1 Concurrent Klassen

`ConcurrentLinkedQueue<E>` mit `.add()`, `.remove()`, `.addFirst(e)`, `.removeFirst()`, `.getFirst()` sowie jeweils mit `...Last()`
`ConcurrentHashMap<K,V>` mit `.put()`, `.get()`, `.entrySet()`, `.values()`
`List<String> list = new ArrayList<String>();`
`List<String> l = Collections.synchronizedList(list);`
auch ...Collection ...Set

2 Threads

Für den Start des Threads muss die Methode `start()` aufgerufen werden. Ein Aufruf von `run()` würde blockieren.

```
new Thread(() -> {
  try { function(); }
  catch (InterruptedException e) {errFun(); } });

public class SimpleThread extends Thread {
  @Override
  public void run() { function(); }}

new SimpleThread('.', 10).start();

public class RThread implements Runnable {
  @Override
  public void run() { function(); }
}
new Thread(new RunnableThread('.', 10));
```

Eine vierte Möglichkeit wäre eine anonyme innere Klasse.

`t.setDaemon (true);` verhindern das Beenden des Programms NICHT. Der Garbage Collector ist ein solcher Thread.

3 Synchronisationsprimitiven

3.1 Monitor

Die Synchronisierung mittels **synchronized** ist reentrant.

```
public class MyClass {
  public synchronized void one() { dangerous_things(); }
  public synchronized void two() { dangerous_things(); }
  public void partlySync() {
    non_dangerous_things();
    synchronized(this) { dangerous_things(); }
  }
  public synchronized void put(int amount)
  throws InterruptedException {
    while (stock + amount > capacity) { wait(); }
    stock += amount; notifyAll();
  }
}
```

Mit `wait(long timeout)` kann man auch nach einem Timer automatisch geweckt werden.

```
private object syncObject = new object();
lock(syncObject) {
   while (amount > balance) { Monitor.Wait(syncObject);
   balance -= amount;
}
```

3.2 Semaphoren

Semaphore sind Marken und können auch negativ initialisiert werden. Falls keine Marke frei ist (<=0), blockiert `acquire()`.

```
public class Warehouse {
  private final Semaphore upperLimit(cap, fair);
  private final Semaphore lowerLimit(0, fair);

  @Override
  public void put(int amount)
    throws InterruptedException {
    upperLimit.acquire(amount);
    lowerLimit.release(amount);
  }
}
```

3.3 Locks & Conditions

Wichtig: bei `await()` wird der Lock zwischenzeitig abgegeben. Er muss auch wieder erlangt werden. Es gibt auch `ReentrantReadWriteLock()` mit `rwl.readLock().unlock()`, `rwl.writeLock().lock()` und `rwl.writeLock().newCondition()`

```
public class Warehouse {
  private final Lock monitor;
  private final Condition nonFull, nonEmpty;
  private final int capacity, stock;

  public Warehouse(int capacity, boolean fair) {
    monitor = new ReentrantLock(fair);
    nonFull = monitor.newCondition();
    nonEmpty = monitor.newCondition();
    this.capacity = capacity;
  }
  @Override
  public void put(int amount)
  throws InterruptedException {
    monitor.lock();
    while (stock + amount > capacity){nonFull.await();}
    stock += amount; nonEmpty.signalAll();
    monitor.unlock();
  }
}
```

3.4 CountDownLatch

"Einweg-Synchronisationspunkt". Kann **nicht wiederverwendet** werden.

```
public class RaceControl {
  CountDownLatch carsReady =
    new CountDownLatch(NOF_RACE_CARS);
  CountDownLatch startSignal =
    new CountDownLatch(1);

  protected void waitForAllToBeReady()
    throws InterruptedException {
    carsReady.await();
  }

  public void readyToStart() {
    carsReady.countDown();
  }
}
```

3.5 CyclicBarrier

Die Cyclic Barrier startet automatisch sobald N Threads warten:

```
CyclicBarrier raceStart = new CyclicBarrier(N);
raceStart.await();
```

Die Barrier kann auch für mehrere Runden eingesetzt werden. Bei einer Exeption z.B. `InterruptedException`, sind alle betroffen. `getParties()` ermittelt die Anzahl Teilnahmer der Barriere.

3.6 Exchanger

Ermöglicht es, zwischen zwei Threads Objekte auszutauschen. `exchange(obj1)` wartet, bis der andere Thread auch `exchange(obj2)` aufgerufen hat.

```
Exchanger<Integer> exchanger = new Exchanger<>();
int out = exchanger.exchange(in);
```

4 Race Conditions, Deadlocks & Starvation

4.1 Race Conditions

Data Races treten auf wenn unsynchronisiert read/write auf den gleichen Speicher (Variable, Array-Element, ...). Race-Conditions treten auf, wenn die Critical Sections nicht ausreichend geschützt sind. Collections aus `java.util` nicht threadsafe. Auf Synchronisierung kann **verzichtet** werden, wenn entweder Unveränderlichkeit (z.B. Java **final**) oder veränderliche Objekte in Objekte eingesperrt sind (Object Confinement).

4.2 Deadlocks

treten unter folgenden Bedingungen auf: wenn *geschachtelte Locks*, *zyklische Wartеabhängigkeiten*, *gegenseitiger Ausschluss* und *kein Timeout* gemeinsam vorkommen.
Sie lassen sich durch Einführung einer *linearen Sperrordnung* oder *grobgranulare Sperrung* lösen.
Live-Locks sind Deadlocks, bei denen permanent CPU verbraucht wird.

4.3 Starvation

tritt auf, wenn einem Thread immer wieder die Möglichkeit zu arbeiten weggeschnappt wird. Dies lässt sich durch faire Synchronisationsprimitiven lösen.

5 Thread Pools

besitzen eine *Task Queue*, in welche Tasks eingereiht werden; von dort werden sie dann von einem `Worker Thread` herausgenommen und bearbeitet. Worker Threads sind Daemon-Threads. Seit Java 7/8 gibt es den `ForkJoinPool`. Beim Beenden des Programms werden die Worker-Threads einfach terminiert, weil sie Deamon-Threads sind.

```
ForkJoinPool tp = new ForkJoinPool();
ForkJoinTask<?> l = tp.submit(() -> count(lPart));
ForkJoinTask<?> r = tp.submit(() -> count(rPart));
int result = l.join() + r.join();
```

5.1 Recursive Action

```
public class ParAction extends RecursiveAction {
  @Override
  protected void compute() {
     ParAction left = new ParAction(array,
     l, (l + r) / 2);
     ParAction right = new ParAction(array,
     (l + r) / 2, r);
     invokeAll(left, right); // ist fork() und join()
  }
}
class CountTask extends RecursiveTask<Integer> {
    protected Integer compute() {
        CountTask left = new CountTask(leftPart);
        CountTask right = new CountTask(rightPart);
        left.fork(); right.fork();
        return right.join() + left.join();
    }
}
```

```
pool.invoke(new MergeSortTask(array, 0, array.length));
// oder
new MergeSortTask(array, 0, array.length).invoke();
```

Man kann auch den Pool auswählen. Entweder einen eigenen ForkJoinPool, oder den *Common Pool*. Diesen teilt man mit dem Rest der JVM, er wird noch nicht empfohlen. Direkt auf dem Common Pool starten: `new CountTask(args).invoke();`

5.2 Work Stealing

Der Pool hat eine globale Task Queue, die Worker Threads eine lokale Queue damit sie nicht nach jedem Task die globale sperren müssen, um neue zu holen. Wenn ein Worker Thread seine Queue abgearbeitet hat, und er keine mehr in der globalen findet, kann er auch Tasks aus der Queue eines anderen Worker Threads holen.

6 Future

`CompletableFutures` laufen automatisch auf dem Common Pool.

```
public CompletableFuture<String> asyncDl(String l) {
  //.runAsync ohne Rückgabewert also
  //CompletableFuture<Void>
  return CompletableFuture.supplyAsync(
    () -> downloadUrl(l));
}

CompletableFuture<String> download =
downloader.asyncDownloadUrl(link);
download.thenAccept(
  result -> System.out.println(result));
System.out.println(download.get());

CompletableFuture<String> f = CompletableFuture
.supplyAsync(()-> "3")
.thenApply((String e) -> "abc");
```

`thenApply((x) -> {})` für Funktion mit Rückgabe (CompletableFuture<T> nachher), `thenAccept((x) -> {})` für Handler ohne Rückgabe (CompletableFuture<Void> nachher) (typischerweise am Ende). `thenRun(() -> {})` ohne Argument. `.allOf(fut1, fut2).thenAccept(...)` wenn alle fertig sind, `.anyOf(fut1, fut2).thenAccept(...)` wenn einer davon fertig ist. Ein leerer future ist `CompletableFuture.runAsync(() -> {})`. Mit all.join() bekommt man null oder ein Resultat.

7 .NET und TPL

7.1 TPL (Task Parallel Library)

ist ein Work Stealing Thread Pool. Die TPL erkennt geschachtelte Tasks automatisch. Die TPL erzeugt selber neue Tasks, wenn sie merkt dass der Durchsatz sinkt (Hill-Climbing Algorithmus). Somit können Deadlocks bei Task-Abhängigkeiten verhindert werden (ausser man setzt die maximale Anzahl mit `ThreadPool.SetMaxThreads(n)`).

```
Task<int> task = Task.Run(() => {
  return 42;
});
task.Result; // blockiert
// task.Wait ohne <> Rückgabewert

Task.Run(() => {
  Task<int> left = Task.Run(
    () => Count(leftPart));
  Task<int> right = Task.Run(
    () => Count(rightPart));
  int result = left.Result + right.Result
});
task.Result; // blockiert
// task.Wait ohne <> Rückgabewert
```

7.2 Einfache Parallelität

```
Parallel.For(0, array.Length,
  i => DoComputation(array[i])
);
IEnumerable<string> left = null, right = null;
Parallel.Invoke(
    () => QuickSort(array, left, right),
    () => QuickSort(array, left, right)
    () => left = Generate(k),
^^I() => right = Generate(nofPairs - k - 1)
);

bookCollection.AsParallel().AsOrdered().
  Where(book => book.Title.Contains("Concurrency")).
  Select(book => book.ISBN)

//java
bookCollection.parallelStream().unordered().
filter(book -> book.getTitle().contains("Concurrency"))
map(book -> book.getISBN());

List<Task> tasks = new List<Task>();
foreach(string link in links) {
    tasks.Add(Task.Run(() => DownloadWebsite(link)));
}
Task.WhenAll(tasks).ContinueWith(
  pd => Console.WriteLine("{t} ms")).Wait();
```

7.3 async/await

erlauben das teil-asynchrone Ausführen von Funktionen. Der Compiler zerlegt die **async**-Funktion in zwei Hälften: bis zum ersten **await** wird die Funktion vom Caller synchron ausgeführt. Der Rest wird auf einem TPL-Thread erledigt.
await darf nur in **async**-Funktionen vorkommen; **async**-Funktionen müssen ein **await** enthalten.
Hinter den Kulissen baut **await** die nachfolgenden Aufrufe in eine Continuation. Ist der Aufrufer ein UI-Thread, dann wird die Continuation auf den UI-Thread dispatched, andernfalls auf einen normalen TPL-Threads.
Achtung: `async/await` kann zu einem Threadwechsel innerhalb eines Funktionsaufrufs führen!

8 Java UI

```
public ComputerGUI(Computer computer) {
  this.computer = computer;
  computer.addObserver(this);
}

ForkJoinPool tp = new ForkJoinPool();
startButton.addActionListener(event -> {
  tp.submit(() -> {
    String r = computer.calc();
    SwingUtilities.invokeLater({
      () -> resultLabel.setText("Result: " + r));
      // kann auch rekursiv nochmal tp.submit(...)
      // und auch dort nochmal .invokeLater(...)
  })
});
public void update(Observable o, Object arg) {
    SwingUtilities.invokeLater(
      () -> statusLabel.setText(computer.getStatus()));
      //SwingUtilities.invokeAndWait(
      // () -> print(...)) für synchroner Aufruf
}
```

9 Memory Models

9.1 Java

9.1.1 Atomicity

Zugriff auf Variable (Lesen/Schreiben) ist atomar für primitive Datentypen bis 32 Bit, Objekt referenzen, **volatile long** und **volatile double**.

9.1.2 Visibility

Die Sichtbarkeit ist garantiert bei Locks Release & Acquire, **volatile**-Variablen (für while true), **final**-Variablen nach dem Ende des Konstruktors, Thread-Start/Join und Task-Start/Ende.

9.1.3 Ordering

volatile garantiert, dass kein Reordering über einen Zugriff (r/w) auf diese Variable hinaus statt findet. Das Ordering vor und nach **volatile** folgt innerhalb des Threads der *As-if-Serial* Semantik. Das heisst, der Compiler darf optimieren falls die Semantik innerhalb des Threads gleich bleibt.
Zwischen Threads ist Ordering nur bei **volatile**-Variablen und bei Synchronisationsbefehlen garantiert. **volatile**-Variablen führen nicht zu Locking.

9.2 Atomare Operationen in Java

Atomare Operationen garantieren Ordering und Visibility. Beispiele sind `getAndSet(newVal)`, `getAndAdd(delta)` und `updateAndGet(lambda)`.
Mit **boolean** `compareAndSet(old, new)` kann man atomar auf einen Wert prüfen, und falls dieser stimmt, einen neuen Wert setzen. Der Rückgabewert zeigt ob die Ersetzung stattgefunden hat.
Achtung, bei diesem Beispiel kann man in das ABA-Problem laufen. Idee: berechne etwas mit dem alten Wert, und schreibe das Resultat, falls sich das Objekt nicht verändert hat.

```
AtomicInteger bal = new AtomicInteger(0);
bal.addAndGet(amount);
//--
do{
   oldBalance = bal.get();
   newBalance = oldBalance - amount;
   if (amount > oldBalance) {
    return false;
   }
  } while (!bal.compareAndSet(oldBalance, newBalance));
//--
do {
    oldV = var.get();
    newV = calcChanges(oldV);
} while (!var.compareAndSet(oldV, newV)
```

Es gibt neben AtomicInteger (-Long, -Boolean, ...) für primitive Typen auch AtomicReference<T> für Referenzen.

9.3 ABA-Problem

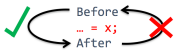
bei unserem Beispiel kann es sein, dass man erst den Wert A liest, sich dann an die Arbeit macht und schliesslich wieder den Wert A liest und denkt, es habe sich nichts geändert. In der Zwischenzeit kann aber ein anderer Thread das Objekt angefasst und B geschrieben haben, was er dann wieder mit A ersetzt. Davon merken wir nichts, wir können also mit diesem Test nicht davon ausgehen dass das Objekt nicht verändert wurde.

9.4 .NET

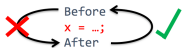
Unterschied zu Java:

- long und double auch mit **volatile** NICHT atomar
- Visibility implizit durch Ordering gegeben
- Ordering: volatile ist nur eine Partial Fence

Bei .NET sind also Umordnungen in eine Richtung erlaubt. Will man einen Full Fence, nutzt man `Thread.MemoryBarrier();`
Angenommen, wir haben ein **volatile int** x. Dann ist das folgende beim **Lesen** erlaubt:



Und das folgende beim **Schreiben**



9.5 Interlocked Atomic

```
Interlocked.Add(ref balance, 1);
Interlocked.Exchange(ref balance, 30);
// CompareExchange: if old_value, change to 20.
while(old_value != Interlocked.CompareExchange(ref
balance, 20, old_value);
Volatile.Read(ref m_flag) //oder .Write wenn variable
//nicht volatile
```

10 CUDA

10.1 Vokabular

Eine GPU besteht aus mehreren **Streaming Multiprocessors (SM)**. Jeder SM besteht aus mehreren **Streaming Processors (SP)**.

In CUDA ist eine GPU ein **Grid**, das aus mehreren **Blöcken** besteht. Jeder Block hat mehrere **Threads**. Threads werden zu je 32 als **Warps** zusammengefasst.

SIMD ist die Abkürzung für "Single Instruction Multiple Data" und entspricht dem Paradigma der Vektorparallelität. GPUs sind gut für SIMD-Applikationen geeignet, schliesslich führen alle Cores die gleiche Instruktion auf unterschiedlichen Daten aus.

10.2 CUDA

CUDA ist eine Architektur von Nvidia und arbeitet mit sogenannten **Kernels**, welche auf der GPU laufen.

Divergenz heisst, dass im selben Warp unterschiedliche Pfade vorhanden sind (z.B. if/else). Dies führt zu einem Performance-Problem.

```
__global__
void VectorAddKernel(float *A, float *B,
    float *C, int N) {
    int i = blockIdx.x * blockDim.x
        + threadIdx.x
    // bounds check
    if (i < N) {
        C[i] = A[i] + B[i];
    }
}
```

10.3 CUDA Memory Management

es gibt die drei Funktionen `cudaMalloc`, `cudaFree` und `CudaMemCpy`.

10.4 Maximale Thread-Zahl

Gegeben:

- max. Threads per Block: 1024
- max. Resident Blocks: 8
- max. Resident Threads: 1536

Wir wollen einen Vektor mit **1500** Elementen parallel bearbeiten.
Ausrechnen, dann nach folgender Priorität auswählen:

1. Alle Threads müssen in SM passen
2. am wenigsten unnütze Threads
3. am meisten Threads per Block

10.5 CUDA-Grundgerüst

Ohne Error Handling!

```
void CudaVectorAdd(float* A, float* B,
  float* C, int N) {
  size_t size = N * sizeof(float);
  float *d_A, *d_B, *d_C;

  cudaMalloc(&d_A, size); // d_B, d_C
  cudaMemcpy(d_A, A, size,
    cudaMemcpyHostToDevice); // d_B

  int blockSize = 512; //max 1024 Threads
  //Anzahl Blöcke = gridSize und aufgerundet
  int gridSize = (N + blockSize - 1) / blockSize;
  VectorAddKernel<<<gridSize,
    blockSize>>>(d_A, d_B, d_C, N);

  cudaMemcpy(C, d_C, size,
    cudaMemcpyDeviceToHost); // nur d_C
  cudaFree(d_A); // d_B, d_C
}
```

10.6 Function Keywords

- `__global__` läuft auf Device, Aufruf vom Host
- `__device__` läuft auf Device, Aufruf vom Device
- `__host__` läuft auf Host, Aufruf vom Host

10.7 Launch Configuration

muss dynamisch bestimmt werden und sich am Problem und den Fähigkeiten des Devices orientieren (ermitteln mit `cudaGetDeviceProperties()`.
Aus Effizienzgründen sollte die Blockgrösse ein Vielfaches von 32 sein. Grosse Blöcke haben den Vorteil, dass die Threads interagieren können.

10.8 Memory Access

im Device Global Memory ist massiv teurer als Zugriff im Shared Memory (`__shared__`). Um Memory **Coalescing** (mehrere Ladevorgänge zu einem zusammenfassen) zu ermöglichen, sollten Speicherzugriffe möglichst wie folgt aussehen:

```
data[(Ausdruck ohne threadIdx.x) + threadIdx.x]
```

Oft kann mit dem Vertauschen von Zeile und Spalte eine Optimierung erreicht werden.

10.9 Synchronisierung

kann mittels `__syncthreads()` erreicht werden, was alle Threads in einem Block zum Synchronisieren zwingt

10.10 Effiziente Matrix-Multiplikation

Ohne Memory Guards!

```
__shared__ float Asub[TILE_SIZE][TILE_SIZE];
__shared__ float Bsub[TILE_SIZE][TILE_SIZE];

int tx = threadIdx.x, ty = threadIdx.y;
int col = blockIdx.x * TILE_SIZE + tx;
int row = blockIdx.y * TILE_SIZE + ty;

for (int tile = 0; tile < nofTiles; tile++) {
  Asub[ty][tx] = A[row * K
    + tile* TILE_SIZE + tx];
  Bsub[ty][tx] = B[(tile * TILE_SIZE
    + ty) * M + col];
  __syncthreads();
  for (int ksub = 0; ksub < TILE_SIZE; ksub++) {
    sum += Asub[ty][ksub] * Bsub[ksub][tx];
  }
  __syncthreads();
}
C[row * M + col] = sum;
```

11 Cluster / MPI (Message Passing Interface)

Cluster weisen spezielle Eigenschaften auf. Shared Memory gibt es nur innerhalb eines Nodes. Sie sind aber mit General Purpose CPUs ausgestattet. Sie tauschen Informationen mit Dateien oder über Sockets aus.
Basiert auf dem Actor- bzw. dem CSP-Modell und ist sehr gut für heterogene Umgebungen geeignet. MPI ist ein Industristandard und ermöglicht **SPMD** (Single Program Multiple Data), da jeder Node das gleiche Programm mit anderen Daten ausführt.
Ein Task kann mittels `mpiexec -n 100 Program.exe` auf einem HPC Cluster gestartet werden.

```
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (rank == 0) {
 int value = rand(); int to;
 for (to = 1; to < size; to++) {
  MPI_Send(&value, 1, MPI_INT, to, 0,
  ↪  MPI_COMM_WORLD);
 }
} else {
 int value; int source = 0;
 MPI_Recv(&value, 1, MPI_INT, source, 0,
 ↪  MPI_COMM_WORLD,
 MPI_STATUS_IGNORE);
 printf("%i received by %i", value, rank);
}

MPI_Send(&value, LENGTH, MPI_INT, receiverRank,
↪  tag,
MPI_COMM_WORLD);
MPI_Recv(&value, LENGTH, MPI_INT, senderRank, tag,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Allreduce(&value, &total, LENGTH_per_process,
↪  MPI_INT,
MPI_SUM, MPI_COMM_WORLD) //MPI_MAX, MPI_PROD
```

- Scatter: einer verteilt die Werte an alle anderen - `MPI_Scatter(&input_array, size, MPI_INT, &output_value, recv_count, MPI_INT, rootId, MPI_COMM_WORLD)`
- Gather: alle senden verschiedenee Werte an einen - `MPI_Gather(&input_value, count, MPI_INT, &output_array, recv_count_per_process, MPI_INT, rootId, MPI_COMM_WORLD)`
- `MPI_Barrier(MPI_COMM_WORLD)` - Warte, dass alle Prozesse Barriere erreichen
- `MPI_Bcast(&input, 1, MPI_INT, rootId, MPI_COMM_WORLD)`

12 Reactive Programming

Asynchrone Datenflüsse führen dazu, dass automatisch parallelisiert werden kann, die Verteilbarkeit gut ist und keine Data-Races, Deadlocks und Starvation auftreten können.

13 Software Transactional Memory (STM)

Nimmt Ideen aus der Datenbankwelt und versucht damit das Problem des **Shared Mutable State** ohne Locks und Starvation anzugehen. Es gibt auch Hardware-Implementationen. Meistens wird **Optimistic Concurrency Control (OCC)** (Rollback bei Konflikt) als Umsetzung verwendet. STM kennt dadurch kein Locking, keine Low-Level Data-Races und keine Deadlocks.
In Java gibt es ScalaSTM, ein deskriptiver Ansatz und somit ein relativ einfaches Programmiermodell. Die Implementierung ist aber sehr komplex und "teuer".

```
final Ref.View<Integer> balance = STM.newRef(0);
final Ref.View<LocalDate> lastUpdate =
  STM.newRef(LocalDate.now());
void withdraw(int amount) {
   STM.atomic(() -> {
      if (balance.get() < amount) {
         STM.retry();
      }
      balance.set(balance.get() - amount);
      lastUpdate.set(LocalDate.now());
   });
}
```

14 Actor Modell

versucht das Problem zu lösen, dass herkömmliche Sprachen nicht für Nebenläufigkeit entworfen wurden. Kein Shared-Memory und somit keine Low-Level Data-Races.

14.1 Active Objects

sind Objekte welche ein Eigenleben führen. Im Actor-Modell gibt es nicht einen "Chef", der den Objekten befiehlt was sie tun sollen, sondern die Objekte kommunizieren untereinander.

14.2 Akka

ist eine Scala-unterstützte Implementierung des Actor-Modells. Akka Actors verfügen über eine **Mailbox** um Nachrichten zu empfangen (mittels `onReceive` im Actor).

```
public class NumberPrinter extends UntypedActor {
    public void onReceive (final Object message) {
        if (message instanceof Integer) {
            System.out.print(message);
        } else {
            getSender().tell(msg, getSelf());
        }
        // weitere Message-Arten (bas. auf Klasse)
    }
}
ActorSystem system =
    ActorSystem.create("System");
ActorRef printer =
    system.actorOf(Props.create(
        NumberPrinter.class));

for (int i = 0; i < 100; i++) {
    printer.tell(i, ActorRef.noSender());
}

system.shutdown();
```

14.3 ActorRef

speichert eine Referenz auf eine Instanz eines Actors. Dadurch wird verhindert, dass man direkt auf Variablen und Methoden des Actors zugreifen kann. Falls der Actor neu gestartet werden muss, behält er seine Adresse. Man kann die ActorRef in Nachrichten verschicken.

14.4 Remoting

wird dadurch vereinfacht dass Nachrichten immutable sind. Ein Lookup für einen Actor kann mittels `system.actorSelection(urlString)` durchgeführt werden. Das Ergebnis (eine `ActorSelection`) kann 0-n Aktoren umfassen und zu einer `ActorRef` aufgelöst werden.

14.5 Messaging

in Akka ist grundsätzlich asynchron. Es kann jedoch mittels Futures synchron auf eine Antwort gewartet werden.
Messages müssen Serializable sein und immutable. Sie dürfen nur final-Felder haben. Sie dürfen nicht über Methoden mit Seiteneffekten verfügen. Collections müssen in `Collections.unmodifiableList` verpackt werden.

```
Future<Object> result = // immer Object
    Patterns.ask(actorRef, msg, timeout);
result.get();

public class Booking {
    final String name;
    public Booking(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
```

14.6 Akka Laufzeitsystem

setzt typischerweise auf `ForkJoinPools` auf, es wird aber aus Effizienzgründen nicht ein Thread pro Actor verwendet.
Synchrones Senden und Empfangen von Nachrichten führt zu Warteabhängigkeiten, was wiederum zu Deadlocks führen kann. Deshalb wird von synchroner Kommunikation abgeraten.

14.7 Akka Supervision

bezeichnet das Überwachen von Actors durch andere Actors. Eltern überwachen per Default ihre Kinder. Bei einer Exception wird der Supervisor benachrichtigt und muss entscheiden wie es weiter geht. Resume: Kind soll weitermachen. Restart: Kind neustarten. Stop: Kind beenden. Escalate: seinem Supervisor melden, dass er selber nicht weiss wie reagieren

14.8 System Shutdown

erfolgt durch das Stoppen der Actors: (alle rekursiv)
Mittels `getContext().stop(actorRef)` wird einem Actor mitgeteilt, dass er *nach* Bearbeitung der aktuellen Message anhalten soll.
Mit `actor.tell(PoisonPill.getInstance(), sender)` wird eine Terminierungsnachricht eingereiht, die den Actor stoppt.
Als *last measure* gibt es noch `actor.tell(Kill.getInstance(), sender)` was eine Supervision-Behandlung auflöst.

15 Beispiele

15.1 CountDownLatch mit Semaphore

```
class CountDownLatch {
    private int count;
    private Semaphore gate = new Semaphore(0);

    public CountDownLatch (int count) {
        this.count = count;
    }
    public void countDown() {
        gate.release();
    }
    public void await() {
        gate.acquire(count);
        gate.release(count);
    }
}
```

15.2 Fukushima-Mailbox

```
public class Mailbox {
  private AtomicReference<Object> item =
  new AtomicReference<>();
  public void put(Object value) {
    while (!item.compareAndSet(null, value)) {
      Thread.yield();
    }
  }
  public Object get() {
    while (true) {
      Object value = item.get();
      if (value != null &&
      item.compareAndSet(value, null)) {
        return value;
      }
      Thread.yield();
    }
  }
}
```

15.3 Multi-Acquire Semaphore

```
public class Semaphore {
  private int counter = 0, waiting = 0;
  public synchronized void acquire(int amount)
  throws InterruptedException {
    waiting++;
    try {
      while (counter < amount) { wait(); }
      counter -= amount;
    } finally {
      waiting--;
    }
  }

  public synchronized void release(int amount) {
    counter += amount;
    notifyAll();
  }
  public synchronized int nofWaitingThreads() {
    return waiting;
  }
}
```

15.4 Cyclic Barrier

```
public class CyclicBarrier {
  private final int parties;
  private int count = 0;
  private Semaphore open, closed, mutex;

  public CyclicBarrier(int parties) {
    open = new Semaphore(parties);
    closed = new Semaphore(0);
    mutex = new Semaphore(1);
  }

  public void await() {
    open.acquire();
    mutex.acquire();
    count++;
    if (count == parties) closed.release(parties);
    mutex.release();
    closed.acquire();
    mutex.acquire();
    count--;
    if (count == 0) open.release(parties);
    mutex.release();
  }

  public synchronized void await()
  throws InterruptedException {
    entered++;
    if (entered == parties) {
      exited = 0;
      notifyAll();
    }
    while (entered < parties) { wait(); }
    exited++;
    if (exited == parties) {
      entered = 0;
      notifyAll();
    }
    while (exited < parties) { wait(); }
  } }
```

15.5 Read Write Lock

```
public class ReadWriteLock {
  private int readers;
  private Semaphore wlock, mutex;^^I
  public ReadWriteLock() {
    wlock = new Semaphore(1);
    mutex = new Semaphore(1);
  }

  public void writelock() { wlock.aquire(); }
  public void writeunlock() { wlock.release(); }
  public void readlock() {
    mutex.aquire();^^I^^I
    if(readers == 0) { wlock.aquire(); }
    ++readers; mutex.release();
  }
  public void readunlock() {
    mutex.aquire();
    if(readers == 1) { wlock.release(); }
    --readers; mutex.release();
  } }
```

15.6 Read-Write Locks

```
public class UpgradeableReadWriteLock {
    private int readers;
    private Thread upgradable;
    private boolean writer;

    public synchronized void readLock()
    throws InterruptedException {
        while (writer) {
            wait();
        }
        ++readers;
    }
    public synchronized void readUnlock() {
        --readers;
        notifyAll();
    }
    public synchronized void upgradeableReadLock()
    throws InterruptedException {
        while (upgradable != null || writer) {
            wait();
        }
        upgradable = Thread.currentThread();
    }
    public synchronized void upgradeableReadUnlock() {
        upgradable = null;
        notifyAll();
    }
    public synchronized void writeLock()
    throws InterruptedException {
        while (readers > 0 || writer ||
        (upgradable != null &&
        upgradable != Thread.currentThread())) {
            wait();
        }
        writer = true;
    }
    public synchronized void writeUnlock() {
        writer = false;
        notifyAll();
    } }
```

15.7 Semaphore

```
public class Semaphore {
    private int count;
    private ReentrantLock l;
    private Condition full;

    public Semaphore(int count, boolean fair) {
        this.count = count;
        l = new ReentrantLock(fair);
        full = l.newCondition();
    }
    public void acquire(int amount)
    throws InterruptedException {
        l.lock();
        while (count < amount) { full.await(); }
        count -= amount; l.unlock();
    }
    public void release(int amount) {
        l.lock();
        count += amount; full.signalAll();
        l.unlock(); }}
```

15.8 Exchanger

```
public class Exchanger<T> {
  private int entered = 0, exited = 2;
  private Object[] items = new Object[2];
  public synchronized T exchange(T item)
      throws InterruptedException {
    while (exited < 2) {
      wait();
    }
    int other = (entered + 1) % 2;
    items[entered] = item;
    entered++;
    if (entered == 2) {
      exited = 0;
      notifyAll();
    }
    while (entered < 2) {
      wait();
    }
    T result = (T)items[other];
    items[other] = null;
    exited++;
    if (exited == 2) {
      entered = 0;
      notifyAll();
    }
      return result;
    }
}
```