

ESOP - Innere Klassen & ~~Threads~~

Assoc. Prof. Dr. Mathias Lux
ITEC / AAU

Quellen / Sources



- Mössenböck (2014) Sprechen Sie Java



- The Java Tutorials:
 - Inner Classes:
<http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html>
 - Threads:
<http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>
- Sierraq & Bates (2005) Head First Java
 - Inner Classes: p.380+
 - Threads: p.490+

Innere Klassen



- Drei Typen von verschachtelten Klassen
 - Innere Klassen
 - Lokale Klassen
 - Anonyme Klassen

```
public class OuterClass {  
    private int number;  
  
    class InnerClass {  
        int innerNumber = 42;  
    }  
}
```

} Innere Klasse

Warum Innere Klassen?



- Gruppierung von Elementen
 - Klassen, die nur an einer Stelle benutzt werden.
- Lesbarer Code
 - Code ist da, wo er benutzt wird.

Warum Innere Klassen?



Erweiterung Geheimnisprinzip

- Ist B eine innere Klasse von A, dann kann B auf *private members* von A zugreifen.
- A muss weniger preisgeben.

```
// with inner class
public class A {
    private int x, y;

    class B {
        int add() {
            return x + y;
        }
    }
}
```



```
// without inner class
public class A {
    protected int x, y;
}

class B {
    A a;
    int add() {
        return a.x + a.y;
    }
}
```



Shadowing / Scope

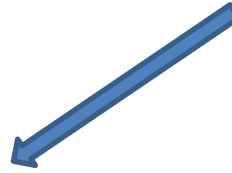


```
public class ShadowTest {
    public int x = 0;
    class FirstLevel {
        public int x = 1;
        void methodInFirstLevel(int x) {
            System.out.println("x = " + x);
            System.out.println("this.x = " + this.x);
            System.out.println("ShadowTest.this.x = " +
                ShadowTest.this.x);
        }
    }
    public static void main(String... args) {
        ShadowTest st = new ShadowTest();
        ShadowTest.FirstLevel fl = st.new FirstLevel();
        fl.methodInFirstLevel(23);
    }
}
```

```
// output:
x = 23
this.x = 1
ShadowTest.this.x = 0
```

- Innere Klassen sind an Instanzen gebunden, nicht Klassen.
- Instanzen innerer Klassen benötigen eine Instanz der äußeren Klasse.

Ausnahme: static nested classes



Lokale Klassen



- Lokale Klassen sind innere Klassen
 - die in einer Methode definiert werden
- Lokale Klassen können auf Variablen und Methoden der äußeren Klasse zugreifen
 - lokale Variablen müssen aber *final* sein
 - ab Java 8 genügt *effectively final*

Lokale Klassen



```
public class Greet {  
    // define an interface  
    interface HelloThere {  
        public void greet(String name);  
    }  
  
    public static void main(String[] args) {  
        class GreetEnglish implements HelloThere {  
            @Override  
            public void greet(String name) {  
                System.out.printf("Hello %s!\n", name);  
            }  
        }  
  
        HelloThere h = new GreetEnglish();  
  
        h.greet("Mathias");  
    }  
}
```

- Interface wird in Klasse definiert.
- Lokale Klasse weil in der Methode.

Anonyme Klassen



- Anonyme Klassen sind lokale Klassen ohne Namen
- Instanziierung und Deklaration fallen zusammen
 - Code wird kompakter.

Anonyme Klassen



```
public class GreetAnon {  
  
    // define an interface  
    interface HelloThere {  
        public void greet(String name);  
    }  
  
    public static void main(String[] args) {  
        HelloThere h = new HelloThere() {  
            @Override  
            public void greet(String name) {  
                System.out.printf("Hello %s!\n", name);  
            }  
        };  
  
        h.greet("Mathias");  
    }  
}
```

- Lokale Klasse anonym
- Oft in Verwendung oft um Interfaces dynamisch zu implementieren

Beispiel KeyListener.



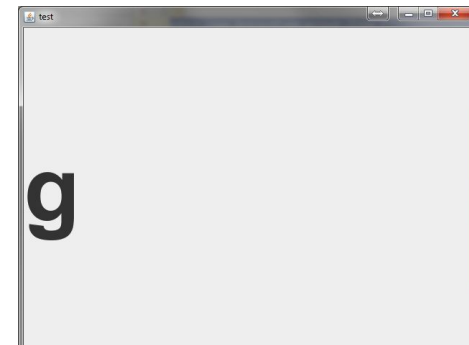
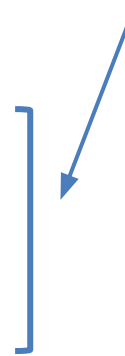
```
public class KeyboardFrame extends JFrame {
    JLabel label = new JLabel();

    public KeyboardFrame() throws HeadlessException {
        super("test");
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setSize(640, 480);
        label.setFont(label.getFont().deriveFont(128f));
        add(label, BorderLayout.CENTER);

        addKeyListener(new KeyAdapter() {
            public void keyReleased(KeyEvent keyEvent) {
                if (!keyEvent.isActionKey())
                    label.setText(Character.toString(
                        keyEvent.getKeyChar()));
            }
        });
    }

    public static void main(String[] args) {
        KeyboardFrame kf = new KeyboardFrame();
        kf.setVisible(true);
    }
}
```

- Listener als Beispiel
 - Behandeln Events
 - implementierungsspezifisch
- Anonyme Klasse für
 - genau diesen Frame
 - genau diese Aktion

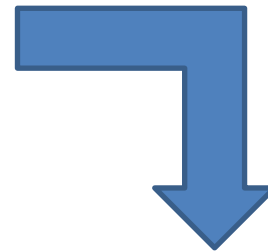


Lambda-Ausdrücke



- Verkürzte Form der anonymen Klassen
 - zur besseren Übersicht
 - weniger Source-Code

```
Collections.sort(myNames, new Comparator<String>() {  
    @Override  
    public int compare(String o1, String o2) {  
        return -o1.compareTo(o2);  
    }  
});
```



```
Collections.sort(myNames, (o1, o2) -> -o1.compareTo(o2));
```

Form von Lambda-Ausdrücken



- Komma-getrennte Liste von Parametern in runden Klammern
 - Klammer optional bei nur einem Parameter
- Pfeil-Token „ \rightarrow “
- Körper (body) als Ausdruck oder Block
 - Wird nur eine Ausdruck angegeben, dann wird um „return“ erweitert

Beispiel



```
myNames.forEach(new Consumer<String>() {  
    @Override  
    public void accept(String s) {  
        System.out.println(s);  
    }  
});
```



```
myNames.forEach(s -> System.out.println(s));
```

Parameter

Pfeil

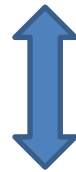
Körper

Lambda-Ausdrücke



- Folgende zwei Ausdrücke liefern dasselbe Ergebnis.

```
Collections.sort(myNames, (o1, o2) -> -o1.compareTo(o2));
```



```
Collections.sort(myNames, (o1, o2) -> { return -o1.compareTo(o2); } );
```

Threads

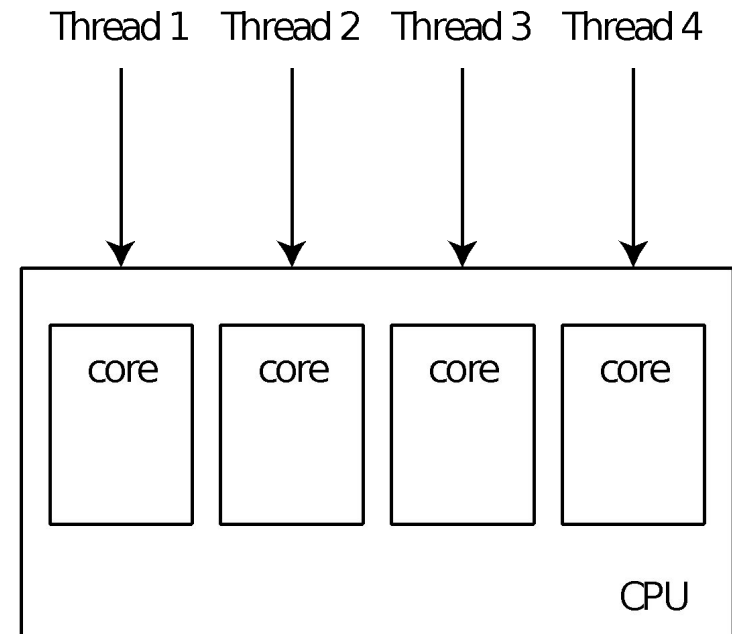
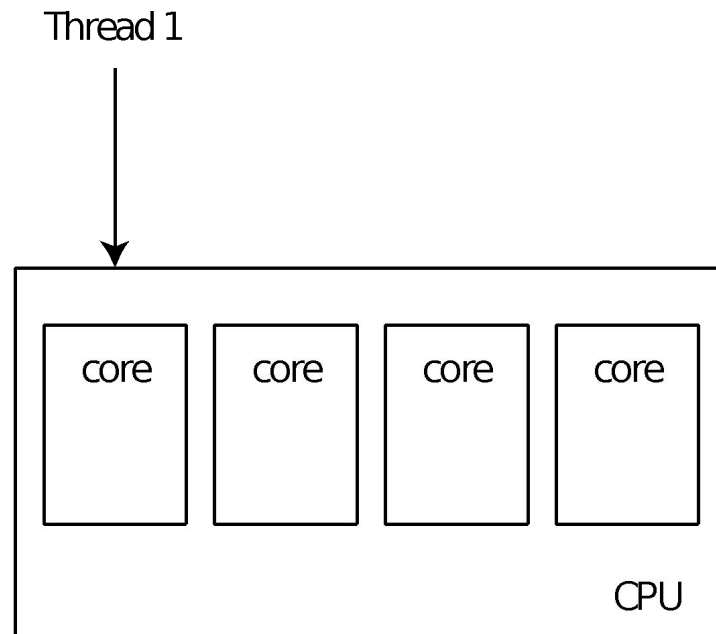


- Prozesse
 - Eigene Umgebung,
 - Stichwort Inter-Process Communication
- Thread
 - Teilen sich eine Umgebung (Variablen, etc.)
 - „light weight processes“

Threads



- Single threaded vs. multi threaded



Beispiele



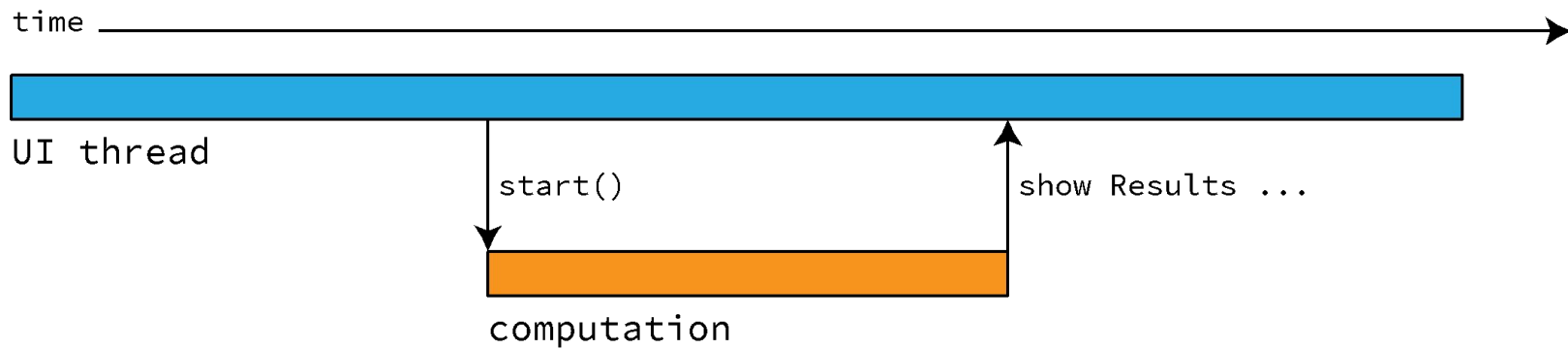
- Datenverarbeitung
 - Es gibt n Datenpakete und k Threads
 - Jeder Thread bearbeitet n/k Datenpakete
- User Interfaces
 - „*Never mess up the UI thread ...*“
- Threads in Computerspielen
 - Zeichnen der Oberfläche
 - Berechnung der KI
 - Netzwerkkommunikation

Threads in Java



- Jeder Thread
 - ist in einem Thread-Objekt gekapselt.
 - wird mit `start()` gestartet.
 - danach wird asynchron die `run()` Methode ausgeführt.

Laufzeit-Diagramm



Threads in Java



- Implementierung des Runnable Interface
 - Nutzung als Thread durch Konstruktor
`new Thread(Runnable)`
- Ableitung der Thread Klasse
 - Überschreiben von `run()`



Beispiel



```
public class ThreadExample implements Runnable {
    String text;
    int count = 5000;

    public ThreadExample(String text) {
        this.text = text;
    }

    public static void main(String[] args) {
        new Thread(new ThreadExample("-")).start();
        new Thread(new ThreadExample("0")).start();
        new Thread(new ThreadExample("/")).start();
        new Thread(new ThreadExample("|")).run();
    }

    @Override
    public void run() {
        while (count-- > 0) System.out.print(text);
    }
}
```

- Vier Threads
- Drei werden mit `start()` gestartet
 - Warum?
- Ergebnis?



Problem: Blockierung



- Im Fall von geteilten Ressourcen
 - Ergebnisliste, Festplatte, Variable, ...
- Fall A: Ressource blockiert
 - Festplatte kann immer nur von einem Thread gelesen werden.
- Fall B: Nicht Thread-Safe
 - Es kann gleichzeitig auf Ressource zugegriffen werden.

Beispiel: Threads



```
public class ThreadConcurrencyExample implements Runnable {
    boolean up;           // increment or decrement
    int count = 1000;     // each runs # times
    static int sum;        // shared variable

    public ThreadConcurrencyExample(boolean up) {
        this.up = up;
    }

    public void run() {
        while (count-- > 0)
            if (up)
                sum++;
            else
                sum--;
    }

    public static void main(String[] args) {
        boolean upMe = true;
        LinkedList<Thread> t = new LinkedList<Thread>();
        for (int i = 0; i < 10; i++) { // creating 10 threads
            Thread thread = new Thread(new ThreadConcurrencyExample(upMe));
            upMe = !upMe;
            thread.start();
            t.add(thread);
        }
        // making sure to wait for them to end:
        for (Iterator<Thread> iterator = t.iterator(); iterator.hasNext(); ) {
            try {
                iterator.next().join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("sum = " + sum);
    }
}
```

- Threads greifen gleichzeitig auf Variable zu
 - $n/2$ Threads inkrementieren
 - $n/2$ Threads dekrementieren
- Wie lautet das Ergebnis?

Threads kontrollieren



- Thread.sleep(ms)
 - Lässt aktuellen Thread *ms* schlafen
- t.join()
 - Stellt aktuellen Thread an Thread t an.

Threads synchronisieren



- Keyword *synchronized*
 - Schützt Code-Teil
 - Auf Basis eines Monitor-Objekts

```
boolean up;           // increment or decrement
int count = 1000;     // each runs # times
static int sum;       // shared variable
static Object monitor = new Object();

public ThreadConcurrencyExample (boolean up) {
    this.up = up;
}

public void run() {
    while (count-- > 0)
        synchronized (monitor) {
            if (up)
                sum++;
            else
                sum--;
        }
}
```

Threads synchronisieren



- Blockierende Objekte
 - Sind thread-safe
 - Blockieren durch interne Mechanismen
- Beispiele
 - Vector, HashTable, StringBuffer
 - BlockingQueue

	<i>Throws exception</i>	<i>Special value</i>	<i>Blocks</i>	<i>Times out</i>
Insert	<u>add(e)</u>	<u>offer(e)</u>	<u>put(e)</u>	<u>offer(e, time, unit)</u>
Remove	<u>remove()</u>	<u>poll()</u>	<u>take()</u>	<u>poll(time, unit)</u>
Examine	<u>element()</u>	<u>peek()</u>	<i>not applicable</i>	<i>not applicable</i>