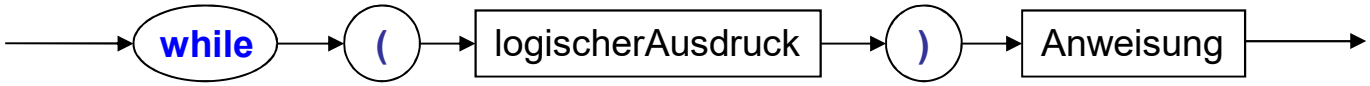
Kopfgesteuerte Schleife

kopfgesteuerteSchleife



- Bei Zählschleifen muss die Anzahl der Wiederholungen zu Beginn der Schleife feststehen
- Dies ist aber nicht immer möglich

- Beispielproblem: suche in einem Telefonbuch einen bestimmten Eintrag
- Lösung: gehe das Telefonbuch von vorne nach hinten durch. Breche die Suche ab, sobald der gerade betrachtete Eintrag mit dem gesuchten Namen übereinstimmt

- Deshalb nötig: Iterationsform mit einer Abbruchbedingung
- Vorsicht: Abbruchbedingung sollte irgendwann auch zum Abbruch der Schleife führen

Beispiel ggT

```
ggT(x,y):
Falls x = 0 ist, dann ist y das Ergebnis
ansonsten
    wiederhole, solange y ≠ 0 gilt
        falls x > y ersetze x durch x - y
        ansonsten   ersetze y durch y - x
    x ist das Ergebnis
```

$$ggT(x,y) := \begin{cases} x & falls\ y = 0 \\ y & falls\ x = 0 \\ ggT(x-y,y) & falls\ x > y \\ ggT(x,y-x) & sonst \end{cases}$$

```
public class ggT  {
    public static void main (String[] args)  {
        int x = 43158, y = 26364;         // Beispielwerte
        if(x == 0) {
            // gebe das Ergebnis / den ggT aus
            System.out.println (y);
        } else {
            while (y != 0)  {
                if (x > y) {
                    x = x - y;
                } else {
                    y = y - x;
                }
            }
            // das Verfahren brach ab und in x steht das Resultat
            System.out.println (x);
        }
    }
}
```

Beispiel: Berechnung Quadratwurzel

- Problem: berechne zu einem Wert x $\in$ R den Quadratwurzelwert y=$\sqrt{x}$
- Eine Lösung: Verfahren von Heron
- Wende wiederholt folgende Formel an: $$y_{n+1} = \frac{y_n + \frac{x}{y_n}}{2}$$
- Es gilt: $\lim \xrightarrow{n\to\infty} y_n = \sqrt{x}$

- Der Startwert y_0 kann ein beliebiger Wert größer als 0 sein, beispielsweise y_0=x
- Hier kein Beweis, dass
  - das Verfahren für beliebiges x≥0 korrekt den Wert $\sqrt{x}$ berechnet
  - das Verfahren für jeden Startwert $y_0 > 0$ konvergiert

- Wir wenden wiederholt die Formel an und brechen die Berechnung ab, wenn wir "nahe genug" an den korrekten Wert gelangt sind
- Das ist der Fall, wenn $|y_n^2 - x| < \varepsilon$ für ein ε unserer Wahl

Programm zum Heron-Algorithmus

```
public class Heron {
    public static void main(String[] args) {
        double x = 8351.0;              // Beispielwert
        double epsilon = 0.000001;      // gewuenschte Genauigkeit
        double fehler;                  // Fehlerabschaetzung
        double y;                       // Annaeherungswert y_n

        // beliebigen Startwert angeben (probieren Sie es aus!)
        y = x;

        // Absolutbetrag fuer Fehlerabschaetzung
        fehler = Math.abs(x - y*y);

        while(fehler > epsilon) {
            // Berechnungsvorschrift fuer bessere Naeherung
            y = (y + x/y) / 2.0;
            // Absolutbetrag fuer Fehlerabschaetzung
            fehler = Math.abs(x - y*y);
            // wir geben zur Kontrolle die derzeitige Naeherung aus
            System.out.println("y=" + y);
        }
    }
}
```

Ausgabe:

```
y=4176.0
y=2088.9998802681994
y=1046.4987435038079
y=527.2393430977079
y=271.53922469803194
y=151.14676459891066
y=103.1988495800834
y=92.06014714295677
y=91.38629045342645
y=91.38380603899313
y=91.38380600522174
```

Zwischenstand

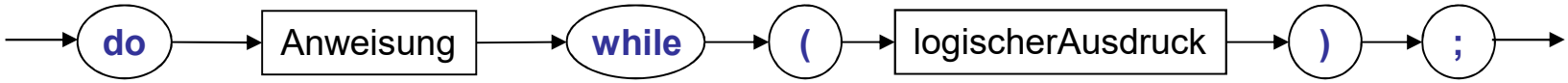
- In einer kopfgesteuerten Schleife ergibt sich die Anzahl der Iterationen / das Ende der Schleife erst während der Schleifenausführung über das Abbruchkriterium
- Damit sind jetzt auch Schleifen möglich, die niemals enden

Reflektion

- Könnte man das Heron-Verfahren (ohne das Verfahren an sich zu verändern) auch mit Hilfe einer Zählschleife angeben? Wenn ja, wie? Wenn nein, wieso nicht?

Fußgesteuerte Schleife

fußgesteuerteSchleife



- Das letzte Beispiel (Heron) zeigte, dass es manchmal sinnvoll sein kann, die Abbruchbedingung am Ende der Schleife auszuführen
- Deshalb: fußgesteuerte Schleife

- Wichtiger Unterschied zur kopfgesteuerten Schleife: der Schleifenrumpf (die Anweisung) wird auf jeden Fall einmal ausgeführt!

Heron Version 2

```
public class Heron {
    public static void main(String[] args) {
        double x = 8351.0;              // Beispielwert
        double epsilon = 0.000001;      // gewuenschte Genauigkeit
        double fehler;                  // Fehlerabschaetzung
        double y;                       // Annaeherungswert y_n

        // beliebigen Startwert angeben (probieren Sie es aus!)
        y = x;

        do {
            // Berechnungsvorschrift fuer bessere Naeherung
            y = (y + x/y) / 2.0;
            // Absolutbetrag fuer Fehlerabschaetzung
            fehler = Math.abs(x - y*y);

            // wir geben zur Kontrolle die derzeitige Naeherung aus
            System.out.println("y=" + y);
        } while(fehler > epsilon);
    }
}
```

Unterschiede zur ersten Version:

- Die Fehlerabschätzung wird nur noch an einer Stelle berechnet
- Die y-Formel wird mindestens einmal ausgeführt (wieso ist das kein Problem?)

Alle drei Iterationsformen ineinander überführbar?

- Zählschleife → kopfgesteuerte Schleife (umgekehrt nicht allgemein angebbar!)

```
for(Initialisierung; Test; Update)
    Anweisung
```

```
Initialisierung;
while(Test) {
    Anweisung
    Update;
}
```

- kopfgesteuerte Schleife → fußgesteuerte Schleife

```
while(Test)
    Anweisung
```

```
if(Test) {
    do {
        Anweisung
    } while(Test);
}
```

- fußgesteuerte Schleife → kopfgesteuerte Schleife

```
do
    Anweisung
while(Test);
```

```
Anweisung;
while(Test) {
    Anweisung
}
```

Lösungsansatz: sei gierig

- Aufgabe: gebe Wechselgeld zurück mit der kleinsten Anzahl an Münzen
  - Geldmünzen sind: 200,100,50,20,10,5,2,1 Cent
  - Zu zahlender Betrag ist 8,47 Euro
  - Hingelegt wurden 10 Euro

- Gieriger Ansatz (engl. greedy): führe immer den nächsten Schritt aus, der bei diesem Schritt am meisten Profit bringt
- Hier angewandt: solange das verbleibende Rückgeld ungleich 0 ist
  - nehme die größte mögliche Münze und lege diese hin
  - reduziere das verbleibende Rückgeld um diesen Betrag

- Beispiel: verbleibendes Rückgeld: 10 Euro - 8,47 Euro = 153 Cent
  - Münze: 100 Cent, verbleibendes Rückgeld 53
  - Münze: 50 Cent, verbleibendes Rückgeld 3
  - Münze: 2 Cent, verbleibendes Rückgeld 1
  - Münze: 1 Cent, verbleibendes Rückgeld 0

Zwischenstand

- In einer fußgesteuerten Schleife ergibt sich die Anzahl der Iterationen / das Ende der Schleife ebenfalls erst während der Schleifenausführung über das Abbruchkriterium
- Damit sind ebenfalls Schleifen möglich, die niemals enden
- In einer fußgesteuerten Schleife wird der Schleifenrumpf mindestens einmal durchlaufen

Reflektion

- Überlegen Sie sich ein (praktisches) Beispiel, bei dem eine fußgesteuerte Schleife vorteilhaft wäre gegenüber einer kopfgesteuerten Schleife?

Abweichung von der normalen Schleifenausführung

- Manchmal ist es nützlich, wenn man innerhalb des Schleifenrumpfs den Rest des Schleifenrumpfs ignorieren kann / die Schleife abbrechen kann

- `continue;` ignoriere Rest des Schleifenrumpfs. Anstatt den Schleifenrumpf weiter auszuführen, mache sofort mit Schleifentest weiter (kopf-/fußgesteuert) bzw. fahre bei einer Zählschleife mit Update-Operation fort.
- `break;` in einem Schleifenrumpf: breche innerste Schleife ab (zur Erinnerung: weitere Anwendung von `break` war bereits in switch-Anweisung)

- Beispiel 1: lese Zahlen von der Tastatur ein und addiere diese, bis eine negative Zahl eingegeben wird
- Beispiel 2: lese 10 Zahlen von der Tastatur ein und berechne die Summe aller Quadratwurzelwerte dieser Zahlen bis auf die Eingabezahlen, die negativ sind

- Weiterführender Hinweis: `break` und `continue` gibt es auch mit Sprungmarken

Beispiel 1

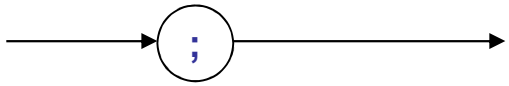
```
import java.util.*;
public class ZahlenEinlesen {
    public static void main(String[] args) {

        // Scanner von der Tastatur anlegen
        Scanner sc = new Scanner(System.in);
        int summe = 0;

        System.out.println("Geben Sie ganze Zahlen ein. -1 beendet Eingabe");
        while(true) {
            int zahl = sc.nextInt();
            if(zahl < 0)
                break;
            summe = summe + zahl;
        }

        // eingelesene Daten ausgeben
        System.out.println("Die Summe der Werte ist: " + summe);

        // Scanner abschliessen
        sc.close();
    }
}
```

Beispiel 2

```
import java.util.*;
public class ZahlenEinlesen {
    public static void main(String[] args) {

        // Scanner von der Tastatur anlegen
        Scanner sc = new Scanner(System.in);
        double summe = 0.0;

        System.out.println("Geben Sie 10 ganze Zahlen ein.");
        for(int i=0; i<10; i=i+1)
            double zahl = sc.nextDouble();
            if(zahl < 0.0)
                continue;
            summe = summe + Math.sqrt(zahl);
        }

        // eingelesene Daten ausgeben
        System.out.println("Die Summe der Wurzelwerte ist: " + summe);

        // Scanner abschliessen
        sc.close();
    }
}
```

Leere Anweisung

LeereAnweisung

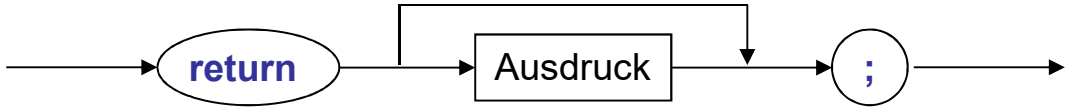


- An manchen Stellen ist syntaktisch genau eine Anweisung verlangt
- Hat man nichts Sinnvolles zu tun, so kann man an diesen Stellen die leere Anweisung einsetzen
- Alternative: leerer Block `{ }`
- Beispiel:

```
/**
 * ermittle die erste natuerliche Zahl groesser als 1000, die von 1
 * beginnend sich jeweils durch Verdopplung der Vorgaengerzahl ergibt
 */
public class LeereAnweisung {
    public static void main(String[] args) {
      int zahl;
      for(zahl=1; zahl<=1000; zahl=zahl+zahl)
          // hier muss nichts mehr getan werden
          ;
      System.out.println("Die gesuchte Zahl ist " + zahl);
    }
}
```

return Anweisung

returnAnweisung



- Nur im Zusammenhang mit Methoden genutzt
- Wird später mit Methoden diskutiert

Lösungsstrategie: erschöpfendes Suchen

- Für manche Probleme gibt es keine effizienten Lösungen, die das Problem in polynomieller Zeit lösen
- Mögliche Lösungsstrategie: erschöpfende Suche im Lösungsraum

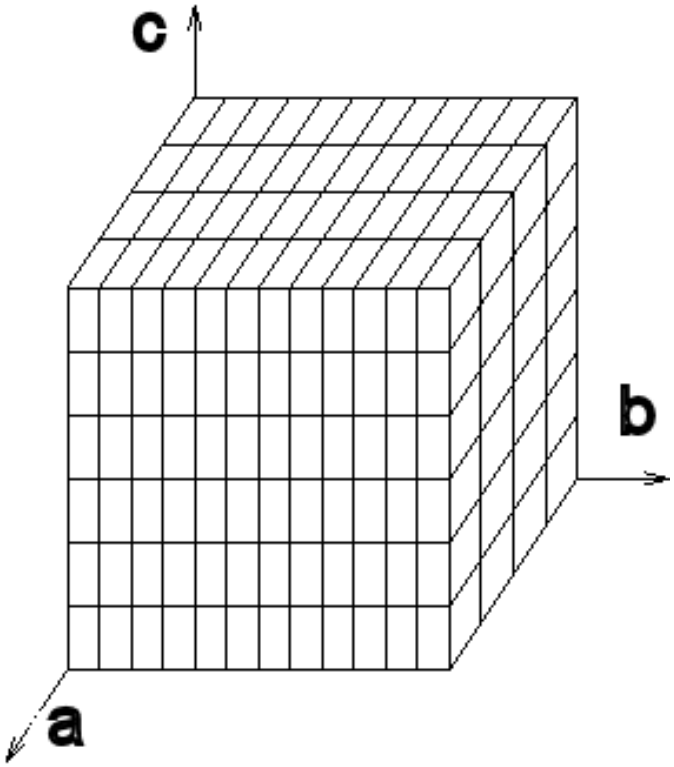
- Prinzip des Ansatzes anhand der Großen Fermatschen Satzes:
  Es gibt keine Zahlen $a,b,c,n \in N$, $a,b,c \geq 1$, $n > 2$, so dass $a^n+b^n=c^n$
- (Behauptung wurde 1993 von Andrew Wiles allgemein bewiesen)
- Hier: Entscheide diese Vermutung für $a,b,c,n \in N$, $a,b,c,n < max \in N$

- Vierdimensionaler Suchraum, charakteristischer Parameter ist max

- Mögliche Optimierung: nutze Symmetrien (z.B. $a^n+b^n=b^n+a^n$), um den Suchraum zu verkleinern

4-dimensionaler Suchraum

- Jeder diskrete Punkt im 4-dimensionalen Raum $[1,max]^4$ muss untersucht werden (n ab n>2)
- Hier aus anschaulichen Gründen nur für 3 Dimensionen gezeigt
- In welcher Reihenfolge die Punkte untersucht werden, spielt hier keine Rolle



Programm (ohne jegliche Optimierung)

```
public static void main(String[] args) {
    // lese Wert aus Kommandozeile ein, bis zu dem getestet werden soll
    long max = Long.parseLong(args[0]);
    boolean gefunden = false;

    for(long a=1; a<=max; a=a+1) {
        for(long b=1; b<=max; b=b+1) {
            for(long c=1; c<=max; c=c+1) {
                for(long n=3; n<=max; n=n+1) {
                    if(hoch(a,n) + hoch(b,n) == hoch(c,n)) {
                        System.out.println("gefunden: " + a + ", " + b + ", " + c + ", " + n);
                        System.out.println(hoch(a,n) + " + " + hoch(b,n) + " = " + hoch(c,n));
                        gefunden = true;
                        break;   // Frage: was bewirkt das hier und was nicht?
                    }
                }
            }
        }
    }
    if(!gefunden) {
        System.out.println("keine Loesung gefunden");
    }
}
```

Anmerkung: `hoch(x,y)` soll eine Bezeichnung sein für die Berechnung x^y

Frage: Was wird u.a. bei a,b,c,n=16 passieren? Wieso?

Zwischenstand

- In allen Schleifenformen lässt sich eine Iteration / die gesamte Schleife vorzeitig abbrechen
- Wo syntaktisch eine Anweisung verlangt ist, kann die leere Anweisung Sinn machen

Reflektion

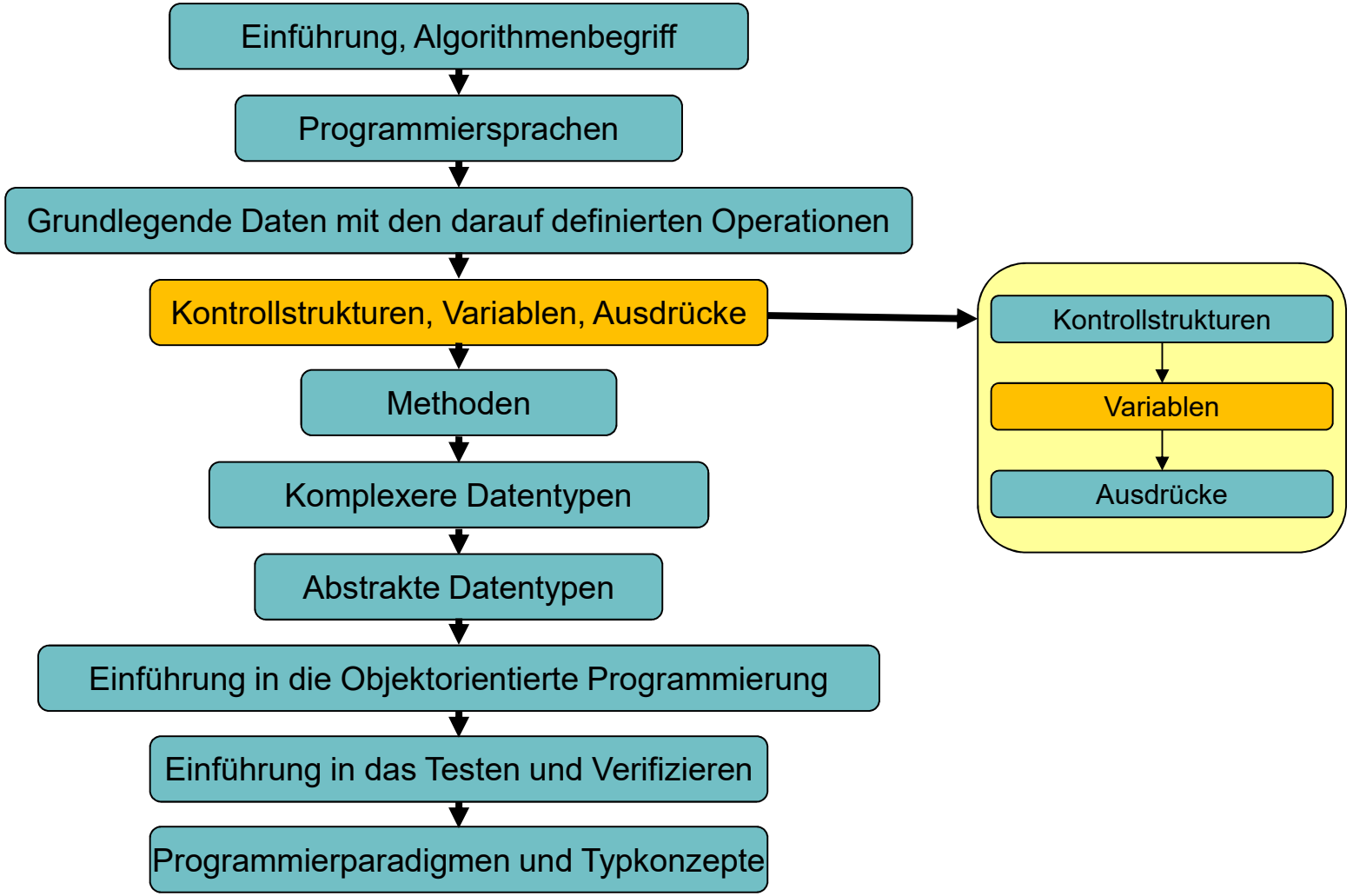
- Wie könnte man es erreichen, in einer zweifach geschachtelten Schleife aus dem innersten Schleifenrumpf heraus die gesamte Schleife abzubrechen?

Zusammenfassung

- Kontrollstrukturen sind Anweisungen, die den Ablauf bestimmen

- Selektion trifft Auswahl aus alternativen Anweisungen
- Iteration wiederholt Anweisung
- Block bündelt Anweisungen in Sequenzanweisung

Inhalt dieser Veranstaltung



Variablen genauer

- Variablen dienen der Zwischenspeicherung von Werten
  - um sie an späterer Stelle zu nutzen
  - um Mehrfachberechnungen zu vermeiden

- Eigenschaften von Variablen:
  - Name: jede Variable hat einen frei-wählbaren Namen, den man bei der Deklaration angibt
  - Typ: jede Variable hat genau einen Typ, den man bei der Deklaration angibt
  - Speicheradresse: jeder Variablen wird vom Compiler eine Hauptspeicheradresse zugewiesen. Unter dieser Adresse ist immer der aktuelle Wert der Variablen zu finden.
  - Wert: zu jedem Zeitpunkt hat eine Variable einen Wert, der sich aber im Laufe der Berechnung ändern kann (Zuweisung)

- In einer Deklaration wird also der Name, der Typ und implizit die Speicheradresse einer Variablen festgelegt. Wird kein Initialisierungsausdruck angegeben, so ist der Wert nicht definiert.

Gültigkeitsbereich

- Durch die Deklaration einer Variablen wird auch ihr Gültigkeitsbereich festgelegt
- Der Gültigkeitsbereich einer Variablen ist der Programmbereich, in dem diese Variable unter ihrem (einfachen) Namen bekannt ist
- Folgerung: der gleiche Name kann mehrfach in verschiedenen nicht-überlappenden Gültigkeitsbereichen verwandt werden (ist aber meist keine gute Idee!)

- Regeln derzeit:
  - Wird eine Variable in einem Block deklariert (blocklokale Variable), so ist der Gültigkeitsbereich dieser Variablen auf diesen Block beschränkt (inklusive innerer Blöcke)
  - Wird eine Variable in einer Zählschleife deklariert, so ist der Gültigkeitsbereich die gesamte Zählschleife (inklusive allem, was im Rumpf steht)

Beispiel

```
public class Gueltigkeitsbereich4 {
    public static void main(String[] args) { // Hier beginnt ein Block B1
        double d0 = 0.0;

        { // Hier beginnt ein innerer Block B2
            double d1 = 1.0;
            System.out.println("d1=" + d1);
        } // Hier endet der Block B2

        { // Hier beginnt ein weiterer innerer Block B3
            double d1 = 2.0;
            System.out.println("d1=" + d1);
        } // Hier endet der Block B3

        for(int i=0; i<3; i=i+1) {
            System.out.println(i);
        }

    }  // Hier endet der Block B1
}
```

Sichtbarkeit

- Wie gesehen, ist der gleiche Name in nicht-überlappenden Gültigkeitsbereichen kein Problem
- Was aber, wenn die Gültigkeitsbereiche sich überlappen?
- (Bei blocklokalen Variablen alleine kann diese Situation nicht auftreten)
- Dann kann eine Variable gültig, aber vorübergehend unsichtbar / verdeckt sein
- Um ein Beispiel dazu zu konstruieren, müssen wir etwas vorgreifen

```
public class Sichtbarkeit2 {

    static double d = 1.0;   // Name d Kategorie 1

    public static void main(String[] args) {
        {
            double d = 2.0;   // Name d Kategorie 2

            System.out.println("1.Stelle d=" + d);
        }

        System.out.println("2.Stelle d=" + d);
    }
}
```

Gültigkeitsbereich von d=1.0

Gültigkeitsbereich von d=2.0
In diesem Block ist d=1.0 zwar gültig, aber nicht sichtbar (wird verdeckt)

Lebensdauer

- Was passiert im folgendem Beispiel?

```
public class Lebensdauer {
    public static void main(String[] args) {

        for (int i=1; i < 5; i=i+1) {
            // hier beginnt ein Block mit einer Variablen j
            int j;
            j = i;
        }
    }
}
```

- Mit jedem Eintritt in diesen Block wird eine **neue** Variable j erzeugt!
- Block-lokale Variablen werden mit Eintritt in den Block erzeugt und sterben, wenn der Block verlassen wird
- Analog für Variablen, die in Zählschleifen deklariert werden
- Mit dem Sterben geht auch der Inhalt / Wert verloren
- Die Lebensdauer einer Variablen gibt den Teil der Programmausführung zwischen Erzeugen und Löschen dieser Variablen an

Konstanten

- Manche "Variablen" sollen ihren Wert nie ändern (Konstanten).
- Beispiel:
  `double PI = 3.1415;  // das ist so und wird immer so bleiben`
- Solche Variablen kann man in Java mit dem Zusatz `final` markieren.
- Vorteile:
  - Der Compiler überprüft, dass man nach der Initialisierung keine Veränderung mehr am Variablenwert vornimmt.
  - Der Compiler kann eventuell den generierten Maschinencode besser optimieren.
- Beispiel:

```
public class FahrenheitNachCelsiusTest {
    public static void main(String[] args) {
        final double faktor = 5.0 / 9.0;
        double fahrenheit = 72;
        double celsius = faktor * (fahrenheit - 32.0);
        System.out.println(fahrenheit + " Grad F sind in Celsius " + celsius);
    }
}
```

Zusammenfassung

- Jede Variable hat einen Namen, einen Typ, eine Speicheradresse und einen Wert, der sich ändern kann

- Im Zusammenhang mit Variablen sind wichtig:
  - Gültigkeitsbereich
  - Sichtbarkeit (evtl. Einschränkung des Gültigkeitsbereichs)
  - Lebensdauer