# 08-Threads-03

Objektorientierte Programmierung | Matthias Tichy

Software Engineering
Programming Languages

universität ulm

# Lernziele

- Executor Framework
- Futures
- Demos: PI berechnen in verschiedenen nebenläufigen Varianten

# Executor Framework

# Verteilung von Arbeiten auf Threads

- Manuelle Verteilung von Arbeiten auf Threads ist schwierig
  - Performance
  - Synchronisation

- Executor Framework:
  - Definition von Aufgaben durch den Entwickler
  - Parallele Ausführung durch ein Framework

# Executor

> Ein Executor führt das übergebene Runnable an irgendeinem Zeitpunkt in der Zukunft aus.

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

- Ausführung:
  - Im Aufrufthread
  - In einem neuen Thread
  - durch einen Threadpool

https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/concurrent/Executor.html

# ThreadPoolExecutorService

> Erzeugt einen Pool von Threads und führt jedes Runnable auf einem der Threads aus.

```
ExecutorService executorService =
Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());
executorService.execute(() -> {
// ...
}
executorService.shutdown()
```

- Vorteile:
  - Keine manuelle Verteilung notwendig
  - Konfigurationsmöglichkeiten

https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/concurrent/ThreadPoolExecutor.html

# ExecutorService

- Ausführungsmöglichkeiten

```
public interface ExecutorService extends Executor {
  <T> Future<T> submit(Callable<T> task);
  Future<?> submit(Runnable task);

  <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
        throws InterruptedException;
  <T> T invokeAny(Collection<? extends Callable<T>> tasks)
        throws InterruptedException, ExecutionException;

  void shutdown();
...
```

https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/concurrent/ExecutorService.html

# Factory: Executors

Auswahl von factory-Methoden

- Ausführung mit einer vorgegebenen Anzahl an Threads

```
ExecutorService newFixedThreadPool(int nThreads)
```

- Eine Queue, #Threads dynamisch

```
ExecutorService newCachedThreadPool()
```

- Ausführung fest in einem einzelnen Thread

```
ExecutorService newSingleThreadExecutor()
```

- Mehrere Queues, #Threads dynamisch

```
ExecutorService newWorkStealingPool()
```

# Konfigurationsmöglichkeiten

Beispielhafte Auswahl für ThreadPoolExecutor

- `ThreadFactory`, um z.B. die Prioritäten der Threads zu beeinflussen
- `keepAliveTime`, wie lange dürfen Threads idle bleiben bevor sie beendet werden
- Queuing Strategien:
  - Direct Handoff: Wenn kein freier Thread existiert, wird ein neuer erzeugt.
  - Unbounded Queues: Wenn kein freier Thread existiert, wird der Task in einer Queue gespeichert
  - Bounded Queues: Queue voll → Task abgelehnt

# ScheduledExecutorService

Auswahl von Methoden

> Ausführung von Tasks nach einem Delay und/oder wiederholt mit einer Rate.

```
public interface ScheduledExecutorService extends ExecutorService {
    public ScheduledFuture<?> schedule(Runnable command,
                                       long delay, TimeUnit unit);

    public <V> ScheduledFuture<V> schedule(Callable<V> callable,
                                           long delay, TimeUnit unit);
    public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                                                  long initialDelay,
                                                  long period,
                                                  TimeUnit unit);
...
```

# Futures

# Future<V>

Auswahl von Methoden

> Ein Future repräsentiert das zukünftige Ergebnis einer asynchronen Operation.

- Rückgabe eines Futures vermeidet Blockierung
- Prüfung auf den Zustand: fertig/nicht fertig, abgebrochen / nicht abgebrochen
- Blockierend (evtl. mit Timeout) das Ergebnis (Typ V) holen

- Siehe JavaScript: Promise

# Future

Auswahl von Methoden

> Ein Future repräsentiert das zukünftige Ergebnis einer asynchronen Operation.

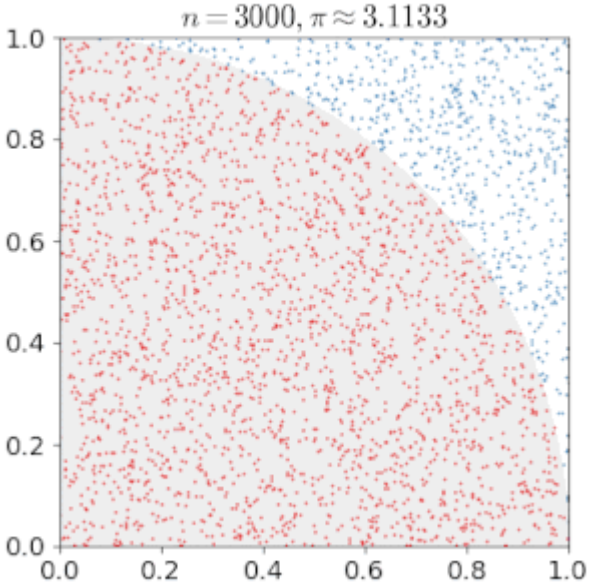
```
public interface Future<V> {
  boolean cancel(boolean mayInterruptIfRunning);
  boolean isCancelled();
  boolean isDone();
  V get() throws InterruptedException, ExecutionException;
  V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

# Concurrent PI Calculations

# Annäherung an π

Montel-Carlo-Experiment

- Berechne n-mal: Zufallspunkte (x, y), zwischen 0 und 1
- Das Verhältnis zwischen den Zufallspunkten innerhalb des Kreises zu allen n Zufallspunkten ist annäherungsweise ein Viertel von π
- Vergleich unterschiedlicher Varianten hinsichtlich Skalierung durch Nebenläufigkeit




Pi 30K.gif by Nicoguaro, CC BY 3.0

# Varianten (1/2)

- `SingleThread`
- `IteratorPI`: Stream: Map / Reduce
- `PIteratorPI`: ParallelStream: Map / Reduce
- `SimpleThreads`
- `FixedThreadPool`

# Varianten (2/2)

- `BigChunks`
- `BigChunksFutures`
- `BigChunksFuturesLocalRNG`
- `BigChunksFuturesLocalRNGSplits`
- `BigChunksFuturesLocalQueue`

# Lernziele

- Executor Framework
- Futures
- Demos: PI berechnen in verschiedenen nebenläufigen Varianten