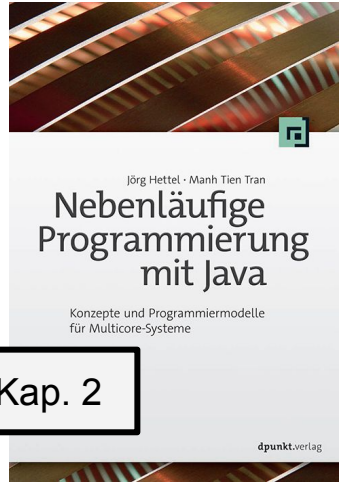
Parallele Programmierung

Java Threads

Überblick

- Thread-Nebenläufigkeit
- Start eines Threads in Java
  - durch Ableiten von **`Thread`**
  - **Laboraufgabe**
  - durch Implementieren von **`Runnable`** und **`Thread`** im Konstruktor Übergeben
  - **Laboraufgabe**
- Lebenszyklus von Java Threads
  - Beenden von Threads
  - **Laboraufgabe**
  - Warten auf das Ende anderer Threads
  - Lebenszyklus von Threads
  - **Laboraufgabe**



Kap. 2

Thread-Nebenläufigkeit

Nebenläufigkeit

- Programmteil unabhängig von anderen Programmteilen
- kann parallel zu anderen nebenläufigen Teilen abgearbeitet werden
- Wenn zu einem Zeitpunkt mehr nebenläufige Teile zur Abarbeitung anstehen als Kerne (*CPU Cores*) zur Verfügung stehen, können diese Teile auch durch einen Scheduler durch Wechsel zwischen den Threads überlappend abgearbeitet werden (“Pseudo-Multitasking”).
- JVM startet immer den “main”-Thread, in dem das Hauptprogramm abgearbeitet wird.
    - Daneben startet die JVM nebenläufige Teile zur Verwaltung des main-Threads (z.B. Garbage Collection).

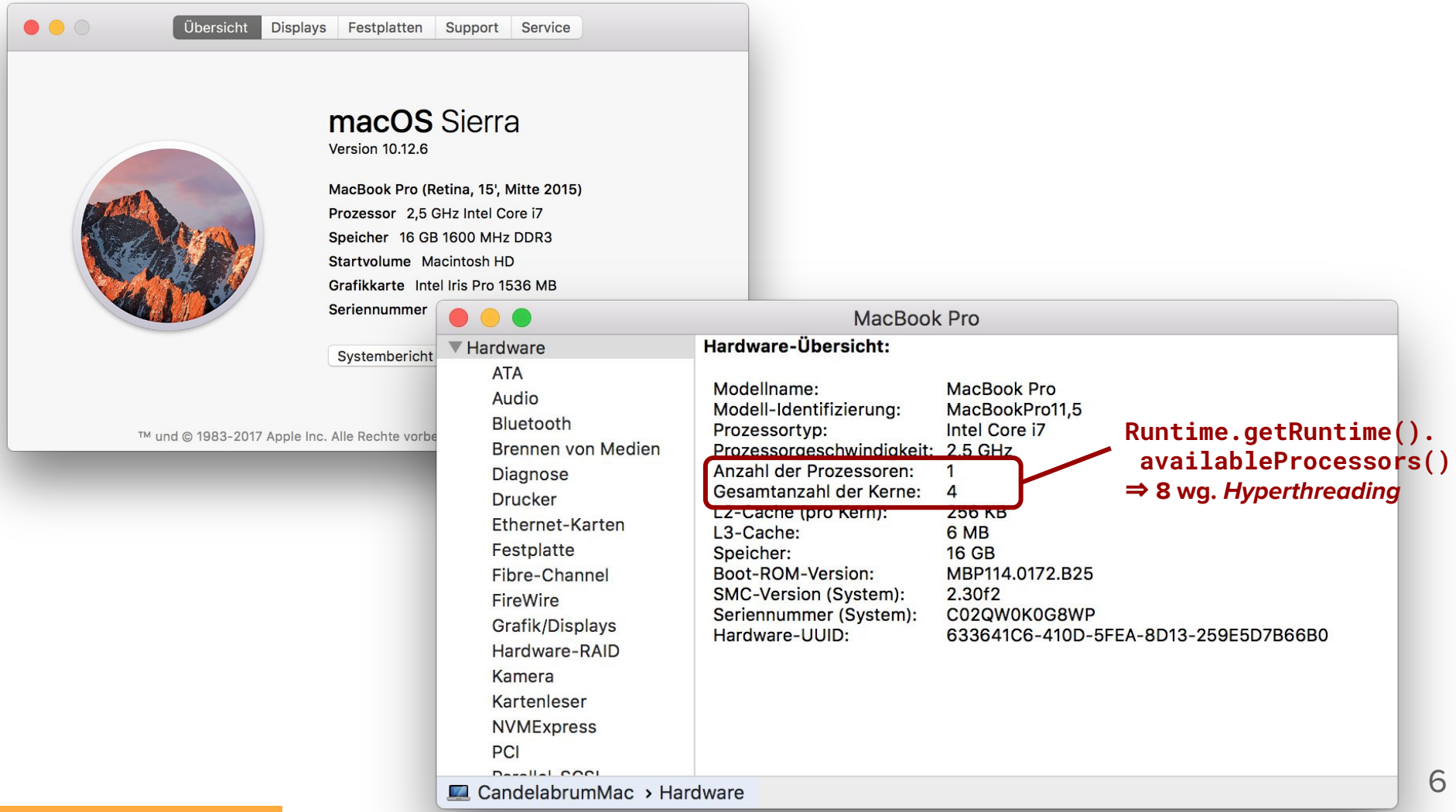
Folie mit Anmerkungen

Grundlagen zum Java-Thread

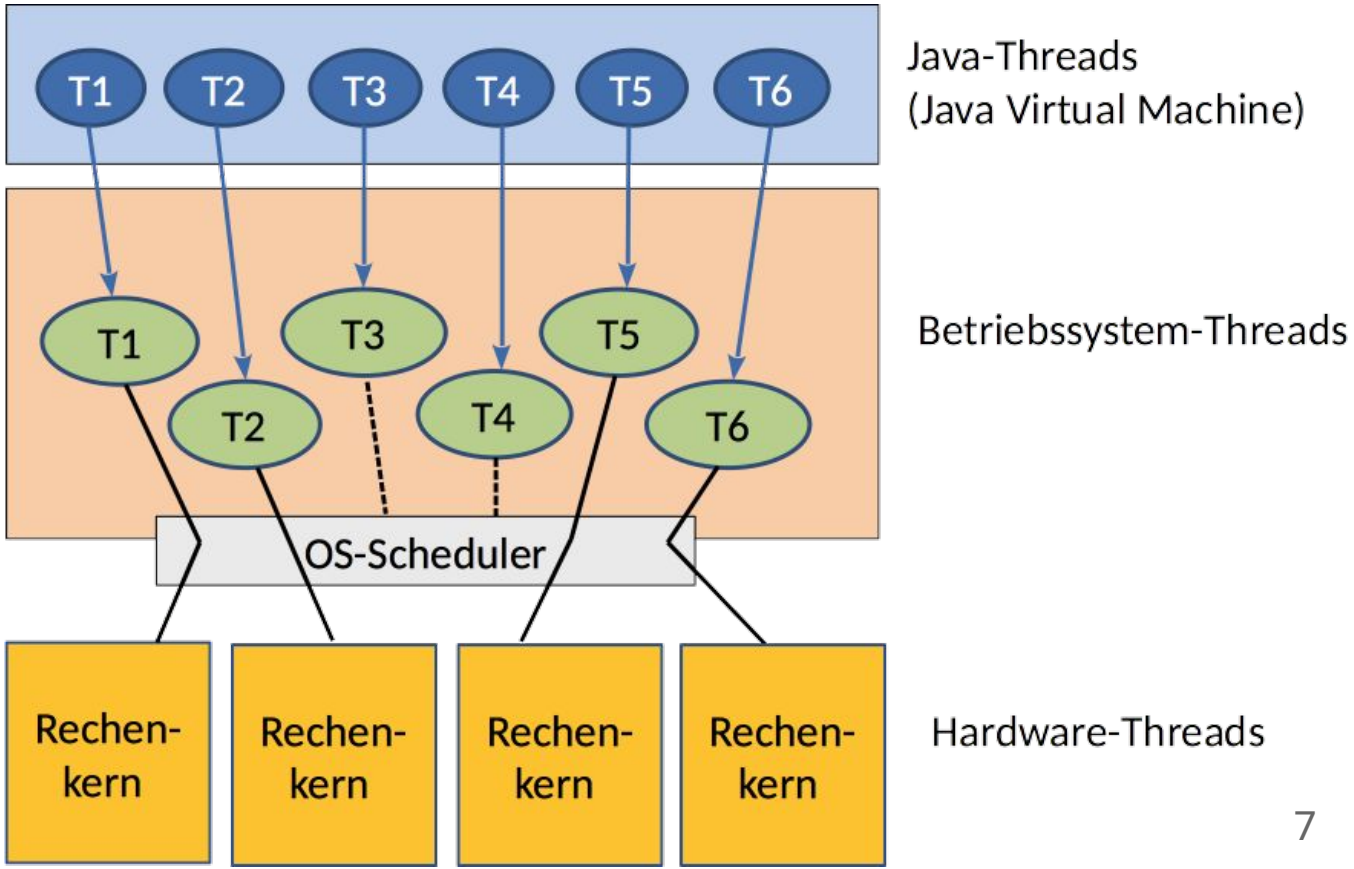
```
public class MainThread {
    public static void main(String[] args) {
        // Anzahl der Prozessoren abfragen
        var nr = Runtime.getRuntime().availableProcessors();
        System.out.println("Anzahl der Prozessoren " + nr);
        // Eigenschaften des main-Threads
        var self = Thread.currentThread();
        System.out.println("Name : " + self.getName());
        System.out.println("Priorität : " + self.getPriority());
        System.out.println("ID : " + self.getId());
    }
}
```

Anzahl Cores

der Thread, in dem die Methode ausgeführt wird, hier *main-Thread*


Übersicht
Displays
Festplatten
Support
Service
macOS Sierra
Version 10.12.6
MacBook Pro (Retina, 15', Mitte 2015)
Prozessor 2,5 GHz Intel Core i7
Speicher 16 GB 1600 MHz DDR3
Startvolume Macintosh HD
Grafikkarte Intel Iris Pro 1536 MB
Seriennummer
Systembericht
™ und © 1983-2017 Apple Inc. Alle Rechte vorbe
MacBook Pro
Hardware
ATA
Audio
Bluetooth
Brennen von Medien
Diagnose
Drucker
Ethernet-Karten
Festplatte
Fibre-Channel
FireWire
Grafik/Displays
Hardware-RAID
Kamera
Kartenleser
NVMExpress
PCI
Hardware-Übersicht:
Modellname: MacBook Pro
Modell-Identifizierung: MacBookPro11,5
Prozessortyp: Intel Core i7
Prozessorgeschwindigkeit: 2,5 GHz
Anzahl der Prozessoren: 1
Gesamtanzahl der Kerne: 4
L2-Cache (pro Kern): 256 KB
L3-Cache: 6 MB
Speicher: 16 GB
Boot-ROM-Version: MBP114.0172.B25
SMC-Version (System): 2.30f2
Seriennummer (System): C02QW0K0G8WP
Hardware-UUID: 633641C6-410D-5FEA-8D13-259E5D7B66B0
CandelabrumMac › Hardware
Runtime.getRuntime().
availableProcessors()
⇒ 8 wg. Hyperthreading

Zuordnung der Java-Threads zu einzelnen Kernen



Bildquelle: Hettel & Tran, 2016

Start eines Threads in Java

Start eines Threads in Java

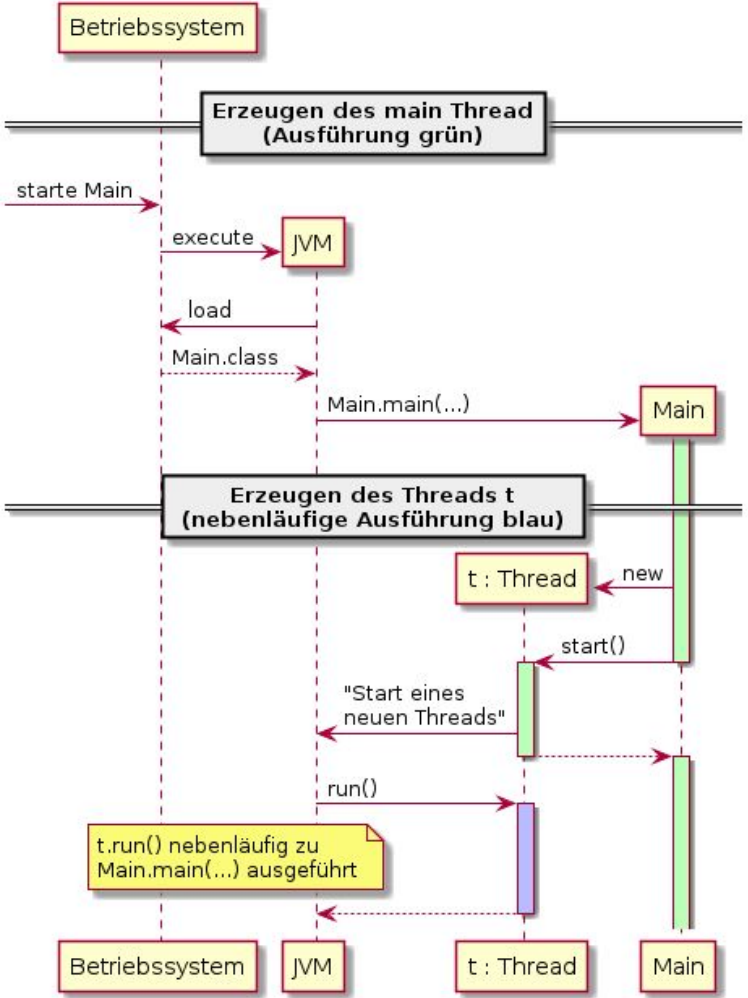
- Man leitet direkt von der Klasse **`Thread`** ab und überschreibt die **`run`**-Methode.
- Man stellt eine Klasse bereit, die das **`Runnable`**-Interface implementiert. Ein Objekt dieser Klasse wird auch oft als *Task* bezeichnet. Es wird dann einem **`Thread`**-Objekt zur Ausführung übergeben.

Die so vorbereitete **`Thread`**-Instanz wird durch Aufruf der Methode `start()` zur nebenläufigen Abarbeitung markiert. Der Java-Scheduler sorgt dafür, dass Kontextwechsel zu diesem Thread erfolgen bzw. dass er in einer Execution-Engine (“Prozessor”) abgearbeitet wird. Wann Wechsel zu diesem Thread erfolgen, liegt aber in der Verantwortung des Schedulers, der eine eigene Strategie verfolgen kann. Die ***Priority*** Eigenschaft des Threads sollte dabei berücksichtigt werden.

Sequenzdiagramm für das Starten eines neuen Threads

Erzeugen des main Thread

- Das Betriebssystem startet die Java Virtual Machine (JVM).
- Die JVM lädt die zu startende Klasse (hier `Main.class`) mit einem System Call und
- startet die main-Methode in einem eigenen Thread (“main Thread”) hier als grüne Lifeline dargestellt.

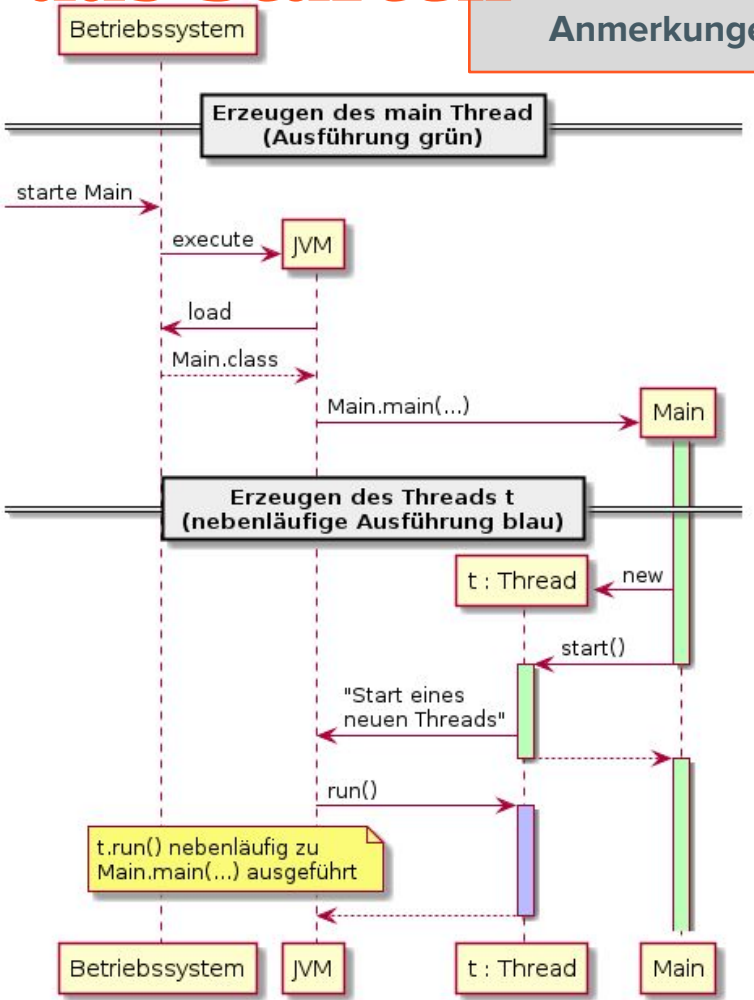


Sequenzdiagramm für das Starten eines neuen Threads

Folie mit Anmerkungen

Erzeugen des Threads t

- Aus dem main Thread (grüne Lifeline) wird ein neues Thread-Objekt instanziiert.
- An diesem Objekt wird die Methode `start()` aus dem main Thread heraus aufgerufen.
- `start()` bewirkt, dass die JVM einen neuen Thread bereitstellt, und die `run()` Methode von t darin ausführt.
- Die run() Methode wird nebenläufig zum main Thread ausgeführt (blaue Lifeline).



Laboraufgabe (15 Minuten)
Thread durch Vererbung von Thread

`pp.01.01-Inheritance`

Thread-Konstruktor und Runnable

```
public Thread(Runnable target)
public Thread(Runnable target, String name)
```

Dem Konstruktor von **`Thread`** kann man ein Objekt als Parameter übergeben, das das **`Runnable`**-Interface implementiert. Optional kann auch ein **`String`** für die ***Name***-Eigenschaft mit übergeben werden.

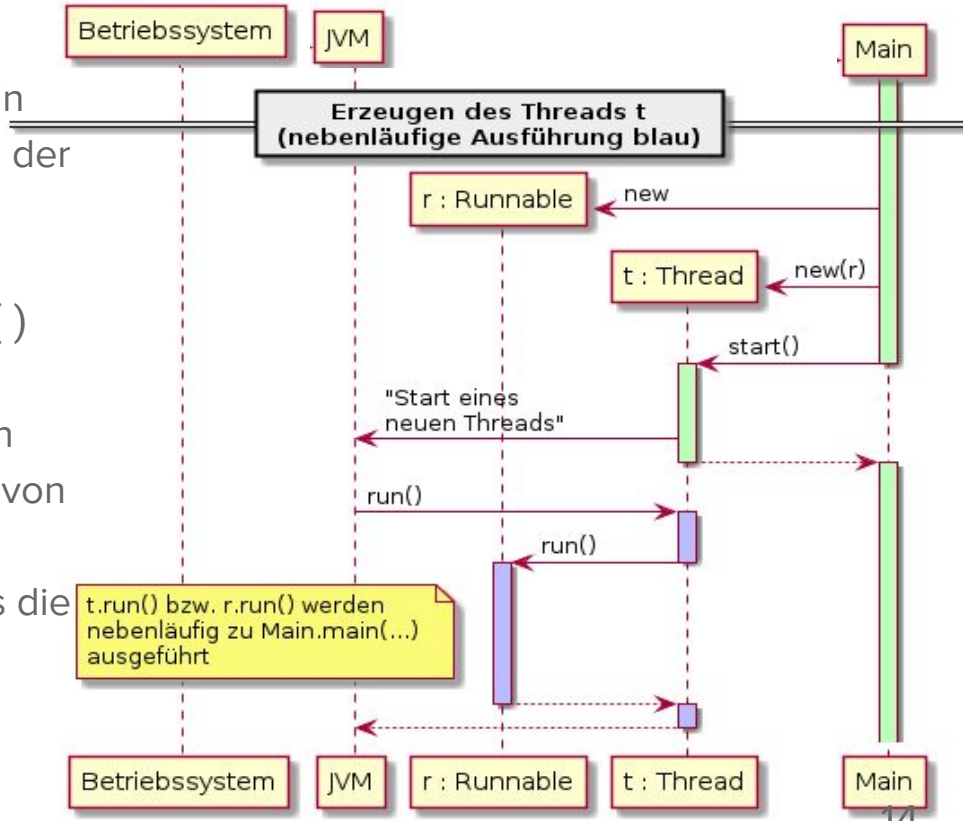
```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

Starten eines neuen Threads mithilfe eines Runnable-Objekts

Folie mit Anmerkungen

Erzeugen des Threads t

- Aus dem main Thread (grüne Lifeline) wird ein neues Thread-Objekt instanziiert. Dabei wird der Konstruktor verwendet, der ein "Runnable-Objekt" als ersten Parameter hat.
- An diesem Objekt wird die Methode `start()` aus dem main Thread heraus aufgerufen.
- `start()` bewirkt, dass die JVM einen neuen Thread bereitstellt, und die `run()` Methode von t darin ausführt.
- Die run() Methode von t besteht daraus, dass die run() Methode des "Runnable-Objekts" ausgeführt wird.
- Die run() Methoden werden nebenläufig zum main Thread ausgeführt (blaue Lifelines).



Runnable als anonyme innere Klasse

```
new Thread(new Runnable()    {

    @Override
    public void run() {
        //...
    }
});
```

Runnable als Lambda-Ausdruck

```
new Thread(                () ->


                          {
        //...
    }
);
```

oder kurz:

```
new Thread(() -> {...});
```

Laboraufgabe (15 Minuten) Thread durch Erzeugen und Übergeben eines Runnable

`pp.01.02-Runnable`

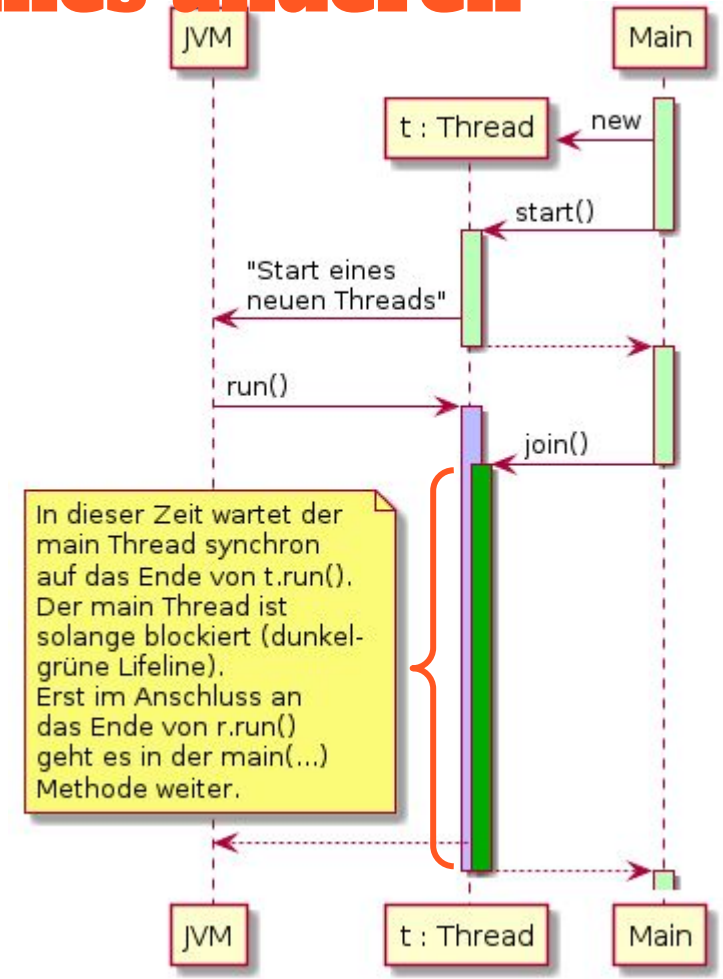
Lebenszyklus von Java Threads

Warten auf das Ende eines anderen Threads

```
void join()
void join(long millis)
void join(long millis, int nanos)
```

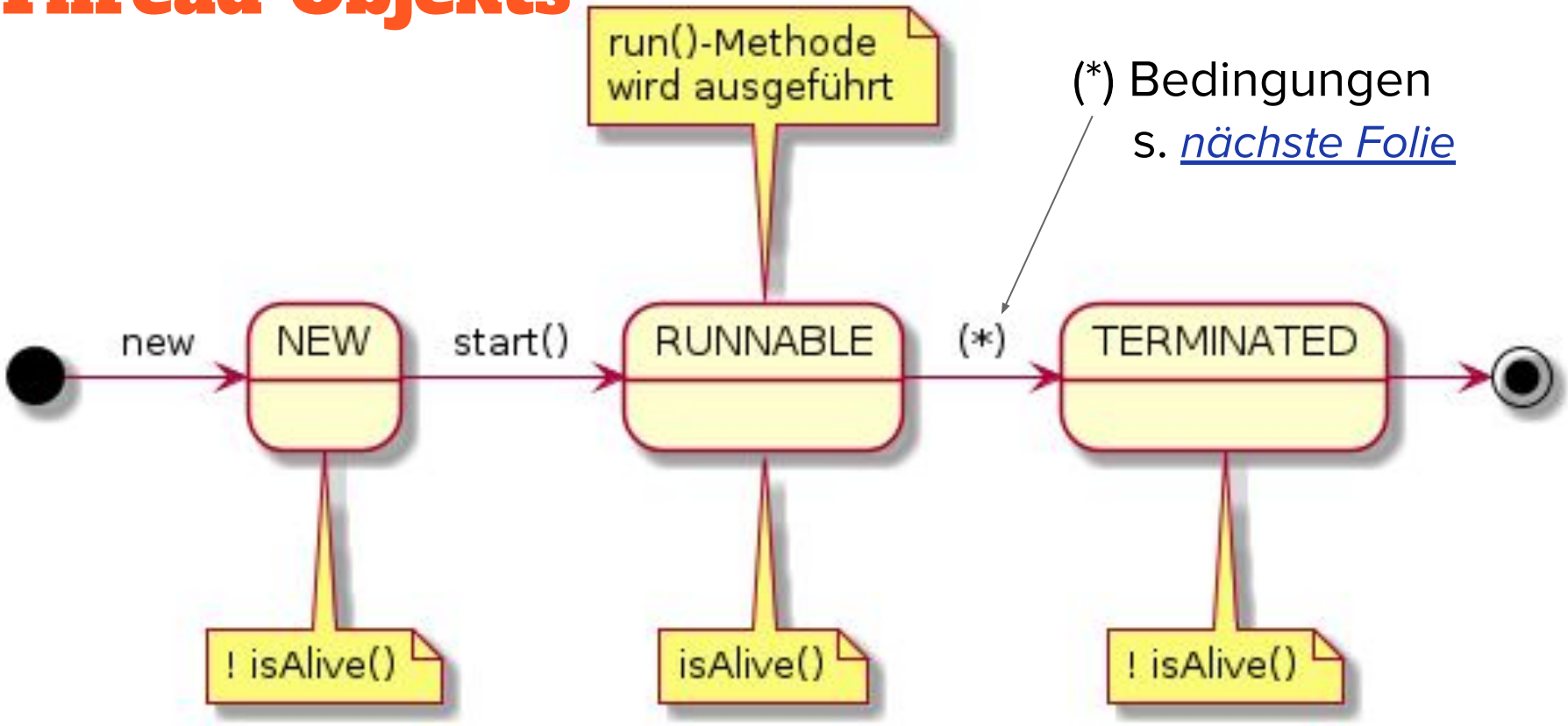
Die Varianten mit Parameter warten für eine gewisse Zeit. Sollte der Thread, auf dessen Ende gewartet wird, bis dahin nicht beendet sein, geht es trotzdem weiter.

`millis == 0` bedeutet
“für immer” (genau wie `join()`)



Vereinfachter Lebenszyklus eines Thread-Objekts

Folie mit Anmerkungen



Bildquelle: Hettel & Tran, 2016 (abgewandelt)

Thread-Ende

- reguläres Ende der `run()`-Methode
- nicht behandelte Exception in `run()`
- ein anderer Thread ruft `stop()` am zu beendenden Thread auf (***deprecated*** ebenso wie `pause()` und `resume()`!)
- hat Daemon-Eigenschaft (`Thread.currentThread().setDaemon(true)`) und alle User-Threads sind beendet
- `System.exit()` wird in irgendeinem Thread der JVM aufgerufen

Stoppen eines Threads durch Variablen-Signal

Folie mit Anmerkungen

```java
public class Main {
    public static void main(String[] args)
            throws InterruptedException {
        var task = new Task();
        var thread = new Thread(task);
        thread.start();
        Thread.sleep(4000);
        task.stopRequest();
        thread.join();
    }
}
```

```java
public class Task implements Runnable {
    private volatile boolean stopped;
    private volatile Thread self;

    public boolean isStopped() {
        return this.stopped;
    }

    public void stopRequest() {
        this.stopped = true;
        if (this.self != null) {
            this.self.interrupt();
        }
    }

    @Override
    public void run() {
        this.self = Thread.currentThread();
        while (!isStopped()) {
            // ... arbeiten ...
        }
        // ... aufräumen ...
    }
}
```

Stoppen eines Threads durch Variablen-Signal (Alternative)

```java
public class Main {
    public static void main(String[] args)
        throws InterruptedException {
        var thread = new Task();
        thread.start();
        Thread.sleep(4000);
        thread.stopRequest();
        thread.join();
}
```

Alternativ: Hier erbt der nebenläufige Task von `Thread`, statt dem Thread-Konstruktor im main-Thread ein `Runnable` als Parameter zu übergeben.

```java
public class Task extends Thread {
    private volatile boolean stopped;
    private volatile Thread self;

    public boolean isStopped() {
        return this.stopped;
    }

    public void stopRequest() {
        this.stopped = true;
        if (this.self != null) {
            this.self.interrupt();
        }
    }

    @Override
    public void run() {
        this.self = Thread.currentThread();
        while (!isStopped()) {
            // ... arbeiten ...
        }
        // ... aufräumen ...
    }
}
```

Stoppen eines Threads durch Variablen-Signal (Alternative)

```java
public class Main {
    public static void main(String[] args)
        throws InterruptedException {
        var thread = new Task();
        thread.start();
        Thread.sleep(4000);
        thread.stopRequest();
        thread.join();
}
```

In diesem Fall kennt der Thread sich selbst und kann damit den Interrupt anders umsetzen:

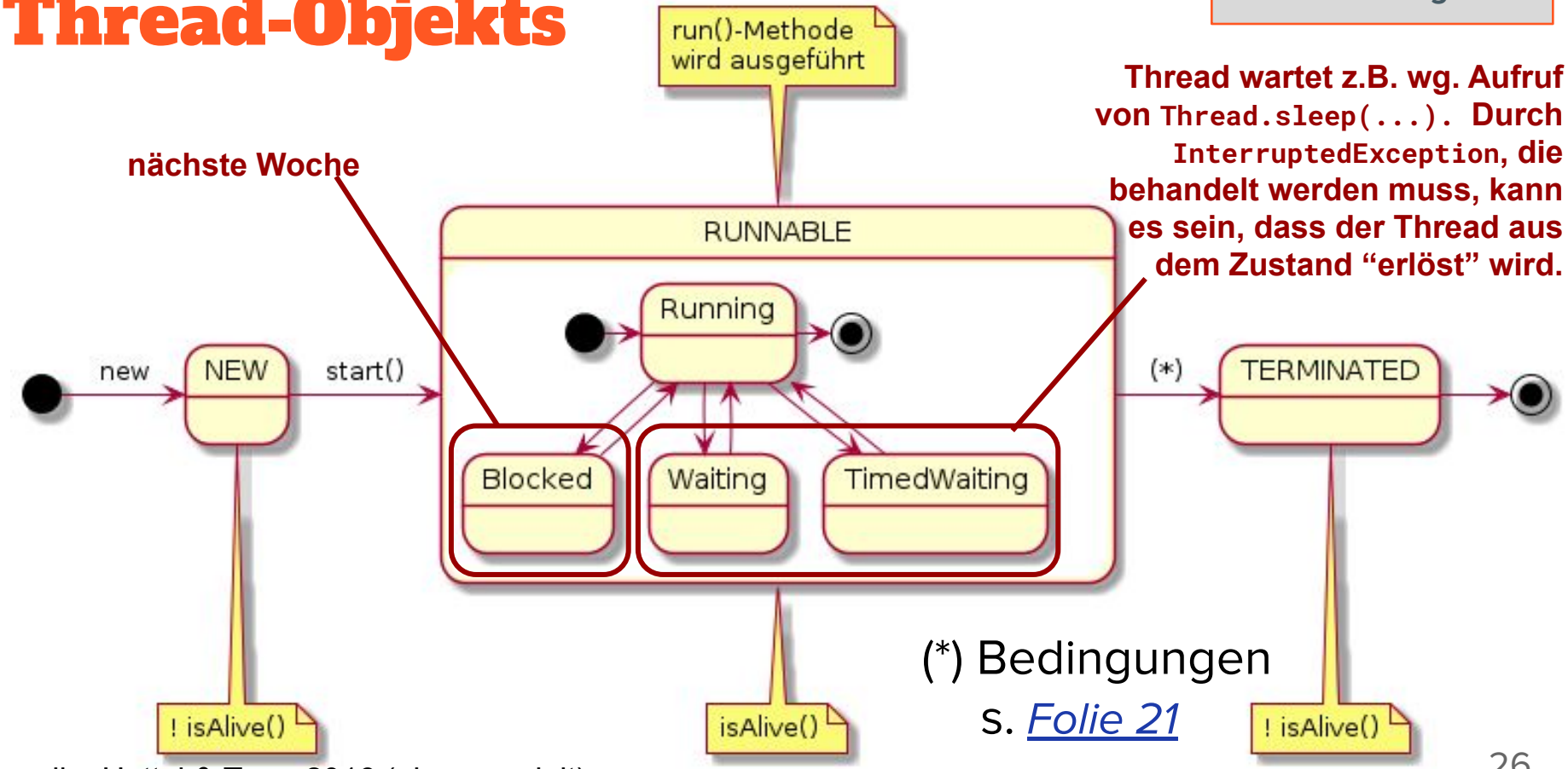
```java
public class Task extends Thread {
    private volatile boolean stopped;


    public boolean isStopped() {
        return this.stopped;
    }

    public void stopRequest() {
        this.stopped = true;
        if (this.isAlive()) {
            this.interrupt();
        }
    }

    @Override
    public void run() {

        while (!isStopped()) {
            // ... arbeiten ...
        }
        // ... aufräumen ...
    }
}
```

Laboraufgabe (15 Minuten)
Stoppen eines Tasks durch Variablen Signal

`pp01.03-EndThread`

Lebenszyklus (Zustände) eines Thread-Objekts

Folie mit Anmerkungen



(*) Bedingungen
s. *Folie 21*

Bildquelle: Hettel & Tran, 2016 (abgewandelt)

Laboraufgabe (25 Minuten)
Umsetzung des *Observer Patterns*

`pp01.04-TaskObserver`