Verteilte Software - Java - Nebenläufigkeit 1

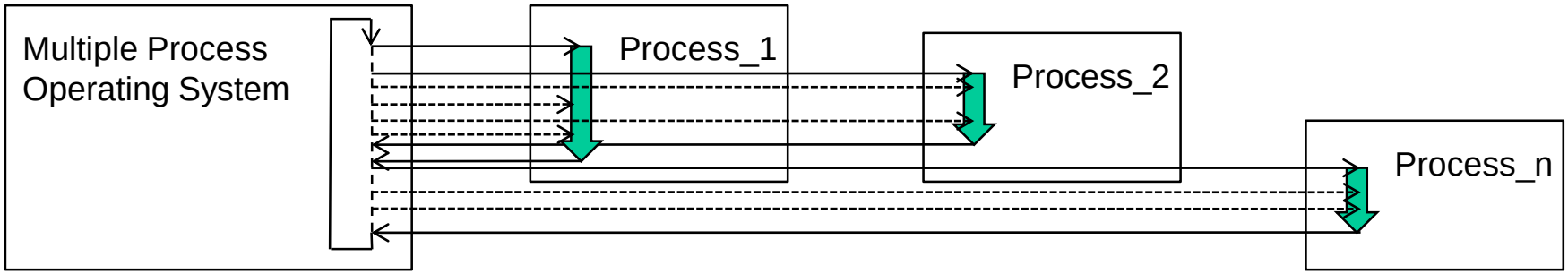
Nebenläufigkeit (mitunter auch ***Parallelität***) ist die Eigenschaft eines Systems, mehrere Aufgaben (Anweisungen etc.) gleichzeitig ausführen zu können. (Wikipedia)

Echte Parallelität: Prozesse laufen auf unterschiedlichen Prozessoren (Prozessorkernen)
Virtuelle (Quasi) Parallelität: Prozesse laufen auf einem CPU

Prozess

aktives Programm + Daten + Ausführungsumgebung
unabhängig von anderen Prozessen
eigener Adressraum

Multitasking: Betriebssystem verwaltet mehrere „schwergewichtige Prozesse“ (schaltet um)

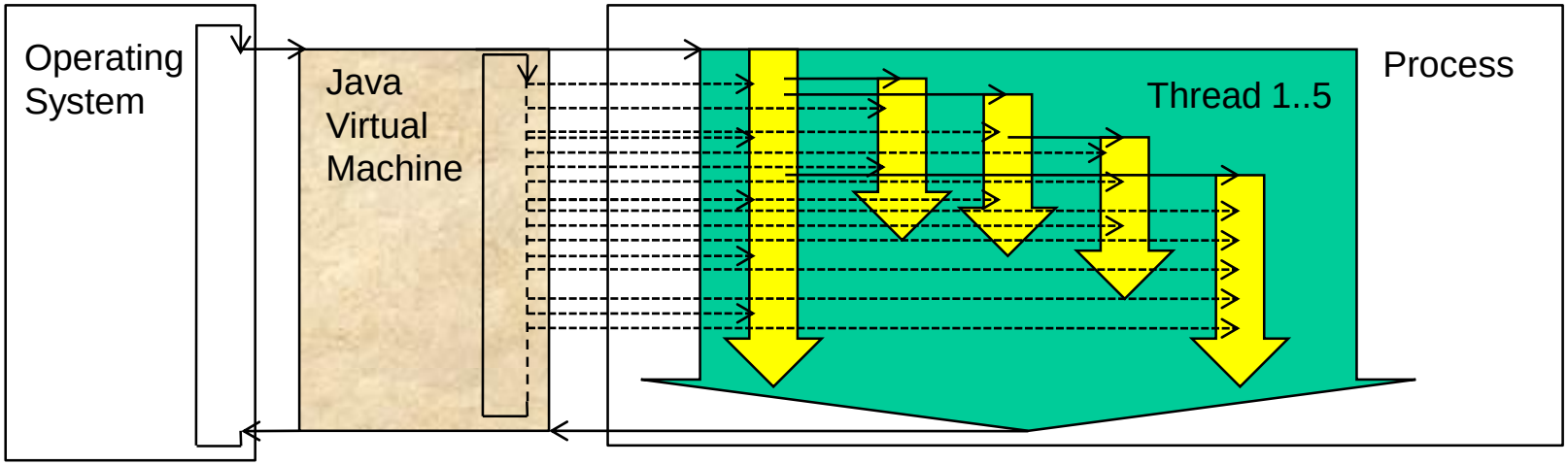


Quelle: Prof. B. Steinbach, Vorlesung „Verteilte Software“

Multithreading: mehrere Programmfäden werden parallel ausgeführt

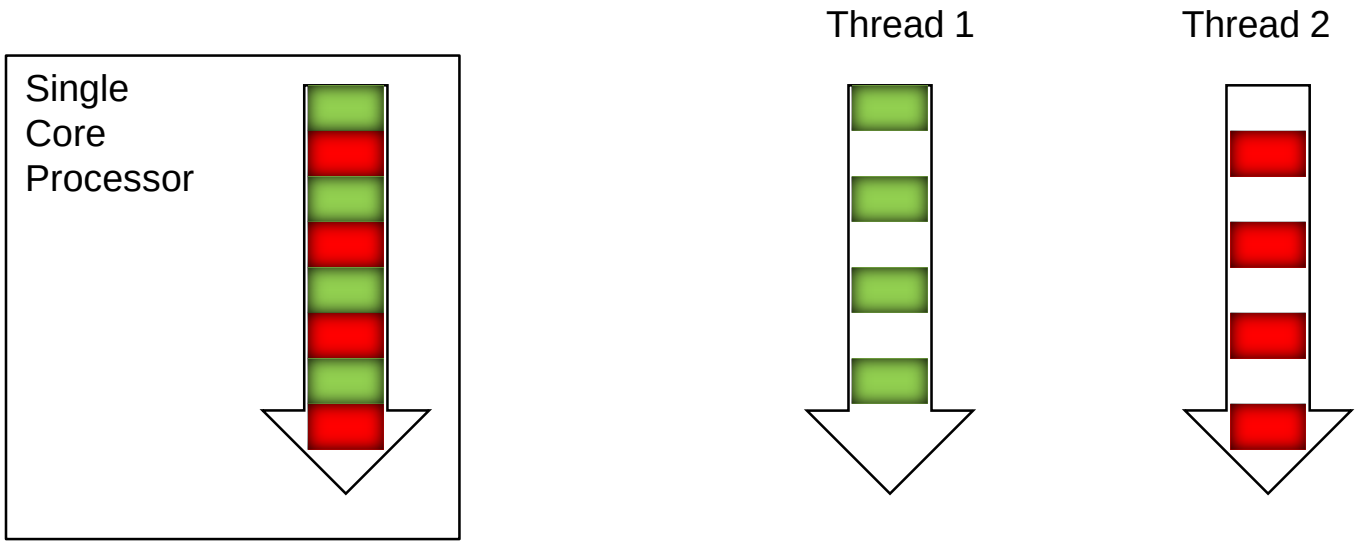
Thread

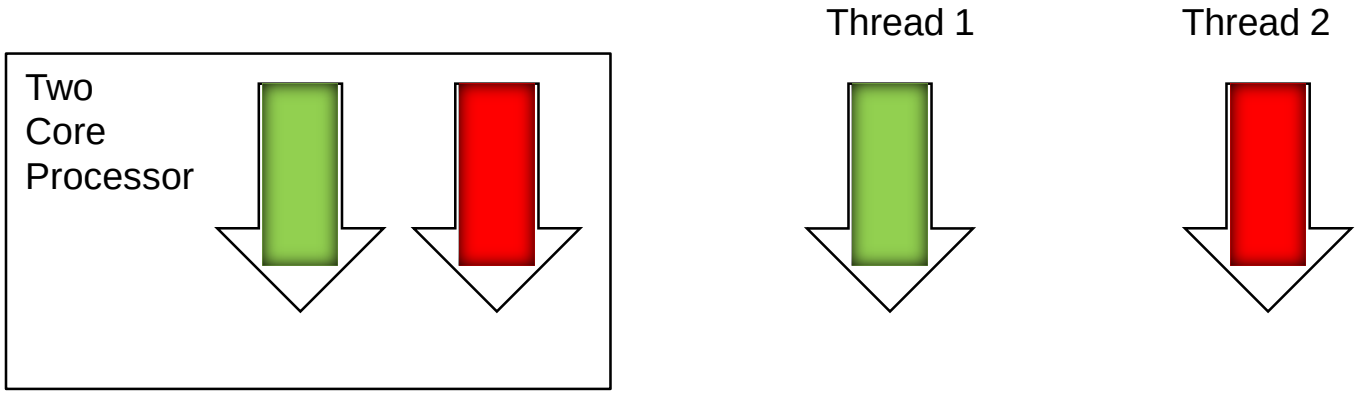
leichtgewichtige Prozesse (leicht wechelbar)
laufen parallel, ggf. durch die virtuelle Maschine simuliert
(VM regelt Aufteilung und Synchronisation)
benutzen die gleichen Ressourcen
(den gleichen Adressraum)



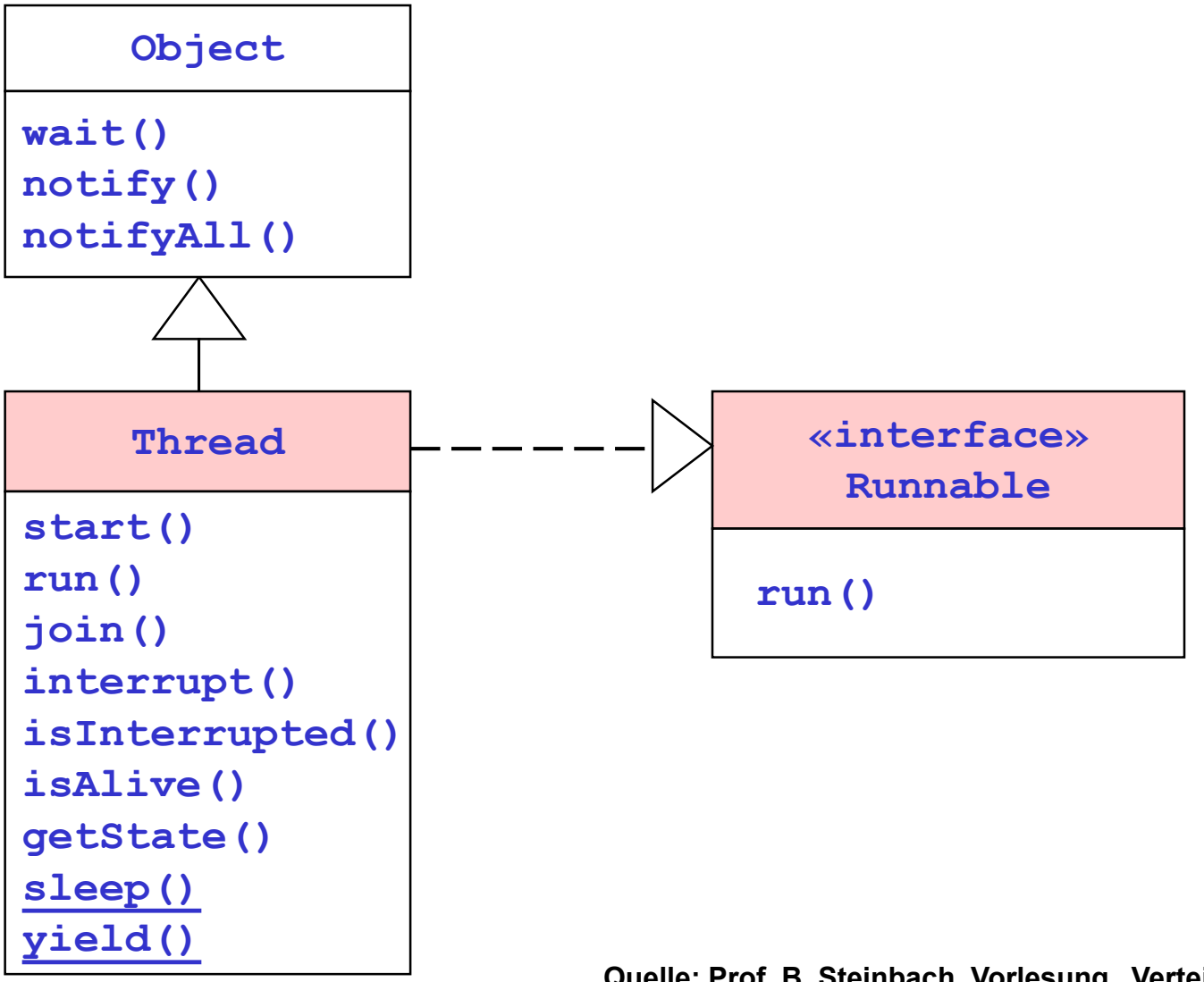
Quelle: Prof. B. Steinbach, Vorlesung „Verteilte Software“

Verteilte Software - Java - Threads 3





Quelle: Prof. B. Steinbach, Vorlesung „Verteilte Software“



Quelle: Prof. B. Steinbach, Vorlesung „Verteilte Software“

Klasse Thread

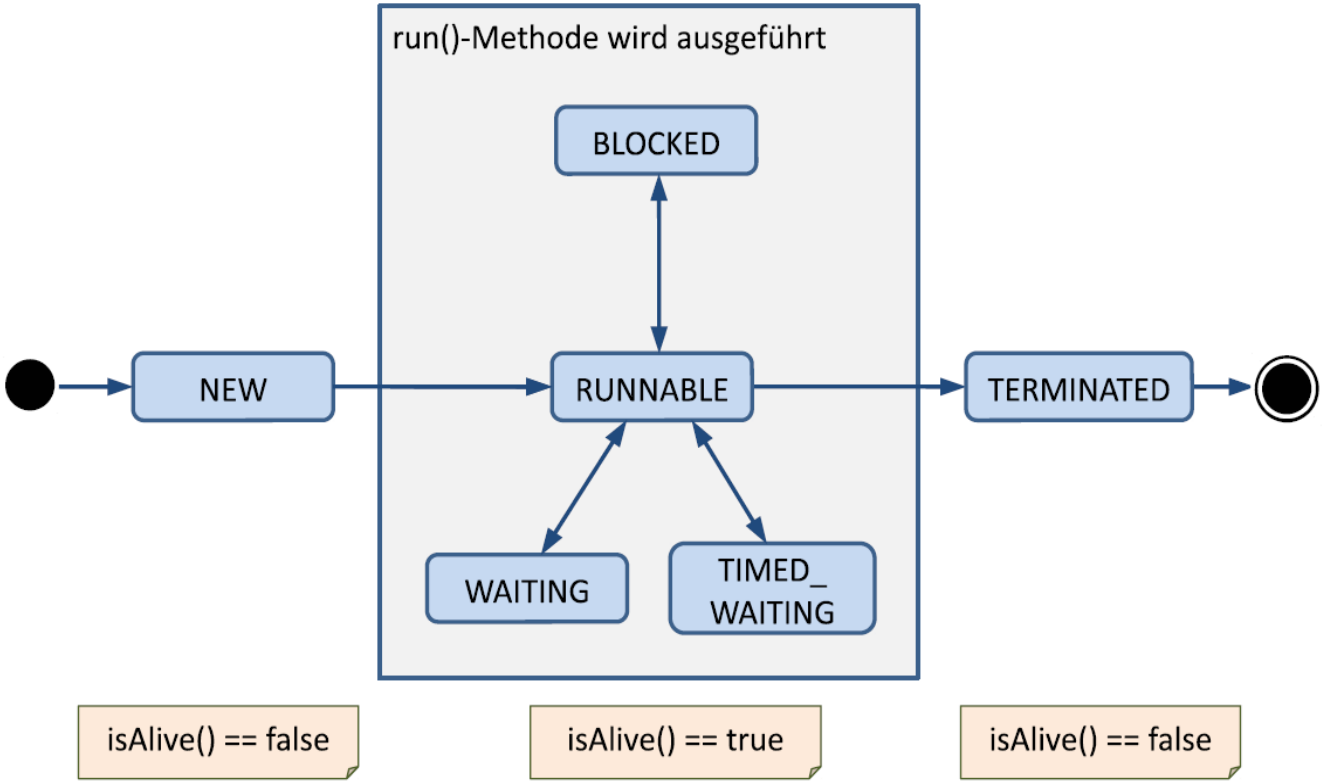
Konstruktoren (8):

```
Thread()
Thread(Runnable target, String name)
```

Methoden

run() in den abgeleitenen Klassen unbedingt zu implementieren
start() startet Thread
join() der aufrufende Thread lässt auf sich warten
interrupt() beendet die Ausführung vorzeitig
Achtung: Direkter Abbruch von außen nicht möglich
isInterrupted()
fragt ab, ob Thread angehalten werden soll
isAlive() fragt ab, ob Thread bereits läuft
getState() Abfrage des Zustandes
sleep() versetzt in den Zustand TIMED_WAITING (static)
yield() lässt eine Runde aus (static)
wait() warte auf die Weckinformation durch den anderen Thread
notify() wecke einen einzelnen Thread
notifyAll() wecke alle Threads

Lebenszyklus (Zustände) eines Thread-Objektes



synchronized

Object.notifiy()
Object.notifyAll()

Object.wait()
Thread.join()
Thread.sleep()

Object.notifiy()
Object.notifyAll()

Quelle: Jörg Hettel, Manh Tien Tran: Nebenläufige Programmierung mit Java, dpunkt.verlag

Thread erzeugen und ausführen

Unterklasse definieren und run-Methode redefinieren

```
public class ThreadX extends Thread{
     public void run(){
    // Achtung! Wenn die Methode zu Ende,
    // ist auch Thread zu Ende
    }
}
```

- Objekt anlegen

```
ThreadX x=new ThreadX();
```

- Nebenläufige Ausführung der run-Methode veranlassen

```
x.start();
```

Thread abbrechen

- **interrupt()-**Methode setzt Interrupt-Flag
- **isInterrupted()** fragt Interrupt-Flag ab
- direkter Abbruch nicht möglich!
- **bei Verwendung von sleep und join muss** InterruptException behandelt werden

```
ThreadA t=new ThreadA();
t.start();
Thread.sleep( 2000 );
t.interrupt();
```

```
public void run()
  {
     while ( ! isInterrupted() )
    {
      System.out.println( "Und er läuft und er läuft und er läuft" );
      try
      {
       Thread.sleep( 500 );
      }
      catch ( InterruptedException e )
      {
       interrupt();
       System.out.println( "Unterbrechung in sleep()" );
      }
    }
    System.out.println( "Das Ende" );
  }
};
```

Auf Thread-Ende warten

- join()-Methode

```
Thread a = new ThreadA();
Thread b = new ThreadB();
a.start();
b.start();
a.join(); //warten auf a
b.join(); //warten auf b
```

```
public class MyThread extends Thread {
  private String id;
  public MyThread (String id) { this.id = id; }
  public void run() {
         for (int i = 0; i < 3; i++)
         try {
           sleep(Math.round(1000.0 * Math.random()));
           System.out.println(id +" cycle "+ i);
         }
         catch (InterruptedException e) { interrupt(); }
  }
}
public class MainThread {
     public static void main(String[] args) {
          MyThread t1, t2;
          t1 = new MyThread("t1");
          t2 = new MyThread("t2");
          t1.start();
          System.out.println("t1 " + t1.getState());
          System.out.println("t2 " + t2.getState());
          t2.start();
          System.out.println("t1 " + t1.getState());
          System.out.println("t2 " + t2.getState());
     }
}
```

```
t1 RUNNABLE
t2 NEW
t1 TIMED_WAITING
t2 RUNNABLE
t1 cycle 0
t1 cycle 1
t2 cycle 0
t2 cycle 1
t1 cycle 2
t2 cycle 2
```

```
public class Worker extends Thread {
   public void run(){
     try {
       int time = (int)(1000.0 * Math.random());
       sleep(time);
       System.out.println("time " + time + " ms");
     }
     catch (InterruptedException e) {
       interrupt();
     }
  }
}

public class MainThread {
   public static void main(String[] args) {
     Worker w1 = new Worker();
     Worker w2 = new Worker();
     w1.start();
     w2.start();
     try {
       w1.join();
       w2.join();
     }
     catch (InterruptedException e){ interrupt(); }
     System.out.println("Beide fertig");
  }
}
```

time 482 ms
time 515 ms
Beide fertig

weitere Methoden

final String getName() liefert den Namen des Threads
final void setName(String name) ändert den Namen des Threads
static Thread Thread.currentThread() liefert den aktuellen Thread
final int getPriority() liefert die Priorität des Threads
final void setPriority(int newPriority) setzt die Priorität des Threads

Default-Name: Thread-X
Priorität: MIN_PRIORITY=1 bis MAX_PRIOTITY=10, (NORM_PRIORITY=5)
anderenfalls **IllegalArgumentException**

```
ThreadNaming t1=new ThreadNaming();
ThreadNaming t2=new ThreadNaming();
System.out.println("Name of t1:"+t1.getName());
System.out.println("Name of t2:"+t2.getName());
//…
```

Name of t1:Thread-0
Name of t2:Thread-1

Interface Runnable

- wenn Basisklasse bereits vorhanden (einer der Gründe)

```
class Counter extends BaseClass implements Runnable
{
  @Override public void run()
  {
    for ( int i = 0; i < 100; i++ )
      System.out.println( i );
  }
}
```

- Runnable deklariert NUR die run-Methode
- zum Ausführen wird zusätzlich ein Thread („Executer“) benötigt

```
Thread t = new Thread( new Counter() );
t.start();
```

- Threadobjekt t verwendet beim Starten nicht die eigene run-Methode sondern die von Counter

```
public class Ansager {
        private String spruch="This is my favorite song: ";
        public String getSpruch(){ return spruch;
        }
}
public class SingenderAnsager extends Ansager implements Runnable {
        String text;

        public SingenderAnsager(String text) {
                this.text = text;
        }
        @Override
        public void run() {
             System.out.println(getSpruch());
             while (!Thread.interrupted()){
                  System.out.println(text);
                  try {
                    Thread.sleep((long) (Math.random()*1000));
                  } catch (InterruptedException e) {
                      Thread.currentThread().interrupt();
                      e.printStackTrace();
                  }
             }
       }
}
```

```
public class Buehne {
public static void main(String[] args) {
       SingenderAnsager sa1=new SingenderAnsager ("Lala");
       SingenderAnsager sa2=new SingenderAnsager ("Trala");
       Thread th1=new Thread(sa1);
       th1.start();
       Thread th2=new Thread(sa2);
       th2.start();
       try {
           Thread.sleep(2000);
       } catch (InterruptedException e) {
               Thread.currentThread().interrupt();
               e.printStackTrace();
       }
//      th1.interrupt();
//      th2.interrupt();
        System.exit(0);
     }
}
```

Dämonen

- erbringen Dienstleistungen
- arbeiten im Hintergrund

```
setDaemon(true)
setDaemon(false)
Boolean isDaemon()
```

Synchronisation

die Koordinierung des zeitlichen Ablaufs mehrerer nebenläufiger Prozesse bzw. Threads (in einem Programm, auf einem Computer, im verteilten System)

Zweck:

- Vermeidung der Dateninkonsistenzen durch den gleichzeitigen Zugriff
- Zur gemeinsamen Nutzung beschränkter Ressourcen (z.B. von Peripheriegeräten.
- Zur Interprozesskommunikation (Übergabe von Nachrichten zwischen den Prozessen)
- Zur Steuerung von Unterprozessen durch Signale (Systemnachrichten): das Abbrechen von Prozessen oder das Warten darauf, dass sie terminieren.

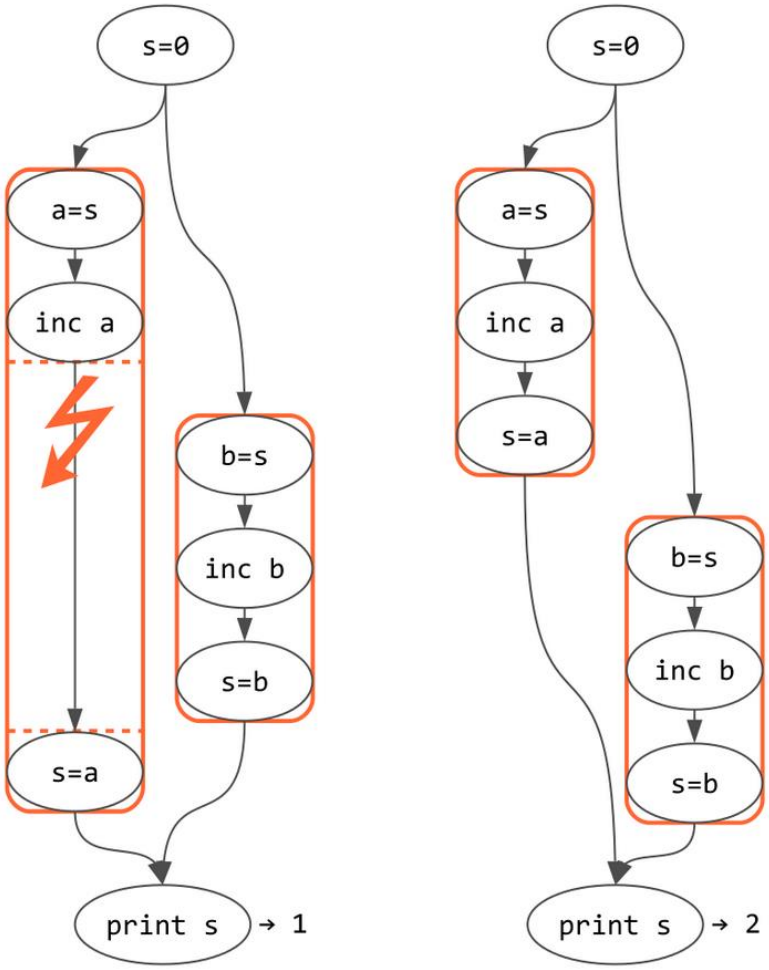
Synchronisation von Threads

Threads laufen im gleichen Prozess
und nutzen die gleichen Daten

Problem:
Thread-Umschaltung beim
nicht atomaren
Befehlen (ist i++ atomar?)

kein Problem:

- bei immutable (thread-sicheren) Objekten, z.B. String, StringBuffer
- Methoden, die keine Objekteigenschaften verändern, sondern nur lesen
- Variablen im eigenen Stack



Quelle: Wikipedia - Kritischer Abschnitt (Benutzer Nomen4Omen)

Kritische Abschnitte
zusammenhängende Programmblöcke, die während der Ausführung nicht unterbrochen werden dürfen

MUTEX – Verfahren (MUTual EXclusion) – gegenseitiger Ausschluss

- sorgt dafür, dass nur ein Prozess/Thread gleichzeitig auf eine Ressource zugreifen (sich in einem kritischen Abschnitt befindet) darf
- Lösung: an einer Speicherstelle (Semaphor) wird hinterlegt, ob die Ressource gerade genutzt wird oder nicht

Semaphor

Funktionsweise:

- Ein Zähler wird der mit Anzahl maximal verfügbaren Ressourcen initialisiert.
- Ein Prozess muss vor dem Zugriff die Operation „Reservieren“ aufrufen, und danach die Operation „Freigeben“. Der Zähler wird um 1 heruntergezählt oder erhöht.
- Beim Zählerstand 0 muss der reservierende Prozess warten.

https://de.wikipedia.org/wiki/Semaphor_(Informatik)

Semaphor (Lösung von Dijkstra, 1965):

- Datenstruktur besteht aus einem Zähler und einer Warteschlange
- **Initialisierungsoperation**: der Zähler wird auf einen nicht negativen Wert und die Warteschlange auf leer gesetzt
- ***P*-Operation**: der Zähler wird dekrementiert, ist der Zähler jedoch kleiner als 0 wird der aufrufende Prozess blockiert und der Warteschlange hinzugefügt
- ***V*-Operation**: der Zähler wird inkrementiert. Ein Prozess wird aus der Warteschlange, falls sie nicht leer ist, entnommen und setzt die Aktionen fort, die dem *P*-Aufruf folgen.

- Eine Implementierung der Semaphor-Mechanismen ist **konzeptionell im Betriebssystem** anzusiedeln.
- Nachteil: hohe Fehleranfälligkeit

Monitor (um1973)

Funktionsweise:

- Kapselung eines kritischen Bereiches (nicht zwingend ein Objekt)
- Abstraktionsebene: höhere Programmiersprache, Synchronisationsprimitive werden automatisch vom Compiler erzeugt
- bei Betreten des Bereiches wird eine Sperre gesetzt, beim Verlassen des Bereichs wird die Sperre zurückgenommen

Java-Konstrukte:

Konstrukt	eingebautes Schlüsselwort	Java Standardbibliothek
Schlüsselwort/ Typen	synchronized	java.util.concurrent.locks.Lock

Nutzungsschema	synchronized {
	//…
	}
		lock.lock();
		//...
		lock.unlock();

synchronized

Block: `synchronized(Objekt){ /*…*/}`
ein Block mit expliziten Angabe eines zu sperrenden Objektes

Methode: `synchronized Typ name (Parameter){ /*…*/}`

```
public class Daten{
    //…
    public synchronized void method(){
        //…
    }
}
```

eine Methoden (Sperre auf this-Objekt)

- Hat ein Thread den Monitor betreten, sorgt JVM dafür, dass andere Threads warten
- ist beim Eintritt in den Monitor die Sperre bereits gesetzt, hat das aktuelle Thread zu warten bis der Konkurrent die Sperre frei gibt

```
public class Daten {
    public int mehr, weniger, summe;
}

public class MainThread {
    public static void main(String[] args) {
        Daten daten = new Daten();
        daten.mehr = 0;
        daten.weniger = 33;
        ThreadX x = new ThreadX(daten);
        ThreadY y = new ThreadY(daten);
        x.start();
        y.start();
    }
}
```

```
x:  1 + 32 = 33
y:  1 + 32 = 33
x:  2 + 31 = 33
x:  3 + 30 = 33
x:  4 + 29 = 33
x:  5 + 28 = 33
y:  5 + 28 = 33
y:  5 + 28 = 33
y:  5 + 28 = 33
y:  5 + 28 = 33
x:  6 + 27 = 33
y:  6 + 27 = 33
y:  6 + 27 = 33
y:  6 + 27 = 33
y:  6 + 27 = 33
y:  6 + 27 = 33
x:  7 + 26 = 33
x:  8 + 25 = 33
x:  9 + 24 = 33
x: 10 + 23 = 33
```

```
y:  1 + 33 =  0
y:  1 + 32 =  0
x:  1 + 32 = 33
y:  2 + 32 = 33
y:  2 + 31 = 33
x:  2 + 31 = 33
y:  3 + 31 = 33
y:  3 + 30 = 33
x:  3 + 30 = 33
y:  4 + 30 = 33
y:  4 + 29 = 33
x:  4 + 29 = 33
y:  5 + 29 = 33
y:  5 + 28 = 33
x:  5 + 28 = 33
x:  6 + 27 = 33
x:  7 + 26 = 33
x:  8 + 25 = 33
x:  9 + 24 = 33
x: 10 + 23 = 33
```

```
public class ThreadX extends Thread {
    Daten daten;

    ThreadX(Daten d) {
        daten = d;
    }
    public void run() {
        for(int i = 0; i < 10; i++) {
         synchronized(daten) {
            try {
              daten.mehr++;
              Thread.sleep(200);
              daten.weniger--;
              Thread.sleep(200);
              daten.summe = daten.mehr + daten.weniger;
              System.out.printf("x: %2d + %2d = %2d%n",
                                              daten.mehr, daten.weniger, daten.summe);
            }
            catch(InterruptedException e) {
               interrupt();
            }
         }
        Thread.yield();
     }
   }
}
```

```
public class ThreadY extends Thread {
    Daten daten;

    ThreadY(Daten d) {
        daten = d;
    }
    public void run() {
        for(int i = 0; i < 10; i++) {
          synchronized(daten) {
             try {
                System.out.printf("y: %2d + %2d = %2d%n",
                                  daten.mehr, daten.weniger, daten.summe);
                Thread.sleep(200);
             }
             catch(InterruptedException e) {
               interrupt();
             }
         }
         Thread.yield();
      }
    }
}
```

Ein gesperrtes Objekt besitzt zusätzlich zur Sperre eine Warteliste

wait, notify - Methoden der Klasse Object

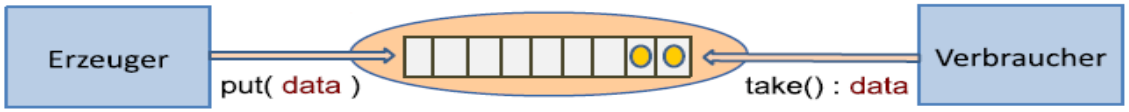
wait versetzt den aktuellen Thread in den Zustand wartend

notify weckt einen Thread, der auf Objekt wartet und ändert sein Zustand in runnable

notifyAll benachrichtigt alle wartende Threads

nur innerhalb eines synchronized-Blockes erlaubt!

Das Erzeuger-Verbraucher-Muster (producer consumer pattern)




```
public class Thread_9
{
    static public void main(String args[])
    {
        Behaelter behaelter=new Behaelter(100, 200);

        Thread thread[] = new Thread [5];

        thread[0] = new ZuThread(70, 100, behaelter);
        thread[1] = new ZuThread(40, 300, behaelter);
        thread[2] = new ZuThread(30, 600, behaelter);
        thread[3] = new AbThread(50, 500, behaelter);
        thread[4] = new AbThread(90, 200, behaelter);

        for (int i = 0; i < thread.length; i++)
          thread[i].start();
    }
}
```

Quelle: Prof. B. Steinbach, Vorlesung „Verteilte Software“

```
zu 70 level 70
zu 70 level 140
zu 40 level 180
ab 50 level 130
zu 30 level 160
zu 40 level 200
ab 90 level 110
zu 70 level 180
ab 50 level 130
zu 30 level 160
zu 40 level 200
ab 90 level 110
zu 30 level 140
```

```
class ZuThread extends Thread
{Behaelter b;
  int menge, intervall;

 //Konstruktor

  public void run()
  { for (int i=0;i<3;i++)
    {
      sleep(intervall);
      b.zu(menge);
    }
  }
}
class AbThread extends Thread
{Behaelter b;
  int menge, intervall;

  //Konstruktor

  public void run()
  { for (int i=0;i<2;i++)
    {
      sleep(intervall); }
      b.ab(menge);
    }
  }
}
```

```
class Behaelter
{
   private int min, max, level;

   Behaelter(int min, int max)
   {
     this.min = min;
     this.max = max;
     level = 0;
   }

   synchronized void zu(int x)
   {
     while ((level + x) > max) wait();
     level += x;
     System.out.println("zu "+ x +" level "+ level);
     notifyAll();
   }

   synchronized void ab(int x)
   {
     while ((level - x) < min) wait();
     level -= x;
     System.out.println("ab "+ x +" level "+ level);
     notifyAll();
}
}
```

Interface Lock, Klasse ReentrantLock
Interface Condition

Paket: java.util.concurrent.locks

Methoden: (interface Lock)

`void lock()`
markiert der Anfang des kritischen Blocks, blockiert bis der Lock frei ist

`boolean tryLock()`
wartet nicht, kehrt mit einem false zurück

`boolean tryLock(long time, TimeUnit unit)`
`throws InterruptedException`
wartet die angegebene Zeitspanne

`void unlock()`
verlässt den kritischen Block

`Condition newCondition()`
liefern ein Condition-Objekt zum Aufrufen von **await** und **signal**

Methoden des interface Condition

versetzen in Zustand wartend:

```
public void await() throws InterruptedException
public boolean await(long time, TimeUnit unit)
                                    throws InterruptedException
public boolean awaitUntil(Date deadline)
                                    throws InterruptedException
```

wecken:

```
public void signal()
public void signalAll()
```

ReadWriteLock-Konzept

unterscheidet zwischen lesenden und schreibenden Zugriffen (Sperren zum Lesen und Schreiben)

```
interface ReadWriteLock:
public Lock readLock()
public Lock writeLock()
```

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

//Erzeugung
static ReentrantLock lock=new ReentrantLock();
static Condition condition=lock.newCondition();


lock.lock();
try
{
   //Ein kritischer Abschnitt
   if (…) condition.await();
   else condition.signalAll();
}
finally
{
   lock.unlock();
}
```

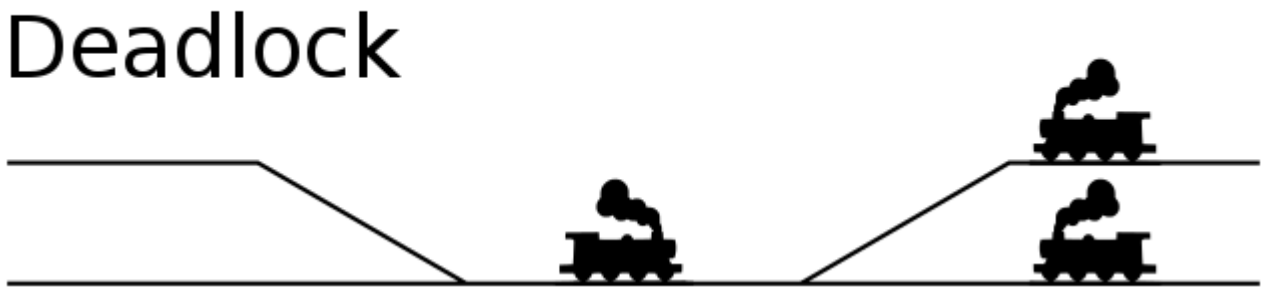
Beispiel

```
void zu(int x) throws InterruptedException {
   lock.lock();
   while ((level + x) > max) condition.await();
   level += x;
   System.out.println("zu "+ x +" level "+ level);
   condition.signalAll();
   lock.unlock();
}

void ab(int x) throws InterruptedException {
   lock.lock();
   while((level - x) < min)  condition.await();
   level -= x;
   System.out.println("ab "+ x +" level "+ level);
   condition.signalAll();
   lock.unlock();
}
```

Deadlock

Wikipedia: Deadlock oder Verklemmung bezeichnet in der Informatik einen Zustand, bei dem eine zyklische Wartesituation zwischen mehreren Prozessen auftritt, wobei jeder beteiligte Prozess auf die Freigabe von Betriebsmitteln wartet, die ein anderer beteiligter Prozess bereits exklusiv belegt hat.



Quelle: https://www.wikiwand.com/de/Deadlock_(Eisenbahn)

Thread1:
- Lock1
  - Lock2
  - //…
  - Unlock2
- Unlock1

Thread2:
- Lock2
  - Lock1
  - //…
  - Unlock1
- Unlock2

Lösung: die gleiche Reihenfolge von Sperren

Thread1:
- Empfange
- Sende

Thread2:
- Empfange
- Sende

Lösung:

Thread2:
- Sende
- Empfange

Thread-sichere Container

- synchronized-geschützte Container:
  Vector, Stack, HashTable und Dictionary

- synchronisierte Hüllklassen (Wrapper), static-Methoden zum „Einhüllen“

```
List<Person> liste = new ArrayList<Person>();
List<Person> syncListe =
    Collections.synchronizedList(liste);

Map<String,String> map = new HashMap<String,String>();
Map<String,String> syncMap =
    Collections.synchronizedMap(map);
```

- Klasse Collections bietet die **unmodifiable**-Methoden für bestehende Collections, die nur „View-Sicht“ erlauben

- Concurrent-Collections, die unterschiedliche Sicherungsalgorithmen für alle üblichen Container, implementieren

Concurrent Collections (Auswahl)

Paket : `java.util.concurrent`

List:

- `CopyOnWriteArrayList`

Set:

- `CopyOnWriteArraySet`

Map:

- Interface `ConcurrentMap`
- Implementierungen
  - `ConcurrentHashMap`

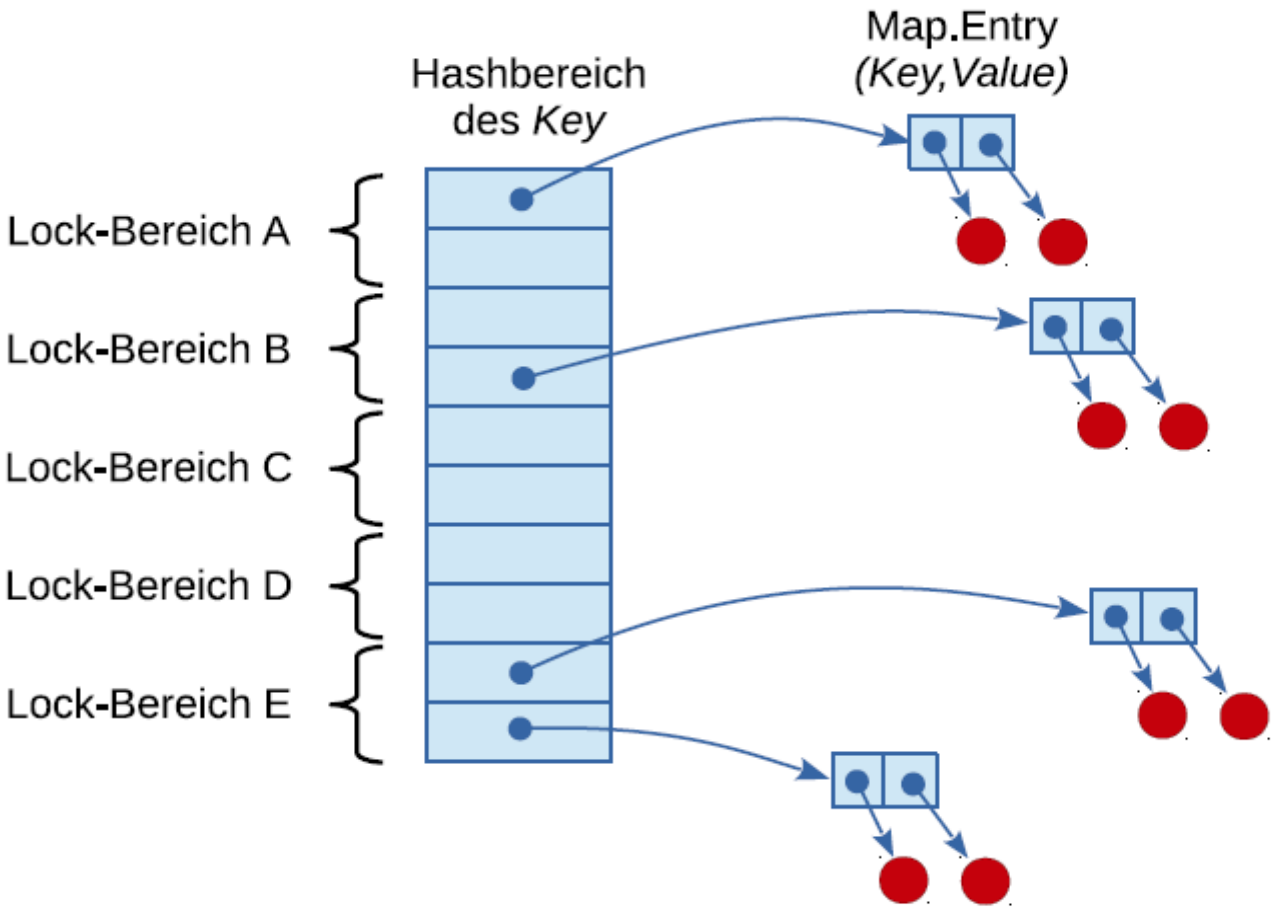
Queue:

- Interface `BlockingQueue`
- Implementierungen
  - `ArrayBlockingQueue`
  - `LinkedBlockingQueue`

CopyOn-Collections: die Änderungen an eine Kopie, das atomare Ersetzen

ConcurrentHashMap

Hashbereich wird in mehrere Lock-Bereiche aufgeteilt



Quelle: Jörg Hettel, Manh Tien Tran: Nebenläufige Programmierung mit Java, dpunkt.verlag

ArrayBlockingQueue - Beispiel

```
public class Transaction {
    public int from;
    public int to;
    public double amount;
    public Transaction(int from, int to, double amount) {
        this.from = from;
        this.to = to;
        this.amount = amount;
    }
}
```

```
public class TransactionConsumer extends Thread {
   public ArrayBlockingQueue<Transaction> transactionQueue;

   public TransactionConsumer(ArrayBlockingQueue<Transaction> transQueue){
      this.transactionQueue = transQueue;
   }

   public void run() {
      while (!isInterrupted()){
         try {
            Transaction transaction=transactionQueue.take();
            double betrag=transaction.amount;
            int von=transaction.from;
            int zu=transaction.to;
            System.out.println(betrag+" "+von+" -> "+zu);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
      }
   }}
```

```
public class TransactionProducer {
   public static void main(String[] args) {
      ArrayBlockingQueue<Transaction> transactionQueue=
                           new ArrayBlockingQueue<Transaction>(100);
      TransactionConsumer bank=new TransactionConsumer(transactionQueue);
      transactionQueue.add(new Transaction(1,2,100));
      transactionQueue.add(new Transaction(1,3,200));
      transactionQueue.add(new Transaction(1,4,100));
      bank.start();

      Scanner eingabe=new Scanner(System.in);
      double betrag=eingabe.nextDouble();
      while (betrag>0){
        int from=eingabe.nextInt();
        int to=eingabe.nextInt();
        transactionQueue.add(new Transaction(from,to,betrag));
        betrag=eingabe.nextDouble();
      }
      eingabe.close();
   }
}
```

Klasse Thread als Service (Wiederholung)

```
public class MyRunnable implements Runnable
{
   public MyRunnable() { }
   @Override
   public void run() {
      //...
   }
}

public class MainRunClass
{
   public static void main(String[] args) {
      //...
      MyRunnable myRunnable = new MyRunnable();
      Thread thread = new Thread(myRunnable);
      myThread.start();
      //...
   }
}
```

Threadpool

- häufig sind die Aufgaben nicht von sehr langer Dauer
- viele einzelne Threads belasten Betriebssystem

Lösung: Threadpools, die eine bestimmte Anzahl von Threads verwalten

- bekommen Runnable (Callable) - Objekt übergeben
- je nach Art des Pools wird das Objekt sofort einem Thread zugeteilt oder erst in eine Queue gestellt
- wenn das Runnable ausgeführt wurde, kann Threadpool weitere noch wartende Aufgaben übernehmen

Paket `java.util.concurrent`

Interfaces (einige)

`Executor` – Abstraktion aller Klassen, die Runnable-Tasks ausführen

`void execute(Runnable command)`

`ExecutorService` – stellt zusätzlich submit-Methode und weitere Methoden zur Verfügung

`void shutdown()`
`Future<V> submit(Callable<V> task)`
`Future<?> submit(Runnable task)` **–** `future.get() liefert null`
`und weitere`

`Callable<V>` - ein Task mit Ergebnis und Exception

`V call() throws Exception`

`Future<V>` - repräsentiert das Ergebnis einer (asynchronen) Ausführung

`V get()`
`V get(long timeout, TimeUnit unit)`
`boolean isCancelled()`
`boolean cancel(boolean mayInterruptIfRunning)`
`boolean isDone()`

Runnable vs. Callable

- beides sind Interfaces und haben nur eine Methode
- aber Callable liefert ein Ergebnis

```
public interface Runnable {
	public void run();
}

public interface Callable<V>{
	public V call() throws Exception;
}
```

Paket `java.util.concurrent`

Class

`Executors` – Factory- und Utility-Methoden für Interfaces und Klassen des Pakets, z.B.

Die Methoden zum Erzeugen von ExecutorService-Objekten mit verschiedenen Konfigurationen:

static ExecutorService newSingleThreadExecutor():
erzeugt ein Service, der genau einen Thread startet

static ExecutorService newCachedThreadPool():
erzeugt einen Thread-Pool mit wachender Größe

static ExecutorService newFixedThreadPool(int nThreads)
erzeugt ein Thread-Pool mit maximale Thread-Anzahl und einer Warteschlange

und weitere

Interface ExecutorService als Service für Runnable

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class MainRunClass
{
   public static void main(String[] args){
      //...
      MyRunnable myRunnable = new MyRunnable();
      ExecutorService singleThreadExecutor =
                          Executors.newSingleThreadExecutor();
      singleThreadExecutor.execute(myRunnable);
      singleThreadExecutor.shutDown();
      //ein ExecutorService ist zu beenden,
      //sonst wartet er auf weitere Runnables
      //...
   }
}
```

Interface ExecutorService als Service für Callable

```
import java.util.concurrent.Callable;

public class MyCallable implements Callable<String>
{
    public MyCallable(){
    }

    @Override
    public String call() {
        // …
        return "result";
    }
}
```

```java
import java.util.concurrent.Future;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.ExecutionException;

public class MainCallClass {
   public static void main(String[] args) {
      //...
      ExecutorService exec = Executors.newSingleThreadExecutor();
      Future<String> future = exec.submit(new MyCallable());

      try {
         System.out.println(future.get());
         // get blockiert, bis das Ergebnis zur Verfügung steht!
      }
      catch(InterruptedException | ExecutionException ex) {
         ex.printStackTrace();
      }
      exec.shutdown();
      //...
   }
}
```

```
import java.util.concurrent.Callable;

public class SumCallable implements Callable<Long> {
    private long untereGrenze, obereGrenze;

    public Long call() throws Exception {
        long sum=0;
        for (long i=untereGrenze;i<obereGrenze;i++) sum+=i;
        Thread.sleep(1000);
        return sum;
    }
    public SumCallable(long untereGrenze, long obereGrenze) {
        super();
        this.untereGrenze = untereGrenze;
        this.obereGrenze = obereGrenze;
    }
}
```

```
import java.util.concurrent.Future;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.ExecutionException;

public class SumMain{
   public static void main(String[] args) {
      ExecutorService exec = Executors.newFixedThreadPool(10);
      Future<Long> future1 = exec.submit(new SumCallable(100,1000));
      Future<Long> future2 = exec.submit(new SumCallable(1001,10000));
      try {
         System.out.println(future1.get()+future2.get());

      }
      catch(InterruptedException | ExecutionException ex) {
         ex.printStackTrace();
      }
      exec.shutdown();
   }
}
```

Literatur:

Christian Ullenboom : Java ist auch eine Insel , Rheinwerk

Jörg Hettel, Manh Tien Tran: Nebenläufige Programmierung mit Java, dpunkt.verlag

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/package-summary.html

http://www.straub.as/java/threads/