

Software-Entwicklung 1

V06: Objektgeflechte

Prof. Maalej & Team




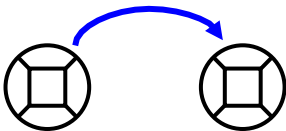
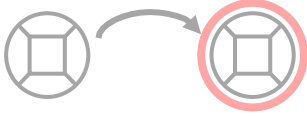
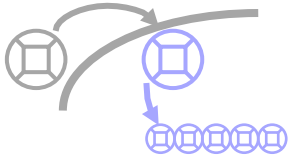
@maalejw



Status der 5. Übungswoche

Zeit	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
Vor mittag	Gruppe 1 Erfüllt: 80%	Gruppe 3 Erfüllt: 67%	Gruppe 5 Erfüllt: 80%	Gruppe 6 Erfüllt: 83%	Gruppe 8 Erfüllt: 82%
Nach mittag	Gruppe 2 Erfüllt: 80%	Gruppe 4 Erfüllt: 71%	Vorlesung	Gruppe 7 Erfüllt: 79%	

Inhaltliche Gliederung von SE1

Stufe	Titel	Themen u.a.	Woche
1	 + +  Algorithmisches Denken	Prozedur, Fallunterscheidung, Zählschleife, Bedingte Schleife	1 – 2
2	 Objektorientierte Programmierparadigma	Klasse, Objekt, Konstruktor Methode, Parameter, Feld, Variable, Zuweisung, Basistypen	3 – 5
3	 Benutzung von Objekten	Klasse als Typ, Referenz, UML Schleife, Rekursion, Zeichenketten	6 – 8
4	 Testen, Interfaces, Static, Arrays	Black-Box-Test, Testklasse, Interface, Sammlungen benutzen, Arrays	9 – 10
5	 Sammlungen	Sammlungen implementieren: Array-Liste, verkettete Liste, Hashing; Sortieren; Stack; Graphen	11 – 14

Überblick

1

Datentypen – Rückblick

2

Benutzerdefinierte Typen

3

UML

Der Typbegriff

- Unter dem Begriff **Typ** (oder auch **Datentyp** genannt) versteht man:
„die Zusammenfassung von Wertebereichen und Operationen zu einer Einheit.“ [Informatik-Duden]
- Für jeden Typ ist nicht nur die **Wertemenge** definiert, sondern auch die **Operationen**, die auf diesen Werten zulässig sind

Java-Beispiele:



Datentyp: int

Wertemenge: { -2^{31} ... $2^{31}-1$ }

Operationen: ganzzahlig Addieren,
ganzzahlig Multiplizieren, ...

Datentyp: boolean

Wertemenge: {true, false}

Operationen: Und, Oder, ...

Elementare Typen in Java



Datentyp	Zweck	Größe	Wertemenge	Beispiel
byte	ganze Zahlen	1 byte	-128 bis +127	byte b = 65;
short		2 bytes	-32.768 bis +32.767	short s = 65;
int		4 bytes	-2^{31} bis $2^{31} - 1$	int i = 65;
long		8 bytes	-2^{63} bis $2^{63} - 1$	long i = 65L;

Typprüfung bei statischer Typisierung

- Jeder Variable, Konstante, jedem Literal und jedem Ausdruck ist ein fester, nicht änderbarer Typ zugeordnet
- Der typ ist überprüfbar: **Typprüfung**
- In statisch typisierten Sprachen (Java, C#, C++, Pascal, Eiffel...) prüft der Compiler dies zur Übersetzungszeit



Beispiel:

Die **Addition** ist als binäre Operation auf zwei **int** Zahlen definiert, nicht aber für eine Zahl und einen Wahrheitswert.

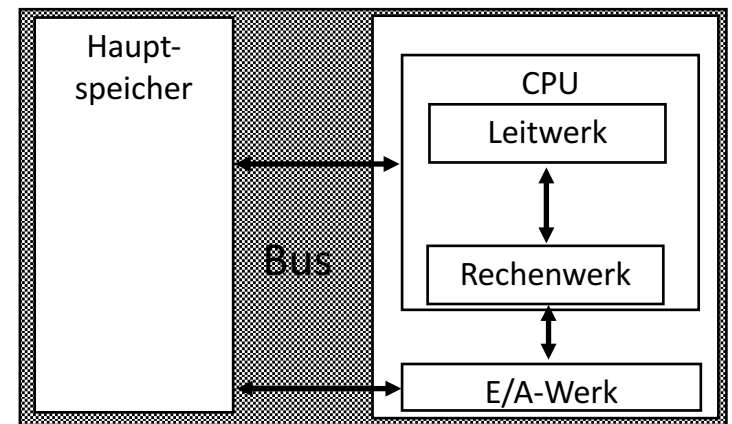
```
int sum = 12 + 6;  
int result = 12 + false;  
// Typfehler!
```

Smalltalk ist eine dynamisch typisierte Programmiersprache. Variablen werden nicht mit einem Typ deklariert. Dynamisch typisierte Sprachen gestatten nur eine Laufzeitprüfung.

Warum Typisierung von Programmiersprachen?

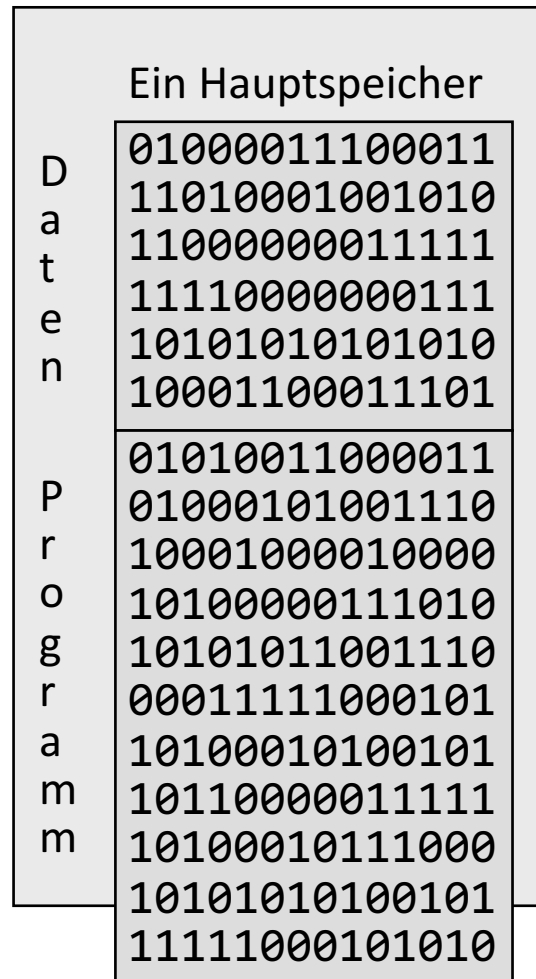
In von Neumann-Rechnern:

- Programme und Daten stehen im selben Speicher
- Der Hauptspeicher ist in Zellen gleicher Größe unterteilt, die durchgehend adressierbar sind
- Die Maschine benutzt Binärcodes für die Darstellung von Programm und Daten

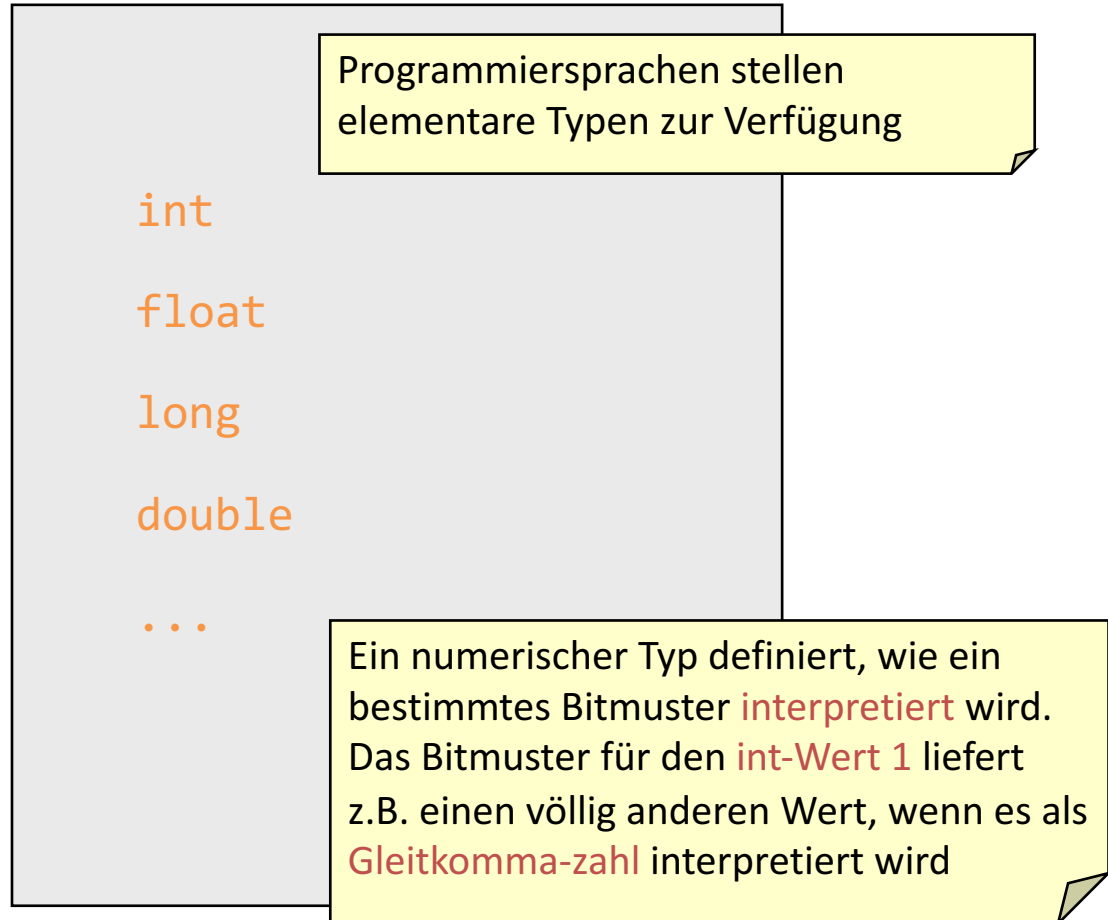


Historisch: elementare Datentypen

NACKT:
Maschinenprogramme



LEICHT BEKLEIDET:
Imperative Sprachen seit Fortran



Der „klassische“ Typbegriff

- In imperativen Programmiersprachen bezieht sich der Typbegriff auf Werte, die als Daten in Variablen gehalten werden. Daher spricht man oft von **Datentypen**.
- Damit verbunden ist die Vorstellung, dass **jeder Wert** zu genau einem Datentyp gehört, und dass es dafür **zulässige Operationen** gibt.
- In statisch typisierten (imperativen) Programmiersprachen wird jedem Bezeichner vor seiner Verwendung ein fester Typ zugeordnet; dies nennt man **Deklaration**.
- Wesentliche Arbeiten zum klassischen Typkonzept stammen von C.A.R. ("Tony") **Hoare**. Sie sind heute noch wegweisend.



Sir Charles Antony Richard Hoare

Der Typbegriff nach Hoare

- A type determines the *class of values* which may be assumed by a *variable* or *expression*.
- Every value belongs to one and only one type.
- The *type of a value* ... may be deduced from its *form* or *context*, *without* any knowledge of its value as computed at run time.
- Each *operator* expects operands of some fixed type, and delivers a result of some fixed type ...
- The properties of the values of a type and of the primitive operations defined over them are specified by means of a set of axioms.
- *Type information* is used in a high-level language both to *prevent or detect meaningless constructions* in a program, and to determine the method of *representing and manipulating data* on a computer.
- The types in which we are interested are those already familiar to *mathematicians*; namely, Cartesian Products, Discriminated Unions, Sets, Functions, Sequences, and Recursive Structures.

Ein Typ definiert eine Menge an Werten, die eine Variable oder ein Ausdruck annehmen kann.

Jeder Wert gehört zu genau einem Typ.

Typinformation ist statisch aus dem Quelltext ermittelbar.

Operatoren sind getypt (Bsp.: && in Java erwartet boolesche Operanden)

Ein Typ definiert Operationen...

Typinformation schützt und legt Semantik fest.

©Hoare

C.A.R Hoare, *Notes on Data Structuring*. In: Dahl, Dijkstra, Hoare: Structured Programming. Academic Press, 1972.
[Einer DER Klassiker über Datenstrukturen.]

Überblick

1

Datentypen – Rückblick

2

Benutzerdefinierte Typen

3

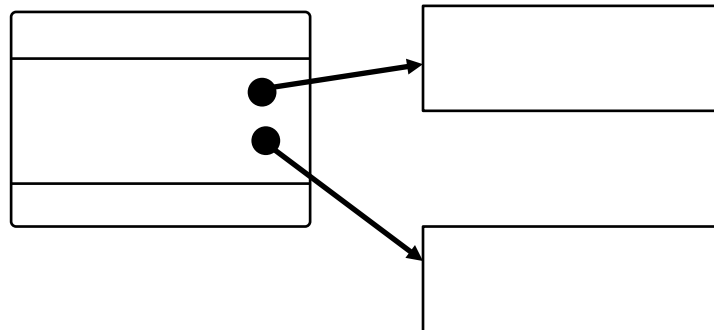
UML

Reichen die elementaren Datentypen?

- Verfügbaren Datentypen reichen nicht aus um Objekte des **Anwendungsbereichs** zu modellierenden
- Auf der Basis vorgegebener Datentypen sollen **anwendungsbezogene Datentypen** bereitgestellt werden
- Zwei Lösungsansätze:
 - Große Vielfalt **vordeklarerter Datentypen**
 - Kleiner Satz von elementaren Typen und flexible **Kombinationsmechanismen**, sodass **neuer Datentypen** definiert werden können
 - **Benutzerdefinierte Typen**
 - Wird in fast allen modernen Sprachen verwendet

Referenztypen

- Bisher waren die Felder der Klassen von elementaren Datentypen
- Menge der möglichen Zustände durch die Deklarationen von Variablen und Konstanten im Klassentext zur Übersetzungszeit festgelegt
- Diese Begrenzung wird durch dynamische **Objektstrukturen** (Objektgeflechte) aufgehoben
- Bei Objektgeflechten kann die Anzahl der Objekte zur Laufzeit variieren
- Voraussetzung für Objektgeflechte sind **Referenztypen**



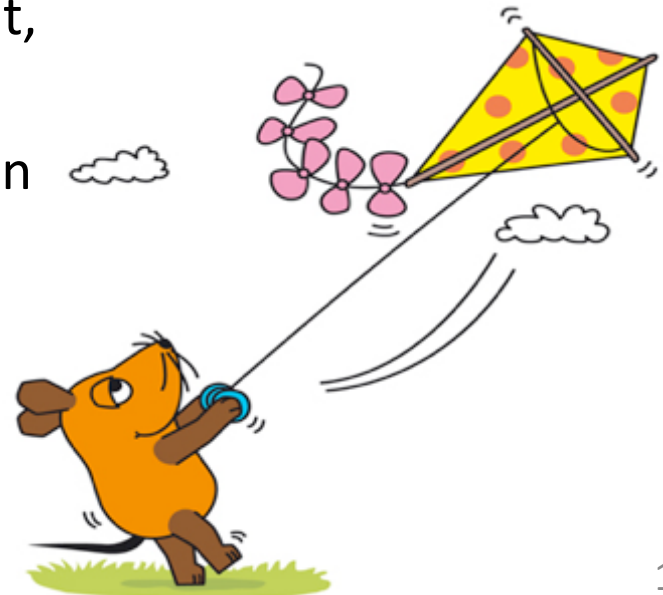
Konto
_saldo : int
...

Ampel
_rot : boolean
_gelb : boolean
_gruen: boolean
...

Waage
_gewicht : int
...

Referenzen allgemein

- Die Verbindung zwischen Klient und Dienstleister besteht aus einer expliziten **Referenz** (auch Verweis, Zeiger, Pointer)
- Ergebnis der **Erzeugung** des Dienstleister-Objekts (Konstruktoraufruf) wird eine Referenz geliefert
- Diese Referenz ist die „**Adresse**“ des neu erzeugten Objektes
- Die Referenz wird als ein **Wert** behandelt, der einer sog. **Referenzvariablen** im Klienten-Objekt zugewiesen werden kann

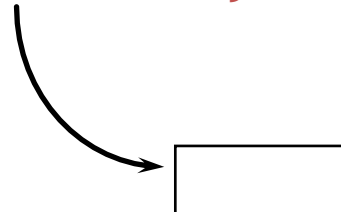


Wertvariablen

- Kennzeichen einer **Variable**:
 - Variablen müssen **deklariert** werden
 - Ein **Name** dient als **Bezeichner**
 - Bei der Deklaration muss ein **Typ** angegeben werden
- Bei den bisher betrachteten Variablen handelte es sich um **Wertvariablen**, da der verwendete Typ jeweils ein Werttyp war und zur Laufzeit die Variable mit einem Wert belegt wurde

Deklaration:

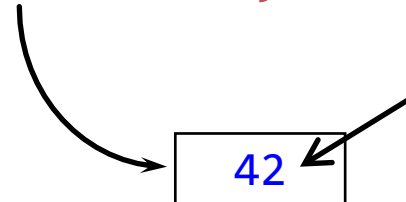
int antwort;



*Speicherplatz für
eine Variable vom
Typ int*

Zuweisung:

antwort = 42;

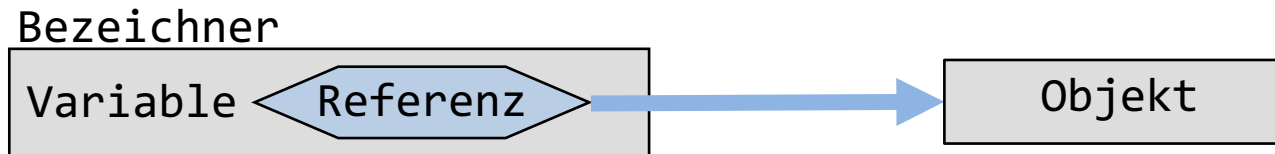


Belegung des
Speicherplatzes
mit einem Wert

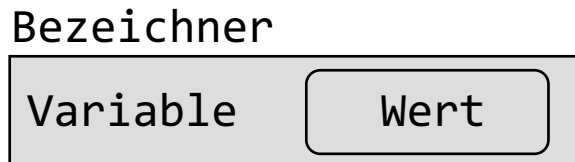
Referenzvariablen

- Bei einer **Referenzvariablen** sind zwei Dinge zu unterscheiden:
 - Ihre Belegung mit einer **Referenz auf ein Objekt**,
 - das **referenzierte Objekt**.
- Gegenüber einer Wertvariablen wird ein zusätzlicher Verweis (eine Indirektion) verwendet

Referenzvariable



Wertvariable

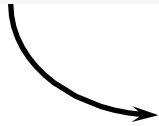


Referenzvariablen sind typisiert

- Auch Referenzvariablen haben einen Typ, einen **Referenztyp**
- **Jede Klasse** in Java definiert einen Referenztyp

Deklaration

```
Konto meinKonto;
```

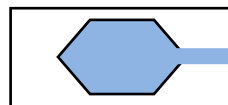
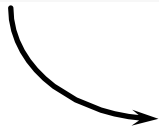


Konto

Speicherplatz für eine Variable vom Typ **Konto**

Zuweisung

```
meinKonto = new Konto();
```



Konto

Referenz



: Konto

_saldo = 2000
_verzinsung = 1.5

Referenztypen sind Typen

- Legen die **Menge** der Elemente und die möglichen **Operationen** auf den Elementen des Typs fest
- Die Elemente eines Referenztyps sind die Exemplare der Klasse
 - Die Wertemenge ist **unbeschränkt**
- Die Operationen des Referenztyps, sind die Methoden, die an den Exemplaren der Klasse aufgerufen werden können.
 - Compiler erkennt bei der Übersetzung die zulässigen Operationen

```
Kreis sonne;  
sonne = new Kreis();  
sonne.farbeAendern(„gelb“);  
sonne.
```

```
boolean    equals(Object)  
void       farbeAendern(String)  
Class<?>   getClass()  
void       groesseAendern(int)  
int        hashCode()  
void       horizontalBewegen(int)  
void       langsamHorizontalBewegen(int)  
void       langsamVertikalBewegen(int)  
void       nachLinksBewegen()  
void       nachObenBewegen()  
void       nachRechtsBewegen()  
void       nachUntenBewegen()
```

Kreis

```
void groesseAendern(int  
durchmesser)
```

Aendere den `_durchmesser` dieses Kreises in `durchmesser` (Angabe in Bildschirmpunkten). `durchmesser` muss groesser gleich Null sein.

Parameters

`durchmesser` - Neuer Durchmesser dieses Objekts

Datentyp: Kreis

Wertemenge: Menge der Kreis-Exemplare

Operationen:
farbeAendern,
groesseAendern, ...

Schnittstelle und Typ

- **Öffentliche Methoden** einer Klasse definieren die Schnittstelle ihrer Exemplare
- Inzwischen wissen wir zusätzlich:
 - Eine Klasse definiert auch einen **Typ** (einen Referenztyp)
 - Wir können **Referenzvariablen** dieses Typs deklarieren
 - Die **Operationen**, die wir über Referenzvariablen aufrufen können, sind genau die **öffentlichen Methoden** der Klasse, die den Typ definiert.

Aufrufbaren
Operationen eines
Referenztyps == Schnittstelle der
definierenden Klasse

Referenzen in Java



- **Alle Objekte** in Java werden über Referenzen verwendet
- Bei der **Zuweisung** einer Referenzvariablen wird die **Referenz kopiert**, nicht das referenzierte Objekt!
- Der **Gleichheitstest** mit dem Operator „==“ auf Referenzvariablen prüft die **Gleichheit der Referenzen** (zeigen sie auf dasselbe Objekt?), nicht der referenzierten Objekte
- Referenzvariablen können den besonderen Wert **null** haben
 - „zeigt auf kein Objekt“
 - Exemplarvariablen werden automatisch auf diesen Wert initialisiert
- Der Zugriff auf die Methoden eines referenzierten Objekts erfolgt über die **Punktnotation** (als Methodenaufruf)

Referenzen und boolesche Ausdrücke

- Wenn der Wert eines Ausdrucks schon durch eine Teilauswertung feststeht, wird der Rest des Ausdrucks nicht weiter ausgewertet (JLS § 15.23. u. 15.24.).
- Beispiele:

```
true || (energie < 10) // ist immer true (unabhängig von energie)
```

```
false && (_saldo > 0) ist immer false (unabhängig von _saldo)
```

- Gängig ist eine Überprüfung auf einen Wert ungleich null

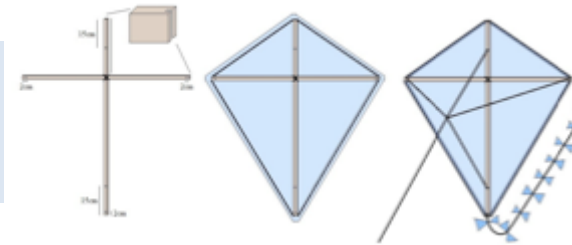
```
if (konto != null && konto.istGedeckt(betrag))
```

- Für den Fall, dass die Variable konto null enthält wird Auswertung gestoppt
- **NullPointerException** beim Methodenaufwurf wird verhindert
- Das ist nicht in allen Programmiersprachen so eindeutig geregelt wie in Java!

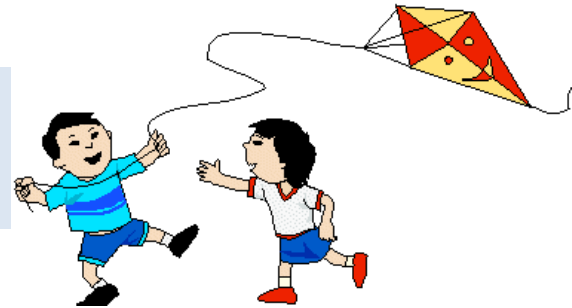
Wie kommt ein Klient an eine Referenz?

- Drei Möglichkeiten um das in einer Methode zu bekommen:

Das Klient-Objekt **erzeugt** das Dienstleister-Objekt **innerhalb** der **Methode** selbst



Es erhält die Referenz auf den Dienstleister unmittelbar als **Parameter** der **Methode**

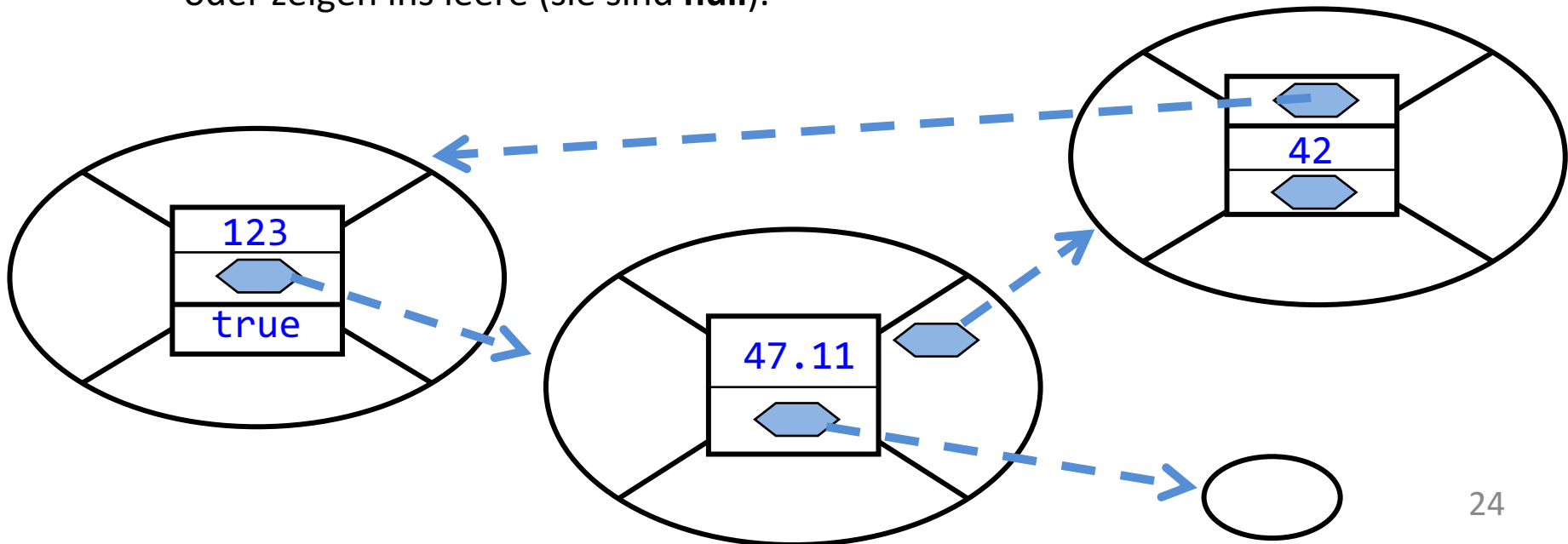


Er hat bei seiner eigenen Erzeugung oder bei einem **vorigen Methodenaufruf** eine Referenz erhalten, die er in einem **Feld abgelegt** hat; sie steht ihm dann in allen Methoden zur Verfügung



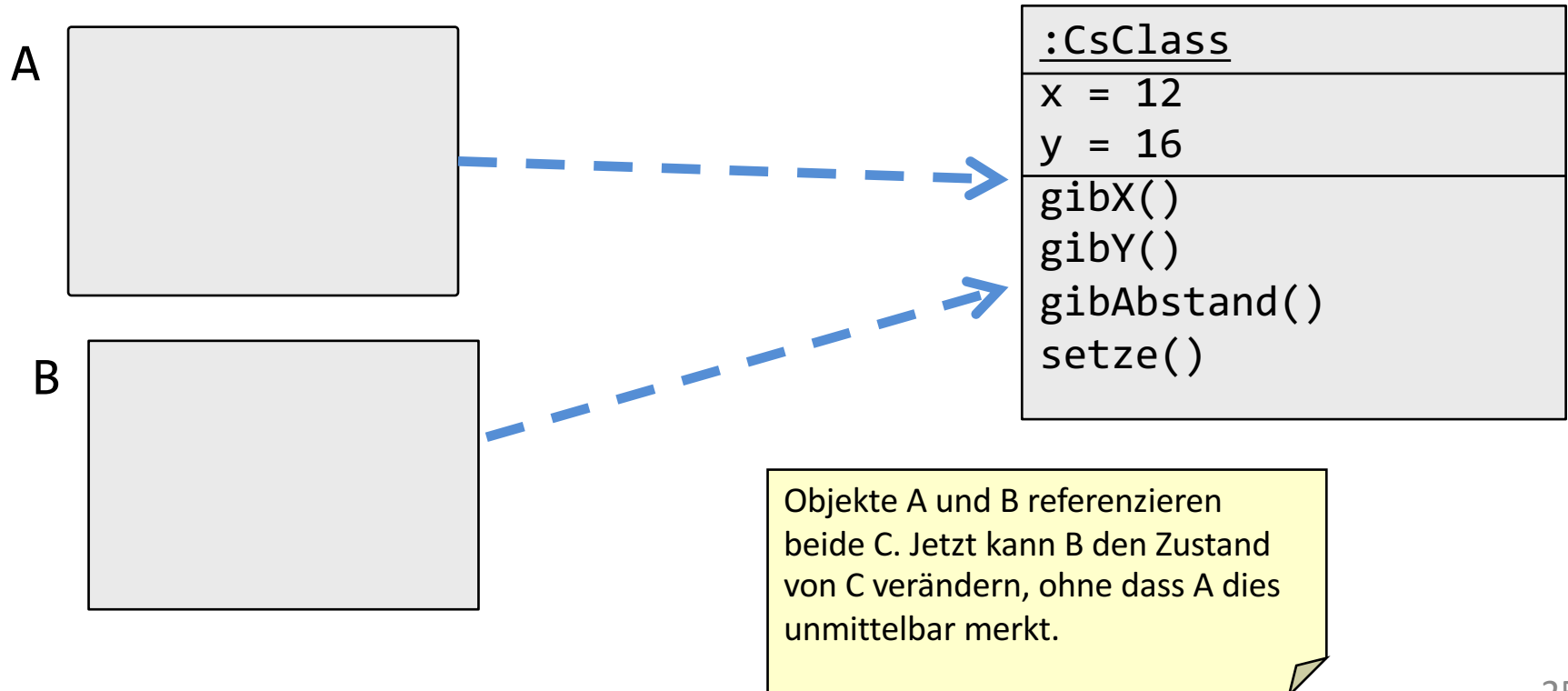
Allgemeines Objektmodell von Java

- **Objekte** enthalten die in ihrer erzeugenden Klasse festgelegte Struktur von Feldern. Die jeweilige Belegung der Felder mit Werten und Referenzen definiert den Zustand eines Objekts.
- **Werte:**
 - Auswahl der Werttypen in Java fest vorgegeben (**int** etc.)
- **Referenzen:**
 - zeigen auf andere Objekte (es entstehen **Objektgeflechte**),
 - oder zeigen ins leere (sie sind **null**).



Alias-Problem

- Zwei Referenzvariablen in **verschiedenen Objekten** können auf **dasselbe Objekt** verweisen.
- Lokal ist oft nicht entscheidbar, ob sich Veränderungen am Zustand eines referenzierten Objekts ergeben haben



Alias-Problem: Problem oder Chance?

- Es können **beliebig komplizierte** Strukturen über Referenzen konstruiert werden
- **Starke Verbindungen** mit Referenzen in einem Softwaresystem **erschweren** die **Wartbarkeit** und die formale Betrachtungen zur Korrektheit
- Andererseits können mit Referenzen auch sehr mächtige und effiziente Strukturen gebaut werden



Zusammenfassung I

- 1 Java unterscheidet fundamental zwei Typfamilien: **Werttypen** und **Referenztypen**.
- 2 Die Menge der **Werttypen** ist **fest** in der Sprache **definiert** und kann nicht erweitert werden.
- 3 Referenztypen werden durch Klassen definiert; es können **beliebig neue Referenztypen** definiert werden.
- 4 Referenztypen sind das zentrale Mittel objektorientierter (und auch imperativer) Programmiersprachen, um **Objektgeflechte** zu konstruieren.

Zusammenfassung II

5 **Referenzen** oder **Zeiger** sind in Programmiersprachen unterschiedlich realisiert. Teilweise kann der Wert einer Referenz selbst verändert werden (z.B. in C und C++).

6 Dadurch werden Programme **schwerer wartbar und beherrschbar**.

7 Java ist in dieser Hinsicht eine **sichere** Sprache: Die Referenzen auf Objekte können nicht manipuliert/verändert werden.

8 Java ist außerdem eine einfache Sprache: Alle Parameter werden **per Wert** übergeben, auch die Referenzen auf Objekte.

Überblick

1

Datentypen – Rückblick

2

Benutzerdefinierte Typen

3

UML

Objektorientierte Aktivitäten

ca.1995

Objektorientierte
Metamodelle

Welche Modellierungsmittel
haben wir zur Verfügung?

ca. 1992

Objektorientierte
Vorgehensmodelle

Wie gehen wir in
Projekten vor?

Wie analysieren
wir einen
Anwendungsbereich?

ca. 1988

Objektorientierte
Analyse

Objektorientierte
Modellierung

Wie modellieren wir
ein System für einen
Anwendungsbereich?

Wie programmieren
wir mit einer
oo Sprache?

Objektorientierte
Programmierung

ca. 1967

Objektorientierter
Entwurf

ca. 1986

Wie entwerfen
wir größere
Programme? 30

Die UML als Notation und Technik

- Bei Analyse, Modellierung und Programmierung benutzen wir eine einheitliche Notation - die Unified Modeling Language (UML)
- UML ist
 - eine Sammlung von **Diagrammtypen** und Modellierungstechniken, die ursprünglich aus 3 objektorientierten Methoden zusammengestellt wurde
 - heute ein Quasi-Standard für die Darstellung von objektorientierten Modellen
- UML wurde ursprünglich von einer Firma (Rational) entwickelt, wird aber jetzt von einem weltweiten Konsortium (OMG) betreut.

<http://www.omg.org/technology/documents/formal/uml.htm>

OMG™ is an international, open membership, not-for-profit computer industry consortium. OMG Task Forces develop enterprise integration standards for a wide range of technologies, and an even wider range of industries. OMG's modeling standards enable powerful visual design, execution and maintenance of software and other processes.

Die Unified Modeling Language

- The UML is a language for
 - visualizing...
 - specifying...
 - constructing...
 - documenting...

...the artifacts of a software-intensive system



Die UML ist eine Sprache, um die Elemente eines software-intensiven Systems zu

- visualisieren
- spezifizieren
- konstruieren
- dokumentieren

Die Diagrammtypen der UML

Unser Fokus in SE1

- **Structure Diagrams:**

- Class Diagram
- Object Diagram
- Composite Structure Diagram (2.0)
- Component Diagram
- Deployment Diagram
- Package Diagram

- **Behavior Diagrams:**

- Activity Diagram
- Use Case Diagram
- State Machine Diagram

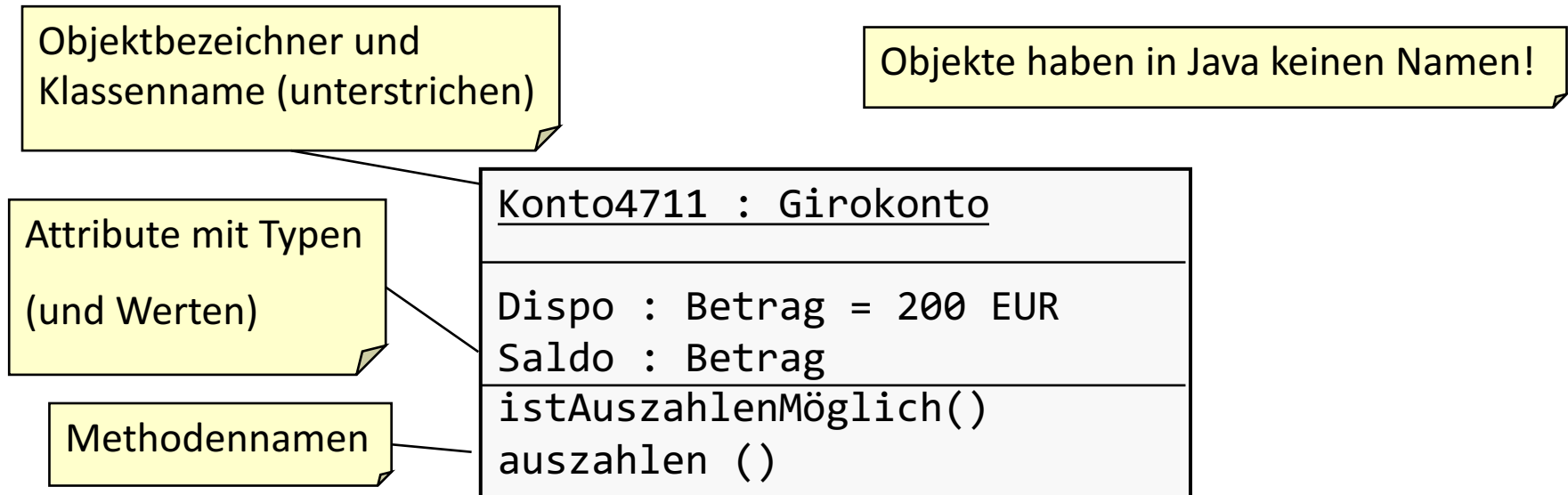
- **Interaction Diagrams:**

- Sequence Diagram
- Communication Diagram
- Interaction Overview Diagram (2.0)
- Timing Diagram (2.0)



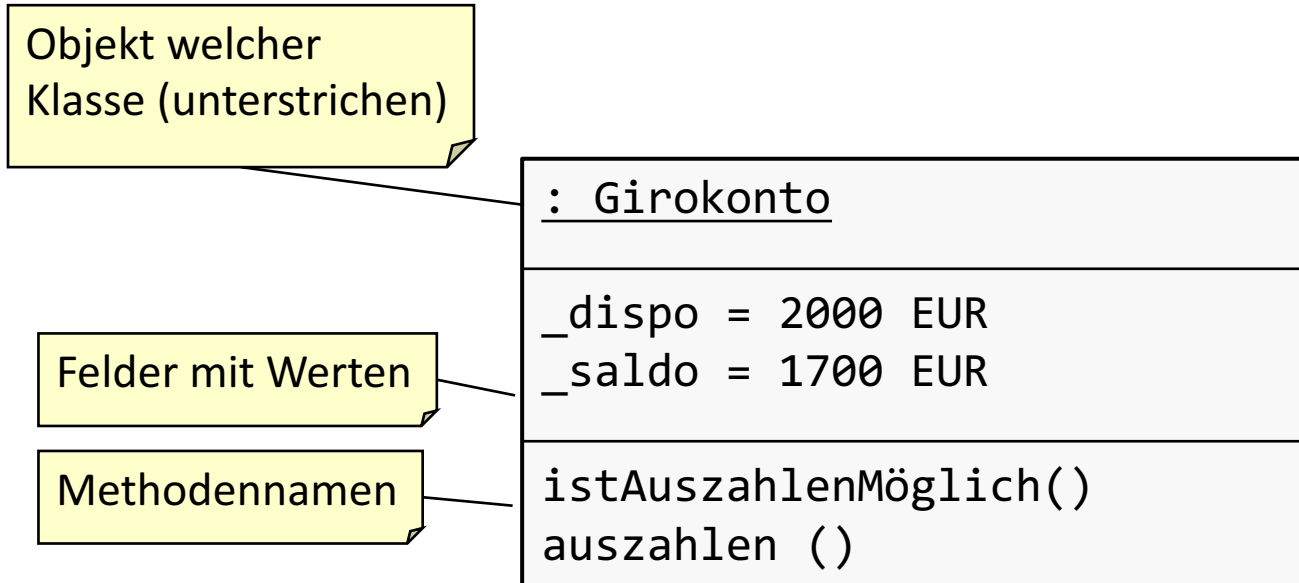
OMG-Unified Modeling Language, v2.0

Objektdiagramm, formal korrekt



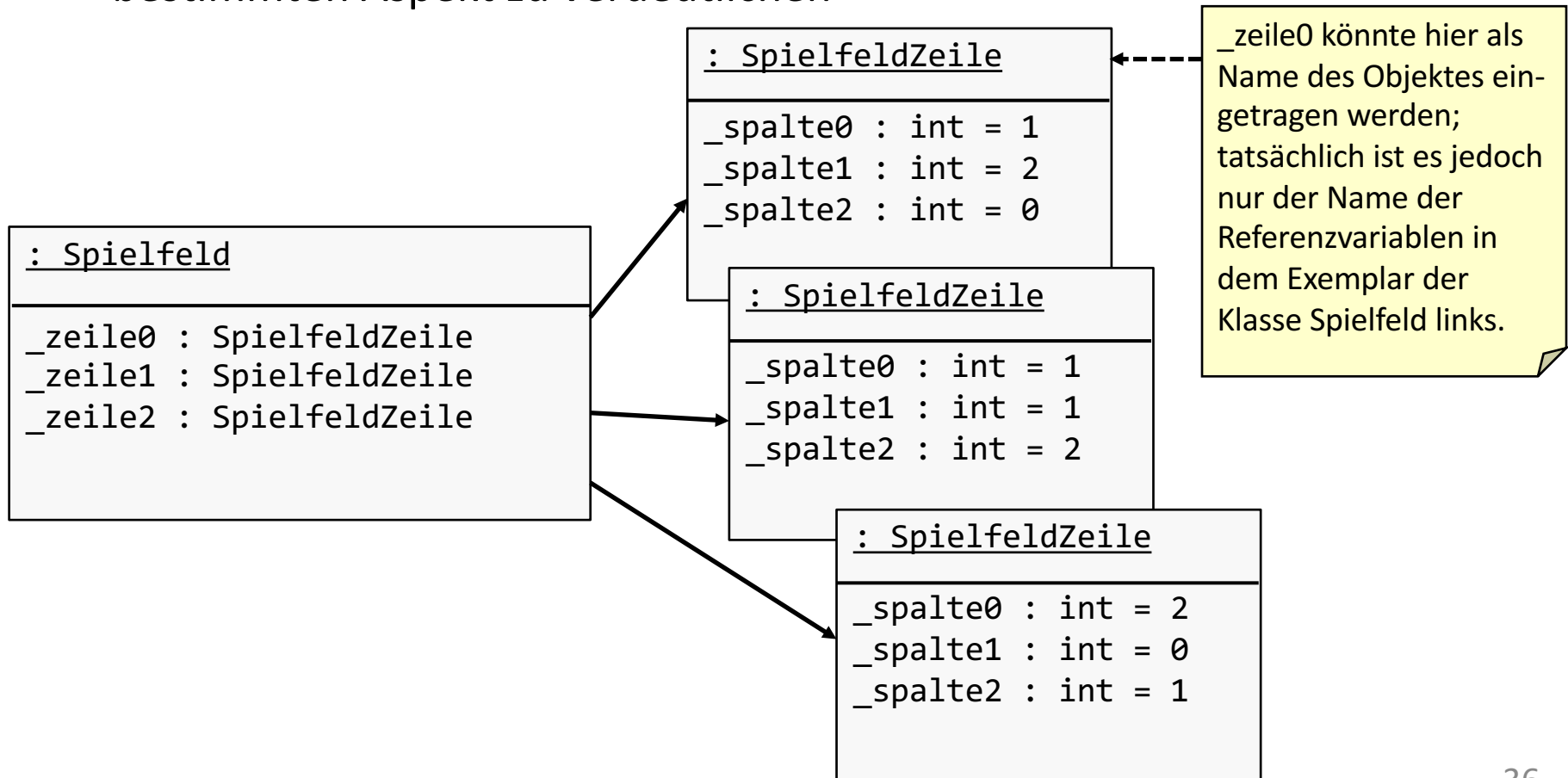
- Entweder der Objektbezeichner oder der Klassenname dürfen weggelassen werden; fehlt der Objektbezeichner, muss ein Doppelpunkt vor dem Klassennamen stehen.
- Felder heißen in der UML **Attribute**; bei ihnen kann der Typ oder der konkrete Wert weggelassen werden.
- Methodennamen können weggelassen werden.

Objektdiagramm, pragmatisch



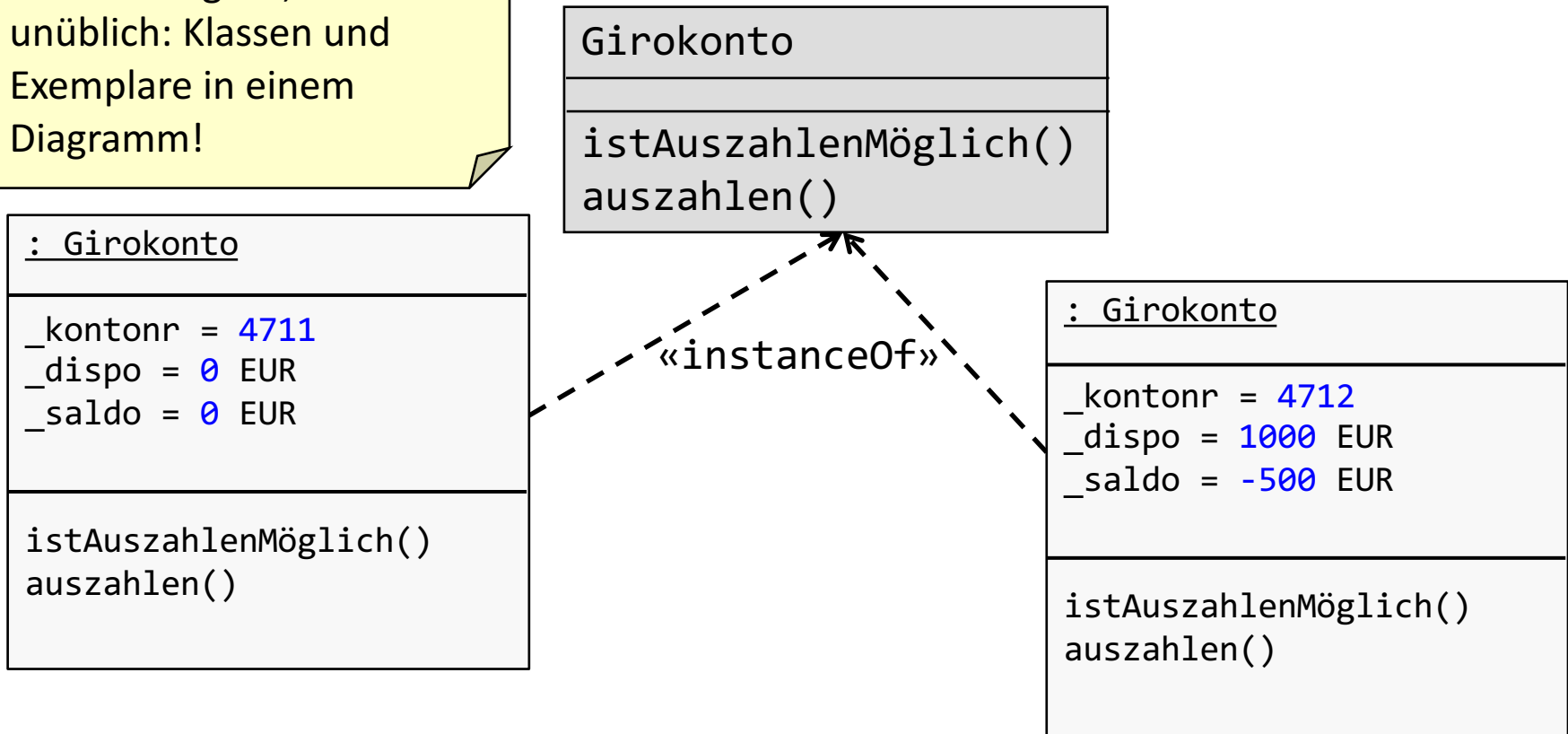
Objektdiagramme liefern Schnappschüsse

- Ein Objektdiagramm ist ein Schnappschuss eines laufenden Programms
- Es zeigt nur einen Ausschnitt des Objektgeflechts zur Laufzeit, um einen bestimmten Aspekt zu verdeutlichen



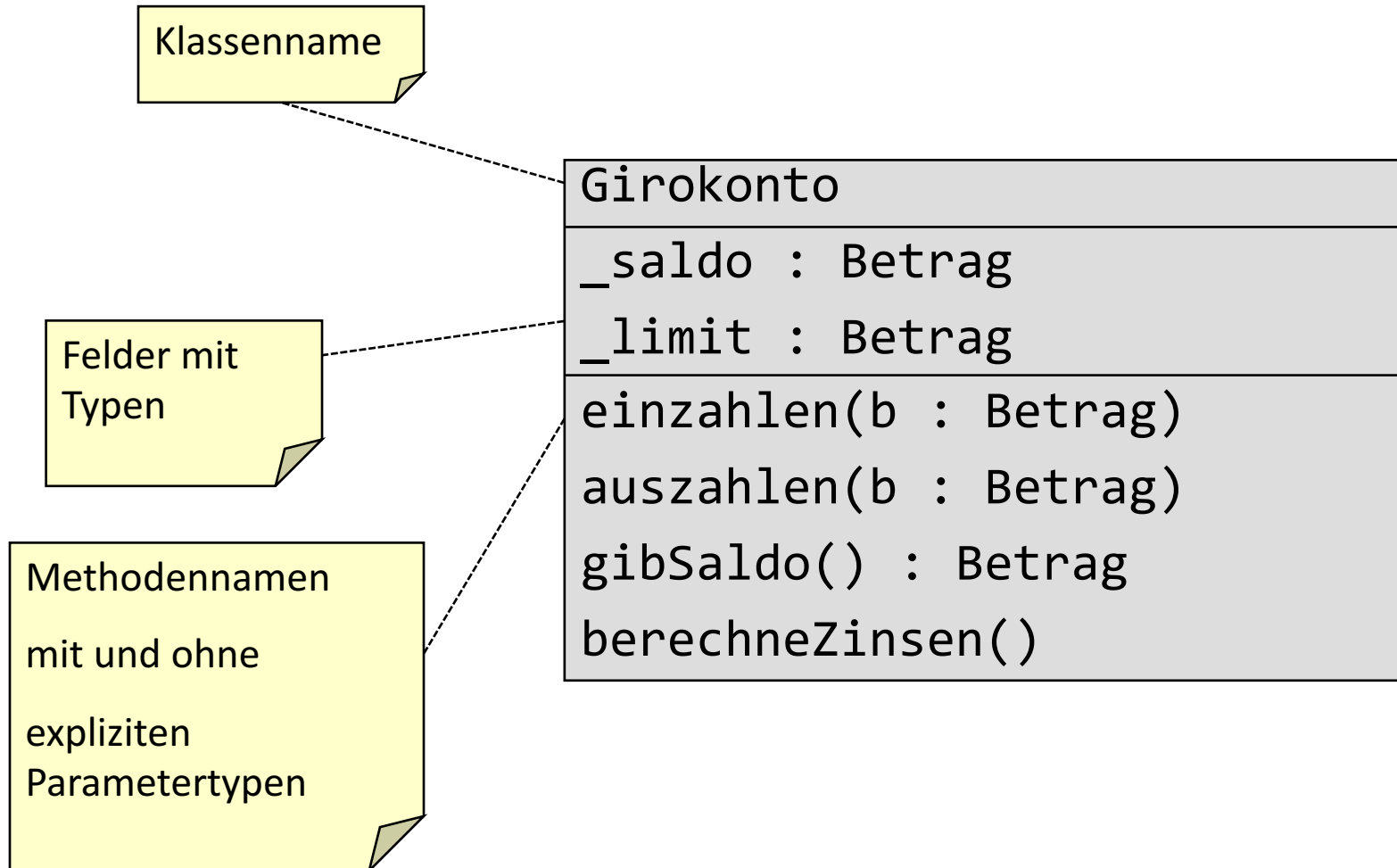
Objekte sind Exemplare von Klassen

In UML möglich, aber
unüblich: Klassen und
Exemplare in einem
Diagramm!

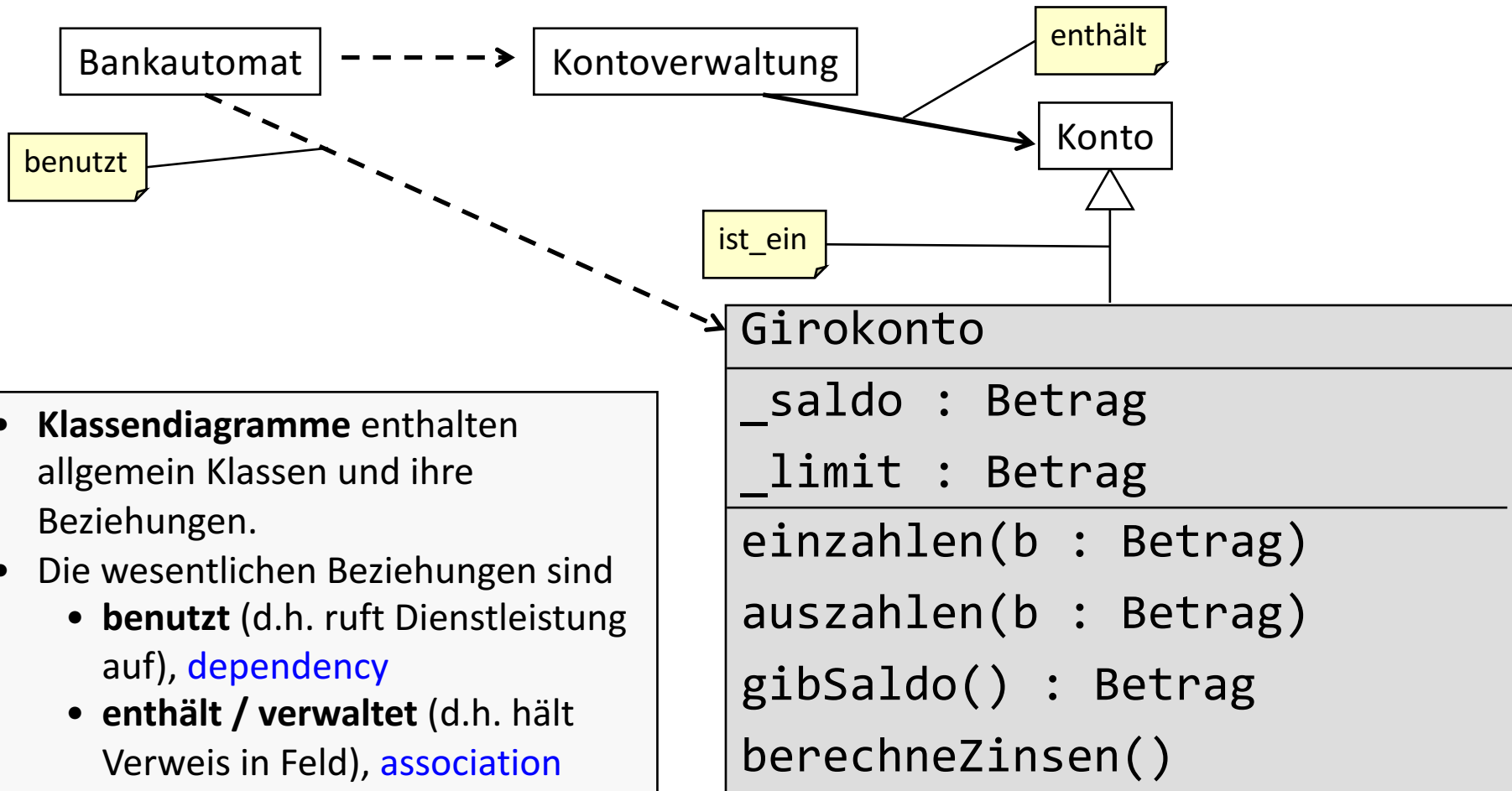


Die Klasse legt die Initialisierung, das Verhalten und die Struktur jedes Exemplars fest. Aber jedes Exemplar kann einen eigenen Zustand haben.

Klassendiagramme (1)

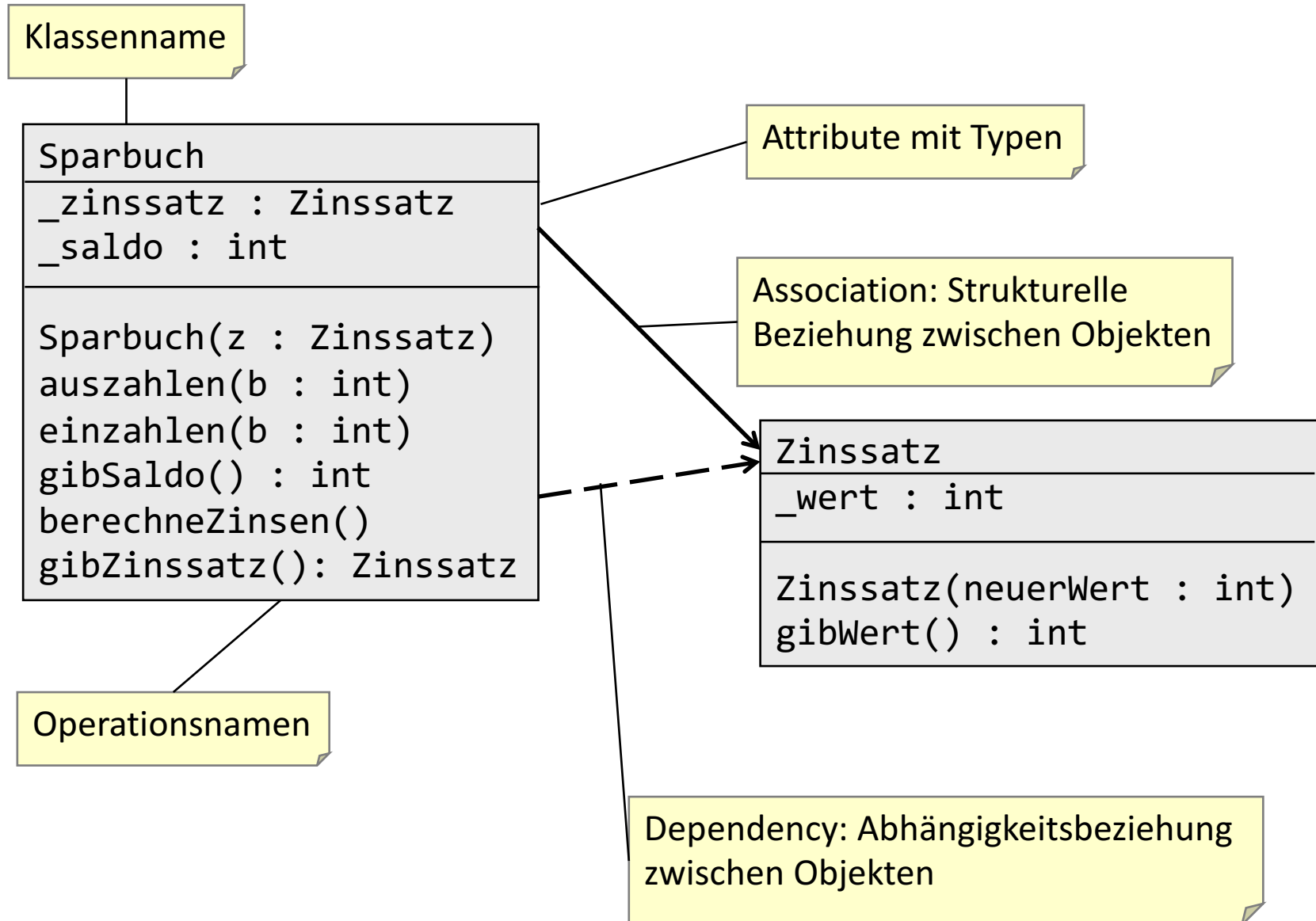


Klassendiagramme (2)



- **Klassendiagramme** enthalten allgemein Klassen und ihre Beziehungen.
- Die wesentlichen Beziehungen sind
 - **benutzt** (d.h. ruft Dienstleistung auf), **dependency**
 - **enthält / verwaltet** (d.h. hält Verweis in Feld), **association**
 - **ist_ein** (d.h. erbt von), **generalization**

Noch einmal: Ein UML-Klassendiagramm



Zusammenfassung

1

UML ist eine grafische Sprache für die Beschreibung von Software-Systemen.

2

UML bildet einen Quasi-Standard für objektorientierte Systeme und ist sehr umfangreich.

3

Die wichtigsten Diagrammtypen der UML die **Klassendiagramme** und die **Objektdiagramme**

4

Für den Einstieg in die UML ist das Buch „**UML Distilled**“ von Martin Fowler zu empfehlen (im Deutschen „UML konzentriert“)