

Algorithmen & Datenstrukturen

Threadpools

Threads wiederverwenden

Roger Diehl



Inhalt

- Ausführen von nebenläufigen Aufgaben
- Ein einfacher Worker Pool Executor
- Java Thread Pool Executors
- Executor mit eigener Thread Factory
- Tipps für das Arbeiten mit Threadpools
- Zusammenfassung

Lernziele

- Sie wissen bei welcher Art von Threads, Sie einen Thread wieder verwenden oder einen Thread neu starten.
- Sie kennen die Nachteile, wenn man einen Thread einzeln startet.
- Sie können das nebenläufige Ausführen einer Aufgabe von der technischen Realisierung trennt.
- Sie kennen das Thread Pool Konzept.
- Sie können die **Executors**-Klasse zur Erzeugung von Thread Pools adäquat einsetzen.
- Sie können die Grösse eines Thread Pools für eine entsprechende Aufgabe abschätzen und konfigurieren.

Ausführen von nebenläufigen Aufgaben

Anmerkungen zur Ausführung von Aufgaben und Threads

- Zahlreiche Aufgaben, die nebenläufig ausgeführt werden sollen, sind oft nur von kurzer Dauer und treten nicht unbedingt regelmässig auf.
- Für jede neue Aufgabe einen Thread erzeugen, belastet unnötig das Betriebssystem.
- Eine grosse Anzahl von Threads wirkt sich negativ auf die Systemleistung aus (zur Erinnerung: $\text{Speedup} < 1$).
 - Es macht Sinn die Menge der erzeugten Threads zu beschränken
- In der Praxis wird man deshalb weniger mit rudimentären Thread-Objekten arbeiten, sondern mit sogenannten Threadpools.
- siehe OOP Input → **O15_IP_Nebenläufigkeit; Abschnitt «Executor API»**

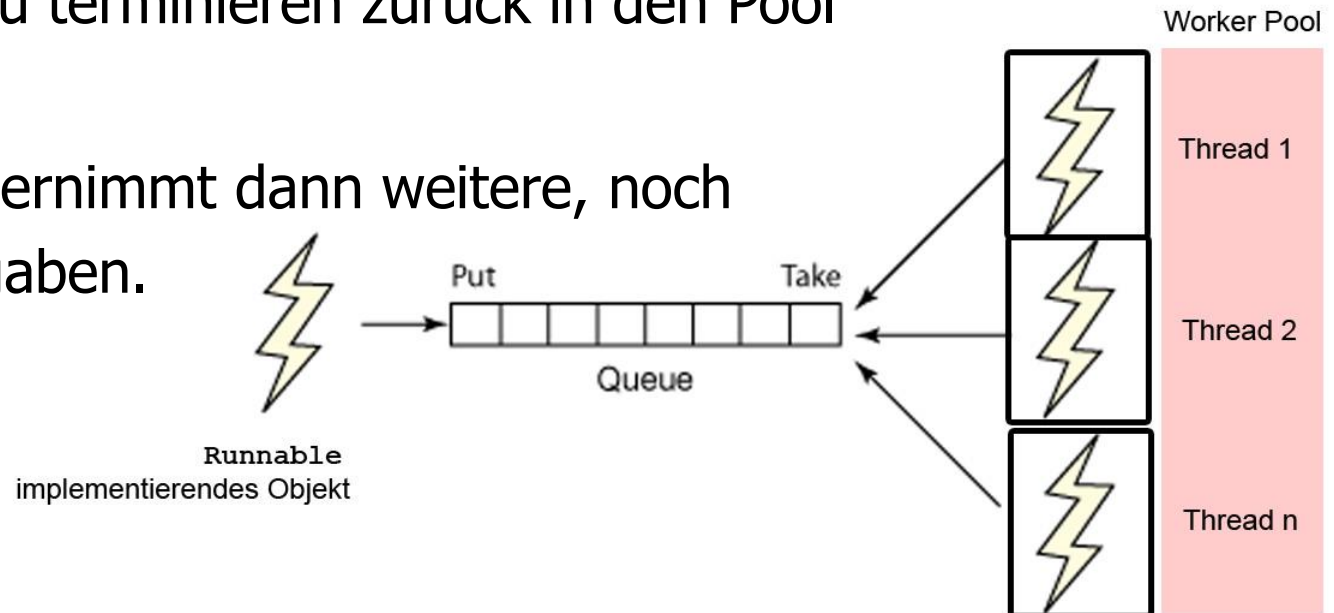
Noch ein paar Anmerkungen zu Threads

- Zur nebenläufigen Ausführung einer Aufgabe (Objekt vom Typ **Runnable**) ist immer ein Thread notwendig.
- Bei Erzeugen eines Thread muss ein Objekt vom Typ **Runnable** dem Thread-Konstruktor übergeben werden.
 - Es gibt keine Getter- oder Setter-Methoden.
- Der Thread beginnt mit der Abarbeitung der Aufgabe vom Typ **Runnable** sofort nach dem Aufruf von **start**.
- Der Aufruf von **start** auf dem Thread-Objekt ist nicht zweimal möglich – **IllegalThreadStateException**.
- **Wunsch:** Eine Abstraktion, die das nebenläufigen Ausführen der Aufgabe von der technischen Realisierung trennt.

Ein einfacher Worker Pool Executor

Das Thread Pool Konzept

- Ein Threadpool verwaltet eine Anzahl von Threads.
- Soll eine Aufgabe nebenläufig durchgeführt werden, so übergibt man dem Pool ein entsprechendes **Runnable**-Objekt.
- Je nach Art des Pools wird es sofort einem Thread zugeteilt oder erst in eine Queue gestellt und später bearbeitet.
- Wenn der Thread das Runnable ausgeführt hat, wird er ohne zu terminieren zurück in den Pool gestellt.
- Der Thread übernimmt dann weitere, noch wartende Aufgaben.



Executor – eine Implementierung

- Der Executor stellt ein Interface dar und enthält die Methode **execute** mit einem Parameter vom Typ **Runnable**.
 - Die Schnittstelle ist damit ähnlich der eines Threads.
 - Steht im Package **java.util.concurrent** zur Verfügung.
- Mit dem Executor ist die Entkopplung möglich, zwischen nebenläufig ausführbaren Aufgaben und der technischen Realisierung der nebenläufigen Ausführung.

```
public interface Executor {  
    /**  
     * Führt die gegebene Aufgabe irgendwann in der Zukunft aus.  
     * @param command ausführbare Aufgabe.  
     */  
    void execute(Runnable command);  
}
```

Worker Pool Executor – Erzeugung

- Die Klasse `PlainWorkerPool` implementiert einen `Executor`.
- Sie erzeugt eine festgelegte Anzahl Threads, welche Aufgaben aus einer Queue holen und ausführen.

```
public final class PlainWorkerPool implements Executor {
```

```
    private final BoundedBuffer<Runnable> workQueue;
```

```
    private final int nWorkers;
```

Grösse der Queue

Anzahl Worker Threads

```
    public PlainWorkerPool(final int capacity, final int nWorkers) {  
        this.workQueue = new BoundedBuffer<>(capacity);  
        this.nWorkers = nWorkers;  
        for (int i = 0; i < this.nWorkers; ++i) {  
            activate();  
        }  
    }  
}
```

Codeskizze

Worker Pool Executor – Worker Thread

- Methode **activate** erzeugt und startet einen Worker Thread.

```
private void activate() {  
    final Runnable runLoop = () -> {  
        try {  
            while (true) {  
                final Runnable r = workQueue.take();  
                r.run();  
            }  
        } catch (InterruptedException e) {  
            LOG.debug(e);  
        }  
    };  
    final Thread thread = new Thread(runLoop);  
    thread.setDaemon(true);  
    thread.start();  
}
```

Codeskizze

Aufgabe aus der Work Queue holen.

Aufgabe ausführen.

Thread stirbt, wenn der main-Thread beendet wird.

Worker Pool Executor – Aufgabe entgegen nehmen

- Die Methode `execute` nimmt die Aufgaben (vom Typ `Runnable`) als Parameter entgegen und speichert diese in die Work Queue (hier vom Typ `BoundedBuffer`) ab.
 - Da in diesem Beispiel die Work Queue blockierend ist, ist auch Methode `execute` blockierend, wenn die Kapazität der Queue voll ist.

Codeskizze

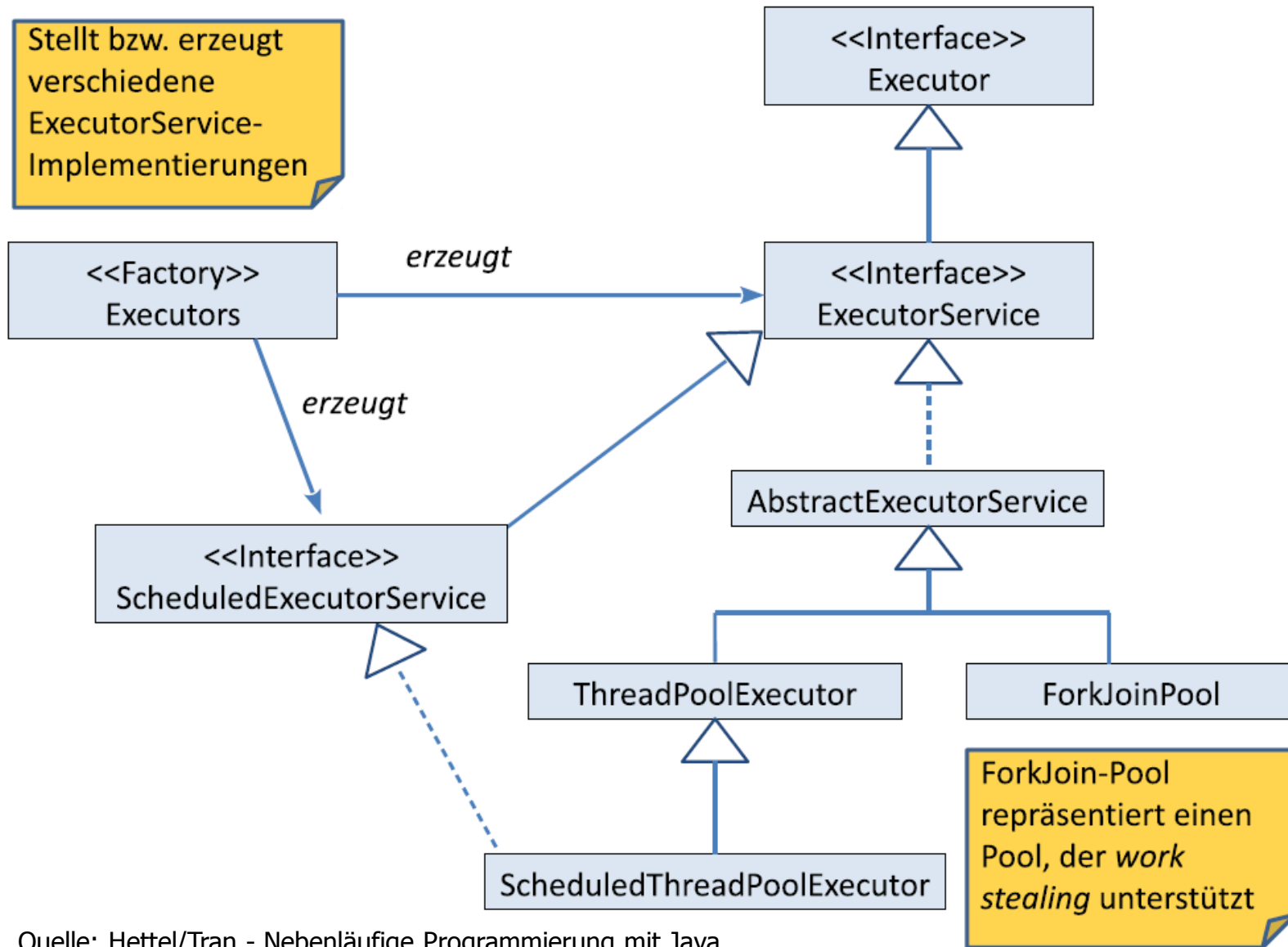
```
@Override
public void execute(Runnable command) {
    try {
        workQueue.put(task);
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}
```

Java Thread Pool Executors

Executors-Klasse

- Wie gezeigt, kann man eigene Threadpools implementieren.
- In der Praxis sollten aber die von Java angebotenen Möglichkeiten verwendet werden.
- Die **Executors**-Klasse bietet Fabrikmethoden an, zur Erzeugung von Threadpools folgender Typen:
 - **ThreadPoolExecutor**
 - **ScheduledThreadPoolExecutor**
 - **ForkJoinPool**
- Die Fabrikmethoden liefern alle eine **ExecutorService**- bzw. **ScheduledExecutor**-Service-Implementierung zurück.
 - siehe OOP Input → **O15_IP_Nebenläufigkeit; Folie 24**

Klassenhierarchie des Poolkonzepts



Quelle: Hettel/Tran - Nebenläufige Programmierung mit Java

Fabrikmethoden der Executors-Klasse (Auswahl)

▪ **newCachedThreadPool()**

- Diese Fabrikmethode liefert einen Pool wo bei Bedarf neue Worker Threads erzeugt werden.
- Unbenutzte Threads bleiben für 60 Sekunden erhalten.

▪ **newFixedThreadPool(int nThreads)**

- Diese Fabrikmethode liefert einen Pool mit **nThreads** Threads.
- Die Warteschlange für übergebene Aufgaben ist unbeschränkt.
- Stirbt ein Thread aufgrund eines Fehlers, wird er durch einen neuen ersetzt.

▪ **newScheduledThreadPool(int coreSize)**

- Diese Fabrikmethode liefert einen **ScheduledExecutorService**.
- Damit werden Aufgaben nach einer gegebenen Verzögerung bzw. periodisch ausgeführt.

Beispiel: Pool mit fixer Anzahl Worker Threads

- Die **Aufgabe** gibt jeweils **200** Buchstaben auf das Terminal aus.

```
public static void main(final String[] args) throws InterruptedException {
    final ExecutorService executor = Executors.newFixedThreadPool(2);
    for (int nTask = 1; nTask <= 4; nTask++) {
        final char ch = (char) (64 + nTask);
        final char chStop = 'X';
        executor.execute(() -> {
            LOG.info(Thread.currentThread().getName() + " starts "+ch);
            for (int i = 0; i < 200; i++) {
                System.out.print(ch);
                if (ch == chStop && i == 100) {
                    Thread.currentThread().stop();
                }
            }
            System.out.println("");
            LOG.info(Thread.currentThread().getName() + " finished "+ch);
        });
    }
    TimeUnit.MILLISECONDS.sleep(100);
    LOG.info("shutdown");
    executor.shutdown();
}
```

2 Worker Threads

Aufgabe übergeben

NUR zu Demozweck – Thread stirbt

Thread Pool beenden

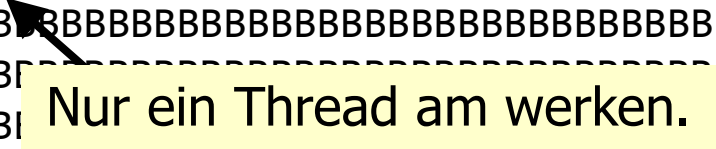
Codeskizze

Experiment: Pool mit einem Thread

■ Sequentielle Abarbeitung

- Nach dem **shutdown** werden die zugewiesenen Aufgaben noch abgearbeitet, neue werden aber mit einer **RejectedExecutionException** abgewiesen.
- Zu beachten ist, dass **shutdown** kein blockierender Aufruf ist.

```
2017-03-23 09:45:42,902 INFO - pool-2-thread-1 starts A
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
2017-03-23 09:45:42,905 INFO - pool-2-thread-1 finished A
2017-03-23 09:45:42,905 INFO - pool-2-thread-1 starts B
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
2017-03-23 09:45:42,906 INFO - pool-2-thread-1 finished B
2017-03-23 09:45:42,907 INFO - pool-2-thread-1 starts C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
2017-03-23 09:45:42,908 INFO - pool-2-thread-1 finished C
2017-03-23 09:45:42,999 INFO - shutdown
```



Experiment: Pool mit zwei Threads

- Nebenläufige Abarbeitung der Aufgaben.
 - Zu beachten ist, dass echt-parallel so viele Threads laufen wie Threading-Pipes zur Verfügung stehen.

```
2017-03-23 09:53:20,732 INFO - pool-2-thread-1 starts A
2017-03-23 09:53:20,732 INFO - pool-2-thread-2 starts B
AAAAAAAAABBBAAAAAAAAAAAAAAAAABAAAAAAAAABAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA/
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAA/
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAA
AABBBBBBABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
2017-03-23 09:53:20,737 INFO - pool-2-thread-1 finished A
2017-03-23 09:53:20,737 INFO - pool-2-thread-1 starts C
BBBBBBBCCCCCCCCCCCCCCCCBBBBBBCCBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
2017-03-23 09:53:20,738 INFO - pool-2-thread-2 finished B
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
2017-03-23 09:53:20,739 INFO - pool-2-thread-1 finished C
2017-03-23 09:53:20,830 INFO - shutdown
```

Zwei Threads am werken.

Experiment: Ein Thread aus den Pool stirbt

- Erzeugung eines Pools mit zwei Threads.
- Nebenläufige Abarbeitung der Aufgaben.
- Während der Abarbeitung der Aufgabe stirbt der Thread.
- Ein neuer Thread wird erzeugt.

```
2017-03-23 09:57:57,917 INFO - pool-2-thread-1 starts A
2017-03-23 09:57:57,917 INFO - pool-2-thread-2 starts B
BAAAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBDDDDDDAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA Thread-2 stirbt. BABBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
2017-03-23 09:57:57,921 INFO - pool-2-thread-3 starts C
CCCCCAAACCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCAAAAAAAAAAAAAAAAAAAACCACCCCCCCCCC
2017-03-23 09:57:57,922 INFO - pool-2-thread-1 finished A
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
Neuer Thread wurde erzeugt. CCCCCCCCCCCCCC
2017-03-23 09:57:57,923 INFO - pool-2-thread-3 finished C
2017-03-23 09:57:58,015 INFO - shutdown
```

Executor mit eigener Thread Factory

Thread für den Pool selbst konfigurieren

- Bei den meisten Fabrikmethoden kann eine **ThreadFactory** als Argument übergeben werden.
 - `newCachedThreadPool(ThreadFactory threadFactory)`
 - `newFixedThreadPool(int nThreads, ThreadFactory threadFactory)`
 - `newScheduledThreadPool(int coreSize, ThreadFactory threadFactory)`
- Die **ThreadFactory** ist ein Interface, das einen neuen, selbst definierten Thread erzeugt.
 - `Thread newThread(Runnable r)`
 - Einfachste Implementierung:

```
public final class SimpleThreadFactory implements ThreadFactory {  
    @Override  
    public Thread newThread(final Runnable r) {  
        return new Thread(r);  
    }  
}
```

Quelle: Java API Dokumentation

Beispiel: Pool mit Daemon Threads

- Ein solcher Pool wird automatisch heruntergefahren, sobald der letzte Thread fertig ist.

```
public static void main(final String[] args) {  
    final ExecutorService executor;  
    executor = Executors.newFixedThreadPool(3, new ThreadFactory() {  
        @Override  
        public Thread newThread(Runnable r) {  
            final Thread thread = new Thread(r);  
            thread.setDaemon(true);  
            return thread;  
        }  
    });  
    for (int nTask = 1; nTask <= 5; nTask++) {  
        executor.execute(() -> {  
            // hier wird gearbeitet...  
        });  
    }  
}
```

Codeskizze

Alle Threads haben nun die Daemon-Eigenschaft

Daemon-Threads werden gestoppt, sobald keine Nicht-Daemon-Threads* mehr laufen.

Tipps für das Arbeiten mit Threadpools

Grundsätzliche Empfehlungen aus O15_IP_Nebenläufigkeit

- Nebenläufigkeit kann nicht nur die Performance steigern, sondern auch das Programmiermodell vereinfachen!
- Einfache, wenige, langlaufende nebenläufige Threads: Können gut mit **Thread** und **Runnable** realisiert werden.
- Sobald man mit vielen, eher kurzen und häufigen Threads rechnen muss: Unbedingt Konzept der **Executor**-Services nutzen, da einfacher, viel effizienter und auch mit Rückgabewerten.
- Voneinander unabhängige Aufgaben (fire&forget-Prinzip) sind in der Regel völlig problemlos.
- Vorsicht bei Abhängigkeiten, z.B. bei Zugriff auf **gemeinsame** Daten(-strukturen): → **Synchronisation** notwendig, ansonsten drohen Inkonsistenzen und Fehler!

Anzahl der Pool-Threads

- Eine angemessene Poolgrösse hängt einerseits von der Anzahl und andererseits von der Art der zu bearbeitenden Aufgaben ab.
- Die Frage ist, ob sie eher IO-intensiv oder rechenintensiv sind.
- Goetz et al.* geben folgende Faustregel für die Anzahl der Pool-Threads an:

$$N_{threads} = N_{cpu} \cdot U_{cpu} \cdot \left(1 + \frac{W}{C}\right)$$

- mit

N_{cpu} = Anzahl der zur Verfügung stehenden Kerne

U_{cpu} = Auslastung der CPU, $0 < U_{cpu} < 1$

W/C = Verhältnis zwischen Warte- und Rechenzeit

- Für rechenintensive Tasks und Volllastung sollte folgendes gewählt werden

$$N_{threads} = N_{cpu} + 1$$

- **`Runtime.getRuntime().availableProcessors() + 1`**

* Brain Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes und Doug Lea.
Java Concurrency in Practice (Addison-Wesley, 2006)

Zusammenfassung

- In Java werden viele praktische Konzepte für den Umgang mit Threads angeboten, die auch in der Praxis angewendet werden sollten.
- Durch die Klassenmethoden von **Executors** können bequem verschiedene Threadpools erzeugt werden, die alle das **ExecutorService**-Interface implementieren.
- Die Pools können sowohl Tasks vom Typ **Runnable** (ohne explizite Wertrückgabe durch Aufruf der **execute**-Methode) als auch vom Typ **Callable** mit Wertrückgabe durch die **submit**-Methode (siehe O15_IP_Nebenläufigkeit)* ausführen.
- Es gibt ausserdem die Möglichkeit mit Hilfe der **ThreadFactory** zur Konfiguration von Threads eines Pools.

*wird noch genauer behandelt

Fragen?