Modularisierung

mit Klassen und Objekten

Martin Häcker & Felix Schwarz

Wir werden uns in drei Schritten der Objektorientierung nähern
* Modularisierung von Methoden
* Modularisierung von DAten (Höhere Datenstrukturen)
* Objekte (als Zusammenführung der zwei Konzepte)

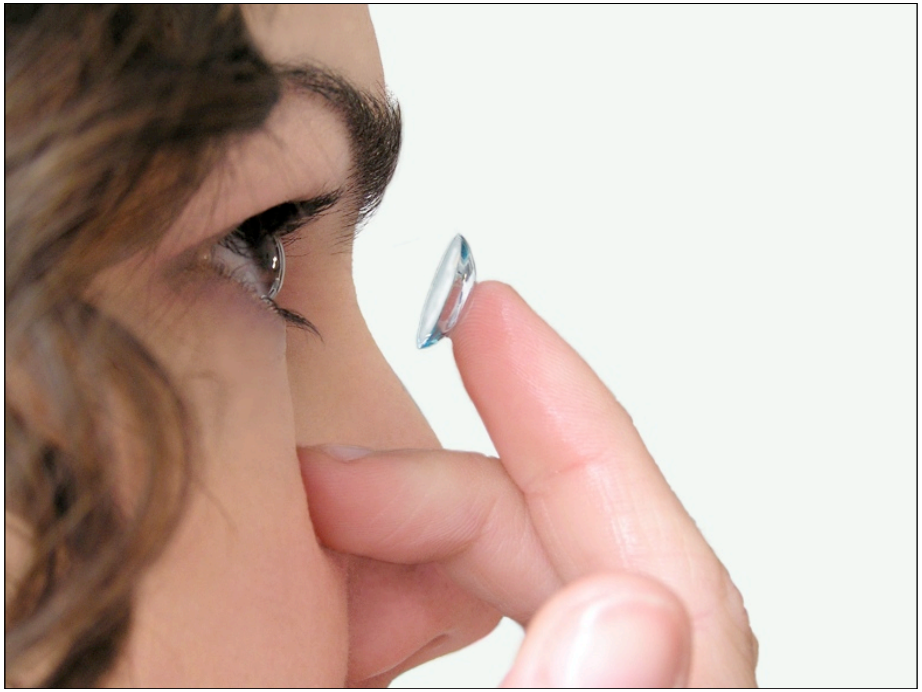
Zum mitschreiben ermutigen! (Mit Begründung?) -> Aber nicht Beispiele -> die stehen online!

Zuerst Modularisierung an sich: Warum? DAS Handwerkszeug der Informatiker

Was ist Modularisierung?
Zwei Betrachtungsweisen: Das ganze besteht aus Teilen (Top-Down) / Teile bilden das Ganze (Bottom-Up) Top-Down zuerst
Überall (in der Welt) gibt es oder haben wir gedankliche Trenner eingezogen
Diese Trenner an Beispielen Verdeutlichen

Der Mensch
Mensch ist unendlich kompliziert
* Reflexe / Atmen
* Gewollte koordination -> Schreiben
* Zusammenspiel der Muskeln
* Zusammenspiel der Zellen / kommunikation über hormone...

Instinktiv sehen wir Teile:
* Kopf, Bauch, Arme
* Top-Down!
Wenn wir Teile genauer Betrachten (und erst dann) sehen wir weitere details
Drei Beispiele um diese Denkweise zu verdeutlichen



3

1. Kopf:
* Nase, Haare, Augebrauen, Augen
* Augen die Wieder aus teilen bestehen
* Vielleicht eine Sehschwäche haben

4

2. Mund:
* Lippen, Zähne, Bart?

5

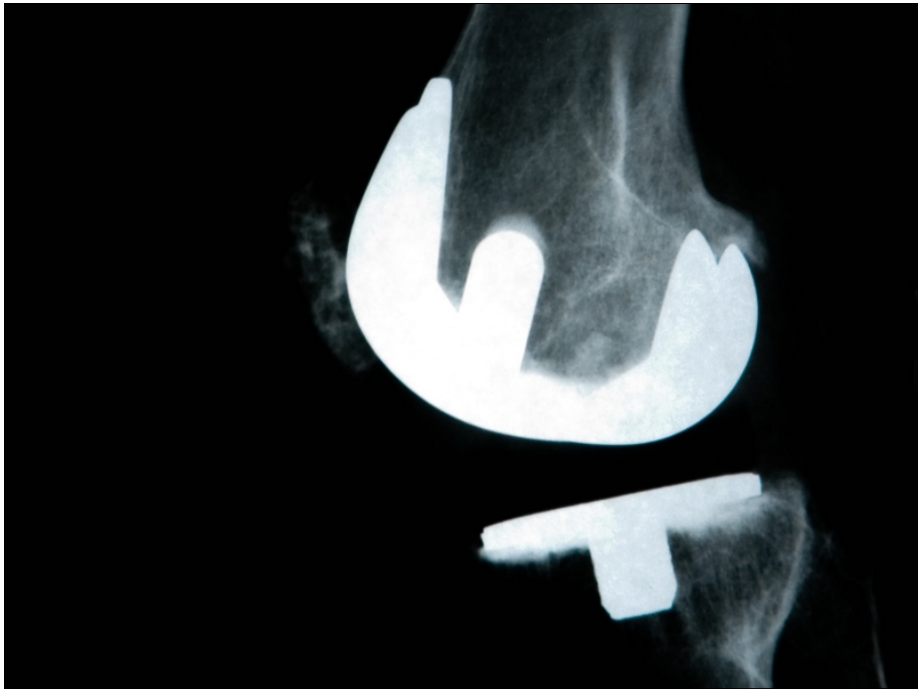
Genauer Betrachtet (ohne den rest)
* Schiefe Zähne?
* Piercing?

Details die wir erst wahrnehmen wenn wir uns um dieses Modul kümmern

6

3. Beine

die genauer betrachtet...



7

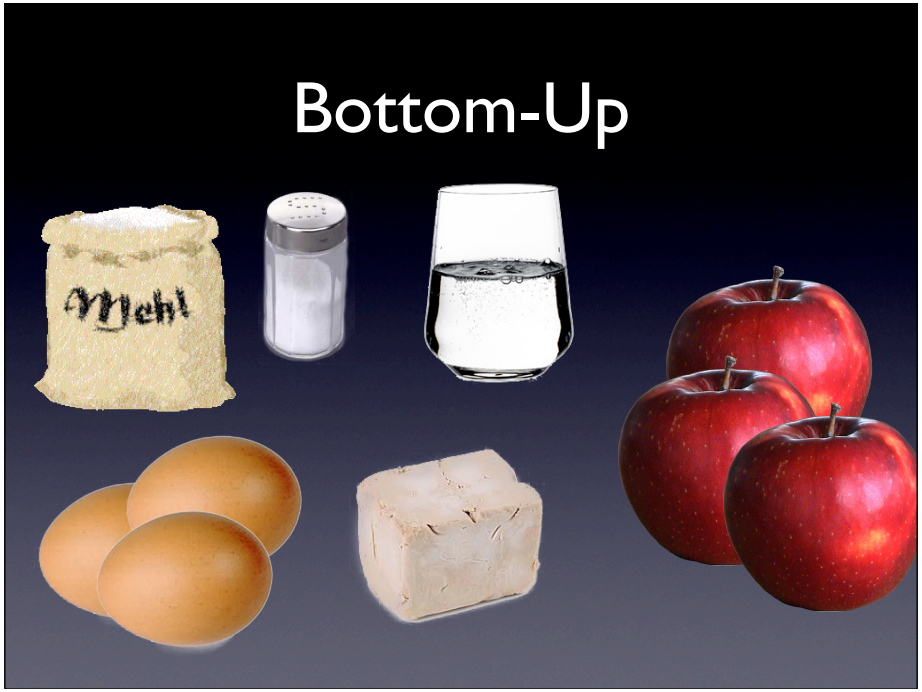
... Prothesen enthalten können

Denkt man Top-Down (Aus was besteht etwas)
Dann nimmt man _ERST_ wenn man die Details genauer untersucht
Deren „implementation“, deren „innereien“ wahr.

Im Ganzen ist nur wichtig „WAS“ etwas tut
Erst wenn man es für sich anschaut, wird interessant „WIE“ es das tut.

Das ist, was abstraktion ausmacht
-> Das ermöglich uns über Menschen / beliebige Dinge nachzudenken
ohne das wir an tausenden details hängenbleiben und unser Hirn explodiert.

So wichtig: Daher noch ein Beispiel es anders herum zu betrachten: Viele Zutaten...



also Bottom-Up
viele Zutaten (eier, mehl, wasser, hefe...)

...

lassen sich zu einem ganzen zusammenfügen

In dem die einzelnen Teile nicht mehr zu sehen sind
das MEHR ist als nur die Summe seiner Teile

Man sieht die Teile _wirklich_ nicht mehr
-> Natureigenschaft / Fähigkeit ermöglich komplexe Probleme zu lösen

10

Ihr habt also ein handwerk
Das ist: komplexität beherschen
Euer wichtigstes werkzeug ist abstraktion
Die möglichkeit Teile des ganzen für sich verstehen -> ohne den rest

Schon die römer kannten das

macht kleine projekte „nur“ elegant
aber große überhaupt erst möglich

mit diesem Vorlauf, jetzt in die Details: zuerst: Methoden

Modularisierung von Code

Erster schritt für Modularisierung
-> CODE

Wird überall gemacht:
* Opal: Dateien .sign .impl
* Java: Dateien mit .java

Hier gibt es nichts neues - das habt ihr die letzten Tage schon alles benutzt

```
class PointModule {
  static boolean arePointsEqual
        (float xx, float xy,
         float yx, float yy)
    { ... }
  static float distanceBetweenPoints
        (float xx, float xy,
         float yx, float yy)
   { ... }

}
```


jeweils mit einer „class BlahBlub“
die man von anderen Dateien aus verwenden kann

Darin stehen die Methoden die zusammengehören
Was nicht dazugehört -> kommt woanders hin
-> (auch wenn man es an dieser stelle (für abstand) braucht
-> dafür gibts das „MathModule“ oder wie man es nennt

Auswirkungen

```
class Math {
    static float sqrt(float positiveNumber) { ... }
    static double E = ...;
    static double PI = ...;
    static double pow(double base, double exponent)
    { ... }
    // ...
}


float distanceBetweenPoints(float ax, float ay,
                            float bx, float by) {
    return Math.sqrt(Math.pow(ax - bx, 2)
                     + Math.pow(ay - by, 2));
}
```


Wenn man so ein Modul definiert, dann kann man mit der „.“ syntax auf die Methoden zugreifen:

Hier Math.sqrt und pow

In der tat gibt es schon ein Modul „Math“ -> mehr dazu aber erst heute Nachmittag

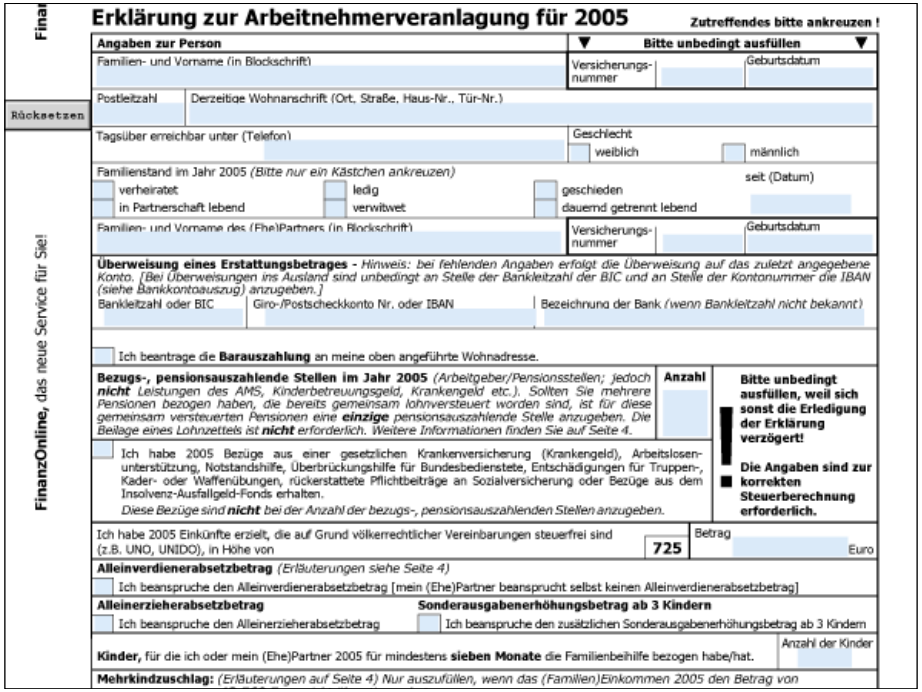
Java bietet noch mehr
-> Packages, Information Hiding
-> Selber anlesen / recherchieren (nach dem kurs)

Das war es schon -> Weiter zu Modularisierung von Daten

Modularisierung von Daten

Bis hierher könnt ihr in Java nur primitive Werte an methoden übergeben - int, boolean, String (!)
Die einzige bisher bekannte Verallgemeinerung ist der Array -> mehrere vom gleichen

In der realen Welt sind Daten aber in den meisten Fällen komplexer!
-> Color: r, g, b / c,m,y,k
-> Bruch: Zähler, Nenner
-> Steuererklärung



Erklärung zur Arbeitnehmerveranlagung für 2005

Zutreffendes bitte ankreuzen !

Angaben zur Person — Bitte unbedingt ausfüllen

Familien- und Vorname (in Blockschrift) | Versicherungsnummer | Geburtsdatum

Rücksetzen

Postleitzahl | Derzeitige Wohnanschrift (Ort, Straße, Haus-Nr., Tür-Nr.)

Tagsüber erreichbar unter (Telefon) | Geschlecht: weiblich, männlich

Familienstand im Jahr 2005 (Bitte nur ein Kästchen ankreuzen) — seit (Datum)

verheiratet, ledig, geschieden, in Partnerschaft lebend, verwitwet, dauernd getrennt lebend

Familien- und Vorname des (Ehe)Partners (in Blockschrift) | Versicherungsnummer | Geburtsdatum

Überweisung eines Erstattungsbetrages - *Hinweis: bei fehlenden Angaben erfolgt die Überweisung auf das zuletzt angegebene Konto. (Bei Überweisungen ins Ausland sind unbedingt an Stelle der Bankleitzahl der BIC und an Stelle der Kontonummer die IBAN (siehe Bankkontoauszug) anzugeben.)*

Bankleitzahl oder BIC | Giro-/Postscheckkonto Nr. oder IBAN | Bezeichnung der Bank *(wenn Bankleitzahl nicht bekannt)*

Ich beantrage die **Barauszahlung** an meine oben angeführte Wohnadresse.

Bezugs-, pensionsauszahlende Stellen im Jahr 2005 *(Arbeitgeber/Pensionsstellen; jedoch **nicht** Leistungen des AMS, Kinderbetreuungsgeld, Krankengeld etc.). Sollten Sie mehrere Pensionen bezogen haben, die bereits gemeinsam lohnversteuert worden sind, ist für diese gemeinsam versteuerten Pensionen eine **einzige** pensionsauszahlende Stelle anzugeben. Die Beilage eines Lohnzettels ist **nicht** erforderlich. Weitere Informationen finden Sie auf Seite 4.* — **Anzahl**

Ich habe 2005 Bezüge aus einer gesetzlichen Krankenversicherung (Krankengeld), Arbeitslosenunterstützung, Notstandshilfe, Überbrückungshilfe für Bundesbedienstete, Entschädigungen für Truppen-, Kader- oder Waffenübungen, rückerstattete Pflichtbeiträge an Sozialversicherung oder Bezüge aus dem Insolvenz-Ausfallgeld-Fonds erhalten.

*Diese Bezüge sind **nicht** bei der Anzahl der bezugs-, pensionsauszahlenden Stellen anzugeben.*

Bitte unbedingt ausfüllen, weil sich sonst die Erledigung der Erklärung verzögert!

Die Angaben sind zur korrekten Steuerberechnung erforderlich.

Ich habe 2005 Einkünfte erzielt, die auf Grund völkerrechtlicher Vereinbarungen steuerfrei sind (z.B. UNO, UNIDO), in Höhe von **725** | Betrag Euro

Alleinverdienerabsetzbetrag *(Erläuterungen siehe Seite 4)*

Ich beanspruche den Alleinverdienerabsetzbetrag [mein (Ehe)Partner beansprucht selbst keinen Alleinverdienerabsetzbetrag]

Alleinerzieherabsetzbetrag

Ich beanspruche den Alleinerzieherabsetzbetrag

Sonderausgabenerhöhungsbetrag ab 3 Kindern

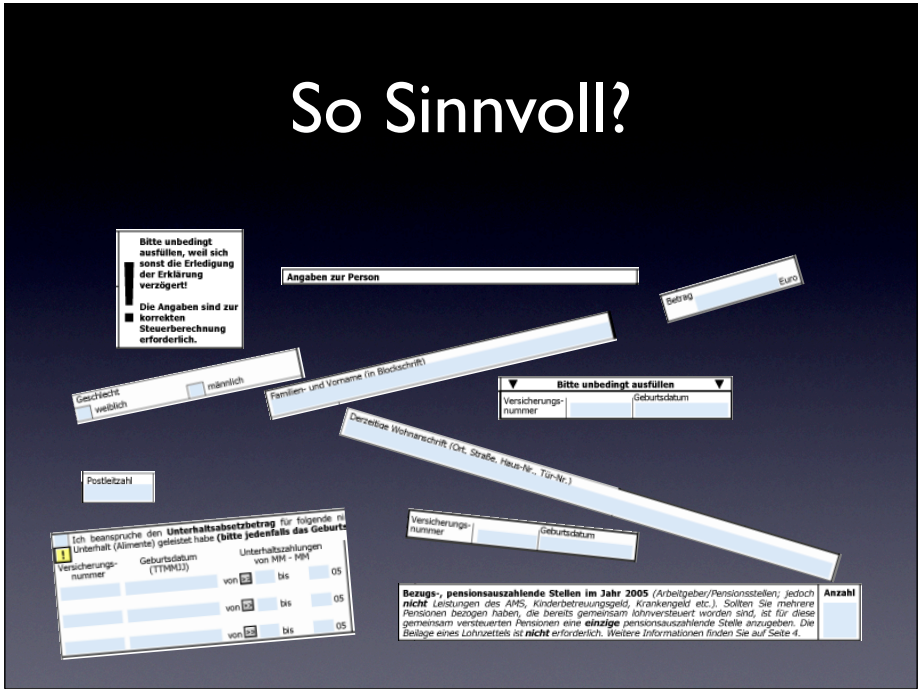
Ich beanspruche den zusätzlichen Sonderausgabenerhöhungsbetrag ab 3 Kindern

Kinder, für die ich oder mein (Ehe)Partner 2005 für mindestens **sieben Monate** die Familienbeihilfe bezogen habe/hat. | Anzahl der Kinder

Mehrkindzuschlag: *(Erläuterungen auf Seite 4) Nur auszufüllen, wenn das (Familien)Einkommen 2005 den Betrag von*

FinanzOnline, das neue Service für Sie!

Name, Versicherungsnummer,
Anschrift: Postleitzahl, Straße, Hausnummer
Telefonnummer, Geschlecht, Familienstand
Konto: BLZ, Kontonummer, Bankname
...
Probleme beim Programmieren



Stellt euch vor, eine (viele!) Funktion mit 800 Parametern, für jeden teil der Steuererklärung eine
Widerspricht Grundsätzen der Abstraktion!

Steuersoftware -> viele Stellen die nur Steuererklärung als ganzes weiterreichen:
* ganzes: von öffnen / speichern modul <-> anzeigemodul <-> netzwerkmodul (verschicken an finanzamt) <-> druckenmodul
* teile: Anzeigemodul -> detailanzeige für erste/zweite seite -> für bestimmte felder
Überall wird in Einheiten kommuniziert: Wenn jedes mal 800 Variablen -> *tod*
Das witzige: In jeder Software ist das so (nur hoffentlich nicht ganz so schlimm)

Damit große Programme überhaupt möglich werden und kleine klar und fehlerfrei
braucht man: Modularisierung auf Datenebene
Programm soll nichts annehmen über Daten die es nur weiterleitet -> nur das was es gerade braucht
Alles einzeln weiterleiten führt zu Fehlern, Aufwand, viel Schreibarbeit – ist nicht elegant

Syntax - Problem

```
float red, green, blue;
float cyan, magenta,
yellow, black;

???? cmykFromRGB(????)
{ ... }
```


Beispielwechsel, damit es besser auf die Folie geht
Zweck: Umrechnung der Farbe von RGB (Red Green Blue) nach CMYK (Cyan Magenta Yellow Blue)
mehrere Probleme:
* Wie gibt man zurück? (Vier werte werden gebraucht)
* Die drei Teile von RGB-Farben getrennt - einzelne Werte für sich machen aber keinen Sinn

Wir wollen einen "Kleber" damit wir Datenobjekte zusammenhalten können
(Wenn Zeit: primitv: argumente: hinschreiben, array zusammen, sprachen die das machen, als string, als lookuptable...)

Syntax - Lösung

```
CMYK from(RGB aColor) {
 return buildCMYK(computeC(aColor),
                  computeM(aColor),
                  computeY(aColor),
                  computeK(aColor));
}
```


(Wishfull Thinking als mächtige Abstraktionstechnik)
Wir hätten gerne eine Prozedur die diese Daten CMYK und RGB
als Abstrakter Datentyp (siehe Opal) der die Werte zusammenfasst
und auch methoden die ihn wieder zerpflücken können

Dann gehts! -> Aber wie?

Syntax - Neu

```
class RGB {
    float r;
    float g;
    float b;
}
```


Am beispiel von einer Farbe
kein Static!

Neuer Typ eingeführt RGB -> Rückgabetyp (nächste folie)

Syntax - Verwendung

```
static RGB createRGB(float r,
                     float g,
                     float b) {
    RGB aColor = new RGB();
    aColor.r = r;
    aColor.g = g;
    aColor.b = b;
    return aColor;
}
```


Neu:
* rückgabetyp
* Erzeugen mit new (wie arrays! -> die sind nämlich so etwas ähnliches)
* Zugreifen auf die Elemente mit „.“
* dann zurückgeben

Syntax - Verwendung II

```
static void printColor(RGB aColor)
{
  System.out.println("aColor: " +
                     aColor.r + ", ",
                     aColor.g + ", ",
                     aColor.b);
}
```


* Erhalten als Argument
* zugreifen auf variablen mit „.“

schreiben und lesen ist gleich!

Syntax - Verwendung III

```
RGB black = createRGB(0, 0, 0);
RGB white;
white = createRGB(1, 1, 1)
```


Zwei möglichkeiten:
* Direkt definieren und initialisieren
* Erst Definieren, dann initialisieren

Mann nimmt das was gerade praktischer ist (scope größer als da wo man information für initialisieren hat)

Man muss nur darauf achten das man die Variable initialisiert hat bevor man sie verwendet

Das ist alles was es zu Modularisierung von Daten (Höheren Datenstrukturen) zu sagen gibt
* Noch ein paar häufige Fehler hinweisen

Gotcha - static

```
class RGB {
    static float r, g, b;
}

RGB black = createColor(0, 0, 0);
RGB white = createColor(1, 1, 1);
printColor(black);
```


Seht ihr den Unterschied bei RGB? -> static

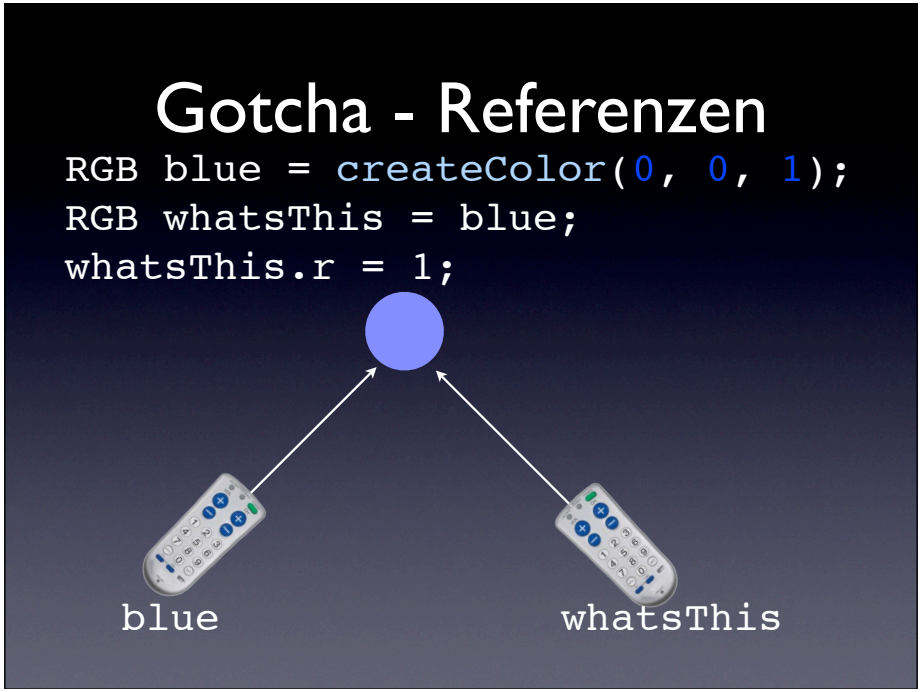
was passiert mit static?
Was gibt das aus?

(nach nächster slide zurück, dann) durch das weglassen vo static kann man mehrere haben

Problem.

-> zurück, nachsehen was herauskommen sollte

Static von allen geteilt
mehrere = kein static -> vorsicht



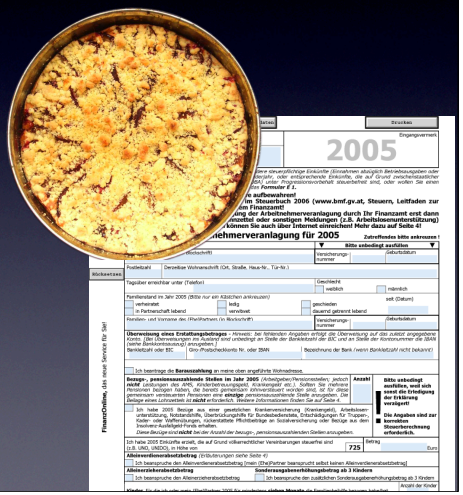
Referenzen sind „Fernbedienungen“ für die Datenobjekte

Wenn man sie also kopiert -> dann hat man nur zwei „Fernbedienungen“, nicht zwei Datenobjekte
-> genauso bei arrays!
-> alles was mit „new“ angelegt wird

DAS unterscheidet diese Daten von int, double, boolean
Da hat man wirklich kopien gemacht
(wegen Performance, wäre heute unwichtig / anders gelöst)

Zusammengefasst

- Module
- Datenstrukturen

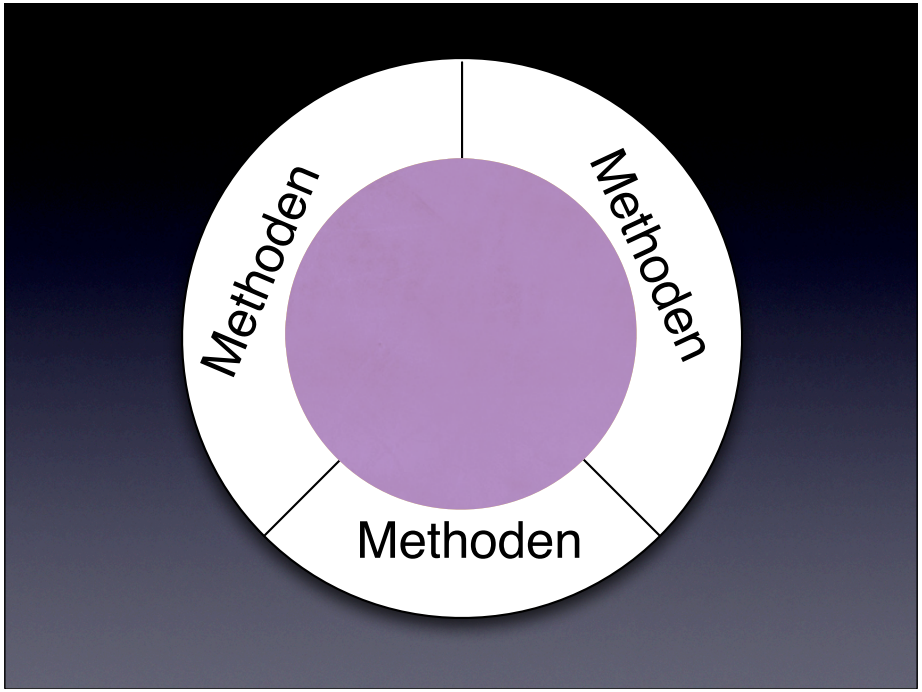


Wir haben zusammengehörige Funktionen zu Modulen zusammengefasst
Wir haben zusammengehörige Daten zu höheren Datenstrukturen zusammengefasst

Zeit das wir die beiden Konzepte zusammenfassen

Alle gespannt?

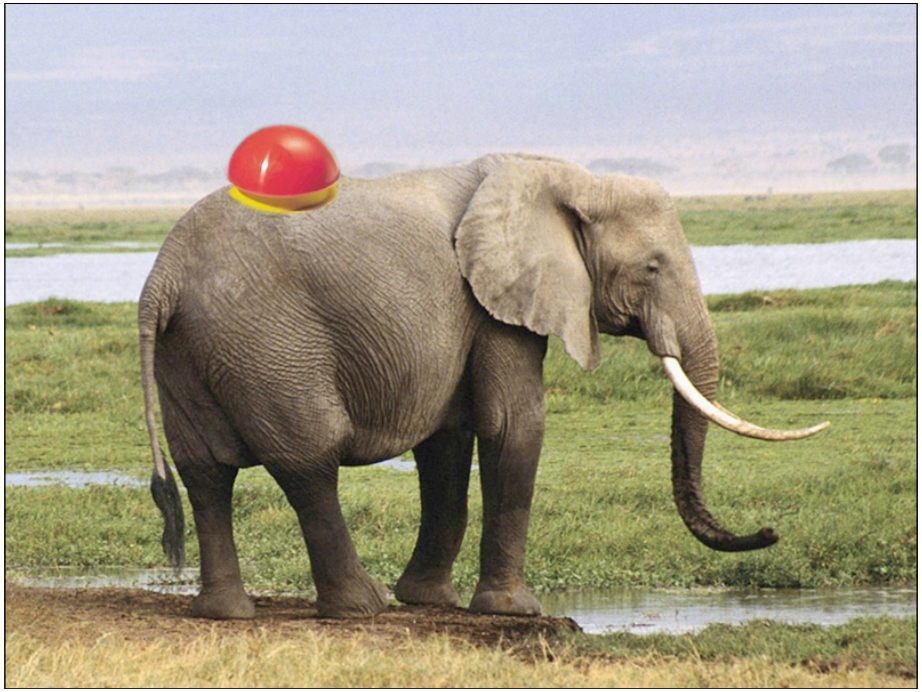
Das schicke: Es gibt eigentlich nichts neues hier, alles was Objektorientierung ausmacht habt ihr schon gesehen. (bzw. alles davon was wir hier behandeln)
Denn Objektorientierung ist nur eine Programmiertechnik! (Die ihr in allen Sprachen verwenden könnt)
Konzept Module und Datentypen zusammen: Schutzschicht von Methoden



Konzept Module und Datentypen zusammen: Schutzschicht von Methoden
-> ermöglicht austauschen des Datentyps - solange methoden gleich bleiben

Beispiel Punkt: Kartesische- vs. Polar-Kooridnaten

Umgangssprachlicher Definitionsversuch:

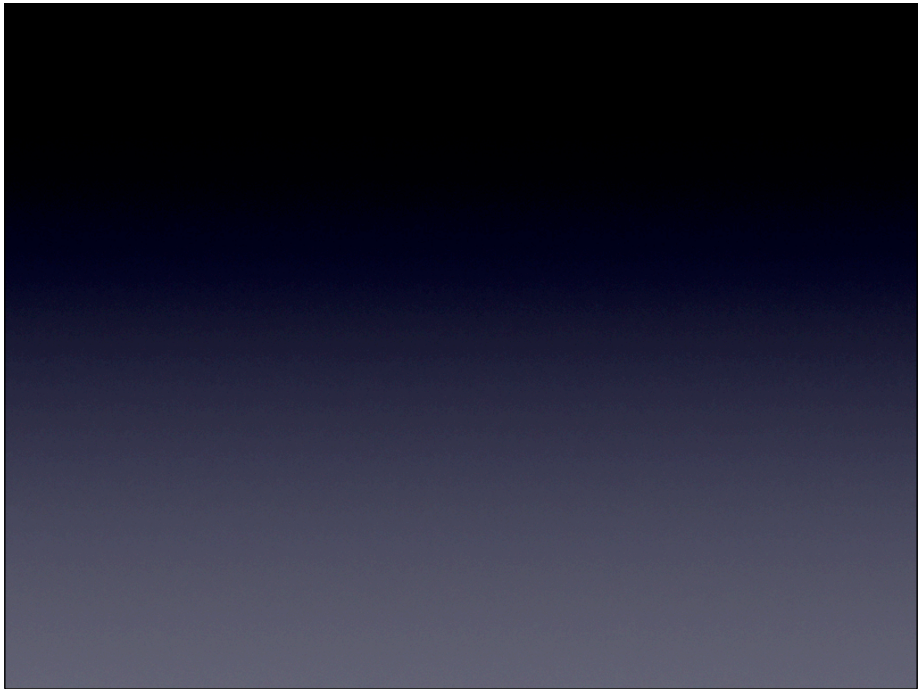


STATISTIKER jagen das erste Tier, das sie sehen, n-mal und nennen es Elefant.

MATHEMATIKER jagen Elefanten, indem sie nach Afrika gehen, alles entfernen, was nicht Elefant ist und ein Element der Restmenge fangen.

OBJEKTORIENTIERTE PROGRAMMIERER bestehen darauf, daß der Elefant eine Klasse sei, und somit schließlich seine Fang-Methoden selbst mitzubringen habe.

DAS sind WIR!



was heist das in der praxis?
Aufgefallen, das Syntax für Module und Höhere Datenobjekte ähnlich ist
Bei Datenobjekten machte das weglassen von Static den unterschied das man mehrere haben konnte.

Genauso bei Methoden

Bekannte Syntax

```
class Point {
    float x,y;
}
class PointModul {
    static Point createPoint(float x, float y)
    { ... }
    static boolean arePointsEqual(Point aPoint,
                                  Point bPoint)
    { ... }
    static float distanceBetweenPoints(Point aPoint,
                                       Point bPoint)
    { ... }
}
```


Die Syntax ist euch jetzt schon bekannt

Ab hier ist alles nur noch Syntaktischer Zucker, der es leichter machen soll den Code zu schreiben

1. Schritt -> alles in eine Klasse

1. Schritt

```
class Point {
    float x,y;

    static Point createPoint(float x, float y)
    { ... }
    static boolean arePointsEqual(Point aPoint,
                                  Point bPoint)
    { ... }
    static float distanceBetweenPoints(Point aPoint,
                                       Point bPoint)
    { ... }
}
```


Es wird eine Klasse - nicht mehr Zwei

Als nächstes das static

2. Schritt

```
static boolean arePointsEqual(Point aPoint,
                              Point bPoint)
    { ... }
static float distanceBetweenPoints(Point aPoint,
                                   Point bPoint)
    { ... }
```


Wir betrachten zuerst die beiden Methoden

Diese Methoden verlieren ihr static

Auswirkungen

```
boolean arePointsEqual(Point aPoint,
                       Point bPoint)
 { ... }
```

```
boolean arePointsEqual(
                       Point bPoint)
 { ... }
```


der erste parameter heißt jetzt this
allerdings wird er im Source-Code nicht geschrieben
-> Java fügt ihn in jeder methode automatisch ein - auch wenn man ihn nirgends sieht
* this hat immer den typ, von der Klasse in der eine Methode steht (wenn sie nicht static ist!!!)

next: Man benennt die Methoden und argumente dann meist noch etwas anders

Auswirkungen II

```
boolean isEqualToPoint(Point other)
 { ... }
```


Man benennt die Methoden und argumente dann meist noch etwas anders
Man wählt namen ausgehend von „sich“ und „anderen“

arePointsEqual -> isEqualToPoint
distanceBetweenPoints -> distanceToPoint

is/are für methoden die booleans zurückgeben

this - wird nicht geschrieben! ist trotzdem da

Auswirkung III

```
Point zero = createPoint(0,0);
Point one = createPoint(1,1);
arePointsEqual(zero, one);
```

```
zero.isEqualToPoint(one);
```


Man schreibt das jetzt so

der erste punkt vorne
dann punkt
dann der methodennamen
dann das zweite argument

3. Schritt

```
class Point {
    float x,y;

    static Point createPoint(float x, float y)
    { ... }
    boolean isEqualToPoint(Point other)
    { ... }
    float distanceToPoint(Point other)
    { ... }
}
```


Da man die createXXX Methode immer braucht, gibt es dafür eine Abkürzung

createPoint wird zu einem „Konstruktor“
-> auch wenn aufgrund dessen was technisch passiert der Name „Initialisierer“ vielleicht passender wäre

Auswirkungen

```
Point zero = createPoint(0,0);
```

```
Point zero = new Point(0, 0);
```


Das ermöglicht es zu schreiben
was kürzer ist als

Auswirkungen II

```
static Point createPoint
                (float x, float y){
    Point aPoint = new Point();
    aPoint.x = x;
    aPoint.y = y;
    return aPoint;
}
Point(float x, float y) {
    this.x = x;
    this.y = y;
}
```

Der Konstruktor ist auch nicht static!
Hat also AUCH einen unsichtbaren ersten Parameter „this“
Das heißt, man muss nicht mehr von hand selber ein neues objekt anlegen
-> das ist schließlich ausserhalb mit „new Point(1, 1)“ schon passiert
-> und man kann es direkt verwenden (denn es ist ja neu)
-> return fällt weg (ist ja schon zugewiesen)

```
class Point {
    float x,y;

    Point(float x, float y) {
        this.x = x;
        this.y = y;
    }
    boolean isEqualToPoint(Point other) {
        return this.x == other.x
            && this.y == other.y;
    }
    float distanceToPoint(Point other)
    { ... }
}


 Point aPoint = new Point(3, 17);
 aPoint.isEqualToPoint(new Point(3, 17));
```


Review

createPoint -> Konstruktor
Konstruktor und isEqualToPoint nutzen this

erzeugen jetzt nicht mehr mit createXX direkt, sondern über den Konstruktor

Das wars: Ihr könnt jetzt Objektorientiet programmieren.
* gibt noch kleinigkeiten: Polymorphie etc.... aber das müsst ihr selber lernen

Das ist alles was es zu Objektorientiertheit zu sagen gibt

Objekte, die ihre Methoden mitbringen

Das schwierige ist, festzustellen was zu einem Objekt gehört
aber das machen wir hier nur ein ganz bisschen – da gehört viel praxis dazu
Open Source-Software ist ganz toll dafür – sowohl für gute, als auch für schlechte Beispiele

Objektorientiert heißt: Elefant bringt „fangen“-Methode mit

Viel spaß in der Übung, wo ihr diese Konzepte jetzt umsetzen werdet