



Fachhochschule Nordwestschweiz  
Hochschule für Technik

# Einführung in die Programmierung Objekt-orientierte Programmierung 1

# Klassen

Michael Faes

# Arten von Dekomposition

## *Prozedurale Dekomposition*

- Aufteilung einer Aufgabe in Teil-Aufgaben
- Verb- / Handlungs-orientiert

## 1. Hälfte des Semesters

## *Objekt-orientierte Dekomposition*

- Aufteilung eines Systems in kleinere, interagierende Teile (Objekte!)
- Nomen- / Objekt-orientiert

## 2. Hälfte des Semesters

*Programmiererin muss beide Methoden zur Problemlösung kennen!*

# Objekt-orientierte Programmierung

# Handlungs- vs. Objekt-orientiert

Alte Betriebssysteme (DOS, Unix):

```
Command Prompt

C:\>mkdir Test

C:\>cd Test

C:\Test>cd ..

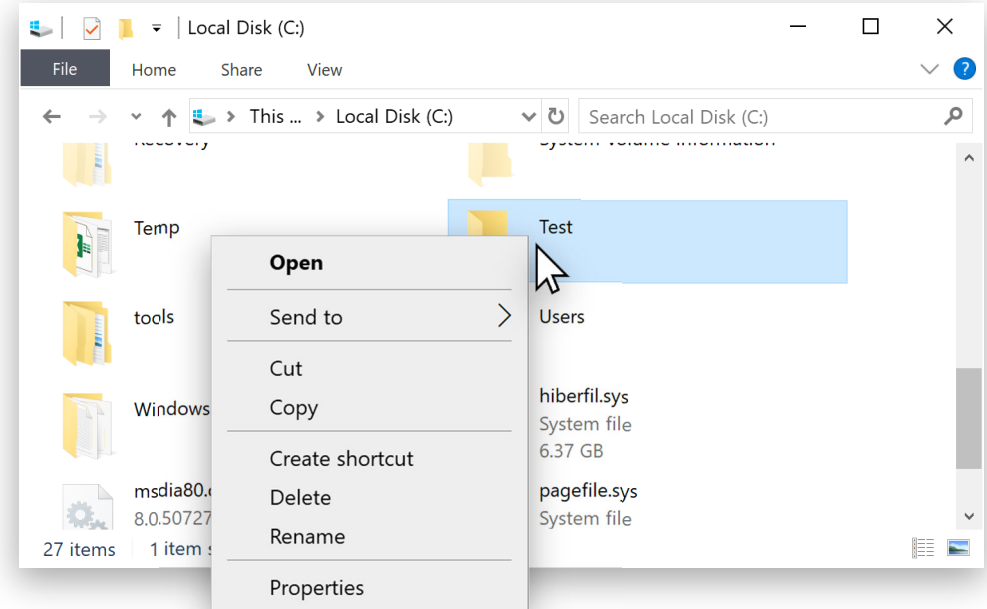
C:\>del Test
C:\Test\*, Are you sure (Y/N)? y

C:\>
```

Interaktionsmuster: Verb Nomen

In Java: `methode(argument)`

Moderne Betriebssysteme:



Interaktionsmuster: Nomen Verb

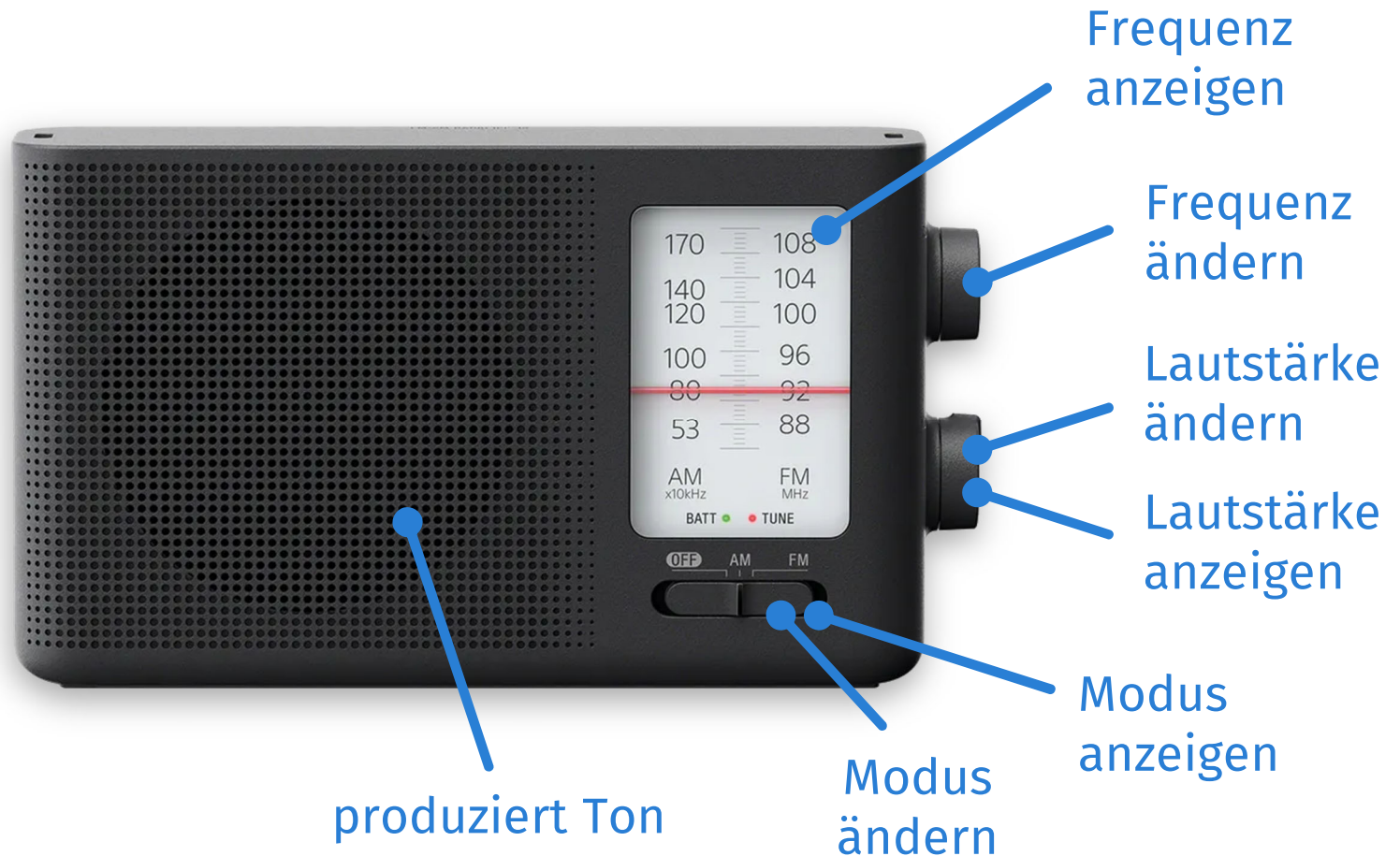
In Java: `objekt.methode()`

«Nachricht an Objekt senden»

# Objekte

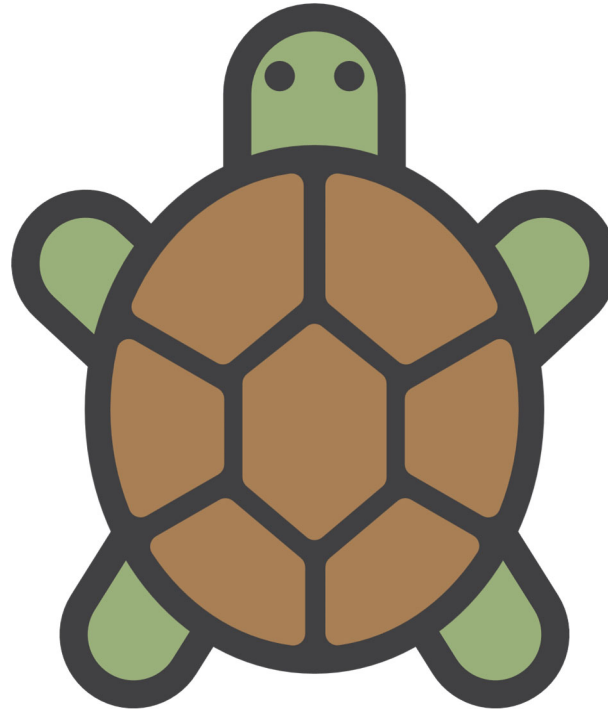
Objekt = Daten + Verhalten  
(Eigenschaften/Zustand)

Modus: AM/FM/aus  
Frequenz  
Lautstärke  
Abmessungen  
Gewicht  
Farbe  
...



# Objekte in Programmen

x-Position  
y-Position  
Ausrichtung  
Stift oben/unten  
Stiftfarbe  
Stiftdicke  
Geschwindigkeit



vorwärts gehen  
rückwärts gehen  
nach links drehen  
nach rechts drehen  
Stift heben  
Stift senken  
Stiftfarbe ändern  
Stiftdicke ändern  
Geschw. ändern

# Übung 1: Modellierung als Objekte

Überlegen Sie sich zu zweit, welche Daten zu folgenden Objekten gehören und welches Verhalten man dazu programmieren könnte.

- a) Ein Film auf Netflix
- b) Eine Spielkarte in einem Online-Pokerspiel
- c) Ein Kommentar auf [www.20min.ch](http://www.20min.ch)
- d) Ein Modulanlass im Einschreibungssystem der HT
- e) Ein eigenes Beispiel für ein Objekt in einem Softwaresystem

# Objekte und Client-Code

Objekte sind (normalerweise) selber keine vollständigen Programme, sondern *Komponenten in einem Programm*.

## Objekte:

- Ein Scanner
- Ein Random
- Das System.in
- Das System.out

```
public class Wuerfeln {  
    public static void main(String[] args) {  
        Scanner scanner = new Scanner(System.in);  
        System.out.print("Anzahl Seiten? ");  
        int seiten = scanner.nextInt();  
        Random random = new Random();  
        int wurf = random.nextInt(seiten) + 1;  
        System.out.println(wurf);  
    }  
}
```

*Client-Code* der Objekte



# Klassen und Objekte


Für komplexere Programme brauchen wir eigene Typen von Objekten!

## Beispiele:

- Streaming-App: Filme, Serien, Staffeln, Folgen, Users, ...
- 3D-Game: Koordinaten, Polygone, Texturen, Gegner, Items, ...
- Einschreibesystem: Module, Anlässe, Studierende, Dozierende, ...

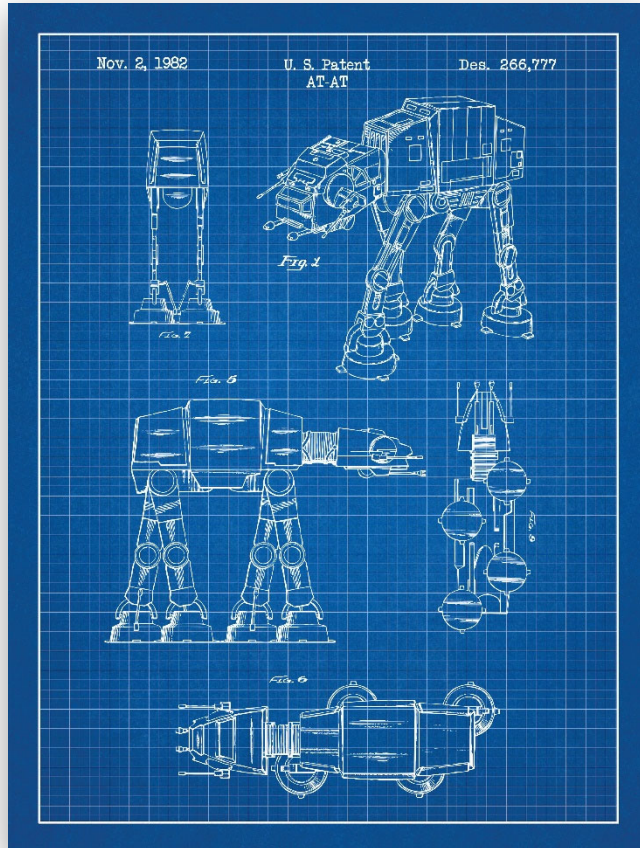
Eigene Typen von Objekten werden durch *Klassen* definiert! Klassen sind *Baupläne* für Objekte und definieren:

1. Die **Daten**, welche zu jedem Objekt gehören (Eigenschaften/Zustand)
2. Welches **Verhalten** die Objekte haben
3. Wie ein Objekt **erstellt** wird

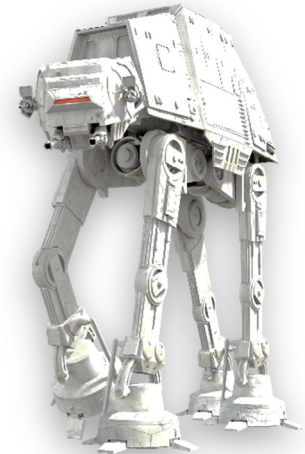
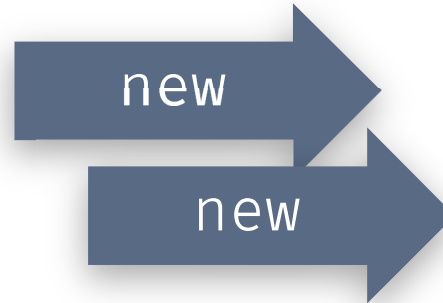


Welche «Nachrichten»  
das Objekt versteht

# Klassen und Objekte

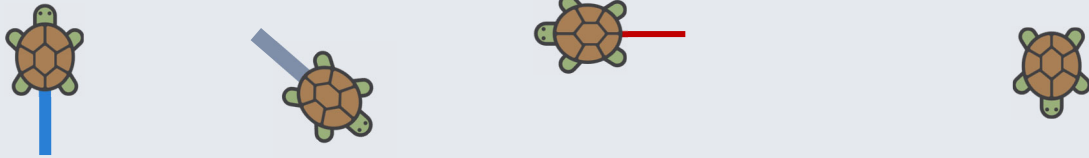


**Klasse:** Bauplan



**Objekte:** Exemplare  
(auch «Instanzen»)

# Klassen und Objekte: Beispiele

Klasse	mögliche Objekte
Film	<i>Inception, Star Wars, Back to the Future, ...</i>
Modul	eipr, oopl1, oopl2, sweGL, webeC, ...
Country	Schweiz, Deutschland, Frankreich, ...
Point2D	(0, 0), (1, 1), (-3.5, 1.7), (100, 100), ...
Turtle	
Random	(nicht darstellbar...)

# Eine Beispielanwendung

```
EN für oopI1? 4.2
MSP-Note für oopI1? 4.8
EN für sweGL? 5.1
MSP-Note für sweGL? 4.6
EN für dtpC? 3.7
MSP-Note für dtpC? 4.0
EN für mgli? 2.9
MSP-Note für mgli? 4.1
```

```
oopI1: 4.5
sweGL: 4.9
dtpC: 3.9
mgli: 3.5
```

```
Bestandene Module:      3
Gesammelte Credits:     9
Schnitt (bestandene): 4.433333...
```

Daten für jedes Modul:

- Name
- Anzahl Credits
- Erfahrungsnote
- MSP-Note

# Programmieren ohne Objekte

Arbeiten mit diesen komplexen Daten mit bisherigen Mitteln mühsam:

```
String name = "oopI1";  
int credits = 3;  
double erfNote = 4.2;  
double mspNote = 4.8;  
  
String name2 = "sweGL";  
int credits2 = 3;  
double erfNote2 = 4.6;  
double mspNote2 = 3.7;  
  
...
```



```
String[] namen = new String[...];  
int[] credits = new int[...];  
double[] erfNoten = new double[...];  
double[] mspNoten = new double[...];
```

```
namen[0] = "oopI1";  
credits[0] = 3;  
erfNoten[0] = 4.2;  
mspNoten[0] = 4.8;
```

```
namen[1] = "sweGL";  
credits[1] = 3;  
erfNoten[1] = 4.6;  
mspNoten[1] = 3.7;
```

...

vier Arrays!..

so viele Indizes...

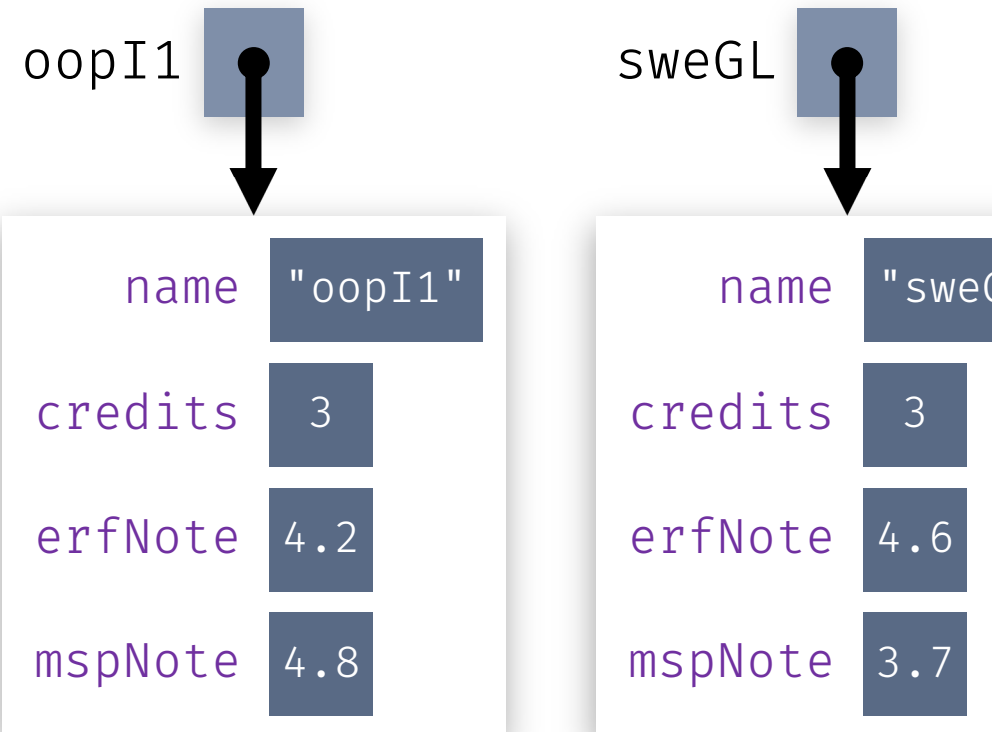


immer noch

# Programmieren *mit* Objekten

Wenn wir doch nur eine Modul-Klasse hätten...

```
Modul oopI1 = new Modul("oopI1", 3);  
oopI1.erfNote = 4.2;  
oopI1.mspNote = 4.8;  
  
Modul sweGL = new Modul("sweGL", 3);  
sweGL.erfNote = 4.6;  
sweGL.mspNote = 3.7;  
  
...  
  
Modul[] module = {oopI1, sweGL, ...};
```



Programm ist auch verständlicher, da die «Dinge», um die es geht, direkt im Code sichtbar sind.

# Attribute, Methoden, Konstruktoren, ...

Bauen schrittweise unsere eigene Modul-Klasse!

Hauptkomponenten einer Klasse:

- **Attribute:** speichern *Eigenschaften/Zustand* des Objekt  
(auch «fields»/«Felder», «Instanzvariablen», ...)
- **Methoden:** definieren *Verhalten* der Objekte
- **Konstruktoren:** Codestücke, die ein Objekt initialisieren, wenn es mittels *new* frisch *erstellt* wird
- **Datenkapselung:** schützt den Zustand des Objekts vor externem Zugriff (engl. «encapsulation»)

nächste Woche

Jede Version von Module wird neue Komponente hinzufügen.

*Nur Endversion nächste Woche ist richtiger Objekt-orientierter Stil!*

# Attribute und Methoden



# Attribute

**Attribut (Field):** Eine Variable innerhalb eines Objekts.

Jedes Modul-Objekt enthält diese vier Variablen!

```
public class Modul {  
    String name;  
    int credits;  
    double erfNote;  
    double mspNote;  
}
```

In eigener Datei  
Modul.java

Null-Initialisierung  
aller Attribute

Klassen-Code

```
Modul oopI1 = new Modul();  
oopI1.name = "oopI1";  
oopI1.credits = 3;  
Modul sweGL = new Modul();  
sweGL.name = "sweGL";  
System.out.println(oopI1.name + ": " + oopI1.credits);  
System.out.println(sweGL.name + ": " + sweGL.credits);
```

Dot-  
Notation

Client-Code

oopI1: 3  
sweGL: 0



# Statische vs. Instanz-Methoden

Wie Verhalten hinzufügen? Bisher mit *statischen* Methoden:

```
public static double endNote(Modul modul) {  
    return (modul.erfNote + modul.mspNote) / 2;  
}
```

Verwendung:

```
double note = endNote(oopI1);
```



Stattdessen **Instanz-Methode**: Methode *innerhalb eines Objekts*, welche direkt auf diesem Objekt arbeitet.

```
public class Modul {  
    ...  
    public double endNote() {  
        ...  
    }  
}
```

kein static!

kein Parameter!

Verwendung:

```
double note = oopI1.endNote();
```



# Instanz-Methoden

Scope von Attributen

```
public class Modul {  
    String name;  
    int credits;  
    double erfNote;  
    double mspNote;  
  
    public double endNote() {  
        return (erfNote + mspNote) / 2;  
    }  
}
```

name	"oopI1"
credits	3
erfNote	4.2
mspNote	4.8

```
p... double endNote() {  
    ...  
}
```

Teil vom Objekt!

Instanz-Methoden sind Teil von jedem Objekt und können direkt auf die Attribute dieses Objekts zugreifen.

Aus Sicht des Objekts: «Der Client fragt **mich** nach **meiner** Endnote.»

```
Modul oopI1 = new Modul();
```

```
...
```

```
oopI1.erfNote = 4.2;
```

```
oopI1.mspNote = 4.8;
```

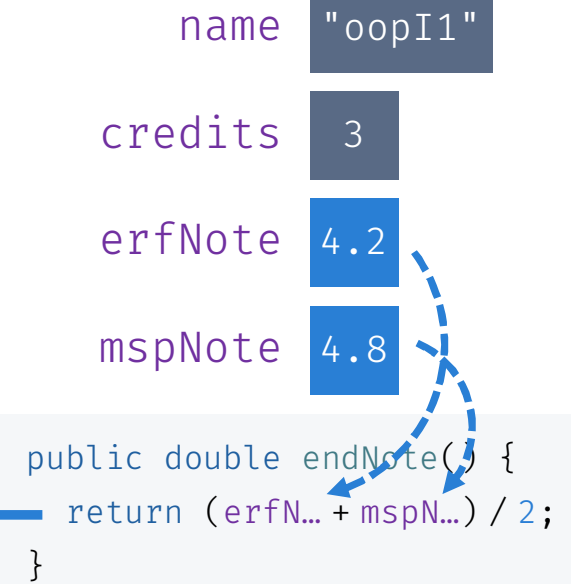
```
Modul sweGL = new Modul();
```

```
...
```

```
sweGL.erfNote = 4.6;
```

```
sweGL.mspNote = 3.7;
```

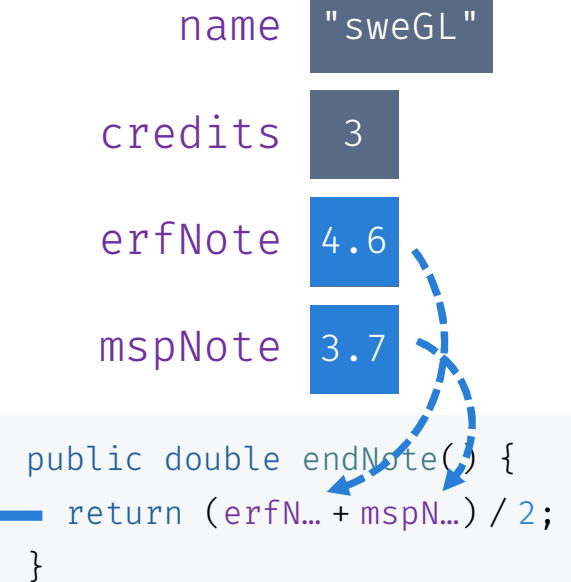
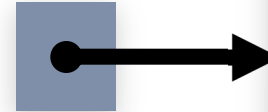
oopI1



4.4

```
oopI1Note = oopI1.endNote();
```

sweGL



4.15

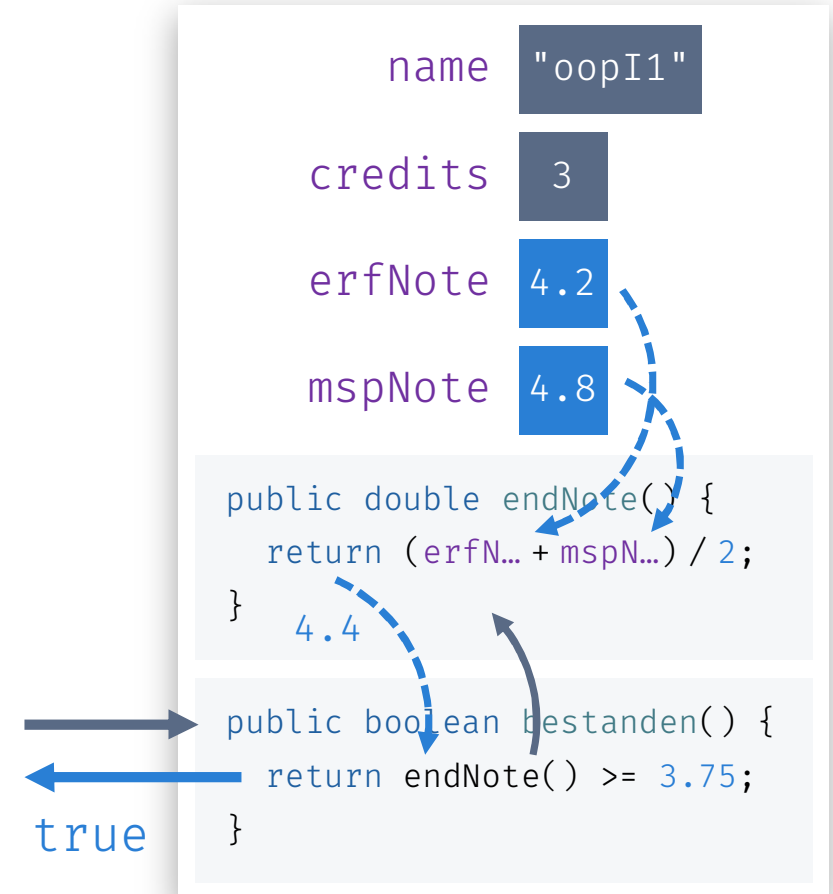
```
sweGLNote = sweGL.endNote();
```

«Versteckter» Parameter: *Ziel-Objekt*

# Methoden rufen Methoden auf

Weitere praktische Methode: `bestanden()`

```
public class Modul {  
    String name;  
    int credits;  
    double erfNote;  
    double mspNote;  
  
    public double endNote() {  
        return (erfNote + mspNote) / 2;  
    }  
  
    public boolean bestanden() {  
        return endNote() >= 3.75;  
    }  
}
```



# Übung 2: Instanz-Methode schreiben

In der Vorlage von dieser Woche finden Sie den aktuellen Stand der Modul-Klasse und ein paar Unit-Tests. Ändern Sie die Implementation der Methode `endNote`, so dass sie folgende Anforderungen erfüllt:

1. Die Endnote soll auf *eine Nachkommastelle* gerundet werden.  
**Tipp:** `Math.round(x)` rundet eine Zahl `x` auf *eine ganze Zahl*.
2. Falls die Erfahrungsnote oder die MSP-Note noch 0 ist, wird als Endnote auch 0 zurückgegeben werden.

Verwenden Sie die Unit-Tests, um Ihre Lösung zu überprüfen.

# Die `toString`-Methode

Unsere `Modul`-Klasse spielt noch nicht schön mit Strings zusammen:

```
int i = 42;  
System.out.println(i);  
  
Modul m = new Modul();  
m.name = "oopI1";  
m.credits = 3;  
System.out.println(m);
```

```
42  
Modul@6a41eaa2
```



```
System.out.println("i ist " + i);  
System.out.println("m ist " + m);
```

```
i ist 42  
m ist Modul@6a41eaa2
```



Wenn man Objekt auf Konsole ausgibt, oder mit Strings zusammenhängt, ruft Java eine spezielle `toString-Methode` auf dem Objekt auf.

- Falls keine definiert, wird eine Default-Version verwendet!

# Eigene toString-Methode

toString-Methode muss *genau* folgende Form haben:

```
public String toString() {  
    // erstelle String-Repräsentation und gib zurück  
}
```

Für die Modul-Klasse zum Beispiel:

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

Jetzt funktioniert + und println:

```
System.out.println(p);  
System.out.println("m ist " + m);
```

toString wird  
versteckt aufgerufen

```
oopI1 (3 ECTS)  
m ist oopI1 (3 ECTS)
```





# Konstrukturen & Overloading

# Konstruktor definieren

Initialisieren von Module-Objekten umständlich:

```
Modul oopI1 = new Modul();  
oopI1.name = "oopI1";  
oopI1.credits = 3;
```



Wollen Initialisieren in *einer* Anweisung:

```
Module oopI1 = new Modul("oopI1", 3);
```

Dafür brauchen wir eigenen *Konstruktor* in Modul-Klasse:

Klassenname

```
public Modul(String initName, int initCredits) {  
    name = initName;  
    credits = initCredits;  
}
```



sieht aus wie Methode,  
aber ohne Rückgabotyp

# this

**Nochmals:** Instanz-Methoden haben «versteckten» *Parameter*, welcher erlaubt, auf Attribute und Methoden von *Ziel-Objekt* zuzugreifen.

Dieser versteckte Parameter ist eine Referenz, die auf «dieses» Objekt zeigt. Kann auch ausdrücklich mit «**this**» verwendet werden:

In Methoden:

```
public double endNote() {  
    return (this.erfNote + this.mspNote) / 2;  
}
```

In Konstruktoren:

```
public Modul(int initName, int initCredits) {  
    this.name = initName;  
    this.credits = initCredits;  
}
```

# Methoden-Aufrufe und `this`

`this` zeigt immer auf das Objekt, in dem sich die Methode befindet, die *gerade ausgeführt* wird:

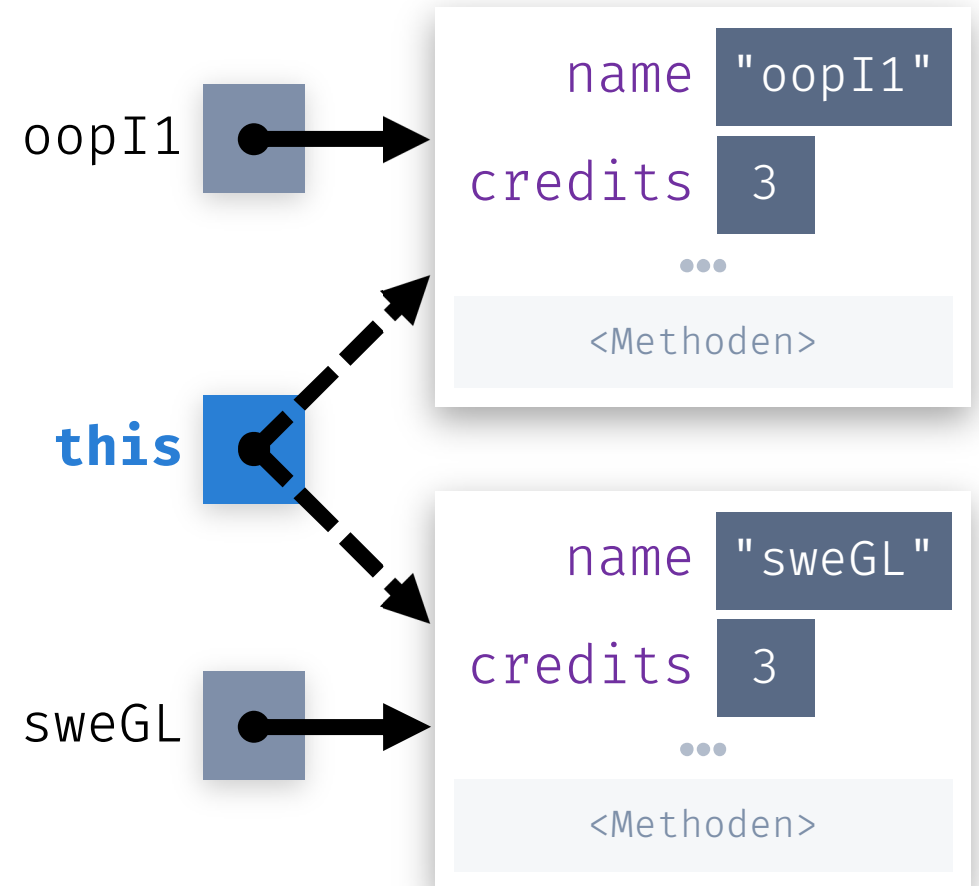
```
Modul oopI1 = new Modul("...", 3);  
Point sweGL = new Point("...", 3);
```

```
note1 = oopI1.endNote();
```

```
note2 = sweGL.endNote();
```

Ein Aufruf `oopI1.endNote()` bedeutet eigentlich:

1. Setze `this` auf `oopI1`
2. Rufe Methode `endNote` auf



# Konstrukturen und `this`

Hauptverwendung von `this`: Vermeiden von *Namenskonflikten* bei Konstrukturen.

Ohne:

```
public Modul(String initName, int initCredits) {  
    name = initName;  
    credits = initCredits;  
}
```



umständlich

Geht nicht:

```
public Modul(String name, int credits) {  
    name = name;  
    credits = credits;  
}
```

Zuweisung zu  
sich selbst...

«Verstecken» Attribute  
mit selbem Namen



falsch!

Mit `this`:

```
public Modul(String name, int credits) {  
    this.name = name;  
    this.credits = credits;  
}
```



# Übung 3: Klassen und Konstruktoren

Finden und erklären Sie die Fehler in folgenden Klassen:

```
public class Land {  
    String name;  
    int einwohner;  
  
    public Land(String name, int einw) {  
        this.name = name;  
        einwohner = einw;  
    }  
}
```

```
public class Point2D {  
    int x;  
    int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```


```
public class Color {  
    int red;  
    int blue;  
    int green;  
  
    public Color(int r, int g, int b) {  
        r = red;  
        b = blue;  
        g = green;  
    }  
}
```

```
public class Line {  
    Point2D start;  
    Point2D end;  
    Color width;  
  
    public void Line() {}  
}
```

# Übung 3: Klassen und Konstruktoren

Finden und erklären Sie die Fehler in folgenden Klassen:

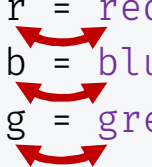
```
public class Land {  
    String name;  
    int einwohner;  
  
    public Land(String name, int einw) {  
        this.name = name;  
        einwohner = einw;  
    }  
}
```



(aber inkonsistent)

```
public class Point2D {  
    int x;  
    int y;  
  
    public Point2D(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

```
public class Color {  
    int red;  
    int blue;  
    int green;  
  
    public Color(int r, int g, int b) {  
        r = red;  
        b = blue;  
        g = green;  
    }  
}
```



Attribute werden nicht gesetzt

```
public class Line {  
    Point2D start;  
    Point2D end;  
    Color width;  
  
    public void Line() {}  
}
```

Methode, nicht Konstruktor

# Default-Konstruktor

Ohne selber geschriebenen Konstruktor fügt Java automatisch einen *Default-Konstruktor* ohne Parameter hinzu.

**Aber:** Wenn wir selber einen schreiben, nicht mehr!

```
Modul m = new Modul();
```



kompiliert nicht mehr

wollen wir  
vielleicht auch...

Müssen selbst noch einen Konstruktor ohne Parameter hinzufügen:

```
public Modul() {  
    name = null;  
    credits = 0;  
}  
public Modul(String name, int credits) {  
    this.name = name;  
    this.credits = credits;  
}
```



Overloading

```
Modul m1 = new Modul();  
Modul m2 = new Modul("oopI1", 3);
```



# Overloading

**Grundregel:** Methoden innerhalb einer Klasse brauchen unterschiedliche Namen, da im selben Scope.



```
public class ScopeProblem {  
    public void foo() { ... }  
    public void foo() { ... }  
}
```

**Overloading (Überladen):** Wenn sich *Anzahl oder Typen* der Parameter unterscheiden, darf trotzdem gleicher Name verwendet werden!



```
public class Overloading {  
    public void foo() { ... }  
    public void foo(int i) { ... }  
    public void foo(double d) { ... }  
    public void foo(int i, int j) { ... }  
}
```

# Overloading-Beispiele

Haben schon einige überladene Methoden von anderen Klassen verwendet:

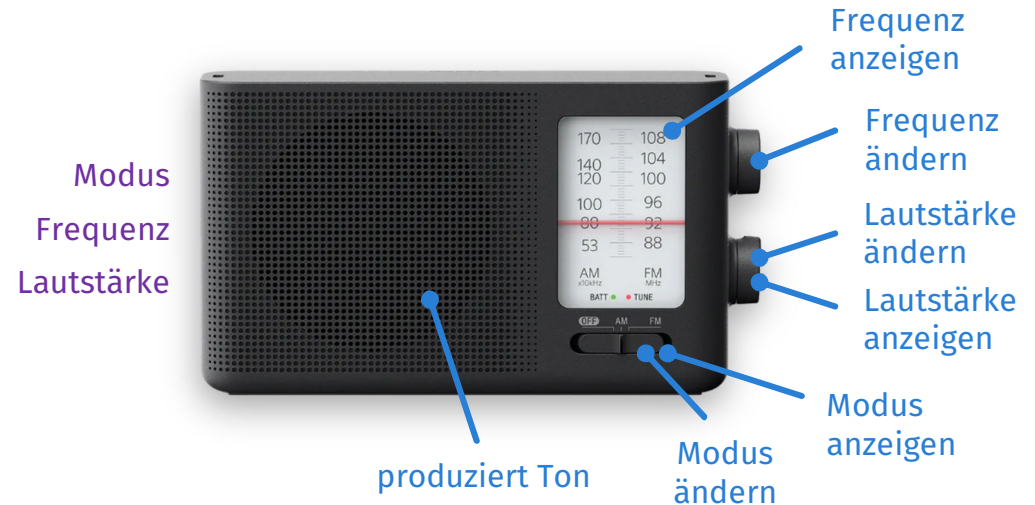
```
x = random.nextInt();           // ganzer int-Bereich
y = random.nextInt(10);         // von 0 bis und mit 9
z = random.nextInt(1, 7);       // von 1 bis und mit 6
```

```
System.out.println();           // leere Zeile
System.out.println(42);         // gib int aus
System.out.println(3.1415);     // gib double aus
System.out.println("OOP1");     // gib String aus
```

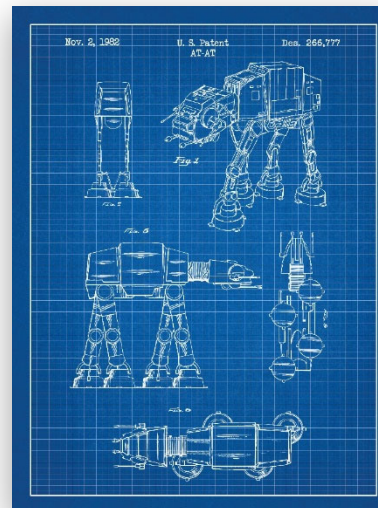
```
int zahl = Math.max(10, 20);    // Max von ints
double d = Math.max(0.3, 1.2);  // Max von doubles
```

# Take-Home Messages

1. Objekte bestehen aus *Daten* + *Verhalten* (+ *Erstellen*)



2. Klassen sind *Bauplan*,  
Objekte sind *Exemplare*



### 3. Komponenten einer Klasse:

*Attribute*

*Konstruktoren*

*Instanz-Methoden*

```
public class Modul {  
    String name;  
    int credits;  
    double erfNote;  
    double mspNote;  
  
    public Modul(String name, int credits) {  
        this.name = name;  
        this.credits = credits;  
    }  
  
    public double endNote() {  
        if (erfNote == 0 || mspNote == 0) {  
            return 0;  
        } else {  
            double schnitt = (erfNote + mspNote) / 2;  
            return Math.round(schnitt * 10.0) / 10.0;  
        }  
    }  
  
    public boolean bestanden() {  
        return endNote() >= 3.75;  
    }  
}
```

# Fragen?

