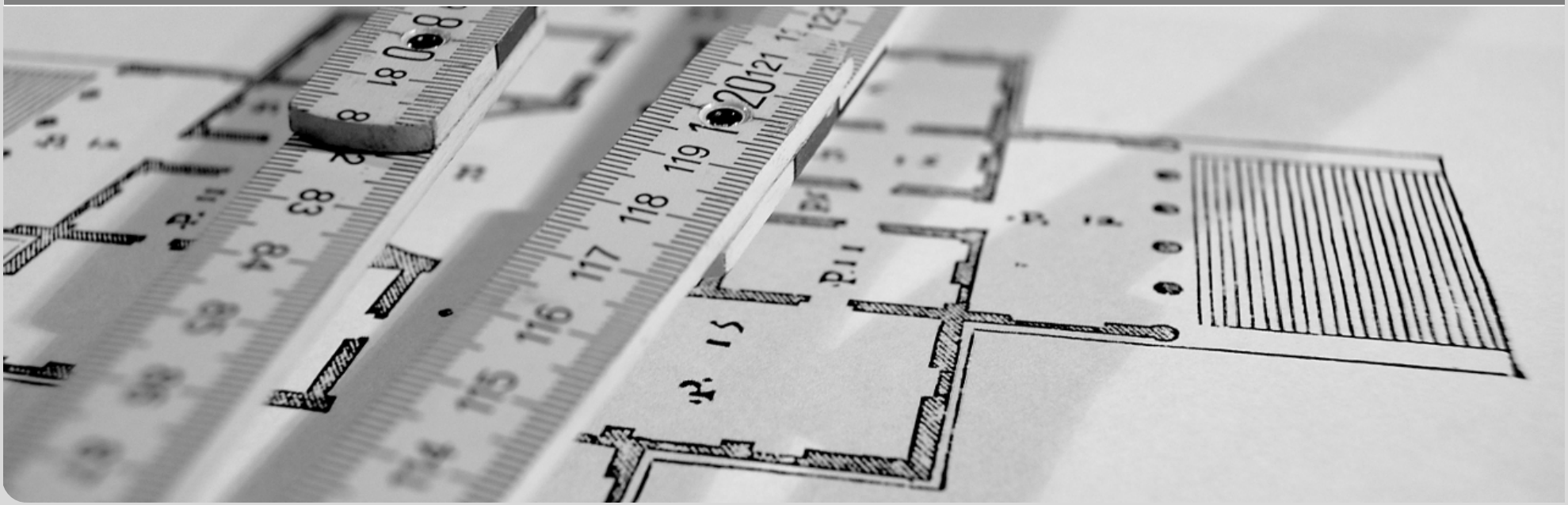
Karlsruher Institut für Technologie

# Softwaretechnik 1 - 5. Tutorium

Tutorium 03

Felix Bachmann | 10.07.2017

KIT - INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION (IPD)



# Allgemeines

## Nächstes Mal letztes Tutorium

- irgendwelche Wünsche für das letzte Tut?
  - etwas bestimmtes wiederholen?
  - falls euch noch was einfällt, schreibt mir eine Mail
    $\Longrightarrow$ felix.bachmann@ewetel.net

# Allgemeines

## Nächstes Mal letztes Tutorium

- irgendwelche Wünsche für das letzte Tut?
  - etwas bestimmtes wiederholen?
  - falls euch noch was einfällt, schreibt mir eine Mail
    $\Longrightarrow$ felix.bachmann@ewetel.net

## Evaluation vom letzten Mal

- nochmal Danke fürs Mitmachen!

# Allgemeines

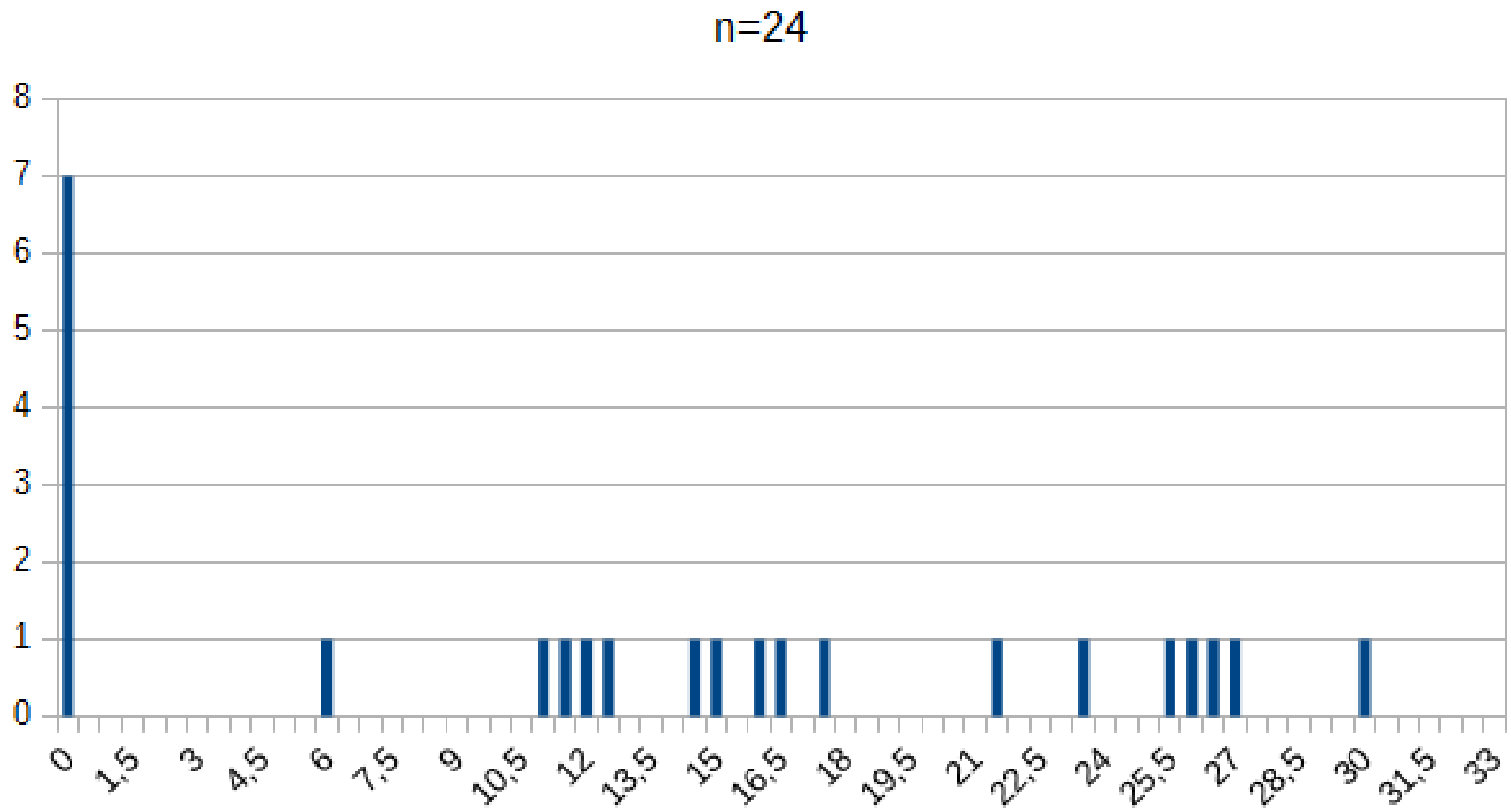
## Nächstes Mal letztes Tutorium

- irgendwelche Wünsche für das letzte Tut?
  - etwas bestimmtes wiederholen?
  - falls euch noch was einfällt, schreibt mir eine Mail
    $\Longrightarrow$ felix.bachmann@ewetel.net

## Evaluation vom letzten Mal

- nochmal Danke fürs Mitmachen!
- häufigster Kritikpunkt: nicht so gut lesbarer Tafelanschrieb
  $\Longrightarrow$ versuche ich besser zu machen :)

# 4. Übungsblatt Statistik




∅ 13 bzw 18,4 von 25+8

# 4. Übungsblatt - Häufige Fehler (A3)

## Aufgabe 3 (GUI für Geometrify): 6,56 bzw. 11,25 von 10+7

```java
// won't work from the jar
File f = new File("src/main/resources/bla.png");

// use one of the following (which one depends on your needs):
this.getClass().getResource("bla.png");
Thread.currentThread().getContextClassLoader().getResource("bla.png");
System.class.getResource("bla.png");
```

# 4. Übungsblatt - Häufige Fehler (A3)

## Aufgabe 3 (GUI für Geometrify): 6,56 bzw. 11,25 von 10+7

```
// won't work from the jar
File f = new File("src/main/resources/bla.png");

// use one of the following (which one depends on your needs):
this.getClass().getResource("bla.png");
Thread.currentThread().getContextClassLoader().getResource("bla.png");
System.class.getResource("bla.png");
```

- keine leeren JPanels o.a. Objekte benutzen, um Platz zwischen Objekten zu erzeugen
  $\Longrightarrow$ geht schöner, performanter mit LayoutManagern

# 4. Übungsblatt - Häufige Fehler (A3)

## Aufgabe 3 (GUI für Geometrify): 6,56 bzw. 11,25 von 10+7

```java
// won't work from the jar
File f = new File("src/main/resources/bla.png");

// use one of the following (which one depends on your needs):
this.getClass().getResource("bla.png");
Thread.currentThread().getContextClassLoader().getResource("bla.png");
System.class.getResource("bla.png");
```

- keine leeren JPanels o.a. Objekte benutzen, um Platz zwischen Objekten zu erzeugen
  $\Longrightarrow$ geht schöner, performanter mit LayoutManagern
- fileChooser.setFileFilter(filter) anstatt fileChooser.addChoosableFileFilter(filter)

mehr zur Aufgabe 3 nach dem Parallelität-Teil!

# 5. Übungsblatt Statistik

# Häufige Fehler

## Allgemein

# Häufige Fehler

## Aufgabe 1 (Architekturstile): ∅ von 5

# Häufige Fehler

## Aufgabe 1 (Architekturstile): ∅ von 5

# Häufige Fehler

## Aufgabe 2 (Iterator für Plug-Ins): ∅ von 6

# Häufige Fehler

## Aufgabe 2 (Iterator für Plug-Ins): ∅ von 6

# Häufige Fehler

## Aufgabe 2 (Iterator für Plug-Ins): ∅ von 6

## Aufgabe 3 (Umstrukturierung von Geometrify): ∅ von 8

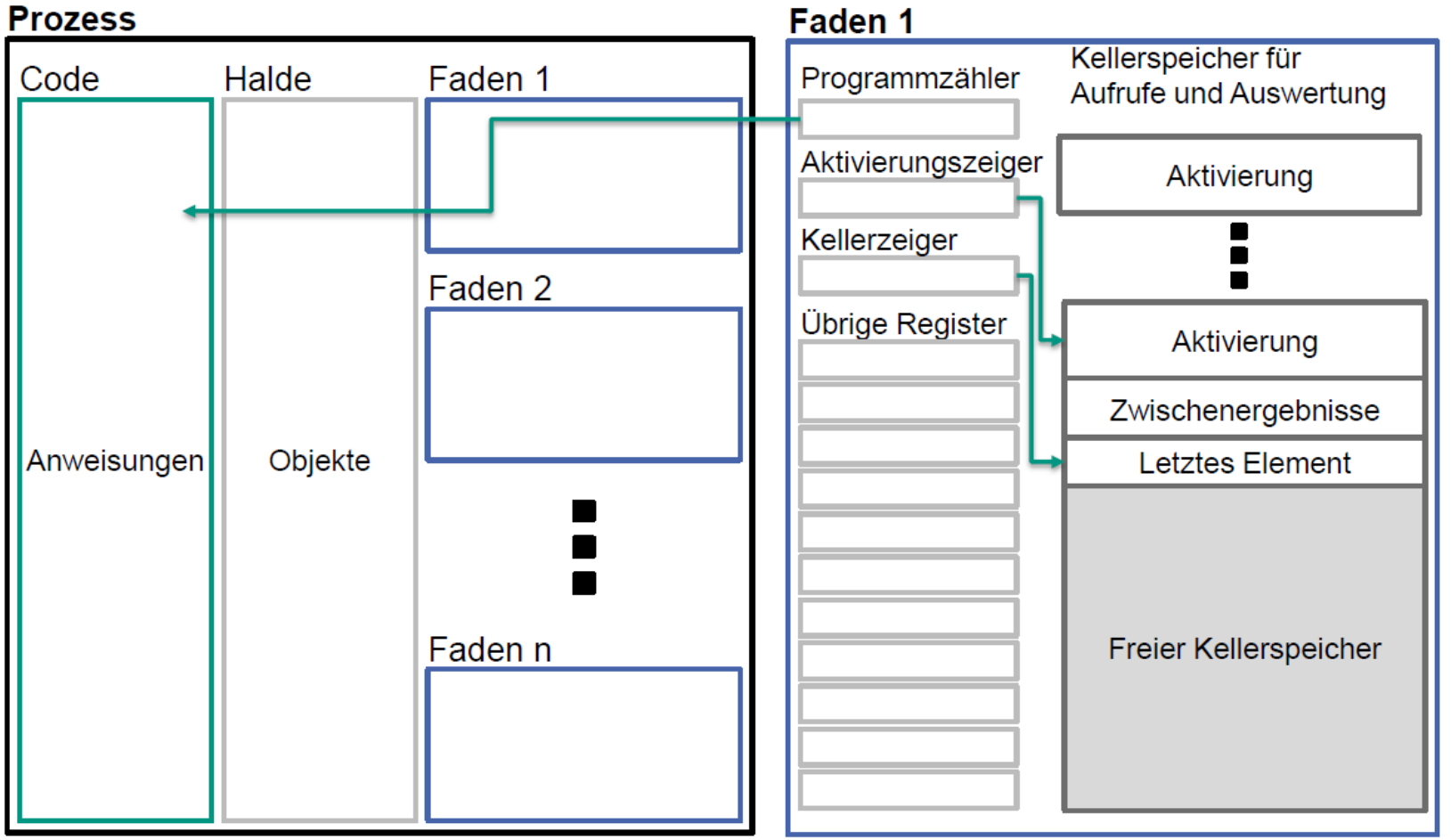
# Häufige Fehler

## Aufgabe 4 (Reimplementierung von Geometrify): ∅ von 7+2

# Häufige Fehler

## Aufgabe 4 (Reimplementierung von Geometrify): ∅ von 7+2

# Parallelität - Grundlagen



# Parallelität - Grundlagen

- Prozess = Programm in Ausführung

# Parallelität - Grundlagen

- Prozess = Programm in Ausführung
- jeder Prozess hat eigenen Adressraum (= Speicherbereich im Arbeitsspeicher)

# Parallelität - Grundlagen

- Prozess = Programm in Ausführung
- jeder Prozess hat eigenen Adressraum (= Speicherbereich im Arbeitsspeicher)
- jeder Prozess hat mindestens einen Thread

# Parallelität - Grundlagen

- Prozess = Programm in Ausführung
- jeder Prozess hat eigenen Adressraum (= Speicherbereich im Arbeitsspeicher)
- jeder Prozess hat mindestens einen Thread
- Threads existieren innerhalb eines Prozesses

# Parallelität - Grundlagen

- Prozess = Programm in Ausführung
- jeder Prozess hat eigenen Adressraum (= Speicherbereich im Arbeitsspeicher)
- jeder Prozess hat mindestens einen Thread
- Threads existieren innerhalb eines Prozesses
  - Threads haben den gleichen Heap und Code

# Parallelität - Grundlagen

- Prozess = Programm in Ausführung
- jeder Prozess hat eigenen Adressraum (= Speicherbereich im Arbeitsspeicher)
- jeder Prozess hat mindestens einen Thread
- Threads existieren innerhalb eines Prozesses
  - Threads haben den gleichen Heap und Code
    $\Longrightarrow$ alle Threads innerhalb eines Prozesses arbeiten mit denselben Objekten und demselben Code

# Parallelität - Grundlagen

- Prozess = Programm in Ausführung
- jeder Prozess hat eigenen Adressraum (= Speicherbereich im Arbeitsspeicher)
- jeder Prozess hat mindestens einen Thread
- Threads existieren innerhalb eines Prozesses
  - Threads haben den gleichen Heap und Code
    $\Longrightarrow$ alle Threads innerhalb eines Prozesses arbeiten mit denselben Objekten und demselben Code
  - Threads haben eigene Stacks und Befehlszeiger (Programmzähler)

# Parallelität - Grundlagen

- Prozess = Programm in Ausführung
- jeder Prozess hat eigenen Adressraum (= Speicherbereich im Arbeitsspeicher)
- jeder Prozess hat mindestens einen Thread
- Threads existieren innerhalb eines Prozesses
  - Threads haben den gleichen Heap und Code
    $\Longrightarrow$ alle Threads innerhalb eines Prozesses arbeiten mit denselben Objekten und demselben Code
  - Threads haben eigene Stacks und Befehlszeiger (Programmzähler)
    $\Longrightarrow$ Threads haben eigene lokale Variablen und können beliebigen Code des Prozesses ausführen

# Parallelität - Motivation

```java
// will freeze the gui when the button is clicked
JButton heavy = new JButton("Freeze");
heavy.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // heavy work
    }
});
```

# Parallelität - Motivation

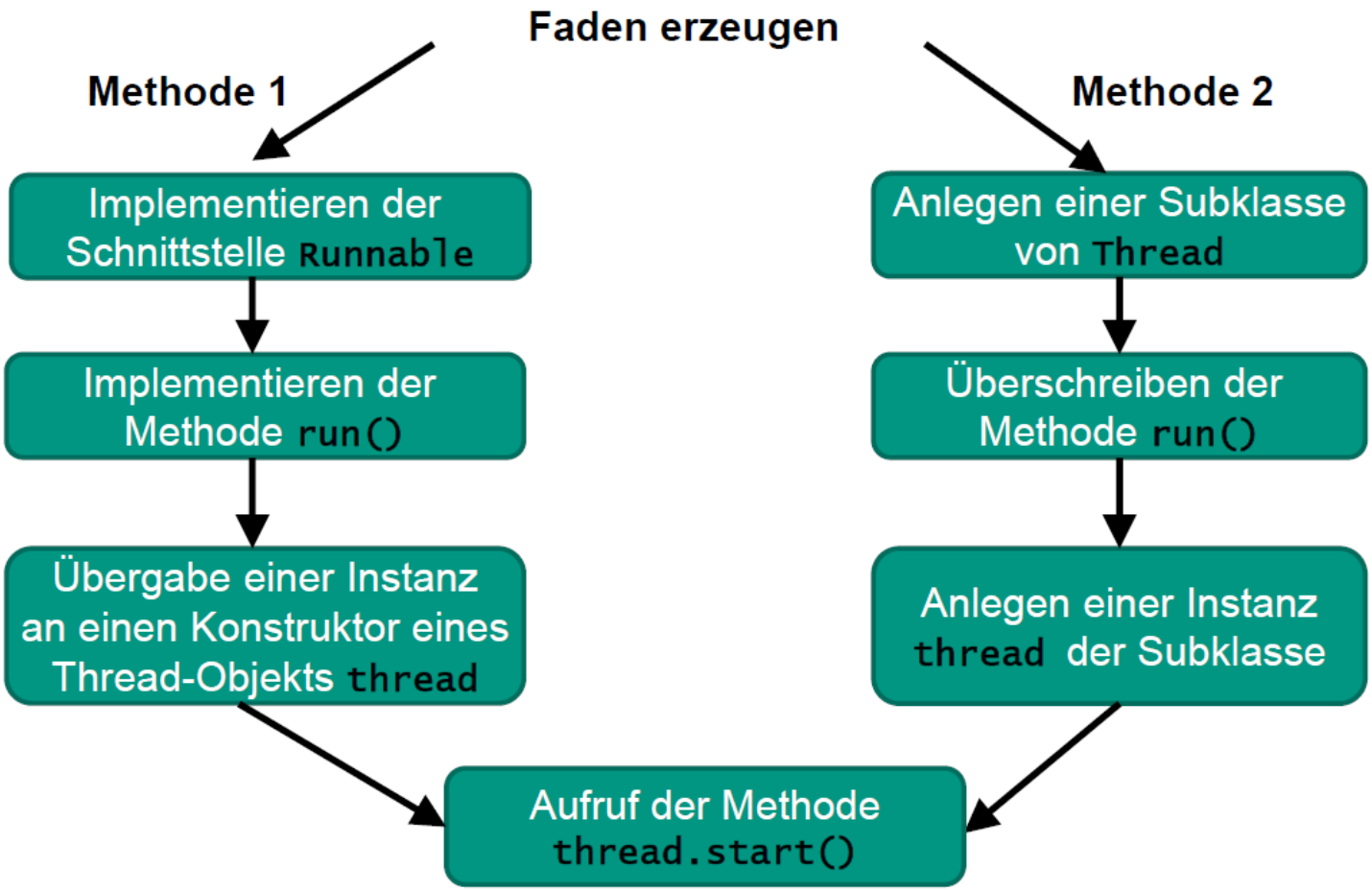
```java
// will freeze the gui when the button is clicked
JButton heavy = new JButton("Freeze");
heavy.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // heavy work
    }
});
```

- “normales“ sequentielles Programm = 1 Prozess mit 1 Thread
- paralleles Programm = 1 Prozess mit mehreren Threads

# Parallelität - in Java

- in Java zwei Möglichkeiten einen Thread zu erstellen
- bereits in Java enthalten:
  - Interface java.lang.Runnable
  - Klasse java.lang.Thread

# Parallelität - in Java



# Parallelität - in Java

```java
// method 2
Thread method2Thread = new Thread() {
    @Override
    public void run() {
        // do stuff
    }
};
method2Thread.start();

// method 1
Thread method1Thread = new Thread(new Runnable() {
    @Override
    public void run() {
        // do stuff
    }
});
method1Thread.start();

method2Thread.join(); // wait until completion
method1Thread.join();
```

# Parallelität - in Java

```java
// method 2
Thread method2Thread = new Thread() {
    @Override
    public void run() {
        // do stuff
    }
};
method2Thread.start();

// method 1
Thread method1Thread = new Thread(new Runnable() {
    @Override
    public void run() {
        // do stuff
    }
});
method1Thread.start();

method2Thread.join(); // wait until completion
method1Thread.join();
```

## Wichtig!

- immer Thread.start() aufrufen, nicht Thread.run()

# Parallelität - in Java

```java
// method 2
Thread method2Thread = new Thread() {
    @Override
    public void run() {
        // do stuff
    }
};
method2Thread.start();

// method 1
Thread method1Thread = new Thread(new Runnable() {
    @Override
    public void run() {
        // do stuff
    }
});
method1Thread.start();

method2Thread.join(); // wait until completion
method1Thread.join();
```

## Wichtig!

- immer Thread.start() aufrufen, nicht Thread.run()
- Thread.run() würde run() sequenziell aufrufen, start() kehrt direkt zurück, nachdem Thread gestartet wurde

# Parallelität - Synchronisation

- Problem: Zugriff auf globale Variablen/ Objekte passiert nicht parallel, Unterbrechungen möglich
- Folge: ggf. falsche Ergebnisse

**Faden 1**

```
                    // globalVar == 1
if (globalVar > 0) {
                                        if (globalVar > 0) {
                                          globalVar--;
                                        }
  globalVar--;
}
```

**Faden 2**

# Parallelität - Synchronisation

- nicht nur ein theoretisches Beispiel!

```
for (int i = 0; i < 100; i++) {
    Thread method2Thread = new Thread() {
        @Override
        public void run() {
            if (x > 0) {
                x--;
            }
        }
    };
    Thread method1Thread = new Thread(new Runnable() {
        @Override
        public void run() {
            if (x > 0) {
                x--;
            }
        }
    });
    method2Thread.start();
    method1Thread.start();
    method2Thread.join(); // wait until completion
    method1Thread.join();
    if (x != 0) {
        System.out.println(x);
    }
    x = 1;
}
```

# Parallelität - Synchronisation

- nicht nur ein theoretisches Beispiel!

```
for (int i = 0; i < 100; i++) {
    Thread method2Thread = new Thread() {
        @Override
        public void run() {
            if (x > 0) {
                x--;
            }
        }
    };
    Thread method1Thread = new Thread(new Runnable() {
        @Override
        public void run() {
            if (x > 0) {
                x--;
            }
        }
    });
    method2Thread.start();
    method1Thread.start();
    method2Thread.join(); // wait until completion
    method1Thread.join();
    if (x != 0) {
        System.out.println(x);
    }
    x = 1;
}
```

```
<terminated> Test (2) [Java Application] C:\Program Files\Java\jdk1.8.0_91\bin\javaw.exe (30.06.2017, 15:50:29
-1
-1
```

# Parallelität - Synchronisation

- Ziel: Zugriff auf gemeinsam genutzte Daten synchronisieren

# Parallelität - Synchronisation

- Ziel: Zugriff auf gemeinsam genutzte Daten synchronisieren
  - kritische Abschnitte schützen

# Parallelität - Synchronisation

- Ziel: Zugriff auf gemeinsam genutzte Daten synchronisieren
  - kritische Abschnitte schützen
  - Wettlaufsituationen vermeiden

**Kritischer Abschnitt (critical section)**

Codeabschnitt, wo Zugriffe auf gemeinsam genutzte Daten stattfindet

# Parallelität - Synchronisation

- Ziel: Zugriff auf gemeinsam genutzte Daten synchronisieren
  - kritische Abschnitte schützen
  - Wettlaufsituationen vermeiden

## Kritischer Abschnitt (critical section)

Codeabschnitt, wo Zugriffe auf gemeinsam genutzte Daten stattfindet

## Wettlaufsituation (race condition)

Verhalten des Programms hängt von der zeitlichen Abfolge der Operationen ab (Wann wird welcher Thread abgebrochen?)

# Parallelität - Synchronisation

- Ziel: Zugriff auf gemeinsam genutzte Daten synchronisieren
  - kritische Abschnitte schützen
  - Wettlaufsituationen vermeiden

## Kritischer Abschnitt (critical section)

Codeabschnitt, wo Zugriffe auf gemeinsam genutzte Daten stattfindet

## Wettlaufsituation (race condition)

Verhalten des Programms hängt von der zeitlichen Abfolge der Operationen ab (Wann wird welcher Thread abgebrochen?)

- Idee: Monitor einführen

# Parallelität - Monitor

- Idee: Monitor einführen

# Parallelität - Monitor

- Idee: Monitor einführen
- Bereich im Code markieren, den nur ein Thread gleichzeitig ausführen kann und dabei nicht unterbrochen werden kann

# Parallelität - Monitor

- Idee: Monitor einführen
- Bereich im Code markieren, den nur ein Thread gleichzeitig ausführen kann und dabei nicht unterbrochen werden kann
- Schlüsselwort in Java synchronized

# Parallelität - Monitor

- Idee: Monitor einführen
- Bereich im Code markieren, den nur ein Thread gleichzeitig ausführen kann und dabei nicht unterbrochen werden kann
- Schlüsselwort in Java synchronized
- es wird immer an einem Objekt synchronisiert, als Argument bei synchronized

# Parallelität - Monitor

- Idee: Monitor einführen
- Bereich im Code markieren, den nur ein Thread gleichzeitig ausführen kann und dabei nicht unterbrochen werden kann
- Schlüsselwort in Java synchronized
- es wird immer an einem Objekt synchronisiert, als Argument bei synchronized
- Thread t kommt an eine mit synchronized(Objekt) markierte Stelle

# Parallelität - Monitor

- Idee: Monitor einführen
- Bereich im Code markieren, den nur ein Thread gleichzeitig ausführen kann und dabei nicht unterbrochen werden kann
- Schlüsselwort in Java synchronized
- es wird immer an einem Objekt synchronisiert, als Argument bei synchronized
- Thread t kommt an eine mit synchronized(Objekt) markierte Stelle
  - es wird geprüft, ob der Monitor Objekt gerade frei ist

# Parallelität - Monitor

- Idee: Monitor einführen
- Bereich im Code markieren, den nur ein Thread gleichzeitig ausführen kann und dabei nicht unterbrochen werden kann
- Schlüsselwort in Java synchronized
- es wird immer an einem Objekt synchronisiert, als Argument bei synchronized
- Thread t kommt an eine mit synchronized(Objekt) markierte Stelle
  - es wird geprüft, ob der Monitor Objekt gerade frei ist
  - ist der Monitor frei, kommt t in den kritischen Abschnitt und der Monitor ist besetzt, bis t den Abschnitt wieder verlässt

# Parallelität - Monitor

- Idee: Monitor einführen
- Bereich im Code markieren, den nur ein Thread gleichzeitig ausführen kann und dabei nicht unterbrochen werden kann
- Schlüsselwort in Java synchronized
- es wird immer an einem Objekt synchronisiert, als Argument bei synchronized
- Thread t kommt an eine mit synchronized(Objekt) markierte Stelle
  - es wird geprüft, ob der Monitor Objekt gerade frei ist
  - ist der Monitor frei, kommt t in den kritischen Abschnitt und der Monitor ist besetzt, bis t den Abschnitt wieder verlässt
  - ist der Monitor besetzt, wird t blockiert, bis der kritische Abschnitt frei ist

# Parallelität - Monitor

private Object o = new Object();

```
 9         for (int i = 0; i < 100000; i++) {
10             Thread method2Thread = new Thread() {
11                 @Override
12                 public void run() {
13                     synchronized (o) {
14                         if (x > 0) {
15                             x--;
16                         }
17                     }
18                 }
19             };
20             Thread method1Thread = new Thread(new Runnable() {
21                 @Override
22                 public void run() {
23                     synchronized (o) {
24                         if (x > 0) {
25                             x--;
26                         }
27                     }
28                 }
29             });
30             method2Thread.start();
31             method1Thread.start();
32             method2Thread.join(); // wait until completion
33             method1Thread.join();
34             if (x != 0) {
35                 System.out.println(x);
36             }
37             x = 1;
38         }
```

# Parallelität - Monitor

private Object o = new Object();

```
 9         for (int i = 0; i < 100000; i++) {
10             Thread method2Thread = new Thread() {
11                 @Override
12                 public void run() {
13                     synchronized (o) {
14                         if (x > 0) {
15                             x--;
16                         }
17                     }
18                 }
19             };
20             Thread method1Thread = new Thread(new Runnable() {
21                 @Override
22                 public void run() {
23                     synchronized (o) {
24                         if (x > 0) {
25                             x--;
26                         }
27                     }
28                 }
29             });
30             method2Thread.start();
31             method1Thread.start();
32             method2Thread.join(); // wait until completion
33             method1Thread.join();
34             if (x != 0) {
35                 System.out.println(x);
36             }
37             x = 1;
38         }
```

<terminated> Test (2) [Java Application] C:\Program Files\Java\jdk1.8.0_91\bin\javaw.

# Parallelität - wait() und notify()

synchronized an Methoden = synchronized(this){Methoden-Rumpf}

# Parallelität - wait() und notify()

synchronized an Methoden = synchronized(this){Methoden-Rumpf}

```
synchronized void produce(Product p) {
    while(buffer.isFull()) {
        //Tue nichts
    }
    buffer.add(p);
}
```

**Thread 1**

```
synchronized void consume() {
    while(buffer.isEmpty()) {
        //Tue nichts
    }
    buffer.remove();
}
```

**Thread 2**

## Probleme

- “busy waiting“ verschwendet Rechenzeit

# Parallelität - wait() und notify()

synchronized an Methoden = synchronized(this){Methoden-Rumpf}

```
synchronized void produce(Product p) {
    while(buffer.isFull()) {
        //Tue nichts
    }
    buffer.add(p);
}
```

**Thread 1**

```
synchronized void consume() {
    while(buffer.isEmpty()) {
        //Tue nichts
    }
    buffer.remove();
}
```

**Thread 2**

## Probleme

- “busy waiting“ verschwendet Rechenzeit
- wartender Produzent blockiert Konsument, der dann nichts konsumieren kann

# Parallelität - wait() und notify()

- Idee: brauchen Mechanismus, der es erlaubt den Monitor freizugeben, während man auf etwas wartet

# Parallelität - wait() und notify()

- Idee: brauchen Mechanismus, der es erlaubt den Monitor freizugeben, während man auf etwas wartet
- dazu braucht man natürlich auch einen Mechanismus, der es erlaubt wartende Threads aufzuwecken

# Parallelität - wait() und notify()

- Idee: brauchen Mechanismus, der es erlaubt den Monitor freizugeben, während man auf etwas wartet
- dazu braucht man natürlich auch einen Mechanismus, der es erlaubt wartende Threads aufzuwecken
- in Java: wait() und notify() bzw. notifyAll()

# Parallelität - wait() und notify()

- Idee: brauchen Mechanismus, der es erlaubt den Monitor freizugeben, während man auf etwas wartet
- dazu braucht man natürlich auch einen Mechanismus, der es erlaubt wartende Threads aufzuwecken
- in Java: wait() und notify() bzw. notifyAll()

**Thread 1**

```
synchronized void produce(Product p) {
    while(buffer.isFull()) {
        this.wait();
    }
    buffer.add(p);
    this.notifyAll();
}
```

**Thread 2**

```
synchronized void consume() {
    while(buffer.isEmpty()) {
        this.wait();
    }
    buffer.remove();
    this.notifyAll();
}
```

# Parallelität - wait() und notify()

```
synchronized void produce(Product p) {
    while(buffer.isFull()) {
        this.wait();
    }
    buffer.add(p);
    this.notifyAll();
}
```

**Thread 1**

```
synchronized void consume() {
    while(buffer.isEmpty()) {
        this.wait();
    }
    buffer.remove();
    this.notifyAll();
}
```

**Thread 2**

- Kann man die while-Schleifen jetzt nicht durch eine if-Abfrage ersetzen?

# Parallelität - wait() und notify()

Karlsruher Institut für Technologie

```
synchronized void produce(Product p) {
    while(buffer.isFull()) {
        this.wait();
    }
    buffer.add(p);
    this.notifyAll();
}
```

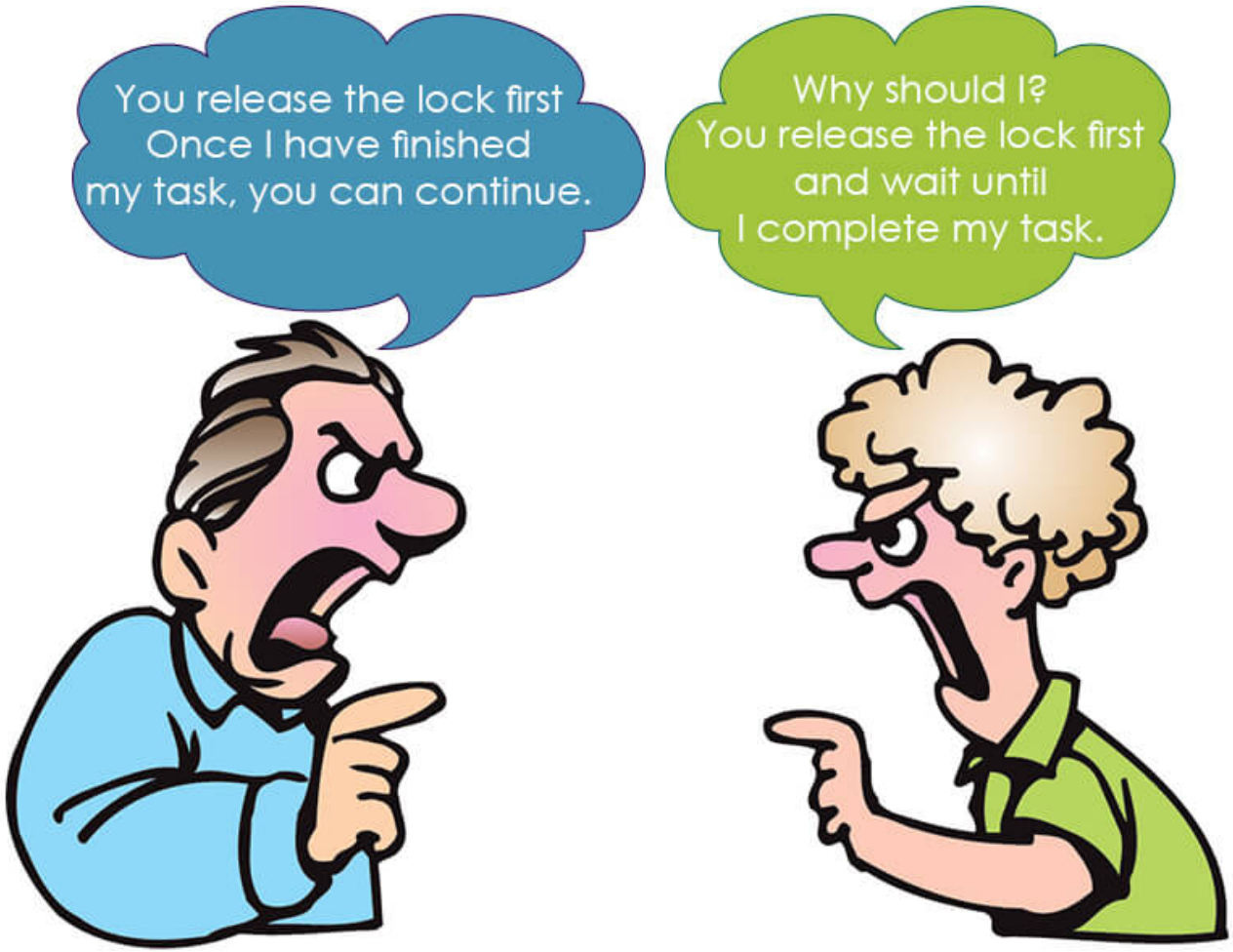
**Thread 1**

```
synchronized void consume() {
    while(buffer.isEmpty()) {
        this.wait();
    }
    buffer.remove();
    this.notifyAll();
}
```

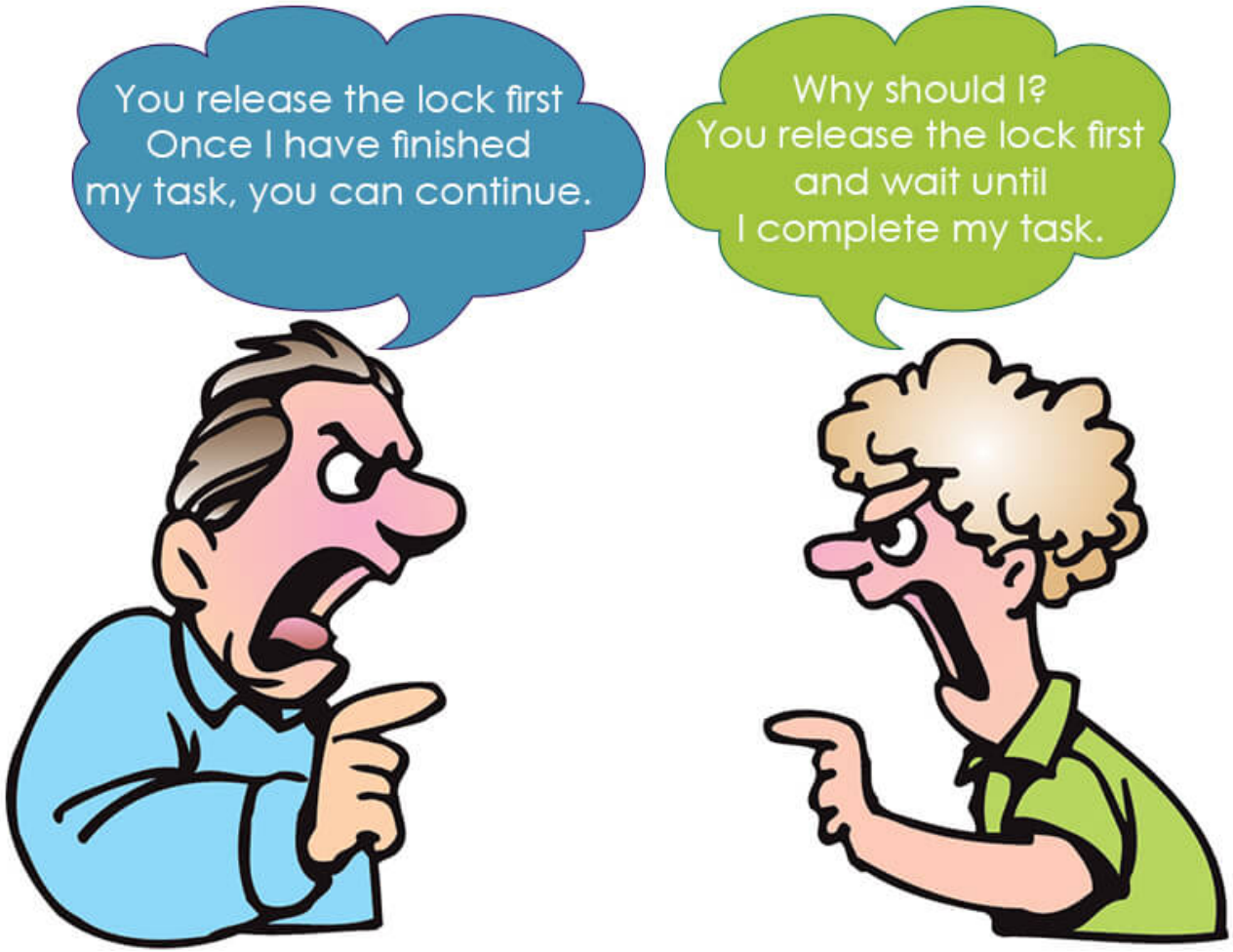
**Thread 2**

- Kann man die while-Schleifen jetzt nicht durch eine if-Abfrage ersetzen?
  - Nein, dann würde nach dem Aufwecken nicht nochmal geprüft werden, ob die Bedingung mittlerweile falsch ist.

# Parallelität - Verklemmung (deadlock)



# Parallelität - Verklemmung (deadlock)




- Thread A hält Monitor B und benötigt Monitor C
- Thread B hält Monitor C und benötigt Monitor B

# Parallelität - Verklemmung (deadlock)

Karlsruher Institut für Technologie

**Thread 1:**

```
synchronized(Papier) {
    synchronized(Stift) {
        maleMandala();
    }
}
```

**Thread 2:**

```
synchronized(Stift) {
    synchronized(Papier) {
        maleMandala();
    }
}
```

klassicher Deadlock!

# Parallelität - Verklemmung (deadlock)

Lösungsansatz: Monitore immer in gleicher Reihenfolge anfordern

**Thread 1:**

```
synchronized(Papier) {
    synchronized(Stift) {
        maleMandala();
    }
}
```

**Thread 2:**

```
synchronized(Papier) {
    synchronized(Stift) {
        maleMandala();
    }
}
```

# 4. Übungsblatt - A3 mit Threads

## Aufgabe 3 (GUI für Geometrify): 6,56 bzw. 11,25 von 10+7

```
public Test() {
    // calling Swing methods from arbitrary threads may result in unexpected behaviour
    // because most Swing Components are not thread safe!
    new JFrame("HelloWorld").setVisible(true);

    // instead use the swing event dispatch thread every time you paint, build,... Swing components
    SwingUtilities.invokeLater(new Runnable() {
        @Override
        public void run() {
            new JFrame("HelloWorld").setVisible(true);
        }
    });

}
```

siehe auch: `https://docs.oracle.com/javase/tutorial/uiswing/concurrency/dispatch.html`

# 4. Übungsblatt - A3 mit Threads

## Aufgabe 3 (GUI für Geometrify): 6,56 bzw. 11,25 von 10+7

```java
// will freeze the gui when the button is clicked
JButton heavy = new JButton("Freeze");
heavy.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // heavy work
    }
});

// will not freeze the gui
JButton light = new JButton("Don't freeze");
light.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // use a new thread to handle heavy work
        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                // heavy work
            }
        });
        // starts the thread, "main thread" will return immediately
        t.start();
    }
});
```

# Tipps - 6. Übungsblatt

## Aufgabe 1: Kontrollfluss-orientiertes Testen

- Zwischensprache benutzen
- Definitionen der verschiedenen Abdeckungen anschauen

# Tipps - 6. Übungsblatt

## Aufgabe 1: Kontrollfluss-orientiertes Testen

- Zwischensprache benutzen
- Definitionen der verschiedenen Abdeckungen anschauen

## Aufgabe 2: Codeinspektion

- an das Format halten

# Tipps - 6. Übungsblatt

## Aufgabe 3: Parallelisierung von Geometrify

- Berechnung der Samples parallelisieren
- Zahl der benutzten Threads abhängig machen von der Anzahl der Prozessor-Kerne

# Tipps - 6. Übungsblatt

## Aufgabe 3: Parallelisierung von Geometrify

- Berechnung der Samples parallelisieren
- Zahl der benutzten Threads abhängig machen von der Anzahl der Prozessor-Kerne

## Aufgabe 4: Alternative Parallelisierungsverfahren

- theoretische Überlegungen, was man sonst noch so parallelisieren könnte
- Sinnhaftigkeit, Aufwand, etc. prüfen

# Tipps - 6. Übungsblatt

## Aufgabe 5: Parallelisierungswettbewerb

- Aufgabe 3 verbessern und Laufzeit messen

# Denkt dran!

## Abgabe

- Deadline am 19.7. um 12:00
- Aufgabe 1,2,4 und Beschreibung, Laufzeitprofil von Aufgabe 5 handschriftlich

# Bis dann! (dann := 24.07.17)