252-0027
Einführung in die Programmierung

7. Klassen und Objekte

Manuela Fischer, Malte Schwerhoff

Departement Informatik
ETH Zürich

7. Klassen und Objekte

Motivation: Softwareentwicklung im Grossen

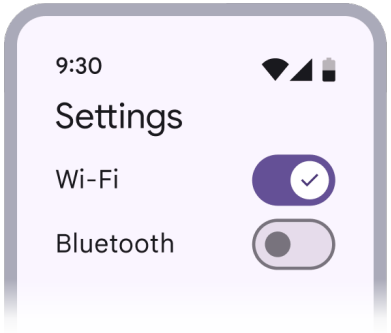
- Zur Erinnerung: Grosse Programme werden aus einzelnen *Modulen* zusammengesetzt

- Module können sich signifikant in Grösse und Funktionalität unterscheiden
  - Datenbank – (fast) alleinstehende *Anwendungen* («*applications*») / *Services* («*service*»)
  - GUI-Bibliothek – relativ grosse *Bibliotheken* («*libraries*»)
  - `java.util.Random` – eine sehr kleine Bibliothek

- In objektorientierten Sprachen sind Klassen das wichtigste Werkzeug bzw. Programmiersprachkonstrukt, um solche Module zu entwickeln

Anforderungen an solche Module

Beispiel: Umschalter («toggle») aus einer GUI-Bibliothek



1. **Zustand («state»):** An- oder ausgeschaltet; Farbe; …
2. **Funktionalität/Operationen:** Umschalten (an, aus, an, …); Abfragen, ob momentan an-/ausgeschaltet; …
3. **Instanzen**
   - Nutzung: Gleiche Funktionalität, aber individueller Zustand
   - Lebenszyklus: Erzeugen, nutzen, abräumen/löschen

- Klassen bündeln Zustand mit dazu passender Funktionalität, sie erlauben das Erzeugen vieler baugleicher Instanzen (= Objekte)

Bild mit Umschaltern: material.io

Mini-Modul `java.util.Random`

- Instanzen erzeugen

```
Random r1 = new Random(); // Zwei separate …
Random r2 = new Random(); // … Instanzen
```

- Funktionalität

```
int x = r1.nextInt(77); // 0 ≤ x < 77
double y = r1.nextDouble(); // 0.0 ≤ y < 1.0
```

- Zustand? Ja, intern, sonst wären Pseudozufallszahlen nicht reproduzierbar

```
Random r1 = new Random(0);          Random r2 = new Random(0);
int x1 = r1.nextInt();              int x2 = r2.nextInt();
int y1 = r1.nextInt();              int y2 = r2.nextInt();
                // Es gilt: x1 == x2 ∧ y1 == y2
```

- `java.util.Random` ist als Klasse implementiert

Perspektivwechsel

- Gerade erläutert: Allgemeine Anforderungen an Module
  - Zustand + Operationen (+ Instanzen erzeugen)
- Das war neu (für uns) und diente als Motivation für Klassen
- Jetzt: Perspektivwechsel hin zu uns bereits Bekanntem
  - Arrays kennen wir bereits: Speichern Daten (= Zustand?), die wir lesen und schreiben können
  - Frage: Wofür noch Klassen – reichen Arrays nicht aus?

Motivation: Sportdatenanalyse Teil II

- Sie wollen Sportdaten analysieren, z.B.
  - Durchschnittsgrösse
  - Anzahl Spieler/innen über Durchschnittsgrösse
- **Neu:** Analyse soll an Mitglieder gemailt werden

```
Anzahl Mitglieder? 3
Groesse in cm: 165
E-Mail: mf@eprog.ch
Groesse in cm: 164
E-Mail: ms@eprog.ch
Groesse in cm: 158
E-Mail: lsb@eprog.ch
Durchschnitt in cm = 162.3
Anzahl ≥ Durchschnitt: 2 (66%)
```

- Könnte wie rechts gezeigt aussehen (Eingabe unterstrichen)

Schritt 1: Daten einlesen

```
...
Scanner input = new Scanner(System.in);
System.out.print("Anzahl Mitglieder? ");
int personCount = input.nextInt();
int[] height = new int[personCount];
String[] email = new String[personCount];

// Read each person's data
for (int i = 0; i < personCount; i++) {
  System.out.print("Groesse in cm: ");
  height[i] = input.nextInt();
  System.out.print("E-Mail: ");
  email[i] = input.nextEMail();
}
...
```

- **Problem: Personendaten werden** auf zwei Arrays **verteilt**
  - Bezug nur indirekt, durch Arrayindizes (`height[i]`, `email[i]` gehören zusammen)
  - Personen z.B. nach Grösse sortieren kann Bezug zwischen Grösse und E-Mail zerstören
- Mehr Dimensionen (Name, Wohnort, …) ➜ mehr Arrays ➜ noch fehleranfälliger
- Mehrdimensionale Datensätze sind die Regel, nicht die Ausnahme

`Scanner.nextEMail()` gibt es zwar nicht, aber etwas ähnliches

Ähnliches Problem

- **Daten:** GPS-Koordinaten und Namen von Städten
- **Ziel:** Luftlinienentfernung zwischen Städten berechnen

```
Scanner input = new Scanner(…);
int count = input.nextInt();

int[] latitude = new int[count];
int[] longitude = new int[count];
String[] name = new String[count];

// Read each city's data
for (int i = 0; i < locCount; i++) {
  latitude[i] = input.nextInt();
  ...
}
...
```

Beobachtungen und Anforderungen

- Das erste Beispiel involviert Personen
  - Die Daten jeder Person sollten in einem Personen-Objekt gebündelt und gespeichert werden, denn sie gehören zusammen
  - Für Personen könnte die Funktionalität «zügelnNach(…)» oder «bekommtJugendRabatt()» sinnvoll sein
- Das zweite Beispiel involviert Städte
  - Daten einer Stadt in einem entsprechenden Stadt-Objekt speichern
  - Sinnvolle Funktionalität z.B. «entfernungVon(…)»

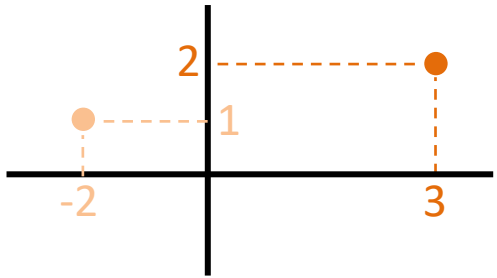
Objektorientiertes Programmieren (OOP)

- **Objektorientiertes Programmieren** («**OOP**»): *Programmierparadigma*, das ein Programm als eine Menge aufeinander einwirkender Objekte organisiert
  - Jedes Objekt stellt Methoden zur Verfügung, die seinen Zustand verändern (können) und/oder Werte zurückliefern
  - Objekte interagieren: Ein Objekt ruft Methoden anderer Objekte auf (nutzt deren Funktionalität/deren «Services»)
  - Das gewünschte Gesamtprogrammverhalten ergibt sich aus dieser Interaktion
- Andere Programmierparadigmen existieren; in *Functional Programming* lernen Sie ein weiteres kennen
- Viele Sprachen mischen mittlerweile unterschiedliche Paradigmen

Punkte im 2D-Raum: Anforderungskatalog

1. **Alle Punkte sollen baugleich sein:**
   Selbe Attribute (Daten) und Methoden (Funktionalität)

Attributname	**Beschreibung**
x	x-Koordinate
y	y-Koordinate
Methodenname	**Beschreibung**
translate(dx, dy)	Verschiebt Punkt um dx und dy
…	…




```
point.x = 3; // Setze x-Koordinate auf 3 …
point.y = 2; // … und y-Koordinate auf 2
point.translate(-5, -1); // Verschiebe x, y auf neu -2, 1
```

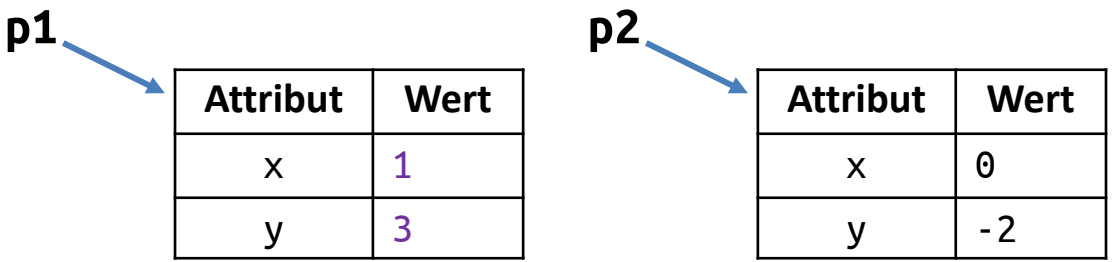
Kann so mit *jedem* Point-Objekt arbeiten

Punkte im 2D-Raum: Anforderungskatalog

1. Alle Punkte sollen baugleich sein:
   Selbe Attribute (Daten) und Methoden (Funktionalität)

2. **Verschiedene Instanzen (= Objekte) haben verschiedene Attribut*werte***

```
p1.x = 3;
p1.y = 2;
p2.x = 0;
p2.y = -2;
p1.translate(-2, 1);
```



Punkte im 2D-Raum: Anforderungskatalog

3. Wir können beliebig viele Instanzen (= Objekte) erzeugen

```
Point p1 = new Point();       // x, y standardmässig 0, 0
Point p2 = new Point(-3, 2); // x, y initialisiert auf -3, 2
Point p3 = new Point(0, 7);  // … auf 0, 7
…
```

Attribut	Wert
x	0
y	0

Attribut	Wert
x	-3
y	2

Attribut	Wert
x	0
y	7

Punkte im 2D-Raum: Anforderungskatalog

4. **Können Punkte ausgeben (→ später)**

```
Point p = new Point(-3, 2);
System.out.println(p); // Ausgabe z.B. "(-3, 2)"
```

5. **Können Punkte auf Gleichheit testen (→ viel später)**

```
Point p1 = new Point(3, 2);
Point p2 = new Point(2, 3);
p2.translate(1, -1);
if (p1.equals(p2)) { … } // true
```

7. Klassen und Objekte

Point-Klasse und deren Attribute

```
public class Point {
  int x;
  int y;
}
```

Attribute *jedes* Point-Objekts

Dieser Code deklariert eine **Klasse** («**class**») mit Namen `Point`

- `Point` ist nun ein weiterer Java-Typ (wie `String`, `Random`, …)
- Jedes `Point`-Objekt besitzt zwei **Attribute** («**attributes**»):
  Mit Namen `x` und `y`, beide vom Typ `int`

Alternative Begriffe für «attribute»: «field», «member variable», «object variable», «property»

Klassen und Attribute – Allgemein

- Ein Klasse kann *beliebig viele* Attribute *beliebigen* Typs besitzen
  - Syntax für Attributdeklarationen entspricht der von Variablen in Methoden: *type name;*
- `public class` *`Name`* muss in Datei *`Name`*`.java` abgespeichert werden
- Jedes Objekt hat seine eigenen Attribut*werte*
- Die Menge der Attributwerte bestimmt den **Zustand** («**state**») eines Objekts

```
// In Datei Name.java
public class Name {
   type1 attrName1;
   type2 attrName2;
   type3 attrName3;
   …
}
```

Point-Objekte erzeugen

```java
// in Point.java
public class Point {
  int x;
  int y;
}
```

```java
// in MyApp.java
public class MyApp {
  public static void main(String[] args) {
    Point p1 = new Point();
    Point p2 = new Point();
  }
}
```

- Der Code rechts **instanziiert** («**instantiates**») zwei `Point`-Objekte, d.h. zwei eigenständige Exemplare vom Typ `Point`
  - `p1` und `p2` sind *Referenzen*, so wie Arrays — bald mehr dazu
- Dafür wird der **new-Operator** verwendet – kennen wir schon von Arrays
  - Liefert eine Referenz auf das neu erzeugte Objekt zurück

Klasse `Point` ist nicht ausführbar/keine eigenständige Applikation (da keine `main`-Methode). Klasse `MyApp` hingegen schon.

Klassen und Objekte – Allgemein

- Eine *Klasse* ist die Vorlage (Schablone, Bauplan), die alle *Objekte* dieses (Klassen-)Typs beschreibt
  - Objekte werden gemäss dieser Vorlage erschaffen
  - *Attribute* beschreiben den möglichen Zustandsraum
  - *Methoden* (später) beschreiben Funktionalität
- Objekte sind **Instanzen** (Exemplare) einer Klasse
  - Arrays sind eine Ausnahme: Objekte ohne zugehörige Klasse
- Die Menge aller (möglichen) Instanzen einer Klasse bildet den entsprechenden Typ

```
// Vorlage
public class Name {
  type1 attrName1;
  type2 attrName2;
  …
}
```

```
// Instanzen
Name n1 = new Name();
Name n2 = new Name();
…
```

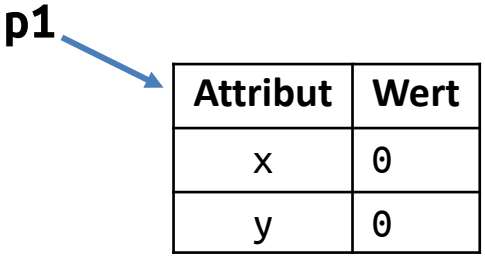
Klassen und Objekte – Allgemein

- Eine *Klasse* ist die Vorlage (Schablone, Bauplan), die alle *Objekte* dieses (Klassen-)Typs beschreibt
- Objekte müssen erzeugt werden, bevor mit ihnen gearbeitet werden kann
  - Üblicherweise mittels `new ClassName(…)`
  - Alternativ mittels Literalen (Strings, Arrays)
    - `String name = "Ravi";`
    - `double[] grades = {4.0, 5.5, 5.0, 3.75};`

Standardinitialisierung von Attributen

```java
public class Point {
  int x;
  int y;
}
```

```java
Point p1 = new Point();
// p1.x == 0 && p1.y == 0
```



Attribute werden mit *Null-äquivalenten* Werten vorbelegt (kennen wir bereits von Arrays)

- `int`: `0`, `double`: `0.0`
- `boolean`: `false`
- Referenzen (`Arrays`, `String`, …): `null`

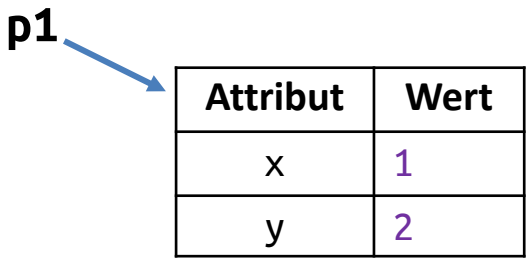
Wer es genau wissen möchte: Java Language Specs §4.12.5

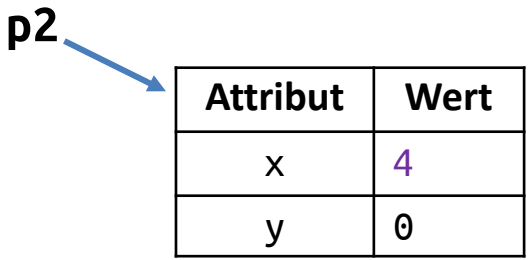
Attribute lesen und schreiben

```
public class Point {
  int x;
  int y;
}
```

```
Point p1 = new Point();
Point p2 = new Point();
```

```
p1.x = 1;
p1.y = 2;
p2.x = p1.y * 2;
println("p2.y = " + p2.y);
// Ausgabe "p2.y = 0"
```





Attribute lesen und schreiben – Allgemein

- Ein Klasse kann *beliebig viele* Attribute *beliebigen* Typs besitzen
- Auf Attribute wird mittels der **Punktnotation** («**dot notation**») zugegriffen
  - **Lesen:** `reference.attrName`
  - **Schreiben:** `reference.attrName = expr;`
- Wir sagen «die Referenz wird **dereferenziert**» («**dereferenced**») um auf ein Attribut (später auch eine Methode) des dahinterliegenden Objekts zuzugreifen

```
public class Name {
  type1 attrName1;
  type2 attrName2;
  …
}
```

```
Name n1 = new Name();

n1.attrName1 = expr1;
type2 v = n2.attrName2;
…
```

Attribute: Weiteres Beispiel

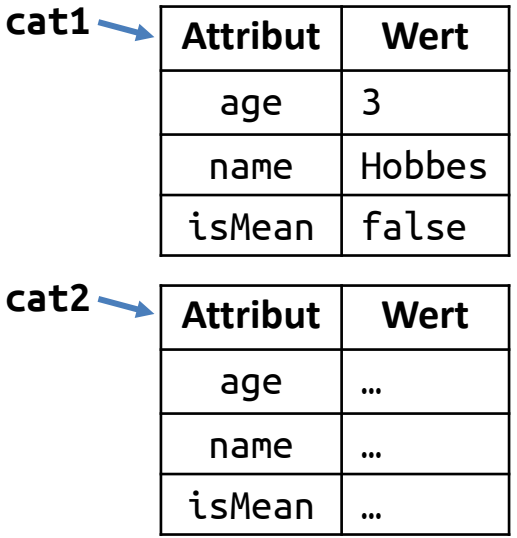
Lese- und Schreibzugriffe auf Objektattribute können auch komplexer sein bzw. Teil komplexerer Ausdrücke sein.

```
public class Cat {
  int age;
  String name;
  boolean isMean;
}
```

```
Cat cat1 = new Cat();
Cat cat2 = new Cat();
```

```
cat1.name = "Hobbes";
cat1.age = 3;

cat2.name = scanner.next(…);
if ("Scratchy".equals(cat2.name)) {
  cat2.isMean = true;
}
cat2.age++;
```



`length`-Attribut von Arrays

- Erinnerung: Jedes Array (spezielle Java-Objekte, da keine zugrundeliegende Klasse) hat das Attribut `length`
  - Lesezugriff: `myArray.length`
  - Schreibzugriff: nicht möglich

```
int[] myArray = new int[3];

// Lesezugriff
while (… < myArray.length) …

// Schreibzugriff
myArray.length = …;
```

- Auch hier: Die (Referenz)Variable `myArray` wird *dereferenziert*, um auf das `length`-Attribut zuzugreifen

Klassentypen sind Referenztypen

- Variablen vom Typ einer Klasse sind *Referenzen auf Objekt*
  - *Referenzsemantik* kennen wir bereits von Arrays
  - Können unser Wissen um Benützung & Verhalten (z.B. Seiteneffekte) von Arrays auf Klasseninstanzen (Objekte) übertragen

- Schauen wir uns im Folgenden trotzdem noch einmal an, da nicht ganz einfach

Deklaration und Initialisierung

```
Point p1 = new Point();

Point p2;
p2 = new Point();
```

- Referenzgetypte lokale Variablen
  - Werden wie üblich deklariert
  - Müssen vor der Nutzung initialisiert werden, andernfalls gibt es einen Compilerfehler

```
Point p3;
System.out.println(p3);
p3 = new Point();
```

```
Error: Variable p3 might not have been
initialized
```

`new`-Operator

Der `new`-Operator

1. Erzeugt ein neues Objekt (reserviert Speicher, default-initialisiert Attribute)
2. Liefert eine Referenz darauf zurück
   - I.d.R. speichern wir diese in einer Variable ab (z.B. `Point p = new Point()`)
   - Wir können sie aber auch «direkt» verwenden:

```
void doSomething(Point p) {
  System.out.println(p.x + "," + p.y);
  p.x = 3;
  …
}
```

```
doSomething(new Point());
// Ausgabe: "0,0"
```

Referenzen und Zuweisungen

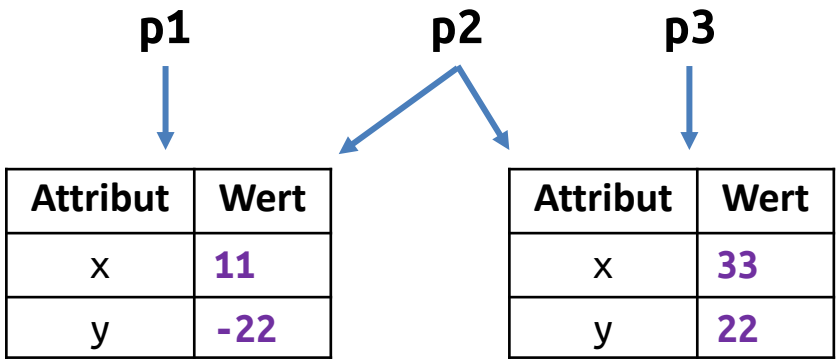
Zuweisungen zu Referenzen ändern, auf welche Objekte diese zeigen – genau wie wir das schon von Arrays kennen.

```
Point p1 = new Point();
p1.x = 11;

Point p2 = new Point();
p2.y = 22;

Point p3 = p2;
p3.x = 33;

p2 = p1;
p2.y = -22;
```



Frage: `Point`-Klasse

Welche Ausgabe wird erzeugt?

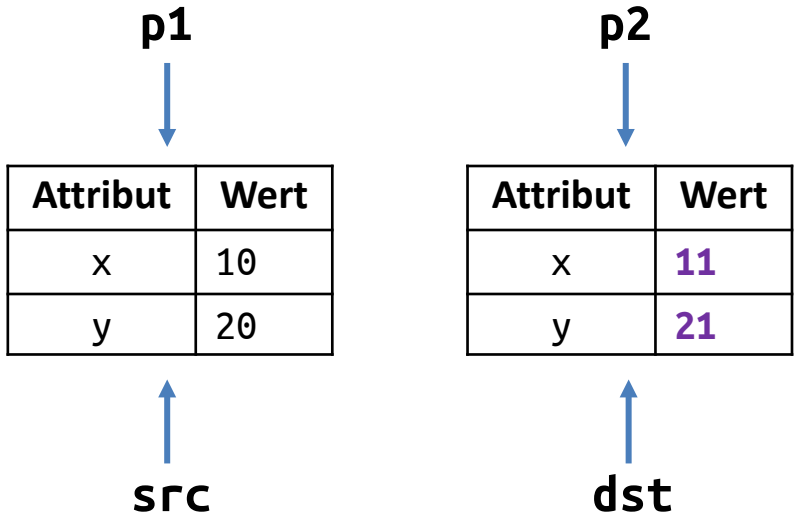
```
Point p1 = new Point();
Point p2 = p1;
p1.x = 1;
p2.y = 2;
p1 = new Point();
p2.x = p1.x + 3;
p1.y = 4;
System.out.println(
    p1.x + "," + p1.y + "," +
    p2.x + "," + p2.y);
```

Referenzen und Methoden

Bei Methodenaufrufen werden Objekte per Referenz übergeben – genau wie wir das schon von Arrays kennen.

```
Point p1 = new Point();
p1.x = 10;
p1.y = 20;
Point p2 = new Point();
copyAndInc(p1, p2);
```

```
void copyAndInc(Point src, Point dst) {
  dst.x = src.x + 1;
  dst.y = src.y + 1;
}
```



Frage: `Point`-Klasse und Methodenaufrufe

Welche Ausgabe wird erzeugt?

```
Point p1 = new Point();
Point p2 = p1;
p1.x = 10;
p1.y = 20;
copyAndInc(p1, p2);
System.out.println(
    p1.x + "," + p1.y + "," +
    p2.x + "," + p2.y);
```

```
void copyAndInc(Point src, Point dst) {
  dst.x = src.x + 1;
  dst.y = src.y + 1;
}
```

Objekte und `null`: Anwendungen

- Erinnerung: Eine Referenzvariable kann den Wert `null` haben
  - Bedeutet, dass diese Variable nicht mehr/noch nicht auf ein Objekt zeigt
  - `null` kann nicht *dereferenziert* werden → `NullPointerException`
- Anwendungen
  1. [Häufig] Ausdrücken, dass (z.B.) ein Attribut aktuell nicht gesetzt/verfügbar ist.
     Hier: Keine Elternadresse (mehr) hinterlegt.
  2. [Selten] Intention explizit machen: Variable als «wird nicht mehr gebraucht» bzw. «soll nicht mehr gebraucht werden» markieren.

```
Student me = new Student();
…
me.parentAddress = null;
```

```
Point loc = new Point();
…
loc = null;
```

Objekte und `null`: Anwendungen

3. Standardwert für die Initialisierung von referenzgetypten *Attributen*

```
public class Cat {
  int age;
  String name;
}
```

```
Cat kitty = new Cat();
…println(kitty.name); // "null"
…println(kitty.name.length());
  // NullPointerException, da kitty.name == null
```

4. Standardwert für die Initialisierung von referenzgetypten *Arrayelementen*

```
Cat[] kittens = new Cat[3]; // cat == {null, null, null};
String allCaps = kittens[1].name.toUpperCase();
  // NullPointerException, da kittens[1] == null
```

`NullPointerExceptions` verhindern

- Prüfen, ob eine Referenz `null` ist
  - `if (kittens[i] != null) {… do something with kittens[i] …}`

- Verhindert das Auftreten einer *Exception* (Laufzeitfehler)
  - Wir lernen später mehr über Exceptions und auch wie ein Programm auf Exceptions reagieren kann.
  - Aber: Exceptions, bzw. darauf zu reagieren, sollte die Ausnahme sein – ein Programm sollte das Auftreten so gut wie möglich verhindern.

Kombination: Arrays, Objekte, `null`

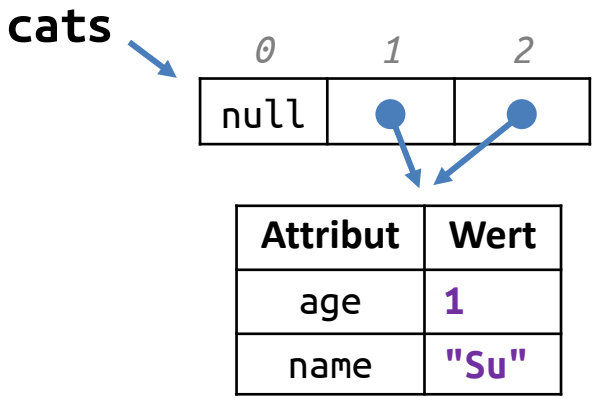
- Können unser bisheriges Wissen kombinieren

```
Cat[] cats = new Cat[3];

cats[1] = new Cat();

cats[1].age = 1;
cats[1].name = "Su";

cats[2] = cats[1];
```



- Referenzgetypte Arrays werden mit `null` vorbelegt
- Referenzgetypte Attribute werden mit `null` vorbelegt
- Zuweisungen zu Referenzen erzeugen Alias, keine Kopien

Kombination: Arrays, Objekte, `null`

Ziel: Methoden, die Katzennamen mit mehr als *N* Buchstaben ausgibt

(Wir nehmen hier an, dass `cats != null`)

```
…
printNames(cats, 3);
```

v1

```
void printNames(Cat[] cats, int minLen) {
  for (int i = 0; i < cats.length; i++) {
    if (cats[i].name.length() >= minLen) {
      System.out.println(cats[i].name);
    }
  }
}
```

Problem(e)?

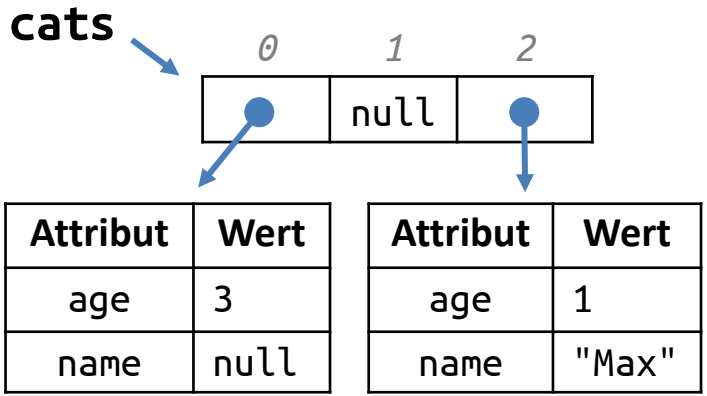
Kombination: Arrays, Objekte, `null`

Problem vorherige Code (v1): Falls `cats[i] == null`, dann produziert die Dereferenzierung **`cats[i].`**`name` eine `NullPointerException`

v2

```
void printNames(Cat[] cats, int minLen) {
  for (int i = 0; i < cats.length; i++) {
    if (cats[i] != null &&
        cats[i].name.length() >= minLen) {

      System.out.println(cats[i].name);
    }
  }
}
```



Problem?

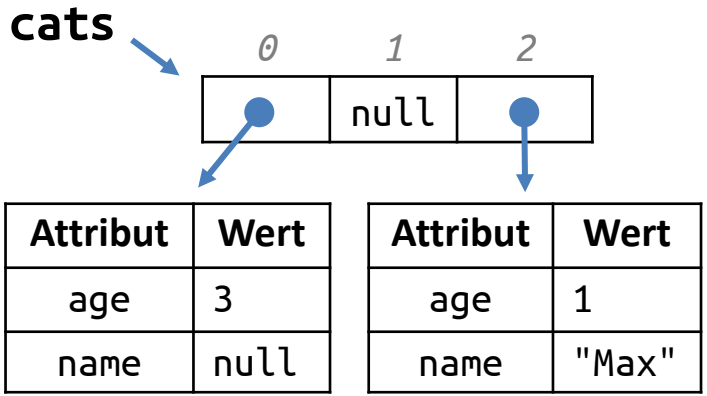
Kombination: Arrays, Objekte, `null`

Problem vorherige Code (v2): Falls `cats[i].name == null`, dann produziert `cats[i].name.length()` eine `NullPointerException`

v3

```
void printNames(Cat[] cats, int minLen) {
  for (int i = 0; i < cats.length; i++) {
    if (cats[i] != null &&
        cats[i].name != null &&
        cats[i].name.length() >= minLen) {

      System.out.println(cats[i].name);
    }
  }
}
```



Dieses Vorgehen – erst prüfen, ob Dereferenzierung möglich – ist weit verbreitet

Fernsehgerät: huffpost.com | SRF-Beitrag: youtube.com

Frage: Verhalten äquivalent?

```
void transform(Cat[] cats) {
  for (int i = 0; i < cats.length; i++) {
    if (cats[i] != null &&
        cats[i].name != null) {

      cats[i].name =
        cats[i].name.toUpperCase();
    }
  }
}
```

```
void transform(Cat[] cats) {
  for (int i = 0; i < cats.length; i++) {
    Cat cat = cats[i];

    if (cat != null &&
        cat.name != null) {

      cat.name = cat.name.toUpperCase();
    }
  }
}
```

7. Klassen und Objekte

Motivation: Arbeitszeit berechnen

- Die Datei `hours.txt` enthält die folgenden Daten:

ID	Name	Arbeitsstunden (jeweils ein Tag)
123	Paula	12.5 8.1 7.6 3.2
456	Erik	4.0 11.6 6.5 2.7 12
789	Linqi	8.0 8.0 8.0 8.0 7.5

- Ziel: Programm schreiben, das über Arbeitsstunden und –lohn informiert. Mögliche Informationen bzw. Ausgabe:

```
Paula (ID#123) worked 31.4 hours (7.85 hours/day), earned 964 CHF
Erik (ID#456) worked 36.8 hours (7.36 hours/day), earned 1130 CHF
Linqi (ID#789) worked 39.5 hours (7.9 hours/day), earned 1213 CHF
```

Was brauchen wir?

Modellierung von Personen durch eine Klasse `Person`

- Attribute
  - ID – `int`
  - Name – `String`
  - Arbeitsstunden – `double[]`
  - …
- Funktionalität
  - Arbeitsstunden aufsummieren
  - Stundenlohn mal Arbeitsstunden
  - Durchschnitt Stunden pro Tag
  - …

Erste Bausteine

```
public class Person {
  int id;
  String name;
  double[] hours;
}
```

```
public static Person readPerson(Scanner source) { … }
```

```
Person[] staff = new Person[input.nextInt()];
Scanner input = new Scanner(new File("hours.txt"));
for (int i = 0; i < staff.length; i++) {
  staff[i] = readPerson(input);
}
```

- Klasse `Person`
- Methode `readPerson()` um ein `Person`-Objekt aus `hours.txt` auszulesen
- Schleife (z.B. in `main()`) um alle Personen in ein Array einzulesen

Dateien (`new File(…)`) kommen später. Blosses Wissen, dass wir Daten aus einer Datei auslesen können, reicht hier aus.

Gewünschte Berechnung

- Gegeben: Angestellte als Array `Person[] staff`
- Berechne (z.B. in `main()`): Gesamtlohn aller Angestellten

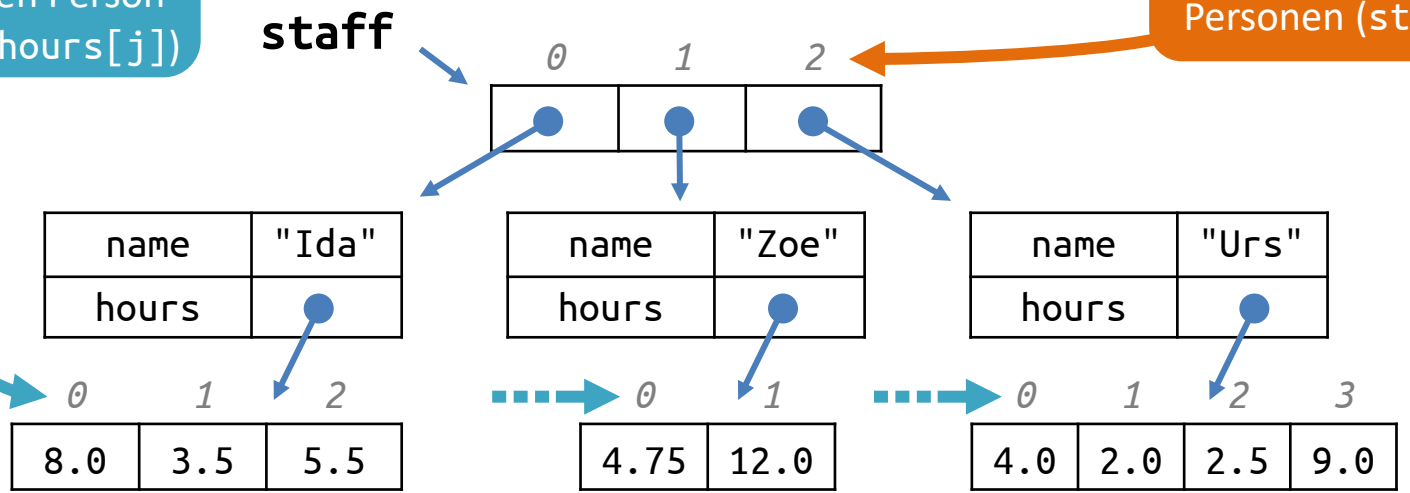
```
// Compute staff's total salary
double totalSalary = 0;
for (int i = 0; i < staff.length; i++) {
  // Sum up single staff member's working hours
  double sum = 0;
  for (int j = 0; j < staff[i].hours.length; j++) {
    sum += staff[i].hours[j];
  }
  totalSalary += sum * 30.70; // Add single salary to total
}
```

Gewünschte Berechnung: Reingezoomt

```
for (int i = 0; i < staff.length; i++) {
  double sum = 0;
  for (int j = 0; j < staff[i].hours.length; j++) {
    sum += staff[i].hours[j];
  }
  totalSalary += sum * 30.70;
```

Innere Schleife:
j iteriert über Stunden der aktuellen Person (staff[i].hours[j])

Äussere Schleife:
i iteriert über Personen (staff[i])



Beobachtungen

```
double totalSalary = 0;
for (int i = 0; i < staff.length; i++) {
  double sum = 0;
  for (int j = 0; j < staff[i].hours.length; j++) {
    sum += staff[i].hours[j];
  }
  totalSalary += sum * 30.70;
}
```

- Berechnung des Gehalts pro Person
  - Ist für *sich allein genommen* eine *sinnvolle* Berechnung …
  - … die auch für *andere Berechnungen gebraucht* wird, z.B. Durchschnittslohn pro Tag
- Chance hoch, dass der Code an vielen Stellen (und in verschiedenen Programmen) immer wieder auftaucht → **Redundanz** (Copy & Paste)

Wiederverwendbarkeit durch Methoden

Statt Code zu kopieren, verschieben wir die Berechnung in eine Methode

- Code nun wiederverwendbar (Methodenaufruf)

```
public static double computeSalary(Person person) {
  double sum = 0;
  for (int j = 0; j < person.hours.length; j++) {
    sum += person.hours[j];
  }
  return sum * 30.70;
}
```

```
// Z.B. in main()
double totalSalary = 0;
for (int i = 0; i < staff.length; i++) {
  totalSalary += computeSalary(staff[i]);
}
```

Noch besser (da flexibler) wäre es, den Stundenlohnt von 30.70 als einen zweiten Parameter mitzugeben

Zwischenstand: Gesamtsicht

```java
public class Person {
  int id;
  String name;
  double[] hours;
}
```

Beobach-
tungen?

```java
public class StaffManager {
  public static void main(…) {
    … // Read staff from file

    double totalSalary = 0;
    for (int i = 0; i < staff.length; i++) {
      totalSalary += computeSalary(staff[i]);
    }
  }

  public static double computeSalary(Person person) {
    double sum = 0;
    for (int j = 0; j < person.hours.length; j++) {
      sum += person.hours[j];
    }
    return sum * 30.70;
  }
}
```

Probleme mit dieser Programmstruktur

1. Problem: Weiterhin (mögliche) **Redundanz**

- `computeSalary(…)` ist Teil von `StaffManger` (dem Klienten)
  - Viele Klienten müssten `computeSalary(…)` implementieren
  - Diese Implementationen arbeiten *ausschliesslich* mit den Attributen von Person-Objekten, sind also *unabhängig* von den Klienten selbst

Einschub: Begriff «**Klient**» («**client**»)

- Klient = Code der einen Typ/Klasse bzw. dessen Objekte nutzt
- Hängt von der Perspektive ab: A Klient von B, B Klient von C (oder A)

Probleme mit dieser Programmstruktur

2. Problem: `computeSalary(…)` muss **Implementationsdetails** («Innenleben») der Klasse `Person` kennen

```
// Auszug computeSalary(Person person)
for (int j = 0; j < person.hours.length; j++) {
  sum += person.hours[j];
}
```

- Implementationsdetails (z.B. Attribute) können sich ändern
- Änderungen an Klasse `Person` brechen möglicherweise die (vielen) `computeSalary()`-Implementation(en)
- Erschwert das Warten und das Weiterentwickeln von Software

Eine mögliche Änderung

- Array `hours` enthält die Anzahl Stunden, die eine Person an einem Tag gearbeitet hat
- Was, wenn wir zwischen Normalzeit und Überstunden unterscheiden wollen?
  - Ersten acht Stunden/Tag sind Normalzeit → 30,70 CHF pro Stunde
  - Weitere Stunden gelten als Überstunden → 25% Zuschlag
- Konsequenz: `Person`–Klasse *und* `computeSalary(…)` in *allen* Klienten müssen angepasst werden

```
public class Person {
  int id;
  String name;
  double[] hours;
}
```

Neu: Mit Überstunden

```
public class Person {
  int id;
  String name;
  double[] hours;
  double[] overtime;
}
```

Mussten Person *und* Klienten anpassen …

```
public static double computeSalary(Person person) {
  double sumStd = 0;
  for (int j = 0; j < person.hours.length; j++) {
    sumStd += person.hours[j];
  }
  double sumOvt = 0;
  for (int j = 0; j < person.overtime.length; j++) {
    sumOvt += person.overtime[j];
  }
  return sumStd * 30.70 + sumOvt * 38.40;
}
```

Modularität verbessern

- **Allgemeines Ziel: Modularität**
  - Wir wollen Klassen («Dienstleister») entwickeln, die (innerhalb eines bestimmten Rahmens) von den Klienten unabhängig (weiter)entwickelt werden können
  - Andere Perspektive: Wir wollen Klienten entwickeln, die von der Implementation der Klassen («Dienstleister») unabhängig sind.
- **Vorgehen: Daten und Operationen bündeln**
  - Bekannt: Klassen legen internen Aufbau (Attribute) von Objekten fest, d.h. woraus sich deren Zustand ergibt
  - Neu: Objekte können auch Operationen («Dienste») auf ihrem Zustand anbieten
    - Berechnung des Gehaltes (`computeSalary(…)`) ist eine Operation, die eng mit den Daten eines `Person`-Objektes verbunden ist
    - *Jeder* `Person`-Klient kann dann `person.computeSalary()` nutzen

`computeSalary()` als Objektmethode

```
public class Person {
  int id;
  String name;
  double[] hours;
  double[] overtime;
  public static double computeSalary(Person person) {
    double sumStd = 0;
    for (int j = 0; j < this.hours.length; j++) {
      sumStd += this.hours[j];
    }
    double sumOvt = 0;
    for (int j = 0; j < this.overtime.length; j++) {
      sumOvt += this.overtime[j];
    }
    return sumStd * 30.70 + sumOvt * 38.40;
  }
}
```

- Methode jetzt innerhalb der Klasse
- Kein `static`
  (sonst keine Objektmethode)
- `Person` kein Parameter mehr
  - Objektmethoden haben stets Zugriff auf die Attribute *ihres* Objekts (= `this`)
  - Aber woher kennen Objektmethoden *ihr* Objekt?

Terminologie: *Objekt-* oder *Instanzmethode*; auch nur *Methode*

Expliziter Parameter vs. implizites Aufruferobjekt (this)

Es seien `p1`, `p2` zwei `Person`-Objekte.

Bisher (keine Objektmethode)

```
// Ausserhalb Klasse Person
static double computeSalary(Person p) {
… p.hours … // p1/p2.hours
}
```

```
double s1 = computeSalary(p1);
double s2 = computeSalary(p2);
```

Zugriff auf Attribute eines *expliziten* Methodenparameters.

Neu (Objektmethode)

```
// In Klasse Person
double computeSalary() {
… this.hours … // p1/p2.hours
}
```

```
double s1 = p1.computeSalary();
double s2 = p2.computeSalary();
```

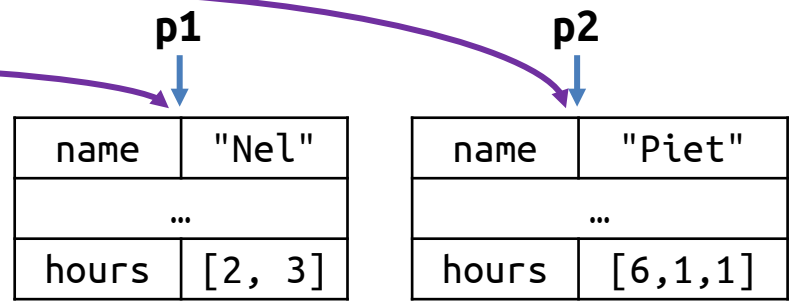
Zugriff auf Attribute des **Aufruferobjekts** («**receiver object**») via **`this`**-Referenz (*impliziter* Methodenparameter).

Objektaufruf und `this` am Beispiel (animiert)

```
public class Person {
  double[] hours;
  …

  public double computeSalary() {
    double sumStd = 0;
    for (int j = 0; j < this.hours.length; j++) {
      sumStd += this.hours[j];
    }
    … // Überstundenberechnung fehlt hier
    return sumStd * 30.70;
  }
}
```

```
// p1, p2 Person-Objekte
double s1 = p1.computeSalary(); // 153.50
double s2 = p2.computeSalary(); // 245.60
```



In jeder Objektmethode ist `this` eine Referenz vom Typ der entsprechenden Klasse (hier: `Person`)

Visualisierung vereinfacht: String `name` und Array `hours` (Referenzen) jeweils separate Objekte

Bekanntes gilt weiterhin

1. Aufruferreferenz darf nicht `null` sein

```
Person[] bosses = new Person[3];
bosses[0].computeSalary();
```

```
java.lang.NullPointerException:
Cannot invoke "Person.computeSalary()"
because "bosses[0]" is null
```

2. Auch Objektmethoden können beliebig viele Parameter deklarieren

```
// In class Person
public double computeSalary(double rateStd, double rateOvt) {
  … // Stundenberechnung
  return sumStd * rateStd + sumOvt * rateOvt;
}
```

```
Person p = …;
double salary =
  p.computeSalary(30, 30.01);
```

3. Auch Objektmethoden müssen nichts zurückgeben (Rückgabetyp `void`)

```
// In class Person
public void printSalary() { … }
```

Delegierende Objektmethoden

```
public class Person {
  …

  public double getStandardSalary() {
    // … this.hours …
    return sumStd * 30.70;
  }

  public double getOvertimeSalary() {
    // … this.overtime …
    return sumOvt * 38.40;
  }

  public double getSalary() {
    return this.getStandardSalary() + this.getOvertimeSalary();
  }

  public void printSalary() {
    System.out.println(this.name + " earned " + this.getSalary());
  }
}
```

Auch Objektmethoden können *andere* (Objekt-)Methoden aufrufen.

```
Person p = …;
p.printSalary();
```

Aufruf `p.printSalary()` ➜ alle `this` auf dieser Folie beziehen sich zur Laufzeit auf Objekt `p`.

Die Objektmethoden werden also hier im Kontext des Objekts `p` ausgeführt.

Explizites `this`-Dereferenzieren unnötig …

Statt `this.attribute` auch nur `attribute` möglich, d.h. `this` kann weggelassen werden (und wird dann implizit dereferenziert).

```
public class className {
  T attribute;
  …

  public T' methodName(…) {
    … attribute …
  }
}
```

=

```
public class className {
  T attribute;
  …

  public T' methodName(…) {
    … this.attribute …
  }
}
```

Andere Syntax, selbe Semantik

… ausser bei Shadowing

- Lokale Variablen (inkl. Parameter) dürfen selben Namen wie Attribute haben
- Attribute werden durch gleichnamige Variablen **verdeckt** («**shadowing**»)

```
public class Counter {
  int value;

  public void incrementBy(int value) {
    value = value + value;
  }
}
```

Alle drei `value` beziehen sich auf den Parameter, *nicht* das Attribut

- Explizites `this` kann dann genutzt werden, um auf Attribute zuzugreifen

```
this.value = this.value + value;
```

Jetzt wird das `value`-Attribute erhöht

Ein neues Beispiel: Klasse `Point`

```java
// A point represents an (x, y) location.
public class Point {
  int x;
  int y;
}
```

Methode	Beschreibung
`getSlope()`	Gibt die Steigung dieses Punktes (Linie zum Ursprung) zurück
`setLocation(x, y)`	Setze die x- und y-Koordinaten dieses Punktes
`translate(dx, dy)`	Verschiebt diesen Punkt um (dx, dy)
`getDistance(p)`	Entfernung zwischen diesem Punkt (x_1, y_1) und Punkt p (x_2, y_2): $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
`getDistanceFromOrigin()`	Entfernung zwischen diesem Punkt und dem Ursprung/Nullpunkt

Java: `Math.sqrt(x)` für $\sqrt{x}$; `Math.pow(x, 2)` für x^2.

Terminologie: Attribut-lesende/-schreibende Objektmethoden

```
// A point represents an (x, y) location.
public class Point {
  int x;
  int y;

  public double getSlope() {
    return y / (double) x;
  }

  public void setLocation(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

- Objektmethoden die Attribute nur lesen heissen **Accessor** oder **Getter**
  - Z.B. `getSlope()`
- Falls (auch) schreibend: **Mutator** oder **Setter**
  - Z.B. `setLocation()`

`getSlope()`: `x == 0` möglich ➜ Sonderfall (Fehlermeldung? Unendlich?), den wir hier der Einfachheit halber ignorieren.

Direkter Attributzugriff oder Delegation

```
public class Point {
  int x;
  int y;

  …

  public void translate(
      int dx, int dy) {

    x = x + dx;
    y = y + dy;
  }
}
```

`translate()`: Zwei mögliche Implementationen

- Links: Direktes Setzen von `x, y`
- Unten: Indirektes setzen via Delegation

```
public void translate(
    int dx, int dy) {

  setLocation(x + dx, y + dy);
}
```

Unterschiede hier vernachlässigbar. Allgemein:

- Delegation → Code-Wiederverwendung
- Indirektion → Evtl. schwieriger zu verstehen, was genau passiert

Zugriff auf eigene und andere Attribute

```java
// In class Point
public double getDistance(Point other) {
  double dx = other.x - x;
  double dy = other.y - y;

  return Math.sqrt(dx * dx + dy * dy);
}
```

- Berechnung der Entfernung durch (Lese-) Zugriff auf die Attribute des eigenen und des anderen Objekts.
- Hier nicht gebraucht, aber möglich: Schreibzugriff auf Attribute beider Objekte

```java
Point p = new Point();
p.x = 2;
p.y = -2;

Point q = new Point();
q.x = 5;
q.y = 4;

System.out.println(
  p.getDistance(q)); // 6.7
```

`getDistance()`: `other == null` möglich ➜ Sonderfall (Fehlermeldung), den wir hier der Einfachheit halber ignorieren.

Mehr Potential für Wiederverwendung

```
// In class Point
public double getDistanceFromOrigin() {
  return Math.sqrt(x * x + y * y);
}
```

```
// Alternative: Delegation
public double getDistanceFromOrigin() {
  return getDistance(new Point());
}
```

x, y des neuen Objekts mit `0` vorbelegt

- Delegation macht erneute Implementation der Distanz-Formel unnötig
- Delegation insbesondere dann sinnvoll, wenn wiederverwendete/r Berechnung/Code komplex bzw. umfangreich

7. Klassen und Objekte

Objekte als String repräsentieren: `toString()`

```
Point p = new Point();
p.x = 2;
p.y = -2;

…println(p);  // Point@2b39
…println("p = " + p); // Point@2b39
```

=

```
…println(p.toString());
…println("p = " + p.toString());
```

- Aktuell: Speicheradresse des Objekts als Ausgabe → Nicht hilfreich
- Ziel: Sinnvolle Repräsentation ausgeben, z.B. `"p = (2, -2)"`

Anderes Vorgehen, selbes Resultat

- `println(p)`: Ruft intern `p.toString()` auf — können wir auch extern, wie rechts gezeigt
- Standardimplementation von `toString()` gibt *`Klassenname@Speicheradresse`* aus
- String-Konkatenation `"p = " + p`:
  - Ein String wird benötigt → Compiler schreibt `… + p` um zu `… + p.toString()`
  - Können wir auch selbst, wie rechts gezeigt

Eigene `toString()`-Implementation

- Für eigene Typen kann `toString()` **überschrieben** werden («**override**»), indem die Klasse eine entsprechende Methode implementiert.
- Ermöglicht eine sinnvolle textuelle Standardrepräsentation

```
public class Point {
  int x;
  int y;

  public String toString() {
    return "(" + x + ", " + y + ")";
  }
}
```

```
Point p = new Point();
…println(p); // (0, 0)
p.x = 2;
p.y = -2;
…println("p = " + p); // p = (2, -2)
```

Für Vorwisser: `@Override` kommt später

Eigene `toString()`-Implementation

- **Wichtig:** Die überschreibende `toString`-Methode muss die folgende Signatur haben: `public String toString()`
- Andernfalls deklarieren wir «bloss» eine weitere, gewöhnliche Methode

```
public class Point {
  int x;
  int y;

  public String toString(String sep) {
    return "(" + x + sep + y + ")";
  }
}
```

Interner Aufruf von *nicht überschriebener* `toString()`

```
Point p = new Point();
…println(p); // Point@3632be31
…println(p.toString(":")); // (0:0)
```

Expliziter Aufruf von `toString(String)`

Verschachtelte Strukturen: Arrays

Wird ein Array ausgegeben (Erinnerung: via `Arrays.toString`), dann wird die `toString()`-Methode jedes Array-Elements aufgerufen

```
public class Point {
  int x;
  int y;

  public String toString() {
    return "(" + x + ", " + y + ")";
  }
}
```

```
Point[] trace = new Point[2];
trace[0] = new Point();
trace[0].y = 1;
trace[1] = new Point();
trace[1].x = 3;
```

```
…println(trace);
  // [LPoint;@4f970963
```

```
…println(Arrays.toString(trace));
  // [(0, 1), (3, 0)]
```

Eigene verschachtelte Strukturen

```
public class Point {
  … // Wie auf vorheriger Folie
}
```

```
public class Line {
  Point from;
  Point to;

  public String toString() {
    return from + " -> " + to;
  }
}
```

```
Line line = new Line();
line.from = new Point();
line.to = new Point();
line.to.x = 4;
line.to.y = -1;
…println(line); // (0, 0) -> (4, -1)
```

Die String-Repräsentation komplexerer Objekte wird oft aus den String-Repräsentationen ihrer Einzelteile zusammengesetzt.

7. Klassen und Objekte

Erzeugen von Objekten

```
Point location = new Point(); // x/y beide 0
location.x = -3;
location.y = 11;
```

- Objekterzeugung bisher in zwei Schritten
  1. Neues Objekt instanziieren (und Attribute standard-initialisieren)
  2. Den Attributen die gewünschten Werte zuweisen
- Ziel: Attribute direkt bei der Objektinstanziierung wie gewünscht setzen

```
Point p1 = new Point();       // x/y beide 0
Point p2 = new Point(-3, 11);// x/y gleich -3/11
Point p3 = new Point(p2);     // x/y gleich p2.x/y
```

Bereits gesehen

```
new Scanner(System.in);
```

Konstruktoren

Ein **Konstruktor** («**constructor**») initialisiert den Zustand eines neuen Objektes.

```
// In class className
public className(T1 param1, …) {
  statements
}
```

- Deklarationssyntax: Wie Methode, aber *ohne* Rückgabetyp
- Methodenname muss gleich Klassenname sein

- Wird ausgeführt (aufgerufen), wenn der `new`-Operator angewandt wird
- Hat *keinen* Rückgabewert
- Konstruktoren sind keine (normalen) Methoden – mehr in späteren Kapiteln

Beispiel-Konstruktoren für Klasse `Point`

```java
public class Point {
  int x;
  int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

```java
// In class Point
public Point(Point other) {
  assert other != null;

  this.x = other.x;
  this.y = other.y;
}
```

```java
// In class Point
public Point() {
  this.x = 0;
  this.y = 0;
}
```

```java
Point p1 = new Point(2, 9);
// p1.x == 2, p1.y == 9
```

```java
Point p2 = new Point(p1);
// p2.x == 2, p2.y == 9
```

```java
Point p3 =
  new Point();
// p3.x/y beide 0
```

Mehrere Konstruktoren → selbe Regeln wie für das Überladen von Methoden

Standardkonstruktor

- Eigene Konstruktoren (wie die eben gezeigten) sind *optional*
- Wenn *kein* eigener Konstruktor deklariert wird, stellt Java den **Standard/Default-Konstruktor** («**default constructor**») bereit
  - Hat keine Parameter
  - Setzt alle Attribute auf Standardwerte (`0`, `null`, usw.)

```
public class Point {
  int x;
  int y;
}
```

```
Point p = new Point();
// p.x/y beide 0
```

OK: Kein eigener Konstruktor, Standardkonstruktor daher verfügbar

```
// In class Point
public Point(int x, int y) {
  this.x = x;
  this.y = y;
}
```

```
Point p = new Point();
```

Compilerfehler: Standard-konstruktor nicht verfügbar

Anderes Beispiel: Klasse `User`

```
public class User {
  String name;    // Pflicht
  int age;        // -1 falls unbekannt
  Point location; // null falls unbekannt

  public User(String name, int age, Point location) {
    this.name = name;
    this.age = age;
    this.location = location;
  }
}
```

- Jetzt möglich: `new User("Jana", 75, currentLocation)`
- Auch möglich: `new User("Jana", -1, null)`.
  Aber stattdessen nur `new User("Jana")` wäre praktischer.

Delegierende Konstruktoren

- Konstruktoren können andere Konstruktoren aufrufen
  - Syntax: `this(`*`parameters`*`)`
  - Falls genutzt: Muss die *erste* Anweisung im Rumpf des Konstruktors sein!

```
// In class User
public User(String name, int age, Point location) {
  this.name = name;
  this.age = age;
  this.location = location;
}
```

```
// In class User
public User(String name) {
  this(name, -1, null);
  // this.name == name
  // this.age == -1
  // this.location == null
}
```

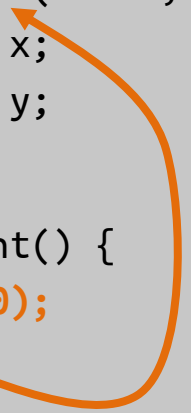
- Verhindert Redundanz in Konstruktoren

Delegierender `Point`-Konstruktor

```
public class Point {
  int x;
  int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public Point() {
    this(0, 0);
  }
}
```



Auch der *parameterlose* Konstruktor kann oft mittels Delegation implementiert und so (wieder) angeboten werden

- Insbesondere, wenn es sinnvolle Standardwerte für alle Attribute gibt

```
Point origin = new Point();
// origin.x == origin.y == 0
```

Konstruktoren: Häufige Fehler

1. Versehentliches Verstecken von Attributen durch lokale Variablen:

```
public class Point {
  int x;
  int y;

  public Point(int initialX, int initialY) {
    int x = initialX;
    int y = initialY;
  }
}
```

- Lokale Variablen x, y existieren nur während der Konstruktorausführung
- Attribute x, y wurden nicht auf die gewünschten Initialwerte gesetzt

Konstruktoren: Häufige Fehler

2. Versehentlich einen Konstruktor-Rückgabetyp deklarieren:

```
public class Point {
  int x;
  int y;

  public void Point(int initialX, int initialY) {
    x = initialX;
    y = initialY;
  }
}
```

- Kein Konstruktor, sondern bloss eine reguläre Methode
- Erlaubt, aber vermutlich nicht gewollt

Konstruktoren: Häufige Fehler

3. Parameterlosen Default-Konstruktor verwenden, wenn dieser aufgrund eigener Konstruktoren nicht zur Verfügung steht

```
public class Point {
  int x;
  int y;

  public Point(int initialX, int initialY) {
    …
  }
}
```

```
Point origin = new Point();
```

```
Unresolved compilation problem:
Constructor Point() is undefined
```

Einschub: Attribut-Initialisierung mit Konstanten

- Können Attribute auch deklarieren *und direkt initialisieren*

```
public class Lunch {
  String when = "12:00";
}
```

```
Lunch lunch = new Lunch();
// lunch.when == "12:00"
```

- Derartige initialisierte Attribute
  - Werden vom Default-Konstruktor (falls vom Compiler erstellt) *nicht* mehr auf null-äquivalente Werte gesetzt
  - Können von Konstruktoren weiterhin auf beliebige Werte gesetzt werden
- Manchmal praktisch, aber eher selten verwendet. Für uns nicht weiter relevant.

7. Klassen und Objekte

Motivation

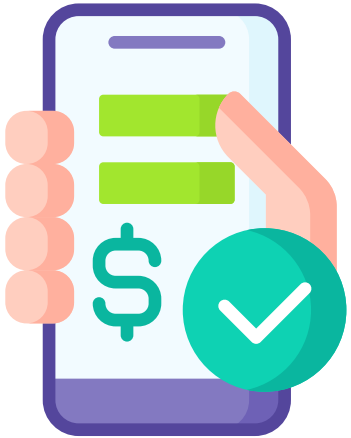
- `Rational`-Objekte modellieren rationale Zahlen (Brüche), z.B. $\frac{1}{3}$ oder $\frac{6}{8}$

```
public class Rational {
  int numerator;   // Zähler
  int denominator; // Nenner != 0
}
```

- Objektzustand nur *konsistent*, falls `denominator != 0`
  - Eine solche Konsistenzbedingung heisst auch *Objektinvariante*
- Problem: Attribute können aktuell von (allen) Klienten beliebig geändert werden. Z.B. `rat.denominator = 0;`.
- Ziel: Objektzustand kapseln und nur bestimmte Änderungen zulassen

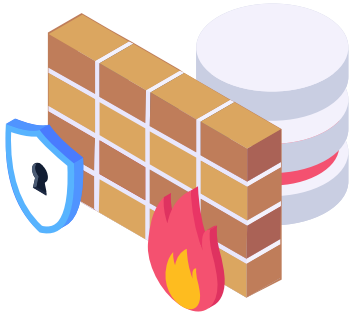
Analogie

- Ziel: Zustand kapseln, nur bestimmte Änderungen zulassen
- Analogie: Kontoverwaltung
  - Bank erlaubt keine *direkte* Änderung der Kontostände (`meinKonto.guthaben += 10000`)
  - Nur indirekte Änderung erlaubt (Einzahlen, Abheben, Überweisen, …), für die bestimmte Regeln durchgesetzt werden. Beispielsweise:
    - Kann keinen negativen Betrag abheben
    - Überweisung von Konto K_1 nach K_2 → Gesamtvermögen gleich



«Encapsulation» (Datenkapselung)

- Ziel: Zustand kapseln, nur bestimmte Änderungen zulassen
- Kernideen hinter «**Encapsulation**» (**Datenkapselung**)
  1. Objektzustand (= Attribute) vor direktem Klientenzugriff schützen
  2. Konstruktoren implementieren, die konsistenten Zustand erzwingen
  3. Lesezugriff via Accessor/Getter-Methoden kontrollieren
  4. Schreibzugriff via Mutator/Setter-Methoden kontrollieren



- Vorteile (mindestens)
  1. Konsistenzbedingungen (Objektinvarianten) garantieren → gleich
  2. Änderungen/Weiterentwicklung vereinfachen → etwas später
- Nachteil (mindestens): Komplexerer (oder zumindest mehr) Code

Sichtbarkeit von Attributen

- Attribute können durch **Zugriffsmodifikatoren** («**access modifiers**») in ihrer **Sichtbarkeit** («**visibility**») eingeschränkt werden
- Diese Modifier sind Teil der Attribut-Deklaration
  - `public` *`type attribute`* – Für alle Klienten sichtbar
  - `protected` *`type attribute`* – Für Vererbung relevant ➜ später
  - *`type attribute`* – Für Java-Packages relevant ➜ später
  - `private` *`type attribute`* – Nur für Objekte derselben Klasse sichtbar

Haben wir bisher genutzt
(ohne Packages für uns wie `public`)

Beispiel: Encapsulation für `Rational`

1. Schritt: Attribute vor Klienten verstecken → `private`

2. Schritt: Konstruktor, der konsistenten Zustand garantiert

```
public class Rational {
  private int num;
  private int den; // INV: != 0

  public Rational(int n, int d) {
    assert d != 0; 1
    num = n;
    den = d;
  }
}
```

Laufzeitfehler, falls Nenner gleich 0 → verhindert inkonsistente Objekte **1**

```
// Client-Code (ausserhalb Rational)
Rational rat = new Rational(1, 3); 1

rat.den = 0; 2
```

Compilerfehler, da privates Attribut für den Klienten nicht sichtbar **2**

Besser (lernen wir später kennen): `if (d != 0) throw IllegalArgumentException("…");` statt `assert d != 0;`

Beispiel: Encapsulation für `Rational`

3. Schritt: Lesezugriff durch Accessor/Getter-Methoden

(Erinnerung: Methoden, die den Objektzustand nicht verändern)

```
public class Rational {
  …

  public int getNominator() {
    return nom;
  }

  public int getDenominator() {
    return den;
  }
}
```

```
Rational rat = new Rational(1, 3);
…println(rat.getNominator());   // 1
…println(rat.getDenominator()); // 3
```

Hier (aber nicht immer) sinnvoll: Ein Getter pro Attribut

- `getNominator()`, `getDenominator()`

Beispiel: Encapsulation für `Rational`

4. Schritt: Schreibzugriff durch Mutator/Setter-Methoden

(Erinnerung: Methoden, die den Objektzustand verändern)

```
public class Rational {
  …

  public void setNominator(int n) {
    nom = n;
  }

  public void setDenominator(int d) {
    assert d != 0;
    den = d;
  }
}
```

```
Rational rat = new Rational(1, 3);
rat.setNominator(2));    // OK
rat.setDenominator(0));  // Laufzeitfehler
```

- Hier (aber nicht immer) sinnvoll: Ein Setter pro Attribut
  - `setNominator()`, `setDenominator()`
- `setDenominator()` erzwingt Objektinvariante (Nenner ≠ 0)

Zugriff private Attribute eigener Objekte

```
public class Rational {
  private int num;
  private int den; // INV: != 0

  public Rational(int n, int d) {
    assert d != 0;
    num = n;
    den = d;
  }

  public Rational(Rational other) {
    num = other.num;
    den = other.den;
  }

  public String toString() {
    return num + "/" + d;
  }
}
```

- Private Attribute einer Klasse sind für *alle* Objekte dieser Klasse sichtbar
  - Können gelesen *und* geschrieben werden
- Alle Zugriffe im Code rechts daher OK
  - Konstruktor `Rational(int, int)`:
    - `this`-Objekt, schreibend
  - Konstruktor `Rational(Rat other)`:
    - `this`-Objekt, schreibend
    - `other`-Objekt, lesend
  - Methode `toString()`:
    - `this`-Objekt, lesend

Mehr zu Objektinvarianten

- Eine **Objektinvariante** («**object invariant**») ist eine Bedingung, die alle Instanzen einer Klasse erfüllen müssen
  - z.B. gerade gesehen: $\forall\, r \in \texttt{Rational} \;.\; r.\texttt{denominator} \neq 0$
- Diese Bedingung muss vor und nach der Ausführung jeder Methode gelten
  - Konstruktoren müssen sie etablieren ➜ Nachbedingung
  - Jede Methode darf sie annehmen und muss sie erhalten ➜ Vor- *und* Nachbedingung
- Sollte immer in der Klassendefinition aufgeführt werden
  - Wichtige Dokumentation, auch wenn nicht für Beweise verwendet
  - Essenziell für die Weiterentwicklung einer Klasse – z.B. wenn andere Entwickler/innen später neue Methoden hinzufügen

Objektinvarianten: Öffentlich vs. privat

Objektinvarianten können in zwei Kategorien eingeteilt werden:

1. Eine *öffentliche* Objektinvariante
   - Ist (auch) für Klienten relevant: z.B. weil Klienten bei Settern nicht jeden Wert übergeben dürfen – wie bei `Rationals setDenominator()`
   - Muss daher anhand öffentlicher Attribute/Methoden ausdrückbar sein: z.B. $\forall\, r \in$ `Rational` $\,.\, r.$`getDenominator()` $\neq 0$
2. Eine *private* (rein interne) Objektinvariante
   - Ist für Klienten irrelevant/Existenz kann von Klienten nicht beobachtet werden
   - Aber für die Datenstruktur intern wichtig: z.B. für einen (binären) Suchbaum
     - Keine Zyklen; jeder Baumknoten hat zwei Kindsknoten
     - Werte im linken Teilbaum kleiner als im rechten Teilbaum

Code-Weiterentwicklung: Änderungen an der internen Repräsentation

- Kontext: Punkte im 2D-Raum repräsentieren
- Erste Version der entsprechenden Klasse, sowie möglicher Client-Code

```
public class Point {
  double x;
  double y;
}
```

```
Point point = …;
if (point.y == 0) {
  …
}
```

- Später: Andere interne Repräsentation nötig oder gewünscht, z.B.

```
public class Point {
  double radius; // INV: ≥ 0
  double angle;  // INV: [-π, π]
}
```

```
public class Point {
  double[] pair; // INV: length == 2
}
```

Interne Änderungen und abhängige Klienten

Problem:

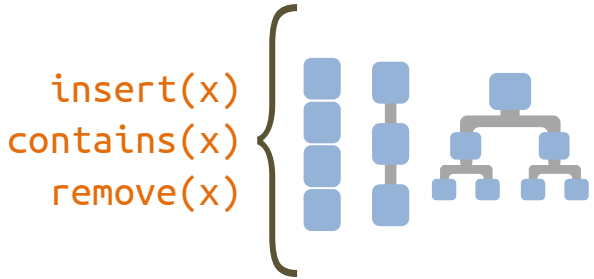
```
public class Point {
  double x;
  double y;
}
```

```
// Klient
if (point.y == 0) {
  …
}
```

- Klienten greifen direkt auf die interne Repräsentation zu
  → sie sind *abhängig* von der internen Repräsentation
- Änderungen daran (Namen, Typen) erzwingen Änderungen in möglicherweise sehr vielen Klienten, z.B. falls eine populäre Bibliothek
- Erschwert Weiterentwicklung (der Bibliothek) und führt zu fragilen Klienten

Encapsulation verhindert Abhängigkeiten

- **Lösung:** Encapsulation und Abstraktion via Schnittstellen
- **Beispiel:** Verwaltung einer Menge von Daten → Collection
  - Viele Implementationen möglich,
    z.B. Tabelle/Array, verkettete Liste, Baumstruktur
    → *interne Repräsentation*
  - Collection bietet Methoden an, die Operationen auf der internen Repräsentation durchführen:
    z.B. Suchen, Einfügen, Löschen
    → *Schnittstelle* («*interface*»)



  - Diese Methoden bieten immer dieselbe Funktionalität an, unabhängig von der internen Repräsentation (die daher jederzeit geändert werden könnte)
    → *Abstraktion*
- Encapsulation (und stabile Schnittstellen) erleichtern Code-Änderungen

Klassendesign: Adaptierbare `Point`-Klasse

```
public class Point {
  private double x;
  private double y;

  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public double getX() {
    return x;
  }

  … // z.B. weitere Getter & Setter;
  … // distanceFrom(Point other); …
}
```

- Klassendesign **mit Encapsulation**
  - Attribute (interner Zustand) sind privat
  - Zugriff darauf nur durch Getter und Setter (Schnittstelle)
  - Klasse vorbereitet auf mögliche Änderungen
- Klienten kennen interne Repräsentation nicht (und müssen es auch nicht)

```
// Klient
if (point.getX() > 0) {
  …
}
```

Andere int. Repr., gleiche Nutzung für Klienten

v2

```
public class Point {
  private double radius; // INV: ≥ 0
  private double degree; // INV: [-π, π]

  public Point(double x, double y) {
    radius = Math.hypot(x, y);
    degree = Math.atan2(y, x);
  }

  public double getX() {
    return radius * Math.cos(degree);
  }

  …
}
```

v3

```
public class Point {
  private double[] pair; // INV: length == 2

  public Point(double x, double y) {
    pair = new double[2];
    pair[0] = x;
    pair[1] = y;
 }

  public double getX() {
    return pair[0];
  }

  …
}
```

Encapsulation: Schlussbetrachtung

- Ermöglicht das Durchsetzen von Objektinvarianten, z.B. `rational.denominator != 0`
- Reduziert die Abhängigkeit von Klienten und vereinfacht so zukünftige Änderungen/Weiterentwicklungen des Codes
- **Encapsulation ist essenziell für grosse, wart- und adaptierbare Software!**

- Weitere Möglichkeiten
  - Rechteverwaltung: Zugriffsmethoden prüfen, ob aktueller Benutzer Daten lesen/ändern darf
  - Logging: Lese-/Schreibzugriffe werden in einer Logdatei nachgehalten

Kurz angesprochen: Sichtbarkeit von Methoden

- Zugriffsmodifikatoren (`public`, `private`, …) gibt es für
  - Attribute — gerade gesehen
  - Methoden — jetzt kurz
  - Klassen — später

- Öffentliche Methoden (`public`) sind für alle Klienten nutzbar
  → definieren die Schnittstelle (Funktionalität) von Objekten
- Private Methoden (`private`) sind nur innerhalb des Objekts verfügbar
  → sinnvoll für rein interne Hilfsfunktionalität

Beispiel: Vergleichbare rationale Zahlen

```
public class Rational {
  private int num;
  private int den; // INV: != 0

  public Rational(int num, int den) {
    assert den != 0;

    this.num = num;
    this.den = den;
  }

  public boolean isEqualTo(Rational other) {
    return num == other.num && den == other.den;
  }
}
```

```
r1 = new Rational(1, 3);
r2 = new Rational(2, 6);
if (r1.isEqualTo(r2)) {
  …
}
```

Problem?

Nicht gezeigt: `isEqualTo()` sollte prüfen, ob `other == null`

Gekürzte rationale Zahlen

- Nutzen des grössten gemeinsamen Teilers (gcd) zum Kürzen
- Interne Hilfsfunktionalität ➜ private Methoden

```
// In class Rational

private int gcd(int x, int y) {
  return x == 0 ? y : gcd(y % x, x);
}

private void reduce() {
  int d = gcd(num, den);
  num = num / d;
  den = den / d;
}
```

```
// In class Rational

public Rational(int num, int den) {
  assert den != 0;

  this.num = num;
  this.den = den;

  reduce();
}
```

Kürzt, z.B. 2/6 zu 1/3
und -32/532 zu -8/133

- Nach welchen weiteren Operationen sollte noch gekürzt werden?

Idee weiteres Beispiel

- **Ziel:** Klasse `Zeitspanne` zur Verwaltung von Stunden und Minuten
- **Gewünschte Nutzung (Schnittstellen)**, z.B.
  - `t = new Zeitspanne(3, 45)` → 3 Stunden und 45 Minuten
  - `println(t)` → `"3:45"`
  - `t.add(new Zeitspanne(0, 20)); println(t)` → `"4:05"`
- **Fragen**, z.B.
  - Attribute – `int stunden; int minuten` vs. `int gesamtMinuten`?
  - Objektinvariante – `0 ≤ minuten < 60` vs. `0 ≤ gesamtMinuten`?
  - Objektinvariante wie durchsetzen
    - `new Zeitspanne(0, 90)` – Umrechnen? Fehlermeldung?
    - `new Zeitspanne(0, -1)` – Auf 0 «aufrunden»? Fehlermeldung?
  - Änderung interne Repräsentation – `stunden`, `minuten` ⟷ `gesamtMinuten`?

7. Klassen und Objekte

Motivation für Methoden & Klassen

- Methoden erlauben Code-Wiederverwendung
  - Motivation für Methoden seit Woche 2

- Zuletzt besprochen: Klassen (z.B. `Person`, `Point`; `Random`)
  - Dienen als Vorlage/Bauanleitung für Objekte
  - Bündeln Zustand (Attribute) und Operationen darauf → Methoden
  - Diese Methoden operieren im Kontext eines Objekts

- Überlegung: Könnten auch Methoden (→ Code-Wiederverwendung) sinnvoll sein, die nicht im Kontext eines Objekts arbeiten?

Methoden ohne Objektbezug?

- Überlegung: Methoden ohne Objektbezug sinnvoll?
- Ja, zum Beispiel mathematische Operationen

```
// Berechnet grössten gemeinsamen Teiler
public int gcd(int x, int y) {
  return x == 0 ? y : gcd(y % x, x);
}
```

```
// Berechnet Quadratwurzel
public double sqrt(double x) {
  …
}
```

- Naheliegende Fragen
  1. Wie und wo *implementieren* wir solche «objektlosen» Methoden?
  2. Wie können sie *aufgerufen* werden?

Statische Methoden

- **Statische Methoden** (**Klassenmethoden**) → *ohne* Objektbezug ausgeführt
- Klasse dient dann als *Namensraum*, der Methoden sinnvoll gruppiert

1. Implementation

```
public class MyMathLib {
  public static int gcd(int x, int y) {
    return x == 0 ? y : gcd(y % x, x);
  }

  public static double sqrt(double x) { … }
}
```

Innerhalb einer Klasse — mit Modifier **static**

2. Externer Aufruf

```
// Client-Code
// (ausserhalb MyMathLib)
int d = MyMathLib.gcd(14, 21);
double r = MyMathLib.sqrt(d);
```

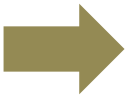
Punktnotation — mit **Klasse** statt Aufruferobjekt

Statisch = ohne Objektbezug = kein `this`

- Statische Methoden → Ausführung ohne Objektbezug

```
public class Villekulla {
  int x;

  public static void print() {
    System.out.println(this.x);
  }
}
```



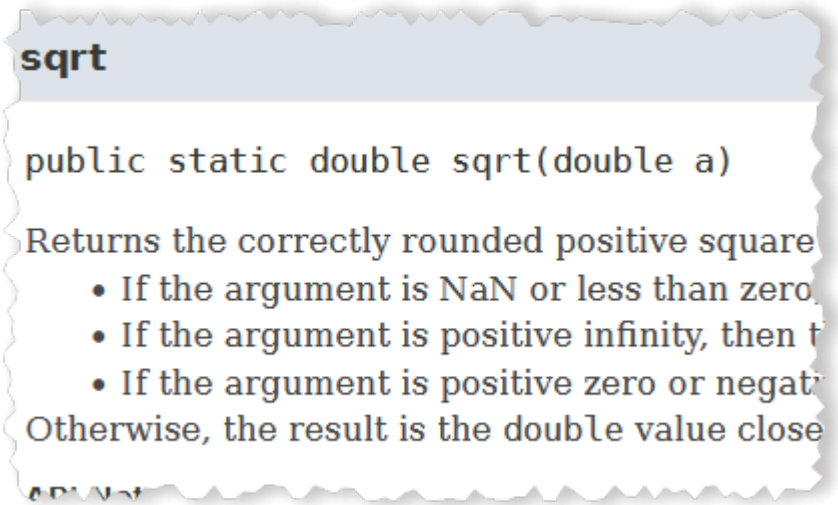
```
Unresolved compilation problem:
   Cannot use this in a static
   context
```

- Obiger Code kompiliert daher nicht
  - Statische Methode → kein Aufruferobjekt (kein Objektbezug)
    → keine `this`-Referenz verfügbar

Leicht andere Fehlermeldung falls keine explizite `this`-Nutzung (nur `println(x)`), aber selbes Grundproblem

Beispiele statischer Methoden

Java-Klasse `Math`: Methoden `Math.sqrt(x)`, `Math.pow(x, y)`



sqrt

```
public static double sqrt(double a)
```

Returns the correctly rounded positive square

- If the argument is NaN or less than zero
- If the argument is positive infinity, then t
- If the argument is positive zero or negat

Otherwise, the result is the double value close

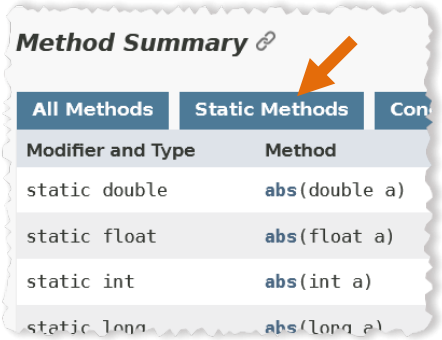
```
jshell> Math.sqrt(9)
$1 ⟹ 3.0

jshell> Math.pow(2, 8)
$2 ⟹ 256.0
```

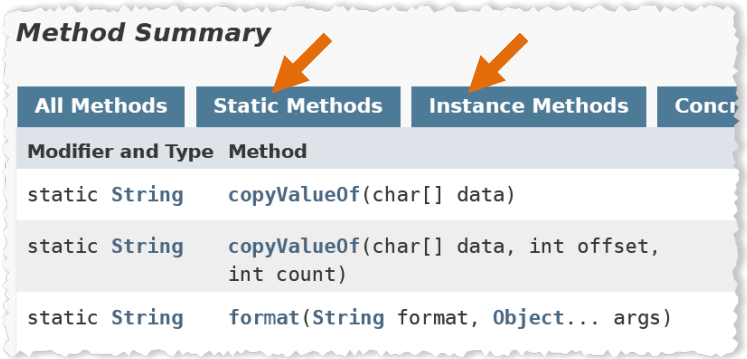
Dokumentation `Math.sqrt()`, `Math.pow()`

Typische Nutzungen statischer Methoden

- **Klasse als reiner Namensraum**
  - Privater Konstruktor (Instanzen verhindern)
  - Nur statische Methoden
- Beispiel: Java-Bibliothek `Math`



- **Mischform**
  - Öffentlicher Konstruktor (Instanzen erlauben)
  - Statische und Objektmethoden
- Beispiel: Java-Bibliothek `String`



Math: Quellcode, Dokumentation | String: Quellcode, Dokumentation

Statische Attribute

- **Statische Attribute** (**Klassenattribute**)
  - Ebenfalls kein Objektbezug: Attribute existieren genau einmal (pro Klasse), nicht mehrfach (pro Instanz)
  - Sind (auch) aus statischen Methoden erreichbar

1. Implementation

```
public class MyMathLib {
  public static double PI = 3.1415;

  public static double degToRad(double degree) {
    return deg * (PI / 180);
  }
}
```

2. Externer Zugriff

```
// Client-Code
// (ausserhalb MyMathLib)
…println(2 * MyMathLib.PI);
```

Statische Attribute: Zugriffskontrolle

- Statische Attribute existieren *global* (an der Klasse)
  → Zugriffskontrolle um so wichtiger

```
// In class MyMathLib
public static double PI = 3.1415;
```

```
// Irgendein Klient
MyMathLib.PI = 0;
```

Typischerweise nicht erwünscht

- **Faustregel 1:** Wenn `public static`, dann auch `final`
  - `final` = kann nach Initialisierung nicht mehr verändert werden = nur Lesezugriff
  - Typische Nutzung: Globale Konstanten

```
// In class MyMathLib
public static final double PI = 3.1415;
```

```
// Irgendein Klient
…println(MyMathLib.PI);
MyMathLib.PI = 0;
```

Erlaubt

Compilerfehler

Statische Attribute: Zugriffskontrolle

- **Faustregel 2:** Andernfalls `private static`
  - Und falls nötig: Kontrollierter externer Zugriff via statischer Getter-/Setter-Methoden
- Typische Anwendung: Statisches Attribut wird gebraucht, um eine Eigenschaft über *alle* Objekte einer Klasse hinweg zu garantieren
- Beispiel: Personen-Objekte müssen eindeutige IDs besitzen

Anwendungsbeispiel

Anforderung: Personen-Objekte müssen *eindeutige* IDs besitzen

static counter: existiert genau 1x

id: pro Objekt

```
public class Person {
  private static int counter = 0;
  private int id;
  private String name;

  public Person(String name) {
    id = counter;
    counter += 1;
    this.name = name;
  }

  public String toString() {
    return name + " (" + id + ")";
  }
}
```

```
Person p1 = new Person("Omer");
System.out.println(p1);
  // Omer (0)

Person p2 = new Person("Omar");
System.out.println(p2);
  // Omar (1)

Person p3 = new Person("Otmar");
System.out.println(p3);
  // Otmar (2)
```

`Static`: Abschluss & Hinweise

- Statische Methoden in der Praxis nicht ungewöhnlich

- Statische Attribute
  - `public`/`private final`: Konstanten – nicht ungewöhnlich
  - `private` (nicht-`final`): Eher selten
  - `public` (nicht-`final`): «Recipe for disaster» 

- Typischer Fehler (Rückmeldung Übungsbetrieb)
  - Statische nicht-final Attribute oft genutzt, um sich ändernde Werte zentral zu hinterlegen und so zwischen Methoden/Objekten auszutauschen
  - Fast nie eine gute Idee! Besser: Austauschen via Parameter/Rückgabewerte oder Objektattribute

7. Klassen und Objekte

Motivation

- **Kontext:** Logistik-Anwendung zur Verwaltung von Bestellungen
- **Ziel:** Bestellstatus repräsentieren
  - *submitted*, *processing*, *packed*, *shipped*, *received*, *paid*, …
- **Erste Idee:** Repräsentation durch Basistypen

```
final int SUBMITTED = 0;
final int PROCESSING = 1;
…
```

```
final String SUBMITTED = "submitted";
…
```

Mögliche Nachteile?

Nachteile Basistypen

1. Zu grosser Wertebereich → ungültige/unsinnige Status
   - `int status = -123456;`
   - `String status = "Suppmidded"`

2. Unsinnige/problematische Operationen
   - `int status = 2 * SUBMITTED + 13;`
   - `String status = (PAID + " in full").toLowerCase();`

Verallgemeinerung

Häufig vorkommende Situation: Benötigen einen Typ mit

- Genau *n* Elementen: z.B. Bestellstatus, Wochentage, Nationalität, Kleidergrösse (XL, …)
- Wenigen, von uns festgelegten Operationen
  - Typischerweise (mindestens)
    - Konvertieren für Ein- und Ausgabe – z.B. `"paid"` ⟷ `PAID`
    - Auf Gleichheit prüfen – z.B. `if (order.status == PAID) { … }`

Enums

Enums (**Aufzählungen**) sind spezielle Klassen, die genau für solche Situationen gedacht sind.

```
public enum Status {
  SUBMITTED,
  PROCESSING,
  …
  PAID
}
```

```
public class Order {
  private Status status = Status.SUBMITTED;
}
```

```
Order order = …;
if (order.status == Status.PROCESSING) { … }
```

- Operator == verhält sich wie erwünscht
- Keine unsinnigen Operationen mehr (z.B. `status = SUBMITTED / 2`)

Nützliche Funktionalität frei Haus

```
enum Status {
  SUBMITTED,
  SHIPPED,
  PAID
}

class Order {
  Status status;
}
```

```
if (order.status == Status.PROCESSING) {…}
```

Vergleichsoperator

```
Status.PAID.name() // → String "PAID"
```

Konvertierung Objekt → String

```
Status.valueOf("PAID") // → Status.PAID
```

Konvertierung String → Objekt

```
if (  order.status.ordinal()
    < Status.PAID.ordinal()) {
  …
}
```

Ordnungsrelation (via ganze Zahlen)

- `SUBMITTED.ordinal() == 0`
- `PAID.ordinal() == 2`

```
String message = switch (order.status) {
  case SUBMITTED, SHIPPED -> "Payment open";
  case PAID -> "Payment received";
};
```

- Integration in switch-Ausdrücken/Anweisungen
  (haben wir nicht vorgestellt, aber relativ selbsterklärend)
- Compilerfehler, falls ein Fall vergessen wurde

Abschluss

- Enum, falls ein Typ mit *n* Elementen benötigt wird
  - Kommt in der Praxis häufig vor
- Nützliche Funktionalität automatisch vorhanden
- Enum-Klassen können auch um eigene Funktionalität erweitert werden: Attribute, Methoden, Konstruktoren
- Für EProg
  - Relevant: Einfache Nutzung von Enums (wie auf den vorherigen Folien)
  - Nicht mehr relevant: Erweiterung von Enum-Klassen um eigene Funktionalität

Zusammenfassung (Java-)Klassen

Eine Klasse kann in einem Softwaresystem verschiedene Rollen einnehmen:

1. Ein ausführbares *Program* enthalten
   - Klasse mit `public static void main(…)`
2. Einen *Typ* definieren und als *Bauanleitung* für Elemente (Objekte/Instanzen) dieses Typs dienen
   - Konstruktoren, Attribute und Methoden (i.d.R. nicht-statisch)
   - Interner Zustand gekapselt («Encapsulation»): private Attribute und evtl. öffentliche Accessor/Mutator-Methoden
   - Beispiele: `Person`, `Point`; `Random`, `String`
3. Einen *Namensraum* (für statische Methoden & Attribute) definieren
   - Beispiele: `MyMathLib`; `Math`, `Arrays`