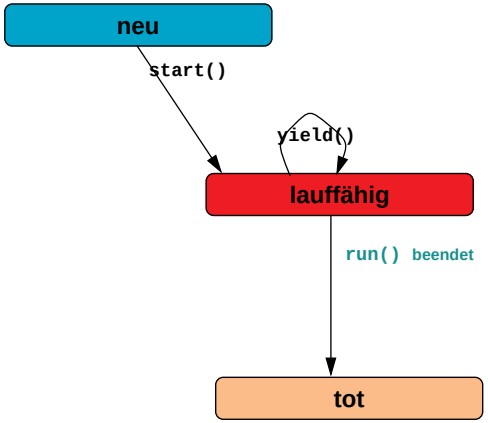
C.8 Zustände von Threads (2)



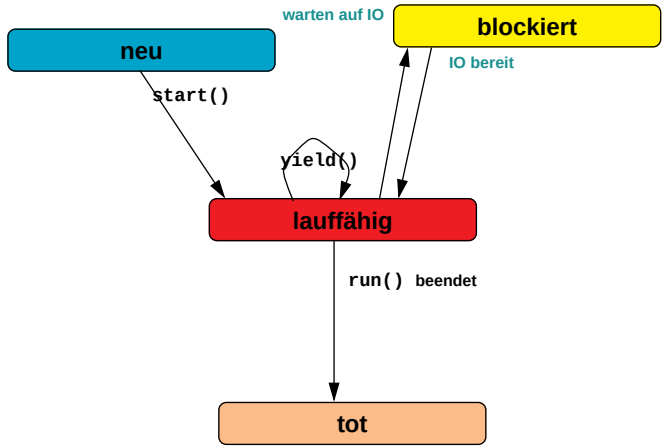
C.8 Zustände von Threads (3)



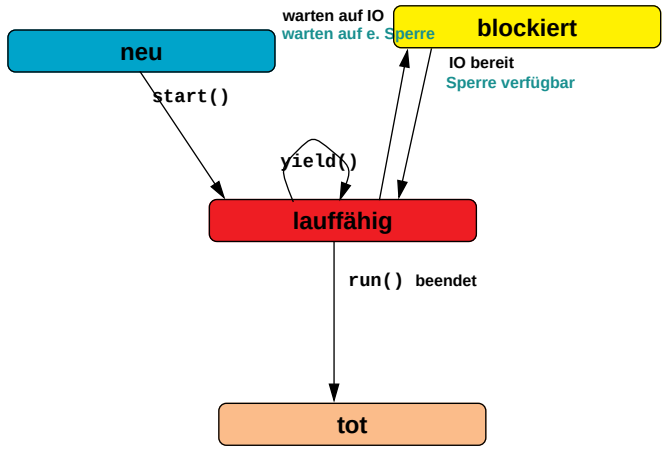
C.8 Zustände von Threads (4)



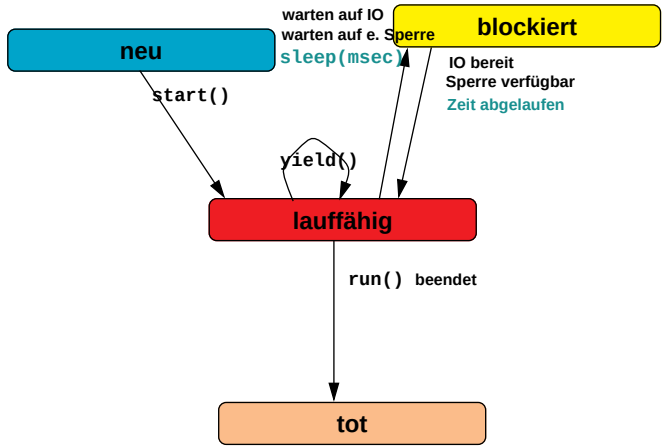
C.8 Zustände von Threads (5)



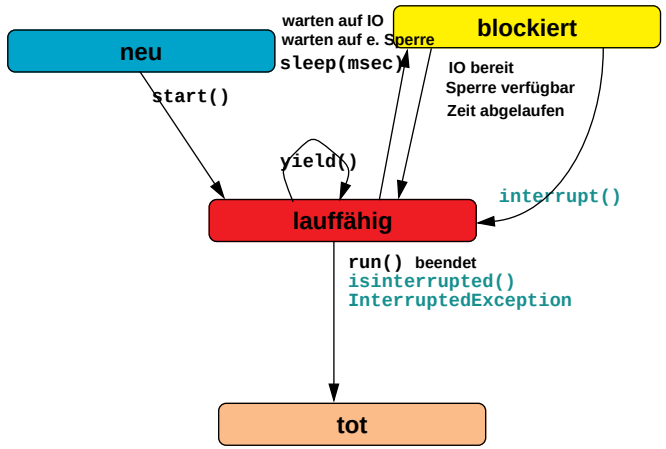
C.8 Zustände von Threads (6)



C.8 Zustände von Threads (7)



C.8 Zustände von Threads (8)



C.9 Veraltete Methoden der Klasse Thread

- **`stop()`**, **`suspend()`**, **`resume()`** sind seit Java 1.2 veraltet
- **`stop()`** gibt alle Sperren des Thread frei - kann zu Inkonsistenzen führen
- **`suspend()`** und **`resume()`** können zu einem Deadlock führen:
  - suspend gibt keine Sperren frei
  - angehaltener Thread kann Sperren halten
  - Thread, der **`resume()`** aufrufen will blockiert an einer Sperre
- **`destroy()`** gibt keine Sperren frei
- **`interrupt()`** wird verwendet um einen Thread kontrolliert zu beenden
  - Es kann **`isInterrupted()`** oder eine Variable genutzt werden
  - Ist der Thread blockiert wird eine Exception ausgelöst

C.10 Multithreading Probleme

```
public class Adder implements Runnable {
  public int a=0;
  public void run() {
    for(int i=0; i<100000; i++) {
      a = a + 1;
    }
  }

  public static void main(String[] args) {
    Runnable value = new Adder();
    Thread t1 = new Thread(value);
    Thread t2 = new Thread(value);
    t1.start();
    t2.start();
    try {
      t1.join();
      t2.join();
    } catch(Exception e) {
      System.out.println("Exception");
    }
    System.out.println("Expected a=200000 but a="+value.a);
  }
}
```

Was ist das Ergebnis dieses Programmes?

zwei Threads erzeugen, mit demselben `Runnable` Objekt

beide Threads starten

auf Beendigung der beiden Threads warten

C.10 Multithreading Probleme (2)

- Ergebnis einiger Durchläufe: 173274, 137807, 150683
- Was passiert, wenn **`a = a + 1`** ausgeführt wird?

```
LOAD a into Register
ADD 1 to Register
STORE Register into a
```

- mögliche Verzahnung wenn zwei Threads beteiligt sind (initial a=0):
  - T1-load:a=0,Reg1=0
  - T2-load:a=0,Reg2=0
  - T1-add:a=0,Reg1=1
  - T1-store:a=1,Reg1=1
  - T2-add:a=1,Reg2=1
  - T2-store:a=1,Reg2=1
- Die drei Operationen müssen *atomar* ausgeführt werden!

C.11 Das Schlüsselwort `synchronized`

- Jedes Objekt kann als Sperre verwendet werden.
- Um eine Sperren anzufordern und freizugeben wird ein **`synchronized`** Konstrukt verwendet.
- Methoden oder Blöcke können als **`synchronized`** deklariert werden:

```
class Test {
       public void n() { ...
              synchronized(this) {
                  ...
              }
       }
}
```

- ein Thread kann eine Sperre mehrfach halten (rekursive Sperre)
- verbessertes Beispiel: **`synchronized(this) { a = a + 1; }`**

C.11 Das Schlüsselwort `synchronized`

- Vereinfachte Schreibweise
  - Anstatt:

```
class Test {
        public void n() {
                synchronized(this) {
                        ...
                }
        }
}
```

  - kann auch folgendes geschrieben werden:

```
class Test {
        public synchronized void n() {
                ...
        }
}
```


C.11 Wann soll `synchronized` verwendet werden?

- **`synchronized`** ist nicht notwendig:
  - wenn Code immer nur von einem Thread ausgeführt wird (single-threaded context)
    - private Methoden, die nur auf Objektvariablen zugreifen
    - ...
  - für "getter" von einfachen Datentypen
- **`synchronized`** sollte verwendet werden:
  - für "getter" auf **`long`**, etc.
  - wenn der Zustand eines Objekts aus mehreren Komponenten zusammengesetzt ist
  - Beim Zugriff auf globale Objekte

C.12 Synchronisationsvariablen (Condition Variables)

- Thread muss warten bis eine Bedingung wahr wird.
- zwei Möglichkeiten:
  - aktiv (polling)
  - passiv (condition variables)
- Jedes Objekt kann als Synchronisationsvariable verwendet werden.
- Die Klasse **`Object`** enthält Methoden um ein Objekt als Synchronisationsvariable zu verwenden.
  - **`wait`**: auf ein Ereignis warten

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

  - **`notify`**: Zustand wurde verändert, die Bedingung könnte wahr sein, einen anderen Thread benachrichtigen
  - **`notifyAll`**: alle wartenden Threads aufwecken (teuer)

C.12 Warten und Sperren

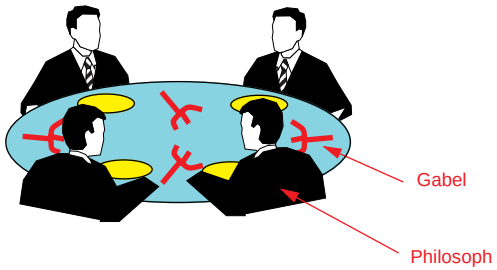
- **`wait()`** kann nur ausgeführt werden, wenn der aufrufende Thread eine Sperre an dem Objekt hält (z.B. **`synchronized()`**-Block)
- **`wait()`** gibt die Sperre frei bevor der Thread blockiert wird (atomar)
- beim Deblockieren wird die Sperre wieder atomar angefordert

C.12Condition Variables - Beispiel

- PV-System: Bedingung: *count > 0*

```
class SimpleSemaphore {
    private int count;
    public SimpleSemaphore(int count) { this.count = count; }
    public synchronized void P() throws InterruptedException{
        while (count <= 0) {
            this.wait();
        }
        count--;
    }
    public synchronized void V() {
        count++;
        this.notify();
    }
}
```


C.13 Deadlock: Das Philosophenproblem



denken → essen → denken → essen → ...

- ein Philosoph braucht beide Gabeln zum Essen
- alle Philosophen nehmen zuerst die rechte Gabel dann die linke → Verklemmung

C.14 Vorraussetzungen für Verklemmungen

- Wiederholung aus dem SOS1 Skript
- notwendige Bedinungen:
  - ◆ exclusive Belegung von Betriebsmitteln (mutual exclusion)
    - die umstrittenen Betriebsmittel sind nur unteilbar nutzbar
  - ◆ Nachforderung von Betriebsmitteln (hold and wait)
    - die umstrittenen Betriebsmittel sind nur schrittweise belegbar
  - ◆ kein Entzug von Betriebsmitteln (no preemption)
    - die umstrittenen Betriebsmittel sind nicht rückforderbar
- hinreichende Bedingung
  - ◆ zirkuläres Warten (circular wait)
    - Existenz einer geschlossenen Kette wechselseitig wartender Prozesse

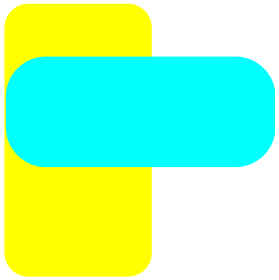
C.15 Deadlocks

- Counter

```
class Counter {
    private int count = 0;

    public synchronized void inc() { count++; }

    public int getCount() {return count; }

    public void setCount(int count) { this.count = count; }

    public synchronized void swap(Counter counter) {
        synchronized (counter) { // Deadlock Gefahr
            int tmp = counter.getCount();
            counter.setCount(count);
            count = tmp;
        }
    }
}
```

C.15Ablauf

- Wie kommt es zum Deadlock?



C.16 Deadlock - Vermeidung(1)

- Verhinderung zyklischer Ressourcenanforderung, Ordnung auf Locks

```
class Counter {
  ...
  public void swap(Counter counter) {
    Counter first = this;
    Counter second = counter;
    if (System.identityHashCode(this)
        < System.identityHashCode(counter)) {
      first = counter;
      second = this;
    }
    synchronized (first) {
      synchronized (second) {
        int tmp = counter.getCount();
        counter.setCount(count);
        count = tmp;
      }
    }
  }
```


C.16Deadlock Vermeidung (2)

- Ressourcen (Locks) werden atomar angefordert:

```
class Counter {
  ...
  static Object lock = new Object();
  public void swap(Counter counter) {
    synchronized (lock) {
      synchronized(this) {
        synchronized (counter) {
          int tmp = counter.getCount();
          counter.setCount(count);
          count = tmp;
        }
      }
    }
  }
}
```


C.17 Nachteile von **`synchronized()`**-Locks

- Methoden **`lock()`** und **`unlock()`** von **`synchronized()`**-Locks sind nur im Java Bytecode sichtbar
- kein Timeout beim Warten auf ein Lock möglich (Deadlock-Erkennung)
- Es kann keine alternative Semantik (z.B. zur Implementierung von fairness) definiert werden.

C.18 Explizite Locks mit `java.util.concurrent.locks`

- Allgemeines Interface: **`Lock`**
- Sperre anfordern
  - **`void lock()`**
  - **`void lockInterruptibly() throws InterruptedException`**
  - **`boolean tryLock()`**
  - **`boolean tryLock(long time, TimeUnit unit) throws InterruptedException`**
- Sperre freigeben:
  - **`void unlock()`**
- "Condition"-Variable für diese Sperre erzeugen
  - **`Condition newCondition()`**

C.19 ReentrantLock Beispiel

```
class Counter {
  ...
  static ReentrantLock lock = new ReentrantLock();
  public void swap(Counter counter) {
    lock.lock();
    try{
      synchronized(this) {
        synchronized (counter) {
          int tmp = counter.getCount();
          counter.setCount(count);
          count = tmp;
        }
      }
    } finally {
    lock.unlock();
    }
  }
}
```


C.18Lock - Implementierungen

- **`ReentrantLock`** - eine Implementierungen von **`Lock`**
  - Sehr ähnliche Semantik wie **`synchronized()`**-Locks
  - Wesentlich bessere Performanz bei heftiger Benutzung
- enthält weitere sinnvolle Methoden:
- Wer hält die Sperre?
  - **`Thread getOwner()`**
- Wer wartet auf die Sperre?
  - **`Collection<Thread> getQueuedThreads()`**
- Wer wartet auf eine Bedingung?
  - **`Collection<Thread> getWaitingThreads(Condition c)`**

C.20 Condition

- Auf Signal warten
  - **`void await() throws InterruptedException`**
    - analog zu **`Object.wait()`**
  - **`void awaitUninterruptibly()`**
  - **`boolean await(long time, TimeUnit unit) throws InterruptedException`**
  - **`boolean awaitUntil(Date deadline) throws InterruptedException`**
- Signalisieren
  - **`void signal()`**
    - analog zu **`Object.notify()`**
  - **`void signalAll()`**
    - analog zu **`Object.notifyAll()`**

C.21 SimpleSemaphore revisited

```
public class SimpleSemaphore {
   private int count;
   private ReentrantLock lock;
   private Condition c;

   public SimpleSemaphore (int count) {
       this.lock = new ReentrantLock();
       this.c = lock.newCondition();
       this.count = count;
   }

    ...
}
```


C.21SimpleSemaphore revisited(2)

```
public class SimpleSemaphore {
    ...
     public void P() throws InterruptedException {
       this.lock.lock();
       while (count <= 0) {
           this.c.await();
       }
       this.count--;
       this.lock.unlock();
   }
   public void V() {
       this.lock.lock();
       this.count++;
       this.c.signal();
       this.lock.unlock();
   }
}
```


C.22 synchronized vs. ReentrantLock()

- **`ReentrantLock`**
  - Mehr Features (timed wait, interruptable waits, ...)
  - Performanter
  - nicht an Codeblöcke gebunden
  - schwierig zu Debuggen
- **`synchronized()`**
  - Viel Code benutzt ihn
  - JVM kann beim Debuggen helfen
  - einfach*er* zu benutzen
  - man kann kein lock.unlock() vergessen

C.23 Die Klasse `java.util.concurrent.Semaphore`

- Ein zählender Semaphor:
  - `Semaphore(int permits, [boolean fair])`
- Belegen / Freigeben:
  - `acquire([int permits]) throws InterruptedException`
  - `acquireUninterruptibly([int permits])`
  - `tryAcquire([int permits, ] [long timeout])`
  - `release([int permits])`

C.24 Thread Managment mit `java.util.concurrent`

- Einheitliche Schnittstelle:

```
// Anstatt:
new Thread(new(RunnableTask())).start()

// Per Executor Interface:
Executor executor = anExecutor;
executor.execute(new RunnableTask1());
executor.execute(new RunnableTask2());
```

- Mögliche Implementierungen:
  - Direkte, synchrone Ausführung
  - Im Hintergrund hintereinander
  - Im Hintergrund paralell
  - ...

C.25 Die Fabrik `Executors`

- Klasse **`Executors`** mit statischen Methoden:
  - **`static ThreadFactory defaultThreadFactory()`**
    - Kann als optionaler Parameter den folgenden Factorys angegeben werden
  - **`static ExecutorService newSingleThreadExecutor()`**
    - Arbeitet "Tasks" nacheinander ab.
    - Implementiert eine "worker queue"
  - **`static ExecutorService newCachedThreadPool()`**
    - Erzeugt bei Bedarf neue Threads
    - Verwendet ggf. Ausführungskontext von beendeten Threads wieder
  - **`static ExecutorService newFixedThreadPool(int nThreads)`**
    - Erzeugt im vorraus n Threads

C.25Subinterface `ExecutorService`

- Ein **`Executor`** zur asynchronen Methodenausführung
- **`Runnable`** mit Rückgabe und Exception: **`Callable<V>`**
  - **`V call() throws Exception`**
- Ausführung Benutzung durch die Klasse **`ExecutorService`**
  - **`<T> Future<T> submit(Callable<T> task)`**
  - **`Future<?> submit(Runnable task)`**
- Ergebnis als Platzhalter: **`Future<V>`**
  - **`boolean cancel(boolean mayInterruptIfRunning)`**
  - **`V get()`**
  - **`V get(long timeout, TimeUnit unit)`**
  - **`boolean isDone()`**