

# **Objekt-Orientierte Programmiersprachen**

Martin Gasbichler, Holger Gast

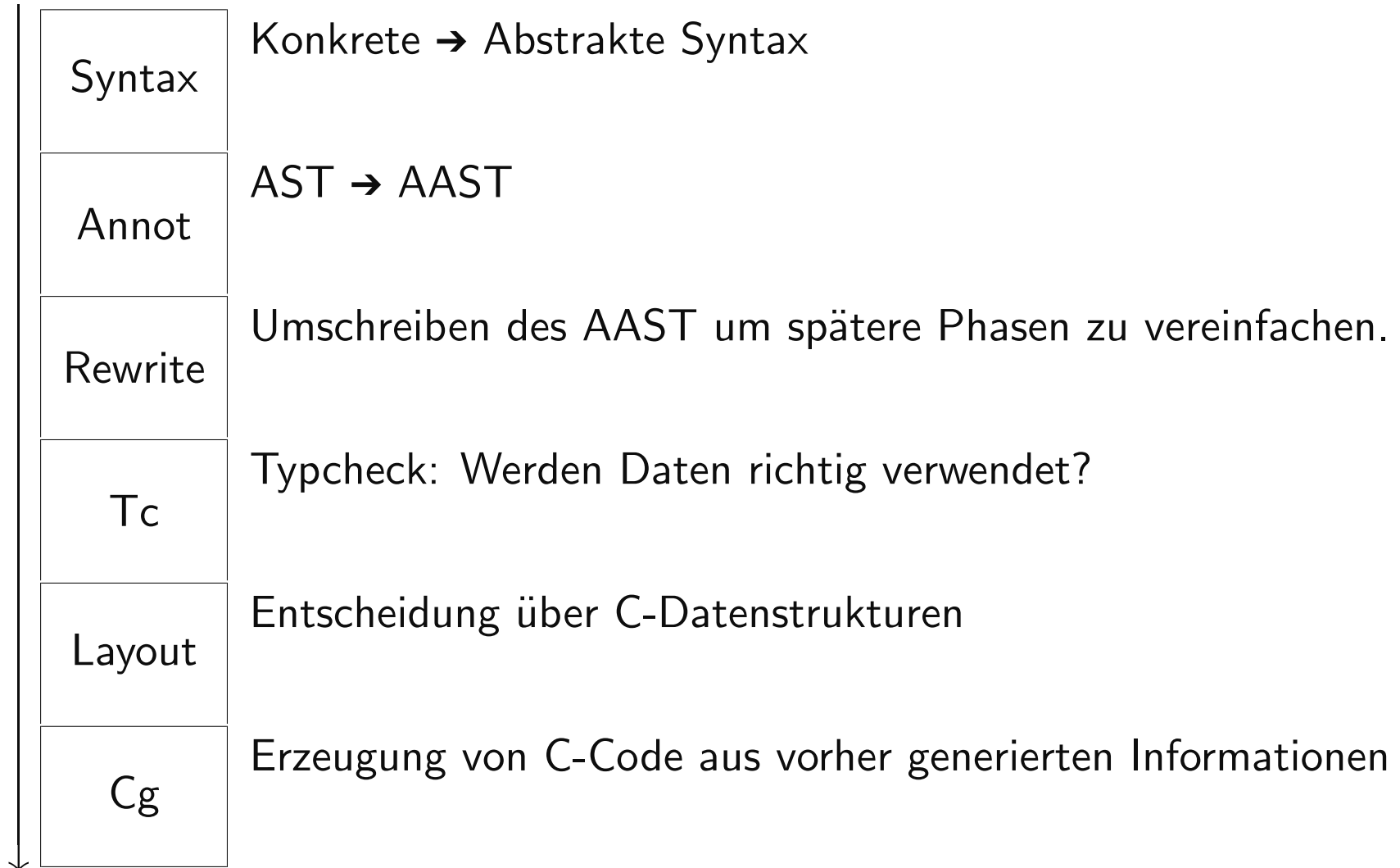
22. November 2005

---

## Bisher: Übersicht Compiler

- Motivation: Wir wollen die Details der Sprache Java zu verstehen
- Am vergangenen Donnerstag: Phasen & Module des Compilers
- Zwischenrepräsentation: Annotierte Abstrakte Syntax (AAST)
- Beispielprogramm `all_methods`: Programmieren mit der AAST
- Übungsblatt dazu: Eindeutigkeit von Namen prüfen

## Wdhg: Module des Compilers



---

Grüne Folien sind Quizfragen!  
→ 45 Sekunden für die Antwort

---

# Endlich: Klassen und Objekte

- Klassen sind zentral in Java
- ⇒ Abhängigkeiten zu vielen anderen Sprachkonzepten
  - Typen
  - Konstruktion von Objekten
  - Datenstrukturen für Objekte
- Ziel heute: Die Konzepte sauber trennen
- Nächste Stunde: Mit minimaler Definition durch den Compiler
- Folgende Wochen: Weitere Konzepte hinzufügen
  - Vererbung
  - Abstrakte Klassen
  - Statische Felder & Methoden

---

## Ausgangspunkt am Beispiel

```
class Point {
    private int x;
    private int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public void moveBy(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
    public int getX() { return x; }
    public int getY() { return y; }
}
```

---

## Ausgangspunkt am Beispiel

```
class Main {
    public static void main(String argv[]) {
        Point p = new Point(10,20);
        p.moveBy(1,2);
        System.out.println_int(p.getX());
        System.out.println_int(p.getY());
    }
}
```

---

## Konzepte im Beispiel

- Klasse
- `this`-Objekt
- Instanzvariable
- Konstruktor
- Felder
- Methoden
- Lokale Variablen
- Parameter
- Methodenaufruf
- Primitive Typen
- Schutzmechanismen
- Arithmetik
- Instanziierung
- Referenzsemantik
- Statische Felder
- Eingebaute Klassen

→ Aufgabe: Erkläre diese Konzepte und damit die Sprache Java

---

An welchen Stellen im Beispiel kann man diese  
Konzepte sehen?

---

## Der Plan

- Konzepte am Beispiel durchgehen  
→ Vorbereitung auf Compilerbau
- Syntax des Beispiels genau lesen  
→ Wie sieht die AAST aus?
- Erwartetes Verhalten von Java am Beispiel nachvollziehen  
→ Wie muss sich der C-Code am Ende verhalten?
- Beziehungen zwischen Konzepten klarstellen  
→ Wo ergeben sich Abhängigkeiten in der Implementierung?
- Notwendige Präzisierungen festhalten  
→ An welchen Stellen müssen wir bei der Implementierung aufpassen?

---

# Klasse und Instanz

```
class Point {  
    ...  
}
```

- Syntax: Eine **Klassendeklaration**
- Eine Klasse definiert eine Menge von Objekten, ihre **Instanzen**

```
new Point(10,20)
```

- Der Ausdruck `new <K(...)>` erzeugt eine neue Instanz von  $K$ , er **instanziert**  $K$ .
- ⇒ **Objekte** sind in Java Instanzen einer Klasse (. . . oder sie sind Arrays)

---

# Klasse

```
class Point {  
    private int x;  
    private int y;  
    public void moveBy(int dx, int dy) {  
        x = x + dx;  
        y = y + dy;  
    }  
}
```

- Syntax: Felddeklaration, Methodendeklaration, Methodenkopf, formale Parameter, Rückgabetyt
- Alle Instanzen einer Klasse haben gemeinsame Daten und Funktionalität
  - Die Daten werden in **Feldern** gespeichert
  - Die Funktionalität liegt in Form von **Methoden** vor
  - Die Klasse definiert die **Implementierung** der Methoden

---

## Klassen als Typen

```
public static void main(String argv[]) {  
    Point p = new Point(10,20);  
    ...  
}
```

- Syntax: Variablendeklaration
- Lokale Variable `p` (→ KvP)
- `p` ist initialisiert mit Instanz von `Point` (→ KvP)
- Der Name der Klasse dient als `Typ` ihrer Instanzen (→ KvP)  
→ Typen beschreiben die Instanzen

---

## Einschub KvP: Typen

- Typen beschreiben die Struktur von Laufzeitwerten
- Der Compiler soll anhand dieser Struktur prüfen, ob ein bestimmter Zugriff auf ein Datum im Programm zur Laufzeit einen Fehler verursachen wird (bzw. könnte, weil “wird” unentscheidbar ist)
- Beispiel:
  - Bei Feldzugriff `x.i` muss `x` ein Objekt sein, das ein Feld `i` besitzt.
  - Der Compiler prüft, ob der Typ von `x` eine Klasse ist, die für ihre Instanzen ein Feld `i` definiert.

---

## Einschub KvP: Typsystem

- Der Compiler muß die Berechnungen des Programms soweit nachvollziehen, daß er die Typen von allen Zwischenergebnissen kennt.
  - Beispiel: `x.f().g()`
    - `x` muß als Typ eine Klasse  $K$  haben, die eine Methode `f` definiert
    - Der Compiler muß prüfen, dass `f` als Rückgabe eine Instanz einer Klasse  $K'$  liefert, die eine Methode `g` enthält.
      - Die Klasse  $K$  deklariert eine Methode `f` mit Rückgabe  $K'$  im Kopf
      - $K'$  hat eine Methode `g`
      - Alle `return` Anweisungen im Rumpf von `f` geben tatsächlich eine Instanz von  $K'$  zurück. (Das passiert vorher beim Check von `f`.)
  - Die **Typregeln** für ein Sprachkonstrukt beschreiben die Einschränkungen, die der Compiler prüfen muss
- ⇒ Beinahe jedes Sprachkonstrukt besitzt Typregeln, die wir explizit besprechen werden.

---

Nenne möglichst viele Typregeln von Java!

---

## Blick in die Java Language Specification

§8: **Class declarations** define new **reference types** and describe how they are implemented.

§4.3: There are three kinds of reference types: **class types**, interface types, and array types

§4.3.1: An object is a **class instance** or an array. The reference values (often just **references**) are pointers to these objects, and a special **null reference**, which refers to no object.

§4.3.1: A class instance is explicitly created by a **class instance creation expression**.

---

## Einschub KvP: Referenzsemantik

- Zum Begriff `references` der JLS
- Objekte haben in Java **Referenzsemantik**
  - Objekte liegen im Speicher getrennt von lokalen Variablen
  - Variablen enthalten nur Referenzen (Zeiger) auf Objekte
  - Bei Zuweisungen werden die Referenzen, nicht die Objekte, kopiert
  - Bei Methodenaufrufen werden Referenzen, nicht Objekte, übergeben
- Folgen
  - Effizient, weil Referenzen sehr klein sind (ein Maschinenwort)
  - Änderungen durch eine Referenz auf ein Objekt sind durch alle Referenzen auf das Objekt sichtbar.
- Am Beispiel: `p.moveBy()` ändert `x`, `y` in dem Objekt, auf das in `p` eine Referenz gespeichert ist

---

## Instanziierung im Detail

§15.9.4: At run time, evaluation of a class instance creation expression is as follows. [. . .] [First,] space is allocated for the new class instance. [. . .] Next, the actual arguments to the constructor are evaluated, left-to-right.[. . .] Next, the selected constructor of the specified class type is invoked.[. . .] The value of a class instance creation expression is a reference to the newly created object of the specified class.

→ Instanziierung besteht eigentlich aus vier Schritten

- Speichieranforderung für das neue Objekt
- Auswertung der Konstruktorargumente
- Aufruf des Konstruktors für das neue Objekt
- Rückgabe des neuen Objekts

---

# Konstruktor

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

- Syntax:  $\langle \textit{Klassenname} \rangle (\langle \textit{Parameter} \rangle) \{ \langle \textit{Rumpf} \rangle \}$
  - Konstruktoren initialisieren neu angelegte Objekte
  - Sie werden vom Compiler bei einem `new` automatisch aufgerufen
- ⇒ Analogie Funktionsaufruf in Mini
- Unterschied: Zugriff auf das neu erzeugte Objekt mit `this`

---

Formuliere einen besonders guten Grund, warum wir die OO-Konzepte mit der JLS vergleichen sollten!

---

## Das this-Objekt

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

- this enthält das Objekt, für das der Konstruktor oder die Methode aufgerufen wurde
- Es ist eine Referenz, man kann die Felder des Objektes ändern

---

## Das this-Objekt

```
public void moveBy(int dx, int dy) {  
    x = x + dx;  
    y = y + dy;  
}
```

- Jeder Feldzugriff erfolgt auf die Felder eines bestimmten Objektes
- Der Compiler ergänzt ein implizites this

```
public void moveBy(int dx, int dy) {  
    this.x = this.x + dx;  
    this.y = this.y + dy;  
}
```

⇒ this kann beim Feldzugriff und Methodenaufruf entfallen

---

## Der Blick in die JLS

§15.8.3: The keyword `this` may be used only in the body of an instance method [. . .] or constructor [. . .] of a class. If it appears anywhere else, a compile-time error occurs.

When used as a primary expression, the keyword `this` denotes a value, that is a [reference to the object for which the instance method was invoked](#), or to the object being constructed.

The type of `this` is the class `C` within which the keyword `this` occurs. [. . .].

- Ist die Einschränkung sinnvoll?
- Offen: Wo ist “*anywhere else*”?
- Typregel: `this` hat als Typ die Klasse, in der es verwendet wird.

---

## Methodendeklaration

```
public void moveBy(int dx, int dy) {  
    x = x + dx;  
    y = y + dy;  
}
```

- Syntax: Methodenkopf, Methodenrumpf, formale Parameter, Anweisungen, Ausdrücke
- Die Methode steht in Instanzen der Klasse `Point` zur Verfügung
- Der Kopf bestimmt, welche Parameter die Methode erwartet und welchen Rückgabewert sie liefert
- Der Rumpf wird beim Aufruf der Methode ausgeführt
- (Dabei ist `this` an das Objekt gebunden, für das die Methode aufgerufen wurde.)

---

## Methodendeklarationen in der JLS

§8.4: A method declares executable code that can be invoked, passing a fixed number of values as arguments.

§8.4.2 The **signature** of a method consists of the **name** of the method and the **number and types of formal parameters** to the method. A class may not declare two methods with the same signature, or a compile-time error occurs.

- Wozu wird die Signatur benötigt? → Überladung (später)
- Offen: Warum gehört der Rückgabetyt nicht zur Signatur?

---

## Methodenaufruf

```
p.moveBy(1,2);  
System.out.println_int(p.getX());
```

- Methode `moveBy` wird mit  $dx = 1$ ,  $dy = 2$  und `this = p` ausgeführt.
- Dadurch läuft der Code dieser Methode ab.
- Danach wird die Methode `getX()` in `p` (ohne Parameter) aufgerufen.
- Danach wird die Methode `println_int` im Objekt `System.out` ausgeführt, das Argument ist das Ergebnis des vorherigen Schrittes.
- Hinweis: *call-by-value* Semantik, primitive Werte und Objekt-Referenzen werden in die Parameter kopiert!

---

## Methodenaufruf in der JLS

§15.12: A **method invocation expression** is used to invoke a class or instance method.

§15.12.4: At run time, method invocation requires five steps. First, a **target reference** [is] computed. Second, the **argument expressions** are evaluated. Third, the accessibility [is checked . . .]. Fourth, **the actual code for the method to be executed is located**. Fifth, a **new activation frame is created** [. . .] and control is transferred to the method code.

- Offen: Unterscheidung *class or instance method*

---

## Die fünf Schritte am Beispiel

```
p.moveBy(1,2);
```

1. Das Zielobjekt ist das in `p` gespeicherte
2. Die Argumentausdrücke sind Literale 1 und 2, die zu `int`-Werten 1 und 2 auswerten
3. Zugriffsrechte: `public` ✓
4. Wir brauchen den Code aus `Point`
5. Es wird eine Ausführungsumgebung angelegt, in der die Parameter mit `dx = 1`, `dy = 2` belegt sind und `this = p` ist.  
Schließlich wird der Code der Methode in dieser Umgebung angelegt

---

Welche Schritte gehören zur **Instanziierung** einer Klasse?

---

## Vergleich: Funktionsaufruf in Mini

```
let rec eval e env =
  match e with
  | Const i -> Int i           für Literale 1,2
  | Var n -> lookup_env env n  nachschlagen von p
  | ...
  | App (rator, rands) ->
    let ratorVal = eval rator env in
    let randVals = List.map (fun e -> eval e env) rands in
    match ratorVal with
    | Clos (paras, body, clos_env) ->
      eval body (extend_env clos_env paras randVals)
    | _ -> failwith "Operator is not a function"
```

- Ähnlich:
  - Argumente rekursiv auswerten
  - Funktion bestimmen
  - Rumpf der Funktion in erweiterter Umgebung auswerten
- Fehlt: Zielobjekt (das `this` Objekt für die Methode)

---

## Ein Seitenblick auf SmallTalk

- Fragen:
  - Sind die Feststellungen am Beispiel allgemeingültig?
  - Ist Java eigentlich ein Sonderfall unter den OO Sprachen?
  - Haben wir die Konzepte richtig verstanden?
- Vergleich SmallTalk
- SmallTalk ist vielleicht *die* objekt-orientierte Sprache schlechthin.
- Erinnerung These: Konzepte sind übertragbar

---

## Ein Seitenblick auf SmallTalk

A *class* describes the implementation of a set of objects that all represent the same kind of system component. The individual objects described by a class are called its *instances*. A class describes how they carry out their operations. [. . .] An object's private properties are a set of *instance variables* that make up its private memory and a set of *methods* that describe how to carry out its operations.

- Quelle: *Goldberg, Robson: SmallTalk 80*, Seite 8
- Parallelen:
  - Methoden
  - Objekte sind Instanzen von Klassen
  - Die Methoden eines Objektes sind durch die Klasse festgelegt, deren Instanz es ist.

---

## Ein Seitenblick von SmallTalk

An object consist of a private memory and some operations. [. . .] A *message* is a request for an object to carry out one of its operations. [. . .] The *receiver*, the object to which the message was sent, determines how to carry out the requested operation. (ebd., Seite 6)

- Parallelen:
  - Methoden
  - Objekte = Daten + Methoden
  - Methodenaufruf für spezifisches Objekt
  - Das `this`-Objekt

---

## Rückblick: Die Konzepte im Beispiel

Welche Konzepte haben wir behandelt?

- |                             |                      |
|-----------------------------|----------------------|
| ✓ Klasse                    | ✓ Methodenaufruf     |
| ✓ <code>this</code> -Objekt | ✗ Primitive Typen    |
| ✓ Instanzvariable           | ✗ Schutzmechanismen  |
| ✓ Konstruktor               | ✗ Arithmetik         |
| ✓ Felder                    | ✓ Instanziierung     |
| ✓ Methoden                  | ✓ Referenzsemantik   |
| ✓ Lokale Variablen          | ✗ Statische Felder   |
| ✓ Parameter                 | ✗ Eingebaute Klassen |

→ Die fehlenden sind nur Spezialfälle und Details