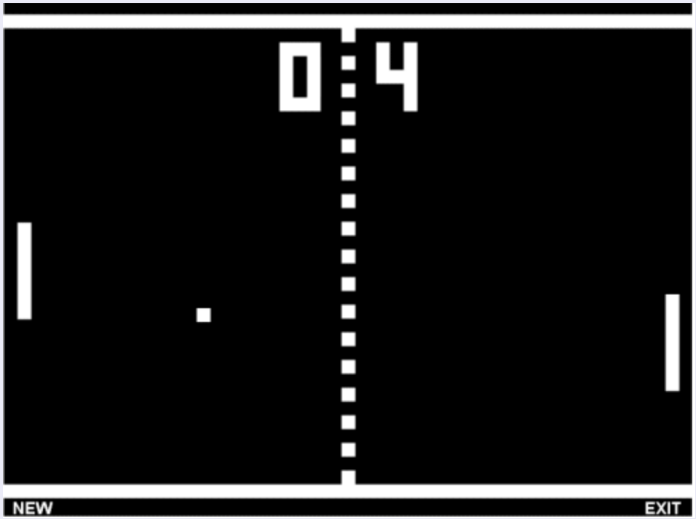
Parallelprogrammierung in Java

Motivation: Pong



Motivation

- Java bietet die Möglichkeit der Parallel-Programmierung, indem man besondere Methoden als eigene Threads (thread) laufen lassen kann.
- Man unterscheidet gemeinhin Prozesse und Threads:
  - Prozesse haben einen eigenen Adressraum (Speicherbereich) und können nur über Betriebssystem-Mittel kommunizieren.
  - Threads haben keinen eigenen Adressraum und können z.B. auch über gemeinsame Variablen kommunizieren.

Beispiel: Client-Server-Anwendungen

Problem

- Server kann zu einem Zeitpunkt nur eine Anfrage bearbeiten
- Server kann nur mit einem Client kommunizieren
- Was machen die anderen Clients?

$\Rightarrow$ Lösung: Threads verwenden

Motivation

- Jeder Thread durchläuft `run()`
- Es werden 5 Thread-Objekte definiert, die mit `start()` nebenläufig gestartet werden
- Der `start()`-Aufrauf bewirkt den `run()`-Aufruf
- Die `main()`-Methode läuft als eigener Thread
- Damit laufen 6 Threads nebenläufig

```
class MyThread extends Thread {
  public MyThread(String name){
    super(name);
  }
  @Override
  public void run() {
    for (int i  = 0; i <= 100; i++)
      System.out.println(this.getName() +
            ": " + i);
}}
class ThreadApplication {
  public static void main(String[] args) {
    for (int i  = 0; i < 5; i++) {
      Thread t  = new MyThread("MyThread "
            +i);
      t.start();
    }
    System.out.println("Main ist fertig");
  }
}
```

Ausgabe

```
MyThread 0: 0
MyThread 3: 0
MyThread 3: 1
MyThread 3: 2
...
MyThread 3: 24
MyThread 3: 25
MyThread 2: 0
MyThread 2: 1
...
MyThread 0: 4
MyThread 0: 5
MyThread 0: 6
Main ist fertig
MyThread 0: 7
MyThread 3: 8
...
```

Threads und Nebenläufigkeit

- Ein Thread ist eine Folge von Anweisungen, die nebenläufig ausgeführt werden können
- Nebenläufigkeit (concurrency)
  - (echte) Parallelität: die Threads laufen auf verschiedenen Prozessoren gleichzeitig
  - Pseudo-Parallelität: die Threads laufen auf genau einem Prozessor ab, wobei die Threads mit einer hohen Taktrate ständig geschechselt werden. Gleichzeitigkeit wird vorgetäuscht
- Jeder Thread besitzt einen eigenen Stack für Methodenaufrufe und Speicherung lokaler Variablen
- Achtung: Die Threads können Zugriff auf gemeinsame Daten haben. Synchronisation ist notwendig

Erzeugung von Threads durch Erweiterung der Klasse Thread

- Die Klasse `Thread` aus `java.lang` wird erweitert, indem die Methode `run()` überschrieben wird
- Der Aufruf der Methode `start()` der Klasse `Thread` bewirkt, dass die JVM die `run()`-Methode als `Thread` nebenläufig ausführt

```
class MyThread extends Thread {
  @Override
  public void run() {
    // ...
  }
}
class ThreadApplication {
  public static void main(String[] args) {
    Thread t = new MyThread();
    t.start();
  }
}
```

Erzeugung von Threads durch Implementierung des Interface Runnable

- Das funktionale Interface `Runnable` aus `java.lang` enthält nur die Methode `run()`
- Das Interface `Runnable` wird durch eine eigene `Runnable`-Klasse implementiert
- Ein `Thread` lässt sich dann mit Hilfe eines `Thread`-Konstruktors definieren, indem ein Objekt der `Runnable`-Klasse als Parameter übergeben wird.
- Das `Thread`-Objekt wird dann mit der Methode `start()` gestartet

```
@FunctionalInterface
interface Runnable {
  void run();
}
class MyRunnable implements Runnable {
  public void run() {
    // ..
}}
```

```
class ThreadApplication {
  public static void main(String[] args){
    Thread t = new Thread(new MyRunnable());
    t.start();
  }
}
```

Runnable-Objekte als Lambda-Ausdrücke

Da `Runnable` ein funktionales Interface ist, dürfen Lambda-Ausdrücke als `Runnable`-Objekte verwendet werden

```
Runnable myRun = () -> System.out.println("myRun laeuft");

new Thread( () -> System.out.println("myRun laeuft") ).start();
```

Mit join auf Beendigung von Threads warten

- Mit der Methode `join()` der Klasse `Thread` wird solange gewartet, bis der Thread zu Ende gelaufen ist
- `join` kann eine `InterruptedException` werfen
- Mit dem `start-join`-Konzept lassen sich sehr einfach Daten-parallele Algorithmen realisieren (Divide and Conquer)

```
public void main(String[] args) thorws InterruptedException {
  Thread t = new Thread(...);
  t.start(); // Starte Thread t
  // ...
  t.join(); // warte, bis Thread t zu Ende gelaufen ist
  // ..
}
```

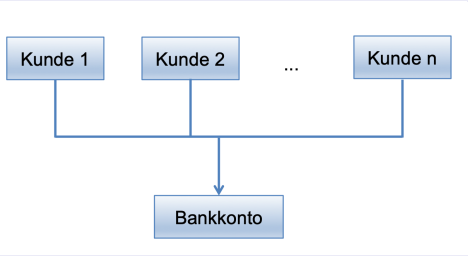
Beispiel

```
class RandomizeArrayThread extends Thread {
  private final double[] a;
  private final int li;
  private final int re;
  public RandomizeArrayThread(double[] a, int li, int
       re) {
    this.li = li;
    this.re = re;
    this.a = a;
  }
  @Override
  public void run() {
    for (int i = li; i < re; i++) {
      a[i] = (int) (1 + Math.random() * 6);
      System.out.println("Eingabe Feld a[" + i + "] "
           + a[i]);
    }
  }
}
```

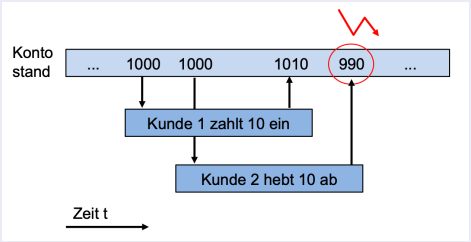
```
public class JoinApplication {
  public static void main(String[] args) throws
       InterruptedException {
    int N = 10;
    double[] a = new double[N];
    Thread t1 = new RandomizeArrayThread(a, 0, N / 2);
    Thread t2 = new RandomizeArrayThread(a, N / 2, N);
    t1.start();
    t2.start();
    t1.join();
    t2.join();
    System.out.println("--- fertig ---");
    for (int i = 0; i < a.length; i++) {
      System.out.println("Ausgabe Feld a[" + i + "] "
           + a[i]);
    }
  }
}
```

Problem bei nebenläufigem Zugriff auf gemeinsame Daten

Verschiedene Kunden greifen auf ein Konto zu



Nebenläufier Zugriff auf dasselbe Konto kann zu Inkonsistenzen führen



Problem bei nebenläufigem Zugriff: Beispiel in Java

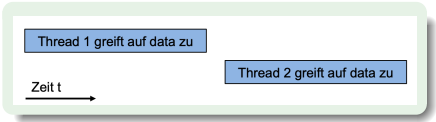
```
public class BankAccount {
  private int balance;
  public BankAccount(int initialBalance) {
    balance = initialBalance;
  }
  public int getBalance() {
    return balance;
  }
  public void deposit(int amount) {
    balance += amount;
}}

public class Customer extends Thread {
  private BankAccount account;
  private int amount;
  public Customer(BankAccount a, int d) {
    account = a;
    amount = d;
  }
  public void run() {
    for (int i = 0; i < 1000; i++)
    account.deposit(amount);
}}
```

```
public class BankTest {
  public static void main(String[] args) throws
       InterruptedException {
    BankAccount a = new BankAccount(1000);
    Thread kunde1 = new Customer(a, +10);
    Thread kunde2 = new Customer(a, -10);
    kunde1.start();
    kunde2.start();
    kunde1.join();
    kunde2.join();
    System.out.println(a.getBalance());
}}
// Es werden 2 Kunden gestartet
// Kunde 1 zahlt 1000mal 10 ein
// Kunde 2 hebt 1000mal 10 ab
// Kontostand hat nicht immer den erwarteten Wert
     balance=1000
```

Synchronisierung mit synchronized-Methode

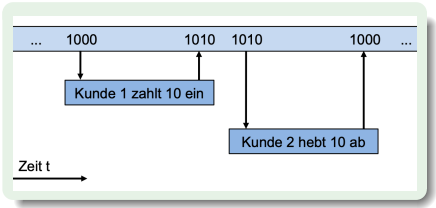
- Bein Eintritt in eine `synchronized`-Methode wird das Objekt gesperrt und bei Austritt wieder freigegeben (locking-Mechanismus)
- Zu einem Zeitpunkt darf daher höchstens ein Thread auf ein gemeinsames Objekt mit einer `synchronized`-Methode zugreifen
- Der Thread, der ein gesperrtes Objekt bearbeiten möchte, wird blockiert, bis das Objekt weider freigegeben wird.
- Beachte: auf verschiedene Objekte darf gleichzeitig zugegriffen werden

```
class GemeinsameDaten {
  public synchronized ... tueEtwas1() {...}
  public synchronized ... tueEtwas2() {...}
  ...
}
GemeinsameDaten data = new GemeinsameDaten();
```



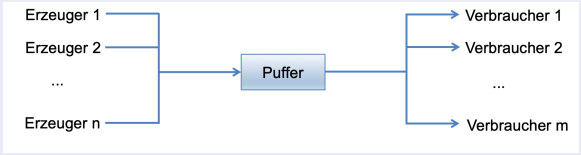
Beispiel mit synchronized in Java

```
public class BankAccount {
  private int balance;
  public BankAccount(int initialBalance) {
    balance = initialBalance;
  }
  public synchronized int getBalance() {
    return balance;
  }
  public synchronized void deposit(int amount) {
    balance += amount;
}}
```



Erzeuger/Verbraucher-Problem

- Es gibt Erzeuger-Threads, die Daten und in Puffer (z.B. Queue) schreiben
- Es gibt Verbraucher-Threads, die Daten vom Puffer holen und verarbeiten
- Zugriff auf Puffer muss synchronisiert werden
- Verbraucher-Threads müssen warten, falls Puffer leer ist
- Falls Erzeuger-Threads Daten im Puffer ablegt, dann müssen wartende Verbraucher benachrichtigt und aktiviert werden.
- Puffer hat begrenzte Kapazität, so dass Erzeuger ggf. warten und vom Verbraucher benachrichtigt werden müssen



Methoden `wait, notify, notifyAll`

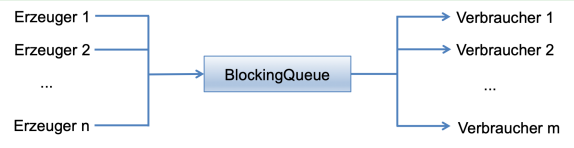
- Mit der Methode `wait` wird ein Thread solange in den Wartezzustand gesetzt, bis eine Bedingung B erfüllt ist. `wait` erfolgt in einer Schleife, da bei der Aktivierung des Threads Bedingung erneut gebprüft werden muss
- Mit der Methode `notifyAll` werden alle wartenden Threads wieder aktiviert
- Mit der Methode `notify` wird irgendein wartender Thread aktiviert
- `wait` und `notifyAll/notify` sollen in `sychronized`-Methoden aufgerufen werden, da auf gemeinsame Daten zugegriffen wird
- `wait, notifyAll/notify` sind in der Klasse`Object` definiert

Methoden `wait, notify, notifyAll`

```
synchronized void doWhenCondition() {
  while (!B)
    wait();
  // Zugriff auf gemeinsame Daten
}

synchronized void changeCondition() {
  // Zugriff auf gemeinsame Daten
  // Bedingung B kann sich un geaendert haben
  // Daher wartende Threads benachrichtigen, um Bedingung B neu zu pruefen
  notifyAll(); // notify();
}
```

Beispiel mit Queue

- Verschiedene Erzeuger-Threads schreiben Daten in Queue
- Verbraucher-Threads holen Daten aus der Queue
- Verbraucher-Threads müssen warten, falls Queue leer
- Sobald Erzeuger-Thread Daten in Queue schreibt, wird irgendein Verbraucher aktiviert



Beispiel mit Queue

```
class BlockingQueue {
  privat final Queue<Integer> myQueue = new LinkedList<>();
  public synchronized void add(int x) {
    myQueue.add(x);
    notify();
  }
  public synchronized int remove() throws InterruptedException {
    while (myQueue.isEmpty())
      wait();
    return myQueue.poll();
  }
}
```

Beispiel mit Queue

```
class Producer extends Thread {
  private final BlockingQueue bq;
  private final int start;
  public Producer(BlockingQueue bq, int
      x) {
    this.bq = bq;
    this.start = s;
  }
  public void run() {
    for (int i = start; i < start+100; i
        ++)
    bq.add(i);
  }
}
```

```
class Consumer extends Thread {
  private final BlockingQueue bq;
  private final String name;
  public Consumer(BlockingQueue bq,
      String n) {
    this.bq = bq;
    this.name = n;
  }
  public void run() {
    for (int i = 0; i < 150; i++)
    try {
      System.out.println(name + ": "+bq.
          remove())
    } catch (InterruptedException ex) {}
  }
}
```

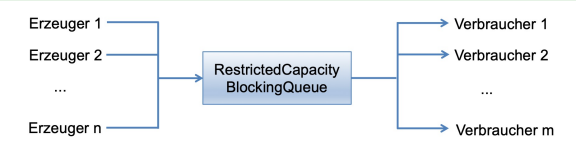
Beispiel mit Queue

```
public static void main (String[] args) {
  BlockingQueue bq = new BlockingQueue();
  Producer p1 = new Producer(bq, 0);
  Producer p2 = new Producer(bq, 1000);
  Producer p3 = new Producer(bq, 1000_000);
  Consumer c1 = new Consumer(bq, "consumer1");
  Consumer c2 = new Consumer(bq, "consumer2");
  p1.start();
  p2.start();
  p3.start();
  c1.start();
  c2.start();
}
```

- Es werden 3 Producer-Threads gestartet, die insgesamt 300 Zahlen in die BlockingQueue schreiben
- Es werden 2 Consumer-Threads gestartet, die insgesamt 300 Zahlen aus der BlockingQueue holen und ausgeben

Beispiel mit kapazitätsbegrenzter Queue

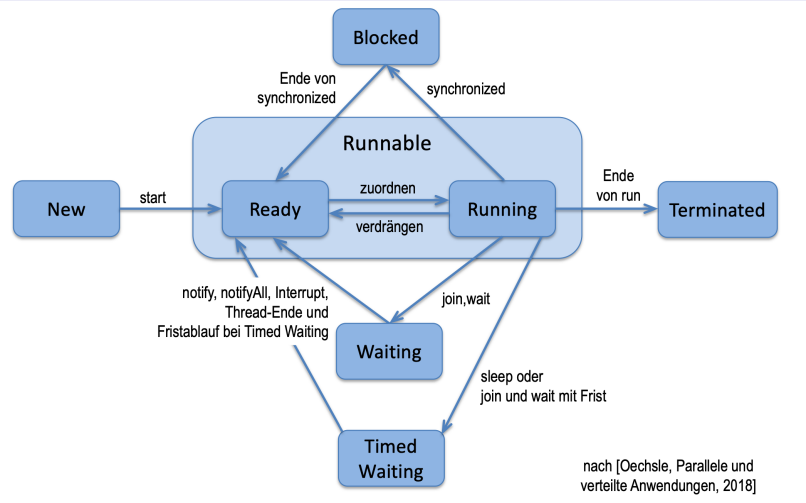
- Verschiedene Erzeuger-Threads schreiben Daten in kapazitätsbegrenzte Queue
- Verbraucher-Threads holen Daten aus der Queue
- Verbraucher-Threads müssen warten, falls Queue leer. Sobald Erzeuger-Thread Daten in Queue schreibt, werden alle wartenden Verbraucher aktiviert
- Erzeuger-Threads müssen warten, falls Queue voll ist. Sobald ein Verbraucher-Thread aus der Queue holt, werden alle wartenden Threads aktiviert



Beispiel mit kapazitätsbegrenzter Queue

```
class RestrictedCapacityBlockingQueue {
  privat final Queue<Integer> myQueue = new LinkedList<>();
  private final int cap = 5;
  public synchronized void add(int x) throws InterruptedException {
    while (myQueue.size() >= cap)
      wait();
     myQueue.add(x);
    System.out.println("added: "myQueue.size());
    notifyAll();
  }
  public synchronized int remove() throws InterruptedException {
    while (myQueue.isEmpty())
      wait();
    int x = myQueue.poll();
    System.out.println("removed: "myQueue.size());
    notifyAll();
    return x:
  }
}
```

Zustände eines Java-Threads



Überblick über Thread-sichere Typen

`java.util.concurrent.atomic`

- Klassen bzw. Methoden: `AtomicInteger, AtomicIntegerArray`

⇒ Beschreibung: Verschiedene gekapselter Basistypen und Felder, die Thread-sicher sind

`Collection`

- Klassen bzw. Methoden: `synchronizedCollection(c), synchronizedList(I), syhcnronizedMap(m), sychronizedSet(s), ...`

⇒ Beschreibung: Verschiedene statische Methoden zum Einhüllen, so dass Thread-Sicherheit gewährleistet ist.

Überblick über Thread-sichere Typen

`java.util.concurrent`

- Klassen bzw. Methoden: `BlockingQueue, ConcurrentMap, ...`

⇒ Beschreibung: Verschiedene Thread-sichere Typen

`Collection`

- Klassen bzw. Methoden: `unmodifiableCollectionc, unmodifiableList(I), unmodifiableMap(m), unmodifiableSet(s), ...`

⇒ Beschreibung: Verschiedene statische Methoden zum Einhüllen von `Collection`-Zypen, so dass sie immutabel und damit Thread-sicher werden

Beispiel mit `AtomicInteger`

```
// Ausschnitt java.util.concurrent.atomic
class AtomicInteger {
  AtomicInteger(int initialValue)
  int get() {...}
  int addAndGet(int delta) {...}
  boolean compareAndSet(int expect, int update) {...}
  int accumulateAndGet(int x, IntBinaryOperator f) {...}
  // ...
}
public static void main(String[] args) {
  AtomicInteger sum = new AtomicInteger(0);
  class RandomSumThread extends Thread {
    public void run() {
      for (int i = 0; i<=1000;i++) {
        int r= (int) (Math.random()*100);
        sum.accumulateAndGet(r, (x,y) -> x+y);
      }
    }
  }
  RandomSumThread t1 = new RandomSumThread(); t1.start();
  RandomSumThread t2 = new RandomSumThread(); t2.start();
  t1.join(); t2.join();
  System.out.println("Sum = "+ sum.get());
}
```

Synchronisierte `Collections`

Die Klasse `Collections` enthält verschiedene

```
List<Integer> intList = new LinkedList<>();
List<Integer> synchList = Collections.synchronizedList(intList);
Map<String, Integer> telBuch = new TreeeMap<>();
Map<String, Integer> synchTelBuch = Collections.synchronizedMap(telBuch);
```

Der Zugriff auf das `Collection`-Object ist synchronisiert

```
List<Integer> intList = new LinkedList<>();
List<Integer> synchIntList = Collections.synchronizedList(intList);
class RandomThread extends Thread {
  public void run() {
    for (int i = 0; i < 1000; i++)
      synchIntList.add(Math.random());
}}
new RandomThread().start();
new RandomThread().start();
```

Synchronisierte `Collections`

Wird in einem Thread über das `Collection-Object c` iteriertund in einem anderen Thread das Object `c` verändert, kann eine ConcurrentModificationException ausgelöst werden

```
List<Double> dbList = new LinkedList<>();
List<Double> synchDoubleList = Collections.synchronizedList(dbList);
class RandomThread extends Thread {
  public void run() {
    for (int i = 0; i < 1000; i++) {
      synchDoubleList.add(Math.random());
      for (double x : synchDoubleList)
        System.out.println(x);
}}}
new RandomThread().start();
new RandomThread().start(); // Vorsicht: ConcurrentModificationException
```

Abhilfe: Iterator-Schlefe in einem `synchronized`-Block

Immutable `Collections`

- Die Klasse `Collections` enthält verschiedene statische Methoden, um en `Collection`-Object in eine Hülle zu packen, so dass nur lesende Operationen durchgeführt werden können. Container wird damit immutabel.
- Es können dann problemlos mehrere Threads lesend auf das `Collection`-Object ohne zusätzliche Sychronisation zugreifen.

```
List<Integer> intList = new LinkedList<>();
intList.add(5);
intList.add(7);
// ...
List<Integer> constList = Collections.unmodifiableList(intList);
Map<String, Integer> telBuch = new TreeMap<>();
telBuch.put("Maier", 1234);
telBuch.put("Anton", 5678);
// ...
Map<String, Integer> constTelBuch = Collections.unmodifiableMap(telBuch);
```

Das Concurrent-Paket

Das Concurrent-Paket

- bisher die grundlegenden Konzepte zur Parallelprogrammierung besprochen
  - Klassen, Schnittstellen: `Thread, Runnable`
  - Synchronisationsmechanismen für Blöcke und Methoden: `synchronized`
  - Methoden der Klasse `Object`: `wait, notify, notifyAll`
- In diesem Abschnitt geht es um `java.util.concurrent`
  - `Executor, Executors, ExecutorService`
  - `Lock, ReentrantLock, Condition`
  - `ArrayBlockingQueue`

Motivation

Die Erzeugung und der Start eines Threads ist wesentlich aufwändiger als der Aufruf einer Prozedur oder Methode

- Thread aus einem Pool von Threads verwenden und ihm die Task zur Ausführung geben. Wenn die Task beendet wird, geht der Thread wieder in den Pool zurück.
- Begrenzung der Anzahl der Threads, die gestartet werden können. Denn wenn die von Java oder dem zugrundeliegenden Betriebssystem gesetzte Grenze überschritten wird, kommt es zum Absturz.

Threadpools

- Threadpools können dann am besten eingesetzt werden, wenn die Tasks homogen und unabhängig sind.
- Eine Mischung von langlaufenden und kurzen Tasks führt leicht zu einer Verstopfung des Threadpools, so dass die Ausführung kurzer Tasks unerwartet lange dauert

Achtung

Bei voneinander abhängigen Tasks riskiert man ein Deadlock, indem z.B. eine Task unendlich lange im Threadpool verbleibt, weil sie auf das Ergebnis einer anderen Task wartet, die aber nicht laufen kann, weil der Threadpool voll ist

Zentrale Schnittstelle `Executor`

```
public interface Executor
{
  void execute (Runnable task);
}
```

- Die Klasse `ThreadPoolExecutor` ist eine Realisierung dieser Schnittstelle
- Die einfachste Möglichkeit zu einem Threadpool zu kommen, besteht im Aufruf der static Methode `newFixedThreadPool (int anzThreads)` der Klasse Executors. Rückgabe ist ein Objekt vom Typ `ExecutorService`, das die Schnittstelle `Executor` mit der Methode `execute (Runnable)` realisiert

Beispiel

```
public class Task implements Runnable {
  private String name;
  private int wartezeit;
  public Task(String name, int wartezeit) {
    this.name = name;
    this.wartezeit = wartezeit;
  }
  public void run() {
    System.out.println("Task " +name+ " beginnt um "+ System.currentTimeMillis() + ".");
    try {
      Thread.sleep(wartezeit);
    } catch (InterruptedException e) {}
    System.out.println("Task " + name + " ist fertig um " + System.currentTimeMillis() + "
        .");
  }
}
```

Aufgabe

- Um diese `Task` auszuführen, benutzen wir einen Thread-Pool. In dem folgenden Programm erzeugen wir einen Pool der Größe 3 und lassen 10 Tasks durchführen. Schließlich warten wir, bis alle Tasks zu Ende gekommen sind.
- nach einem `shutdown()` wird der Threadpool nicht verwendete Threads abschalten und keine neuen Threads zum Laufen bringen
- Allerdings muss er warten, bis laufende Threads zu Ende gekommen sind. Das kann eine gewisse Zeit dauern. Daher gibt es die Methode `awaitTermination (long timeout, TimeUnit timeunit)`, um auf die Beendigung zu warten.

Beispiel

```
public class ExecuteTask {
  public static void main(String[] arg) {
    ExecutorService executor = Executors.newFixedThreadPool(3);
    int sumWartezeit = 1000;
    java.util.Random zufall = new java.util.Random();
    for (int i = 0; i < 10; i++) {
      String name = "" + i;
      int wartezeit = zufall.nextInt(1000);
      sumWartezeit = sumWartezeit + wartezeit;
      Runnable task = new Task(name, wartezeit);
      System.out.println("Task " + name + " mit Wartezeit " + wartezeit + " wird an den Threadpool uebergeben.");
      executor.execute(task);
    }
    try {
      Thread.sleep(sumWartezeit);
      executor.shutdown();
      executor.awaitTermination(sumWartezeit, TimeUnit.MILLISECONDS);
    } catch (InterruptedException e) {}
  }
}
```

Beispiel: Erzeuger/Verbraucher

- Der Buffer besteht aus eier einzelnen `int`-Variablen
- Der Erzeuger schreibt die Werte $1, \ldots, 10$ nacheinander in den Puffer
- Der Verbraucher liest die Werte aus dem Puffer und addiert sie
- Bei korrektem Programmablauf ist die Summe 55

```
public interface Buffer {
  public void set (int value);
  public int get();
}
```

Cont.

```
public class Producer implements Runnable{
  private static Random generator = new Random();
  private Buffer sharedLocation;
  public Producer(Buffer shared) {sharedLocation =
       shared;}
  public void run() {
    int sum = 0;
    for(int count = 1; count <=10; count++) {
      try {
        Thread.sleep(generator.nextInt(3000));
        sharedLocation.set(count);
        sum += count;
        System.out.println(sum);
      } catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
    System.out.println("Terminating Producer.");
  }
}
```

```
public class Consumer implements Runnable {
  private static Random generator = new Random();
  private Buffer sharedLocation;
  public Consumer(Buffer shared) {
    sharedLocation = shared;
  }
  public void run() {
    int sum = 0;
    for (int count = 1; count <= 10; count ++) {
      try {
        Thread.sleep(generator.nextInt(3000))
         sum += sharedLocation.get();
         System.out.println(sum);
      } catch (InterruptedException ex) {
        ex.printStackTrace();
      }
    }
    System.out.println("Terminating Consumer.");
  }
}
```

Cont.

```
public class BufferRealization
    implements Buffer {
  private int buffer = -1;
  public void set(int value) {
    System.out.println("Producer writes
        "+ value);
    buffer = value;
  }
  public int get() {
    System.out.println(Consumer reads "+
        buffer);
    return buffer;
  }
}
```

```
public class SharedBufferedTest {
  public static void main (String[] args
      ) {
    ExecutorService app = Executors.
        newFixedThreadPool(2);
    Buffer sharedLocation = new
        BufferRealization();
    try {
      app.execute(new Producer(
          sharedLocation));
      app.execute(new Consumer(
          sharedLocation));
    } catch (Exception ex) {
      ex.printStackTrace();
    }
    app.shutdown();
  }
}
```

Problem

In der eben vorgestellten Version kann es zu Verdoppelung von Daten oder zu Datenverlust kommen

- Verdoppelung von Daten: Der Konsument liest mehrfach den gleichen Wert
- Datenverlust: Der Produzent schreibt mehrfach nacheinander, ohn dass der Konsument zwischendurch den Wert abholt
- Es ntsteht ein nichtdetrministischer Wert zwischn -10 und 100

Lock, ReentrantLock, Condition

- Die Schnittstlle `Lock` beschreibt u.a. die Methode `lock` und `unlock` zum Setzen/Lösen von Sperren auf ein Objekt sowie die Methode `newCondition` zum Erzeugen einer Bedingungsvariablen
- Bedigungsvariablen werden durch die Schnittstelle `Condition` beschrieben. SIe definiert u.a. die Methiden `await, awaitUntil, signal` sowie `signalAll` für die entsprechenden Monitor-Operationen
- Die Klasse `ReentrantLock` implementiert die Schnittstelle `Lock` Neben dem Standardkonstruktor `ReentrantLock()` gibt es den Konstruktor `ReentrantLock(boolean fair)`, der eine „faire Behandlung“ der wartenden Threads sicherstellt

Beispiel: Erzeuger/Verbraucher (Synchronisiert)

```
public class SynchronizedBuffer implements Buffer {
  private Lock accessLock = new ReentrantLock();
  private Condition canWrite = accessLock.newCondition();
  private Condition canRead = accessLock.newCondition();
  private int buffer = -1;
  private boolean occupied = false;
```

Cont.

```
public void set(int value) {
  accessLock.lock();
  try {
    if (occupied) {
      canWrit.await();
    }
    buffer = value;
    occupied = true;
    canRead.signal();
  } catch (InterruptedException ex){
    ex.printStackTrace();
  } finally {
    accessLock.unlock();
  }
}
```

Cont.

```
  public int get() {
    int readValue = 0;
    accessLock.lock();
    try {
      if (!occupied) {
        canRead.await();
      }
      occupied = false;
      readValue = buffer;
      canWrite.signal();
    } catch (InterruptedException ex){
      ex.printStackTrace();
    } finally {
      accessLock.unlock();
    }
    return readValue;
  }
}
```

Cont.

```
public class SharedBufferedText2 {
  public static void main(String[] args) {
    ExecutorService app = Executors.newFixedThreadPool(2);
    Buffer sharedLocation = new SynchronizedBuffer();
    try {
      app.execute(new Producer(sharedLocation));
      spp.execute(new Consumer(sharedLocation));
    } catch (Exception ex){
      ex.printStackTrace();
    }
    app.shutdown();
  }
}
```

Lösung mit ArrayBlockingQueue

- `ArrayBlockingQueue<E>` implementiert die Schnittstelle `Queue`
- Die Methoden `put` und `take` schreiben ein Element in die Schlange bzw. lesen ein Element aus der Schlange. `put` und `take` warten, falls die Schlange voll/leer ist.

`ArrayBlockingQueue<E>` verwendet zur Implementierung ein Feld fester Größe. Zwei Konstruktoren sind

```
ArrayBlockingQueue(int capacity);
ArrayBlockingQueue(int capacity, boolean fair);
```

Beispiel: Erzeuger/Verbraucher (ArrayBlockingQueue)

```
public calss BlockingBuffer implements Buffer {
  private ArrayBlockingQueue<Integer> buffer;
  public BlockingBuffer() {
    buffer = new ArrayBlockingQueue<Integer>(3);
  }
  pulic void set(int value) {
    truy {
      buffer.put(value)
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
  public int get() {
    int readValue = 0;
    try {
      readValue = buffer.take();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
    return readValue;
  }
}
```

Ausblick: Weitere Klassen und Schnittstellen der Java-API

- Die generische Schnittstelle `callable<E>` ist ähnlich zur Schnittstelle `Runnable`. Sie enthält die Methode `call`, die aber im Gegensatz zur Methoden `run` einen Wert liefert oder eine Ausnahme auslösen kann
- Nebenläufige Implementierungen der Schnittstelle `Map, List` und `Queue`, z.B. `ConcurrentHashMap<K,V>, CopyOnWriteArrayList<E>` und `LinkedBlockingDeque<E>`
- Klassen für Synchronisationsmechanismen, z.B. `Semaphore`