

# Tafelübung 06

## Algorithmen und Datenstrukturen

Lehrstuhl für Informatik 2 (Programmiersysteme)

Friedrich-Alexander-Universität Erlangen-Nürnberg

Wintersemester 2016/2017

Lehrstuhl Informatik 2  
Programmiersysteme



FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

# Übersicht

## Organisatorisches

### $\mathcal{O}$ -Kalkül

Theorie

Beispiele

Laufzeitanalysen

Skalierbarkeit

## Objektorientierte Programmierung

Grundlagen

Klassen-Variable  $\leftrightarrow$  Instanz-Variable

Konstruktoren

Klassen-Methode  $\leftrightarrow$  Instanz-Methode

Getter und Setter

Vollständiges Beispiel

Technisches

## Debugging

Allgemeines

Debugging in Eclipse

# Organisatorisches

Lehrstuhl Informatik 2  
Programmiersysteme



FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

# Prüfungsanmeldung

## Anmeldezeitraum

- bis Freitag, 9. Dezember, 12:00 Uhr (**mittags!**)
- auf meinCampus → Prüfungen → Prüfungsan- und -abmeldung
- bitte für alle Klausuren anmelden, die ihr womöglich schreiben wollt  
~> Abmelden später noch möglich, Anmelden nicht!

## Anmeldung für AuD

- für die **Klausur** regulär anmelden (meinCampus)
- für den **Übungsschein** ist **keine** Anmeldung auf meinCampus erforderlich  
~> stattdessen: Daten im EST aktuell halten und nochmals überprüfen  
~> vor allem Name, Matrikelnummer, ...

# $\mathcal{O}$ -Kalkül

Lehrstuhl Informatik 2  
Programmiersysteme



FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

## $\mathcal{O}$ -Kalkül: Motivation

- gegeben sei ein Algorithmus  $A$  für ein Problem  $P$ 
  - Welche **Laufzeit** hat der Algorithmus  $A$ ?
  - ... im **Vergleich** mit anderen Algorithmen  $A'$  für dasselbe Problem  $P$ ?
- **lineare Faktoren** wie die CPU-Geschwindigkeit sind **uninteressant**
  - im Allgemeinen gilt: schnellere CPU  $\leadsto$  kürzere Laufzeit
  - $\leadsto$  keine „Eigenschaft“ des Algorithmus
- $\leadsto$  **asymptotische Aufwandsanalyse** mittels  $\mathcal{O}$ -Kalkül
  - Verhalten der Programmlaufzeit **in Abhängigkeit der Problemgröße**
  - $\leadsto$  Aussage über **Skalierbarkeit** des Algorithmus, nicht über absolute Laufzeit!

# Asymptotische Schranken

## Definition „Asymptote“ [Wikipedia]

Eine Asymptote [...] ist [...] **eine Funktion**, die sich einer **anderen Funktion** [...] im **Unendlichen** annähert.

## Asymptotisch obere Schranke

Eine Funktion  $g$  ist eine **asymptotisch obere Schranke** für eine Funktion  $f$ , falls

- $f(n)$  höchstens so schnell wächst wie  $g(n)$  und
- ein Umstiegspunkt  $n_0$  existiert, sodass  $f(n)$  ab diesem Punkt immer kleiner oder gleich  $g(n)$  multipliziert mit einer Konstanten  $c$  ist.

## In Formeln: $\mathcal{O}$ -Kalkül

Eine Funktion  $g$  ist asymptotisch obere Schranke einer **Menge von Funktionen**  $f$ :

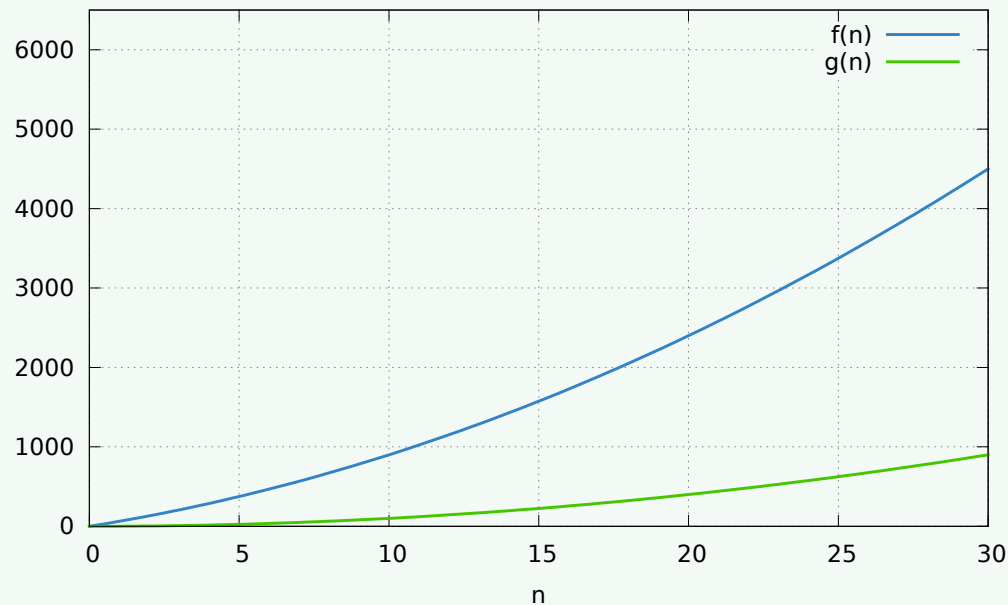
$$\mathcal{O}(g(n)) = \{f : \mathbb{N} \mapsto \mathbb{R}^+ \mid \exists n_0 \in \mathbb{N}, c \in \mathbb{R} \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

# Beispiel

## $\mathcal{O}$ -Kalkül

$$\mathcal{O}(g(n)) = \{f : \mathbb{N} \mapsto \mathbb{R}^+ \mid \exists n_0 \in \mathbb{N}, c \in \mathbb{R} \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

Beispiel:  $f(n) = 3n^2 + 2n \in \mathcal{O}(n^2)$



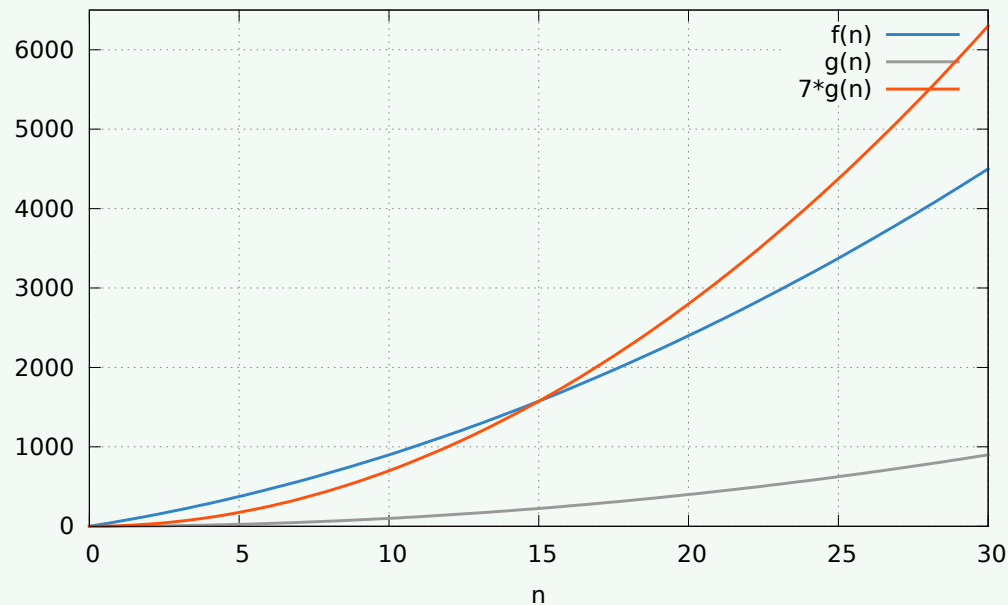


# Beispiel

## $\mathcal{O}$ -Kalkül

$$\mathcal{O}(g(n)) = \{f : \mathbb{N} \mapsto \mathbb{R}^+ \mid \exists n_0 \in \mathbb{N}, c \in \mathbb{R} \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

Beispiel:  $f(n) = 3n^2 + 2n \in \mathcal{O}(n^2)$



## „Rechenregeln“ für den $\mathcal{O}$ -Kalkül

### „Rechenregeln“ für den $\mathcal{O}$ -Kalkül

Es sei:

- $f(n) \in \mathcal{O}(r(n))$
- $g(n) \in \mathcal{O}(s(n))$
- $d > 0$  konstant

Dann gilt:

- $f(n) \pm d \in \mathcal{O}(r(n))$
- $d \cdot f(n) \in \mathcal{O}(r(n))$
- $f(n) \cdot g(n) \in \mathcal{O}(r(n) \cdot s(n))$
- $f(n) + g(n) \in \mathcal{O}(r(n) + s(n)) = \mathcal{O}(\max(r(n), s(n)))$

Außerdem:

- Transitivität:  $x(n) \in \mathcal{O}(y(n)), y(n) \in \mathcal{O}(z(n)) \Rightarrow x(n) \in \mathcal{O}(z(n))$

## Beispiele zum $\mathcal{O}$ -Kalkül

### Beispiele

- $(9n^3 + 17 + 4n) \in \mathcal{O}(n^3)$
- $(4n \cdot \sqrt{n}) \in \mathcal{O}(n \cdot \sqrt{n})$
- $(120095 + 0.01003n) \in \mathcal{O}(n)$
- $(n^5 + 2^n) \in \mathcal{O}(2^n)$
- $(\log_2 n + 0.4^n) \in \mathcal{O}(\log_2 n) = \mathcal{O}(\log n)$ , denn:  $\log_a x = \frac{\log_b x}{\log_b a}$

### Hilfreich

<http://de.wikipedia.org/wiki/Logarithmus> → Rechenregeln

## Laufzeitanalysen mittels $\mathcal{O}$ -Kalkül

- für Laufzeitanalysen: „Zählen“ der auszuführenden Operationen
  - und zwar in Abhängigkeit der Werte der Eingabeparameter
  - dann: Bestimmung der Klasse im  $\mathcal{O}$ -Kalkül
- beim „Zählen“ der Operationen Unterscheidung zwischen:
  - Elementaroperationen:
    - Zuweisung, Vergleich, Fallunterscheidung, arithmetische Operation, ...
    - Kosten: 1
  - Komplexe Kontrollstrukturen:
    - Schleifen, Methodenaufrufe, ...
    - Kosten: Summe der anfallenden Elementaroperationen

## Beispiele zur Laufzeitanalyse (1)

### Beispiel 1

```
public static void example(int n) {  
    for (int i = 0; i < n; ++i) {  
        System.out.println("foo");  
        System.out.println("bar");  
    }  
}
```

Aufwand der Methode `example()` in Abhängigkeit des Parameters `n`?

`example()`  $\in \mathcal{O}(n)$

## Beispiele zur Laufzeitanalyse (2)

### Beispiel 2

```
public static void foo1(int n, int m) {  
    int sum = 0;  
    for (int i = 1; i <= n; i++)  
        for (int j = 1; j <= m; j++)  
            for (int k = 1; k <= j; k++)  
                sum += i*k*j;  
    return sum;  
}
```

Lösung:  $\text{foo1}() \in ?\mathcal{O}(n \cdot m^2)$

- äußere Schleifen:  $n$  bzw.  $m$  Durchläufe
- innere Schleife mit Laufvariable  $k$ :
  - Anzahl Durchläufe steigt in Abhängigkeit der äußeren Laufvariable  $j$ :  
 $k = 1, 2, 3, 4, \dots, m-1, m \Rightarrow \sum_{k=1}^m k = \frac{m \cdot (m+1)}{2} \in \mathcal{O}(m^2)$
  - Zuweisung ist eine Elementaroperation

## Beispiele zur Laufzeitanalyse (3)

### Beispiel 3

```
public static void foo2(int n){  
    int ret = 0;  
    for (int i = 1; i < n; i *= 5) {  
        ret += bar(n); //bar(n) = O(n)  
    }  
}
```

Lösung:  $\text{foo2}() \in ?\mathcal{O}(n \cdot \log_5 n) = \mathcal{O}(n \cdot \log n)$

- bezeichne  $k$  die Anzahl der Durchläufe der for-Schleife  
     $\leadsto$  Werte von  $i$ :  $5^0, 5^1, 5^2, \dots, 5^{k-1}$
- wie häufig wird die Schleife durchlaufen?
- anders gefragt: wann übersteigt  $5^{k-1}$  den Parameter  $n$ ?  
     $\leadsto$  Umkehrfunktion  $\log_5 n$
- in jedem Schleifendurchlauf: Aufwand von  $\mathcal{O}(n)$

# Skalierbarkeit

## Ganz wichtig!

Die Klasse im  $\mathcal{O}$ -Kalkül sagt **nichts über die Geschwindigkeit** eines Algorithmus aus! Sie macht „lediglich“ eine Aussage darüber, wie die Laufzeit des Algorithmus in Abhängigkeit der Problemgröße **skaliert**.

## Skalierbarkeit!?

Wie **verhält** sich die Laufzeit des Algorithmus, wenn man die **Problemgröße** verdoppelt, verdreifacht, ...?

## Vergleich zweier Algorithmen

Seien  $A$  und  $B$  zwei Algorithmen für dasselbe Problem  $P$ , und habe o.B.d.A  $A$  einen **asymptotisch höheren Aufwand**. Dann kann  $A$  dennoch für gewisse (kleine) Problemgrößen **schneller** sein als  $B$ .



## Beispiel zur Skalierbarkeit (1)

### Beispiel: fib()

```
public static int fib(int n) {  
    if (n < 2) {  
        return n;  
    }  
    return fib(n-1) + fib(n-2);  
}
```

### Grobe Skizze zur Bestimmung des asymptotischen Aufwands

- kaskadenartige Rekursion, Abstieg für  $n-1$  und  $n-2$   
     $\leadsto$  Aufwandsklasse  $\mathcal{O}(2^n)$  (es gibt noch kleinere obere Schranke)

## Beispiel zur Skalierbarkeit (2)

### Beispiel: fibDP()

```
static final int MAX_VAL = 46; // damit fib(n) noch in einen int passt
static int[] table;
public static int fibDP(int n){
    if (table == null) {
        table = new int[MAX_VAL+1];
        for (int i = 0; i < table.length; i++) { table[i] = -1; }
    }
    if (table[n] == -1) {
        if (n < 2) { table[n] = n; }
        else { table[n] = fibDP(n-1) + fibDP(n-2); }
    }
    return table[n];
}
```

### Asymptotischer Aufwand von fibDP

- jeder Wert wird maximal einmal berechnet  
     $\leadsto$  Aufwandsklasse  $\mathcal{O}(n)$

## Beispiel zur Skalierbarkeit (3)

### Beispiel: fib() und fibDP

```
public static int fib(int n) {  
    // berechnet fib(n) in  $O(2^n)$   
}  
  
static final int MAX_VAL = 46;  
static int[] table;  
public static int fibDP(int n){  
    // berechnet fib(n) in  $O(n)$   
}
```

### Beobachtung

- für große  $n$  ist fibDP( $n$ ) *deutlich* schneller fertig als fib( $n$ )
  - aber:
    - in fibDP() muss erst die Tabelle initialisiert werden
    - ist  $n$  klein, dauert das länger als die eigentliche Berechnung
- ~> wenn  $n$  klein ist, ist fib( $n$ ) beim ersten Aufruf schneller als fibDP( $n$ )

# Objektorientierte Programmierung

Lehrstuhl Informatik 2  
Programmiersysteme



FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

# Objektorientierte Programmierung

- Objektorientierte Programmierung (OOP): **Programmierparadigma**
  - unterstützt von Java, C++, Python, u.v.w.m.
- Grundkonzepte der OOP:
  - Identifizieren von **logischen Einheiten**:
    - Daten und Funktionen, die auf diesen Daten arbeiten
  - Beschreibung dieser Einheiten durch sog. **Klassen** ( $\leadsto$  „Bauplan“)
  - Erzeugen von konkreten **Objekten** anhand dieser Klassen
    - sog. **Instanziierung** von Klassen
  - dabei Anwendung von **Datenkapselung**:
    - Verbergung von Implementierungs-Details
    - Zugriff auf interne Daten nur über definierte **Schnittstellen**

## Motivierendes Beispiel

### Ohne Objektorientierte Programmierung

```
// ein Rechteck habe eine Breite , eine Hoehe und eine "ID"  
  
// zwei Rechtecke A und B...  
int breiteA = 13, hoeheA = 3, idA = 1;  
int breiteB = 12, hoeheB = 11, idB = 2;  
  
// ...und deren Flaechen  
int flaecheA = breiteA*hoeheA;  
int flaecheB = breiteB*hoeheB;  
  
// Rechteck C entsteht durch Drehung von Rechteck A  
int breiteC = hoeheA, hoeheC = breiteA , idC = 3;
```

### Nicht schön...

Obwohl Breite, Höhe und ID jeweils **logisch zusammengehören**, sind sie in **unabhängigen Variablen** gespeichert. Für jedes neue Rechteck müssten **drei neue Variablen** angelegt werden. Die **Logik** zur **Flächenberechnung** und **Drehung** ist von den Daten getrennt.

## Definition der Klasse Rechteck

- „**Rechteck**“ ist Kandidat für eine Klasse
  - Konzept, das **mehrfach** im Programm benötigt wird
  - mehrere **Daten**, die logisch zusammengehören (Breite, Höhe, ID)
  - **Funktionen**, die auf diesen Daten arbeiten (Fläche berechnen, drehen)

### Gerüst der Klasse Rechteck

```
public class Rechteck {  
    // noch zu füllen  
}
```

## Einschub: Klassen in Java

- öffentliche Klasse  $\mapsto$  Datei mit gleichem Namen
  - Beispiel: öffentliche Klasse Foo muss in der Datei Foo.java stehen
  - was „öffentlich“ genau bedeutet  $\leadsto$  irgendwann anders...

$\leadsto$  nur eine öffentliche Klasse pro Datei...

- aber: es gibt auch nicht-öffentliche Klassen
  - davon beliebig viele in einer Datei möglich

### Beispiel: Bazz.java

```
class Foo {}  
class Bar {}  
public class Bazz {}
```



## Klassen-Variable ↔ Instanz-Variable

- Instanz-Variable:
  - gehört zu einem Objekt
  - jedes Objekt der Klasse hat eine eigene Kopie
- Klassen-Variable (auch statische Variable):
  - gehört zu der Klasse als Ganzes
  - alle Objekte teilen sich diese Variable
- beide werden als „globale“ Variablen innerhalb der Klasse deklariert
  - also „außerhalb von Methoden“
  - Klassen-Variablen mit zusätzlichem Schlüsselwort `static`
- damit lassen sich Attribute implementieren
  - Speicherung von Eigenschaften der Objekte bzw. der Klasse

## Attribute der Klasse Rechteck

- **Objekt-Attribute**  $\mapsto$  Instanz-Variablen
  - Breite und Höhe
  - ID
- **Klassen-Attribut**  $\mapsto$  Klassen-Variable
  - „ID-Zähler“

### Attribute der Klasse Rechteck

```
public class Rechteck {  
    int breite;  
    int hoehe;  
    int id;  
    static int zaehler; // da müssen wir noch was tun...  
}
```

# Konstruktoren

- **Konstruktor**: spezielle Methode
  - wird beim **Erzeugen neuer Objekte** aufgerufen
  - enthält häufig Code zum **Initialisieren** der Attribute
- in Java:
  - Konstruktor muss **genauso heißen** wie die Klasse
  - Konstruktoren haben **keinen Rückgabotyp** (auch nicht void)
  - Konstruktoren können beliebige Liste an **Parametern** haben

## Einschub: Überladen

- Überladen (*Overloading*):
  - mehrere Methoden mit demselben Namen aber unterschiedlicher Signatur
    - Methoden müssen sich in Anzahl/Typen der Parameter unterscheiden
    - unterschiedliche Parameternamen und/oder unterschiedlicher Rückgabetyt alleine reicht nicht

### Beispiel

```
public static int mul(int a, int b) { return a * b; }  
public static float mul(float a, float b) { return a * b; }
```

- im Kontext von Konstruktoren:
  - Namen von Konstruktoren sind durch Name der Klasse vorbestimmt  
~ durch Überladen mehrere Konstruktoren möglich

## Standardkonstruktor in Java

- kein eigener Konstruktor  $\leadsto$  impliziter Standardkonstruktor
  - keine Parameter
  - Initialisierung der Attribute auf Standardwerte
- sobald eigener Konstruktor definiert wird:
  - Standardkonstruktor nicht mehr vorhanden
  - muss bei Bedarf explizit wieder definiert werden

## Konstruktor der Klasse Rechteck

- Konstruktor im Beispiel:
  - Initialisierung von Breite und Höhe auf Parameterwerte
  - Setzen der ID und Erhöhen des ID-Zählers

### Konstruktor der Klasse Rechteck

```
public class Rechteck {  
    // ...  
  
    public Rechteck(int breite, int hoehe) {  
        // Zugriff auf Objekt-Attribute mittels 'this'  
        this.breite = breite;  
        this.hoehe = hoehe;  
        this.id = ++zaehler;  
    }  
}
```

## Klassen-Methode $\leftrightarrow$ Instanz-Methode

- Instanz-Methode:
  - arbeitet auf einem Objekt
  - kann auf Objekt- und Klassen-Attribute zugreifen

~> zum Aufruf wird ein konkretes Objekt benötigt
- Klassen-Methode (auch statische Methode):
  - arbeitet auf der Klasse als Ganzes
  - kann nur auf Klassen-Attribute zugreifen

~> zum Aufruf wird kein Objekt der Klasse benötigt

## Methoden

- **Instanz-Methode**: Fläche berechnen
  - benötigt Zugriff auf Attribute eines konkreten Objekts
- **Klassen-Methode**: Rechteck drehen
  - bekommt zu drehendes Rechteck als Parameter
  - könnte auch als Instanz-Methode implementiert werden!
    - Entscheidung für Klassen-Methode hat hier eher didaktische Gründe 😊

### Methoden der Klasse Rechteck

```
public class Rechteck {  
    // ...  
    public int flaeche() { return breite * hoehe; }  
    public static Rechteck drehen(Rchteck orig) {  
        return new Rechteck(orig.hoehe, orig.breite);  
    }  
}
```



## Qualifizierter Zugriff

- wenn „globale“ Variablen genauso heißen wie lokale Variablen:
  - expliziter Zugriff auf Objekt-Variable: `this`
  - expliziter Zugriff auf Klassen-Variable: `Klassenname`
- auch aus Gründen der **Lesbarkeit** zu empfehlen 😊

### Qualifizierter Zugriff

```
public class Rechteck {  
    // ...  
    public Rechteck(int breite, int hoehe) {  
        this.breite = breite;  
        this.hoehe = hoehe;  
        this.id = ++Rechteck.id;  
    }  
    public int flaeche() { return this.breite * this.hoehe; }  
}
```

## Getter und Setter

Zur Erinnerung: Datenkapselung meint...

- Verbergung von Implementierungs-Details
- Zugriff auf interne Daten nur über definierte **Schnittstellen**

→ kontrollierter Zugriff auf Objekt-Attribute:

- **Getter/Accessor**: (*Name per Konvention*: `getAttribut()`)
  - Lesen eines Objekt-Attributs
- **Setter/Mutator**: (*Name per Konvention*: `setAttribut()`)
  - Schreiben eines Objekt-Attributs
  - Möglichkeit zur Überprüfung des zu schreibenden Werts

## Getter und Setter der Klasse Rechteck (1)

- ID eines Rechtecks:
  - Getter zum Lesen des Werts
  - kein Setter (soll nicht geschrieben werden)
- Breite und Höhe eines Rechtecks:
  - Getter zum Lesen des entsprechenden Werts
  - Setter zum Setzen des entsprechenden Werts
    - dabei Überprüfung, ob zu schreibender Wert  $\geq 0$  ist

## Getter und Setter der Klasse Rechteck (2)

### Getter und Setter der Klasse Rechteck

```
public class Rechteck {  
    // ...  
    public int getID() { return this.id; }  
    public int getBreite() { return this.breite; }  
    public int getHoehe() { return this.hoehe; }  
  
    public void setBreite(int breite) {  
        if (breite >= 0) { this.breite = breite; }  
    }  
    public void setHoehe(int hoehe) {  
        if (hoehe >= 0) { this.hoehe = hoehe; }  
    }  
}
```

### Noch keine richtige Kapselung

Man kann die Attribute „von außen“ immer noch direkt lesen und schreiben, d.h. ohne die Getter und Setter zu verwenden.

## Einschub: Sichtbarkeiten

- Attribute und Methoden können **Sichtbarkeits-Modifizierer** haben
- ~> Von wo aus darf darauf zugegriffen werden?
- **keiner** (*paketsichtbar*): aus den Klassen desselben Pakets
  - **public**: aus allen Klassen heraus
  - **private**: nur aus definierender Klasse heraus
  - **protected**: aus definierender Klasse, allen Unterklassen, und Klassen desselben Pakets (leider...)

### Für unsere Zwecke

Attribute der Klasse Rechteck mit dem Modifizierer **private** deklarieren, um unkontrollierten Zugriff von außerhalb zu verhindern.

# Sichtbarkeiten der Attribute von Rechteck

## Sichtbarkeiten der Attribute von Rechteck

```
public class Rechteck {  
    private int breite;  
    private int hoehe;  
    private int id;  
    private static int zaehler;  
  
    // ...  
}
```

# Fertige Klasse Rechteck...

## Fertige Klasse Rechteck

```
public class Rechteck {
    private int breite;
    private int hoehe;
    private int id;
    private static int zaehler;

    public Rechteck(int breite, int hoehe) {
        this.setBreite(breite);
        this.setHoehe(hoehe);
        this.id = ++zaehler;
    }

    public int getID() { return this.id; }
    public int getBreite() { return this.breite; }
    public int getHoehe() { return this.hoehe; }

    public void setBreite(int breite) { if (breite >= 0) this.breite = breite; }
    public void setHoehe(int hoehe) { if (hoehe >= 0) this.hoehe = hoehe; }

    public int flaeche() { return this.breite * this.hoehe; }
    public static Rechteck drehen(Rchteck orig) {
        return new Rechteck(orig.hoehe, orig.breite);
    }
}
```

## ...und ihre Verwendung

### Ohne Objektorientierte Programmierung

```
// ein Rechteck habe eine Breite, eine Hoehe und eine "ID"

// zwei Rechtecke A und B...
int breiteA = 13, hoeheA = 3, idA = 1;
int breiteB = 12, hoeheB = 11, idB = 2;

// ...und deren Flaechen
int flaecheA = breiteA*hoeheA;
int flaecheB = breiteB*hoeheB;

// Rechteck C entsteht durch Drehung von Rechteck A
int breiteC = hoeheA, hoeheC = breiteA, idC = 3;
```

### Mit Objektorientierter Programmierung

```
Rechteck a = new Rechteck(13, 3);
Rechteck b = new Rechteck(12, 11);

int flaecheA = a.flaeche();
int flaecheB = b.flaeche();

Rechteck c = Rechteck.drehen(a);
```



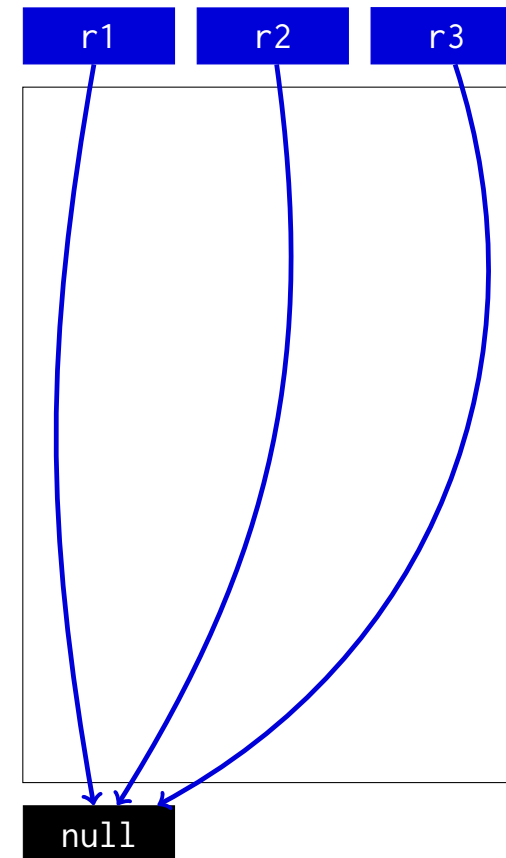
# Heap

- Objekt-Erzeugung  $\leadsto$  Speicher reservieren/allozieren
  - Platz für Objekt-Attribute und Verwaltungs-Informationen
  - Speicher dafür liegt im sog. Heap
  - die Laufzeitumgebung findet eine geeignete Speicherstelle
- Ergebnis einer Objekt-Erzeugung ist eine Referenz
  - Information, wo dazugehöriger Speicher im Heap liegt  $\leadsto$  „Adresse“

# Referenzen

## Code

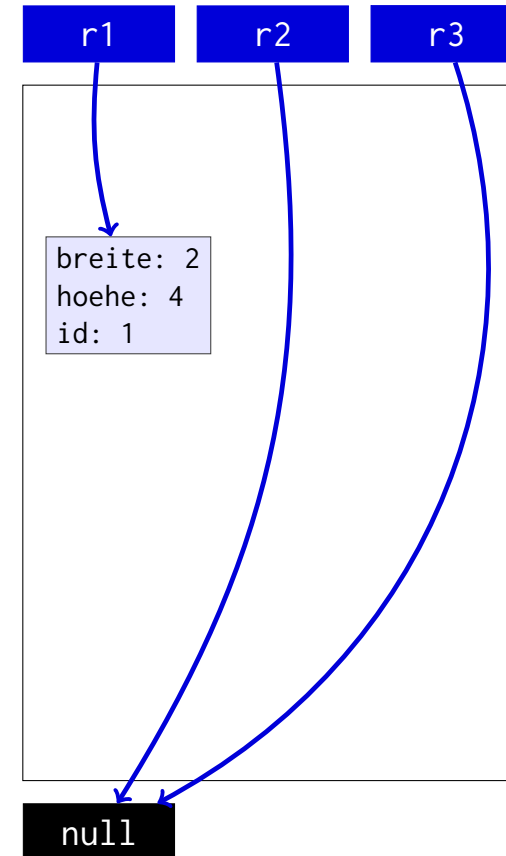
```
Rechteck r1;  
Rechteck r2;  
● Rechteck r3;  
r1 = new Rechteck(2,4);  
r3 = new Rechteck(7,6);  
r2 = r1;  
r2.setBreite(5);  
r1.getBreite();
```



# Referenzen

## Code

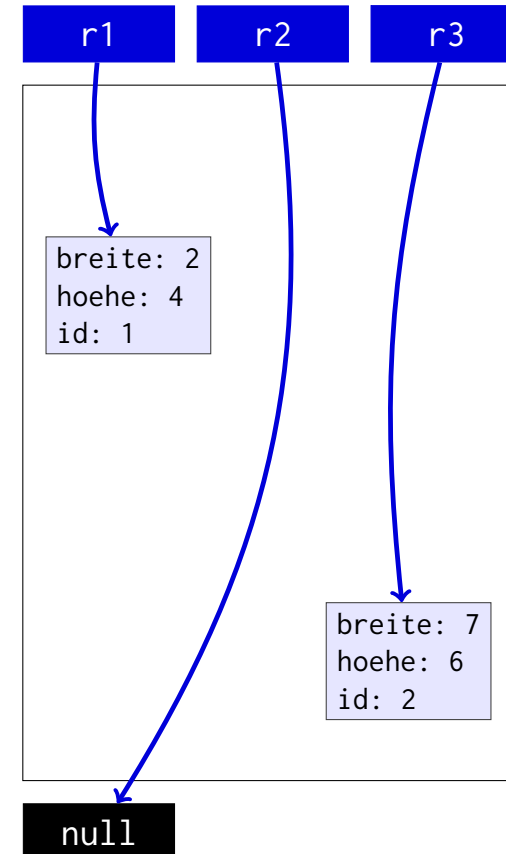
```
Rechteck r1;  
Rechteck r2;  
Rechteck r3;  
● r1 = new Rechteck(2,4);  
  r3 = new Rechteck(7,6);  
  r2 = r1;  
  r2.setBreite(5);  
  r1.getBreite();
```



# Referenzen

## Code

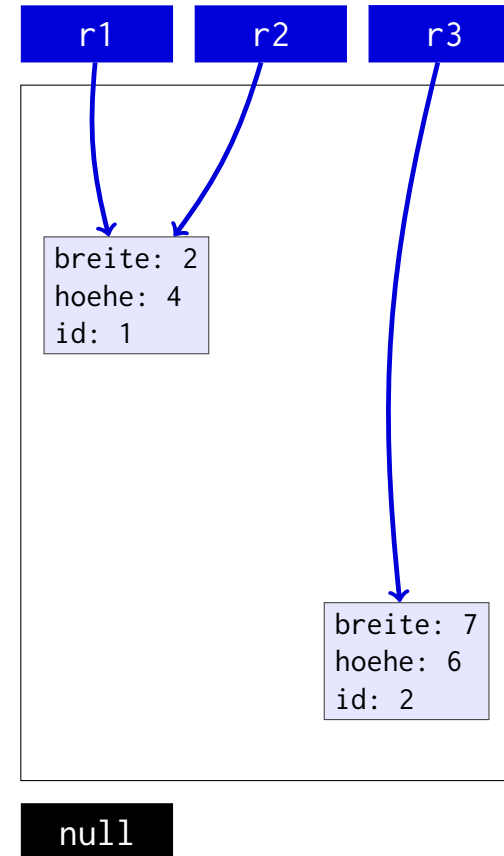
```
Rechteck r1;  
Rechteck r2;  
Rechteck r3;  
r1 = new Rechteck(2,4);  
● r3 = new Rechteck(7,6);  
r2 = r1;  
r2.setBreite(5);  
r1.getBreite();
```



## Referenzen

### Code

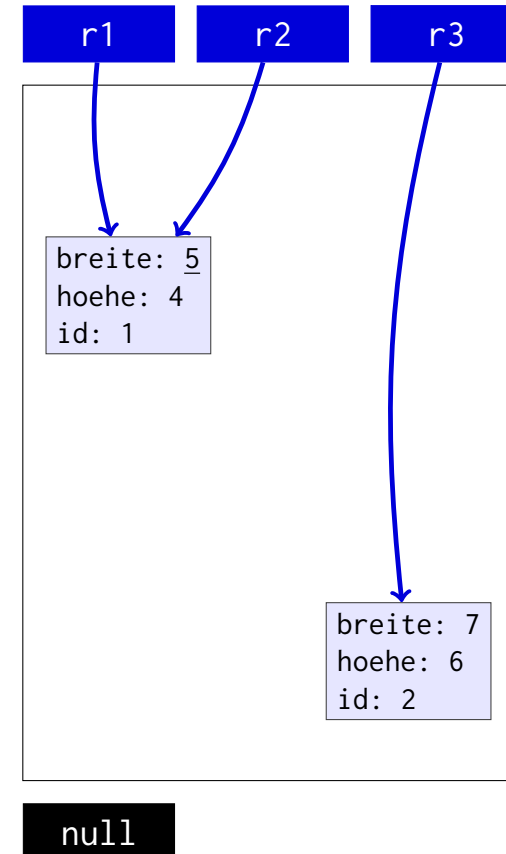
```
Rechteck r1;  
Rechteck r2;  
Rechteck r3;  
r1 = new Rechteck(2,4);  
r3 = new Rechteck(7,6);  
● r2 = r1;  
r2.setBreite(5);  
r1.getBreite();
```



# Referenzen

## Code

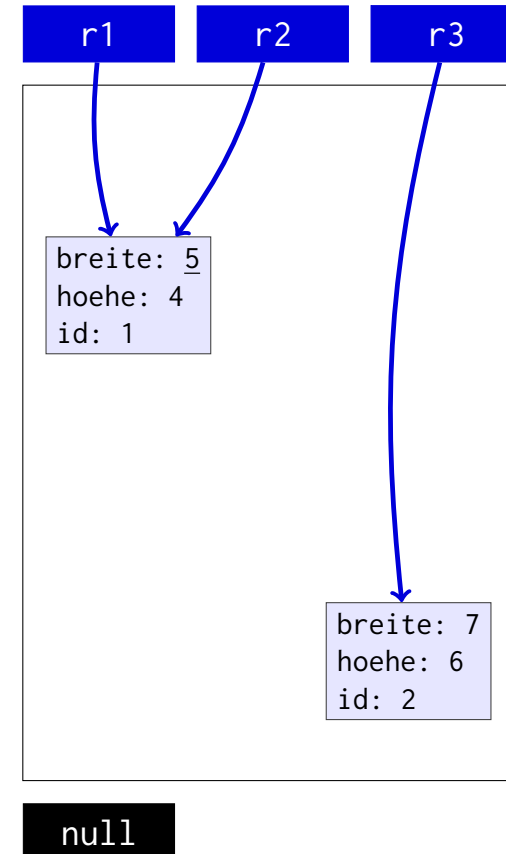
```
Rechteck r1;  
Rechteck r2;  
Rechteck r3;  
r1 = new Rechteck(2,4);  
r3 = new Rechteck(7,6);  
r2 = r1;  
● r2.setBreite(5);  
  r1.getBreite();
```



# Referenzen

## Code

```
Rechteck r1;  
Rechteck r2;  
Rechteck r3;  
r1 = new Rechteck(2,4);  
r3 = new Rechteck(7,6);  
r2 = r1;  
r2.setBreite(5);  
● r1.getBreite();
```



## Vergleich von Objekten

### Vergleich mittels ==-Operator

Der ==-Operator vergleicht die **Referenzen** miteinander, nicht die „Inhalte“ der Objekte, d.h. das Ergebnis ist nur dann true, falls beide Operanden auf **ein und dasselbe Objekt** im Speicher zeigen.

### Inhaltlicher Vergleich von Objekten

Sollen Objekte **inhaltlich** miteinander verglichen werden, *muss* die Methode `equals()` geeignet implementiert werden.



# Vergleich von Rechtecken

## Methode equals() der Klasse Rechteck

```
public class Rechteck {  
    // ...  
  
    public boolean equals(Object anderes) {  
        if (!anderes instanceof Rechteck)  
            return false;  
  
        Rechteck anderesRechteck = (Rechteck) anderes;  
        return anderesRechteck.breite == this.breite &&  
            anderesRechteck.hoehe == this.hoehe;  
    }  
}
```

# Automatic Garbage Collection

- Heap hat nur **begrenzte Größe**  
~> nicht mehr benötigte Objekte müssen wieder **freigegeben** werden
- Objekt-Lebenszeit unabhängig von der Lebenszeit der erzeugenden Funktion  
~> kein einfaches Aufräumen wie beim Stack möglich
- in Java: **automatische Speicherbereinigung**
  - nicht mehr benötigte Objekte werden aus dem Heap entfernt
    - in unregelmäßigen Abständen durch das Laufzeitsystem
    - unreferenzierte Objekte werden identifiziert und „ausgekehrt“
  - Programmierer muss sich nicht um das Freigeben kümmern
- ~> (quasi) **keine Speicherlecks** möglich

# Debugging

Lehrstuhl Informatik 2  
Programmiersysteme



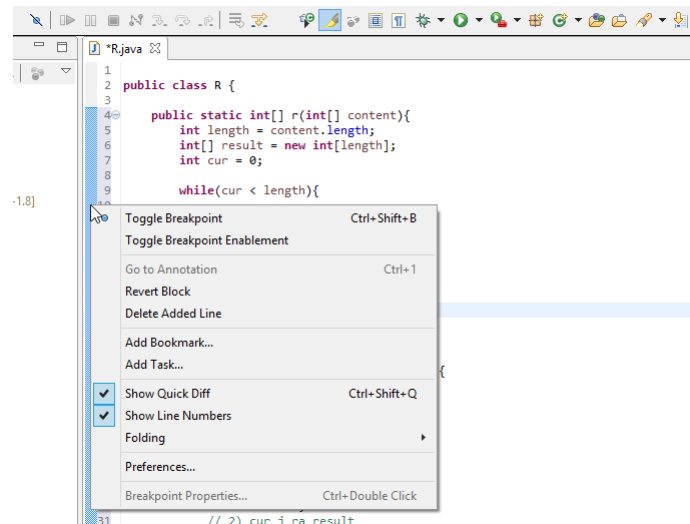
FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

# Debugging

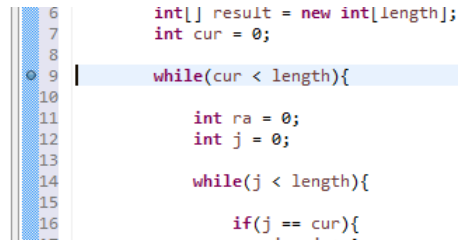
- als **Debugger** bezeichnet man eine bestimmte Art von Werkzeug
  - erleichtert Suche nach Programmierfehlern („**Bugs**“)
- Programm wird **schrittweise** ausgeführt  
~> Variablenwerte können jederzeit inspiziert werden
- mithilfe eines Debuggers kann ein Programmierer nachvollziehen...
  - ...**was** das Programm wirklich tut
  - ...mit **welchen Werten** das Programm an welchen Stellen arbeitet
  - ...**wann** das erste Mal ein evtl. falscher Wert auftritt

# 1. Haltepunkte setzen (I)



- vor Anweisungen mit **Haltepunkt**: Programm wird beim Debuggen pausiert
- in Eclipse:
  - Klick mit der rechten Maustaste auf die Leiste neben dem Code-Editor in der gewünschten Zeile
  - Auswahl von „Toggle Breakpoint“ im Menu

# 1. Haltepunkte setzen (II)



```

6      int[] result = new int[length];
7      int cur = 0;
8
9      while(cur < length){
10
11          int ra = 0;
12          int j = 0;
13
14          while(j < length){
15              if(j == cur){
16

```

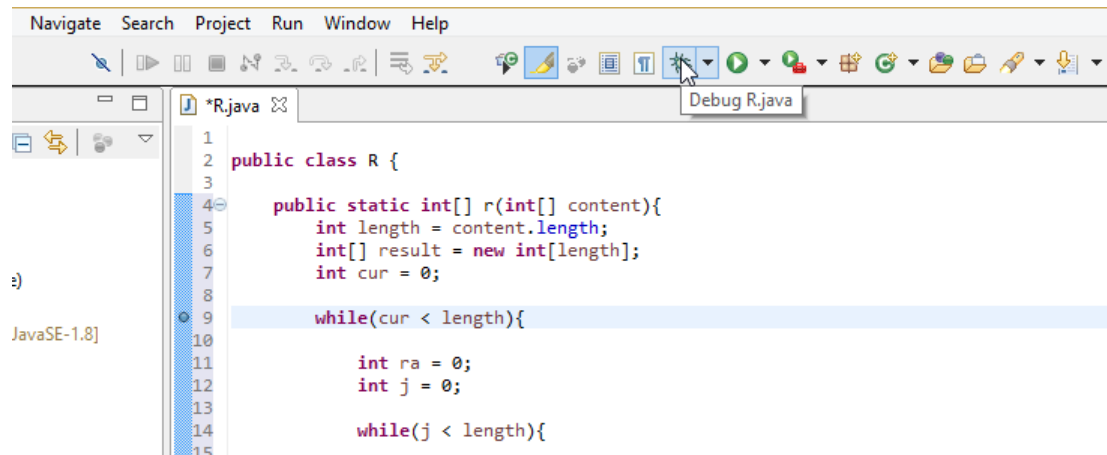
- ein blau ausgefüllter Punkt zeigt den aktivierten Haltepunkt an

## Debuggen von JUnit-Tests

zusätzlich:

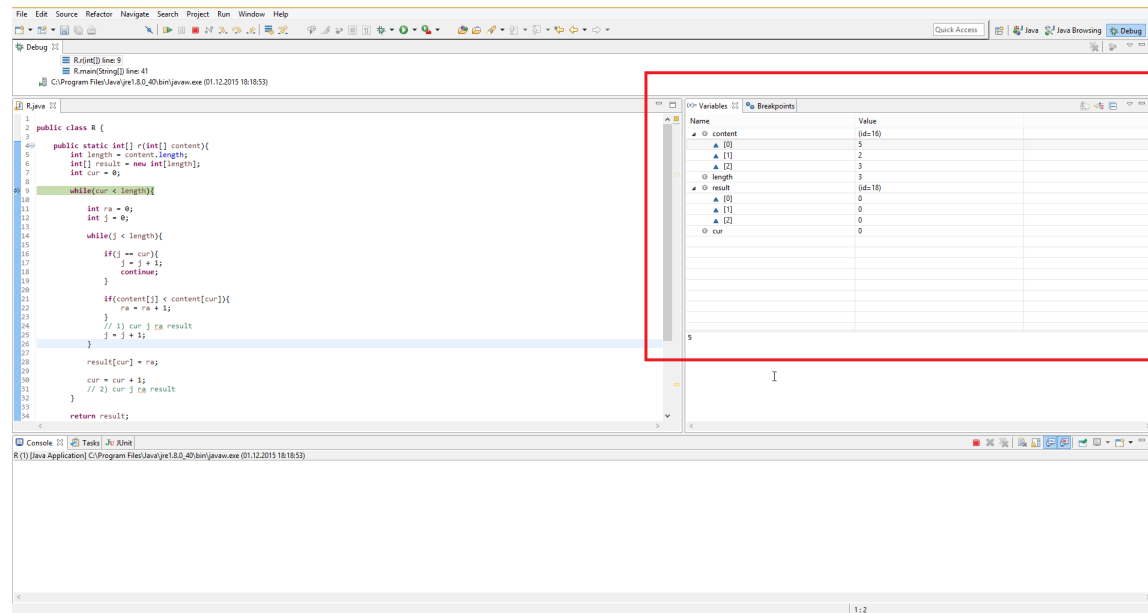
- Rechtsklick auf blauen Punkt → Breakpoint Properties
- „Suspend VM“ aktivieren (sonst: Probleme mit Timeout)

## 2. Debugger starten



- Klick auf das Symbol mit dem „Bug“ neben dem normalen Ausführungssymbol
- ggf. Meldung mit „Yes“ bestätigen, falls gefragt wird, ob „Debug-Perspektive“ geöffnet werden soll

### 3. Variablenfenster ansehen (I)



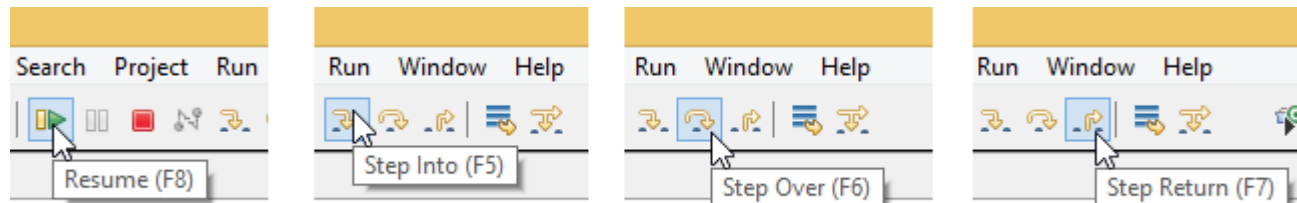
- das Programm wird nun bis zum Erreichen eines Haltepunktes ausgeführt
- die grüne Zeile stellt die nächste Anweisung in der Abfolge da
- neben dem Programmcode erscheint ein neues Fenster, in dem Werte der Variablen und Referenzen angezeigt werden



[illegible]

- bei primitiven Datentypen: Wert steht direkt im Fenster
- bei Referenzvariablen und Arrays: Werte können mit einem Klick auf den Pfeil ausgeklappt und eingesehen werden

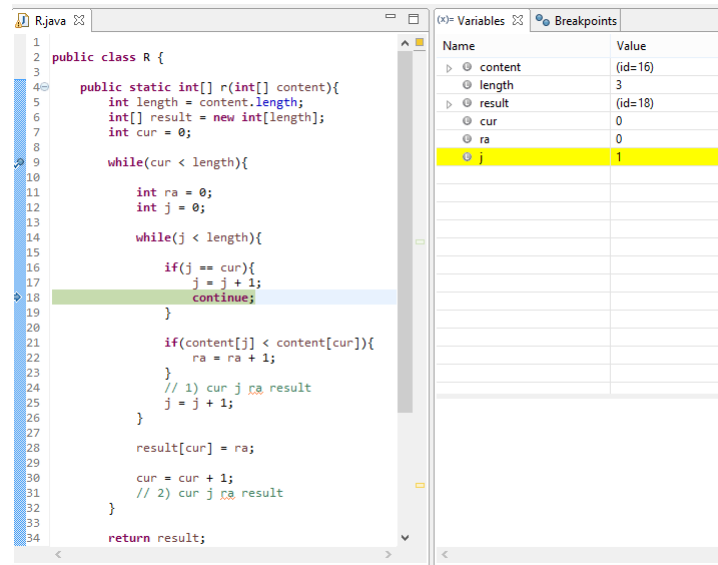
## 4. Das Programm durchlaufen



Möglichkeiten:

- **Resume:** Das Programm wird bis zum nächsten Haltepunkt fortgesetzt.
- **Step Into:** Die nächste Anweisung wird ausgeführt. Ist die aktuelle Anweisung ein Methodenaufruf, wird in diese Methode gesprungen.
- **Step Over:** Die nächste Anweisung in der Methode ausgeführt. Wenn die Anweisung ein Methodenaufruf ist, wird die Methode ausgeführt, aber das Programm erst nach dem Rücksprung wieder angehalten.
- **Step Return:** Die Ausführung wird bis zum Rücksprung aus der aktuellen Methode fortgesetzt und erst hier wieder angehalten.

## 5. Variablen beobachten und Fehler finden



The screenshot shows the RJava IDE with a Java program on the left and a variable watch window on the right. The program is a recursive function to calculate the nth Fibonacci number. The variable watch window shows the current state of the program's variables.

```

1 public class R {
2
3
4     public static int[] r(int[] content){
5         int length = content.length;
6         int[] result = new int[length];
7         int cur = 0;
8
9         while(cur < length){
10
11             int ra = 0;
12             int j = 0;
13
14             while(j < length){
15
16                 if(j == cur){
17                     j = j + 1;
18                     continue;
19                 }
20
21                 if(content[j] < content[cur]){
22                     ra = ra + 1;
23                 }
24                 // 1) cur j ra result
25                 j = j + 1;
26             }
27
28             result[cur] = ra;
29
30             cur = cur + 1;
31             // 2) cur j ra result
32         }
33
34         return result;
35     }
36 }

```

Name	Value
> content	(id=16)
> length	3
> result	(id=18)
> cur	0
> ra	0
> j	1

- bei jedem Stopp die Werte der Variablen betrachten (Änderungen hier gelb).
- feststellen, **wann** und **wieso** diese von ihrem Soll-Wert abweichen.

# Fragen? Fragen!

(hilft auch den anderen)

Lehrstuhl Informatik 2  
Programmiersysteme



FRIEDRICH-ALEXANDER  
UNIVERSITÄT  
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT