

Informatik 1

Klassenmethoden und -variablen

Inhalt

- Klassenmethoden
 - Syntax
 - Rückgabewert
 - Parameterübergabe
 - Laufzeitkeller (call stack)
- Klassenvariablen
 - Deklaration
 - Symbolische Konstanten (final)

Klassenmethoden

- Klassenmethoden fassen Programmanweisungen unter einen eindeutigen Namen zusammen.
- Methoden stellen in Java die kleinsten *wiederverwendbaren* Programmeinheiten dar.
- Der Methodenname muss eindeutig pro Klasse sein.
- Eine Methode wird durch Angabe des Methodennamens und Klasse mit Punkt-Operator dazwischen aufgerufen
 - Klasse kann weggelassen werden, wenn Aufruf in der Klasse der Methode steht
- Nach Aufruf werden die Anweisungen der Methode ausgeführt.
- Bei Aufruf können Werte als Parameter übergeben werden.
- Nach Beenden der Methode, kehrt das Programm zur ursprünglichen Stelle des Aufrufs zurück.
- Methoden müssen deklariert werden, um aufgerufen werden zu können.

```
Math.abs(a+ 1)
```

Aufruf einer Funktion abs,
die in der Klasse Math deklariert ist,
mit dem Ergebnis von a + 1 als Parameterwert

Klassenmethoden

- Deklaration innerhalb Klasse:

```
public class Klassenname {  
    Methodendeklarationen (keine, eine oder mehrere  
}
```

- Syntax Methodendeklaration

```
Modifier static Typ Bezeichner( Parameterliste ) {  
    // Programmanweisungen im Methodenrumpf  
}
```

- *Modifier*: **public** (private, protected), optional
- Typ: jeder Datentyp oder Schlüsselwort **void**
 - void: Methode gibt keine Funktionswert an aufrufende Stelle zurück
- Parameterliste darf fehlen

Klassenmethoden

- **public**
 - Die Methode kann aus anderen Klassen aufgerufen werden
 - Sie ist in allen anderen Klassen sichtbar
- **private**
 - Die Methode kann nur in der "eigenen" Klasse aufgerufen werden.
 - sie ist nur ihrer Klasse sichtbar.
- „nichts“:
 - Die Methode ist in allen Klassen eines Pakets sichtbar.
Vermeiden!
- **protected**
 - Paketsichtbarkeit und sichtbar bei Vererbung
- Informatik 1: nur public und private

Klassenmethoden

- Beispiel
 - „Hallo“ mehrfach auf dem Bildschirm ausgeben
 - Ausgabe in einer Methode programmieren
 - Von main-Methode in anderer Klasse aufrufen
- sehr einfaches, nicht reales Beispiel

```
public class Hallo {  
    public static void halloAusgeben() {  
        System.out.println("Hallo");  
    }  
}
```

```
public class HalloMain {  
    public static void main(String[] args) {  
        Hallo.halloAusgeben();  
        Hallo.halloAusgeben();  
    }  
}
```

Klassenmethoden

- Bei Aufruf einer Methode, können Werte, Parameter, gegeben als Ausdruck übergeben werden
- Die erlaubten Parameter einer Methoden, müssen mit Datentyp gefolgt von einem Namen **deklariert** werden
- Namenskonventionen von Parameter wie bei Variablen
- Mehrere Parameter werden mit , getrennt
- Die Parameternamen müssen eindeutig pro Methode sein
- Parameter dürfen nicht den selben Namen wie eine lokale Variable besitzen
- Innerhalb des Methodenrumpfs, kann über den Parameternamen auf die übergebenen Werte zugegriffen werden
- Syntax *Parameterliste* :

```
Typ1 Bezeichner1, Typ2 Bezeichner2, ... , Typ3 Bezeichner3
```

Klassenmethoden

```
public static Typ foo(Typ1 Bezeichner1, ... , TypN BezeichnerN) {  
}
```

- Anzahl übergebene Parameter muss bei Aufruf exakt überstimmen

```
Typ x;  
x = foo(Ausdruck1, ..., AusdruckN);
```

- Datentypen der Parameter müssen mit deklarierten übereinstimmen
 - Datentyp Ausdruck1 muss passend zu Typ1 sein, usw.
 - Datentyp von x muss passend zu Rückgabedatentyp Typ sein
 - ggf. wird eine widening conversion durchgeführt
- Ausdrücke werden vor Aufruf der Methode von links nach rechts ausgewertet

Klassenmethoden

- Beispiel
 - Funktion für Body-Mass-Index-Berechnung
 - Berechnung hängt von zwei Werten, Körpergröße und Gewicht ab
 - Berechneter Wert soll als Funktionswert zurückgegeben werden
- Parameter im Beispiel bewusst anders benannt
- Rückgabe des Funktionswerts mit **return**-Anweisung

```
public class BodyMassIndex {  
    public static double bodyMassIndexBerechnen(double laenge,  
                                                double masse ) {  
        double bodyMassIndex = 0.0;  
        bodyMassIndex = masse / (laenge * laenge);  
        return bodyMassIndex;  
    }  
}
```

Klassenmethoden

- Unnötige lokale Variablen weglassen
 - Zwischenergebnisse nur in Variablen speichern, wenn diese Variable mehr als einmal verwendet wird
- Rückgabewert bei einfachen Berechnungen nicht in einer lokalen Variablen speichern

```
public class BodyMassIndex {  
    public static double bodyMassIndexBerechnen(double laenge,  
                                                double masse ) {  
        return masse / (laenge * laenge);  
    }  
}
```

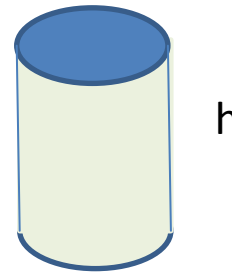
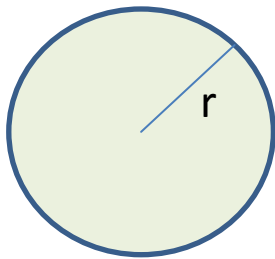
Klassenmethoden

- return-Anweisung
- Syntax:

```
return Ausdruck  
return;
```
- Nur in Methoden erlaubt
- return bricht die Methode immer ab
- Ausdruck muss zuweisungskompatibel zum deklarierten Rückgabewert sein
- return gibt den berechneten Wert des Ausdruck an die aufrufende Stelle zurück
- Bei Rückgabedatentyp void, kann return nur ohne Ausdruck verwendet werden (vermeiden)
- return ist wie jede Anweisung mehrfach erlaubt (vermeiden)
- return sollte immer am Ende einer Methode aufgerufen werden

Klassenmethoden

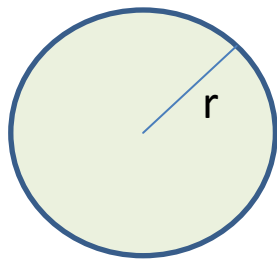
Gegeben:	Radius r Höhe h
Gesucht:	Kreisfläche mit Radius r Volumen Zylinder mit Radius r und Höhe h



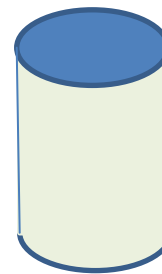
- Klasse mit zwei public-Funktionen
- Berechnungsvorschriften möglichst nur einmal programmieren (Redundanz vermeiden)

Klassenmethoden

Gegeben:	Radius r Höhe h
Gesucht:	Kreisfläche mit Radius r Volumen Zylinder mit Radius r und Höhe h



$$KF = \pi r^2$$



h

$$VZ = KF h$$

- Klasse mit zwei public-Funktionen
- Berechnungsvorschriften möglichst nur einmal programmieren (Redundanz vermeiden)
- Math.PI verwenden

Klassenmethoden

- Methoden mit unterschiedlichen Parameterdatentypen, dürfen gleichen Namen haben
 - Polymorphie (Vielgestaltigkeit) von Funktionen
- Beispiel
 - Klasse Math
 - Funktion `abs(Typ)` existiert vier mal: `int`, `long`, `float`, `double`
- Compiler sucht passende Methode anhand der Parameterdatentypen
 - Bei `foo(1, 2.0)` wird nach `foo(int, double)` gesucht, bei `foo(true, 2.0)` nach `foo(boolean, double)`
- Bei arithmetischen Datentypen wird ggf. eine widening conversion durchgeführt
 - Falls nur `foo(double, double)` existiert, wird bei `foo(1, 2.0)` eine widening conversion von `1` zu `1.0` durchgeführt und die entsprechende Methode aufgerufen
- Polymorphie von Operatoren (Überladung)
 - Binäre Addition `+` existiert in Java mehrfach

Klassenmethoden

- Mit Methoden können (getestete) Programme in anderen Programmen **wiederverwendet** werden.
- Funktionsorientierte Programmbibliothek:
 - Sammlung von Methoden, die zusammengehören
 - Beispiele: Klassen Math, Integer, Float
- Funktionsorientierte Programmierschnittstelle (application programming interface, API)
 - Teilmenge einer Programmbibliothek, die verwendet werden soll (oder darf).
- **Klassenmethoden in Java vermeiden und Objektmethoden verwenden (später)!**

Konventionen Methoden

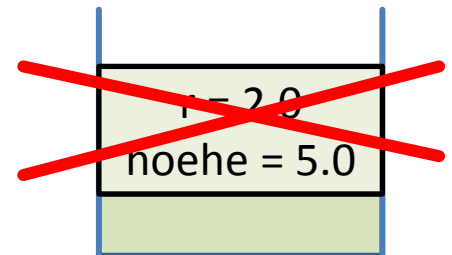
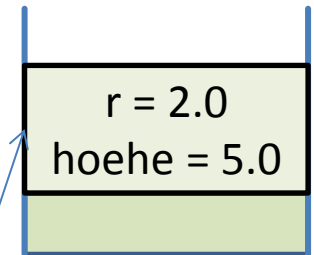
Konventionen	Beispiel
Methodenname Anfangsbuchstabe klein	
Mindestens ein Verb in Präsensform verwenden, das gut beschreibt, was die Methode macht	zahlPiBerechnen() bankkontoAnlegen() vonEhepartnerScheidenLassen()
Methoden soll komplett auf dem Bildschirm passen (ca. 25 Zeilen) Notfalls in mehrere Methoden (private) aufteilen und innerhalb der eigentlichen Methode aufrufen	

Laufzeitkeller

- Laufzeitkeller (call stack)
 - Speicherbereich, der alle Parameter und lokalen Variablen **pro Aufruf** einer Methode enthält
 - Weitere Werte (für uns nicht von Belang):
 - Zwischenergebnisse in Ausdrücken
 - Rücksprungadresse
 - Rückgabewert einer Funktion
 - Werte einer Ausnahme (Exception)
- Genau Implementierung des Laufzeitkellers nicht von Interesse
- Abstrakte Modelvorstellung ist wichtig für das Programmieren

Laufzeitkeller

- Wächst von „unten“ nach „oben“
- Rahmen für einen Aufruf
 - Enthält alle Parameterwerte und Variablen für diesen Aufruf
 - Anzahl Werte und Größe des Rahmen ist für jede Funktion konstant
- Bei Aufruf einer Methode wird ein neuer Rahmen oben auf den Laufzeitkeller gelegt
 - Nur die im Rahmen enthaltenen Werte sind sichtbar
zylinderVolumenBerechnen(2.0, 5.0)
- Der neue Rahmen ist erst gültig, wenn das Programm zur Methode verzweigt
- Bei Beendigung einer Methode, wird der Rahmen verworfen
 - Methode kehrt zur aufrufenden Stellen zurück
 - Rahmen dieses vorherigen Aufrufs oben
 - Alle Variablen und Parameter wieder sichtbar



Laufzeitkeller

- Aufgabe
 - Aufruf machewas1(2)
 - Zustand des Laufzeitkellers skizzieren, bevor der Aufruf von machewas2(b,0) zu ende ist

```
public static void macheWas1(int a) {  
    int b = 1;  
  
    b = machewas2(a + 1, 1);  
  
    a = machewas2(b, 0);  
}  
public static int machewas2(int a, int b)  
{  
    return a + b;  
}
```

Klassenvariablen

- Klassenvariablen existieren genau einmal pro Klasse
- Anderer Begriff: globale Variablen
- Sie werden innerhalb der geschweiften Klammern der Klasse deklariert
- Syntax

```
public class Klassenname {  
    Modifier static Typ Bezeichner;  
    Modifier static Typ Bezeichner = Ausdruck;  
}
```

- Modifier wie bei Klassenmethoden: public, private, ...
- Zugriff wie bei Klassenmethoden
 Klassenname.Bezeichner
 Bezeichner (falls auf Klassenvariable zugegriffen wird,
 in der Klasse mit der Deklaration steht)

Klassenvariablen

- Klassenvariablen müssen vor Verwendung nicht initialisiert werden
- Nicht initialisierte Klassenvariablen haben den voreingestellten Wert des zugehörigen Datentyps:
 - 0 bei ganzzahligen Typen, 0.0 bei Gleitkomma
 - `'\u0000'` bei char
 - false bei boolean

```
public class Klassenname {  
    public static int a;    // 0  
    public static int b = 2;    // 2  
    public static double c;    // 0.0  
    public static double d = 2.0;    // 2.0  
    public static double e = d + 1.5;    // 3.5  
}
```

Klassenvariablen

- Beispiel

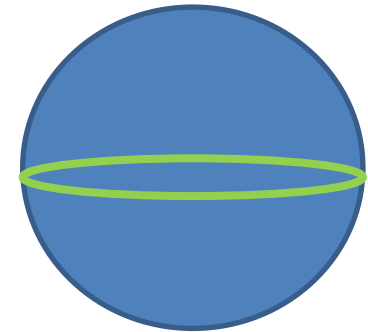
Der Planet Erde existiert genau einmal
Die Erde hat einen **Umfang** am Äquator
und eine **Bahngeschwindigkeit** um die Sonne

Bahngeschwindigkeit kann sich ändern

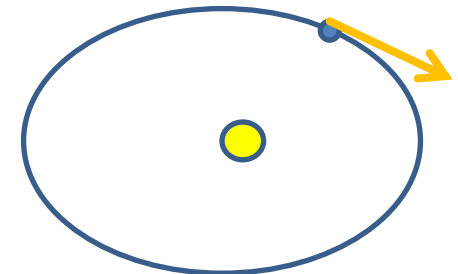
- Implementierung

Eine Klasse Erde mit zwei
Klassenvariablen

Bahngeschwindigkeit über eine
Klassenmethode um einen
Prozentbetrag erhöhen



40075,017 km



29.78 km/s

Im Durchschnitt

Klassenvariablen

- Symbolische Konstanten
 - Variablen, deren Wert sich während des Programmlaufs nie ändert (Äquatorumfang)
 - Schlüsselwort **final**
 - Bei allen Variablendeklarationen (auch Parametern) erlaubt
 - final Variablen können nur genau einmal initialisiert werden
 - Weitere Zuweisungen sind nicht mehr erlaubt
 - Compiler prüft dies
 - Konvention bei symbolischen Konstanten
 - Buchstaben groß schreiben, _ zu Trennung von Teilwörtern verwenden
- UMFANG_AEQUATOR
- Symbolische Konstanten bei konstanten Werten mit einer **Bedeutung** verwenden oder bei Werten, die sich selten, aber im Laufe der Zeit ändern können (Steuersätze)

Math.PI Integer.MAX_VALUE

Klassenvariablen


```
public class Umsatzsteuer {  
    public static final double REGELSATZ = 19.0; // Prozent  
    public static final double ERMAESSIGTER_SATZ= 7.0;  
}
```

- Keine symbolischen Konstanten verwenden:
1.8 * celsius + 32.0 (Formel mit Umrechnungsfaktoren ohne Bedeutung)
a > 0 (a ist positiv)
a = a + 2;
– Aber falls 2 eine Bedeutung hat, z.B.
a = a + ANZAHL_NACHKOMMEN;

Klassenvariablen

- Sichtbarkeit von Variablen
 - Lokale Variable sind vor Klassenvariablen sichtbar
 - Eine lokale Variable a verdeckt eine Klassenvariable a
 - Bei verdeckten Klassenvariablen den Klassennamen davor schreiben, um Namenskonflikt aufzulösen

```
public class A {  
    public static int x = 5;  
  
    public static void main(String [] s) {  
        System.out.println(x); // 5 (Klassenvariable)  
        double x = 2.5; // Klassenvariable verdeckt  
        System.out.println(x); // 2.5  
        System.out.println(A.x); // 5  
    }  
}
```



Klassenvariablen

- Nur für Werte verwenden, die genau einmal existieren
 - Werte einer Konfigurationsdatei
 - Benutzerdaten (Name, Sprache, ...) einer Desktopanwendung
- Ansonsten nur bei symbolischen Konstanten verwenden

Math.PI Erde.AEQUATOR
- Java ist objekt-orientiert:
 - Werte werden üblicherweise in Objekten gespeichert (später)

Schrittweise Verfeinerung

Falls ein Problem nicht direkt mit einem Programm lösbar ist:

Teile das Problem in mehrere Teilprobleme auf

Löse die Teilprobleme mit Schrittweiser Verfeinerung

Füge die Lösungen zu einem Gesamtprogramm zusammen

- In der Praxis im Quelltext durchführbar
 - Problem genau mit einem Kommentar beschreiben
 - So früh wie möglich Programm-Anweisungen verwenden und ggf. mit Kommentaren mischen
 - Komplexe Probleme als Methoden mit Parametern formulieren

Niklaus Wirth. Program Development by Stepwise Refinement, in Communications of the ACM, 14(4):221-227, April 1971

Schrittweise Verfeinerung

Vom Problem zum Programm

1. Problem analysieren
 - Problem genau formulieren und verstehen
 - Insbesondere die Eingabe (Gegeben) und die erwartete Ausgabe (Gesucht) muss definiert sein
 - Wertebereiche der Ein- und Ausgabe sowie deren Bedeutung festlegen
2. Lösungsidee entwickeln (kreativ sein)
 - es gibt nur wenige allgemeine Lösungsansätze für spezielle Problembereiche,, z.B. beim Algorithmenentwurf, Datenbankdesign
 - an Beispielen auf Papier ausprobieren
3. Programmieren
 - z.B. **Schrittweise Verfeinerung** anwenden
4. Testen, um Fehler zu finden
 - Bisher: Programm mit Beispieldaten ausführen und das Ergebnis durch Hinsehen mit dem zugehörigen gesuchten Werten vergleichen.

Schrittweise Verfeinerung

- Beispiel (Rechnerübung)

Gegeben:	Datum im Gregorianischen Kalender Tag (1-31), Monat (1-12), Jahr (ab 1900)
Gesucht:	Wochentag (Mo, Di, ..., So) des Datums

- Lösungsidee
 - Wenn Montag ist, dann ist in 7 Tagen wieder Montag.
 - **Alle Tage seit einem Stichdatum zählen (1.1.1900)**
 - **Resultat modulo 7 teilen**
 - 0 entspricht Wochentag vom 1.1.1900 (Montag)

Schrittweise Verfeinerung

- Klassenmethode für Problem implementieren
- Kommentare verwenden, um noch nicht gelöste Teilproblem zu beschreiben

```
public static int wochentagBerechnen(int tag, int monat,
int jahr) {
    // Alle vergangenen Tage seit einem Stichtag z.B
    1.1.1900 zählen
    // vergangenen Tage mit Rest durch 7 teilen: 0 =
    Montag
    // Rest zurückgeben
}
```

Schrittweise Verfeinerung

1. lokale Variable für die vergangenen Tage deklarieren
2. return Anweisung programmieren mit module 7 Rechnung

```
public static int wochentagBerechnen(int tag, int monat,
int jahr) {
    int vergangeneTage = 0;
    // Alle vergangenen Tage seit einem Stichtag z.B
    1.1.1900 zählen
    return vergangeneTage % 7;
}
```

Ein Teilprobleme noch zu lösen

Schrittweise Verfeinerung

1. In drei sequentielle Teilprobleme aufteilen

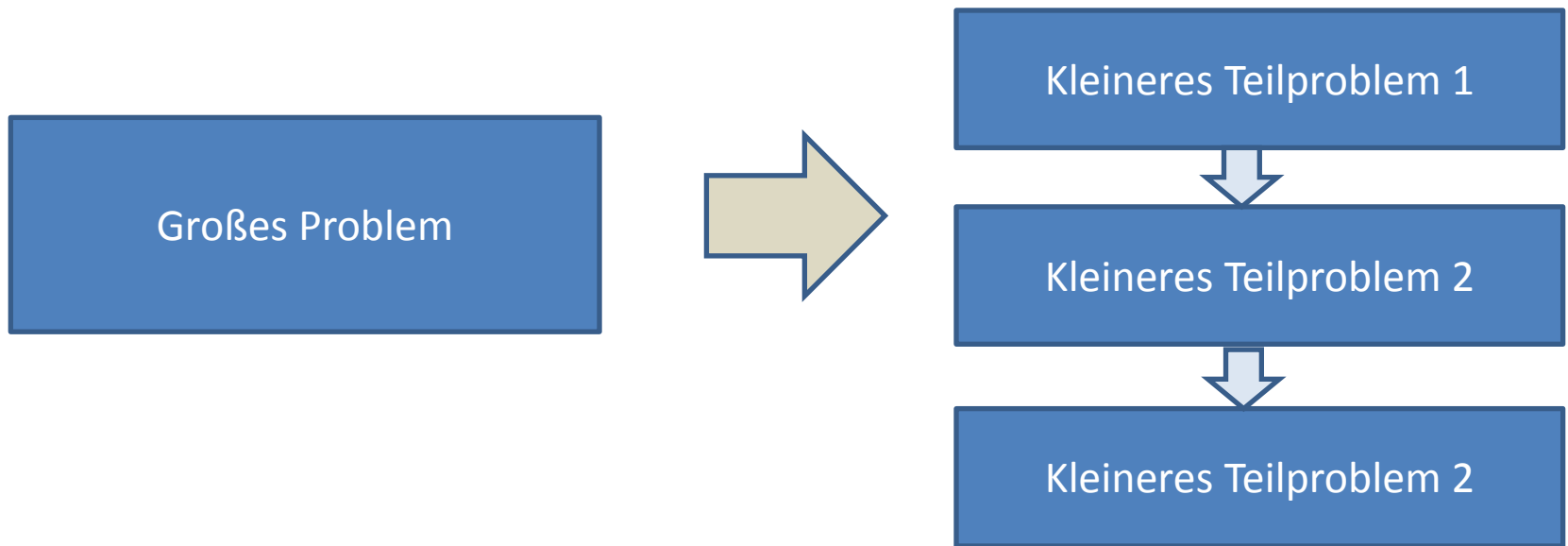
```
public static int wochentagBerechnen(int tag, int monat,
int jahr) {
    int vergangeneTage = 0;
    // Alle vergangenenTage vom 1.1.1900 bis 1.1.jahr
berechnen
    // Alle vergangenen Tage der Monate bis 1.monat.jahr
hinzuaddieren
    // Alle vergangenen Tage hinzuaddieren
    return vergangeneTage % 7;
}
```


Schrittweise Verfeinerung

1. Letzte Teilproblem direkt lösen

```
public static int wochentagBerechnen(int tag, int monat,
int jahr) {
    int vergangeneTage = 0;
    // Alle vergangenenTage vom 1.1.1900 bis 1.1.jahr
    berechnen
    // Alle vergangenen Tage der Monate bis 1.monat.jahr
    hinzuaddieren
    vergangeneTage += tag;
    return vergangeneTage % 7;
}
```

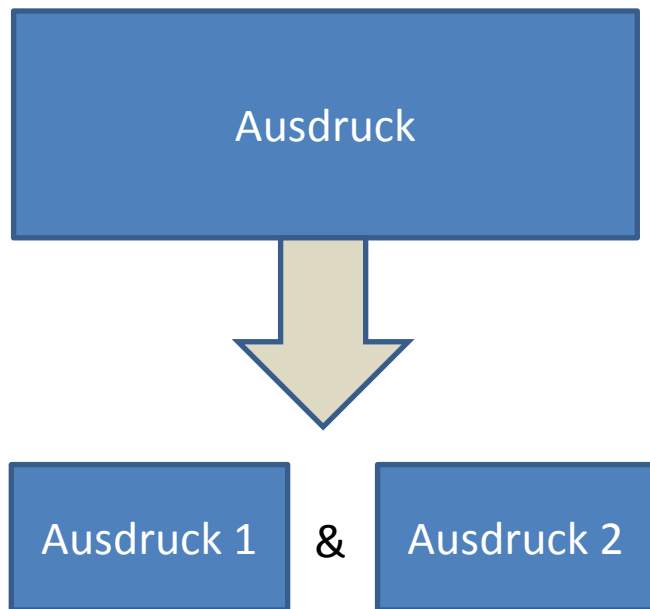
Schrittweise Verfeinerung



- **Verfeinerung in mehrere sequentielle Teilproblem**
- Verfeinerung von alternativen Teilproblemen oder von wiederholt ausgeführten Teilproblemen direkt mit einer Kontrollanweisung angeben (später)

Schrittweise Verfeinerung

- Verfeinerung von Ausdrücken mit Mehrzeilenkommentaren im partiellen Ausdruck möglich



Alice ist hungrig und } es
gibt überhaupt nichts zu
essen

`/* Alice ist hungrig */
 & /* es gibt überhaupt
nichts zu essen */`

Schritt

- Rest Rechnerübung Informatik Bachelor
- Problem bei dieser Vorgehensweise
 - vermutlich wird bei Schaltjahren im Monat Jan, Feb, ein Tag zu viel addiert
 - fällt (hoffentlich) beim Testen auf
 - Programm nach Fehlersuche wieder anpassen
 - Frühes Testen bei dieser Methodik ist wichtig, um rechtzeitig Fehler in den Lösungsideen zu finden

Schrittweise Verfeinerung

- Vorteile:
 - Teilprobleme werden einfacher
 - schon gelöste Teilproblem können teilweise getestet werden
- Nachteile
 - Programme oft unnötig lang, insbesondere durch redundante Berechnungen
 - Nur bei Abläufen anwendbar, aber nicht zur Strukturierung von Daten.
 - Schrittweise Verfeinerung funktioniert nicht immer, z.B. bei sehr schwierigen oder unlösbaren Problemen

Redundanz im Quelltext sind z.B. identische oder ähnliche Programmteile, toter Code (nicht verwendeter Code)