3 – Multithreading

Ein Prozess ist die Abstraktion eines in Ausführung befindlichen Programms

- Ein Prozess kann verschiedene Zustände haben
  - > Rechnend
  - > Wartend
  - > Rechenbereit
- Jeder Prozess hat einen eigenen Adressraum
  - > Keine gemeinsamen Variablen zwischen Prozessen

Ein Thread ist ein *leichtgewichtiger* Prozess

- Ein Thread gehört zu einem Prozess
- Alle Threads eines Prozesses teilen sich denselben Adressraum
  - > Gemeinsame Variablen zwischen Threads möglich

Multithreading
Warum Threads?

Es gibt mehrere Gründe um Threads zu verwenden:

- Effiziente Nutzung von Multiprozessor-Architekturen
- Zeitersparnis
- Ressourcenersparnis (kein „aktives Warten“)

Ein Thread muss wissen in welcher Codezeile er gestartet werden soll

- **Idee**
  - > Der neue Thread führt eine Methode aus
  - > Der Thread wir zerstört, wenn die Methode abgearbeitet wurde

- **Problem**
  - > Ein Funktionszeiger wäre nützlich
  - > Es gibt keine Funktionszeiger in Java

- **Lösung**
  - > Aufruf der nativen Thread-Klasse mit einem Objekt, welches das *Runnable*-Interface implementiert

Multithreading
Das Runnable-Interface

```
public class MyRunnable implements Runnable
{
   public void run()
   {
     // do something useful
   }
}

[...]

Thread t = new Thread(new MyRunnable());
t.start();

[...]
```

Ein Thread beendet sich automatisch, wenn die ausgeführte Methode zu Ende ist

- **Frage**
  > Gibt es eine Möglichkeit auf das Ende eines Threads zu warten?

- **Antwort**
  > Ja.

- **Beispiel**

```
Thread t = new Thread (new MyRunnable());
t.start();
// do something useful
[...]
t.join(); // wait for thread
```

Multithreading
Race Condition

Eine Race Condition tritt auf, wenn zwei oder mehr Threads auf dieselbe Speicherzelle zugreifen und mindestenst ein Thread in die Speicherzelle schreibt

Beispiel

- Angenommen es gibt eine Variable x mit Wert 4
  - Thread 1: `x = 5;`
  - Thread 2: `System.out.println(x);`

- Abhängig von der Ausführungsreihenfolge der Threads wird entweder 4 oder 5 ausgegeben

- Race Conditions führen zu nicht-deterministischem Verhalten des Programms

Ein Deadlock tritt auf, wenn zwei oder mehr Threads jeweils aufeinander warten

Um Race Conditions zu vermeiden ist es gegebenenfalls nötig Threads zu synchronisieren

- Synchronisation bedeutet aktive Beeinflussung der Abarbeitungsreihenfolge der Threads

- Es gibt verschiedene Methoden Threads zu synchronisieren
  - > Atomare Datentypen / Atomare Operationen
  - > Mutex Locks
  - > Semaphoren
  - > Barrieren
  - > …

Eine Atomare Operation ist eine ununterbrechbare Operation

- Kein anderer Thread kann eine Operation ausführen, solange die Atomare Operation ausgeführt wird

Eine Atomarer Datentyp ist ein Datentyp, dessen Operationen atomar sind

- z.B. „AtomicInteger“ in Java

- **Beispiel**

```
AtomicInteger atomic = new AtomicInteger(5);
int nonAtomic = atomic.addAndGet(10);
// nonAtomic is now 15
```

Multithreading
Mutex Lock (1)

Ein Mutex Lock sorgt dafür, dass ein bestimmter Teil des Quellcodes („critical region") nur von einem Thread gleichzeitig ausgeführt wird

- **Beispiel**

```
ReentrantLock mutex = new ReentrantLock();
mutex.lock();
// do something useful }
mutex.unlock();
```

- Der Teil zwischen `lock()` und `unlock()` wird nur von einem Thread zur selben Zeit ausgeführt

Ein Mutex kann auch mit einem `synchronized`-Block realisiert werden

- Für einen `synchronized`-Block wird ein Objekt als Mutex benötigt
- Auch das `this`-Objekt kann als Mutex verwendet werden

- **Beispiel**

```
SomeObject mutex = new SomeObject();
synchronized( mutex );
{
     // do something useful }
}
```

```
public synchronized void func()
{
    // do something useful }
}
```

ist dasselbe wie

```
public void func()
{
    synchronized(this)
    {
        // do something useful
    }
}
```

Eine Semaphore funktioniert ähnlich wie ein Mutex Lock, aber erlaubt bis zu *n* Threads die gleichzeitige Ausführung der kritischen Region

- **Beispiel**

```
int n = 4;
Semaphore s = new Semaphore(n);
s.acquire();
// do something useful
s.release();
```

- Erlaubt bis zu 4 Threads gleichzeitig die kritische Region auszuführen

Eine Pipe ist ein uni- oder bidirektionaler Datenstrom, der nach dem FIFO-Prinzip funktioniert.

- Auch Queue genannt

- **Beispiel**

```
LinkedBlockingQueue < Integer > queue =
      new LinkedBlockingQueue < Integer >();

// Thread a
int t = queue.take (); // blocks if queue is empty

// Thread b
int p = 5;
queue.put(p)
```

Multithreading
Barriere (1)

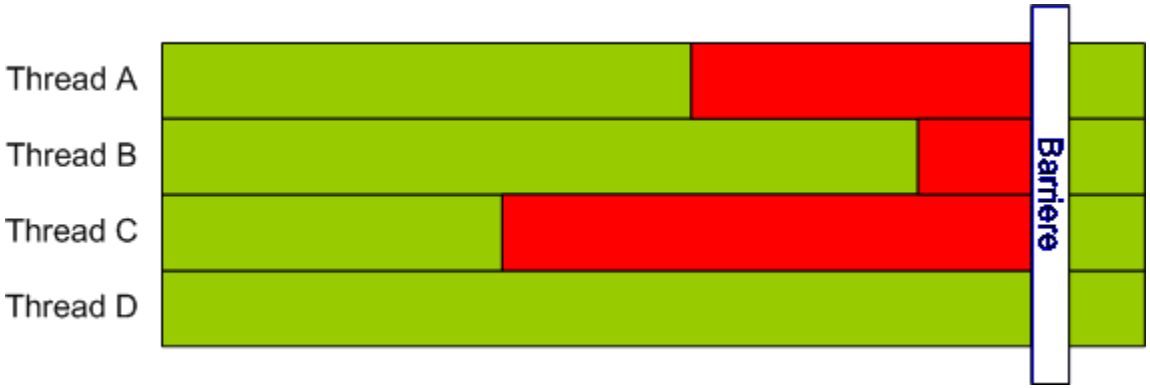
Eine Barriere blockiert alle ankommenden Threads, bis eine bestimmte Anzahl von Threads wartet

- Die Anzahl der maximal wartenden Threads ist einstellbar
- Wenn der letzte Thread die Barriere erreicht, werden alle Threads freigegeben

- **Beispiel**

```
int n = 4;
CyclicBarrier barrier = new CyclicBarrier(n);
try
{
barrier.await();
}
catch( Exception e) { /* do something /* }
```

Multithreading
Barriere (2)



Ein Threadpool ist eine Gruppe von Threads

- Jeder Thread im Threadpool schläft, bis er eine Aufgabe zugeteilt bekommt
- Nach Beendigung der Aufgabe kehrt der Thread in den Threadpool zurück
- Sind keine Threads für eine Aufgabe verfügbar, wird die Aufgabe in eine Queue aufgenommen

- **Beispiel**

```
ExecutorService pool =
     Executors.newFixedThreadPool (5);

Runnable task = new TaskImplementation();
pool.execute( task );
```

Eine Future ist ein Objekt, welches als Platzhalter für noch nicht verfügbare Daten fungiert.

- **Beispiel**

```
ExecutorService pool =
     Executors.newFixedThreadPool(5);

Callable <String > task = new TaskImplementation();
Future <String > f = pool.submit( task );
String result = f.get (); // blocks if necessary
```