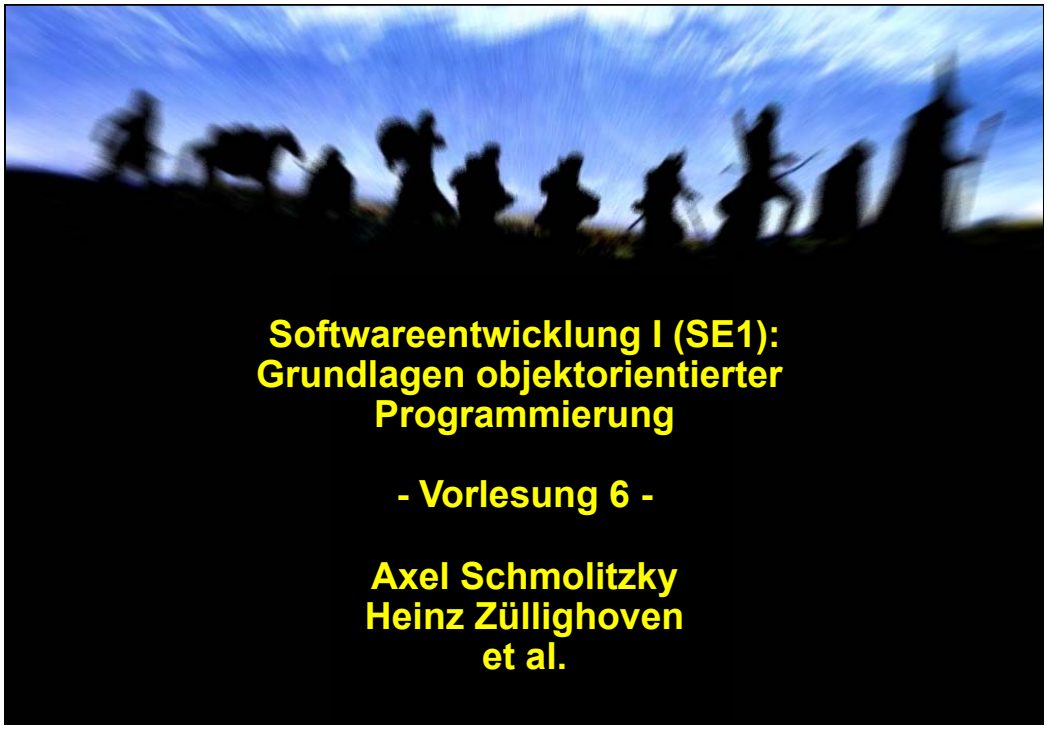
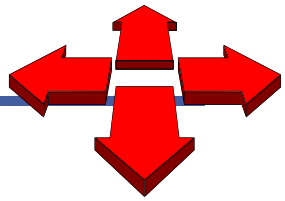
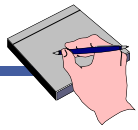


Das Typkonzept imperativer und objektorientierter Programmiersprachen



- Jede imperative und objektorientierte Programmiersprache besitzt **elementare Datentypen**, um numerische und logische Probleme lösen zu können.
- Zusätzlich definieren in objektorientierten Sprachen die **benutzerdefinierten Klassen weitere Typen**.
- Wir diskutieren Gemeinsamkeiten und Unterschiede dieser beiden Typfamilien.

Der Typbegriff (1. Definition)



Im Zusammenhang mit Programmiersprachen hat der Begriff **Typ** oder (oft auch) **Datentyp** eine zentrale Bedeutung:

- „Unter einem Datentyp versteht man die **Zusammenfassung** von **Wertebereichen** und **Operationen** zu einer Einheit."
  [Informatik-Duden]

Dies bedeutet:

- 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: **ganzahlig Addieren**, **ganzzahlig Multiplizieren**, …

Datentyp: `boolean`
Wertemenge: { **wahr**, **falsch** }
Operationen: **Und**, **Oder**, …

Typprüfung

- Wenn jeder Variablen (und Konstanten), jedem Literal und jedem Ausdruck in einem Programm ein fester, nicht änderbarer Typ zugeordnet ist, nennt man dies **statische Typisierung**.
- Als Folge der Typisierung kann für programmiersprachliche Ausdrücke geprüft werden, ob sie „korrekt typisiert“ sind, d.h. ob die einzelnen Komponenten einen passenden Typ besitzen, und ob dem Ausdruck insgesamt ein definierter Typ zugeordnet werden kann. Diese Prüfung nennt man **Typprüfung**.
- In **statisch typisierten Sprachen** (wie Java, C#, C++, Pascal, Eiffel) prüft der Compiler dies zur Übersetzungszeit.

***Smalltalk** ist eine dynamisch typisierte Programmiersprache, in der Variablen nicht mit einem Typ deklariert werden.*

Dynamisch typisierte Sprachen gestatten nur eine Laufzeitprüfung.

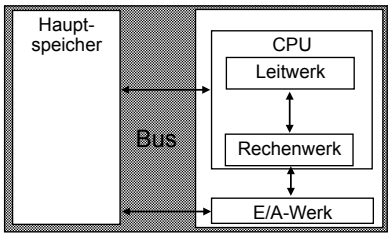
Beispiel: Die **Addition** ist als binäre Operation auf zwei `int` Zahlen definiert, nicht aber für eine `int` Zahl und einen Wahrheitswert.

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


Motivation für die Typisierung von Programmiersprachen: der Von-Neumann-Rechner

Wir erinnern uns:

- 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 (Daten-) Typen

NACKT:
Maschinenprogramme

Ein Hauptspeicher

Daten	01000011100011 11010001001010 11000000011111 11110000000111 10101010101010 10001100011101
Programm	01010011000011 01000101001110 10001000010000 10100000111010 10101011001110 00011111000101 10100010100101 10110000011111 10100010111000 10101010100101 11111000101010

LEICHT BEKLEIDET:
Imperative Sprachen seit Fortran

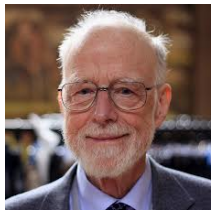
Programmiersprachen stellen elementare Typen zur Verfügung:

```
int
float
long
double
...
```

Ein numerischer Typ definiert, wie ein bestimmtes Bitmuster **interpretiert** werden soll. Das Bitmuster für den **int-Wert** `1` beispielsweise liefert einen völlig anderen Wert, wenn es als **Gleitkomma-zahl** interpretiert wird.

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.

© Hoare

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)

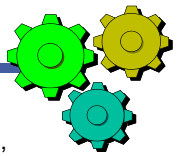
Ein Typ definiert Operationen…

Typinformation schützt und legt Semantik fest.

Die guten, alten 70er Jahre…

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

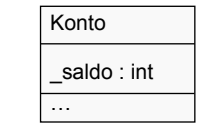
Von den elementaren Datentypen zu benutzerdefinierten Typen

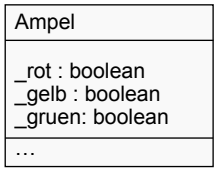


- Software dient zur **Verarbeitung von Anwendungsdaten**. Wir fragen, wie gut die **verfügbaren Datentypen** der verwendeten Programmiersprache zu den zu modellierenden **Gegenständen des Anwendungsbereichs** passen.
- Auf der Basis vorgegebener Datentypen sollen **anwendungsbezogene Datentypen** bereitgestellt werden.
- Zwei Lösungsansätze:
  - Eine große **Vielfalt vordeklarierter Datentypen** (wie in PL/I) soll möglichst viele Anwendungsfälle abdecken.
  - Ein kleiner Satz von elementaren Typen und flexible **Kombinationsmechanismen** (wie in Algol 68) sollen die anwendungsbezogene Definition **neuer Datentypen** erlauben.
- Der Ansatz, durch einen orthogonalen Entwurf von elementaren Typen und Kombinationsmechanismen flexible sog. **benutzerdefinierte Typen** zu ermöglichen, wird in fast allen modernen Sprachen verwendet.

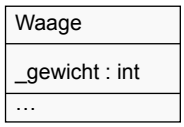
© Sebesta

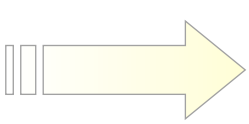
Auf dem Weg zu Objektgeflechten: Referenztypen

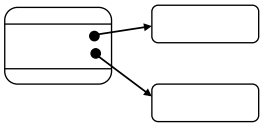




- Bisher haben wir im wesentlichen Klassen kennen gelernt, deren **Felder von elementaren Datentypen** waren.
- Bei Exemplaren dieser Klassen ist die Menge der möglichen Zustände durch die Deklarationen von Variablen und Konstanten im Klassentext zur Übersetzungszeit festgelegt.
- Diese Begrenzung wird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







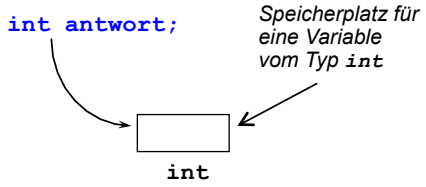
Referenzen allgemein

- Um ein Klienten-Objekt mit einem Dienstleister-Objekt zu verbinden, wird eine explizite **Referenz** (auch Verweis, Zeiger, Pointer) zwischen den Bezeichner im Quelltext des Klienten und das Dienstleister-Objekt geschaltet.
- Als Ergebnis der Erzeugung des Dienstleister-Objekts (jedes Objekt wird durch einen Konstruktoraufruf erzeugt) wird eine Referenz geliefert; diese Referenz ist quasi die „Adresse“ des neu erzeugten Objektes.
- Diese Referenz wird als ein **Wert** behandelt, der einer sogenannten **Referenzvariablen** im Klienten-Objekt zugewiesen werden kann.

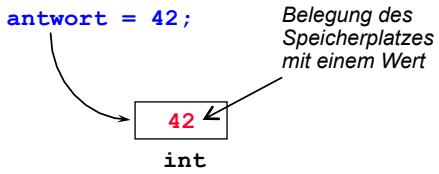
Variablen bisher: imperative Wertvariablen

- Wir haben als elementares imperatives Konzept die **Variable** kennengelernt. Kennzeichen sind:
  - 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.

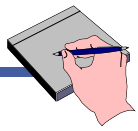
Deklaration:



Zuweisung:

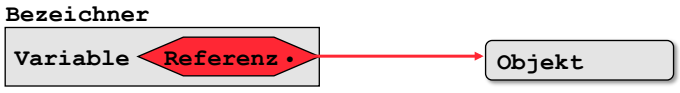


Imperative Referenzvariablen

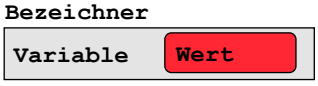


- Bei einer **Referenzvariablen** sind zwei Dinge zu unterscheiden:
  - Ihre Belegung mit einer **Referenz auf ein Objekt**,
  - das **referenzierte Objekt**.
- Gegenüber einer Wertvariablen wird also ein zusätzlicher Verweis (eine Indirektion) verwendet.

Referenzvariable



Wertvariable



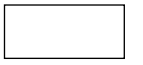
Referenzvariablen sind typisiert

- Auch **Referenzvariablen** haben einen Typ, einen **Referenztyp**. Jede Klasse in Java definiert einen Referenztyp; ihr Name kann als Typ in einer Variablendeklaration verwendet werden:

Deklaration:

```
Konto einKonto;
```

Speicherplatz für eine Variable vom Typ `Konto`

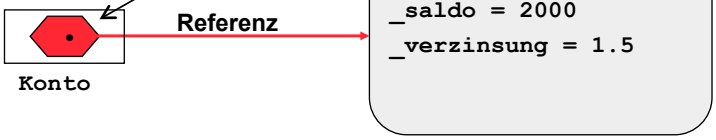


Konto

Zuweisung:

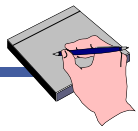
```
einKonto = new Konto();
```

Belegung des Speicherplatzes mit einer Referenz



Konto

Referenztypen sind Typen

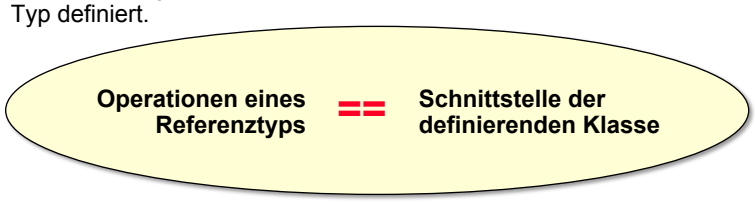


- Wie jeder Typ legt auch ein Referenztyp die Menge seiner Elemente und die möglichen Operationen auf den Elementen des Typs fest.
- Die **Elemente** eines Referenztyps sind die Exemplare der definierenden Klasse.
  - Da beliebig viele Exemplare einer Klasse erzeugt werden können, ist die Wertemenge eines Referenztyps normalerweise unbeschränkt.
- Die **Operationen**, die ein Referenztyp definiert, sind genau die Methoden, die an den Exemplaren der Klasse aufgerufen werden können.
  - Ein Compiler kann bei der Übersetzung anhand des Typs einer Referenzvariablen im Programmtext erkennen, welche Operationen (Methodenaufrufe) auf dieser Variablen zugelassen sind.

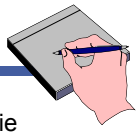
Datentyp: `Konto`
Wertemenge: Menge der Konto-Exemplare
Operationen: `einzahlen`, `auszahlen`, …

Schnittstelle und Typ

- Es besteht ein direkter Zusammenhang zwischen der **Schnittstelle** eines Objektes und seinem **Typ**. Wir wissen bereits von Stufe 1:
  - Die **öffentlichen 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 diese Referenzvariablen aufrufen können, sind genau die **öffentlichen Methoden** der Klasse, die den Typ definiert.



Referenzen in Java

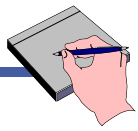


- **Alle Objekte** in Java werden über Referenzen verwendet. Auch die Übergabe eines Objektes als Parameter erfolgt lediglich als Übergabe des Wertes einer Referenz.
- Bei der **Zuweisung** einer Referenzvariablen wird die **Referenz kopiert**, nicht das referenzierte Objekt!
- Der **Gleichheitstest** mit dem Operator „**==**“ auf Referenzvariablen prüft die **Gleichheit der Referenzen** (zeigen sie auf dasselbe Objekt?), nicht der referenzierten Objekte.
- Eine Referenzvariable kann den besonderen Wert `null` haben (für „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).

Auf den Wert einer Referenz selbst kann in Java nicht zugegriffen werden. Das heißt, die Referenz kann nicht als Wert (Adresse) programmiersprachlich manipuliert werden (sog. **Zeigerarithmetik**).

Es gibt auch keine Referenzen auf Variablen o.ä.

Referenzen und boolesche Ausdrücke



- Boolesche Ausdrücke in Java haben eine nützliche Eigenschaft: Wenn der Wert eines Ausdruckes schon durch eine **Teilauswertung** feststeht, wird der Rest des Ausdrucks **nicht weiter ausgewertet** (JLS §15.23. u. §15.24.).
- Beispiele:

  `true || (energie < 10)` ⇒ immer **true** (unabhängig von `energie`)

  `false && (_saldo > 0)` ⇒ immer **false** (unabhängig von `_saldo`)

- Das ist sehr nützlich für Referenzvariablen, bei denen nicht garantiert ist, das sie eine Belegung ungleich **null** haben. Die Anweisung

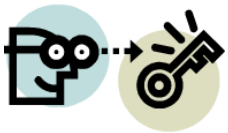
```
if (konto != null && konto.istGedeckt(betrag))
```

- verwendet einen Ausdruck, der für den Fall, dass die Variable `konto` **null** enthält, nicht weiter ausgewertet wird. Eine `NullPointerException` beim Methodenaufruf wird somit verhindert.
- Das ist nicht in allen Programmiersprachen so eindeutig geregelt wie in Java!

Im Java-Sprachgebrauch wird diese Form der Auswertung auch **short-circuit evaluation** genannt.

Wie kommt ein Klient an eine Referenz?

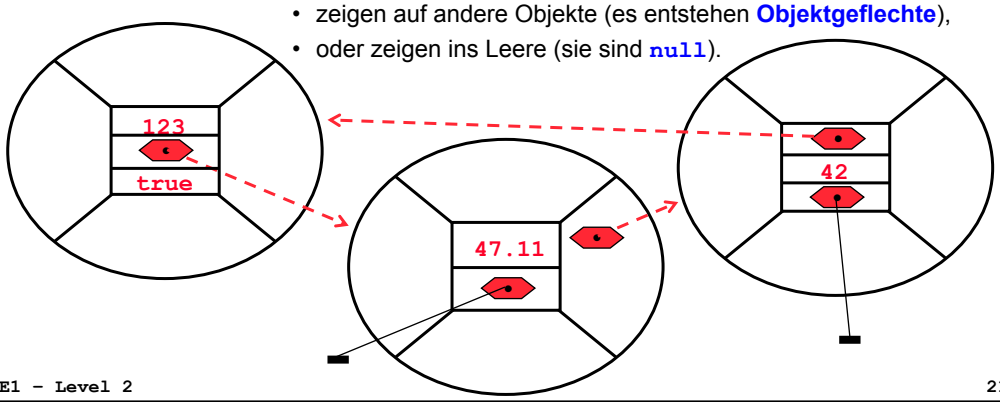
- Es gibt drei Möglichkeiten, wie ein Klient-Objekt vor einem Objektaufruf **innerhalb einer Methode** an eine gültige Referenz auf ein Dienstleister-Objekt kommt:
  - Das Klient-Objekt erzeugt das Dienstleister-Objekt innerhalb der Methode selbst.
  - Es erhält die Referenz auf den Dienstleister unmittelbar als Parameter der Methode.
  - Das Klient-Objekt hat bei seiner eigenen Erzeugung oder bei einem vorigen Methodenaufruf eine Referenz erhalten (oder selbst erzeugt), die es in einem Feld abgelegt hat; sie steht ihm dann in allen Methoden zur Verfügung.



Das allgemeine Objektmodell von Java

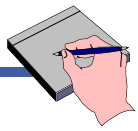
Objekte enthalten die in ihrer erzeugenden Klasse festgelegte Struktur von Feldern. Die jeweilige Belegung der Felder mit Werten und Referenzen definiert den Zustand eines Objekts.

- **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`).

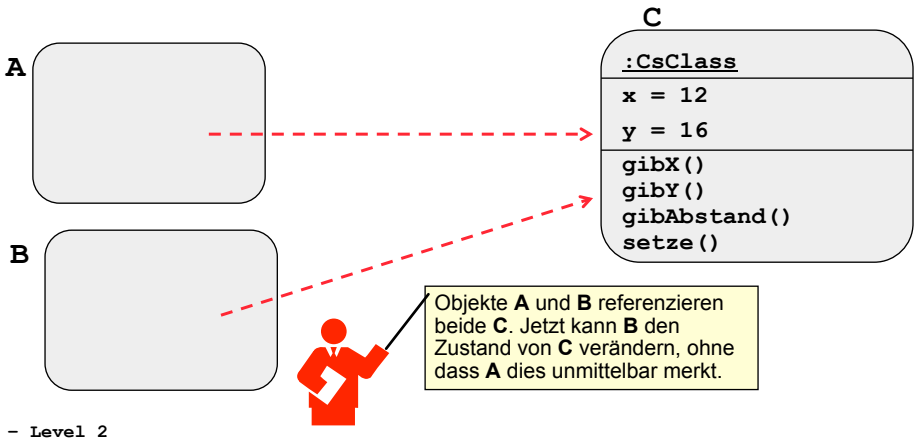


SE1 – Level 2 21

Das Alias-Problem

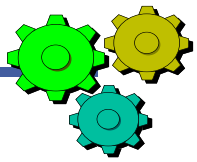


- Mehrere Referenzvariablen (in verschiedenen Objekten) können auf **dasselbe Objekt** verweisen. Damit ist lokal oft nicht entscheidbar, ob sich Veränderungen am Zustand eines referenzierten Objekts ergeben haben oder nicht. Dies ist das **Alias-Problem**, das bei allen Referenztypen auftritt.



SE1 – Level 2 22

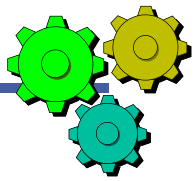
Alias-Problem: Wirklich Problem oder Chance?



- Es können **beliebig komplizierte Strukturen** über Referenzen konstruiert werden.
- Referenzen, die kreuz und quer in einem Softwaresystem Verbindungen herstellen, erschweren die **Wartbarkeit** und machen **formale Betrachtungen zur Korrektheit** erheblich schwieriger.

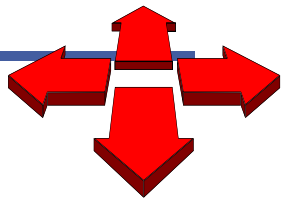
- Andererseits können mit Referenzen auch sehr **mächtige und effiziente Strukturen** gebaut werden.
- Wir sollten deshalb die Stärken und Schwächen von Referenzen **sehr gut kennen**, um in unseren Softwaresystemen **sinnvoll** mit ihnen umgehen zu können.

Zusammenfassung und Diskussion



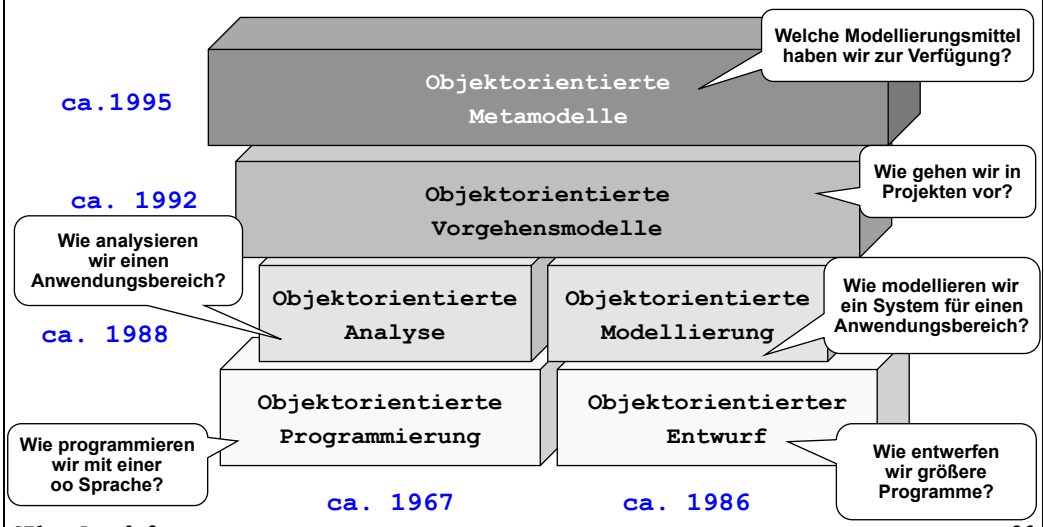
- Java unterscheidet fundamental zwei Typfamilien: **Werttypen** und **Referenztypen**.
- Die Menge der **Werttypen** ist **fest** in der Sprache **definiert** und kann nicht erweitert werden.
- Referenztypen werden durch Klassen definiert; es können **beliebig neue Referenztypen** definiert werden.
- Referenztypen sind das zentrale Mittel objektorientierter (und auch imperativer) Programmiersprachen, um **Objektgeflechte** zu konstruieren.
- **Referenzen** oder **Zeiger** sind in imperativen Programmiersprachen unterschiedlich realisiert. Teilweise kann der Wert einer Referenz selbst verändert werden (siehe *Zeigerarithmetik* in C und C++). Dadurch werden Programme **schwerer wartbar und beherrschbar**.
- Java ist in dieser Hinsicht eine **sichere** Sprache: Die Referenzen auf Objekte können nicht manipuliert/verändert werden.
- Java ist außerdem eine einfache Sprache: Alle Parameter werden **per Wert** übergeben, auch die Referenzen auf Objekte.

Die UML



- Was ist die UML?
- Wir stellen die wesentlichen Diagrammtypen der UML für die objektorientierte Programmierung vor:
  - Klassendiagramme
  - Objektdiagramme

Objektorientierte Aktivitäten



Die UML als Notation und Technik

- Bei Analyse, Modellierung und **Programmierung** benutzen wir eine einheitliche Notation - die **U**nified **M**odeling **L**anguage (**UML**).
- Die 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.

SE1 – Level 2 27

© RATIONAL SOFTWARE

Die Unified Modeling Language (UML)

- The UML is a language for
  - visualizing
  - specifying
  - constructing
  - documenting
- the artifacts of a software-intensive system

Die Diagrammtypen der UML

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

Unser Fokus in SE1

- **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)

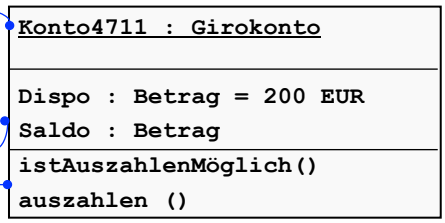
© *OMG-Unified Modeling Language, v2.0*

Objektdiagramm, formal korrekt

Objektbezeichner und Klassenname (unterstrichen)

Attribute mit Typen (und Werten)

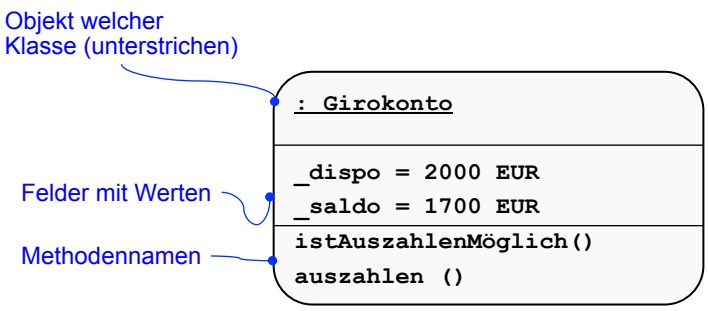
Methodennamen



- 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.

Die UML sieht bei Objektdiagrammen die Möglichkeit vor, Namen zu vergeben. **Objekte haben jedoch keine Namen**, insbesondere in Java nicht!

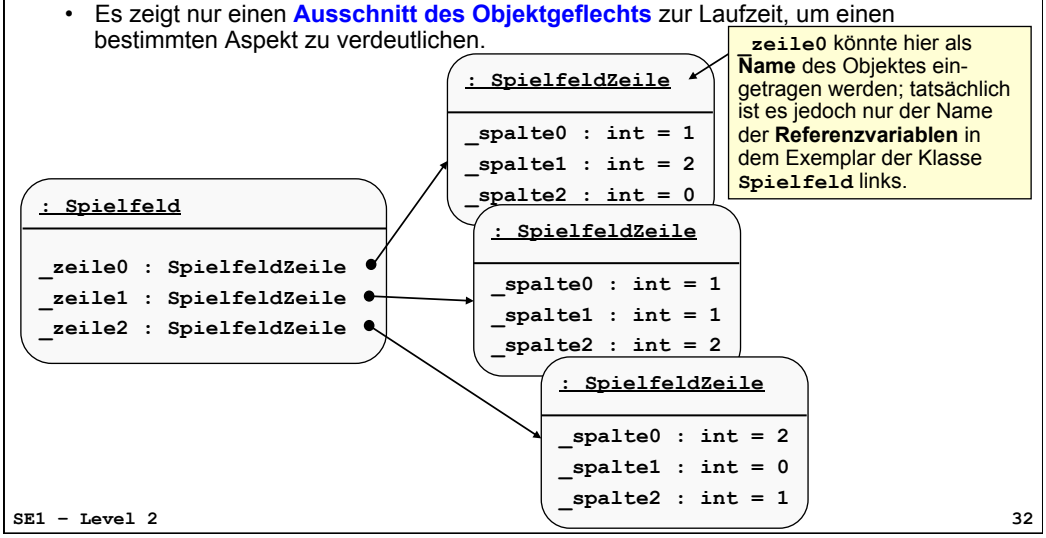
Objektdiagramm, pragmatisch



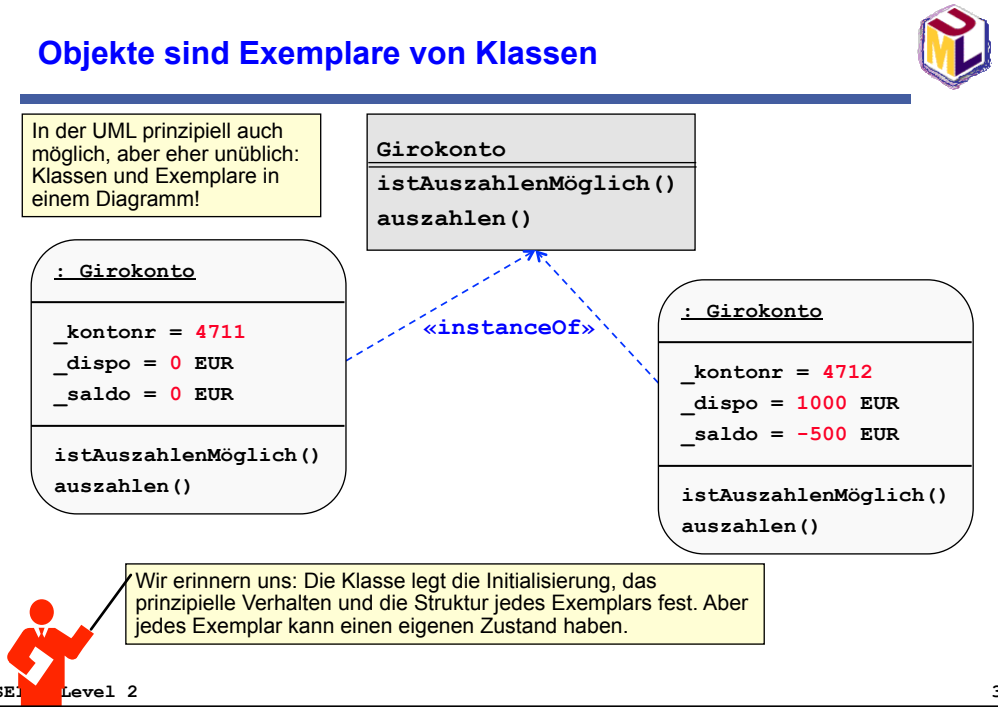
- ***Wir*** stellen Objekte als Rechtecke mit abgerundeten Ecken dar, damit wir sie optisch besser von Klassen (reine Rechtecke) unterscheiden können.
- Objektdiagramme sollten konkrete Werte für die Felder enthalten.

Objektdiagramme liefern Schnappschüsse

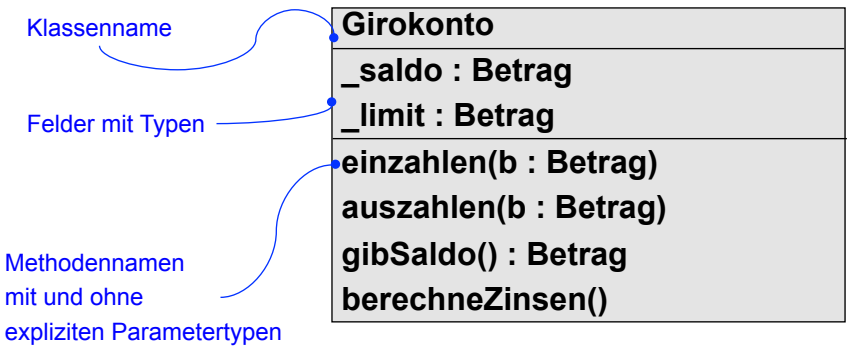
- Ein Objektdiagramm ist ein Schnappschuss eines laufenden Programms.
- Es zeigt nur einen **Ausschnitt des Objektgeflechts** zur Laufzeit, um einen bestimmten Aspekt zu verdeutlichen.



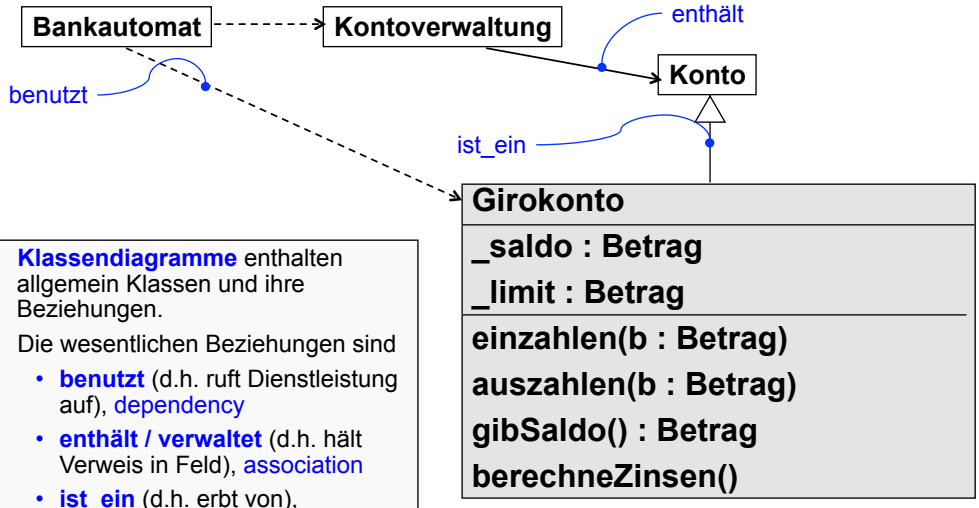
Objekte sind Exemplare von Klassen



Klassendiagramme (1)

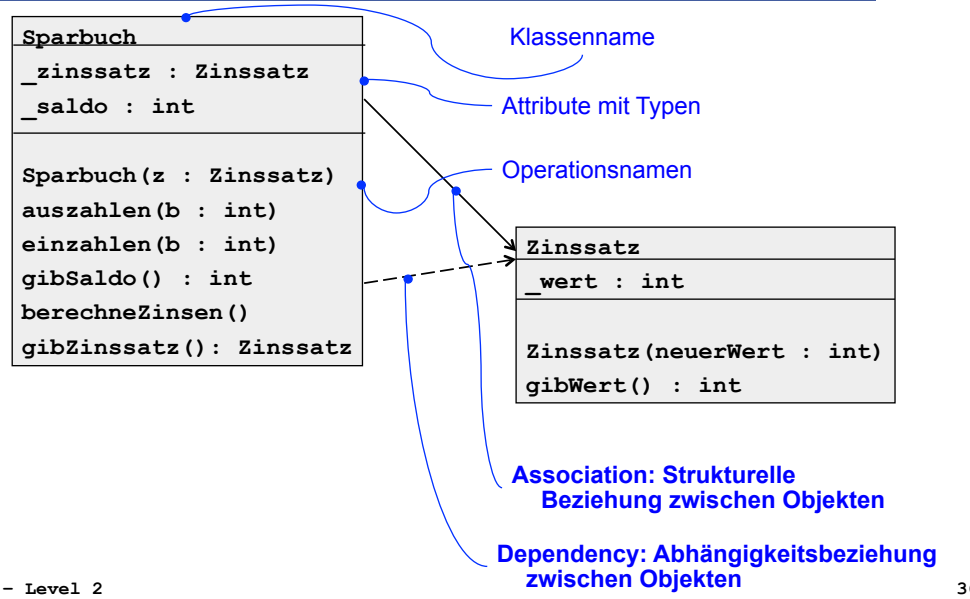


Klassendiagramme (2)

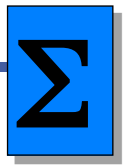


- **Klassendiagramme** enthalten allgemein Klassen und ihre Beziehungen.
- Die wesentlichen Beziehungen sind
  - **benutzt** (d.h. ruft Dienstleistung auf), dependency
  - **enthält / verwaltet** (d.h. hält Verweis in Feld), association
  - **ist_ein** (d.h. erbt von), generalization

Noch einmal: Ein UML-Klassendiagramm



Zusammenfassung



- Die **UML** ist eine grafische Sprache für die Beschreibung von Software-Systemen.
- Die UML bildet einen Quasi-Standard für objektorientierte Systeme und ist in der aktuellen **Version 2.1.2** sehr umfangreich.
- Für den Einstieg in die objektorientierte Programmierung sind die wichtigsten Diagrammtypen der UML die **Klassendiagramme** und die **Objektdiagramme**.
- Für den Einstieg in die UML ist das Buch „**UML Distilled**“ von Martin Fowler zu empfehlen (Addison-Wesley 2003, auch auf Deutsch als „UML konzentriert“ erhältlich).

