

# Objektorientierte Programmierung in Java

## Vorlesung 10 - Parallel Computing

Emily Lucia Antosch

HAW Hamburg

11.11.2024

# Inhaltsverzeichnis

1. Einleitung .....	2
2. Parallelverarbeitung .....	6
3. Klassenbasierte Threads .....	14
4. Interface-basierte Threads .....	20
5. Zustände und ausgewählte Methoden .....	25
6. Synchronisation .....	36
7. License Notice .....	42

# 1. Einleitung

---

- In der letzten Vorlesung haben wir uns mit der Ausgabe und Eingabe beschäftigt
- Sie können nun
  - Ausgabe auf der Konsole in den richtigen Kanal senden und formatieren,
  - Eingabe vom User einfordern
  - und Dateien in Java einlesen.
- Heute geht es weiter mit der **Parallelverarbeitung**.

# 1.1 Wo sind wir gerade?

1. Imperative Konzepte
2. Klassen und Objekte
3. Klassenbibliothek
4. Vererbung
5. Schnittstellen
6. Graphische Oberflächen
7. Ausnahmebehandlung
8. Eingaben und Ausgaben
9. **Multithreading (Parallel Computing)**

- Sie führen Programmcode zeitgleich in nebenläufigen Ausführungssträngen (Threads) aus.
- Sie verändern die Zustände aktiver Threads zur Erzeugung der geforderten Funktionalität.
- Sie synchronisieren Threads und Objekte, um fehlerhafte Datenzustände durch nicht korrekte Ausführungsreihenfolgen zu verhindern.

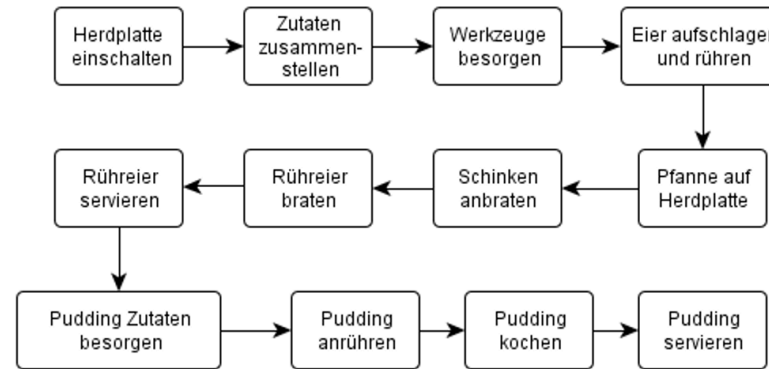
## 2. Parallelverarbeitung

---

# 2.1 Beispiel: Rührei und Pudding

## 2. Parallelverarbeitung

- Sie machen Rührei und Pudding.
- Möglicher Ablauf:



### ? Frage

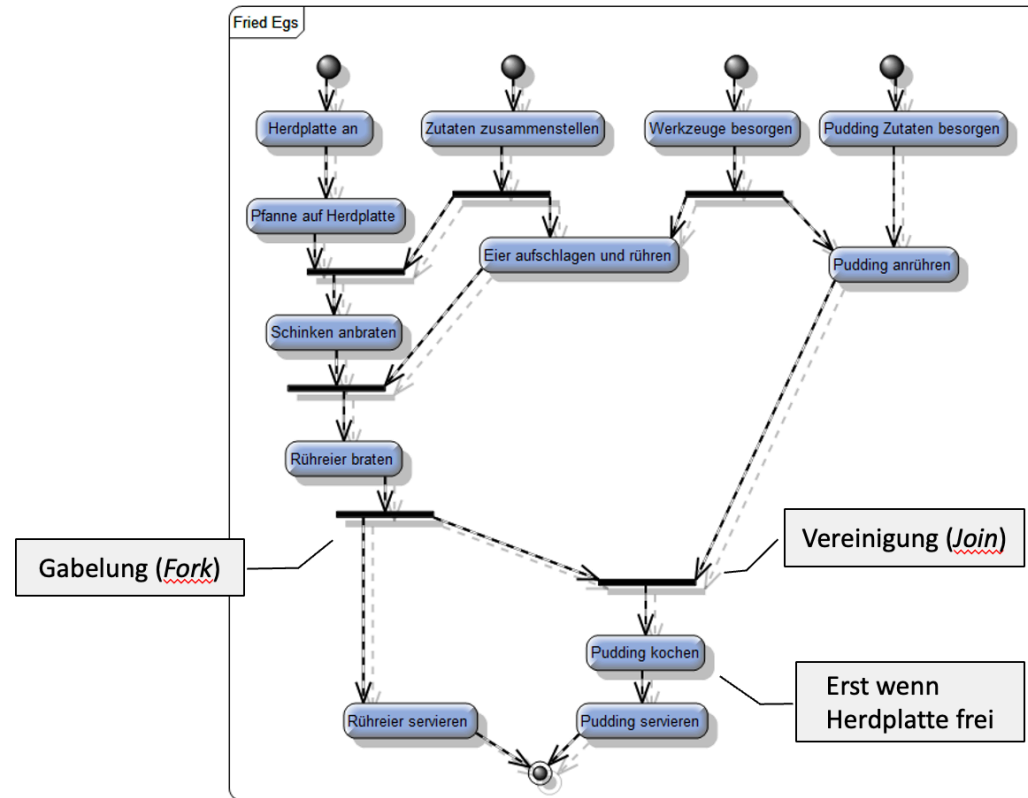
- Wie könnte der Ablauf aussehen, wenn Sie zu viert kochen?
- Einschränkung: Es gibt nur eine Herdplatte



# 2.1 Beispiel: Rührei und Pudding

## 2. Parallelverarbeitung

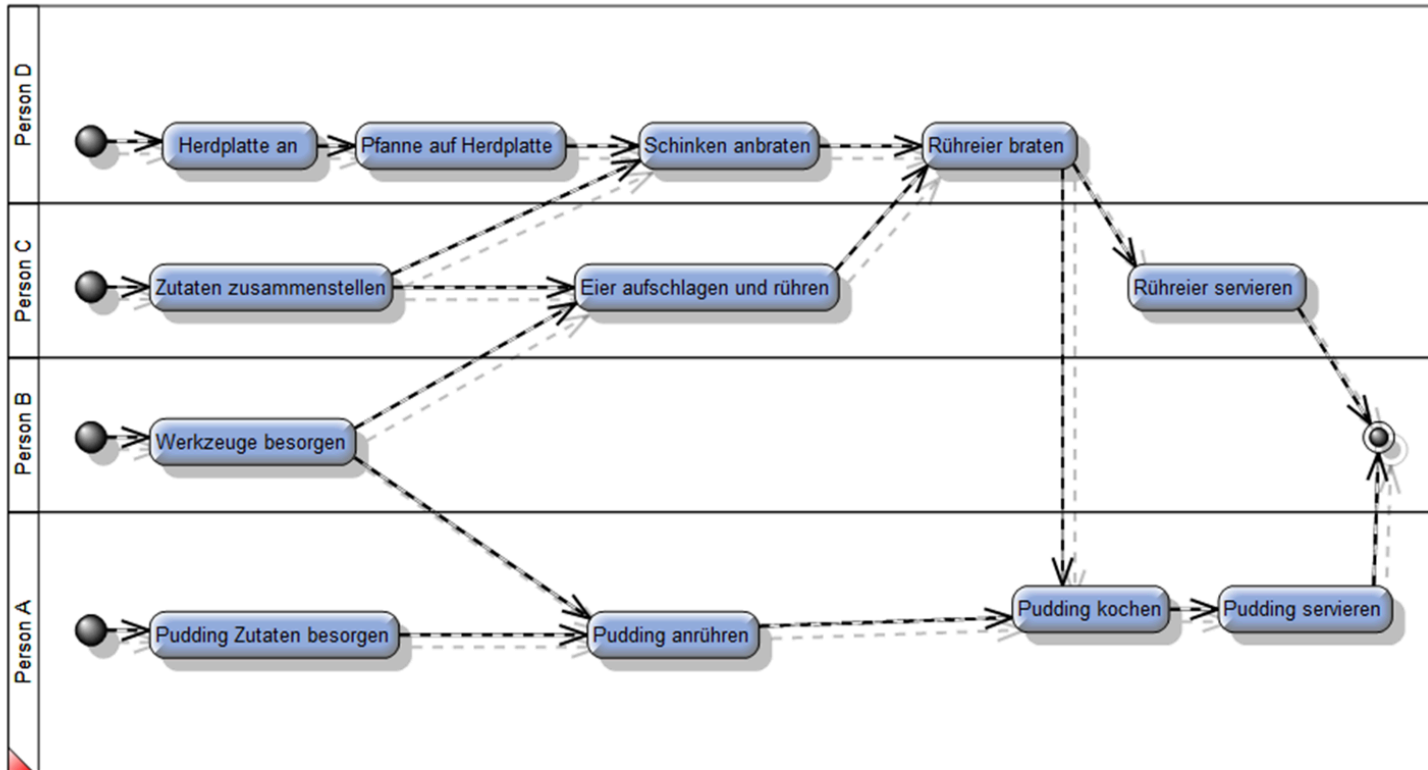
- Mögliche Reihenfolge
- Ressourcenkonflikt: Herdplatte



# 2.1 Beispiel: Rührei und Pudding

## 2. Parallelverarbeitung

- Mögliche Aufteilung auf vier Personen



## 2.1 Beispiel: Rührei und Pudding

## 2. Parallelverarbeitung

- Aufgabe wird in Teilaufgaben zerlegt, die parallel ausgeführt werden können
- Ergebnisse der Teilaufgaben müssen ausgetauscht werden
- Probleme:
  - Abhängigkeiten: Teilaufgaben benötigen Ergebnisse anderer Teilaufgaben
  - Ressourcenkonflikt: Teilaufgaben benötigen dieselbe Ressource
  - Kommunikations-Overhead: Austausch von Ergebnissen benötigt Ressourcen und Zeit
- Aufgaben können nicht beliebig oder automatisch parallelisiert werden.

- Begriffe:
  - Thread (engl. für „Faden“): Ausführungsstrang innerhalb eines Programmes
  - Multithreading: Mehrere (parallele) Ausführungsstränge innerhalb eines Programmes
- Speicher:
  - Threads teilen sich Speicherbereich des Programmes:
  - Teilen sich daher Variablen und Objekte
  - Können effizient (aber unsicher!) über Variablen und Objekte kommunizieren
- Aber: Jeder Thread hat eigenen Aufruf-Stack der aufgerufenen Methoden

### ? Frage

- Kleines Rätsel zwischendurch:
- Zumindest einen parallelen Thread haben wir bereits kennengelernt. Welchen?

- 

-

### ? Frage

- Kleines Rätsel zwischendurch:
  - Zumindest einen parallelen Thread haben wir bereits kennengelernt. Welchen?
- 
- Antwort:
    - Garbage Collector (Speicher nicht referenzierter Objekte freigeben)
  - Beachte:
    - Java-Programme erzeugen beim Start einen main-Thread
    - Setzen hierbei main() als unterste Methode auf den Aufruf-Stack
    - Bei Bedarf zusätzlich ein Thread für den Garbage Collector gestartet
    - Programm beendet, sobald der letzte zugehörige Thread beendet wurde

## 2.1 Beispiel: Rührei und Pudding

## 2. Parallelverarbeitung

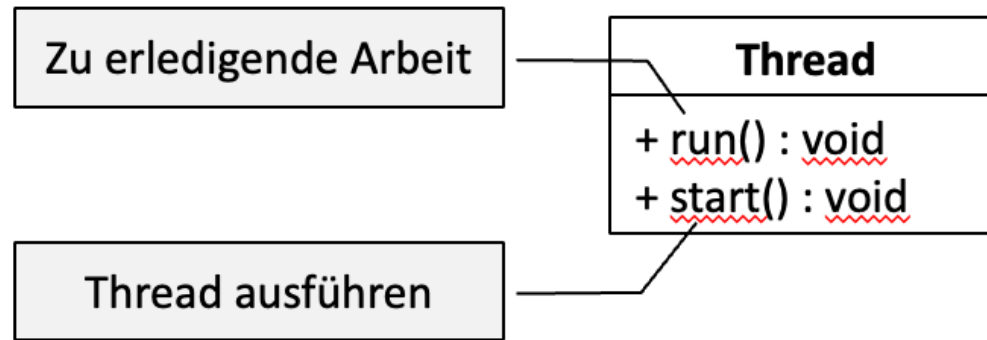
- Teilt Programmen und Threads Rechenzeit (d.h. Prozessoren bzw. Prozessorkerne) zu
- Wartezeiten anderer Threads oder Programme genutzt
- Pseudo-Parallelität:
  - Falls mehr parallele Ausführungsstränge als Prozessoren bzw. Prozessorkerne
  - Scheduler verteilt Rechenzeit scheibchenweise:
    - Ausführung im zeitlichen Wechsel
    - Eindruck, dass Dinge parallel prozessiert werden

# 3. Klassenbasierte Threads

---



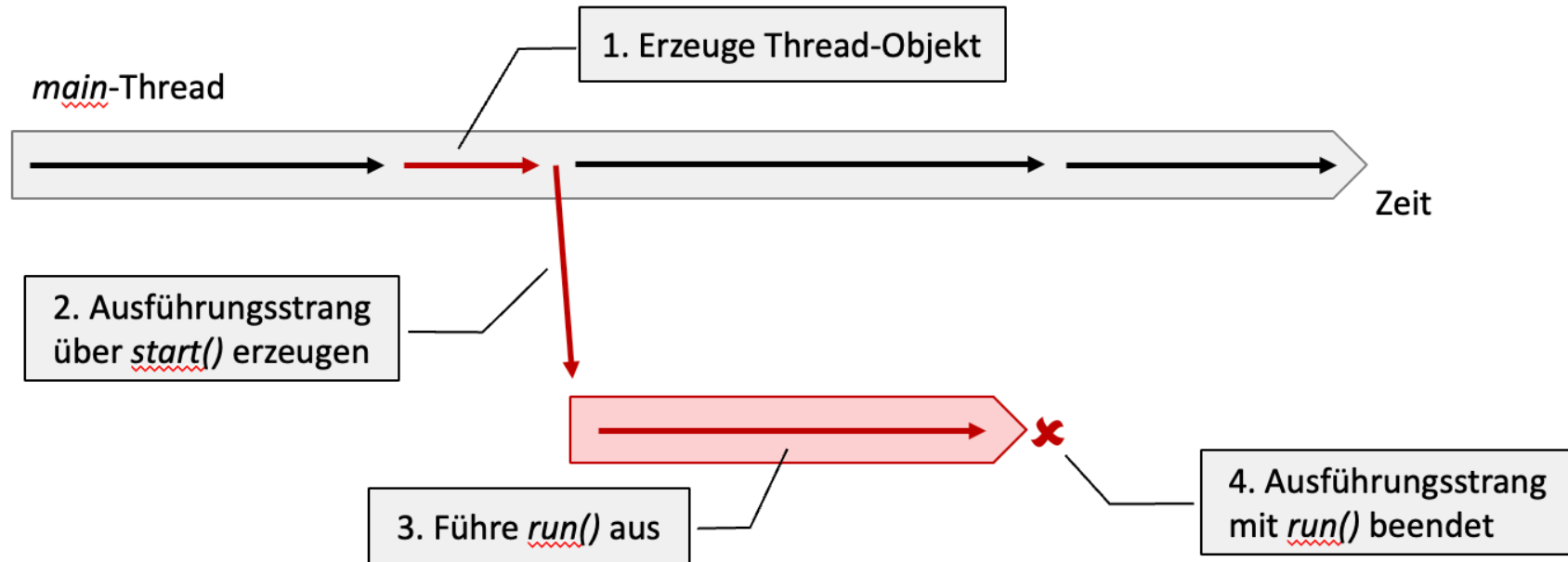
- Threads werden durch Objekte der Klasse Thread erzeugt:
- Methode `start()` erzeugt und startet parallelen Ausführungsstrang
- Methode `run()` enthält Code, der in Ausführungsstrang ausgeführt werden soll
- Ausführungsstrang wird beendet, sobald `run()` beendet wird



# 3.1 Klasse Thread

## 3. Klassenbasierte Threads


- Veranschaulichung



### ☰ Aufgabe 1

- Lassen Sie uns dies implementieren:
- Schreiben Sie ein Programm, das einen zusätzlichen Thread erzeugt.

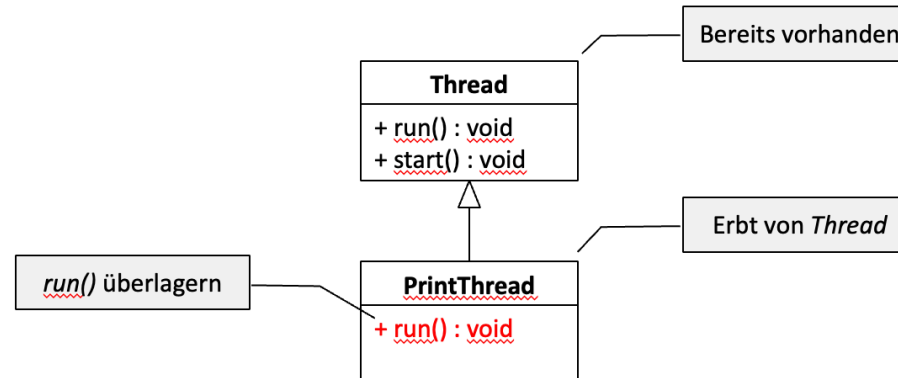
```
1 public class RunThread1 {  
2     public static void main(String[] args) {  
3         Thread thread = new Thread();  
4         System.out.println("Objekt erzeugt");  
5         thread.start();  
6         System.out.println("Thread gestartet");  
7     }  
8 }
```

 Java

### ? Frage

- Aber man sieht ja gar nichts vom Thread!
  - Die run()-Methode der Klasse Thread ist „leer“.
  - Wie können wir den Thread einen Text auf Konsole ausgeben lassen?

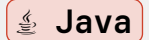
- Ansatz:
  - Die eigentliche Arbeit findet in der `run()`-Methode statt.
  - Die `run()`-Methode der Klasse `Thread` ist „leer“.
- Eigene Thread-Klasse von `Thread` ableiten und `run()` überlagern



### ☰ Aufgabe 2

- Erzeugen Sie in einem zusätzlichen Thread eine Konsolenausgabe.

```
1  public class PrintThread extends Thread {
2      public void run() {
3          System.out.println("Hurra, ich laufe parallel!");
4      }
5  }
6
7  public class RunThread2 {
8      public static void main(String[] args) {
9          PrintThread thread = new PrintThread();
10         System.out.println("Objekt erzeugt");
11         thread.start();
12         System.out.println("Thread gestartet");
13     }
14 }
```

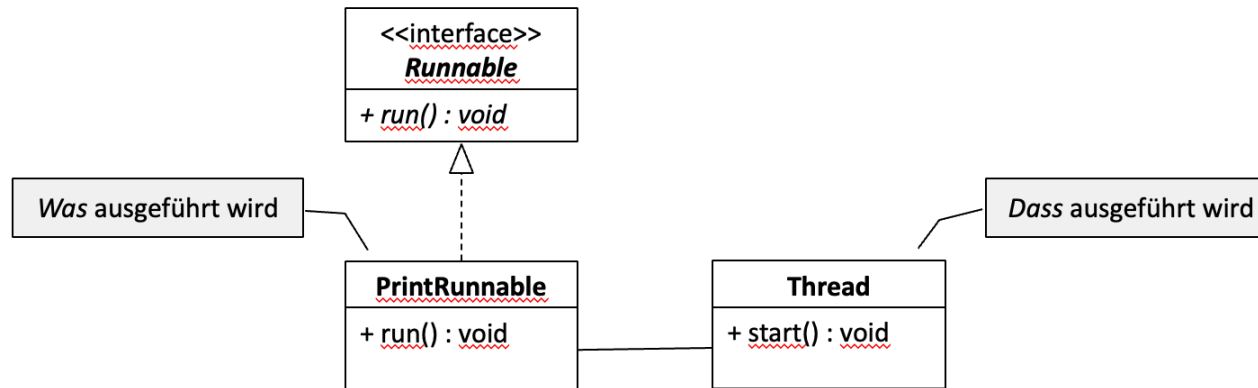


Java

## 4. Interface-basierte Threads

---

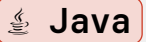
- Alternativ zum Ableiten von Thread:
  - Eigene Klasse implementiert Interface Runnable mit `run()`-Methode
  - Runnable-Objekt wird an Thread-Objekt übergeben: Keine Vererbung erforderlich
- Verantwortlichkeiten:
  - Runnable-Objekt beinhaltet, was ausgeführt werden soll
  - Thread-Objekt beinhaltet alles, was zur Nebenläufigkeit benötigt wird



# 4.1 Interface Runnable

## 4. Interface-basierte Threads

```
1  public class PrintRunnable implements Runnable {
2      public void run() {
3          System.out.println("Hurra, ich laufe parallel!");
4      }
5  }
6
7  public class InterfaceBased {
8      public static void main(String[] args) {
9          PrintRunnable runnable = new PrintRunnable();
10         Thread thread = new Thread(runnable);
11
12         System.out.println("Objekte erzeugt");
13         thread.start();
14         System.out.println("Thread gestartet");
15     }
16 }
```

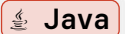




### ? Frage

- Was wird ausgegeben?

```
1  class CounterRunnable implements Runnable {
2      private int counter;
3      public void run() {
4          while (counter < 10)
5              System.out.println("\t\t\tThread counter: " + counter++);
6              System.out.println("\t\t\tExiting run()");
7      }
8  }
9  public class Counters {
10     private static int counter;
11     public static void main(String[] args) {
12         new Thread(new CounterRunnable()).start();
13         while (counter < 10)
14             System.out.println("Main counter: " + counter++);
15         System.out.println("Exiting main()");
16     }
17 }
```



## 4.1 Interface Runnable


- Methoden `run()` und `main()` zählen bis 9
- Nicht vorhersagbar, wer zuerst fertig ist
- Beispielausgabe (rechts):
  - `main`-Thread zuerst beendet
  - Thread mit `run()` läuft weiter

## 4. Interface-basierte Threads

```
Main counter: 0
Main counter: 1
Main counter: 2
Main counter: 3
Main counter: 4
Main counter: 5
Main counter: 6
Main counter: 7
Main counter: 8
Main counter: 9
Exiting main()

Thread counter: 0
Thread counter: 1
Thread counter: 2
Thread counter: 3
Thread counter: 4
Thread counter: 5
Thread counter: 6
Thread counter: 7
Thread counter: 8
Thread counter: 9
Exiting run()
```

`main()` beendet



## 5. Zustände und ausgewählte Methoden

---

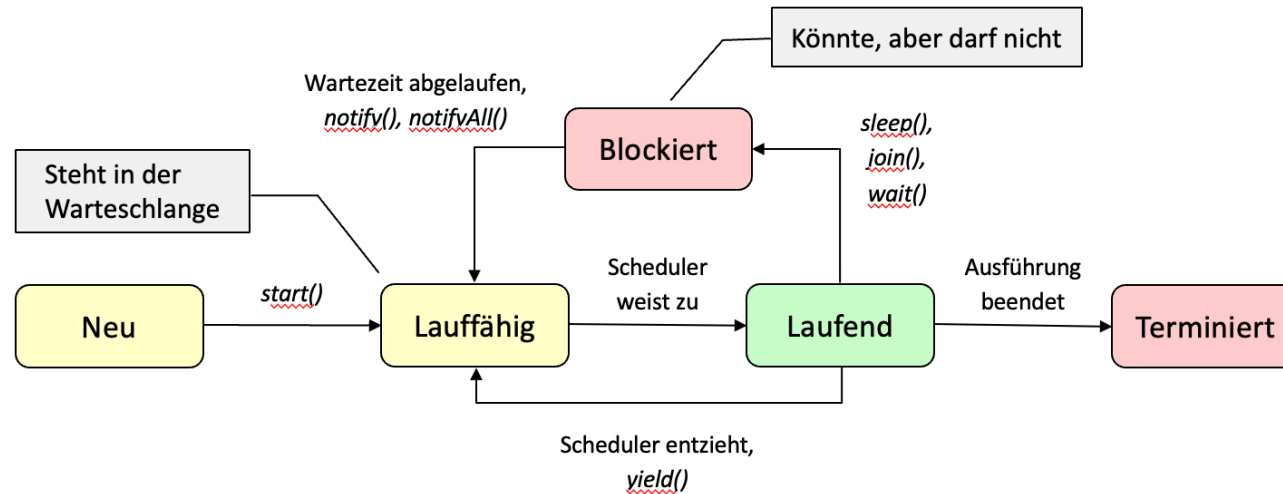
### ? Frage

- Stellen Sie sich vor, Sie wären ein Thread:
  - Welche Zustände könnten Sie sinnvoller Weise einnehmen?
  - Welche Zustandsübergänge wären sinnvoll?
- Vergessen Sie Folgendes nicht:
  - Was passiert, wenn mehr Threads als Prozessoren existieren?
  - Was sollten Sie machen, wenn Sie auf eine Eingabe warten?

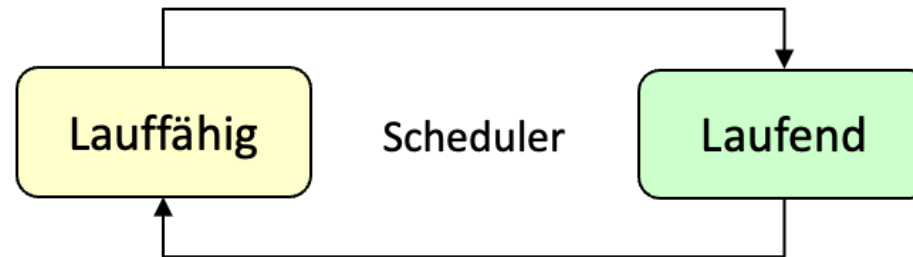
# 5.1 Thread-Zustände

## 5. Zustände und ausgewählte Methoden

- Neu: Java-Objekt erstellt, aber noch nicht als Thread gestartet
- Lauffähig: Bereit, ausgeführt zu werden. Wartet auf Prozessor.
- Laufend: Hat Prozessor und wird gerade ausgeführt
- Blockiert: Wird nicht ausgeführt und würde es auch bei freiem Prozessor nicht
- Terminiert: Thread beendet. Java-Objekt existiert weiterhin!



- Weist Threads Rechenzeit zu (d.h. Lauffähig wird Laufend)
- Entzieht Threads Prozessor wieder (d.h. Laufend wird Lauffähig):
  - Benötigt, falls mehr Threads als Prozessoren existieren
  - Idee: Threads erhalten abwechselnd Rechenzeit
- Steuerung des Verhaltens:
  - Scheduler ist nicht steuerbar
  - Keine Garantie, dass Threads abwechselnd Rechenzeit erhalten
  - `setPriority()` setzt Priorität, aber keine Garantie wie Scheduler sie berücksichtigt
  - „Der Scheduler ist eine Diva!“




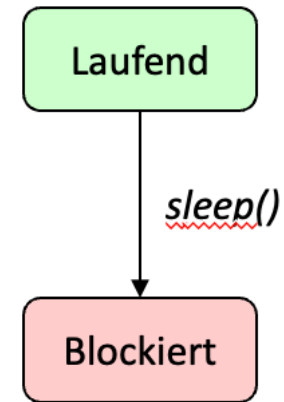
# 5.1 Thread-Zustände

## 5. Zustände und ausgewählte Methoden

- Laufenden Thread eine bestimmte Zeit in Zustand Blockiert versetzen
- Wartezeit in Millisekunden als Parameter (Datentyp long) übergeben
- Vorzeitiges Aufwecken:
  - Thread kann durch Methode interrupt() vorzeitig „aufgeweckt“ werden
  - Wirft hierbei Ausnahme vom Typ InterruptedException

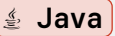
```
1 MyThread thread = new MyThread();
2 thread.start();
3 try {
4     Thread.sleep(1000);
5 } catch (InterruptedException e) {
6     e.printStackTrace();
7 }
```

 **Java**



- Lassen Sie das Fenster blinken (alle 0,75 s Wechsel zwischen gelb und hellgrau):

```
1  public class FlashLight {
2      private boolean isLightOn;
3      private JFrame frame;
4      private FlashLight() {
5          frame = new JFrame("Blinklicht");
6          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7          frame.setSize(300, 250);
8          frame.setVisible(true);
9      }
10     public void switchLight() {
11         isLightOn = !isLightOn;
12         if (isLightOn)
13             frame.getContentPane().setBackground(Color.YELLOW);
14         else
15             frame.getContentPane().setBackground(Color.LIGHT_GRAY);
16     }
17     public static void main(String[] args) {
18         FlashLight flashLight = new FlashLight();
19     }
20 }
```



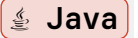


# 5.1 Thread-Zustände

## 5. Zustände und ausgewählte Methoden

- Zum Blinken benötigt:
  - Thread, der alle 0,75 s die Methode switchLight() aufruft

```
1  class FlashThread extends Thread {  
2      private FlashLight flashLight;  
3  
4      public FlashThread(FlashLight flashLight) {  
5          this.flashLight = flashLight;  
6      }  
7  
8      public void run() {  
9          while (true) {  
10             flashLight.switchLight();  
11             try {  
12                 Thread.sleep(750);  
13             } catch (InterruptedException e) {  
14             }  
15         }  
16     }  
17 }
```

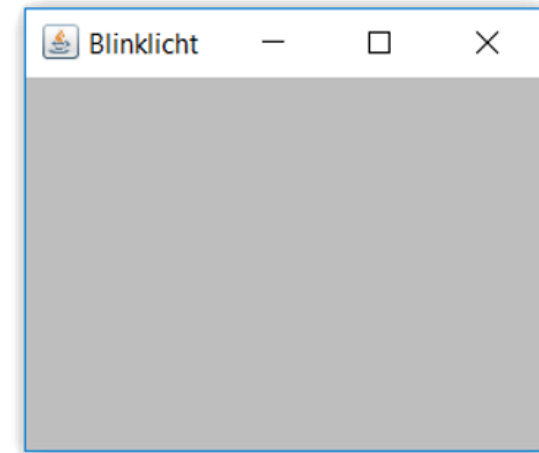
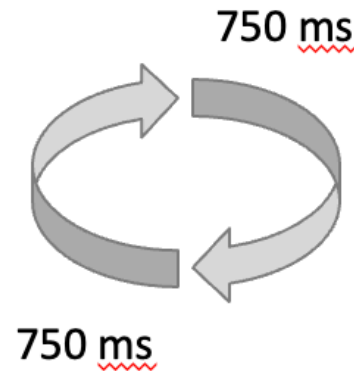
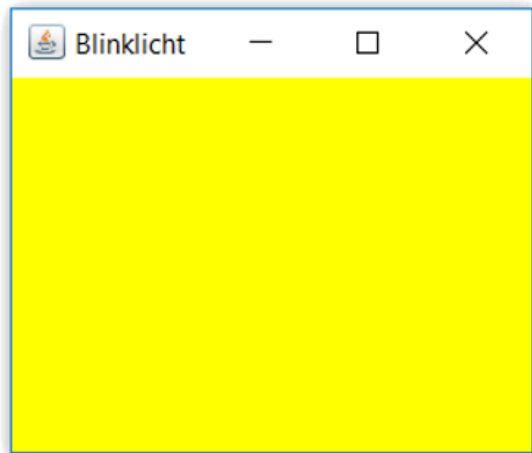
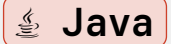


# 5.1 Thread-Zustände

## 5. Zustände und ausgewählte Methoden

- Erzeugung und Starten des Threads in FlashLight:

```
1 public static void main(String[] args) {  
2     FlashLight flashLight = new FlashLight();  
3     FlashThread thread = new FlashThread(flashLight);  
4     thread.start();  
5 }
```



# 5.1 Thread-Zustände

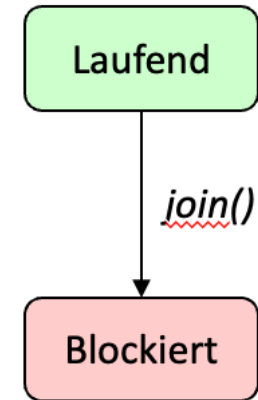
## 5. Zustände und ausgewählte Methoden

- Laufenden Thread auf Ende eines anderen Threads warten lassen
- Beispiel:
  - Wartet bei `thread.join()`, bis thread terminiert
  - Erst dann erfolgt die Konsolenausgabe

```
1 public static void main(String[] args) {  
2     MyThread thread = new MyThread();  
3     thread.start();  
4     thread.join();  
5     System.out.println("We have joined!");  
6 }
```

 Java


- Maximale Wartezeit:
  - Maximale Wartezeit kann als Parameter (Datentyp `long`) angegeben werden
  - Wozu wird dies benötigt? (Man wartet ja schließlich nicht ohne Grund)



### ? Frage

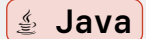
- Was macht dieser Thread?

```
1  public class SleepyThread extends Thread {
2      public void run() {
3          for (int i = 0; i < 5; i++) {
4              System.out.println("I'm sooo tired ...");
5              try {
6                  Thread.sleep(1000);
7              } catch (InterruptedException e) {
8                  e.printStackTrace();
9              }
10         }
11         System.out.println("Okay, I'm awake again.");
12     }
13 }
```

 **Java**

- Es geht noch weiter:
  - Welche Ausgabe wird erzeugt?
  - Welche Ausgabe würde ohne die Zeile `sleepy.join()` erzeugt?
  - Welche Ausgabe würde mit `sleepy.join(1500)` erzeugt?

```
1  public class JoinThreads {
2      public static void main(String[] args) throws InterruptedException {
3          SleepyThread sleepy = new SleepyThread();
4          sleepy.start();
5
6          while (sleepy.isAlive()) {
7              System.out.println("Wake up!");
8              Thread.sleep(400);
9              sleepy.join();
10         }
11         System.out.println("At last ...");
12     }
13 }
```



## 6. Synchronisation

---

- Klasse repräsentiert ein Bankkonto mit Methoden für Ein- und Auszahlungen
- Kontobewegungen parallel über Threads (z.B. Geldautomat, Schalter, Lastschrift)

```
1  public class Account {
2      private double balance;
3
4      public void deposit(double amount) {
5          double newBalance = balance + amount;
6          if (newBalance > balance)
7              balance = newBalance;
8      }
9
10     public void withdraw(double amount) {
11         double newBalance = balance - amount;
12         if (newBalance >= 0.0)
13             balance = newBalance;
14     }
15 }
```



- Was ist denn da passiert?!
  - Sie heben 50 € ab, während 50 € als Überweisung gutgeschrieben werden.
  - Anschließend sind 50 € weniger als zuvor auf dem Konto.

# 6.1 Synchronisation

- Ursache:
  - Threads führen gleichzeitig Methoden `deposit()` und `withdraw()` aus
  - Beide Methoden greifen auf Variable `balance` zu.

Thread 1: <u>deposit()</u>	Betrag	Thread 2: <u>withdraw()</u>	Betrag	<u>balance</u>
<u>newBalance</u> = <u>balance</u> + <u>amount</u> ;	(5050)			5000
		<u>newBalance</u> = <u>balance</u> - <u>amount</u> ;	(4950)	5000
<u>if</u> ( <u>newBalance</u> > <u>balance</u> ) <u>balance</u> = <u>newBalance</u> ;				5050
		<u>if</u> ( <u>newBalance</u> >= 0.0) <u>balance</u> = <u>newBalance</u> ;		4950



# 6.1 Synchronisation

- Zwei Threads teilen sich eine Variable.
  - Race Condition: Ergebnis des Programmes hängt von Zugriffsreihenfolge ab
- Wann ist das Ergebnis davon abhängig, welcher Thread „schneller ist“?
  - Beide Threads lesen die Variable
  - Ein Thread liest, ein Thread schreibt in die Variable
  - Beide Threads schreiben in die Variable
- Antwort:
  - Race Condition, wenn mindestens ein Thread schreibt

Thread 1	Thread 2	<u>Race Condition</u>
Liest	Liest	Keine, beide Threads lesen den gleichen Wert
Liest	Schreibt	Thread 1 kann Wert vor <i>oder</i> nach Änderung durch Thread 2 lesen
Schreibt	Schreibt	Der <i>zuletzt</i> geschriebene Wert bleibt in der Variable

## 6.1 Synchronisation

- Schlüsselwort `synchronized` für Methoden:
  - Objekt wird gesperrt, sobald ein Thread eine synchronisierte Methode betritt
  - Objekt wird wieder freigegeben, wenn Thread die Methode wieder verlässt
- Synchronisierte Methoden (gegenseitiger Ausschluss):
  - Objekt gesperrt: Threads können keine synchronisierten Methoden betreten. (Alle synchronisierten Methoden sind gesperrt, nicht nur die gerade ausgeführte!)
  - Threads warten im Zustand Blockiert bis das Objekt wieder freigegeben wurde.
- Nicht-synchronisierte Methoden:
  - Threads können aber bei gesperrtem Objekt nicht-synchronisierte Methoden betreten.

### ☰ Aufgabe 3

- Helfen Sie Ihrer Bank:
  - Sorgen Sie dafür, dass bei Einzahlungen und Auszahlungen nichts mehr schief geht.

# 6.1 Synchronisation

## 6. Synchronisation

Synchronisierung über synchronized:

```
1  public class Account {
2      private double balance;
3
4      public synchronized void deposit(double amount) {
5          double newBalance = balance + amount;
6          if (newBalance > balance)
7              balance = newBalance;
8      }
9
10     public synchronized void withdraw(double amount) {
11         double newBalance = balance - amount;
12         if (newBalance >= 0.0)
13             balance = newBalance;
14     }
15 }
```



## 7. License Notice

---

## 7.1 Attribution

- This work is shared under the CC BY-NC-SA 4.0 License and the respective Public License
- link(„<https://creativecommons.org/licenses/by-nc-sa/4.0/>“)
- This work is based off of the work Prof. Dr. Marc Hensel.
- Some of the images and texts, as well as the layout were changed.
- The base material was supplied in private, therefore the link to the source cannot be shared with the audience.