

# Task Execution

Vortrag von

*Sirithana Tiranardvanich*

*Rouven Strauß*

über Kapitel 6 des Buches  
„Java – concurrency in practice“

# Ausblick



Welche **Möglichkeiten** gibt es, um **Tasks auszuführen**?

Veranschaulichung: **Webserver**



Wie finde und nutze ich die **Parallelisierbarkeit von Tasks** effizient?

Veranschaulichung: **Webbrowser**



Wie **verzögere** ich die Ausführung von Tasks  
bzw. wie **beschränke** ich **Tasks zeitlich**?

Veranschaulichung: **Webseite**



# Möglichkeiten der Task-Ausführung

---

## Tasks

### **Definition** *Task*:

abstrakte, diskrete Arbeitseinheit

- oft **unabhängig** von anderen Tasks
- konkrete Implementierung: **Runnable**

### **Vorteile** von Tasks:

- **Vereinfachung** der Programmorganisation
- Erleichterung der Fehlerbehebung
- Förderung der **Nebenläufigkeit**



# Möglichkeiten der Task-Ausführung

## Problemstellung

vorgegebene, voneinander unabhängige Tasks

möglichst schnelle Ausführung

Effizienz möglichst unabhängig von Anzahl

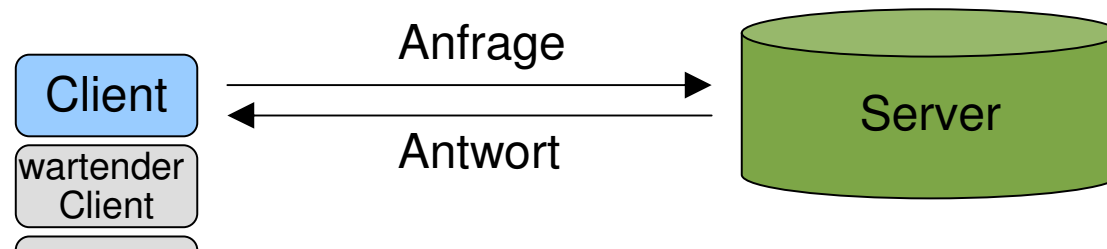
⇒ Veranschaulichung: Webserver

- **Tasks**: einzelne, von anderen unabhängige **Client-Anfragen**
- Möglichkeit von vielen **gleichzeitigen Anfragen**



# Möglichkeiten der Task-Ausführung

## Sequentielle Task-Ausführung



- sehr einfach
- theoretisch korrekt
- jedoch:**
- nur eine Anfrage gleichzeitig bearbeitbar  
⇒ **schlechte Leistung**

- ~~✗~~ hoher Datendurchsatz
- ~~✗~~ Schnelle Rückmeldung
- ~~✗~~ Prozessorauslastung
- ✓ Konstante Leistung



# Möglichkeiten der Task-Ausführung

---

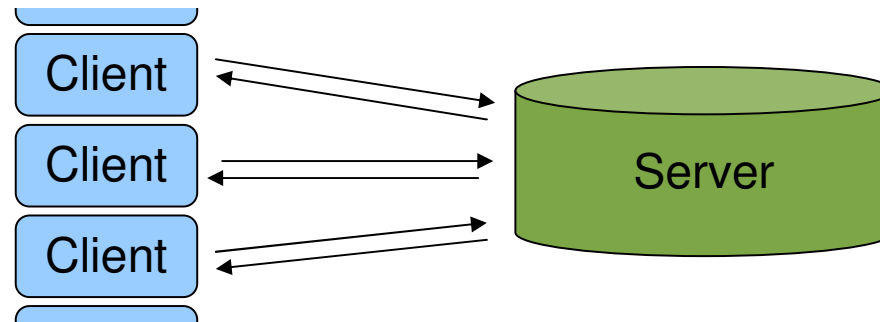
## Sequentielle Task-Ausführung

```
class SingleThreadWebServer {  
    public static void main(String[] args) throws IOException {  
        ServerSocket socket = new ServerSocket(80);  
  
        while (true) {  
            Socket connection = socket.accept();  
            handleRequest(connection);  
        }  
    }  
}
```



# Möglichkeiten der Task-Ausführung

## Task-Ausführung in Threads



- besseres Antwortverhalten
- evtl. schnellerer Datendurchsatz
- jedoch:**
- Unbegrenztes Erstellen von Threads kann zu Serverabsturz führen
- Code muss thread-safe sein

- ✓ hoher Datendurchsatz
- ✓ Schnelle Rückmeldung
- ✓ Prozessorauslastung
- ✗ Konstante Leistung



# Möglichkeiten der Task-Ausführung

## Task-Ausführung in Threads

```
class ThreadPerTaskWebServer {  
    public static void main(String[] args) throws IOException {  
        ServerSocket socket = new ServerSocket(80);  
  
        while (true) {  
            final Socket connection = socket.accept();  
            Runnable task =  
                new Runnable() {  
                    public void run() {  
                        handleRequest(connection);  
                    }  
                };  
  
            new Thread(task).start();  
        }  
    }  
}
```



# Möglichkeiten der Task-Ausführung

---

## Nachteile der unbegrenzten Thread-Erstellung

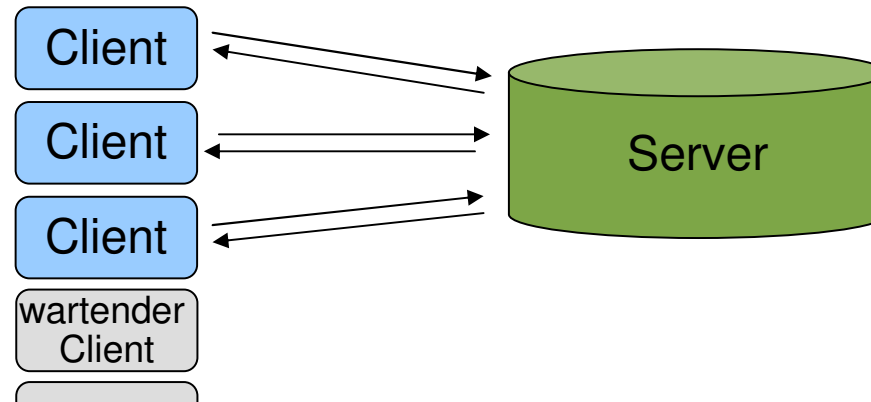
- Thread Lifecycle Overhead
- Ressourcenverbrauch
- Stabilität

**Lösung: Begrenzte Anzahl** an Threads



# Möglichkeiten der Task-Ausführung

## Task-Ausführung in beschränkter Anzahl an Threads



- gutes Antwortverhalten
  - schneller Datendurchsatz
  - optimierte Prozessorauslastung
- insbesondere:**
- unabhängig von Anzahl der Anfragen

- ✓ hoher Datendurchsatz
- ✓ Schnelle Rückmeldung
- ✓ Prozessorauslastung
- ✓ Konstante Leistung



# Möglichkeiten der Task-Ausführung

## Task-Ausführung in beschränkter Anzahl an Threads

```
class TaskExecutionWebServer {  
    private static final int NTHREADS = 100;  
    private static final Executor exec  
        = Executors.newFixedThreadPool(NTHREADS);  
  
    public static void main(String[] args) throws IOException {  
        ServerSocket socket = new ServerSocket(80);  
  
        while (true) {  
            final Socket connection = socket.accept();  
            Runnable task = new Runnable() {  
                public void run() {  
                    handleRequest(connection);  
                }  
            };  
            exec.execute(task);  
        }  
    }  
}
```

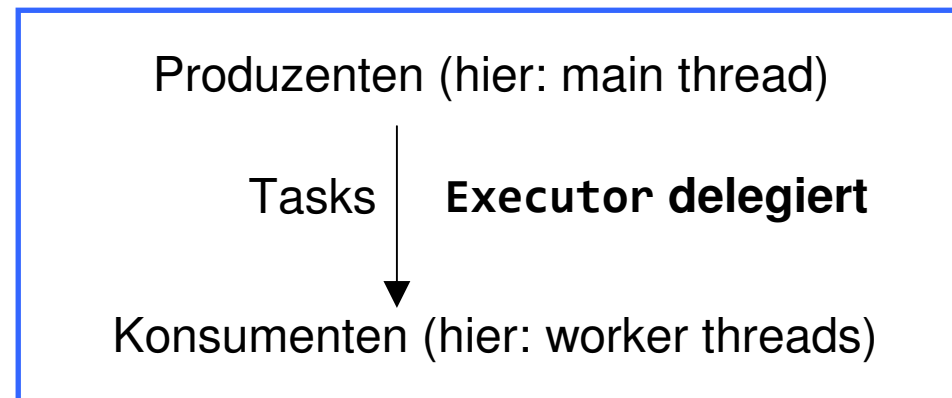


# Möglichkeiten der Task-Ausführung

## Realisierung: Executor

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

- große **Flexibilität** bezüglich Task-Ausführung
- ***producer - consumer pattern***



⇒ einfache, separate Anpassung möglich



# Möglichkeiten der Task-Ausführung

## Thread Pools

- Menge von worker threads in *queue*

- Vorteile:

- **Wiederverwendung** existierender Threads

⇒ keine Verzögerung durch Threaderzeugung

⇒ keine zusätzliche Belastung des Garbage Collectors

- **einfache Anpassung** an gegebene Situation

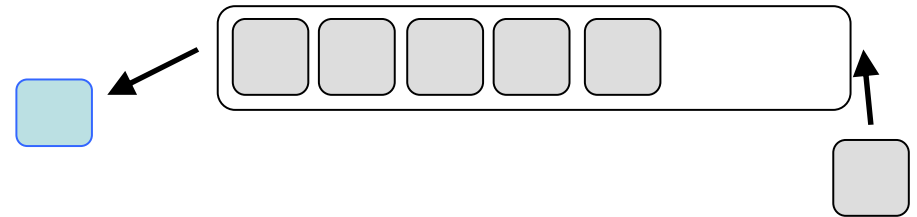
- Varianten:

`newFixedThreadPool`

`newCachedThreadPool`

`newSingleThreadExecutor`

`newScheduledThreadPool`





# Möglichkeiten der Task-Ausführung

## Execution Policy

- Werkzeug zur **Anpassung von Executor und** damit **seines Verhaltens**
- abhängig von verfügbaren Ressourcen

In **welchem Thread** werden Tasks bearbeitet?

In **welcher Reihenfolge** kommen Tasks zur Ausführung?

**Wie viele Tasks** dürfen **parallel** ausgeführt werden?

**Wie viele Tasks** dürfen **in Warteschlange** eingefügt werden?

**Welcher Task** soll notfalls abgewiesen werden?

...



# Möglichkeiten der Task-Ausführung

## Demonstration der Flexibilität

Task-Ausführung in Threads (unbegrenzte Anzahl)

```
public class ThreadPerTaskExecutor implements Executor {  
    public void execute(Runnable r) {  
        new Thread(r).start();  
    };  
}
```

Sequentielle Task-Ausführung

```
public class WithinThreadExecutor implements Executor {  
    public void execute(Runnable r) {  
        r.run();  
    };  
}
```



# Möglichkeiten der Task-Ausführung

## Beenden von Executor

```
public interface ExecutorService extends Executor {  
    void shutdown();  
    List<Runnable> shutdownNow();  
    boolean isShutdown();  
    boolean isTerminated();  
    boolean awaitTermination(long timeout, TimeUnit unit)  
        throws InterruptedException;  
    // ... weitere (nützliche) Methoden zur Task-Übermittlung  
}
```

### Schnittstelle ExecutorService

- Methoden für **Zustandsmanagement**
- Zustände von ExecutorService:

*running* → *shutting down* → *terminated*

```

class LifecycleWebServer { // Webserver mit Shutdown-Unterstützung
    private final ExecutorService exec = ...; // entsprechend
                                                Execution Policy

    public void start() throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (!exec.isShutdown()) {
            try {
                final Socket conn = socket.accept();
                exec.execute(new Runnable() {
                    public void run() { handleRequest(conn); }
                });
            } catch (RejectedExecutionException e) {
                if (!exec.isShutdown()) log("task submission rejected", e);
            }
        }
    }

    public void stop() { exec.shutdown(); }

    void handleRequest(Socket connection) {
        Request req = readRequest(connection);
        if (isShutdownRequest(req))
            stop();
        else dispatchRequest(req);
    }
}

```



# Parallelisieren von Tasks

---

## Problemstellung

Aufgabe soll unterteilt werden

möglichst unabhängige Tasks

effiziente Task-Grenzen

⇒ Veranschaulichung: Webbrowser

● **Aufgabe:** Rendern von Webseite

● **Mögliche Tasks:** Anzeige von Text, Bildern, Grafiken, etc.



# Parallelisieren von Tasks

## Zwei Tasks in einem Thread

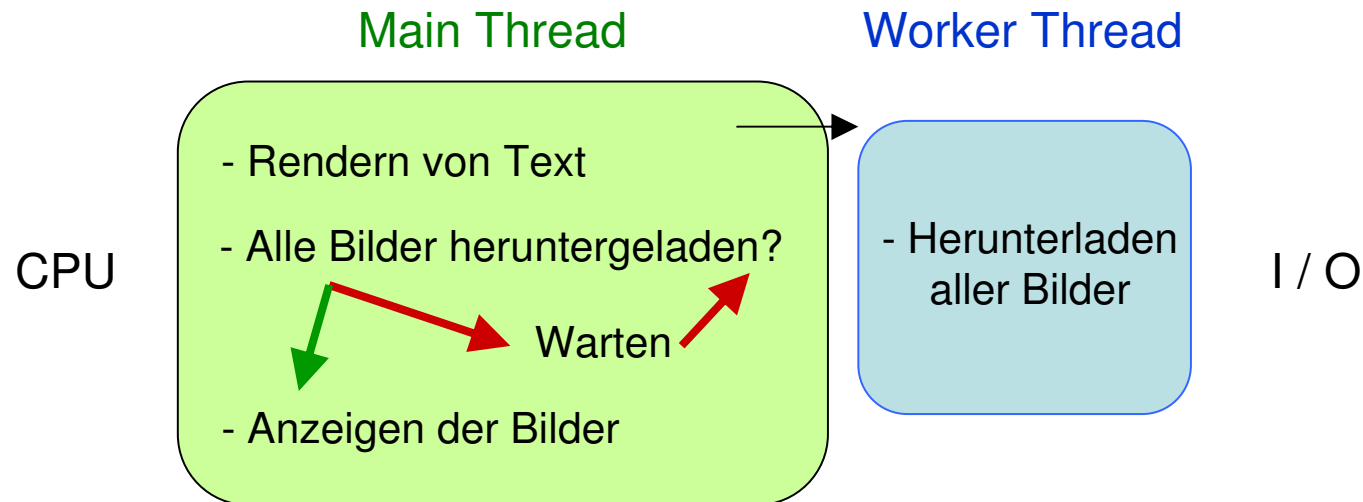
```
public class SingleThreadRenderer {  
    void renderPage(CharSequence source) {  
        renderText(source);  
  
        List<Image> images = new ArrayList<Image>();  
  
        for (ImageAddress imageAddress : scanForImageAddresses(source))  
            images.add(imageAddress.downloadImage());  
  
        for (Image image : images)  
            renderImage(image);  
    }  
}
```

⇒ bessere Lösung: Aufteilen der Aufgabe in Tasks und Ausführung in Threads



# Parallelisieren von Tasks

## Zwei Tasks in zwei Threads



Wünschenswert: Mechanismen, um

- den **Abarbeitungszustand** von Worker **Thread** zu bestimmen
- den Worker Thread etwas **zurückgeben** zu lassen

⇒ interfaces Future und Callable



# Parallelisieren von Tasks

## Realisierung: interface Future

```
public interface Future<V> {
```

```
    boolean cancel(boolean mayInterruptIfRunning);
```

```
    boolean isCancelled();
```

```
    boolean isDone();
```

```
    V get() throws
```

```
        InterruptedException, ExecutionException, CancellationException;
```

```
    V get(long timeout, TimeUnit unit) throws InterruptedException,  
        ExecutionException, CancellationException, TimeoutException;
```

```
}
```



# Parallelisieren von Tasks

## Realisierung: interface Callable

Wie Runnable, jedoch zusätzlich:

- return - Funktionalität
- Werfen von Ausnahmen

```
public interface Callable<V> {  
    V call() throws Exception;  
}
```

`ExecutorService.submit(Callable)` **gibt Future zurück**

⇒ komfortable Überwachung der Task-Ausführung

Alternative: FutureTask

```
public class FutureRenderer {
    private final ExecutorService executor = ...;
                                                    // entsprechend Execution Policy
    void renderPage(CharSequence source) {
        final List<ImageAddress> imageAddresses =
            scanForImageAddresses(source);
```

```
        Callable<List<Image>> task =
            new Callable<List<Image>>() {
                public List<Image> call() {
                    List<Image> result= new ArrayList<Image>();
                    for (ImageAddress imageAddress : imageAddresses)
                        result.add(imageAddress.downloadImage());
                    return result;
                }
            };
```

```
        Future<List<Image>> future = executor.submit(task);
        renderText(source);
```

```
        try {
            List<Image> images = future.get();
            for (Image image : images)
                renderImage(image);
        } catch ... // Abfangen und Behandeln der Exceptions
```

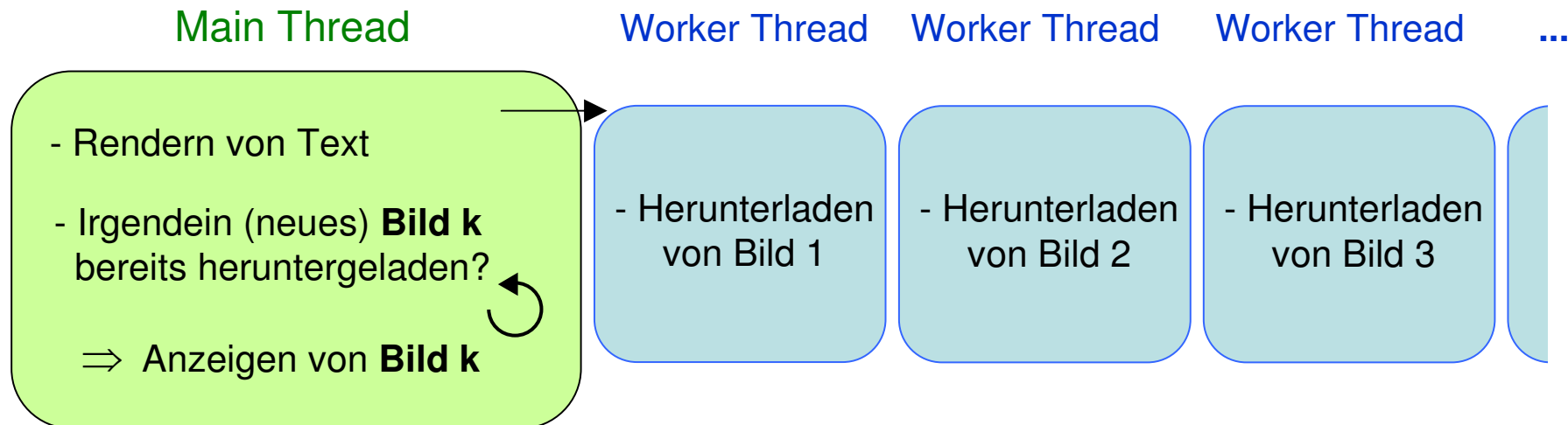
```
    }
```

```
}
```



# Parallelisieren von Tasks

## Mehrere Tasks in mehreren Threads



Wünschenswert: Mechanismus, um

- **nicht ständig** jedes **Future überprüfen** zu müssen
- sondern die **Ergebnisse automatisch zurück** zu erhalten

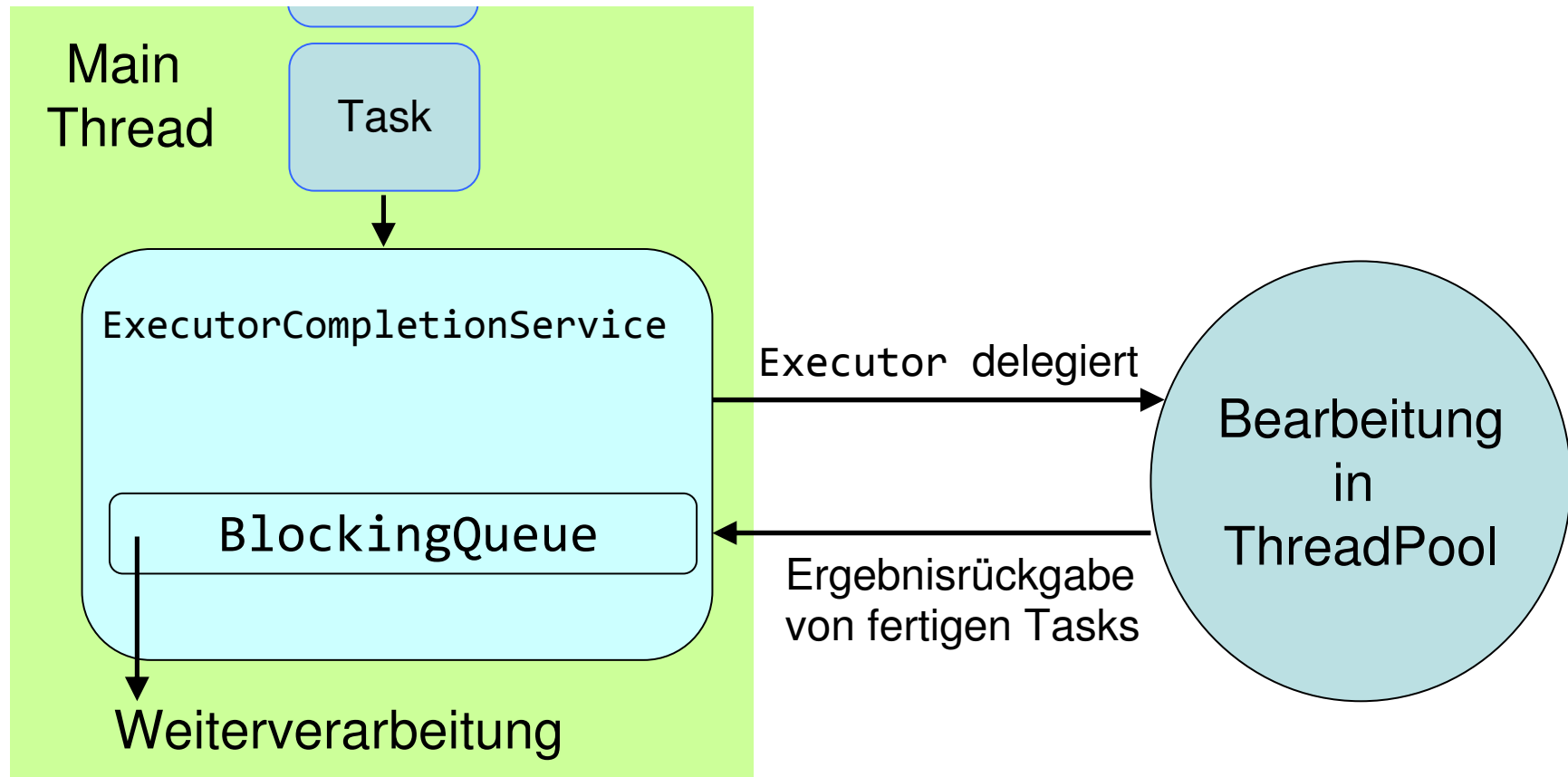
⇒ `interface CompletionService`



# Parallelisieren von Tasks

## Realisierung: CompletionService

entkoppelt Task-Produktion und Konsumieren der Ergebnisse



```

public class Renderer {
    private final ExecutorService executor = ...;
                                                    // entsprechend Execution Policy

    void renderPage(CharSequence source) {
        final List<ImageAddress> imageAddresses = scanForImageAddresses(source);
        CompletionService<Image> completionService =
            new ExecutorCompletionService<Image>(executor);
        for (final ImageAddress imageAddress : imageAddresses)
            completionService.submit(new Callable<Image>() {
                public Image call() {
                    return imageAddress.downloadImage();
                }
            });

        renderText(source);

        try {
            for (int t = 0, n = imageAddresses.size(); t < n; t++) {
                Future<Image> f = completionService.take();
                Image image = f.get();
                renderImage(image);
            }
        } catch ... // Abfangen und Behandeln der Exceptions
    }
}

```



# Verzögerung / zeitliche Begrenzung von Tasks

Timer

vs

ScheduledThreadPoolExecutor

- + verzögerte Tasks
- + periodisch auszuführende Tasks

- einzelner Thread
- unchecked exceptions führen zu Thread-Abbruch
  - ⇒ Timer wird beendet

- + mehrere Threads möglich
- + korrekte Behandlung von Ausnahmen

⇒ ScheduledThreadPoolExecutor



# Verzögerung / zeitliche Begrenzung von Tasks

---

## Problemstellung

Aufgabe soll nicht durch zeitintensive Tasks aufgehalten werden

Abbruch von Tasks ab definiertem Zeitpunkt

⇒ Veranschaulichung: Webseite

- Aufgabe: Ausgabe der kompletten Webseite
- Tasks: Einfügen von Werbung von externen Servern



# Verzögerung / zeitliche Begrenzung von Tasks

## Induktionsanfang :)

```
Page renderPageWithAd() throws InterruptedException {
    long endNanos = System.nanoTime() + TIME_BUDGET;
    Future<Ad> future = exec.submit(new FetchAdTask());
    Page page = renderPageBody();
    Ad ad;

    try {
        long timeLeft = endNanos - System.nanoTime();
        ad = future.get(timeLeft, NANOSECONDS);
    }
    catch (ExecutionException e) {
        ad = DEFAULT_AD;
    }
    catch (TimeoutException e) {
        ad = DEFAULT_AD;
        future.cancel(true);
    }

    page.setAd(ad);
    return page;
}
```



# Verzögerung / zeitliche Begrenzung von Tasks

## Induktionsschritt ;)

- Methode `get(long timeout, TimeUnit unit)` begrenzt Task zeitlich
- jedoch: kein implizites Abbrechen des Tasks

Wünschenswert: Mechanismus, um **nach einer bestimmten Zeitdauer**

- **alle bisherigen Ergebnisse zurück** zu erhalten und
- noch **nicht fertige Tasks abbrechen**

⇒ Methode `invokeAll` des `ExecutorService`

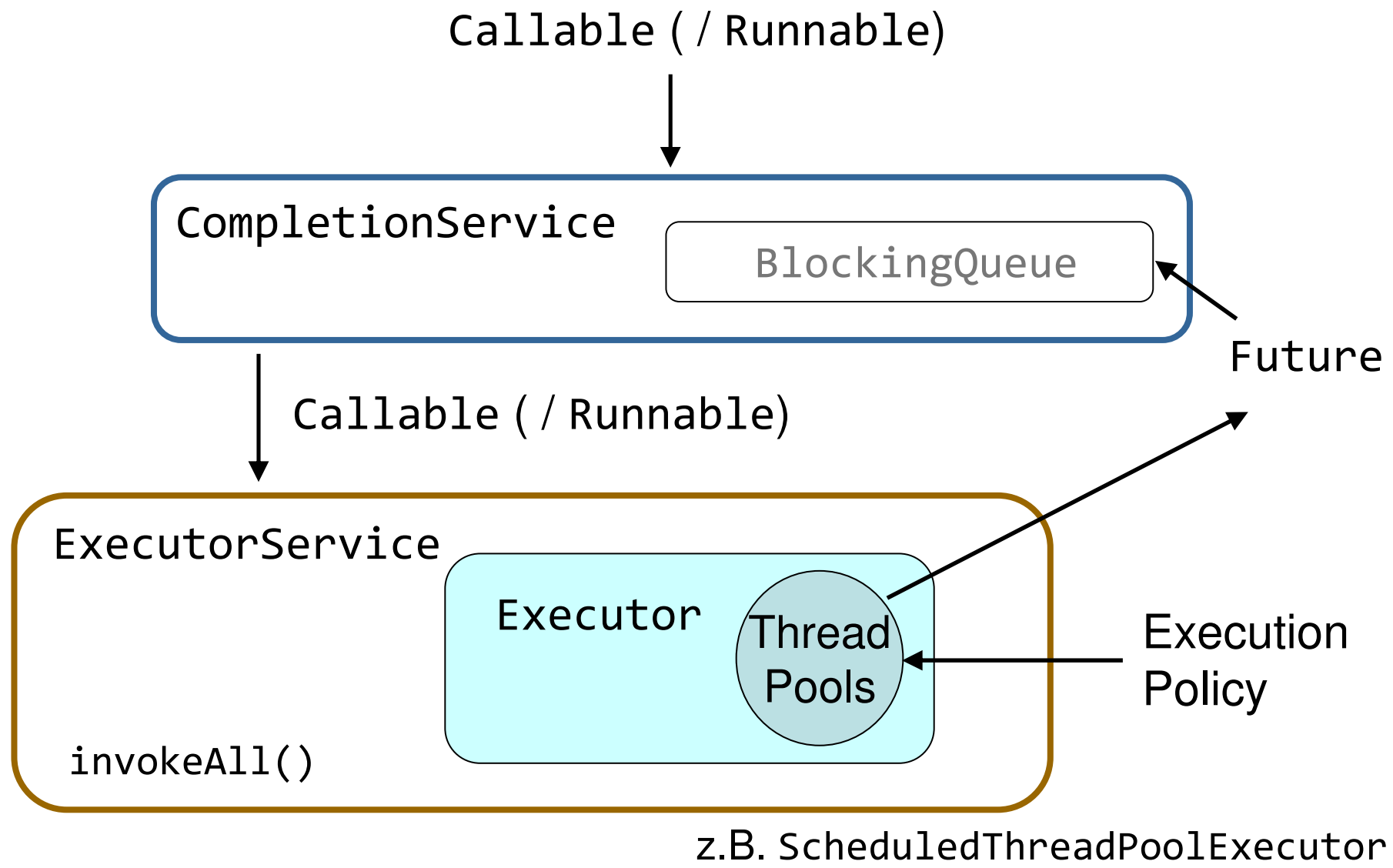
```
private class FetchAdTask implements Callable<Ad> {
    private final Company company;
    public Ad call() throws Exception {
        return company.requestAd();
    }
}
```

```
public List<Ad> getAds(Set<Company> companies, long time, TimeUnit unit)
    throws InterruptedException {
    List<FetchAdTask> tasks = new ArrayList<FetchAdTask>();
    for (Company company : companies)
        tasks.add(new FetchAdTask(company));
    List<Future<Ad>> futures = exec.invokeAll(tasks, time, unit);
    List<Ad> advertisements = new ArrayList<Ad>(tasks.size());

    for (Future<Ad> future : futures) {
        try {
            advertisements.add(future.get());
        } catch (ExecutionException e) {
            advertisements.add(EMPTY_AD);
        } catch (CancellationException e) {
            advertisements.add(EMPTY_AD);
        }
    }
    return advertisements;
}
```



# Rückblick



**Vielen Dank für Eure  
Aufmerksamkeit!**



ERROR: undefined  
OFFENDING COMMAND:  
  
STACK: