

# Software Entwicklung & Programmierung

## Java-Threads

Bilder und Texte der Veranstaltungsfolien und -unterlagen sowie das gesprochene Wort innerhalb der Veranstaltung und Lehr-Lern-Videos dienen allein dem Selbst- bzw. Gruppenstudium. Jede weiterführende Nutzung ist den Teilnehmenden der Moodle-Kurse untersagt, z.B. Verbreitung an andere Studierende, in sozialen Netzwerken, dem Internet!

Darüber hinaus ist ein studentischer Mitschnitt von Webkonferenzen im Rahmen der Lehre nicht erlaubt.

Am Ende dieser Veranstaltung könnt Ihr:

- Die **Kernkonzepte** von **Thread** erläutern
- Mithilfe der Klasse Thread in Java Ihre **eigenen Threads erzeugen**
- Sich in **weitere Literatur** zum Thema Threads **einlesen**

# Agenda


1. **Codeausführung in Java**
2. Threads als Programmierkonzept
3. Threads in Java
4. Locks





- **Source-Code** wird **sequenziell** geschrieben
- Entwickler können sich als Abstraktion vorstellen, dass Source-Code zeilenweise sequenziell ausgeführt wird
- Durch **Kontrollstrukturen** kann die **Reihenfolge** dieser Ausführung beeinflusst werden, nicht aber die sequenzielle Art der Ausführung

```
public static void main(String[] args) {  
    System.out.println("Somebody");  
    System.out.println("once");  
    System.out.println("told me...");  
}
```



# Agenda

1. Codeausführung in Java
2. **Threads als Programmierkonzept**
3. Threads in Java
4. Locks



- Durch Multitasking können Computer-Systeme **mehrere Aufgaben scheinbar gleichzeitig** bearbeiten
- Es gibt verschiedene Konzepte, mit denen Multitasking umgesetzt werden kann, ein einfaches ist das **Time-Sharing**
  - Time-Sharing bedeutet, dass das Computer-System jede einzelne seiner Aufgaben für kurze Zeit bearbeitet, bevor es die Bearbeitung der nächsten Aufgaben fortsetzt
  - Man spricht an dieser Stelle von **Nebenläufigkeit** (engl. concurrency),
  - Beispiel: Ein Computer-System hat die zwei Aufgaben einen Webbrowser und ein Office-Programm auszuführen
    - Das Computer-System führt **alternierend** den Webbrowser für eine Millisekunde aus und anschließend für eine Millisekunde das Office-Programm
    - Es entsteht für den Nutzer die Illusion, dass der Computer beide Programme gleichzeitig ausführt

- Ein Thread ist eine **Sequenz von Anweisungen**
- Betriebssysteme erlauben das Anmelden von Threads
  - Angemeldete Threads werden nach dem beschriebenen Time-Sharing Konzept nebenläufig ausgeführt
  - Die meisten Betriebssysteme führen immer einen Thread für eine kurze Zeitspanne aus und wechseln dann den Thread, der ausgeführt wird
- Die meisten Betriebssysteme erzeugen beim Start eines Programms automatisch einen neuen Thread
  - In **Java** nennt man dies den **main-Thread**, welcher die Anweisungen aus der main-Methode enthält
- Jedes Programm kann eine **beliebige Anzahl Threads** beim Betriebssystem anmelden, dies hat verschiedene Vorteile
  - Beispielsweise können in objektorientierten Programmiersprachen alle Threads eines Programms auf dieselben **Ressourcen** (Objekte im Speicher) zugreifen



# Agenda

1. Codeausführung in Java
2. Threads als Programmierkonzept
3. **Threads in Java**
4. Locks



# Threads in Java

- In Java-Programmen wird jeder Thread durch eine **Instanz der Klasse Thread** repräsentiert
- Zum Starten eines neuen Threads kann die **run-Methode** einer Instanz der Klasse Thread *überschrieben* und dann die **start-Methode** auf dieser Instanz aufgerufen werden
  - In einem neuen Thread werden die Anweisungen der run-Methode nebenläufig ausgeführt
  - Methoden, die aus der run-Methode eines Threads heraus aufgerufen werden, werden auch in diesem Thread ausgeführt

```
Thread myThread = new Thread() {  
    public void run() {  
        System.out.println("Somebody");  
        System.out.println("once");  
        System.out.println("told me...");  
    }  
};
```



Überschreiben der  
run-Methode

```
myThread.start();
```



Starten des Threads



# Beispiel für parallelen Zugriff auf ein Objekt durch mehrere Threads

```
public static void main(String[] args) {  
    LinkedList<Integer> smallList = new LinkedList<>();  
  
    Thread myThread = new Thread() {  
        public void run() {  
            for(Integer item:smallList) {  
                System.out.println(item);  
            }  
        }  
    };  
    myThread.start();  
  
    for(int i=0; i<1000; i++) {  
        smallList.add(i);  
    }  
}
```

## Ausgabe

```
Exception in thread "Thread-0" java.util.ConcurrentModificationException  
    at java.util.LinkedList$ListItr.checkForComodification(LinkedList.java:966)  
    at java.util.LinkedList$ListItr.next(LinkedList.java:888)  
    at ThreadsShowcase$1.run(ThreadsShowcase.java:30)
```

# Beispiel für parallelen Zugriff auf ein Objekt durch mehrere Threads

- Die meisten Datenstrukturen aus dem `java.util.collections` Package versuchen, den nebenläufigen Zugriff durch mehrere Threads zu erkennen und werfen in diesem Fall eine **ConcurrentModificationException**
- Nebenläufige Zugriffe auf eine Datenstruktur können schnell zu **nicht-deterministischen Fehlern** führen, die schwer zu beheben sind
  - Um diese nicht-deterministischen Fehler zu vermeiden bevor sie entstehen, wird die Exception deterministisch geworfen.
- Nicht nur bei nebenläufigen Zugriffen auf Datenstrukturen können nicht-deterministische Fehler auftreten
  - Dies kann auch beim nebenläufigen Arbeiten zweier Threads mit Objekten oder sogar primitiven Datentypen passieren
  - Es ist daher ratsam den Zugriff auf Daten, die sich mehrere Threads teilen (Objekte oder primitive Datentypen) so zu beschränken, dass zu jedem Zeitpunkt nur ein Thread auf diese Daten zugreifen kann



# Agenda

1. Codeausführung in Java
2. Threads als Programmierkonzept
3. Threads in Java
4. **Locks**



- **Locks** können genutzt werden, um Threads aufeinander warten zu lassen
- Threads können Locks *aufnehmen, halten und freigeben*
- Ein Lock kann immer **nur** von **einem** Thread gehalten werden
- Versucht ein Thread ein Lock aufzunehmen, das von einem zweiten Thread gehalten wird, so muss der erste Thread warten bis der zweite Thread das Lock wieder freigibt

# Beispiel Locks

```
public static void main(String[] args) {  
    LinkedList<Integer> smallList = new LinkedList<>();  
    ReentrantLock lock = new ReentrantLock();  
    Thread myThread = new Thread() {  
        public void run() {  
            lock.lock();  
            for(Integer item:smallList) {  
                System.out.println(item);  
            }  
            lock.unlock();  
        }  
    };  
    myThread.start();  
    lock.lock();  
    for(int i=0; i<1000; i++) {  
        smallList.add(i);  
    }  
    lock.unlock();  
}
```

Locks werden in Java  
als Objekte umgesetzt

myThread versucht das Lock  
aufzunehmen

myThread gibt das Lock  
wieder frei

Der main-Thread versucht das  
Lock aufzunehmen

Der main-Thread gibt das Lock  
wieder frei

# Beispiel Locks

```
public static void main(String[] args) {
    LinkedList<Integer> smallList = new LinkedList<>();
    ReentrantLock lock = new ReentrantLock();
    Thread myThread = new Thread() {
        public void run() {
            lock.lock();
            for(Integer item:smallList) {
                System.out.println(item);
            }
            lock.unlock();
        }
    };
    myThread.start();
    lock.lock();
    for(int i=0; i<1000; i++) {
        smallList.add(i);
    }
    lock.unlock();
}
```

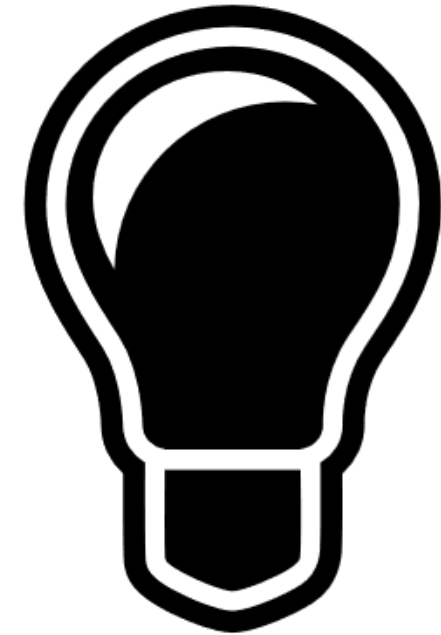
- Ausgabe ist abhängig davon, welcher Thread das Lock zuerst aufnimmt
  - Entweder die Zahlen von 0 bis 999 werden ausgegeben
  - Oder die Liste ist zum Ausgabezeitpunkt leer und deshalb wird nichts ausgegeben
- Eine Concurrent-Modification-Exception kann aber nicht auftreten

# Threads in SEP

- Mögliche **Anwendungsgebiete** von Threads in SEP:
  - Ein Server soll Verbindungen von mehreren Clients annehmen und aufrecht erhalten können
    - Alternative: Bibliotheken und Frameworks für die Erstellung von Server-Anwendungen verwalten meist die Threads für den Anwendungs-Entwickler (bspw.: Spark für HTTP Server)
  - Verteilung von rechenintensiven Aufgaben auf mehrere Threads, um diese schneller abzuarbeiten
    - Alternative: Es können High-Level Schnittstellen der Java-Standardbibliothek genutzt werden, um mehrere Threads automatisch zu verwalten (bspw.: Executors)
- Obwohl es viele Bibliotheken und Frameworks gibt, die einem als Anwendungs-Entwickler die Arbeit mit Threads abnehmen, sollte man sich bewusst sein, dass im Hintergrund dennoch weiterhin Threads verwendet werden.
  - Daher lösen diese Bibliotheken und Frameworks zumeist die typischen Synchronisationsprobleme, die mit Threads einhergehen, nicht automatisch und man muss sich dieser als Entwickler weiterhin bewusst sein



- Ihr solltet nun in der Lage sein...
  - **begründet** zu **entscheiden**, ob und wie in eurem Projekt Threads einsetzen wollt
  - **weitere Literatur** zum Thema Threads und Nebenläufigkeit zu **verstehen**
- Wenn Ihr Threads verwenden wollt, kann euch dieses Tutorial weitere praxisorientierte Informationen geben:
  - <https://docs.oracle.com/javase/tutorial/essential/concurrency/>



# Verwendete Grafiken

- Grafiken von <https://thenounproject.com/>
  - Light Bulb by Alexander Skowalsky

# Vielen Dank für Eure Aufmerksamkeit