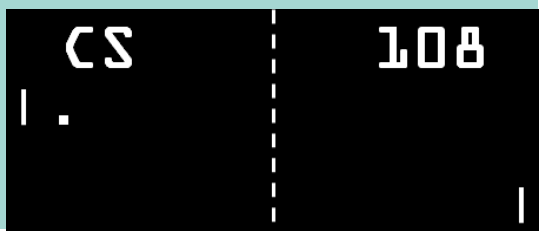
University of Basel

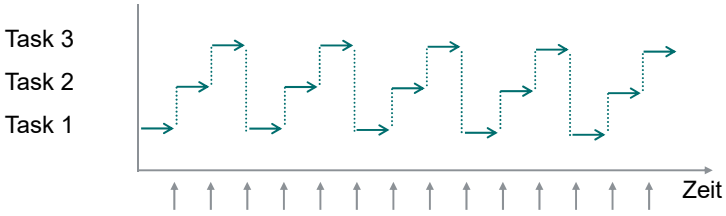
Kapitel 4 – Multithreading
cs108 Programmierprojekt

H. Schuldt, FS 2020



Parallele Tasks (Concurrent Programming)

- Multicore-Computer: mehrere CPUs, die parallel Anwendungen ausführen können (eine pro CPU)
- Parallelität innerhalb einer CPU: Kurze Wiederholung aus CS101 Grundlagen der Programmierung



- Multitasking in modernen Betriebssystemen:
  - es laufen (in jeder CPU) mehrere quasi-parallele Tasks
- In der Realität wird jedem Task eine kurze Zeitspanne zur Verfügung gestellt und das Betriebssystem wechselt danach automatisch zum nächsten Task

Tasks – Prozesse und Threads

Es gibt zwei Arten, diese Nebenläufigkeit zu realisieren:

- Prozesse sind Instrumente zur Ausführung von kompletten eigenständigen Programmen. Die Adressräume unterschiedlicher paralleler Prozesse sind streng voneinander getrennt
- Threads können innerhalb eines Prozesses parallel zur Ausführung von Programmfragmenten laufen. Alle Threads teilen sich einen gemeinsamen Adressraum und können somit auf alle Variablen zugreifen. Der Laufzeit-Overhead zur Erzeugung und Verwaltung von Threads ist relativ gering

Multitasking kann somit durch parallele Prozesse oder parallele Threads erreicht werden

Wo liegt der Vorteil von Threads?

Die Benutzung von Threads hat zwei wesentliche Vorteile gegenüber der Erzeugung von parallelen Prozessen:

- Man benötigt keine Interprozess-Kommunikation (IPC), da alle Threads einen gemeinsamen Adressraum besitzen
- Das Erzeugen und die Verwaltung von Prozessen kann im Gegensatz zu Threads relativ gross sein
- Auf Multiprocessing-Architekturen kann Multithreading die Ausführungszeit signifikant reduzieren
- Das Konzept des Multithreading ist in Java in der Sprache implementiert

Nachteile von Multithreading

- Komplexität: Multithreading stellt Anforderungen an den Nutzer
- Determinismus: Multithreading muss nicht immer identische Ergebnisse liefern

Weitere Nachteile:

- Synchronisations-Overhead durch Zugriff auf gemeinsame Ressourcen
- CPU-Overhead durch Thread-Verwaltung
- Es kann zu Thread-bedingten Fehlern kommen, die schwer zu finden sind

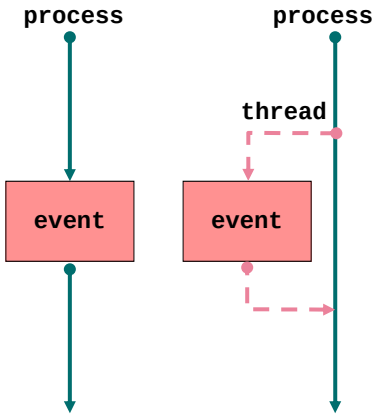
Threads sollten also vorsichtig eingesetzt werden!

Wann sollte man Threads benutzen?

- Wenn ein einzelner Task auf ein Ereignis oder auf Ressourcen warten muss, die nicht für den Rest des Programms benötigt werden

Beispiele:

- Aufteilen von I/O-Subtasks vom Haupttask (z.B. Editieren eines Textes und gleichzeitiges Speichern im Hintergrund in Textverarbeitungs-programmen)
- CPU-intensives Vorverarbeiten (pre-processing) von GUI-Daten
- …



Erzeugung von Threads – Einstieg

- Vererbung von `java.lang.Thread`
- Implementierung der `run()`-Methode
- Starten des Threads mit der `start()`-Methode

Im Projekt:
Interface `java.lang.Runnable` implementieren (siehe 4-17 ff.)

```
public class SimpleThread extends Thread {
    private int countDown = 5;
    private static int threadCount = 0;
    public  SimpleThread() {
        super("" + ++threadCount); // Store the thread name
        start();                   // Start at construction
    }
    public String toString() {
        return "#" + getName() + ": " + countDown;
    }
    public void run() {
        while(true) {
            System.out.println(this);
            if(--countDown == 0)  // Break condition
                return;
        }
    }
}
```

```
public static void main(String[] args) {
        for (int i = 0; i < 5; i++) new SimpleThread(); }
```


Thread-Reihenfolge

Die Abarbeitungsreihenfolge von mehreren Threads wird durch das Scheduling der Java Virtual Machine (JVM) bestimmt

Die Klasse **`Thread`** besitzt Methoden, um dem JVM-Scheduling Vorschläge zu machen:

- `yield()`: Vorschlag zu einem Thread-Wechsel
- `sleep()`: legt den Thread in einen sleep Modus
- `setPriority()`: ändert die Priorität des Threads
- `join()`: wartet bis ein anderer Thread beendet ist

Yielding

- In der `run()`-Methode kann man dem Scheduler einen Hinweis durch `yield()` geben, dass dieser zu einem anderen Thread wechseln soll
- Dies ist jedoch nur ein Hinweis: es existiert keine Garantie, dass der Scheduler der JVM wirklich wechseln wird

```
public class YieldingThread extends Thread {
    private int countDown = 5; private static int threadCount = 0;
    public  YieldingThread() {
        super("" + ++threadCount); // Store the thread name
        start();                   // Start at construction
    }
    public String toString() {
        return "#" + getName() + ": " + countDown;
    }
    public void run() {
        while(true) {
            System.out.println(this);
            if(--countDown == 0)  // Break condition
                return;
            yield();              // Switch thread (maybe)
        }
}
```

```
public static void main(String[] args) {
        for (int i = 0; i < 5; i++) new YieldingThread(); }
```


Sleeping

- Eine Alternative zur `yield()`-Methode ist die `sleep()`-Methode:
  - Sie hält einen Thread für eine bestimmte Zeitdauer (in Millisekunden) an
- `sleep()` muss immer in einem Exception `try-catch` Block ausgeführt werden, da es durch `interrupt()` unterbrochen werden kann
- Analog zu `yield()` gibt es keine deterministische Kontrolle zur Abarbeitungsfolge
  - Es wird nur garantiert, dass der Thread wenigstens die angegebene Dauer in Millisekunden wartet

sleep()-Beispiel

```
public class SleepingThread extends Thread {
   private int countDown = 5; private static int threadCount = 0;
   public  SleepingThread() {
      super("" + ++threadCount);     // Store the thread name
      start();                       // Start at construction
   }
   public String toString() {
      return "#" + getName() + ": " + countDown;
   }
   public void run() {
      while(true) {
         System.out.println(this);
         if(--countDown == 0)       // Break condition
            return;
         try {
            sleep(100);             // Sleep at least 100 ms
         } catch (InterruptedException e) {
            throw new RuntimeException(e);
         }
      }
   }
}
```

```
public static void main(String[] args) {
      for(int i = 0; i < 5; i++)
      new SleepingThread();  }
```


Priority

Die Priorität eines Threads bestimmt, wie oft der Thread vom JVM-Scheduler zur Ausführung gewählt wird:

- Falls mehrere Threads zur Auswahl stehen, dann wird immer der Thread mit der höchsten Priorität favorisiert
- Threads mit geringere Priorität werden seltener ausgewählt als Threads mit höherer Priorität (es gibt keine Deadlocks wegen zu geringer Priorität)

Thread-Prioritäten werden durch `setPriority()` und `getPriority()` bestimmt.

Priority: Beispiel

```
public class PrioritiesThread extends Thread {
  private int countDown = 5;
  private volatile double d = 0;          // No optimization
  public  PrioritiesThread (int priority) {
    setPriority(priority);
    start();                              // Start at construction
  }
  public String toString() {
    return super.toString() + ": " + countDown;
  }
  public void run() {
    while(true) {
      // An expensive, interruptable operation
      for (int i = 1; i < 100000; i++)
        d = d + (Math.PI + Math.E) / (double)i;
      System.out.println(this);
      if(--countDown == 0)                // Break condition
        return;
    }
  }
}
```

```
public static void main(String[] args) {
   new PrioritiesThread(Thread.MAX_PRIORITY);
   for (int i = 0; i < 5; i++)
     new PrioritiesThread(Thread.MIN_PRIORITY); }
```


Dämon-Threads

- Ein Thread ist ein Dämon-Thread (deamon thread, dt.: Hintergrund-Thread), falls das Programm trotz des Dämon-Threads terminiert
- Der Dämon Status eines Threads muss mit `setDaemon()` vor dem Starten gesetzt werden
  - `isDeamon()` liefert den Status

Das Programm wird automatisch beendet, wenn es nur aus Dämon-Threads besteht!

```
public class DaemonThread extends Thread {
  public DaemonThread() {
    setDaemon(true);
    start();      // Start at construction
  }
  public void run() {
    while(true) {
      try {
        sleep(1000); // Sleep a little bit
      } catch (InterruptedException e) {
        throw new RuntimeException(e);
      }
      System.out.println(this);
    }
  }
}
```

```
public static void main(String[] args) {
    for (int i = 0; i < 10; ++i)
      new DaemonThread();
}
```

Joining

- Ein Thread kann auf die Terminierung eines anderen Threads warten, bevor dieser mit der Ausführung weiterfährt
- Falls ein Thread **A** den Befehl **`B.join()`** eines anderen Threads **B** aufruft, dann wartet der Thread **A** solange, bis **B** terminiert (d.h. **`B.isAlive()`** ist false).
- Die **`join()`**-Methode kann unterbrochen werden (wie **`sleep()`**) mit **`interrupt()`**, sie muss deshalb in einen Exception **`try-catch-Block`**
- Der **`interrupt()`**-Aufruf erzeugt eine **`InterruptedException`**. Man muss deshalb immer den **`join()`**-Anweisung innerhalb einer **`try`**-Anweisung benutzen

join()-Beispiel

```
public class Sleeper extends Thread {
  private int duration;
  public  Sleeper(String name,
                  int    sleepTime) {
    super(name);
    duration = sleepTime;
    start();
  }
  public void run() {
    try {
      sleep(duration);
    } catch (InterruptedException e) {
      System.out.println(getName() + \
          " was interrupted");
      return;
    }
    System.out.println(getName() + \
        " has awakened");
  }
}
```

```
public class Joiner extends Thread {
  private Sleeper sleeper;
  public  Joiner(String name,
                 Sleeper sleeper) {
    super(name);
    this.sleeper = sleeper;
    start();
  }
  public void run() {
    try {
      sleeper.join();
    } catch (InterruptedException e) {
      throw new RuntimeException(e);
    }
    System.out.println(getName() + \
        " join completed");
  }
}
```

```
public class Joining {
  public static void main(String[] args) {
    Sleeper kurt     = new Sleeper("Kurt", 9500),
            brad     = new Sleeper("Brad", 9500);
    Joiner  paola    = new Joiner("Paola", kurt),
            angelina = new Joiner("Angelina", brad);
    kurt.interrupt();    }    }
```

Threads und Vererbung …

- Es ist nicht immer möglich eine Klasse, die als Thread laufen soll, von **`Thread`** abzuleiten. Insbesondere dann nicht, wenn diese Klasse Bestandteil einer Vererbungshierarchie ist, die nichts mit Multithreading zu tun hat
- Es gibt in Java keine Mehrfachvererbung, also:

```
public class LongSleeper extends Sleeper, Thread
```

  ist nicht möglich
- In diesem Fall muss man …
  - Ein Interface **`Runnable`** für die Klasse implementieren
  - Eine geeignete **`run()`**-Methode zur Verfügung stellen
  - An den Konstruktor des **`Thread`**-Objekts eine Instanz der Klasse als Argument übergeben
  - Den Thread mit der **`start()`**-Methode aktivieren
- Generell sollte man die Implementierung des Interface **`Runnable`** wenn immer möglich der Ableitung von **`Thread`** vorziehen!

… Threads und Vererbung

- Mit **`Thread.currentThread()`** erhält man eine Referenz auf das **`Thread`**-Objekt.

```
public class RunnableThread implements Runnable {
  private int countDown = 5;
  public  String toString() {
    return "#" + Thread.currentThread().getName() +
      ": " + countDown;
  }
  public void run() {
    while(true) {
      System.out.println(this);
      if(--countDown == 0)
        return;
    }
  }
}
```

```
public static void main(String[] args) {
    for(int i = 1; i <= 5; i++)
      new Thread(new RunnableThread(), "" + i).start();
```

Innere Klassen und Threads ...

- Eine Alternative zu **`Runnable`** ist die Definition einer Inneren Klasse, welche **`Thread`** implementiert.
- Das kann realisiert werden, da innere Klassen auf die Daten der äusseren Klasse zugreifen können.

```
// Using a named inner class:
class InnerThread {
  // Can be accessed by Inner:
  private int countDown = 5;
  private Inner inner;
  private class Inner extends Threads {
    Inner(String name) {
      super(name);
      start();
    }
    public String toString() {
      return getName() + ": " + countDown;
    }
    public void run() {
      while(true) {
        System.out.println(this);
        if(--countDown == 0)
          return;
  } } }         // End of Inner
  public InnerThread(String name) {
    inner = new Inner(name);
} }
```


... Innere Klassen und Threads

- Die innere Klasse kann als anonyme Klasse realisiert werden, wenn sie keine Methode besitzt, welche die Äussere benutzten muss

```
// Using an anonymous inner class:
class AnonymousThread {
  private int countDown = 5;  // the inner Thread is anonymous:
  private Thread t;
  public  AnonymousThread(String name) {
    t = new Thread(name) { // defined as:
      public String toString() {
        return getName() + ": " + countDown;
      }
      public void run() {
        while(true) {
          System.out.println(this);
          if(--countDown == 0)
            return;
        }
      }
    } // end of definition
    t.start();
}  }
```

```
public class ThreadVariations {
  public static void main(String[] args) {
    new InnerThread("InnerThread");
    new AnonymousThread("AnonymousThread");
} }
```

Beispiel: I/O-Tasks vs. rechenintensive Tasks

- Threads werden oft benutzt, um I/O-Operationen von rechenintensiven Tasks zu trennen
- Beispiel Primfaktorzerlegung:
  Darstellung einer positiven natürlichen Zahl als Primzahlprodukt
  - Beispiel: 12 = 2*2*3
- Dies kann in zwei Tasks aufgeteilt werden
  - I/O-Task: Einlesen der Daten
  - Berechnungs-Task: Berechnung der Primfaktoren
    class **`PrimeNumberTools`**

Beispiel: I/O-Tasks vs. rechenintensive Tasks

```
import java.io.*;
public class Prime {
   public static void main(String[] args) {
      PrimeNumberTools pt = new PrimeNumberTools();
      BufferedReader   in = new BufferedReader( new InputStreamReader(
                                new DataInputStream(System.in)));
      long   startTime, elapsedTime;
      double seconds;       // the time required to execute the algorithm
      int num;
      startTime = System.currentTimeMillis();
      try {
         while (true) {
            System.out.print("Bitte eine Zahl eingeben:");
            num = (new Integer(in.readLine())).intValue();
            if (num == -1) {
               break;  }
            pt.printPrimeFactors(num);
         }
      } catch (IOException e) {  }
      elapsedTime = System.currentTimeMillis() - startTime;
      seconds =  elapsedTime / 1000.0;
      System.out.println("\n" + seconds + " seconds of total execution time"
            + " were required to compute all prime factors.");
    } }
```

Beispiel: I/O-Tasks vs. rechenintensive Tasks

```
public class PrimeNumberTools {
  public void printPrimeFactors(int num) {
    int  whichprime = 1, prime;
    long startTime, elapsedTime;
    double seconds;   // time for algorithm
    String prefix = "primeFactors("+num+")= ";
    startTime = System.currentTimeMillis();
    while (num > 1) {
       prime = getPrime(whichprime);
       if (num % prime == 0) {
          System.out.print(prefix+prime);
          prefix = " ";
          num /= prime;
       } else {
          ++whichprime;
       }
    }
    System.out.println();
    elapsedTime = System.currentTimeMillis() - startTime;
    seconds =  elapsedTime / 1000.0;
    System.out.println("\n" + seconds + "seconds of execution time"
                  + "were required to compute one prime factor.");
  }
```

```
public int getPrime(int cnt) {
  int i = 1, ret = 2;
  while (i < cnt) {
    ++ret;
    if (isPrime(ret)) ++i;
  }
  return ret;
}
private boolean isPrime(int num) {
  for (int i = 2; i < num; ++i) {
    if (num % i == 0){
      return false;
    }
  }
  return true;
}
```

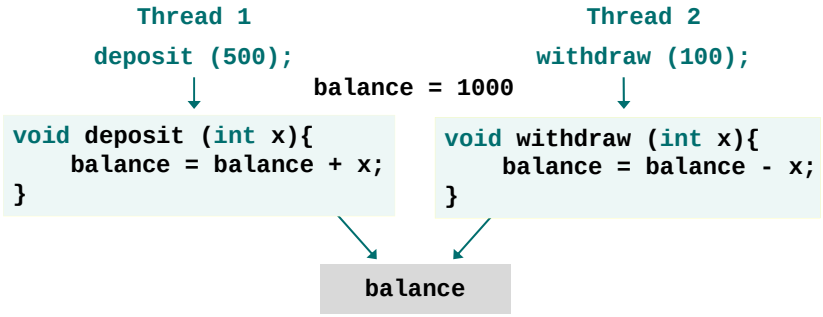

Beispiel: I/O-Tasks vs. rechenintensive Tasks

```
public class ThreadedPrimeNumberTools
   extends PrimeNumberTools
   implements Runnable
{
  private int num;
  public void printPrimeFactors(int num)
  {
     Thread t = new Thread(this);
     this.num = num;
     t.start();
  }
  public void run()
  { super.printPrimeFactors(num); }
}
```

```
import java.io.*;
public class PrimeThreads {
  public static void main(String[] args) {
  ThreadedPrimeNumberTools pt;
  . . .
 try {
   while (true) {
     System.out.print("Bitte Zahl
                      eingeben:");
     System.out.flush();
     num =(new Integer(in.readLine())).intValue();
     if (num <= 0) {
        break;
     }
     pt = new ThreadedPrimeNumberTools();
     pt.printPrimeFactors(num);
   }
 } catch (IOException e) { }
 System.out.print("Waiting for running prime factor threads");
}
```

Threads und Synchronization

- Der Thread-Zugriff auf gemeinsame Ressourcen (z.B. auf gemeinsame Daten) erfordert besondere Aufmerksamkeit



- Führen beide Threads Änderungen auf gemeinsamen Daten durch, so müssen diese synchronisiert werden, denn andernfalls können undefinierte Zugriffe entstehen

Beispiel: Thread & Synchronisationskonflikte

- Falls Master-Thread und "Watcher"-Thread zur gleichen Zeit `ae.getValue()` und `ae.next()` aufrufen, dann kann es in der Objektvariable `i` zu einem Konflikt kommen

```
public class AlwaysEven {
   private int i;
   public  void next() { i++; i++; }
   public  int getValue() { return i;  }  // can be called anytime!!!
   public  static void main(String[] args) {
      final AlwaysEven ae = new AlwaysEven();
      new Thread("Watcher") {            // anonymous thread
         public void run() {
            while(true) {
               int val = ae.getValue();
               if(val % 2 != 0) {
                  System.out.println(val);
                  System.exit(0);
            }  }  }
      }.start();
      while(true)
         ae.next();
   } }
```

Zugriffs-Synchronisation

- Der Zugriff auf gemeinsame Ressourcen wird in der Regel synchronisiert
  - Nur ein Thread kann zu einem bestimmten Zeitpunkt Instanzvariablen modifizieren
- Ein Objekt wird mit einer Lockvariablen blockiert, falls es durch einen Thread modifiziert wird. Dieser Mechanismus wird als Mutual Exclusion (mutex) bezeichnet
- In Java benutzt man dazu **`synchronized`**

```
public class AlwaysEven {
  private int i;
  // next() and getValue() mutually exclude themselves
  public synchronized void next() { i++; i++; }
  public synchronized int getValue() { return i; }
  public static void main(String[] args) {
    // as before ...  }
}
```

- Das Locking bezieht sich auf **`AlwaysEven`**

Thread-Synchronisation

- Falls ein Thread eine **`synchronized`**-Methode eines Objektes A aufruft, dann sind alle Threads, die eine **`synchronized`**-Methode des gleichen Objekts A aufrufen, blockiert
- Falls eine **`static`**-Instanzvariable als **`synchronized`** definiert wird, dann wird der Thread-Zugriff synchronisiert
- Programmfragmente können **`synchronized`** werden:

```
class AClass {
   public void aMethod() {
      // not synchronized!
      System.out.println("Not synchronized.");
      // not synchronized!
      synchronized(this) {
         System.out.println("Synchronized.");
      }
   }
}
```

`wait()`-Methode

- Es stehen in einem Synchronisationsblock folgende Synchronisationsprimitive zur Verfügung:
  - `wait()`
  - `notifyAll()`
  - `notify()`
- Der `wait()`-Methode kann wie der `sleep()`-Methode der Zeitraum zum Warten mitgeteilt werden. Wichtiger Unterschied zu `sleep()`:
  - `wait()` gibt den Lock-Status der Methode frei
  - `wait()` kann mit `notify()` oder `notifyAll()` unterbrochen werden
- Falls kein Argument an `wait()` übergeben wird, dann wartet der Thread solange, bis entweder `notify()` oder `notifyAll()` aufgerufen wird
- Es gibt bei der `wait()`-Methode kein busy-wait

Wie funktioniert `wait()`?

- Da die `wait()`-Methode den Synchronisations-Lock des Objektes wieder frei gibt, muss man es innerhalb eines `synchronized` Blockes benutzen
- Die `wait()`-Methode ist eine Methode der Klasse **`object`** und nicht von **`thread`**
  → deshalb kann die Methode überall aufgerufen werden

```
while (conditionIsNotMet) {
  synchronized(this) {
    try {
      wait();
    } catch (InterruptedException e) {
      throw new RuntimeException(e);
    }
  }
}
```

```
synchronized(x) {
   x.notify ();
}
```

- `wait()` sollte immer in einer **`while`**-Schleife benutzt werden

Synchronization ...

- Die `wait()`, `notify()` und `notifyAll()`-Methoden werden zur Steuerung zeitlicher Abläufe benutzt
  - Die Threads werden nur aktiviert, wenn eine bestimme Bedingung aktiviert oder erreicht ist

- Ein Erzeuger/Verbraucher-Beispiel:
  - ein Koch (`Chef`) produziert Food (repräsentiert als Restaurant Order)
  - ein Kellner (`Waiter`) muss solange warten, bis das Food fertig ist und kann dieses dann dem Kunden liefern (Restaurant Order)

... Synchronisation ...

```
public class Restaurant {
  Order  order;
  public static void main(String[] args) {
    Restaurant restaurant = new Restaurant();
    Waiter waiter = new Waiter(restaurant);
    Chef   chef = new Chef(restaurant, waiter);
    Thread wt = new Thread(waiter); wt.start();
    Thread ct = new Thread(chef); ct.start();
  } }
```

```
class Order {
  private static int i = 0;
  private int count = i++;
  public  Order() {
    if (count == 10) { // constructor
      System.out.println
        ("Out of food, closing");
      System.exit(0);
    }
  }
  public String toString()
  { return "Order " + count; }
}
```

… Synchronisation

```
class Waiter implements Runnable {
  private Restaurant restaurant;
  public Waiter(Restaurant r) {
    restaurant = r;  }

public void run() {
   while(true) {
     while (restaurant.order == null)
     synchronized(this) {
       try {
         System.out.println(
           "Waiter is waiting ");
         wait(); // wait for an order
       } catch (InterruptedException e){
         throw new RuntimeException(e);
       } }
     System.out.println(
       "Waiter got "+ restaurant.order);
     restaurant.order = null;
   } // end of infinite loop
} }
```

- `wait()` wird auf dem Waiter-Objekt `this` ausgeführt

```
class Chef implements Runnable {
  private Restaurant restaurant;
  private Waiter waiter;
  public  Chef(Restaurant r, Waiter w) {
    restaurant = r;
    waiter = w;
    start();   }
  public void run() {
    while (true) {
      if (restaurant.order == null) {
        restaurant.order = new Order();
        System.out.print("Order up! ");
        synchronized(waiter) {
          waiter.notify();
        } }
      try {
       Thread.currentThread().sleep(100);
      } catch (InterruptedException e) {
        throw new RuntimeException(e); }
    } // end of infinite loop
} }
```

- `notify()` muss innerhalb von `synchronized` stehen

Pipe-Kommunikation zwischen Threads …

- Die Kommunikation zwischen Threads kann auch mit `Pipes` anstatt einer `wait()`-Schleife durchgeführt werden
- Java implementiert Pipes mit `PipedWriter` und `PipedReader`

```
public class CommunicationThreads
{
  public static void main(String[] args)
  throws Exception {
    PipedInputStream inPipe  =
           new PipedInputStream();
    PipedOutputStream outPipe =
           new PipedOutputStream(inPipe);
    Producer p = new Producer(outPipe);
    Consumer c = new Consumer(inPipe);
    Thread pt = new Thread(p); pt.start();
    Thread ct = new Thread(c); ct.start();
  }
}
```

... Pipe-Kommunikation zwischen Threads

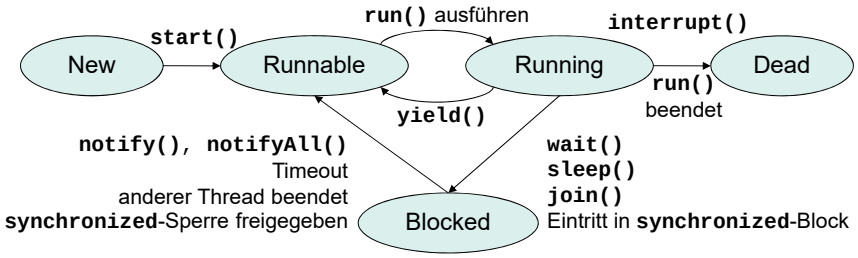
```
class Producer implements Runnable {
 private PipedOutputStream pipe;
 public Producer(PipedOutputStream pipe){
   this.pipe = pipe; }
 public void run() {
    while (true) {
      byte b=(byte)(Math.random()*128);
      try {
        pipe.write(b);
        System.out.println("
         Produzent erzeugte"+b);
      } catch (IOException e){
        System.err.println(e.toString());
      }
      try {
        Thread.sleep((int)
              (100*Math.random()));
      } catch (InterruptedException e){ }
} } } }
```

```
class Consumer implements Runnable {
 private PipedInputStream pipe;
 public Consumer(PipedInputStream pipe){
   this.pipe = pipe;  }
 public void run() {
   while (true) {
     try {
      byte b = (byte)pipe.read();
      System.out.println("Konsument
                    fand " + b);
     }catch (IOException e) {
       System.err.println(e.toString());
     }
     try {
       Thread.sleep((int)
             (100*Math.random()));
     } catch (InterruptedException e){ }
} } } }
```

```
public class CommunicationThreads {
  public static void main(String[] args)
  throws Exception {
    PipedInputStream inPipe  = new PipedInputStream();
    PipedOutputStream outPipe = new PipedOutputStream(inPipe);
    Producer p = new Producer(outPipe);
    Consumer c = new Consumer(inPipe);
    Thread pt = new Thread(p); pt.start();
    Thread ct = new Thread(c); ct.start();
  } }
```


Status eines Threads

- Ein Thread kann sich in folgendem Status befinden:
  - **`New:`** Der Thread wurde erzeugt, aber noch nicht gestartet
  - **`Blocked:`** Der Thread wurde gestartet, aber ein Lock verhindert, dass der Thread vom Scheduler aktiviert werden kann
  - **`Runnable:`** Thread ist gestartet, wird aber gerade nicht ausgeführt
  - **`Running:`** Der Thread wurde gestartet und ist nicht blockiert
  - **`Dead:`** Der Thread hat die **`run()`**-Methode beendet



Blocked Threads

- Ein Thread kann sich aus folgenden Gründen im Status **`blocked`** befinden:
  - die Thread-Ausführung wird durch **`sleep()`** oder **`wait()`** verzögert
  - der Thread möchte ein **`synchronized`** Programmteil ausführen, aber das Objekt befindet sich gerade in einem Lock Modus
  - der Thread erwartet I/O (z.B. **`Pipes`**)

Einen Thread stoppen

- Es wird empfohlen, **`stop()`**, **`suspend()`** und **`resume()`** *nicht* zu benutzen, da Locks nicht freigegeben werden.
- Um einen Thread trotzdem zu stoppen, sollte man ein Flag (**`volatile`**) benutzen

```
class CanStop extends Thread {
  // Must be volatile:
  private volatile boolean stop=false;
  private int counter = 0;
  public void run() {
    while (true) {
       System.out.println(counter++);
       if (stop) {
         System.out.println("Detected
         return;
       }
    }
  }
  public void requestStop() { stop = true; }
}
```

```
public class Stopping {
 public static void main(String[] args){
   final CanStop stoppable = new CanStop();
   stoppable.start();
   new Timer(true).schedule(new TimerTask(){
      public void run() {
        System.out.println
           ("Requesting stop");
        stoppable.requestStop();
      }
    }, 50); //run only 500 milliseconds
  }
}
```

Einen Thread unterbrechen

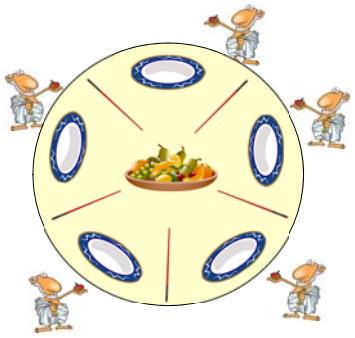
- Man kann keinen `blocked` Thread durch Setzen eines Flags unterbrechen (da der Thread es nicht überprüfen kann!)
- In diesem Fall muss man die `interrupt()`-Methode des Threads aufrufen
- Der Aufruf von `interrupt()` erzeugt eine `InterruptedException`
  - deshalb muss diese in einer `try`-Anweisung abgefangen werden

Deadlocks (dt.: "Verklemmungen")

- Ein Deadlock liegt vor, wenn zwei (oder mehrere) Threads sich gegenseitig und zur gleichen Zeit blockieren
  - beide (alle) sich also in einem `blocked`-Zustand befinden
- Genauer müssen die folgenden Punkte für einen Deadlock erfüllt sein:
  - Eine Methode ist als `synchronized` definiert
  - Einer der beiden Threads hält den Methoden-Lock und wartet gleichzeitig auf Input des/der anderen Threads
  - Thread A wartet auf Thread B → Thread B wartet auf Thread C → … → Thread C wartet auf Thread A

Beispiel: Die fünf speisenden Philosophen

- 1965 stellt E. Dijkstra das Synchronisationsproblem anhand der fünf speisenden Philosophen („dining philosophers“) vor:
  - Philosophen denken oder essen mit Stäbchen
  - Sie benötigen zwei Stäbchen zum Essen, in jeder Hand eines
  - Zyklus: Denken – Essen – Denken – usw.



  - Ein Deadlock entsteht z.B. wenn alle Philosophen gleichzeitig das von ihnen aus gesehen linke Stäbchen nehmen und danach das rechte greifen möchten
  - Es wird Verfahren zur Synchronisation der konkurrierenden Zustände gesucht, bei dem maximale Parallelität herrscht und keiner verhungert

Java Beispiel: Dining Philosophers ...

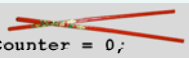
```
class Philosopher implements Runnable {
  private static Random rand =
                 new Random();
  private static int counter = 0;
  private int number = counter++;
  private Chopstick leftChopstick;
  private Chopstick rightChopstick;
  static int ponder = 0;

public Philosopher(Chopstick left,
                   Chopstick right) {
    leftChopstick  = left;
    rightChopstick = right;
    Thread philT = new Thread(this);
    philT.start(); }

public void think() {
    System.out.println(this+"thinking");
    if (ponder > 0)
      try {
        sleep(rand.nextInt(ponder));
      } catch(InterruptedException e) {
        throw new RuntimeException(e);
  }  }
```

```
public void eat() {
  synchronized(leftChopstick) {
    System.out.println(this + "has"
    + this.leftChopstick +
    + "Waiting for"
    + this.rightChopstick); }
  synchronized(rightChopstick) {
      System.out.println(this +
      "eating"); }
}
public String toString() {
   return "Philosoph" + number; }
public void run() {
   while(true) {
     think();
     eat();
} } } // end class Philosopher
```

```
class Chopstick {
  private static int counter = 0;
  private int number = counter++;
  public String toString(){
     return "Chopstick"+ number ; }}
```



… Java Beispiel: Dining Philosophers

```
public class DiningPhilosophers {
  public static void main(String[] args) {
    Philosopher[] philosopher =
        new Philosopher[Integer.parseInt(args[0])];
    Philosopher.ponder = Integer.parseInt(args[1]);
    Chopstick
      left =  new Chopstick(),
      right = new Chopstick(),
      first = left;
    int i = 0;
    while (i < philosopher.length - 1) {
      philosopher[i++] = new Philosopher(left,right);
      left = right;
      right = new Chopstick();
    }
    // deadlock
    philosopher[i] = new Philosopher(left, first);
    // Swapping left/right prevents deadlock:
    // philosopher[i] = new Philosopher(first, left);
  }
}
```


Links

- Java Threads-Tutorial:
  `https://docs.oracle.com/javase/tutorial/essential/concurrency/`
- Artikel zu Threads in Swing-Applikationen:
  `https://docs.oracle.com/javase/tutorial/uiswing/concurrency/initial.html`