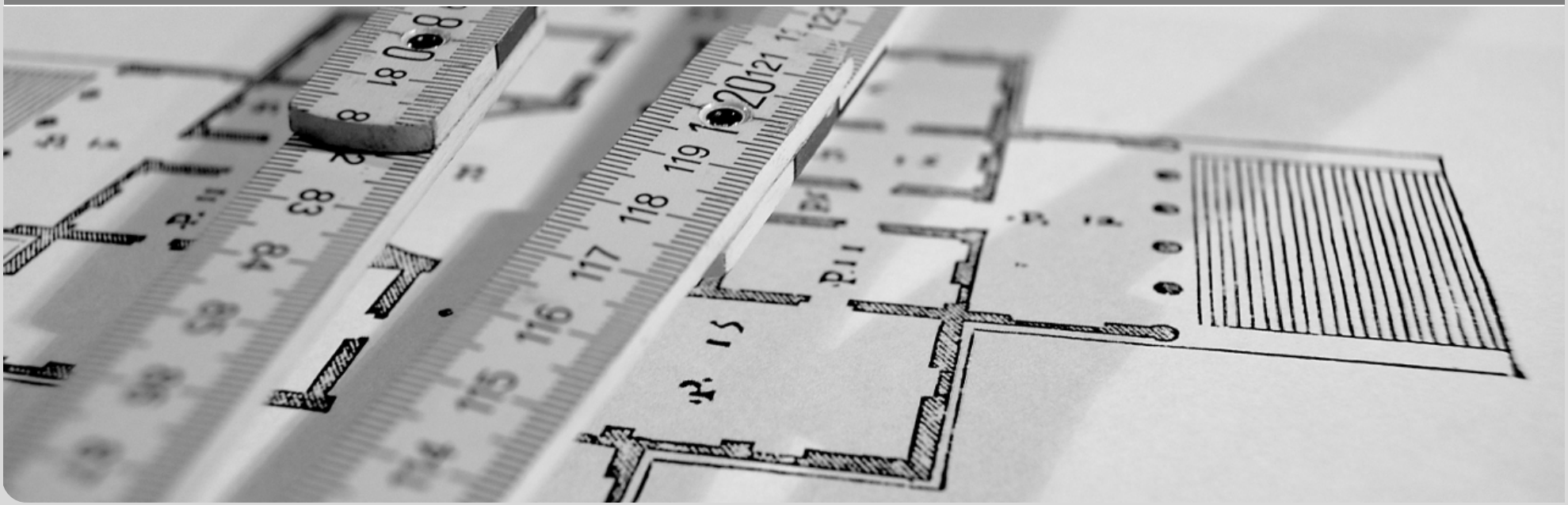
Karlsruher Institut für Technologie

Softwaretechnik 1 - 5. Tutorium

Tutorium 17

Felix Bachmann | 09.07.2019

KIT - INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION (IPD)



Allgemeines

Nächstes Mal ist schon letztes Tutorium

- geplant: grobe Wiederholung + viele Klausuraufgaben

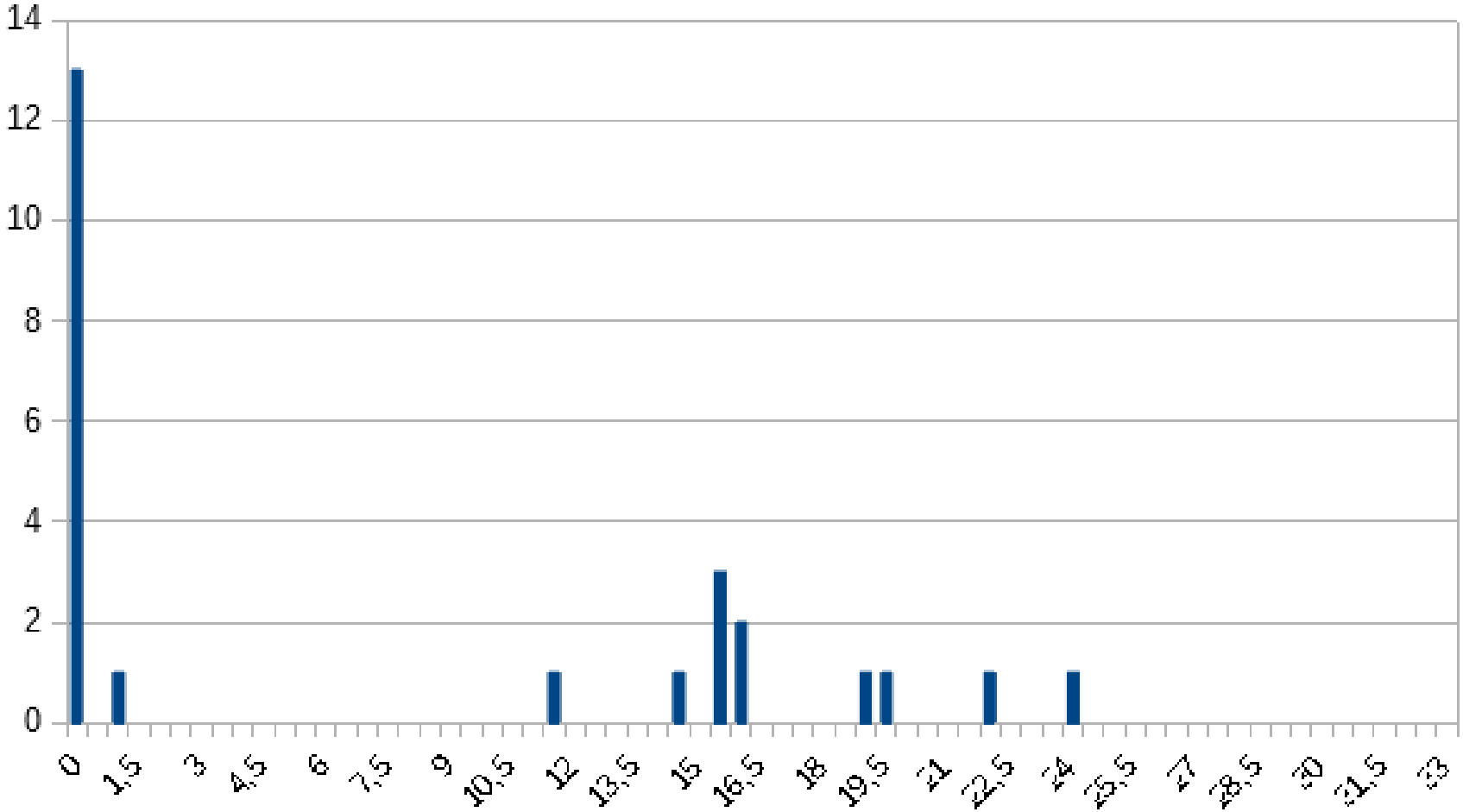
Allgemeines

Nächstes Mal ist schon letztes Tutorium

- geplant: grobe Wiederholung + viele Klausuraufgaben
- ansonsten irgendwelche Wünsche?
  - etwas bestimmtes wiederholen?
  - nochmal irgendwas erklären?
  - Beispiel für XY?
  - falls euch noch was einfällt, schreibt mir eine Mail
    $\Longrightarrow$ felix.bachmann@ewetel.net

Zusatztermin Entwurfmuster

5. Übungsblatt Statistik



Häufige Fehler

Allgemein

- (mal wieder. . .) CheckStyle und JavaDoc
- zu späte Abgabe = nur Korrektur, keine Punkte

Häufige Fehler

Aufgabe 1: Architekturstile

Häufige Fehler

Aufgabe 2: Entwurfsmuster `CacheProvider`

Häufige Fehler

Aufgabe 2: Entwurfsmuster `CacheProvider`

Aufgabe 3: Entwurfsmuster impl. (Iterator, Besucher)

Häufige Fehler

Aufgabe 4: Code → Klassendiagramm

Häufige Fehler

Aufgabe 4: Code → Klassendiagramm

Aufgabe 5: Zustandsmuster(Kaffeemaschine) impl.

- abstrakte Oberklasse/Interface Kaffeemaschine fehlte
  - für Hinzufügen neuer Kaffeemaschine sinnvoll

Häufige Fehler

Aufgabe 6: `git merge` vs `git rebase`

Jetzt letztes Übungsblatt..

Falls ihr noch Punkte braucht, gebt ab!

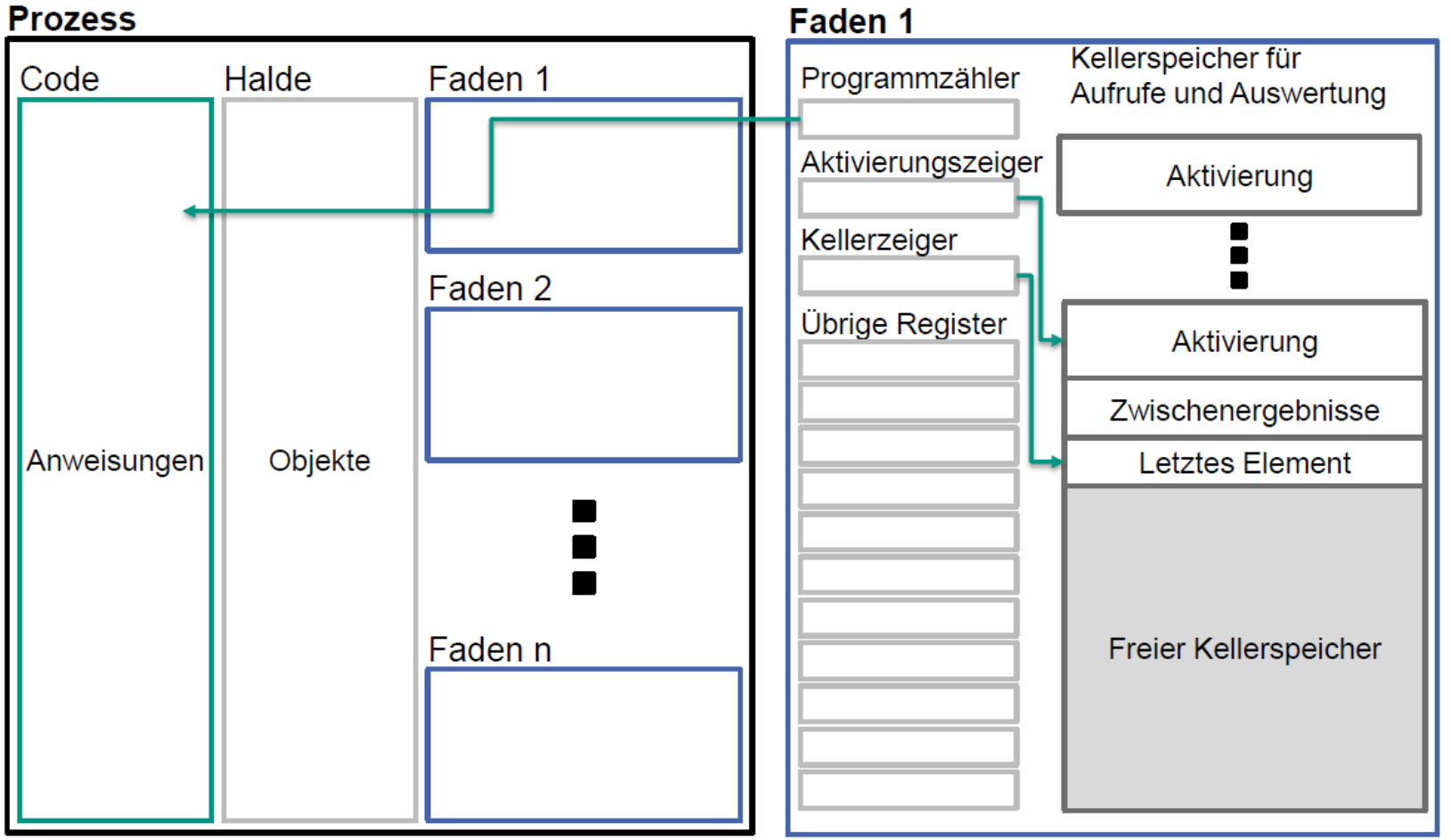
Überblick

Wo sind wir?

Programm vs. Prozess vs. Thread

- Programm = ausführbare Datei
- Prozess = Programm in Ausführung
- Thread = Ausführungseinheit innerhalb eines Prozesses

Grundlagen


Prozess
Code
Halde
Faden 1
Faden 2
Faden n
Anweisungen
Objekte
Faden 1
Programmzähler
Aktivierungszeiger
Kellerzeiger
Übrige Register
Kellerspeicher für
Aufrufe und Auswertung
Aktivierung
Aktivierung
Zwischenergebnisse
Letztes Element
Freier Kellerspeicher

Grundlagen

- Prozess = Programm in Ausführung

Grundlagen

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

Grundlagen

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

Grundlagen

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

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

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 globalen Variablen und demselben Code

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 globalen Variablen und demselben Code
  - Threads haben eigene Stacks und Befehlszeiger (Programmzähler)

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 globalen Variablen und demselben Code
  - Threads haben eigene Stacks und Befehlszeiger (Programmzähler)
    $\Longrightarrow$ Threads haben eigene lokale Variablen und können beliebigen Code des Prozesses ausführen

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
    }
});
```

Motivation

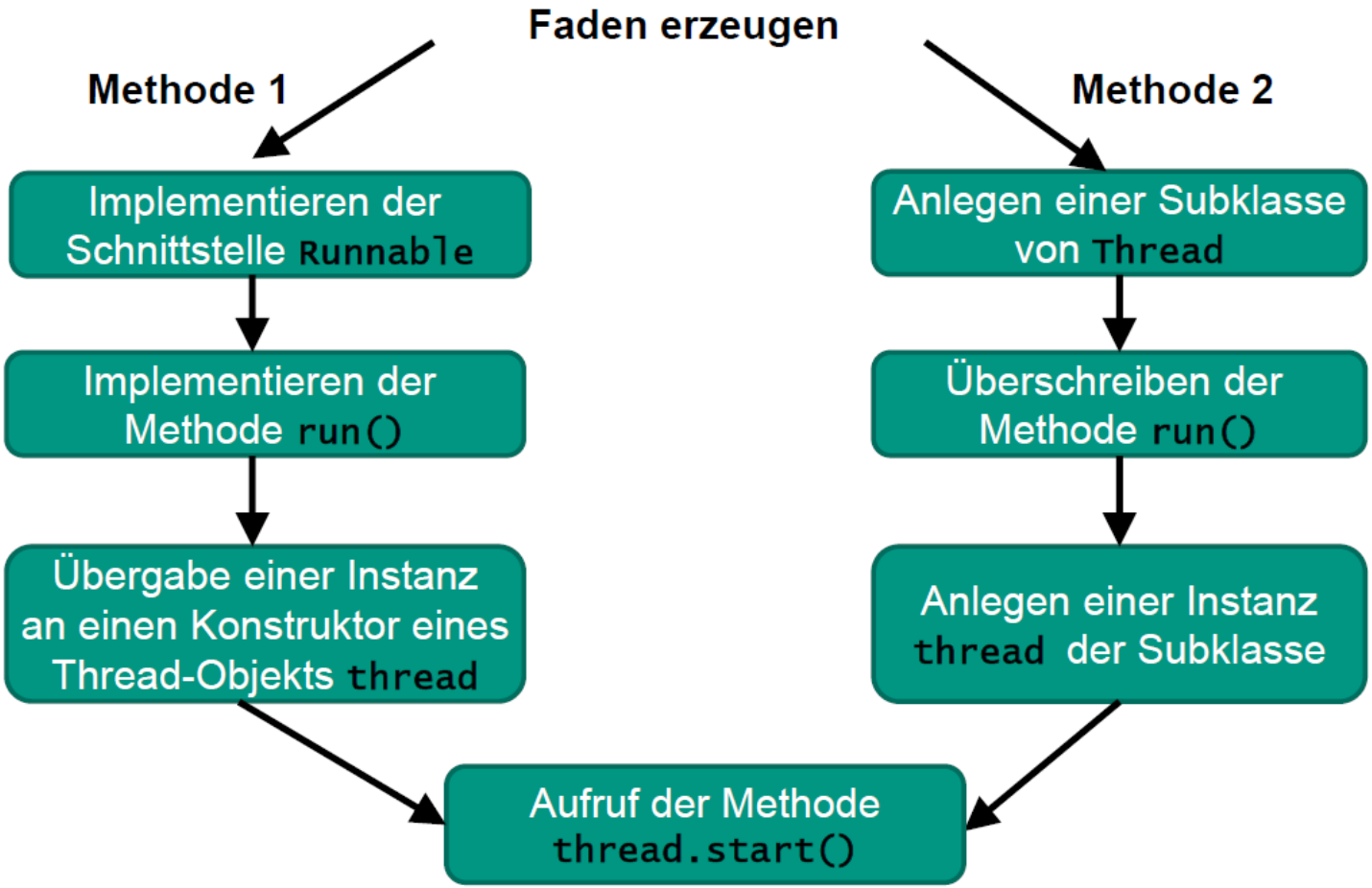
```java
// will freeze the gui when the button is clicked
JButton heavy = new JButton("Freeze");
heavy.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // heavy work
    }
});
```

- “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

Synchronisation

- Problem: Zugriff auf globale Variablen passiert nicht „gleichzeitig“, Unterbrechungen möglich
- Folge: ggf. falsche Ergebnisse

Faden 1 **Faden 2**

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

Nicht nur ein theoretisches Problem

```
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;
}
```

Nicht nur ein theoretisches Problem

```java
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
```

Synchronisation

- Ziel: Zugriff auf gemeinsam genutzte Daten synchronisieren

Synchronisation

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

Synchronisation

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

Kritischer Abschnitt (critical section)

Codeabschnitt, wo Zugriffe auf gemeinsam genutzte Daten stattfinden

Synchronisation

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

Kritischer Abschnitt (critical section)

Codeabschnitt, wo Zugriffe auf gemeinsam genutzte Daten stattfinden

Wettlaufsituation (race condition)

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

Synchronisation

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

Kritischer Abschnitt (critical section)

Codeabschnitt, wo Zugriffe auf gemeinsam genutzte Daten stattfinden

Wettlaufsituation (race condition)

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

- Idee: Monitor einführen

Monitor

- Idee: Monitor einführen

Monitor

- Idee: Monitor einführen
  - Ziel: Bereich im Code markieren,
    - den nur ein Thread gleichzeitig ausführen kann
    - in dem der Thread nicht unterbrochen werden kann

Monitor

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

Monitor

- Idee: Monitor einführen
  - Ziel: Bereich im Code markieren,
    - den nur ein Thread gleichzeitig ausführen kann
    - in dem der Thread nicht unterbrochen werden kann
- Schlüsselwort in Java `synchronized`
- es wird immer an einem Objekt synchronisiert
  - `synchronized(Object o){/* zu schützender Code */ }`

Monitor

- Idee: Monitor einführen
  - Ziel: Bereich im Code markieren,
    - den nur ein Thread gleichzeitig ausführen kann
    - in dem der Thread nicht unterbrochen werden kann
- Schlüsselwort in Java `synchronized`
- es wird immer an einem Objekt synchronisiert
  - `synchronized(Object o){/* zu schützender Code */ }`
- Thread t kommt an eine mit `synchronized(Object o){...}` markierte Stelle

Monitor

- Idee: Monitor einführen
  - Ziel: Bereich im Code markieren,
    - den nur ein Thread gleichzeitig ausführen kann
    - in dem der Thread nicht unterbrochen werden kann
- Schlüsselwort in Java `synchronized`
- es wird immer an einem Objekt synchronisiert
  - `synchronized(Object o){/* zu schützender Code */ }`
- Thread t kommt an eine mit `synchronized(Object o){...}` markierte Stelle
  - es wird geprüft, ob der Monitor gerade frei ist

Monitor

- Idee: Monitor einführen
  - Ziel: Bereich im Code markieren,
    - den nur ein Thread gleichzeitig ausführen kann
    - in dem der Thread nicht unterbrochen werden kann
- Schlüsselwort in Java `synchronized`
- es wird immer an einem Objekt synchronisiert
  - `synchronized(Object o){/* zu schützender Code */ }`
- Thread t kommt an eine mit `synchronized(Object o){...}` markierte Stelle
  - es wird geprüft, ob der Monitor gerade frei ist
  - ist der Monitor frei, kommt t in den kritischen Abschnitt und der Monitor ist besetzt, bis t den Abschnitt wieder verlässt

Monitor

- Idee: Monitor einführen
  - Ziel: Bereich im Code markieren,
    - den nur ein Thread gleichzeitig ausführen kann
    - in dem der Thread nicht unterbrochen werden kann
- Schlüsselwort in Java `synchronized`
- es wird immer an einem Objekt synchronisiert
  - `synchronized(Object o){/* zu schützender Code */ }`
- Thread t kommt an eine mit `synchronized(Object o){...}` markierte Stelle
  - es wird geprüft, ob der Monitor 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

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        }
```

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.exe (30.06.2017, 16:40:24)

Monitor

- `synchronized` an Methoden
  - nicht-statische Methode = `synchronized(this){Methoden-Rumpf}`
  - statische Methode = `synchronized(X.class){Methoden-Rumpf}`

Monitor

- `synchronized` an Methoden
  - nicht-statische Methode = `synchronized(this){Methoden-Rumpf}`
  - statische Methode = `synchronized(X.class){Methoden-Rumpf}`

Aber Achtung

- `synchronized` killt Parallelität
- immer nur ein Thread gleichzeitig im kritischen Abschnitt
- also nur dann benutzen wenn wirklich nötig
  - sonst wird alles wieder sequentiell
- außerdem kritische Abschnitte möglichst klein halten
  - möglichst wenig Arbeit im Abschnitt machen

wait() und notify()

```
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

wait() und notify()

```
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

wait() und notify()

```
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

wait() und notify()

```
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

wait() und notify()

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

wait() und notify()

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

wait() und notify()

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

wait() und notify()

- Idee: brauchen Mechanismus,
  - der es erlaubt den Monitor freizugeben, während man auf etwas wartet
  - 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();
}
```

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?

wait() und notify()

Karlsruher Institut für Technologie

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

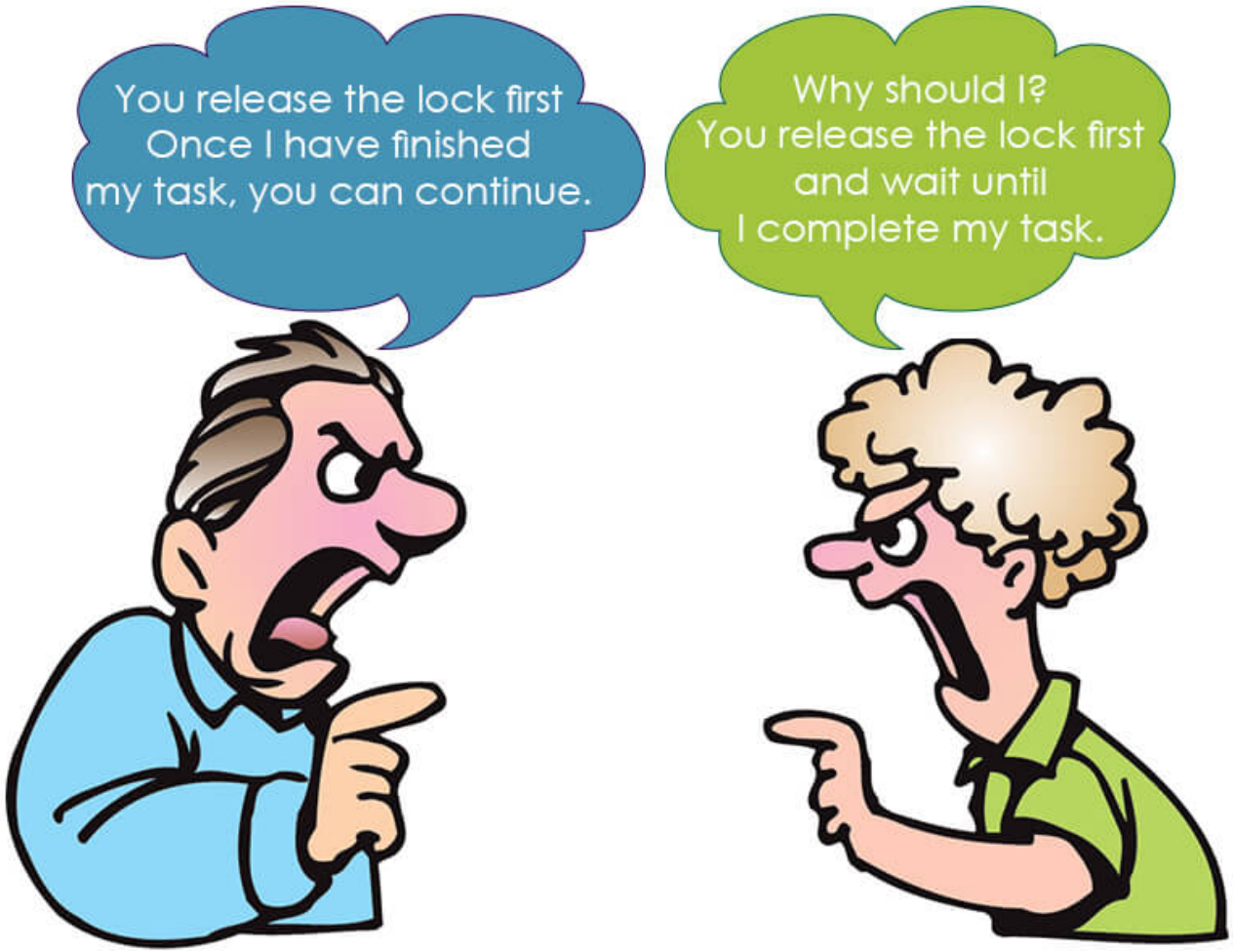
Thread 1

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

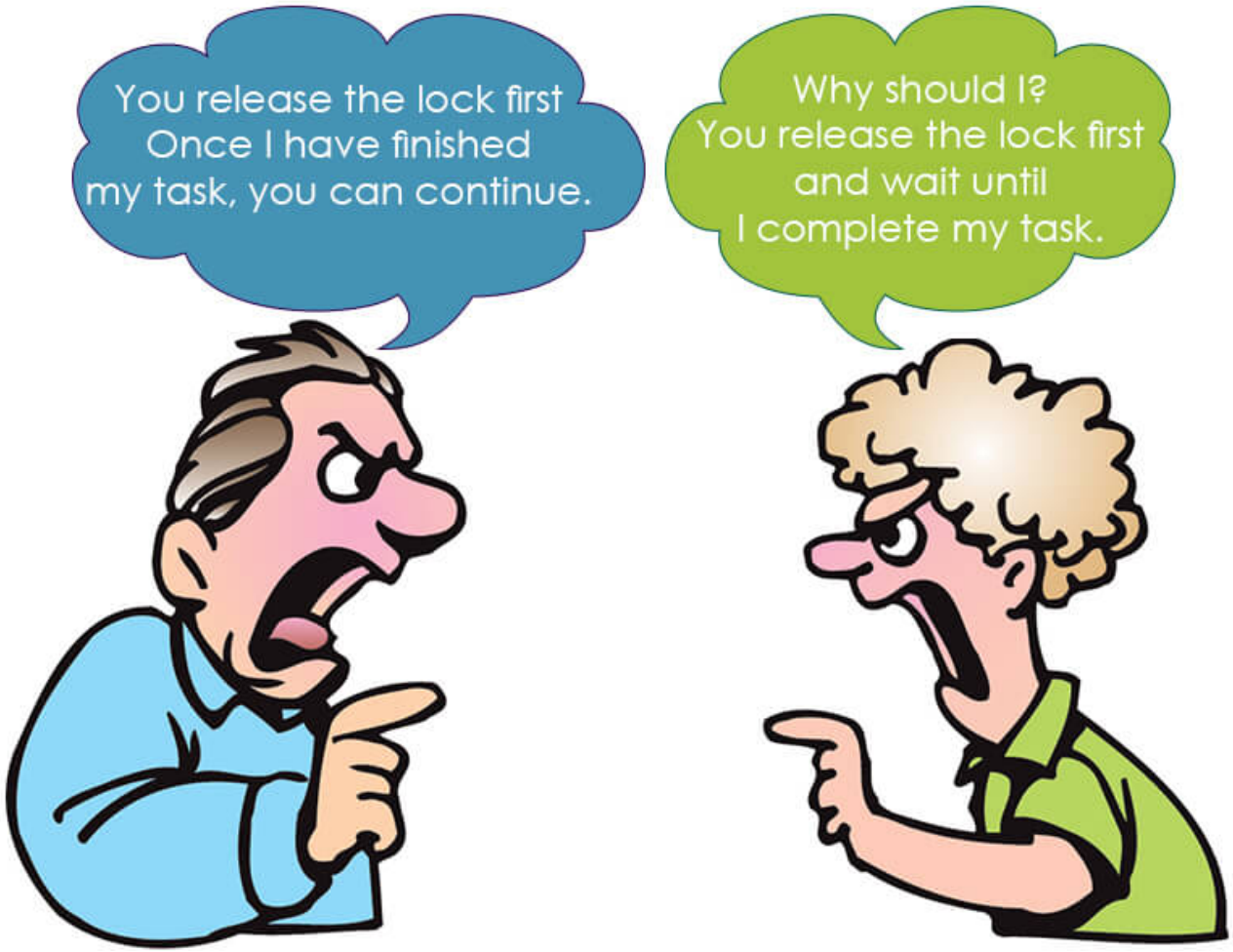
Thread 2

- 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.

Verklemmung (deadlock)



Verklemmung (deadlock)



- Thread A hält Monitor X und benötigt Monitor Y
- Thread B hält Monitor Y und benötigt Monitor X

Verklemmung (deadlock)

Thread 1:

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

Thread 2:

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

klassicher Deadlock!

Verklemmung (deadlock)

Lösungsansatz: Monitore immer in gleicher Reihenfolge anfordern

Thread 1:

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

Thread 2:

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

volatile

- Schlüsselwort `volatile` an Variablen
- Zugriff auf Variablen immer direkt über den Hauptspeicher
  - Caches werden umgangen
- sorgt dafür, dass alle Threads den gleichen Wert sehen

volatile

- Schlüsselwort `volatile` an Variablen
- Zugriff auf Variablen immer direkt über den Hauptspeicher
  - Caches werden umgangen
- sorgt dafür, dass alle Threads den gleichen Wert sehen
- Compiler darf weniger optimieren

volatile

- Schlüsselwort `volatile` an Variablen
- Zugriff auf Variablen immer direkt über den Hauptspeicher
  - Caches werden umgangen
- sorgt dafür, dass alle Threads den gleichen Wert sehen
- Compiler darf weniger optimieren

Löst es das folgende Problem?

`i++` ausgeführt von 2 Threads. `volatile i` ist initial 0.
Ist i nach der Ausführung immer 1?

volatile

- Schlüsselwort `volatile` an Variablen
- Zugriff auf Variablen immer direkt über den Hauptspeicher
  - Caches werden umgangen
- sorgt dafür, dass alle Threads den gleichen Wert sehen
- Compiler darf weniger optimieren

Löst es das folgende Problem?

`i++` ausgeführt von 2 Threads. `volatile i` ist initial 0.
Ist i nach der Ausführung immer 1?

Beispiel-Code

volatile

- Schlüsselwort `volatile` an Variablen
- Zugriff auf Variablen immer direkt über den Hauptspeicher
  - Caches werden umgangen
- sorgt dafür, dass alle Threads den gleichen Wert sehen
- Compiler darf weniger optimieren

Löst es das folgende Problem?

`i++` ausgeführt von 2 Threads. `volatile i` ist initial 0.
Ist i nach der Ausführung immer 1?

Beispiel-Code

- Nein!
  - `volatile` kümmert sich nur darum, dass alle Threads die gleiche Sicht auf den Speicher haben
  - kein Einfluss auf Ausführung
  - kein Schutz vor race conditions

Klausuraufgabe SS14

Parallelität üben

https://deadlockempire.github.io/

Swing Dispatch Thread

```java
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`

Parallelität in GUI nutzen

```
// 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();
    }
});
```

Wie benutzt man nun Parallelität?

1. großes Problem
2. ??? } Parallelität?
3. Profit

Wie benutzt man nun Parallelität?

1. großes Problem in Teilprobleme aufteilen
2. Teilprobleme durch verschiedene Threads parallel lösen lassen
3. Teil-Lösungen zu Gesamt-Lösung zusammensetzen
   - auf critical sections/ race conditions achten
4. Profit = Laufzeit-Reduktion (falls alles richtig gemacht)

Kontrollflussorientiertes Testverfahren (KFO)

- Ziel: “sinnvolle“ Testfälle finden

Vorgehen:

1. gegeben: zu testender Code

Kontrollflussorientiertes Testverfahren (KFO)

- Ziel: “sinnvolle“ Testfälle finden

Vorgehen:

1. gegeben: zu testender Code
2. Code $\Longrightarrow$ Zwischensprache
   - Sprünge umwandeln
   - Grundblöcke finden
   - Grundblöcke prüfen

Kontrollflussorientiertes Testverfahren (KFO)

- Ziel: “sinnvolle“ Testfälle finden

Vorgehen:

1. gegeben: zu testender Code
2. Code $\Longrightarrow$ Zwischensprache
   - Sprünge umwandeln
   - Grundblöcke finden
   - Grundblöcke prüfen
3. Zwischensprache $\Longrightarrow$ Kontrollflussgraph

Kontrollflussorientiertes Testverfahren (KFO)

- Ziel: “sinnvolle“ Testfälle finden

Vorgehen:

1. gegeben: zu testender Code
2. Code $\Longrightarrow$ Zwischensprache
   - Sprünge umwandeln
   - Grundblöcke finden
   - Grundblöcke prüfen
3. Zwischensprache $\Longrightarrow$ Kontrollflussgraph
4. am Kontrollflussgraphen Testfälle finden

Kontrollflussorientiertes Testverfahren (KFO)

- Ziel: “sinnvolle“ Testfälle finden

Vorgehen:

1. gegeben: zu testender Code
2. Code $\Longrightarrow$ Zwischensprache
   - Sprünge umwandeln
   - Grundblöcke finden
   - Grundblöcke prüfen
3. Zwischensprache $\Longrightarrow$ Kontrollflussgraph
4. am Kontrollflussgraphen Testfälle finden
   - Anweisungsüberdeckung
   - Zweigüberdeckung
   - Pfadüberdeckung

KFO: Code → Zwischensprache

- Sprünge umwandeln

```
int a = 9;
System.out.println("blubb");
while (a == 9) {
  int z = 0;
  for (int i = 0; i <= b; i++) {
    n++;
  }
  z++;
  if (a == z) {
    a = 8;
  }
}
```

KFO: Code → Zwischensprache

- Sprünge umwandeln (while, for, if, `x ? a : b`,...)
  - wie geht das für `while`, `if`, `for`? (Tafel)
  - Ziel: nur `if`, `not`, `goto` verwenden. Kein {...}

```
int a = 9;
System.out.println("blubb");
while (a == 9) { <- hier
  int z = 0;
  for (int i = 0; i <= b; i++) { <- hier
    n++;
  }
  z++;
  if (a == z) { <- hier
    a = 8;
  }
}
```

Sprünge wurden umgewandelt

```
01 int a = 9;
02 System.out.println("blubb");
03 if not (a == 9) goto 14;
04 int z = 0;
05 int i = 0;
06 if not (i <= b) goto 10;
07 n++;
08 i++;
09 goto 06;
10 z++;
11 if not (a == z) goto 13;
12 a = 8;
13 goto 03;
14
```

KFO: Code → Zwischensprache

- nächster Schritt: Grundblöcke finden
- Code bis `goto` ist ein Grundblock

```
01 int a = 9;
02 System.out.println("blubb");
03 if not (a == 9) goto 14;

04 int z = 0;
05 int i = 0;
06 if not (i <= b) goto 10;

07 n++;
08 i++;
09 goto 06;

10 z++;
11 if not (a == z) goto 13;

12 a = 8;
13 goto 03;

14
```

- haben erste Grundblöcke gefunden

```
01 int a = 9;
02 System.out.println("blubb");
03 if not (a == 9) goto 14;

04 int z = 0;
05 int i = 0;
06 if not (i <= b) goto 10;

07 n++;
08 i++;
09 goto 06;

10 z++;
11 if not (a == z) goto 13;

12 a = 8;
13 goto 03;

14
```

- haben erste Grundblöcke gefunden
- jetzt: Grundblöcke prüfen
- `goto` dürfen nur auf Start eines Blocks verweisen
- daher: Blöcke ggf. noch weiter unterteilen

```
01 int a = 9;
02 System.out.println("blubb");
03 if not (a == 9) goto 14;

04 int z = 0;
05 int i = 0;
06 if not (i <= b) goto 10;

07 n++;
08 i++;
09 goto 06; <-- kaputt!

10 z++;
11 if not (a == z) goto 13; <-- kaputt!

12 a = 8;
13 goto 03; <-- kaputt!

14
```

- haben erste Grundblöcke gefunden
- jetzt: Grundblöcke prüfen
- `goto` dürfen nur auf Start eines Blocks verweisen
- daher: Blöcke ggf. noch weiter unterteilen

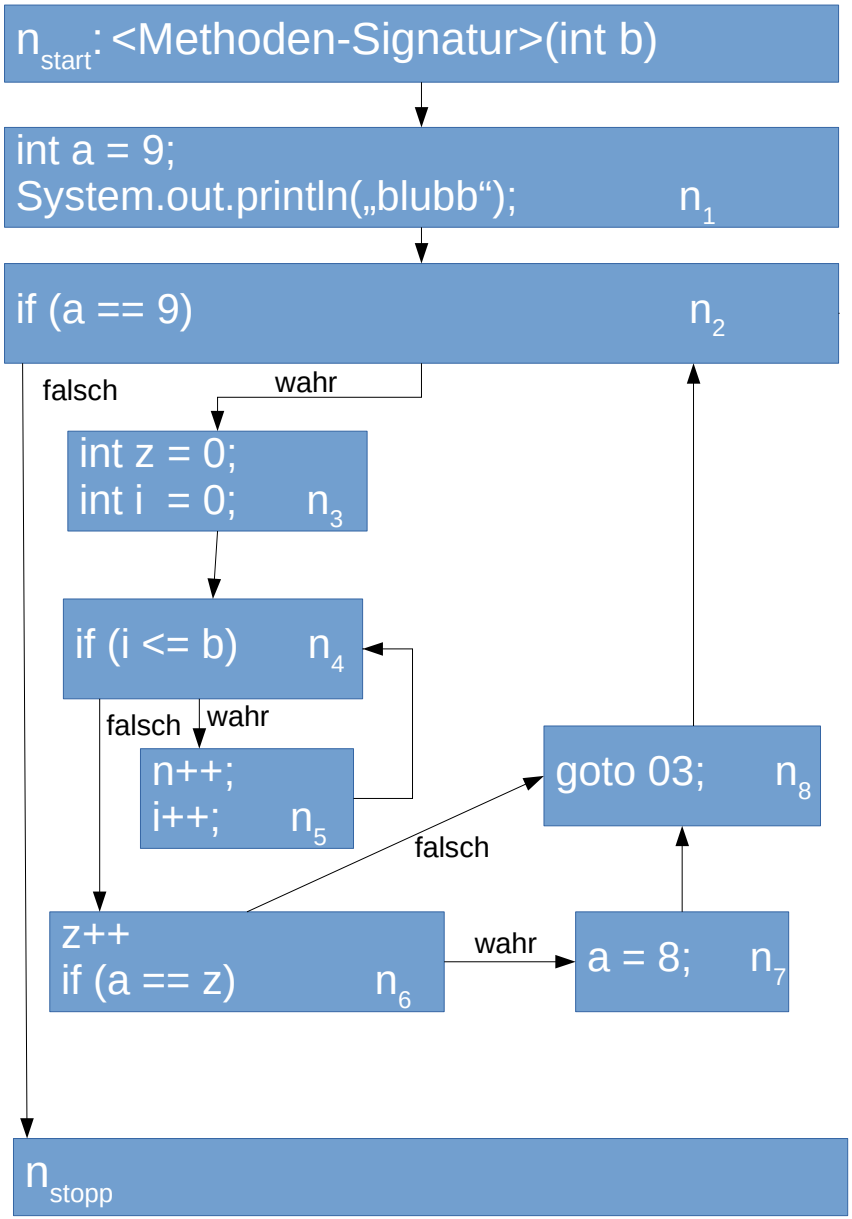
```
01 int a = 9;
02 System.out.println("blubb");

03 if not (a == 9) goto 14;

04 int z = 0;
05 int i = 0;

06 if not (i <= b) goto 10;

07 n++;
08 i++;
09 goto 06;

10 z++;
11 if not (a == z) goto 13;

12 a = 8;

13 goto 03;

14
```

```
01 int a = 9;
02 System.out.println("blubb");

03 if not (a == 9) goto 14;

04 int z = 0;
05 int i = 0;

06 if not (i <= b) goto 10;

07 n++;
08 i++;
09 goto 06;

10 z++;
11 if not (a == z) goto 13;

12 a = 8;

13 goto 03;

14
```

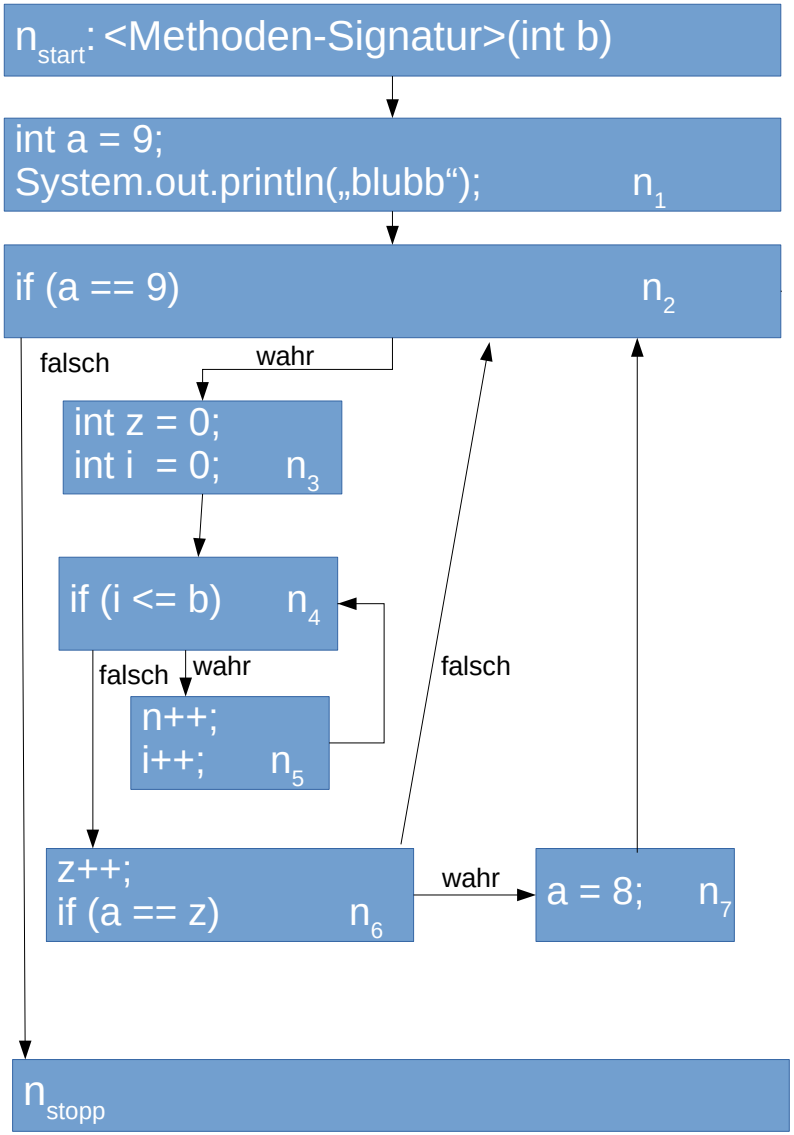
- jetzt sind wir fertig mit den Grundblöcken

```
01 int a = 9;
02 System.out.println("blubb");

03 if not (a == 9) goto 14;

04 int z = 0;
05 int i = 0;

06 if not (i <= b) goto 10;

07 n++;
08 i++;
09 goto 06;

10 z++;
11 if not (a == z) goto 13;

12 a = 8;

13 goto 03;

14
```

- jetzt sind wir fertig mit den Grundblöcken
- als nächstes: Graphen hinzeichnen
- dazu Start- und Endblock hinzufügen
- Blöcke benennen ($n_{start}, n_1, \ldots, n_x, n_{stopp}$)
- Knoten und Kanten hinzeichnen


n_start: <Methoden-Signatur>(int b)
int a = 9;
System.out.println(„blubb“);
n_1
if (a == 9)
n_2
falsch
wahr
int z = 0;
int i = 0;
n_3
if (i <= b)
n_4
falsch
wahr
n++;
i++;
n_5
goto 03;
n_8
falsch
z++
if (a == z)
n_6
wahr
a = 8;
n_7
n_stopp

- goto-Knoten kann man auch weglassen



KFO: Anweisungsüberdeckung

- wähle Testfälle so, dass jeder Grundblock traversiert wird

KFO: Anweisungsüberdeckung

- wähle Testfälle so, dass jeder Grundblock traversiert wird
  $\Longrightarrow$ Entdeckung nicht erreichbarer Code-Abschnitte

KFO: Anweisungsüberdeckung

- wähle Testfälle so, dass jeder Grundblock traversiert wird
  $\Longrightarrow$ Entdeckung nicht erreichbarer Code-Abschnitte
- aber: kein ausreichendes Testkriterium
  - Schleifen einmal durchlaufen reicht schon

KFO: Zweigüberdeckung

- wähle Testfälle so, dass jeder Zweig (=Kante) traversiert wird

KFO: Zweigüberdeckung

- wähle Testfälle so, dass jeder Zweig (=Kante) traversiert wird
  $\Longrightarrow$ Entdeckung nicht erreichbarer Kanten

KFO: Zweigüberdeckung

- wähle Testfälle so, dass jeder Zweig (=Kante) traversiert wird
  $\Longrightarrow$ Entdeckung nicht erreichbarer Kanten
- wie bei Anweisungsüberdeckung
  - Schleifen werden ggf. nicht ausreichend getestet

KFO: Pfadüberdeckung

- Finde **alle** vollständigen, unterschiedlichen Pfade
  - wähle Testfälle so, dass alle gefundenen Pfade durchlaufen werden
- vollständiger Pfad = Anfang bei n_{start}, Ende bei n_{stopp}

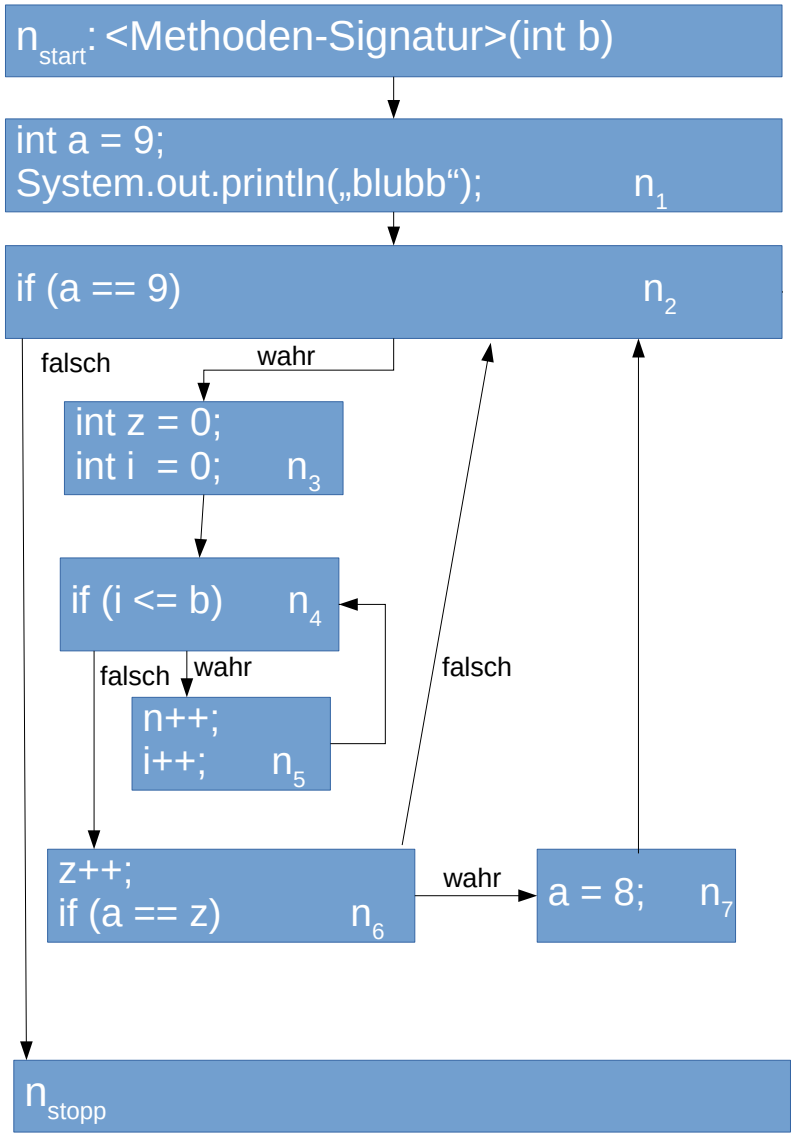
KFO: Pfadüberdeckung

- Finde **alle** vollständigen, unterschiedlichen Pfade
  - wähle Testfälle so, dass alle gefundenen Pfade durchlaufen werden
- vollständiger Pfad = Anfang bei n_{start}, Ende bei n_{stopp}
- nicht praktikabel, da
  - Schleifen die Anzahl der möglichen Pfade stark erhöhen
    - sogar „unendlich viele“, wenn Iterationen von Eingabe abhängig

KFO: Pfadüberdeckung

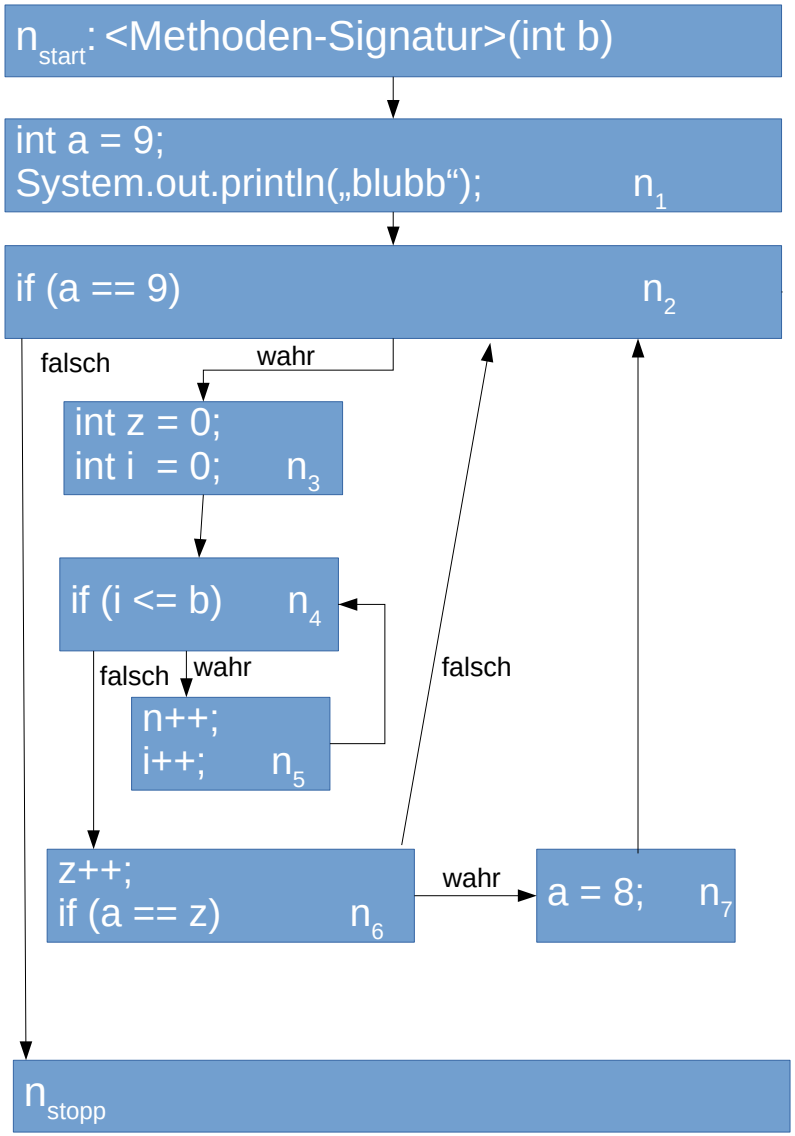
- Finde **alle** vollständigen, unterschiedlichen Pfade
  - wähle Testfälle so, dass alle gefundenen Pfade durchlaufen werden
- vollständiger Pfad = Anfang bei n_{start}, Ende bei n_{stopp}
- nicht praktikabel, da
  - Schleifen die Anzahl der möglichen Pfade stark erhöhen
    - sogar „unendlich viele“, wenn Iterationen von Eingabe abhängig
  - manche Pfade nicht ausführbar sind
    - z.B. sich gegenseitig ausschließende Bedingungen

„Sinnvolle“ Testfälle



- was ist nun unsere Testfallmenge?
  - = welche `b` testen?
- für Anweisungsüberdeckung?
- für Zweigüberdeckung?

„Sinnvolle“ Testfälle



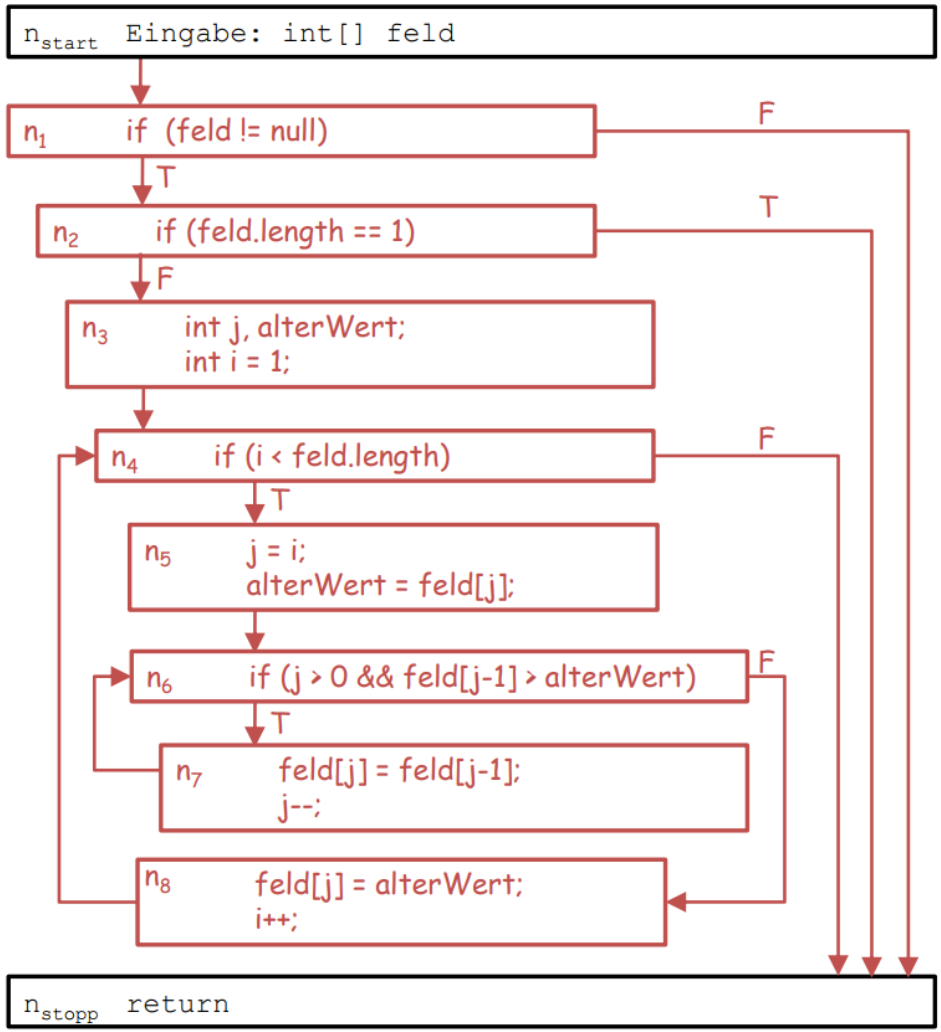
- b=0 erfüllt bereits beides
- Pfad
  $(n_{start}, n_1, n_2, n_3, n_4, n_5, n_4, n_6, n_2, \ldots, n_6, n_7, n_2, n_{stopp})$
- i.A. aber mehrere Testfälle nötig

Klausuraufgabe SS11

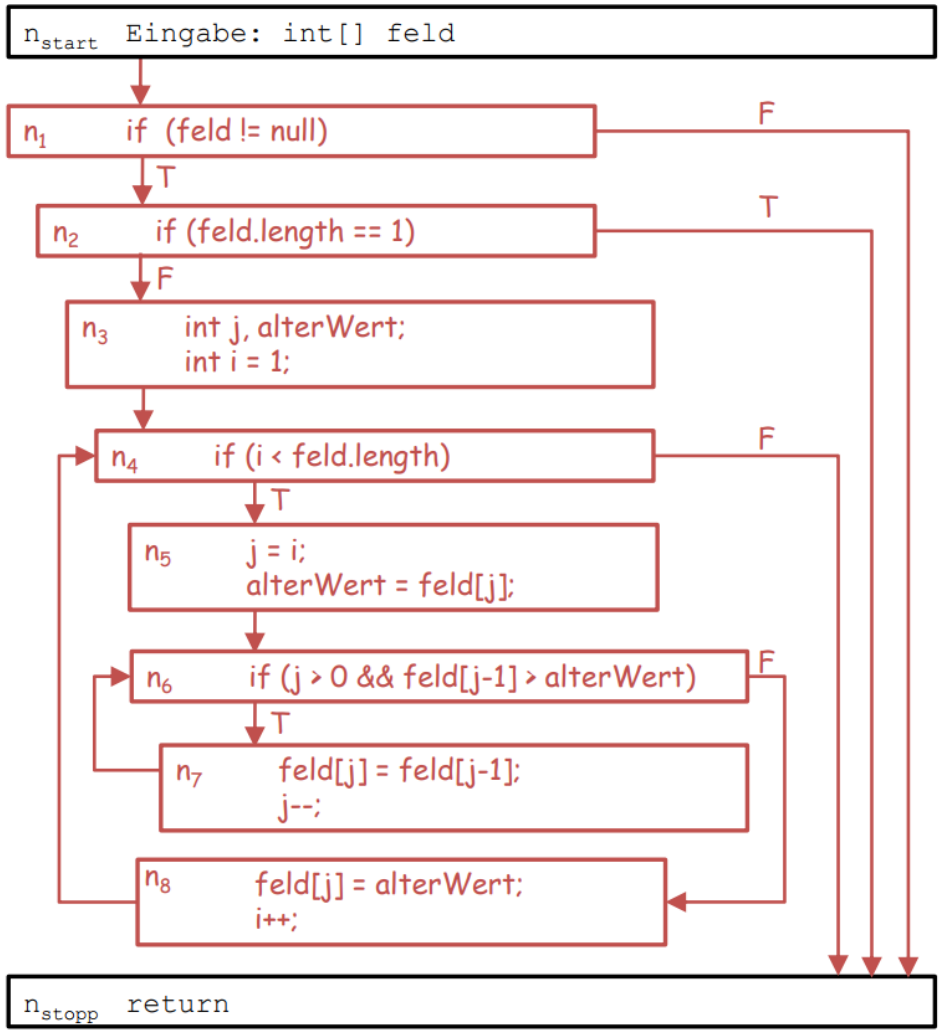
```
01  public void sortiere(int[] feld) {
02    if (feld != null) {
03      if (feld.length == 1) {
04        return;
05      } else {
06        int j, alterWert;
07        for (int i = 1; i < feld.length; i++) {
08          j = i;
09          alterWert = feld[i];
10          while (j > 0 && feld[j - 1] > alterWert) {
11            feld[j] = feld[j - 1];
12            j--;
13          }
14          feld[j] = alterWert;
15        }
16      }
17    }
18  }
```

Erstellen Sie den Kontrollflussgraphen und geben Sie einen Pfad an, der Anweisungsüberdeckung erzielt.

MuLö KFO


n_start Eingabe: int[] feld
n1 if (feld != null)
F
T
n2 if (feld.length == 1)
T
F
n3 int j, alterWert;
int i = 1;
n4 if (i < feld.length)
F
T
n5 j = i;
alterWert = feld[j];
n6 if (j > 0 && feld[j-1] > alterWert)
F
T
n7 feld[j] = feld[j-1];
j--;
n8 feld[j] = alterWert;
i++;
n_stopp return

MuLö KFO



Pfad: $(n_{start}, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_6, n_8, n_4, n_{stopp})$

Kurzschlussauswertung

Achtung, für ÜBs/Klausur

- MuLö ist alt, nicht ganz korrekt
- Kurzschlussauswertung (`&&` bzw. `||`) auflösen (in zwei if)

Tipps - 6. Übungsblatt

Aufgabe 1: Kontrollfluss-orientiertes Testen

- zur Sicherheit Zwischensprache benutzen
- Definitionen der verschiedenen Abdeckungen anschauen

Tipps - 6. Übungsblatt

Aufgabe 1: Kontrollfluss-orientiertes Testen

- zur Sicherheit Zwischensprache benutzen
- Definitionen der verschiedenen Abdeckungen anschauen

Aufgabe 2: Parallelisierung von Shutterpile

- Berechnung des Wasserzeichenbilds parallelisieren

Tipps - 6. Übungsblatt

Aufgabe 1: Kontrollfluss-orientiertes Testen

- zur Sicherheit Zwischensprache benutzen
- Definitionen der verschiedenen Abdeckungen anschauen

Aufgabe 2: Parallelisierung von Shutterpile

- Berechnung des Wasserzeichenbilds parallelisieren
- Thread-Pool spart euch die manuelle Verwaltung der Threads
  - `ExecutorService es = Executors.newFixedThreadPool(amountOfThreads);`
  - `es.execute(myRunnable);` (beliebig viele)
  - `es.shutdown();`

Tipps - 6. Übungsblatt

Aufgabe 1: Kontrollfluss-orientiertes Testen

- zur Sicherheit Zwischensprache benutzen
- Definitionen der verschiedenen Abdeckungen anschauen

Aufgabe 2: Parallelisierung von Shutterpile

- Berechnung des Wasserzeichenbilds parallelisieren
- Thread-Pool spart euch die manuelle Verwaltung der Threads
  - `ExecutorService es = Executors.newFixedThreadPool(amountOfThreads);`
  - `es.execute(myRunnable);` (beliebig viele)
  - `es.shutdown();`
- Tests schreiben für Korrektheit

Tipps - 6. Übungsblatt

Aufgabe 3: Abnahmetests

- Tests schreiben für spezielle Anforderungen
- Teil d) handschriftlich!

Tipps - 6. Übungsblatt

Aufgabe 3: Abnahmetests

- Tests schreiben für spezielle Anforderungen
- Teil d) handschriftlich!

Aufgabe 4: Parallelisierungswettbewerb

- Aufgabe 2 verbessern und Laufzeit messen
- auch hier Teile (Erklärung des Ansatzes) handschriftlich abgeben

Denkt dran!

Abgabe

- Deadline am 11.7. um 12:00
- Aufgabe 1 und Teile von 3 und 4 handschriftlich
- auf jeden Fall abgeben, wenn ihr noch Punkte braucht

Bis dann! (dann := 17.07.18)