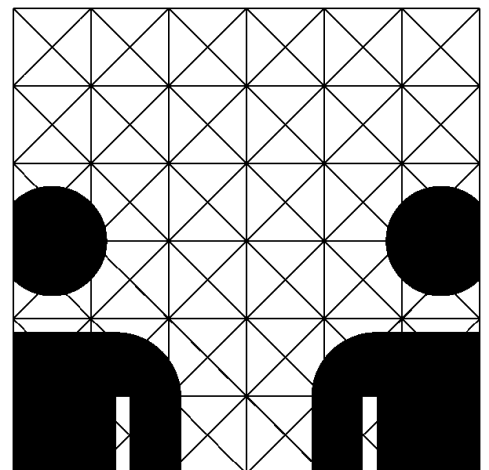


Prüfungsunterlagen  
zur Lehrveranstaltung



## Teil 2

Universität Hamburg  
MIN-Fakultät  
Department Informatik  
WS 2011 / 2012



# Softwareentwicklung I

## SE1

### Grundlagen objektorientierter Programmierung

Axel Schmolitzky  
Heinz Züllighoven  
et al.

## Teil 2

### Verzeichnis der Folien

1. **Das Typkonzept imperativer und objektorientierter Programmiersprachen**
2. Der Typbegriff (1. Definition)
3. Typprüfung
4. Motivation für die Typisierung von Programmiersprachen: der Von-Neumann-Rechner
5. Historisch: elementare (Daten-) Typen
6. Der „klassische“ Typbegriff
7. Der Typbegriff nach Hoare
8. Von den elementaren Datentypen zu benutzerdefinierten Typen
9. Auf dem Weg zu Objektgeflechten: Referenztypen
10. Referenzen allgemein
11. Variablen bisher: imperative Wertvariablen
12. Imperative Referenzvariablen
13. Referenzvariablen sind typisiert
14. Referenztypen sind Typen
15. Schnittstelle und Typ
16. Referenzen in Java
17. Wie kommt ein Klient an eine Referenz?
18. Das allgemeine Objektmodell von Java
19. Das Alias-Problem
20. Alias-Problem: Wirklich Problem oder Chance?
21. Zusammenfassung und Diskussion
22. **Die UML**
23. Objektorientierte Aktivitäten
24. Die UML als Notation und Technik
25. Die Unified Modeling Language
26. Die Diagrammtypen der UML
27. Objektdiagramm, formal korrekt
28. Objektdiagramm, pragmatisch
29. Objektdiagramme liefern Schnappschüsse
30. Objekte sind Exemplare von Klassen
31. Klassendiagramme (1)
32. Klassendiagramme
33. Noch einmal: Ein UML-Klassendiagramm
34. Zusammenfassung

### **35. Strukturierte Programmierung**

- 36. Kontrollfluss
- 37. Die Diskussion um Kontrollstrukturen
- 38. „Go To Statement Considered Harmful“
- 39. Edsger W. Dijkstra zum Go To Statement
- 40. Strukturierte Programmierung
- 41. Darstellungsmittel für Kontrollstrukturen: Flussdiagramme
- 42. Darstellungsmittel für Kontrollstrukturen: Struktogramme
- 43. Zusammengesetzte Anweisungen
- 44. Blöcke
- 45. Fallunterscheidungen: if-Anweisung
- 46. Diskussion: Geschachtelte if-Anweisung
- 47. SE1 Quelltextkonvention zu bedingten Anweisungen
- 48. Ein erstes Beispiel für die switch-Anweisung in Java
- 49. Ein weiteres Beispiel für die switch-Anweisung
- 50. Noch ein Beispiel für die switch-Anweisung
- 51. Und noch ein Beispiel für die switch-Anweisung
- 52. Selektion mit switch: Auswahlanweisung
- 53. Ergänzung der Java Level 1 Syntax um switch
- 54. Zusammengefasst: Auswahlanweisung
- 55. „Play it again, Sam“: Wiederholung durch Schleifen
- 56. Die Grundidee: 1x hinschreiben, mehrfach ausführen lassen
- 57. Die Struktur einer imperativen Schleife
- 58. Beispiele für Schleifenarten
- 59. Abweisende und nicht-abweisende Schleifen
- 60. Bedingte Schleifen
- 61. Aufpassen: Bedingte Schleifen an einem Beispiel
- 62. Zählschleifen
- 63. Realisierung von Schleifen in Java
- 64. Die for-Schleife in Java
- 65. Endlosschleifen

### **66. Statische und dynamische Eigenschaften**

- 67. Laufzeit und Übersetzungszeit
- 68. Laufzeit und Übersetzungszeit (II)
- 69. Veranschaulichungen des Unterschieds
- 70. Ein weiteres Beispiel für den Unterschied
- 71. Sichtbarkeitsbereich
- 72. Sichtbarkeitsbereich objektorientiert
- 73. Beispiel: Sichtbarkeitsbereiche
- 74. Verdecken von Bezeichnern
- 75. Klassischer Fehler: Versehentliches Überdecken
- 76. Sichtbarkeit der Elemente einer Klasse in Java
- 77. Lebensdauer
- 78. Zusammenfassung

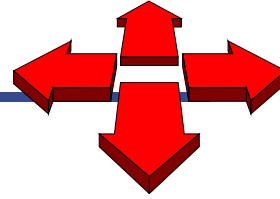
## **79. Rekursion**

- 80. Rekursion: ein erstes Beispiel
- 81. Rekursion
- 82. Rekursion: Grundstruktur
- 83. Rekursion: Der Kontrollfluss
- 84. Der Aufrufstack
- 85. Rekursion: Der Aufrufstack für das Beispiel
- 86. Rekursion: Das Beispiel als iteratives Programm
- 87. Rekursion: DAS Gegenbeispiel
- 88. Rekursion: Wir beginnen etwas zu ahnen...
- 89. Rekursion: Elegante Anwendungen
- 90. Rekursion: Stärken und Schwächen
- 91. Vereinfachtes Speichermodell von Sprachen mit dynamischen Objekten
- 92. Der Heap
- 93. Heap und Aufrufstack
- 94. Beispiel: Speichereinteilung in einem Unix-System
- 95. Java-Objektdiagramme: Schnappschüsse vom Heap
- 96. Der Garbage Collector in Java
- 97. Methoden und Zustandsfelder
- 98. Zusammenfassung

## **99. Strings und Reguläre Ausdrücke**

- 100. Zeichenketten in Programmiersprachen
- 101. Zeichenketten in Java: Literale, Konkatenation
- 102. Escape-Sequenzen in String-Literalen
- 103. Strings in Java: Unveränderlich!
- 104. Gleichheit von Strings in Java
- 105. Wie arbeitet eigentlich ein Compiler?
- 106. Syntaktische Grundelemente
- 107. Ein erstes Beispiel: Das Token „Bezeichner“
- 108. Bezeichner als regulärer Ausdruck
- 109. Reguläre Ausdrücke in Java
- 110. Weitere Beispiele für reguläre Ausdrücke in Java
- 111. Zeichenketten und reguläre Ausdrücke in Java
- 112. Formale Sprachen
- 113. Grammatiken für Sprachen
- 114. Reguläre Ausdrücke und reguläre Sprachen
- 115. Beispiel eines regulären Ausdrucks
- 116. Reguläre Ausdrücke in Java: fast wie in der Theorie
- 117. Kontextfreie und reguläre Sprachen
- 118. Der Unterschied liegt in der Mächtigkeit
- 119. Reguläre Ausdrücke sind effizient umsetzbar
- 120. Zusammenfassung

## 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.

SE1 – Level 2

1

## 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:  $\{-2^{31} \dots 2^{31}-1\}$   
Operationen: **Addieren**,  
**Subtrahieren**, **Multiplizieren**, ...



Datentyp: **boolean**  
Wertemenge:  $\{\text{wahr, falsch}\}$   
Operationen: **Und**, **Oder**, ...

SE1 – Level 2

2

## Level 2: Objekte benutzen Objekte

## 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!
```

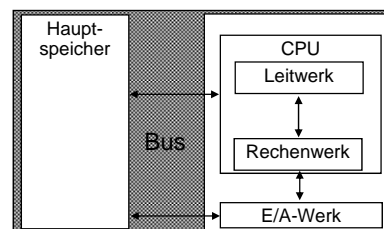
SE1 – Level 2

3

## 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.



SE1 – Level 2

4

## Historisch: elementare (Daten-) Typen

### NACKT: Maschinenprogramme

|                                      |                |
|--------------------------------------|----------------|
| Ein Hauptspeicher                    |                |
| D<br>a<br>t<br>e<br>n                | 01000011100011 |
|                                      | 11010001001010 |
|                                      | 11000000011111 |
|                                      | 11110000000111 |
|                                      | 10101010101010 |
|                                      | 10001100011101 |
| P<br>r<br>o<br>g<br>r<br>a<br>m<br>m | 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 **Gleitkommazahl** interpretiert wird.

## Der „klassische“ Typbegriff



- In imperativen Programmiersprachen bezieht sich der Typbegriff auf Werte. 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.

## Level 2: Objekte benutzen Objekte

## 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.

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.]

SE1 – Level 2

7

## 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 vordeklarerter 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.

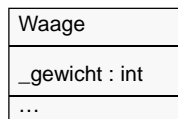
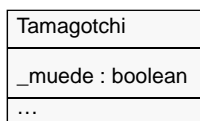
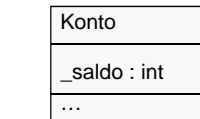
© Sebesta

SE1 – Level 2

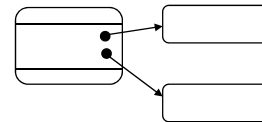
8



## 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**.



SE1 - Level 2

9

## 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 Konstruktoraufwurf 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.



SE1 - Level 2

10

## 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:

```
int antwort;
```

Speicherplatz für  
eine Variable  
vom Typ *int*



### Zuweisung:

```
antwort = 42;
```

Belegung des  
Speicherplatzes  
mit einem Wert



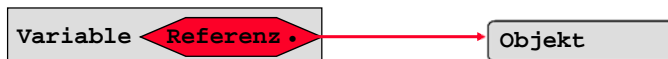
## Imperative Referenzvariablen

- 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

Bezeichner



### Wertvariable

Bezeichner



## 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;
```

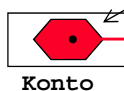
*Speicherplatz für  
eine Variable  
vom Typ **Konto***



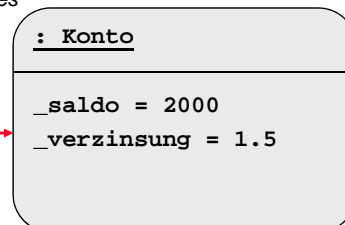
### Zuweisung:

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

*Belegung des  
Speicherplatzes  
mit einer  
Referenz*



**Referenz**



SE1 – Level 2

13

## 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**, ...*

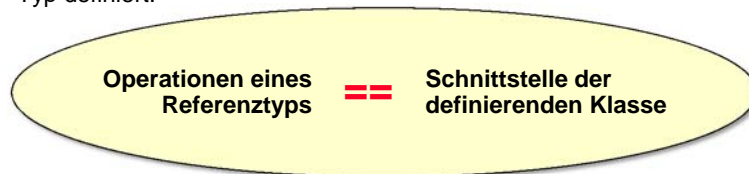
SE1 – Level 2

14

## Level 2: Objekte benutzen Objekte

## 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.



SE1 – Level 2

15

## 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.

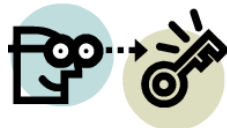
SE1 – Level 2

16

### 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.



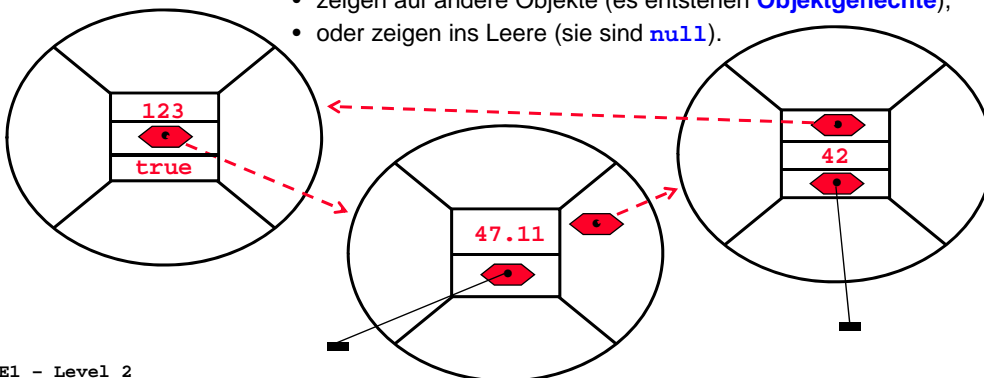
SE1 – Level 2

17

### 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**).



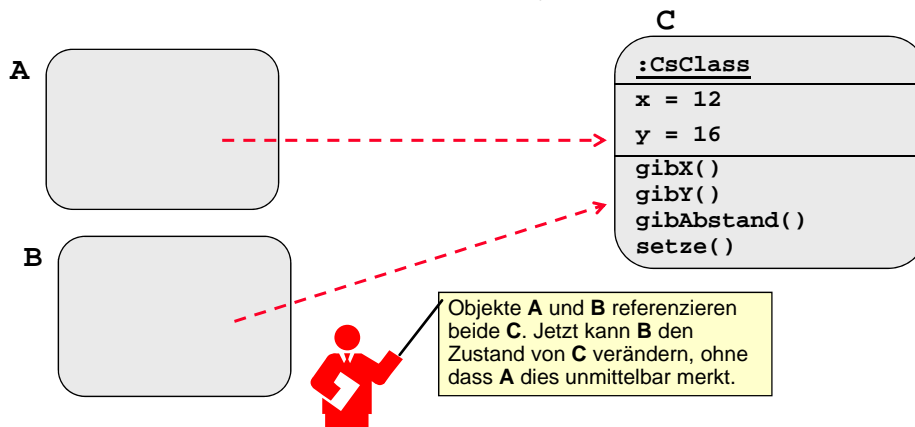
SE1 – Level 2

18

## Level 2: Objekte benutzen Objekte

## 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.

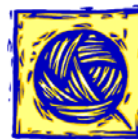


SE1 - Level 2

19

## Alias-Problem: Wirklich Problem oder Chance?

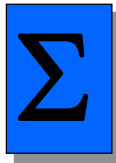
- Es können **beliebig komplizierte Strukturen** über Referenzen konstruiert werden.
- Referenzen, die kreuz und quer in einem Softwaresystem Verbindungen herstellen, erschweren die **Wartbarkeit** und machen **formale Betrachtungen zur Korrektheit** erheblich schwieriger.



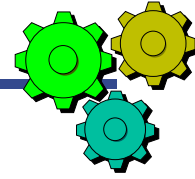
- Andererseits können mit Referenzen auch sehr **mächtige und effiziente Strukturen** gebaut werden.
- Wir sollten deshalb die Stärken und Schwächen von Referenzen **sehr gut kennen**, um in unseren Softwaresystemen **sinnvoll** mit ihnen umgehen zu können.

SE1 - Level 2

20



## Zusammenfassung und Diskussion

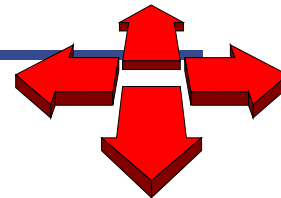


- Java unterscheidet fundamental zwei Typfamilien: **Werttypen** und **Referenztypen**.
- Die Menge der **Werttypen** ist **fest** in der Sprache **definiert** und kann nicht erweitert werden.
- Referenztypen werden durch Klassen definiert; es können **beliebig neue Referenztypen** definiert werden.
- 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.

SE1 – Level 2

21

## Die UML

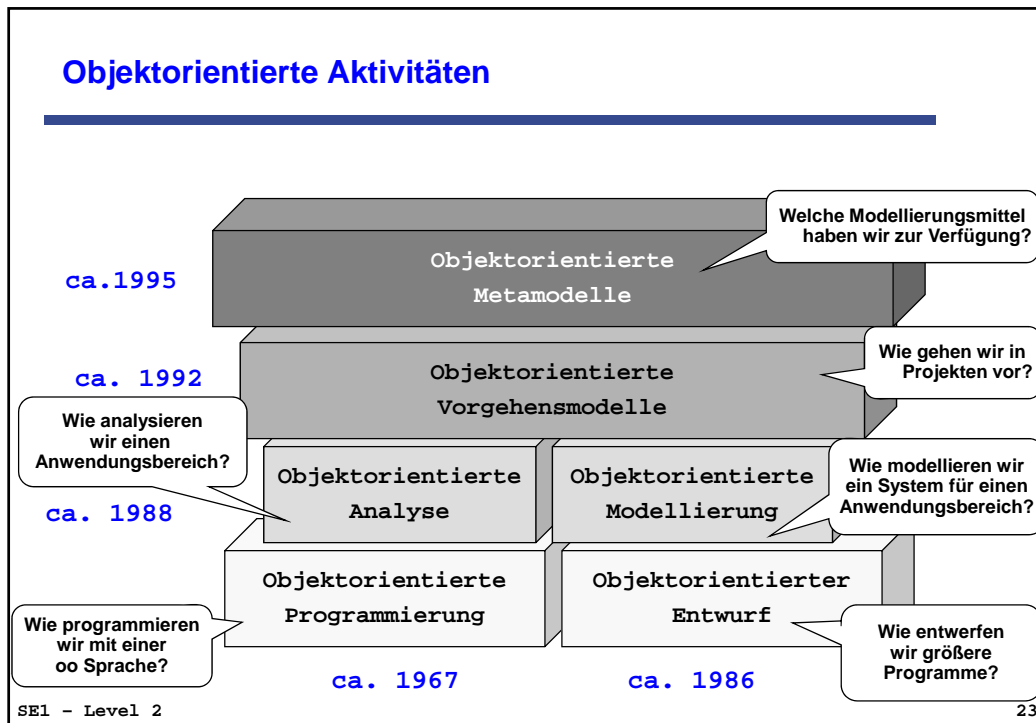


- Was ist die UML?
- Wir stellen die wesentlichen Diagrammtypen der UML für die objektorientierte Programmierung vor:
  - Klassendiagramme
  - Objektdiagramme


SE1 – Level 2

22

## Level 2: Objekte benutzen Objekte



### Die UML als Notation und Technik



- Bei Analyse, Modellierung und **Programmierung** benutzen wir eine einheitliche Notation - die **Unified Modeling Language (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.
- Derzeit aktuell ist die UML Version 2.1.2.
- 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 24





## 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



- **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)

Unser Fokus in SE1

© OMG-Unified Modeling Language, v2.0

SE1 - Level 2

26

### 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!

SE1 – Level 2

27

### Objektdiagramm, pragmatisch

Objekt welcher Klasse (unterstrichen)

Felder mit Werten

Methodennamen

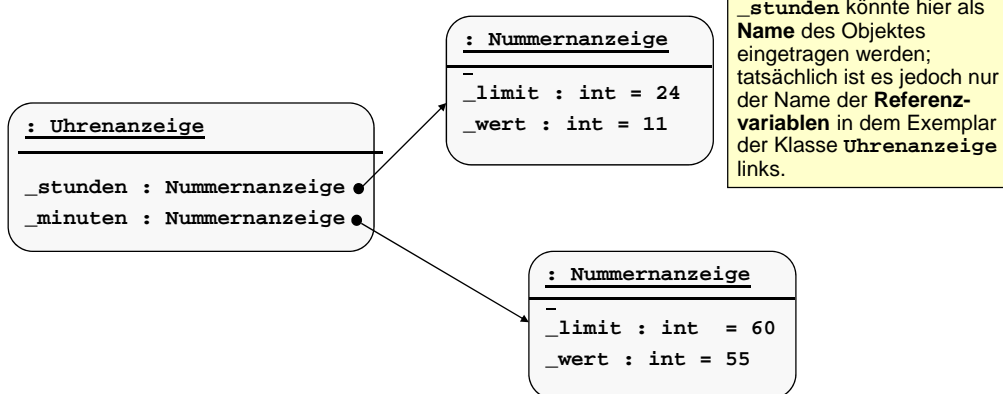
- **Wir** stellen Objekte als Rechtecke mit abgerundeten Ecken dar, damit wir sie optisch besser von Klassen (reine Rechtecke) unterscheiden können.
- Objektdiagramme sollten konkrete Werte für die Felder enthalten.

SE1 – Level 2

28

## Objektdiagramme liefern Schnappschüsse

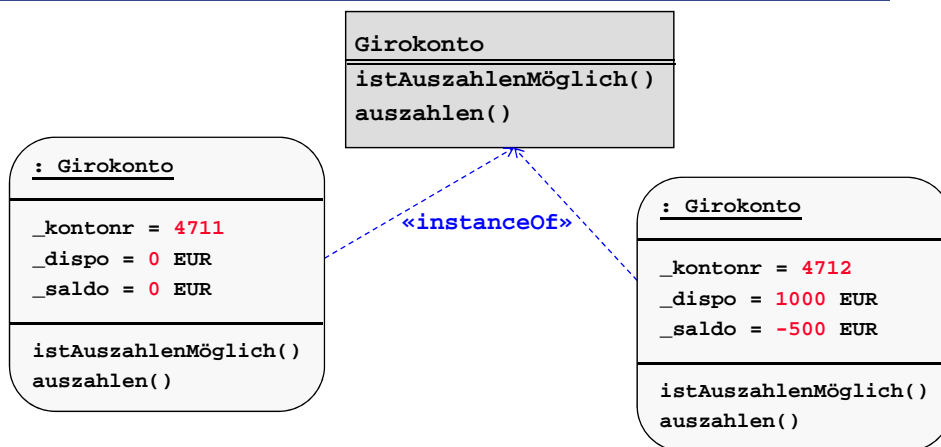
- Ein Objektdiagramm ist ein Schnappschuss eines laufenden Programms.
- Es zeigt nur einen Ausschnitt des Objektgeflechts zur Laufzeit, um einen bestimmten Aspekt zu verdeutlichen.



SE1 - Level 2

29

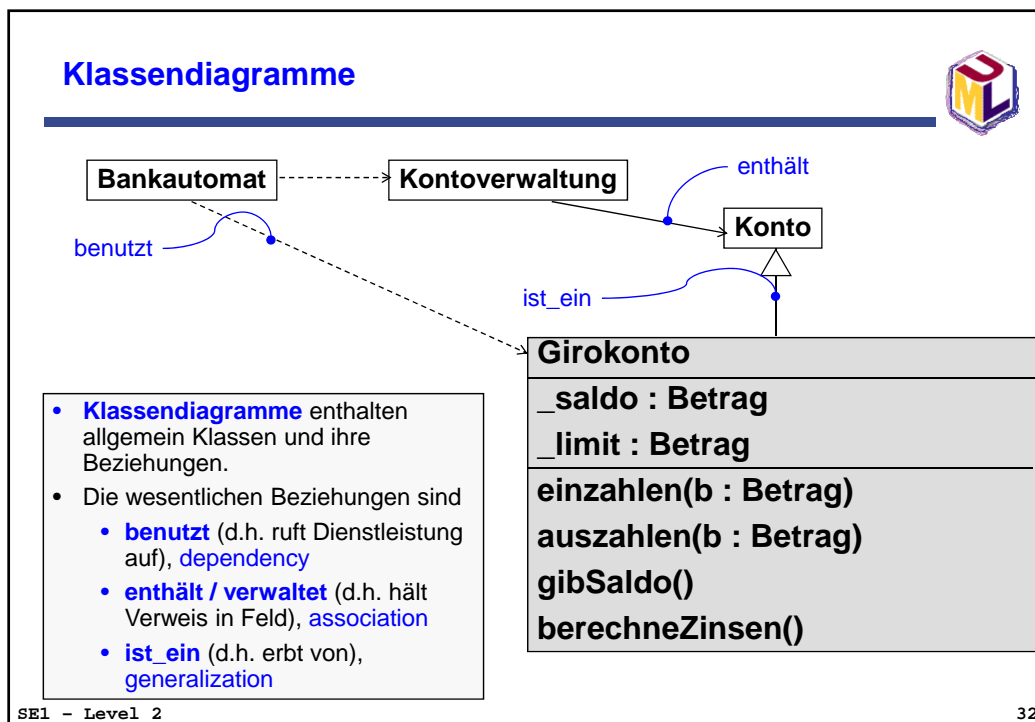
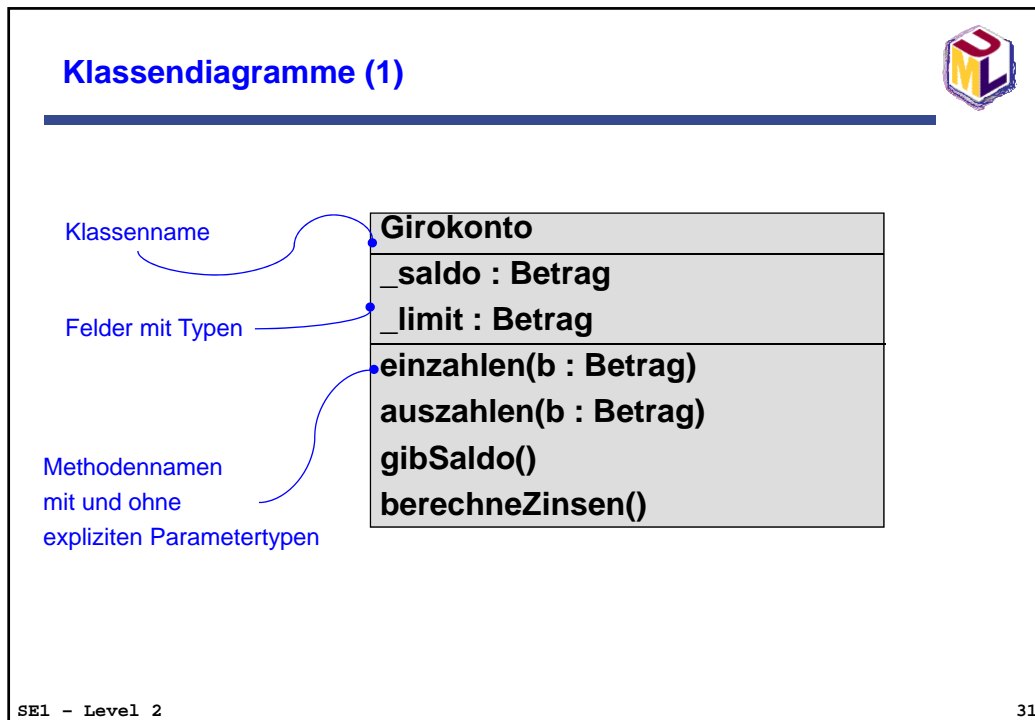
## Objekte sind Exemplare von Klassen

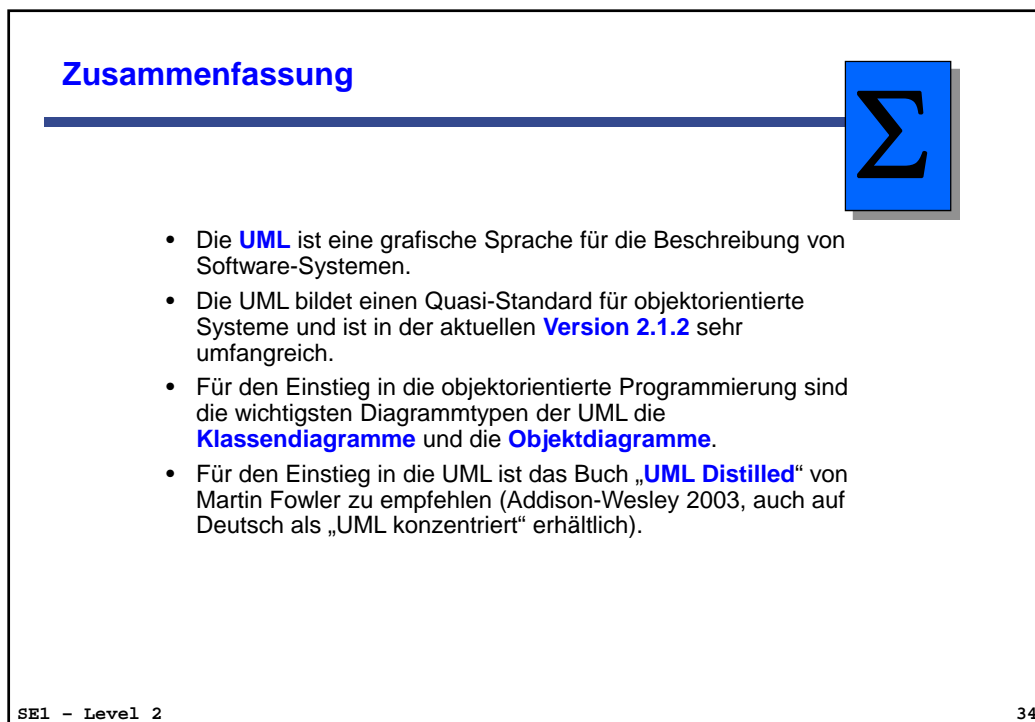
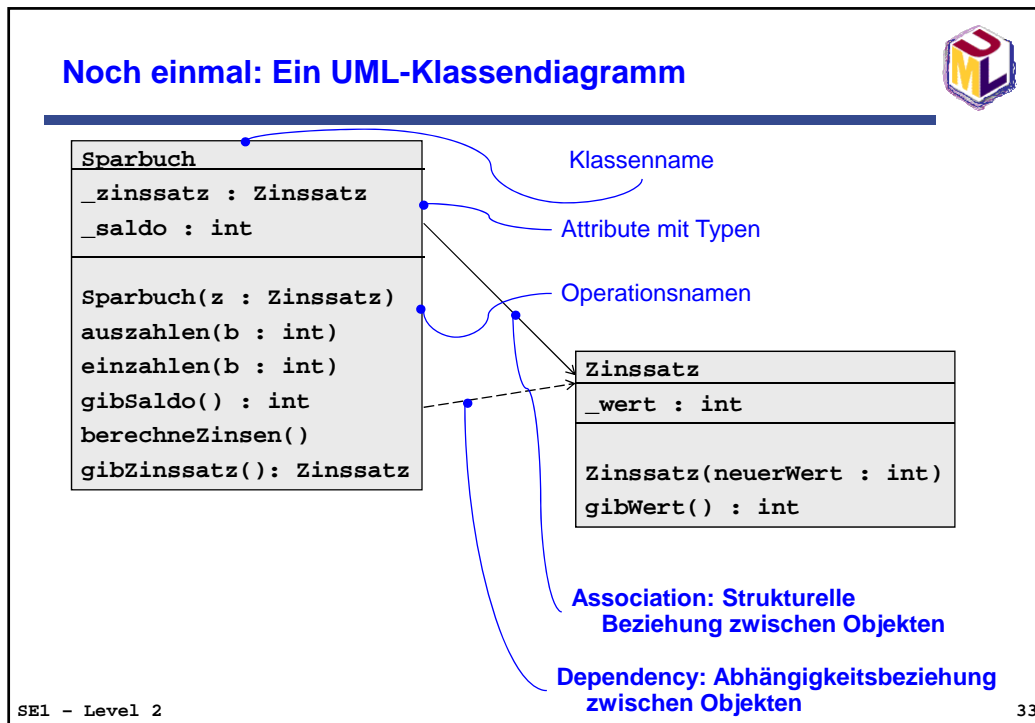


Wir erinnern uns: Die Klasse legt die Initialisierung, das prinzipielle Verhalten und die Struktur jedes Exemplars fest. Aber jedes Exemplar kann einen eigenen Zustand haben.

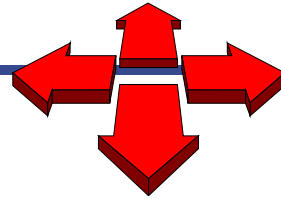
SE1 - Level 2

30





## Strukturierte Programmierung



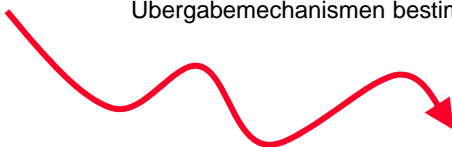
- Die **strukturierte Programmierung** beschränkt sich auf die **Kontrollstrukturen** Sequenz, Verzweigung und Wiederholung und verzichtet auf Sprungbefehle.
- Bei der Verzweigung unterscheiden wir **einfache Verzweigungen** und **Mehrfachverzweigungen**.
- Wiederholungen werden in der klassischen imperativen Programmierung mit Hilfe von **Schleifen-Mechanismen** realisiert.

SE1 – Level 2

35

## Kontrollfluss

- Das Verständnis des **Kontrollflusses** von (imperativen) Programmiersprachen ist eine wesentliche Voraussetzung für die Entwicklung korrekter Programme. Der Kontrollfluss bestimmt die **Reihenfolge**, in der Teile eines Programms ausgeführt werden.
- Der Kontrollfluss kann auf verschiedenen Ebenen betrachtet werden:
  - **Innerhalb einer Anweisung** (etwa im Ausdruck auf der rechten Seite einer Zuweisung) wird er durch die Bindungsstärke und Assoziativität der Operatoren bestimmt.
  - **Zwischen den Anweisungen** einer Methode wird er durch Kontrollstrukturen bestimmt.
  - **Zwischen den Methoden** wird er durch Methodenaufrufe und Übergabemechanismen bestimmt.



Wir erinnern uns: Wir betrachten in SE1 ausschließlich sequenzielle Abläufe eines Programms, d.h. zu einem Zeitpunkt wird nur jeweils eine Anweisung abgearbeitet.

SE1 – Level 2

36

## Die Diskussion um Kontrollstrukturen

- Von Mitte der 60er bis Mitte der 70er wurde in der Softwaretechnik (Informatik) viel über **Kontrollstrukturen** diskutiert.
- Bohm und Jacopini haben 1966 nachgewiesen, dass alle **Algorithmen**, die in **Flußdiagrammen** ausgedrückt werden können, **D-Diagramme** sind und durch entsprechende **Kontrollanweisungen** implementierbar sind.
- Kontrollstrukturen sollen **einen Einstieg** und **einen Ausstieg** (engl.: single entry, single exit) besitzen.

### D-Diagramme [Dijkstra]:

- 1 Eine einfache Aktion ist ein D-Diagramm.
- 2 Sind **A** und **B** D-Diagramme, dann sind auch
 

```
A; B,
if c then A end,
if c then A else B end,
while c do A end
```

 D-Diagramme.
- 3 Nichts sonst ist ein D-Diagramm.

SE1 – Level 2

© Pomberger in Rechenberg, Pomberger

37

## „Go To Statement Considered Harmful“

- Obwohl die **unbedingte Verzweigung (Goto)** ausreicht, alle anderen Kontrollstrukturen nachzubilden, führt ihre uneingeschränkte Verwendung zu unlesbaren und unzuverlässigen Programmen.
- **Hauptgrund:**
  - Durch Goto kann im Ablauf jede beliebige Reihenfolge von Anweisungen unabhängig von ihrer textlichen Anordnung erreicht werden.
- In einem berühmten Leserbrief ("Go To statement considered harmful", CACM, 1968, Vol.11, No.3, pp.147-148) schreibt E.W. **Dijkstra**:
  - *"The goto statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program."*
- Dies hat die **Goto-Debatte** entzündet, die zwar zur softwaretechnischen Ablehnung des uneingeschränkten Goto geführt hat, aber nur wenige Programmiersprachen haben völlig auf dieses Konstrukt verzichtet.



Java bietet keine Goto-Anweisung; allerdings ist **goto** als Schlüsselwort reserviert...

SE1 – Level 2

© Sebesta

38

## Edsger W. Dijkstra zum Go To Statement



“For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code).”

...

Reprinted from Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148. Copyright © 1968, Association for Computing Machinery, Inc.

<http://www.acm.org/classics/oct95/>

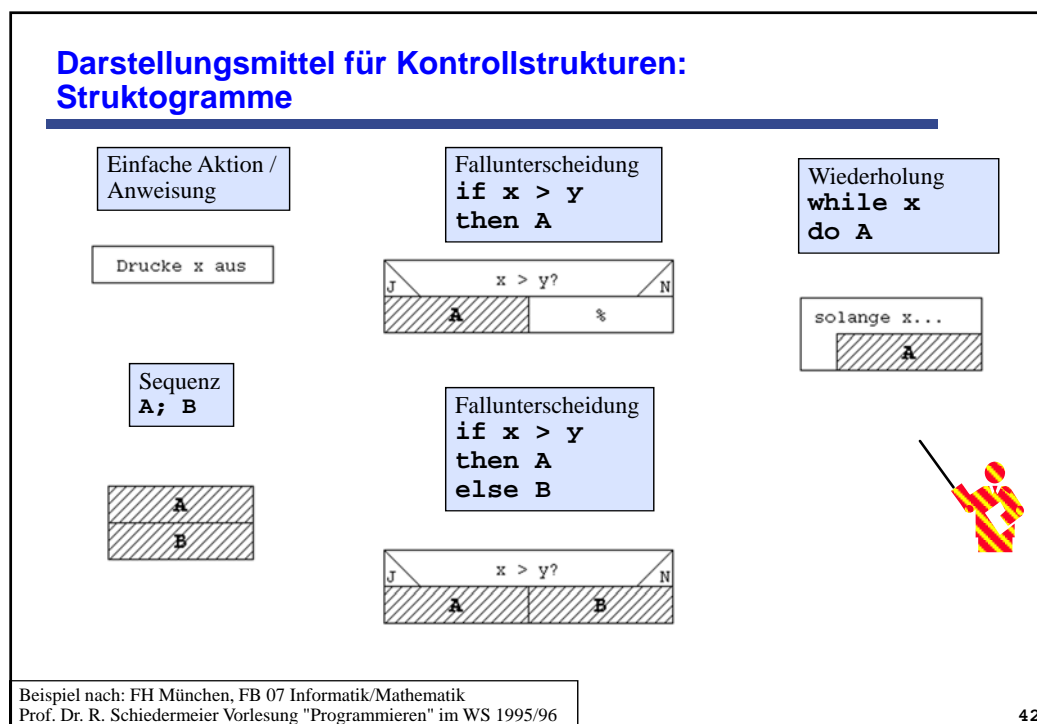
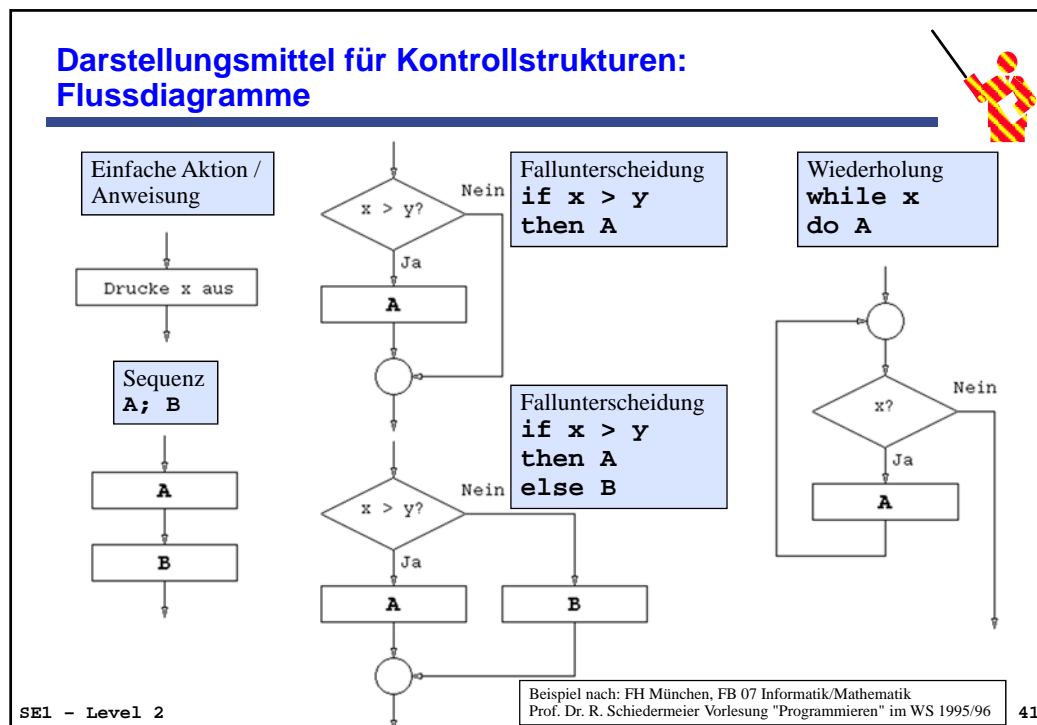
## Strukturierte Programmierung



- Die heute klassischen Kontrollstrukturen der imperativen Programmierung sind **Sequenz**, **Auswahl**, **Wiederholung**.
- Sprachen mit diesen Kontrollstrukturen heißen auch **strukturierte Sprachen**, was die enge Beziehung zur **strukturierten Programmierung** verdeutlicht.
- Der Kern der strukturierten Programmierung ist die **Beschränkung** der Kontrollstrukturen in Programmen auf Sequenz, Auswahl und beschränkte Schleifenkonstrukte.
- Dazu kommen die Verwendung von Prozeduren und (abstrakten) Datentypen.







## Zusammengesetzte Anweisungen



- **Zusammengesetzte Anweisungen** (engl.: compound statements) fassen eine Folge von Anweisungen zu einer einzigen Anweisung zusammen, indem sie syntaktisch klammern (etwa durch **begin** und **end**).
- Da zusammengesetzte Anweisungen programmiersprachlich als **eine einzige Anweisung** gelten, sind sie in Kontrollstrukturen sehr nützlich.

Zusammengesetzte Anweisung in Algol 60:

```
begin
  statement_1;
  ...
  statement_n
end
```



## Blöcke



- **Blöcke** sind zusammengesetzte Anweisungen, die um **lokale Variablen** ergänzt werden.
- Blöcke bilden einen eigenen Sichtbarkeitsbereich (kommt noch).
- Auch ein Block ist syntaktisch geklammert; in Java mit geschweiften Klammern, in Algol-artigen Sprachen auch durch **begin ... end**
- ALGOL-60 war die erste Sprache mit Blöcken.
- Programmiersprachen, die Blöcke kennen, heißen auch **blockstrukturiert**.

```
{
  int a, b;
  ...
  if (a < b)
  {
    int temp; // Block-lokale Variable
    temp = a;
    a = b;
    b = temp;
  }
}
```



Level 1 Syntax

Block:  
 { { BlockStatement } }  
 BlockStatement:  
 LocalVariableDeclarationStatement  
 Statement

## Fallunterscheidungen: if-Anweisung

- Die programmiersprachliche Realisierung von **Fallunterscheidungen** heißt auch **Verzweigung** oder **bedingte Anweisung** (engl.: conditional statement).
- Üblich sind **Zweiweg-** und **Mehrweg-Verzweigungen**.
- Die** Standardform der Zweiweg-Verzweigung ist die **if-Anweisung**. Für Java ist sie syntaktisch folgendermaßen definiert (siehe Java Level 1 Syntax):

**Statement:**  
`if ( Expression ) Statement [ else Statement ]`

The expression must have type boolean, or a compile-time error occurs.



```
if ( a < b )
    min = a;
else
    min = b;
```

## Diskussion: Geschachtelte if-Anweisung

Beispiel 1:

```
if (sum == 0)
    if (count == 0)
        result = 1;
else
    result = 0;
```



- Vorsicht bei geschachtelten if-Anweisungen: ein `else`-Zweig bezieht sich immer auf die letzte if-Anweisung ohne `else`-Zweig.
- Explizite Klammerung hilft, Fehler zu vermeiden (Beispiel 3)!

Siehe dazu auch Punkt 4.7 der Quelltextkonventionen!

Beispiel 2: 

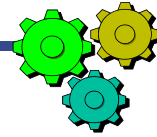
```
if (sum == 0)
    if (count == 0)
        result = 1;
else
    result = 0;
```

- Beispiel 1 erzeugt durch sein Layout einen falschen Eindruck; Beispiel 2 ist korrekt eingerückt.
- Das hier dargestellte Problem heißt „dangling else“ („else“ ohne Bezug).

Beispiel 3: 

```
if (sum == 0)
{
    if (count == 0)
    {
        result = 1;
    }
}
else
{
    result = 0;
}
```

## SE1 Quelltextkonvention zu bedingten Anweisungen



- Bei gewöhnlichen Auswahlanweisungen – **if**-Anweisungen – sollte ein ggf. vorhandenes **else** stets **in einer eigenen Zeile** stehen.
- Allg. gilt: Verwende **immer Blockklammern** bei **if**-Anweisungen, diese sind zwar bei genau einer auszuführenden Anweisung nicht notwendig, erhöhen jedoch die Lesbarkeit des Textes.
- Außerdem ist dies **robuster** gegenüber Änderungen: Wenn beim Eintreten der Bedingung nicht nur eine, sondern auch eine weitere Anweisung ausgeführt werden soll, kann die zweite leicht hinzugefügt werden, ohne dass Klammern eingefügt werden müssen.

Beispiel 3: 

```
if (sum == 0)
{
    if (count == 0)
    {
        result = 1;
    }
}
else
{
    result = 0;
}
```

SE1 – Level 2

47

## Ein erstes Beispiel für die switch-Anweisung in Java

```
switch (gedruckteTaste)
{
    case 'a': ← case label
        bewegeSpielerNachLinks();
        break;

    case 'd':
        bewegeSpielerNachRechts();
        break;

    case 'w':
        bewegeSpielerNachOben();
        break;

    case 's':
        bewegeSpielerNachUnten();
        break;

    case ' ':
        feuereRaketeAb();
}
```

- Abhängig von einer auf der Tastatur gedrückten Taste soll in einem Spiel eine von mehreren Prozeduren aufgerufen werden.

- Als case-Label verwenden wir ausschließlich **Konstanten**, häufig vom Typ **int** oder **char**.

Was passiert, wenn

- die **break**-Anweisung fehlt?



SE1 – Level 2

48

### Ein weiteres Beispiel für die switch-Anweisung

```
char buchstabe = liesZeichenVonTastatur();
boolean istVokal = false;
switch (buchstabe)
{
case 'a':
case 'A':
case 'e':
case 'E':
case 'i':
case 'I':
case 'o':
case 'O':
case 'u':
case 'U': istVokal = true;
}
System.out.print(buchstabe + " ist ");
if (!istVokal)
{
    System.out.print('k');
}
System.out.println("ein Vokal.");
```



- Für eine auf der Tastatur gedrückte Taste soll ausgegeben werden, ob sie einen Vokal liefert oder nicht.

Was passiert, wenn

- zwei **case**-Label denselben Wert haben?



SE1 – Level 2

49

### Noch ein Beispiel für die switch-Anweisung

```
int monat = liesMonatszahlVomBenutzer();
int tage;
switch (monat)
{
case 1:
case 3:
case 5:
case 7:
case 8:
case 10:
case 12:
    tage = 31;
    break;

case 4:
case 6:
case 9:
case 11:
    tage = 30;
    break;

case 2:
    tage = 28;
    break;

default:
    tage = -1;
}
System.out.println("Der Monat " + monat + " hat " + tage + " Tage.");
```



- Für einen Monat im Jahr, den der Benutzer durch eine ganze Zahl benennen soll, soll ausgegeben werden, wie viele Tage er hat.

Links angedeutet ein Beispiel für den **Kontrollfluss**: Wir geben als Benutzer eine **7** ein.

Was passiert, wenn

- keiner der Fälle zutrifft und die **default**-Anweisung fehlt?



SE1

50

## Und noch ein Beispiel für die switch-Anweisung

```
int zahl = liesZehnerpotenzVomBenutzer();
int exponent = 0;
switch (zahl)
{
case 1000000000: ++exponent;
case 100000000: ++exponent;
case 10000000: ++exponent;
case 1000000: ++exponent;
case 100000: ++exponent;
case 10000: ++exponent;
case 1000: ++exponent;
case 100: ++exponent;
case 10: ++exponent;
case 1: System.out.println(zahl + " = 10^" + exponent); break;
default: System.out.println(zahl + " ist keine Zehnerpotenz!");
}
```

- Der Benutzer soll eine ganze Zahl eingeben, die eine Zehnerpotenz ist. Für eine korrekt eingegebene Zehnerpotenz soll der passende Exponent ausgegeben werden, für alle anderen Zahlen eine Meldung, dass es keine Zehnerpotenz ist.



SE1 – Level 2

51

## Selektion mit switch: Auswahlanweisung



- Allgemeines Schema in Java:

```
switch (expression)
{
case value_1: statements_1;
case value_2: statements_2;
               break;
...
default: default_statements;
}
```



Seit Java 1.5 kann auch über die Elemente eines Aufzählungstyps „geswitcht“ werden, seit Java 1.7 auch über Strings. Dies ist hier bewusst ausgelassen.

- Die **Auswahlanweisung** (engl.: case statement, switch statement) ist die übliche Form einer **Mehrweg-Verzweigung**. Aufgrund eines Ausdrucks können mehrere **Fälle** unterschieden und behandelt werden.
- Gegenüber der if-Anweisung ist einiges anders:
  - **Mehrere Ausdruckstypen** können die Auswahl kontrollieren.
  - Es können **einer oder mehrere Fälle** ausgewählt werden.
  - Statt eines (eindeutigen) else-Falles gibt es einen **Standardfall** für alle nicht explizit benannten Fälle.

SE1 – L

52

## Ergänzung der Java Level 1 Syntax um switch

```
Statement:  
  Block  
  if ( Expression ) Statement [ else Statement ]  
  switch ( Expression ) { { SwitchBlockStatementGroup } }  
  ...  
SwitchBlockStatementGroup:  
  { SwitchLabel } { BlockStatement }  
SwitchLabel:  
  case ConstantExpression :  
  default :
```



## Zusammengefasst: Auswahlanweisung

- In Java werden alle nach einem passenden Label folgenden Anweisungen durchlaufen; auch wenn der nächste Label oder der **default**-Label erreicht wird.
- Um dies zu vermeiden, kann die Auswahlanweisung mit **break** verlassen werden. Alternativ ist dies auch mit **return** (Verlassen der Methode) oder **throw** (bisher: Abbrechen des Programms) möglich.
- In einer **switch**-Anweisung darf jeder **case**-Label nur einmal vorkommen.
- Wenn kein **case**-Label zutrifft und kein **default**-Label vorhanden ist, wird die gesamte **switch**-Anweisung übersprungen.



Ein (vermutliches) Negativbeispiel:

```
switch (i)  
{  
  case 1: System.out.println("eins");  
  case 2: System.out.println("zwei");  
  default: System.out.println("viele");  
}
```



## „Play it again, Sam“: Wiederholung durch Schleifen

- Computer sind besonders gut darin, klaglos die einfachsten Dinge beliebig oft zu wiederholen. Insbesondere sind sie dabei auch sehr schnell.

*„Der normale Mensch hat heute mehr Rechenpower zu Hause als ganz Houston, als es ein Apollo-Problem hatte.“*

(Gunter Dueck, 2008)

- **Wiederholungsanweisungen** (engl.: iterative statements) ermöglichen, dass Anweisungen keinmal, einmal oder mehrfach ausgeführt werden.
- In imperativen Sprachen werden sie umgangssprachlich auch als **Schleifen** (engl.: loops) bezeichnet.
- Es stellen sich einige Fragen:
  - Wie ist eine Wiederholungsanweisung aufgebaut?
  - Wie wird die Wiederholung kontrolliert?
  - Wo steht der Kontrollmechanismus innerhalb der Schleife?

## Die Grundidee: 1x hinschreiben, mehrfach ausführen lassen

- Wenn wir eine bestimmte Anweisung mehrfach ausführen lassen wollen, können wir dies erreichen, indem wir die Anweisung mehrfach in den Quelltext schreiben:

```
Anweisung A;  
Anweisung A;  
Anweisung A;  
Anweisung A;
```

- Dies ist meist nicht zweckmäßig, insbesondere, wenn die Anzahl der Ausführungen nicht bereits beim Schreiben des Quelltextes feststeht.
- Stattdessen schreiben wir die Anweisung nur einmal textuell in den Quelltext und sorgen mit einer umgebenden **Kontrollstruktur** dafür, dass diese Anweisung mehrfach ausgeführt wird.

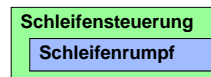
```
Wiederhole 4 x:  
    Anweisung A;  
Ende der Wiederholung
```



## Die Struktur einer imperativen Schleife



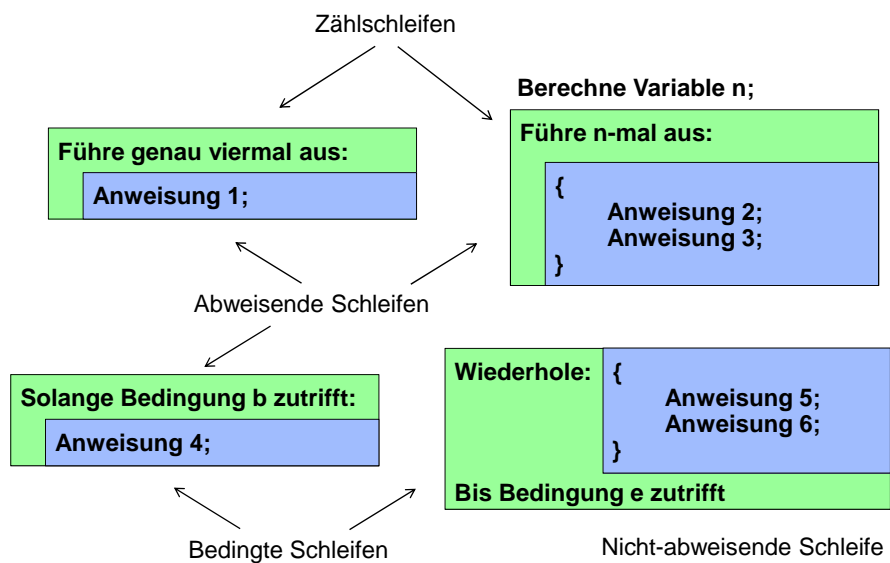
- Wir unterscheiden bei einer Schleife den Schleifenrumpf (engl.: loop body) von der Schleifensteuerung (engl.: loop control):
  - Der **Schleifenrumpf** enthält die zu wiederholenden Anweisungen; üblicherweise ist der Schleifenrumpf ein **Block** (also eine geklammerte Anweisungsfolge, die über eigene Variablen verfügen kann).
  - Die **Schleifensteuerung** steuert die Anzahl der Wiederholungen. Die Schleifensteuerung kann
    - eine feste Anzahl von Wiederholungen definieren
    - oder diese Anzahl von Variablen abhängig machen
    - oder von einer Bedingung, der **Schleifenbedingung**.
- Die Schleifensteuerung können wir uns als eine Art Rahmen oder Klammer um den Schleifenrumpf vorstellen.
- Wichtig: Im Schleifenrumpf können Anweisungen stehen, die Einfluss auf die Schleifensteuerung nehmen.



SE1 – Level 2

57

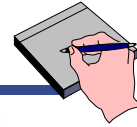
## Beispiele für Schleifenarten



SE1 – Level 2

58

## Abweisende und nicht-abweisende Schleifen



- Eine Schleife nennen wir **abweisend**, wenn es aufgrund der Schleifensteuerung auch dazu kommen **kann**, dass der Schleifenrumpf gar nicht ausgeführt wird.
- Beispielsweise sind alle Schleifen, bei denen zuerst eine Schleifenbedingung geprüft wird, abweisende Schleifen; denn je nach Ergebnis der ersten Auswertung der Bedingung kann der Schleifenrumpf mindestens einmal ausgeführt werden oder gar nicht.
- Abweisende Schleifen werden teilweise auch als **kopfgesteuerte Schleifen** bezeichnet.
- Wird hingegen der Schleifenrumpf auf jeden Fall mindestens einmal ausgeführt, sprechen wir von einer **nicht-abweisenden** Schleife. Sie wird auch als **fuß- oder endgesteuerte Schleife** bezeichnet.

## Bedingte Schleifen



- Die Ausführung einer **bedingten Schleife** ist mit einer **logischen Bedingung** verknüpft.
- Diese Bedingung wird entweder vor (abweisende Schleife) oder nach (endgesteuerte Schleife) **jeder Ausführung** des Schleifenrumpfes **erneut überprüft**.
- Die Bedingung **muss bei jedem** Schleifendurchlauf erneut geprüft werden, weil üblicherweise bei einem Durchlauf Anweisungen ausgeführt werden, die das Ergebnis der Prüfung beeinflussen.
- Unabhängig von der Frage, ob es sich um eine abweisende Schleife handelt oder nicht, kann die Bedingung für eine weitere Ausführung des Schleifenrumpfes **positiv** formuliert sein („Rumpf ausführen, **solange** die Bedingung zutrifft“) oder aus Sicht des Schleifenrumpfes **negativ** („ausführen, **bis** die Bedingung zutrifft“; also **nicht** mehr ausführen, wenn die Bedingung zutrifft).
- Die „Solange-Schleifen“ nennen wir **positiv bedingte Schleifen**, die „Bis-Schleifen“ **ziellorientiert bedingte Schleifen**.

## Aufpassen: Bedingte Schleifen an einem Beispiel

- **Beispiel:** Zwei `int`-Variablen `a` und `b` sollen wiederholt eingelesen und verarbeitet werden, bis beide den Wert `0` haben.
- Diese fachliche Anforderung ist direkt umsetzbar in Pseudo-Code:  
*wiederhole*  
**Schleifenrumpf: Einlesen und Verarbeiten der Werte für a und b**  
*bis (a gleich 0) und (b gleich 0)*
- Das „Problem“ in Java: Es gibt nur positiv bedingte Schleifen; alle bedingten Schleifen in Java werden ausgeführt, **solange** die Schleifenbedingung zutrifft.
- Folglich müssen wir die Bedingung für eine Java-Schleife **negieren**. Aus  
`(a == 0) && (b == 0)`
- wird dann:  
`(a != 0) || (b != 0)`
- Bei dieser **Negation** (logischen Umkehrung) der Bedingung kommen hier die **De Morganschen Regeln** der **Booleschen Algebra** zum Einsatz.

SE1 – Level 2

61

## Zählschleifen



- Bei einer **Zählschleife** ist die Anzahl der Wiederholungen zu Beginn der Schleife festgelegt, entweder durch eine Konstante oder durch die Belegung einer Variablen. Sie sind somit meist abweisend.
- Zählschleifen verfügen üblicherweise über einen **Schleifenzähler** (engl.: loop counter): eine Variable, die im einfachsten Fall die Schleifendurchläufe mitzählt.
- Der Schleifenzähler kann ausschließlich zur Schleifensteuerung dienen, er kann aber auch im Schleifenrumpf verwendet werden.
- Ein Beispiel in Pascal:

```
var i : Integer;  
for i := 1 to 10 do  
begin  
    Writeln('Hallo!');  
    Writeln('Durchlauf ', i);  
end;
```

SE1 – Level 2

62

## Realisierung von Schleifen in Java

Java bietet vier Schleifenkonstrukte zur Realisierung von Wiederholungen, von denen wir vorläufig nur drei betrachten:

**While-Schleife:** positiv bedingt, abweisend

```
while ( boolean_expression )  
    statement
```



**Do-While-Schleife:** positiv bedingt, endgesteuert

```
do  
    statement  
while ( boolean_expression )
```

**For-Schleife:** positiv bedingt, abweisend, ermöglicht u.a. Zählschleifen

```
for ( [ Init_Expr ]; [ Bool_Expr ]; [ Update_Expr ] )  
    statement
```

## Die for-Schleife in Java

- Die for-Schleife in Java ist sehr flexibel:
  - Die gesamte Schleifensteuerung kann zwischen den runden Klammern stehen (einschließlich der Deklaration einer Variablen als Schleifenzähler).
  - **Init\_Expr:** Der Teil der Schleifensteuerung vor dem ersten Semikolon wird einmalig zu Beginn der Schleife ausgeführt.
  - **Bool\_Expr:** Dann wird die Bedingung zwischen den beiden Semikola geprüft. Wenn diese zutrifft, wird der Schleifenrumpf ausgeführt.
  - **Update\_Expr:** Nach Ausführung des Schleifenrumpfes wird der Teil nach dem zweiten Semikolon ausgeführt.
  - Anschließend wird erneut die Bedingung geprüft, der Rumpf evtl. ausgeführt und das Update ausgeführt usw.
  - Alle Teile sind optional.



```
for (int i = 0; i < 10; ++i)  
{  
    System.out.println("Hallo!");  
    System.out.println("Durchlauf " + i);  
}
```

## Endlosschleifen

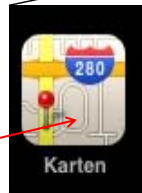
- **Endlosschleifen** (engl.: infinite loop) sind meist ungewollt und deshalb unbeliebt.
- Üblicherweise ist die Schleifenbedingung bei einer Endlosschleife falsch gewählt.
- Die einfachsten Endlosschleifen in Java:

```
while (true)
{
    // endlos wiederholt
}
```

oder:

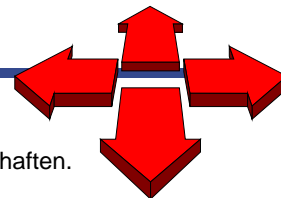
```
for ( ; ; )
```

Die Adresse von  
**Apples**  
Hauptquartier in  
Cupertino, CA:  
**Infinite Loop 1**



## Statische und dynamische Eigenschaften

- Programme haben **statische** und **dynamische** Eigenschaften.
- Die statischen Eigenschaften können bei der Übersetzung überprüft werden; dazu zählt auch die **Sichtbarkeit** von Programmelementen.
- Die dynamischen Eigenschaften zeigen sich bei der Ausführung eines Programms; dazu zählt auch die **Lebensdauer** von Variablen und Objekten.
- Ein **Compiler** überprüft u.a. die statischen Eigenschaften von Programmen; wir untersuchen dies näher.



## Laufzeit und Übersetzungszeit



- Zwei zentrale Begriffe für Programmiersprachen, die durch Compiler übersetzt werden, sind **Laufzeit** und **Übersetzungszeit**.
- **Übersetzungszeit** (engl.: compile time) ist die Zeit, in der ein Compiler den in einer Programmiersprache geschriebenen Quelltext in eine ausführbare Form übersetzt. Hier sind die **statischen Eigenschaften** von Programmen relevant:
  - Welche **Syntaxregeln** gibt es? Welche **Sichtbarkeitsregeln** gelten?
  - Wie ist der Quelltext **strukturiert**? Welche Klassen, Methoden etc. gibt es?
  - Wie **lesbar** ist der Quelltext (**Konventionen** etc.)?



## Laufzeit und Übersetzungszeit (II)



- **Laufzeit** (engl.: run time) ist die Zeit, in der ein Computerprogramm im Rechner von seinem Start bis zur Termination (Beendigung) ausgeführt wird. Hier sind die **dynamischen Eigenschaften** von Programmen relevant:
  - **Semantik: Was** macht das Programm?
  - Wie viele **Objekte** werden **erzeugt**? Welche **Lebensdauer** haben sie?
  - Welche **Methoden** werden **aufgerufen**? Welche **Daten manipuliert**?



## Veranschaulichungen des Unterschieds

- Am Beispiel von lokalen Variablen können wir den Unterschied zwischen Übersetzungs- und Laufzeit veranschaulichen.
- Was wissen wir zur **Übersetzungszeit** über eine lokale Variable? Wir kennen
  - ihren Namen
  - ihren Typ
  - ihre Sichtbarkeit (nur innerhalb ihrer Prozedur)
- Was wissen wir meist erst zur **Laufzeit** über eine lokale Variable?
  - ihre Belegungen (können bei jeder Ausführung anders sein)
  - ihre Lebensdauer (wie lange wird sie benötigt?)
  - ihre Adresse (wo steht sie im Speicher?)



SE1 – Level 2

69

## Ein weiteres Beispiel für den Unterschied

- „Die return-Anweisung ist immer die letzte Anweisung in einer Methode.“
- Was ist dann mit folgender Methode:
 

```
public int max(int a, int b)
{
    if (a > b)
    {
        return a;
    }
    else
    {
        return b;
    }
}
```

  - Gemeint ist: **Zur Laufzeit** ist die return-Anweisung immer die letzte Anweisung in einer Methode!
- Zur besseren **Lesbarkeit** könnte eine lokale Variable für das Ergebnis deklariert werden, das nur einmal am Ende der Methode zurückgegeben wird:
 

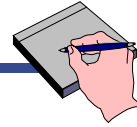
```
public int max(int a, int b)
{
    int max;
    if (a > b)
    {
        max = a;
    }
    else
    {
        max = b;
    }
    return max;
}
```

→
- Diese return-Anweisung ist dann auch **statisch** (zur **Übersetzungszeit**) die letzte Anweisung.

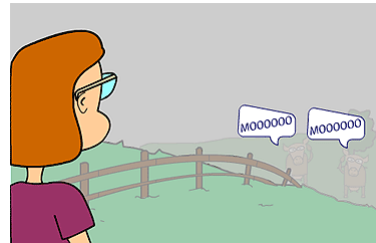
SE1 – Level 2

70

## Sichtbarkeitsbereich



- Ein zentraler Begriff der (imperativen) Programmierung ist der **Sichtbarkeitsbereich** (engl.: scope):
  - Jedem Bezeichner in einem Programm wird ein **Bereich** zugeordnet, in dem er angesprochen und benutzt werden kann.
  - Auf den Wert einer sichtbaren Variablen kann z.B. über ihren Namen zugegriffen werden.
- In imperativen und objektorientierten Sprachen ist der Sichtbarkeitsbereich am **Programmtext (statisch)** feststellbar. Der Sichtbarkeitsbereich eines Bezeichners ist gleich der Programmeinheit, in der der Bezeichner deklariert ist.



SE1 – Level 2

71

## Sichtbarkeitsbereich objektorientiert

- **Methoden** bilden gegenüber ihrer Umgebung einen eigenen **Sichtbarkeitsbereich**, d.h. sie können lokale Variablen benennen und verwalten.
- Alle im Rumpf einer Methode deklarierten **Variablen** sind **nur im Rest des Methodenrumpfes**, aber nicht außerhalb der Methode sichtbar.
- Die **Umgebung** einer Methode ist in objektorientierten Sprachen ihre **Klasse**, sie bildet den unmittelbar übergeordneten Sichtbarkeitsbereich.
- Die **Exemplarvariablen** einer Klasse sind in allen Methoden der Klasse sichtbar, ebenso wie alle Methoden.
- Die **Sichtbarkeitsbereiche von Klasse und Methode** sind in einander **geschachtelt**.
- In Java können Methoden im Inneren noch weiter durch sog. **Blöcke** in Sichtbarkeitsbereiche unterteilt werden.

SE1 – Level 2

72

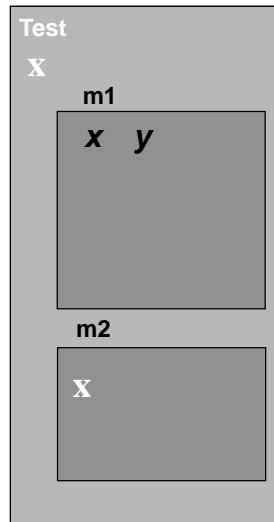


### Beispiel: Sichtbarkeitsbereiche

```
class Test {
    private int x = 0;

    public void start(){
        m1(); m2();
    }

    private void m1(){
        double x,y;
        ...
        x = 1.5;
        ...
    }
    private void m2(){
        ...
        x = 5;
        ...
    }
}
```



- In der Klasse **Test** sichtbar:  
x
- Nur in m1 sichtbar:  
**x, y**
- In m1 verdeckt:  
x
- In m2 ist sichtbar:  
x

SE1 – Level 2

73

### Verdecken von Bezeichnern

- Eine lokale Variable kann den gleichen Bezeichner haben wie eine Variable mit größerer Sichtbarkeit, z.B. eine Exemplarvariable.
- Man sagt dann, dass die lokale Variable die Exemplarvariable „verdeckt“; diese ist dann lokal nicht mehr sichtbar.
- Vorsicht mit dieser Technik des „Verdeckens“, die selten sinnvoll ist und oft zu Fehlern führt.



Hier helfen uns die Quelltextkonventionen: Wenn wir Exemplarvariablen mit führendem Unterstrich benennen (und Parameter und lokale Variablen nicht), kann es nicht zu Überdeckungen kommen.

SE1 – Level 2

74

## Klassischer Fehler: Versehentliches Überdecken

```
class Uhrenanzeige
{
    private Nummernanzeige _stunden;
    private Nummernanzeige _minuten;

    public Uhrenanzeige()
    {
        Nummernanzeige _stunden = new Nummernanzeige(24);
        Nummernanzeige _minuten = new Nummernanzeige(60);
    }
}
```



richtig:



```
class Uhrenanzeige
{
    private Nummernanzeige _stunden;
    private Nummernanzeige _minuten;

    public Uhrenanzeige()
    {
        _stunden = new Nummernanzeige(24);
        _minuten = new Nummernanzeige(60);
    }
}
```

SE1 – Level 2

75

## Sichtbarkeit der Elemente einer Klasse in Java

In Java kann die Sichtbarkeit von Sprachelementen (hier: Methoden und Exemplarvariablen) durch Modifikatoren (engl.: modifiers) festgelegt werden. Wir kennen bisher folgende Modifikatoren für die Elemente einer Klasse:

### public

legt für ein Element der Klasse fest, dass es für Klienten sichtbar und damit öffentlich zugänglich ist. Wir nutzen dies für Methoden, die die Schnittstelle der Klasse bilden sollen.

### private

legt für ein Element der Klasse fest, dass es nur innerhalb der Klasse zugänglich ist. Wir nutzen dies meist für Exemplarvariablen und Hilfsmethoden.



Dazu kommen **protected** und **<default>**, die erst in SE2 thematisiert werden.

SE1 – Level 2

76

## Lebensdauer



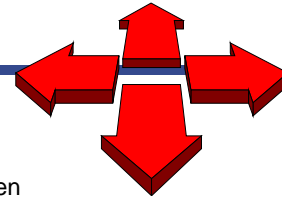
- Die **Lebensdauer** (engl.: lifetime) einer Variablen oder eines Objektes ist eine *dynamische Eigenschaft*. Lebensdauer bezeichnet die Zeit, in der eine Variable (oder ein ggf. damit verbundenes Objekt) während der Laufzeit **existiert**. Während der Lebensdauer ist einer Variablen (oder einem Objekt) **Speicherplatz** zugewiesen.
- Sichtbarkeit und Lebensdauer können unabhängig voneinander sein, wie in folgender Situation:
  - Eine Exemplarvariable **x** ist statisch deklariert, ein entsprechendes Feld eines Objektes hält zur Laufzeit einen Wert.
  - In einer Methode ist eine gleichnamige lokale Variable **x** deklariert, die die Exemplarvariable verdeckt. Obwohl sie weiter im Speicher existiert, ist die Exemplarvariable während der Ausführung der Methode **nicht über den Namen x sichtbar**.

## Zusammenfassung



- Die **strukturierte Programmierung** beschränkt sich auf die **Kontrollstrukturen** Sequenz, Verzweigung und Wiederholung.
- **Schleifenkonstrukte** in imperativen Sprachen sind die einfachste Form für Wiederholungen.
- Wir haben die **Sichtbarkeit** und die **Lebensdauer** von Programmelementen kennen gelernt.
- Die Sichtbarkeit von Programmelementen ist eine **statische** Eigenschaft innerhalb des Programmtextes, die zur **Übersetzungszeit** geprüft werden kann.
- Die Lebensdauer von Programmelementen ist eine **dynamische** Eigenschaft und legt fest, wie lange sie während der **Laufzeit** eines Programms existieren.
- Sichtbarkeit und Lebensdauer hängen teilweise eng zusammen.

## Rekursion



- Prozeduren/Methoden können sich in modernen Sprachen auch **selbst aufrufen** und damit **rekursiv** definiert sein.
- Rekursion ist neben den klassischen Schleifenkonstrukten eine zweite Möglichkeit, **Wiederholungen** zu programmieren.

SE1 – Level 2

79

## Rekursion: ein erstes Beispiel

Ein häufiges Beispiel für die Verwendung einer rekursiven Programmierung ist die Berechnung der **Fakultät** einer Zahl. Die Fakultät  $n!$  ist das Produkt aller natürlichen Zahlen von  $1$  bis  $n$ .  $4!$  beispielsweise ist  $1 * 2 * 3 * 4$ , also  $24$ .

Die mathematische Definition der Fakultät lautet:

- Die Fakultät der Zahl  $0$  ist  $1$
- Die Fakultät einer natürlichen Zahl  $n$ , mit  $n > 0$ , ist  $n * (n-1)!$

rekursive Definition

In Java lässt sich das so notieren:

```
public int fakultaet (int n)
{
    int result;
    if (n == 0)
    {
        result = 1;
    }
    else
    {
        result = n * fakultaet(n-1);
    }
    return result;
}
```

rekursiver Aufruf



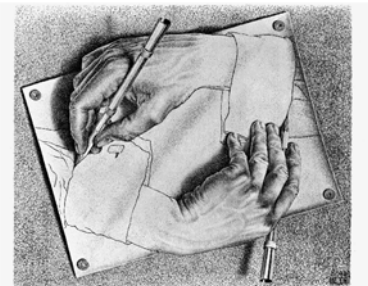
SE1 – Level 2

80

## Rekursion



- Rekursion tritt auf, wenn eine Methode **m** während der Ausführung ihres Rumpfes erneut aufgerufen wird. Damit dieser Prozess nicht endlos läuft („nicht terminiert“), ist eine **Abbruchbedingung** zwingend notwendig.
- Wir unterscheiden:
  - Eine Rekursion ist **direkt**, wenn eine Methode **m** sich im Rumpf selbst ruft.
  - Eine Rekursion ist **indirekt**, wenn eine Methode **m1** eine andere Methode **m2** ruft, die aus ihrem Rumpf **m1** aufruft.
- Der Grundgedanke der Rekursion ist, dass die Methode einen **ersten** Teil eines Problems selbst löst, den Rest in kleinere Probleme zerlegt und sich selbst mit diesen kleineren Problemen aufruft.



SE1 – Level 2

81

## Rekursion: Grundstruktur

```

public <Ergebnistyp> loeseProblem ( <formale Parameter> )
{
    if ( <ProblemEinfachLösbar> )
    {
        return <EinfachesErgebnis>
    }
    else
    {
        <zerlegeProblem>
        <Ergebnis1> = loeseProblem ( <veränderteParameter> );
        <Ergebnis2> = loeseProblem ( <veränderteParameter> );
        ...
        return <ausgewerteteErgebnisse> ;
    }
}

```

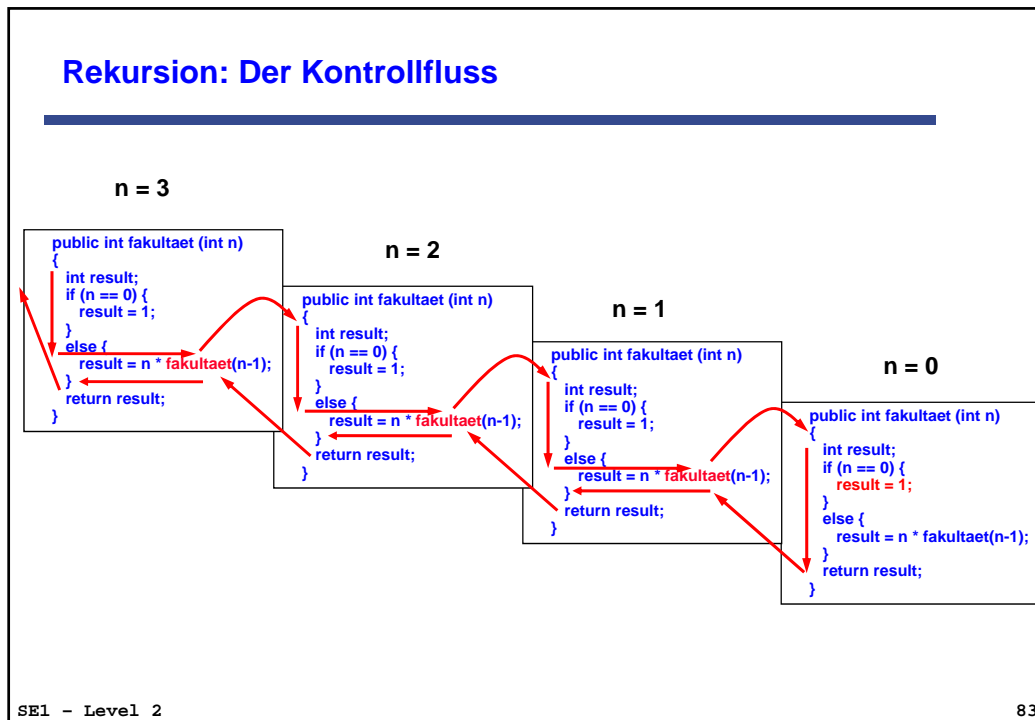
Abbruchbedingung

rekursive Aufrufe



SE1 – Level 2

82



### Der Aufrufstack

- Ein **Aufrufstack** (engl.: call stack oder function stack) ist eine Speicherstruktur, in der **zur Laufzeit** Informationen über die gerade aktiven Methoden gespeichert werden (in sogenannten **Stackframes**).
- Bei jedem neuen Methodenaufruf werden die **Rücksprung-adresse** und die **lokalen Variablen** (schließen die formalen Parameter mit ein) in einem neuen Stackframe auf dem Stack gespeichert. Wenn eine Methode terminiert, wird der zugehörige Stackframe wieder **vom Stack geräumt**.
- In höheren Programmiersprachen wie Java ist der Aufrufstack für die Programmierung zwar nicht zugänglich, Kenntnisse über seine Verwaltung erleichtern jedoch das Verständnis der Programmierung.

SE1 - Level 2 84

### Rekursion: Der Aufrufstack für das Beispiel

- Da die lokalen Variablen auf dem Aufrufstack gespeichert werden, können **rekursive Methodenaufrufe** einfach realisiert werden:
  - Für jeden rekursiven Aufruf wird ein neuer Satz lokaler Variablen in einem Stackframe gespeichert.
  - So kann eine Methode auf jeder Rekursionsstufe auf ihren eigenen lokalen Variablen arbeiten und ihr Funktionsergebnis zurückgeben.
- Für den Beispielausdruck `23 + fakultaet(4)` würde jeder Aufruf folgende Informationen auf dem Stack ablegen:
  - Platz für Ergebnis
  - Argument n
  - Rücksprungadresse in die rufende Methode



### Rekursion: Das Beispiel als iteratives Programm

- Rekursive Programme haben in den meisten imperativen Programmiersprachen kein gutes Speicher- und Ablaufverhalten. Durch die wiederholten Methodenaufrufe wird immer wieder derselbe Programmcode bearbeitet und jedesmal ein neues Segment auf dem Aufrufstack belegt; ein vergleichsweise hoher Aufwand.
- Alternativ lässt sich die Fakultät in Java auch **iterativ** programmieren:

```
public int fakultaet (int n)
{
    int fak = 1;
    for (int i = 1; i <= n; i++)
    {
        fak = i * fak;
    }
    return fak;
}
```



## Rekursion: DAS Gegenbeispiel

- Die **Fibonacci-Zahlen** sind sehr einfach definiert:
  - Die erste Fibonacci-Zahl ist 0.
  - Die zweite Fibonacci-Zahl ist 1.
  - Die n-te Fibonacci-Zahl ergibt sich aus der Summe der (n-1)ten und der (n-2)ten Fibonacci-Zahl.
- Die dritte Fibonacci-Zahl ist demnach 1, die vierte 2, die fünfte 3, die sechste 5 usw.
- Die rekursive Definition lässt sich unmittelbar rekursiv realisieren:

```
public int fibonacci (int n)
{
    switch (n) {
        case 1: return 0;
        case 2: return 1;
        default: return fibonacci(n-1) + fibonacci(n-2);
    }
}
```



Wo ist das Problem?

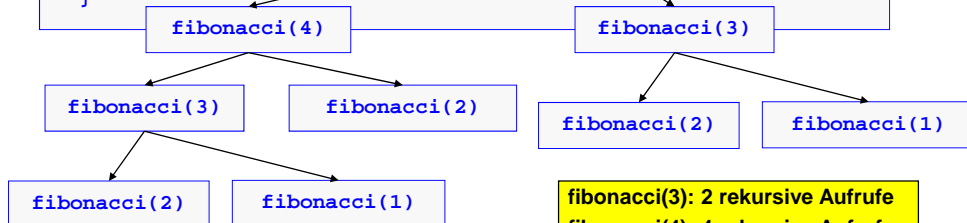
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987

SE1 - Level 2

87

## Rekursion: Wir beginnen etwas zu ahnen...

```
public int fibonacci(5)
{
    switch(5) {
        case 1: return 0;
        case 2: return 1;
        default: return fibonacci(4) + fibonacci(3);
    }
}
```



fibonacci(3): 2 rekursive Aufrufe  
 fibonacci(4): 4 rekursive Aufrufe  
 fibonacci(5): 8 rekursive Aufrufe  
 fibonacci(6): 16 rekursive Aufrufe  
 ...



SE1 - Level 2

88

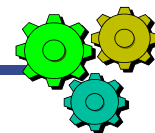


## Rekursion: Elegante Anwendungen

- Rekursion ist besonders in folgenden Fällen geeignet:
  - Wenn die Struktur, die verarbeitet wird, selbst rekursiv definiert ist; darunter fallen zum Beispiel alle **Baumstrukturen** in der Informatik (Syntaxbäume, Entscheidungsbäume, Verzeichnisbäume, etc.).
  - Viele sehr gute **Sortiervverfahren** sind rekursiv definiert, beispielsweise Quicksort und Heapsort.
  - Viele Probleme auf **Graphen** lassen sich elegant rekursiv lösen.
- Im Laufe Ihres Studiums werden Sie noch viele Anwendungsfälle von Rekursion kennen lernen!

Wir werden in SE1 noch einige gute Anwendungen von Rekursion betrachten.

## Rekursion: Stärken und Schwächen

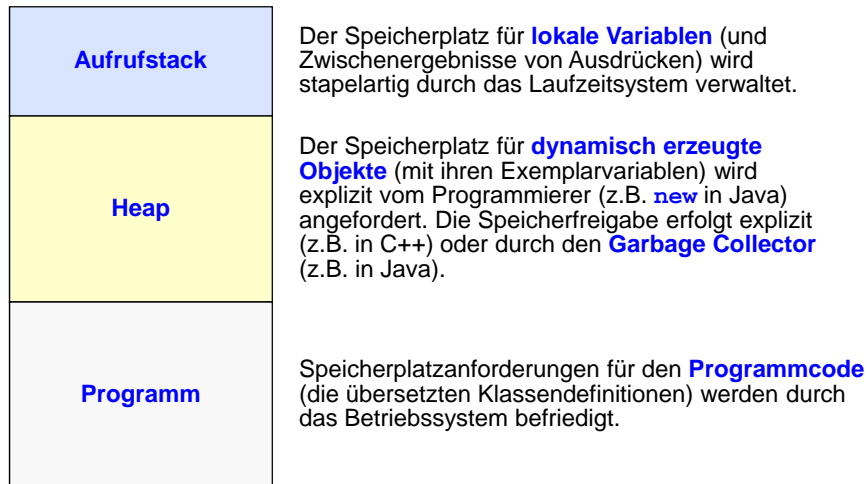


**Steve McConnell's** Einschätzung zu Rekursion:

- Rekursion kann für eine relativ kleine Menge von Problemen **sehr einfache, elegante Lösungen** produzieren.
- Rekursion kann für eine etwas größere Menge von Problemen sehr einfache, elegante und **schwer zu verstehende Lösungen** produzieren.
- Für die meisten Probleme führt die Benutzung von Rekursion zu **sehr komplizierten Lösungen** – in solchen Fällen sind simple Iterationen meist verständlicher. **Rekursion sollte sehr selektiv eingesetzt werden.**

Ergo: Es gibt Situationen, in denen Rekursion sich als gute Lösung anbietet. Es gibt mehr Situationen, in denen Rekursion sich als Lösung **verbietet**.

## Vereinfachtes Speichermodell von Sprachen mit dynamischen Objekten

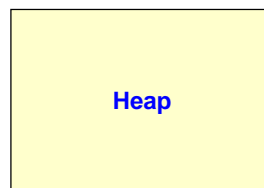


SE1 – Level 2

91

## Der Heap

- Der **dynamische Speicher**, auch **Heap** (engl. für *Halde*, *Haufen*) ist ein Speicherbereich, aus dem zur Laufzeit eines Programmes zusammenhängende Speicherabschnitte angefordert und in **beliebiger Reihenfolge** wieder freigegeben werden können. Die Freigabe kann sowohl manuell als auch mit Hilfe einer automatischen Speicherbereinigung (engl.: garbage collection) erfolgen.
- Eine Speicheranforderung vom Heap wird auch **dynamische Speicheranforderung** genannt.
- Kann eine Speicheranforderung wegen Speichermangel nicht erfüllt werden, kommt es zu einem Programmabbruch (in Java: **OutOfMemoryError**).

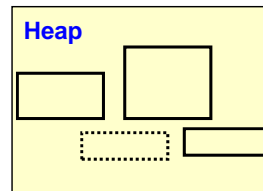
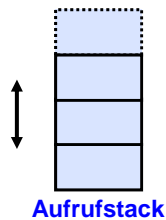


SE1 – Level 2

92

## Heap und Aufrufstack

- Der Unterschied zwischen Aufrufstack und Heap besteht darin, dass beim Aufrufstack angeforderte Speicherabschnitte **strikt in der umgekehrten Reihenfolge** wieder freigegeben werden, in der sie angefordert wurden.
- Beim Aufrufstack spricht man deshalb auch von **automatischer Speicheranforderung**. Die Laufzeitkosten einer automatischen Speicheranforderung sind in der Regel deutlich geringer als die bei der dynamischen Speicheranforderung.
- Allerdings kann bei spezieller Nutzung durch sehr große oder sehr viele Anforderungen der für den Stack reservierte Speicher ausgehen - dann droht ein Programmabbruch wegen **Stapelüberlauf** (in Java: **StackOverflowError**).

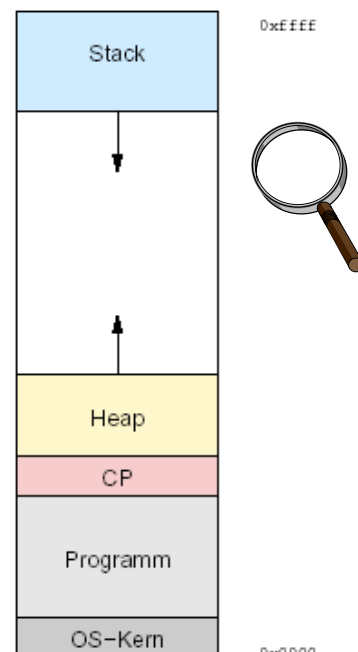


SE1 - Level 2

93

## Beispiel: Speichereinteilung in einem Unix-System

- **Programm:** enthält den eigentlichen Programmtext mit allen Befehlen. Sofern keine selbstmodifizierende Programme zum Einsatz kommen, bleibt das Textsegment während des Programmlaufs unverändert.
- **Constant Pool (CP):** nimmt alle Konstanten und statischen Variablen des Programms auf.
- **Heap:** nimmt alle dynamisch zur Laufzeit des Programms erzeugten Variablen bzw. Objekte auf.
- **Stack:** wird für die Parameterübergabe zwischen Funktionen und für die Speicherung der lokalen Variablen der einzelnen Funktionen benutzt.

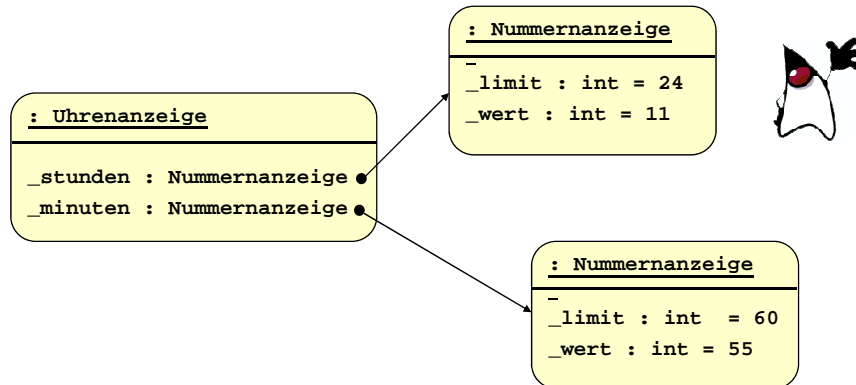


SE1 - Level 2

94

## Java-Objektdiagramme: Schnappschüsse vom Heap

- Ein Objektdiagramm ist in Java immer ein Schnappschuss vom **Heap** eines laufenden Programms.
- Es zeigt einen Ausschnitt des Objektgeflechts zur Laufzeit in der **Virtual Machine**, um einen bestimmten Aspekt zu verdeutlichen.



SE1 - Level 2

95

## Der Garbage Collector in Java

- Mit unserem Wissen über Heap und Stack können wir nun erstmalig nachvollziehen, was der **Garbage Collector** von Java macht.
- Die Voraussetzungen sind:
  - **Alle** Objekte eines Java-Programms liegen im Heap.
  - Auf dem **Aufrufstack** in den Speicherplätzen für die lokalen Variablen liegen entweder primitive Werte oder **Referenzen auf Objekte**.
  - Nur diejenigen Objekte, die **vom Aufrufstack** aus **erreichbar** sind, spielen für die Programmausführung eine Rolle. Alle anderen Objekte im Heap sind „tote“ Objekte.
- Daraus folgt das **Vorgehen** des Garbage Collectors:
  - Er verfolgt in regelmäßigen Abständen, ausgehend von den Referenzen auf dem Stack, transitiv das **gesamte Objektgeflecht** und **markiert** die erreichbaren Objekte. Anschließend werden alle nicht markierten Objekte im Heap **gelöscht**. Dieses Vorgehen aus **Markieren** und **Abräumen** heißt im Englischen **Mark and Sweep**.



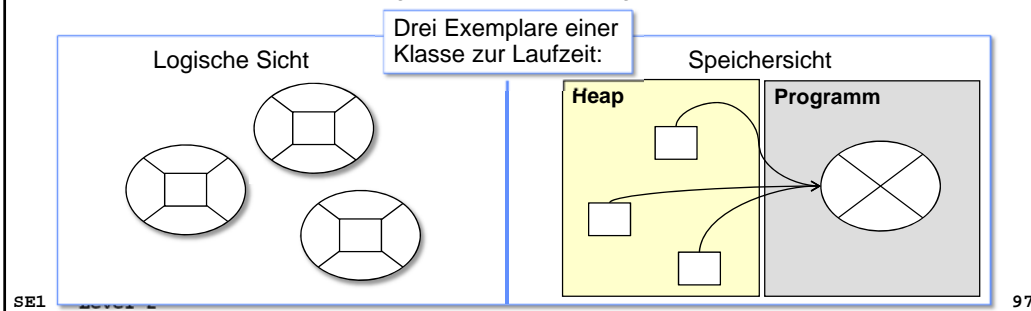
SE1 - Level 2

96

## Level 2: Objekte benutzen Objekte

## Methoden und Zustandsfelder

- Den Zusammenhang zwischen statischen und dynamischen Eigenschaften können wir anhand der Methoden und Felder noch einmal verdeutlichen:
  - zur **Übersetzungszeit** gibt es jede **Methode** nur **einmal**, ebenso wie die **Exemplarvariablen**. Sie sind statisch in den **Klassendefinitionen** beschrieben.
  - zur **Laufzeit** gibt es für jedes Exemplar einer Klasse einen eigenen Satz Zustandsfelder und **logisch** auch einen Satz Methoden; dass ein Satz von Methoden (in der Klasse abgelegt) für alle Exemplare einer Klasse ausreicht, ist lediglich eine Optimierung.

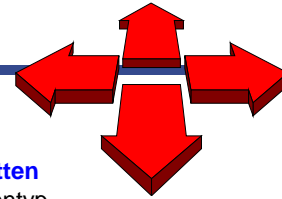


## Zusammenfassung



- Rekursive Methodenaufrufe** sind eine alternative Möglichkeit für Wiederholungen.
- Jede Wiederholung lässt sich **sowohl iterativ als auch rekursiv** formulieren, jeweils mit spezifischen Vor- und Nachteilen.
- Softwaretechnische Überlegungen wie **Verständlichkeit und Sicherheit** spielen bei der Wahl einer geeigneten Realisierung eine wichtige Rolle.
- „Hinter den Kulissen“ moderner Programmiersprachen sind der **Aufrufstack** und der **Heap** zentrale Strukturen für die Verwaltung von Variablen und Objekten.

## Strings und Reguläre Ausdrücke



- Sehr häufig werden bei der Programmierung **Zeichenketten** verarbeitet. Java definiert mit dem Typ **String** einen Datentyp für **unveränderliche** Zeichenketten.
- Programme, als Folgen von Zeichen aufgefasst, lassen sich in elementare Bestandteile zerlegen, die **Token** genannt werden.
- **Reguläre Ausdrücke** sind ein mächtiges Beschreibungsmittel für Token, aber auch für andere Zwecke einsetzbar.

SE1 – Level 2

99

## Zeichenketten in Programmiersprachen



- Moderne Programmiersprachen bieten Unterstützung für **Zeichenketten** (engl.: strings). Eine Zeichenkette ist eine Folge von einzelnen Zeichen.

|   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 4 | . |   | A | d | v | e | n | t | ? |

- Die Anzahl der Zeichen in einer Zeichenkette wird auch als ihre **Länge** bezeichnet. Konzeptuell sind Zeichenketten in ihrer Länge **unbegrenzt**. In einigen Kontexten (z.B. Datenbanken) müssen Zeichenketten jedoch eine fest definierte Maximallänge haben.
- Eine Unterstützung für Zeichenketten ist in allen Anwendungen notwendig, in denen Texte (Prosa, Quelltexte, etc.) verarbeitet werden.
- In objektorientierten Sprachen werden Zeichenketten üblicherweise als Objekte modelliert.

Datentyp: **Zeichenkette**  
 Wertemenge: { Zeichenketten beliebiger Länge }  
 Operationen: **Länge**, **Subzeichenkette**, **Zeichen an Position x**, ...

SE1 – Level 2

100

## Zeichenketten in Java: Literale, Konkatenation



- In Java werden Zeichenketten primär durch die Klasse `String` unterstützt. Diese Klasse definiert, wie alle Klassen in Java, einen Typ.
- `String` ist in Java ein expliziter Bestandteil der Sprache, denn es gibt einige Spezialbehandlungen für diesen Typ:
  - String-Literale** (Zeichenfolgen zwischen doppelten Anführungszeichen) werden vom Compiler speziell erkannt:
 

```
String s = "Banane";
```
  - Der Infix-Operator `+` kann auch auf Strings angewendet werden; er konkateniert (verkettet) zwei Strings zu einem neuen String.
- Von der Klasse `String` gibt es eine javadoc-Darstellung, die alle Methoden beschreibt, die Klienten zur Verfügung stehen:
  - <http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html>

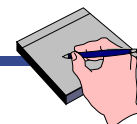


Datentyp: `String`  
 Wertemenge: { `String`-Exemplare beliebiger Länge }  
 Operationen: `length`, `concat`, `substring`, `charAt`, ...

SE1 – Level 2

101

## Escape-Sequenzen in String-Literalen



- Angenommen, wir wollen folgendes ausgeben:
 

```
Bitte einmal "Aaah" sagen!
```
- Erster Versuch:
 

```
System.out.println("Bitte einmal "Aaah" sagen!");
```
- Das Problem: Der Compiler sieht zwei String-Literale, getrennt von dem (ihm unbekannten) Bezeichner `Aaah`, da das zweite Anführungszeichen das erste String-Literal beendet.
- Wenn wir Anführungszeichen in einem String-Literal platzieren wollen, müssen wir eine so genannte **Escape-Sequenz** anwenden:
 

```
System.out.println("Bitte einmal \"Aaah\" sagen!");
```

| Gewünschtes Zeichen | Escape-Sequenz   |
|---------------------|------------------|
| Anführungszeichen   | <code>\ "</code> |
| Backslash           | <code>\\</code>  |
| Zeilenumbruch       | <code>\n</code>  |



SE1 – Level 2

102

## Strings in Java: Unveränderlich!



- Die Klasse **String** in Java definiert Objekte, die **unveränderliche Zeichenketten** sind:
  - Alle Operationen auf Strings liefern Informationen über ein **string-Objekt** (einzelne Zeichen, neue Zeichenketten), **verändern es aber niemals**.
  - Der Infix-Operator **+** verkettet zwei Strings zu **einem neuen** String.
- Strings sind damit sehr **untypische Objekte** in Java, denn sie haben keinen (veränderbaren) Zustand.



### Typischer Fehler:

```
String s = „FckW“;
s.toUpperCase(); // Das Ergebnis dieses Aufrufs verpufft...
```



SE1 – Level 2

103

## Gleichheit von Strings in Java



"Banane" == "Banane"

"Banane" == new String("Banane")



### Das Problem:

!=

Datentyp: **Zeichenkette**  
 Wertemenge: { Zeichenketten beliebiger Länge }  
 Operationen: **Länge**, **Subzeichenkette**, ...



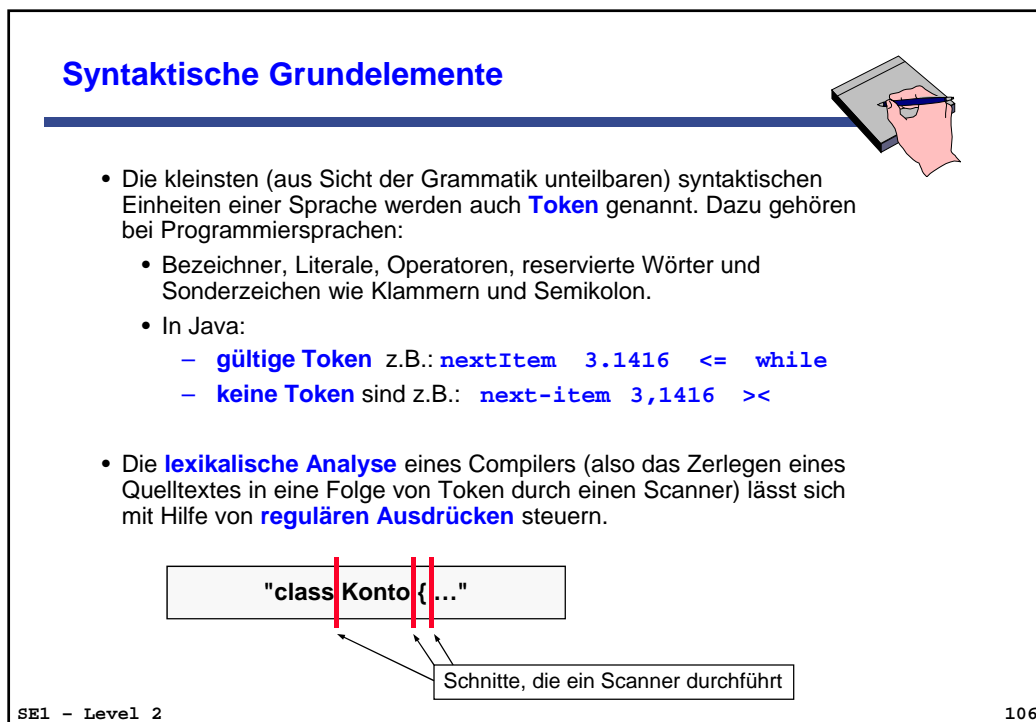
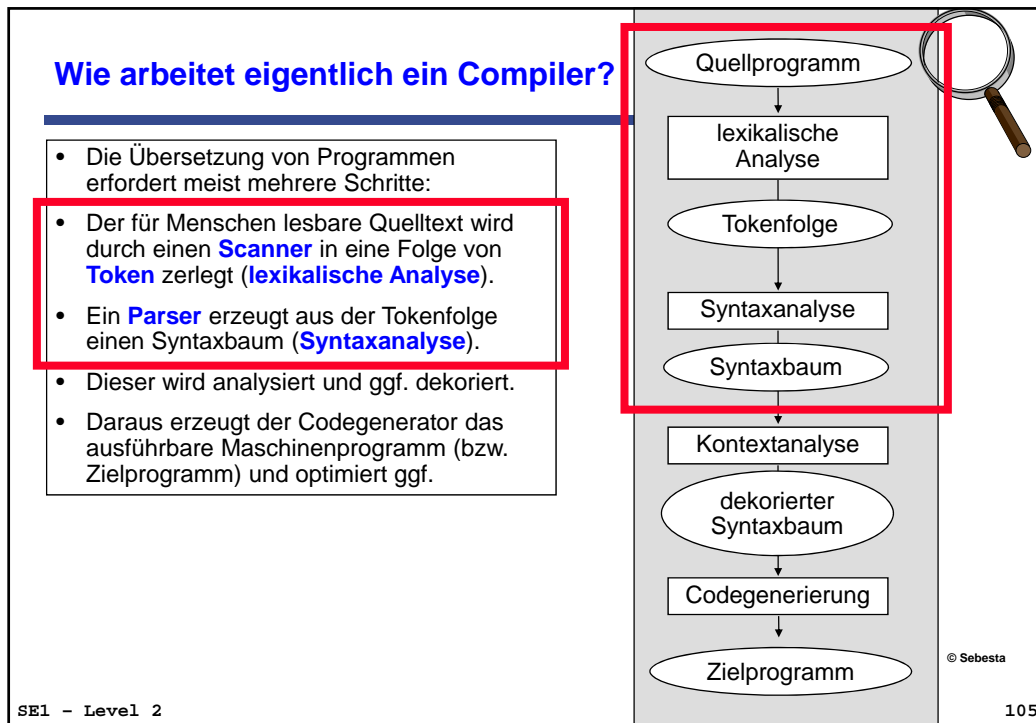
Datentyp: **String**  
 Wertemenge: { **string**-Exemplare beliebiger Länge }  
 Operationen: **length**, **concat**, **substring**, **charAt**, ...

- Weil Strings in Java Objekte sind, werden mit dem Operator **==** lediglich Referenzen verglichen. Zwei String-Objekte können dieselbe Zeichenkette repräsentieren, sind aber dennoch verschiedene String-Exemplare.
- Deshalb: **Strings in Java immer mit der equals-Methode vergleichen!**

SE1 – Level 2

104





## Ein erstes Beispiel: Das Token „Bezeichner“

- Ein sehr häufiges Token ist der **Bezeichner**. Bezeichner werden verwendet, um Variablen, Methoden, Klassen etc. zu benennen.
- Definition eines Bezeichners in **Java** (leicht vereinfacht):
  - Ein Bezeichner besteht aus einem Buchstaben (ein Unterstrich wird auch als ein Buchstabe angesehen), gefolgt von beliebig vielen Buchstaben und Ziffern.
- Wenn wir von einer Zeichenkette **s** (vom Typ **String**) feststellen wollen, ob sie ein gültiger Bezeichner ist, dann fragen wir etwas abstrakter, ob **s** ein Element der **Menge aller gültigen Bezeichner** ist. In Java können wir diese Frage beispielsweise so ausdrücken:
 

```
s.matches(mengeGueltigerBezeichner)
```
- Die Methode **matches** ist in der Klasse **String** definiert und erhält als Parameter einen String, der als **regulärer Ausdruck** aufgefasst wird. Ein regulärer Ausdruck beschreibt eine Menge von Zeichenketten, und die Methode **matches** liefert **true** genau dann, wenn die Zeichenkette **s** ein Element dieser Menge ist, ansonsten **false**.



## Bezeichner als regulärer Ausdruck

- Die Menge aller Zeichenketten, die durch einen regulären Ausdruck beschrieben wird, wird als **reguläre Menge** bezeichnet. Aber wie sieht ein solcher Ausdruck aus?
- Als erstes betrachten wir eine Möglichkeit, unsere vereinfachte Definition eines Java-Bezeichners als regulären Ausdruck zu beschreiben:

```
String mengeGueltigerBezeichner = "[a-zA-Z_][a-zA-Z_0-9]*";
```

- Wenn an einer bestimmten Stelle eines aus einer **Menge einzelner Zeichen** möglich sein soll, dann können diese Zeichen **in eckigen Klammern** angegeben werden:
 

```
h[oa]se
```

 beispielsweise definiert die reguläre Menge { **hose**, **hase** }.
- Zur weiteren Verkürzung erlaubt Java in den eckigen Klammern auch Bereichsangaben mit einem Minuszeichen. Beispielsweise:
 

```
se[1-3]
```

 definiert die reguläre Menge { **se1**, **se2**, **se3** }.
 

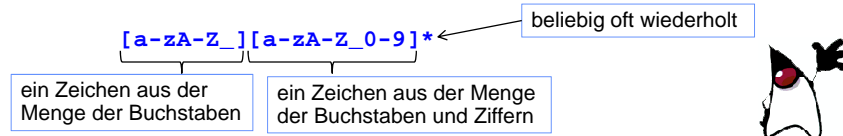
```
[a-z]
```

 definiert alle Kleinbuchstaben von a bis z.



## Reguläre Ausdrücke in Java

- Somit haben wir unseren regulären Ausdruck schon fast verstanden:



- Der `*` ist ein Postfix-Operator und besagt in einem regulären Ausdruck, dass sein Operand beliebig oft auftreten kann (auch gar nicht). In diesem Fall ist der Operand die zweite Menge, die Buchstaben und Ziffern definiert.
- Alternativ zum Postfix-Operator `*` (beliebige Wiederholung) bietet Java zusätzlich die Postfix-Operatoren `+` (beliebige Wiederholung, mindestens einmal) und `?` (entweder einmal oder gar nicht).
- Ein einzelner Punkt (`.`) in einem regulären Ausdruck steht für ein beliebiges Zeichen.

## Weitere Beispiele für reguläre Ausdrücke in Java

- Weitere Beispiele:

```
String s = "ab";
s.matches("ab"); => true, jeder String definiert sich selbst als Ausdruck
s.matches("a"); => false, nur ein teilweiser „Match“
s.matches("aba"); => false, auch knapp daneben
```

```
String re = "[ab][ab]";
s.matches(re); => true (gälte auch für s = "aa" oder "ba" oder "bb")
```

```
re = "(ab)*"; // "ab" beliebig oft wiederholt; runde Klammern gruppieren
s.matches(re); => true (gälte auch für s = "", "abab" oder "ababab" ...)
```

```
re = "."; // ein einzelner Punkt steht für ein beliebiges Zeichen
s.matches(re); => true (gälte auch für s = "xy" oder "69" ...)
```

- Die ausführliche Beschreibung der Syntax regulärer Ausdrücke in Java findet sich unter <http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html#sum>

## Zeichenketten und reguläre Ausdrücke in Java

- Reguläre Ausdrücke sind nicht nur für Compiler nützlich, sondern können viele Formen der Analyse von Zeichenketten/Texten unterstützen.
- Java bietet die besagte Unterstützung für reguläre Ausdrücke, die mit Exemplaren der Klasse `String` arbeitet, seit der Version 1.4 an.
- Neben der Methode `matches` sind in der Klasse `String` weitere Methoden definiert, die mit regulären Ausdrücken arbeiten, u.a.:
  - `String replaceFirst(String regex, String replacement)` – liefert eine neue Zeichenkette als Kopie, in der das erste Vorkommen einer der Zeichenketten, die durch `regex` beschrieben sind, durch `replacement` ersetzt ist;
  - `String replaceAll(String regex, String replacement)` – liefert eine neue Zeichenkette als Kopie, in der alle Vorkommen von Zeichenketten, die durch `regex` beschrieben sind, durch `replacement` ersetzt sind;



<http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html>

SE1 – Level 2

111

## Formale Sprachen



- Jede **formale Sprache** lässt sich verstehen als:
  - eine Menge von **Zeichenketten** eines **Alphabets**  
Gleichbedeutend:
    - eine Menge von **Folgen von Symbolen** eines **Vokabulars** oder **Zeichensatzes**
- **Grammatikregeln** geben an, welche Zeichenketten des Alphabets **Wörter** der Sprache sind, d.h. **syntaktisch korrekt** oder **wohlgeformt** sind.
- Eine Grammatik ist somit eine **Metasprache**, mit der eine andere Sprache beschrieben wird.

Beispiel für ein Alphabet  
(Vokabular, Zeichensatz):  
Die Buchstaben von **a** bis  
**z**.

Beispiel für eine formale Sprache  
über diesem Alphabet:

{**a**, **aha**, **alter**, **aal**, **aabenra**, **aaarghh**, ... }

informell: Die Menge aller Zeichenketten,  
die mit mindestens einem **a** beginnen.  
Wie können wir dies formaler fassen?

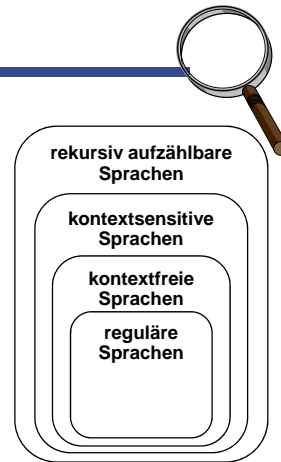
SE1 – Level 2

© Reiser, Wirth S.19f © Sebesta

112

## Grammatiken für Sprachen

- Der Linguist Noam **Chomsky** beschrieb Mitte der 50er Jahre sog. **generative Grammatiken**, um vier Klassen von Sprachen zu definieren:
  - reguläre, kontextfreie, kontextsensitive** und **rekursiv aufzählbare** Sprachen.
  - Reguläre Sprachen bilden die einfachste Klasse – jede höhere enthält die einfacheren.
- Später zeigte sich:
  - Die **Syntax von Programmiersprachen** ist gut als **kontextfreie Sprache** beschreibbar.
  - Die **Token** von Programmiersprachen können als **reguläre Sprachen** beschrieben werden.
- Die uns bereits bekannte **(E)BNF** ist genau so beschreibungsmächtig wie kontextfreie Grammatiken.



- Mehr zur sog. Chomsky-Hierarchie sowie kontextsensitiven und rekursiv aufzählbaren Sprachen in FGI.

## Reguläre Ausdrücke und reguläre Sprachen

- Mit **regulären Ausdrücken** (also einer speziellen Form von Grammatik) können **reguläre Sprachen/Mengen** beschrieben werden.
- Reguläre Ausdrücke über einem Alphabet **A** und der durch sie beschriebenen **regulären Mengen** sind definiert als:
  - a** mit  $a \in A$  ist ein regulärer Ausdruck für die reguläre Menge  $\{a\}$ .
  - Sind **p** und **q** reguläre Ausdrücke für die regulären Mengen **P** und **Q**, dann ist:
    - (p)\*** ein regulärer Ausdruck, der die reguläre Menge **P\*** (Iteration, d.h. beliebig häufige Konkatenation mit sich selbst) bezeichnet,
    - (p+q)** ein regulärer Ausdruck, der die reguläre Menge **P ∪ Q** (Vereinigung) bezeichnet,
    - (pq)** ein regulärer Ausdruck, der die reguläre Menge **P • Q** (Konkatenation) bezeichnet.
  - ∅** ist ein regulärer Ausdruck, der die leere reguläre Menge bezeichnet.
  - ε** ist ein regulärer Ausdruck, der die reguläre Menge  $\{\epsilon\}$  bezeichnet, die nur aus dem leeren Wort **ε** besteht.

## Beispiel eines regulären Ausdrucks



Gegeben sei das **Alphabet**  $\{a,b\}$  und die **reguläre Menge**  $\{aa,ab,ba,bb\}$ .

Diese reguläre Menge wird beschrieben durch die **regulären Ausdrücke**:

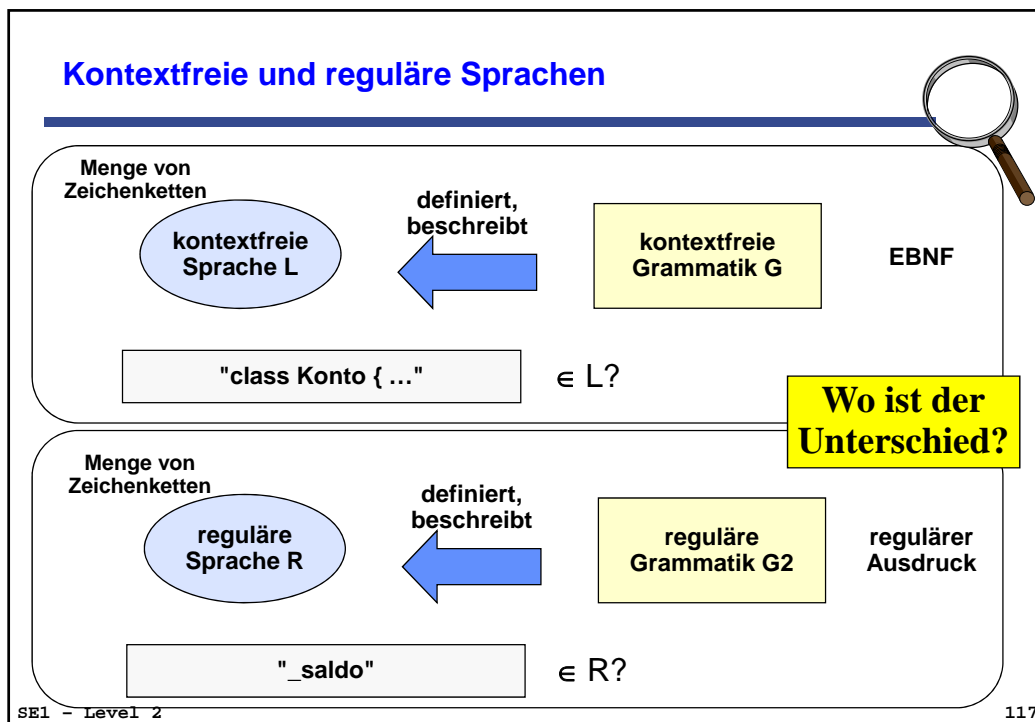
$((aa) + (ab)) + (ba) + (bb)$  bzw.  
 $( (a(a + b)) + (b(a + b)))$  bzw.  
 $((a + b)(a + b)).$



Mehr zu den theoretischen Grundlagen regulärer Ausdrücke in FGI

## Reguläre Ausdrücke in Java: fast wie in der Theorie

- Die pragmatisch in Programmiersprachen eingesetzte Syntax für reguläre Ausdrücke weicht von der Schreibweise, wie sie in der theoretischen Informatik Anwendung findet, teilweise ab.
- Hier die grundlegenden theoretischen Konzepte übersetzt auf die **konkrete Syntax regulärer Ausdrücke** in Java:
  - $(p)^*$  (lies: p beliebig oft) wird in Java genauso notiert, ein  $*$  als Postfix-Operator bedeutet also: der Operand beliebig häufig, auch gar nicht;
  - $(p+q)$  (lies: p oder q) wird notiert als  $p|q$ ;
  - $(pq)$  (lies: p gefolgt von q) wird notiert als  $pq$ .
- Auch in Java können **runde Klammern** zum **einfachen Gruppieren** eingesetzt werden.




### Der Unterschied liegt in der Mächtigkeit

- Kontextfreie Grammatiken sind **beschreibungsmächtiger**; mit ihnen lassen sich beispielsweise **korrekt geklammerte Ausdrücke** (Ausdrücke, die genau so viele schließende wie öffnende Klammern enthalten) beschreiben. **Dies geht nicht mit regulären Ausdrücken!**
- Ein **falscher Versuch** in EBNF:

Expression:

```
{ ( } IntegerLiteral { InfixOp IntegerLiteral } { }
```



- Dieser falsche Versuch kann auch als regulärer Ausdruck formuliert werden.
- Die korrekte Lösung hingegen erfordert eine bestimmte Form der Rekursion, die für reguläre Ausdrücke nicht zugelassen ist:

Expression:

```
( Expression )
...
```

**Warum dann überhaupt reguläre Ausdrücke?**

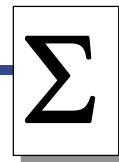
SE1 – Level 2 118

## Reguläre Ausdrücke sind effizient umsetzbar

- Die Syntax einer Programmiersprache ließe sich auch **vollständig** mit einer kontextfreien Grammatik beschreiben.
- Reguläre Ausdrücke werden lediglich aus **Effizienzgründen** für die lexikalische Analyse verwendet.
- Sehr schnelle Erkenner für reguläre Ausdrücke lassen sich automatisiert erstellen; reguläre Ausdrücke werden deshalb beispielsweise von **Suchmaschinen**, **Texteditoren** und auch **Programmiersprachen** wie Java unterstützt.



## Zusammenfassung



- Für die **Darstellung der Syntax** von Programmiersprachen (z.B. in Hand- und Lehrbüchern) haben wir bereits die **EBNF** und **Syntaxdiagramme** kennen gelernt. Sie sind gleichwertig mit kontextfreien Grammatiken.
- Die terminalen Symbole einer EBNF werden auch **Token** genannt. Sie werden häufig mit **regulären Ausdrücken** beschrieben.
- Java bietet seit Version 1.4 eine Unterstützung für reguläre Ausdrücke an.