Informatik II

Für Bachelor of Arts: Translation Studies in Information Technology

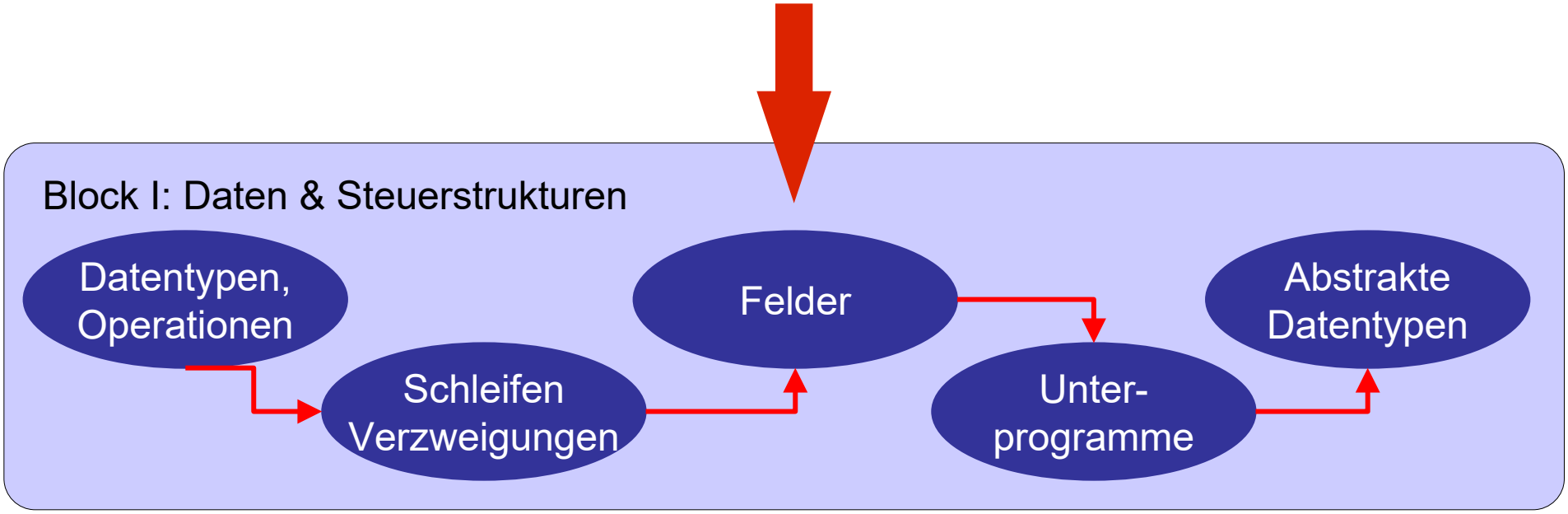
Teil 2 :

Stephan Mechler

Themen von Semester 1/ Wiederholung /
19.03.2024

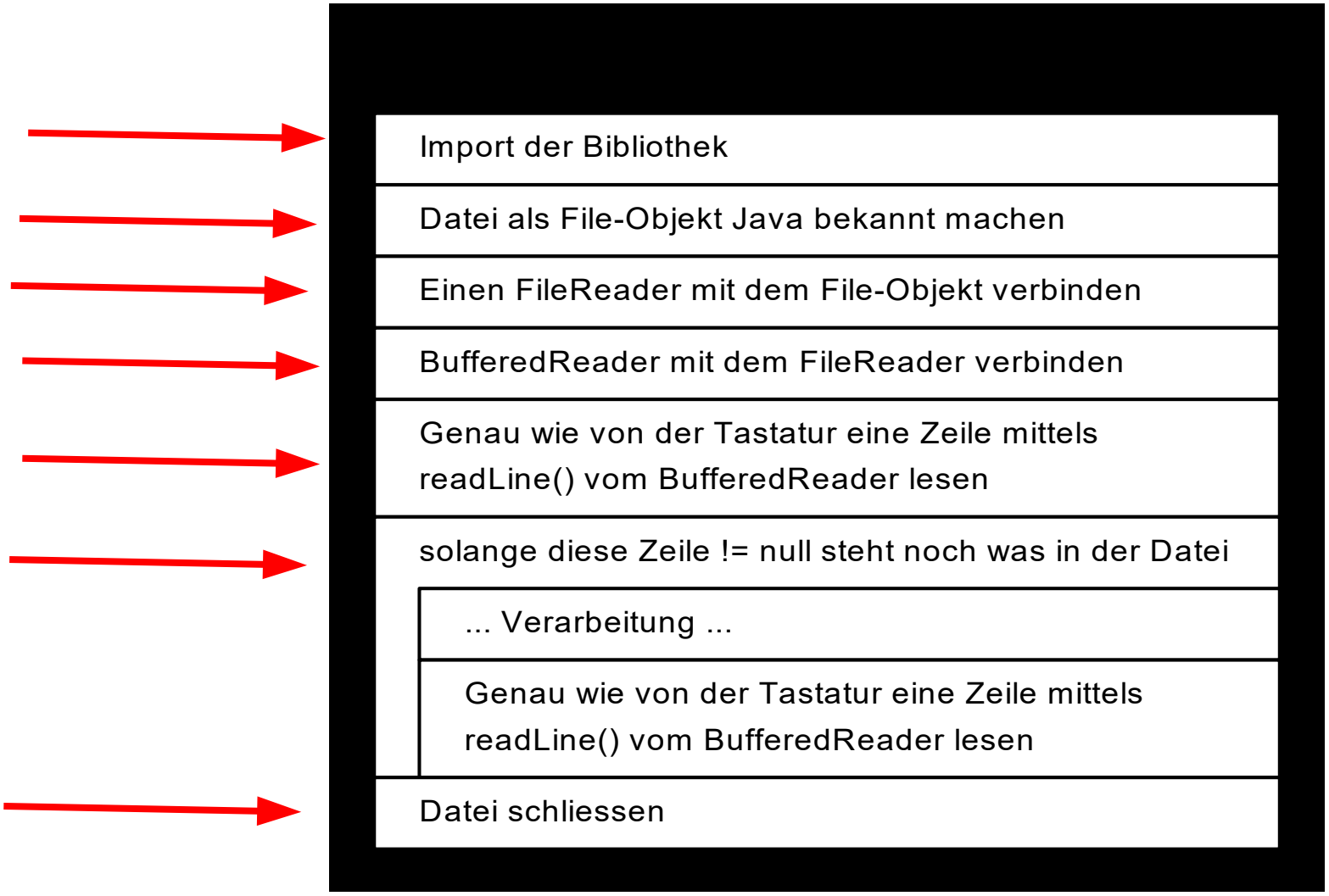
hochschule mannheim

Zwischenstand



Datei Ein-/Ausgabe

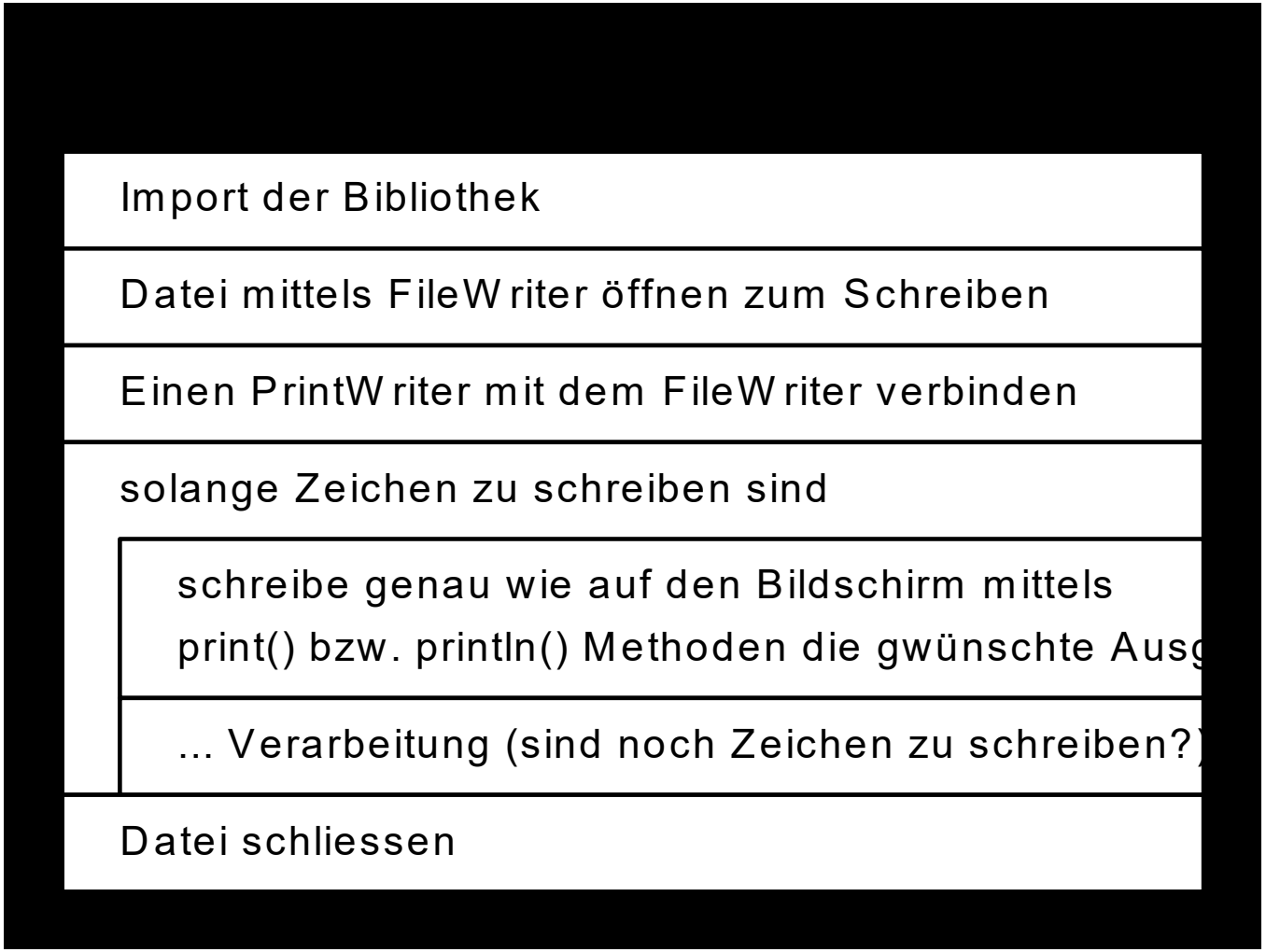
Datei Ein-/Ausgabe – 1



Datei Ein-/Ausgabe – 2

```
import java.io.*;
public class Test {
      public static void main (String[] args) throws IOException {
            File datei = new File("z:\\test.txt");
            FileReader eingabestrom = new FileReader(datei);
            BufferedReader eingabe = new BufferedReader(eingabestrom);

            String  eingabeZeile = eingabe.readLine();
            // Verarbeitungsschleife, solange nicht EOF
            while (eingabeZeile != null) {
                        // Verarbeitung von eingabeZeile genau wie bisher über
                        // die Tastatur (ggf. StringTokenizer, …parse…() etc.)
                        System.out.println(eingabeZeile);
                        eingabeZeile = eingabe.readLine();
            }
            eingabe.close(); // Datei schließen
      }
}
```

Datei Ein-/Ausgabe – 3



Datei Ein-/Ausgabe – 4

```
import java.io.*;
public class Test {
      public static void main (String[] args) throws IOException {
            FileWriter ausgabestrom = new FileWriter("Z:\\out.txt");
            PrintWriter ausgabe = new PrintWriter(ausgabestrom);

            boolean ende = false; // solange false, haben wir was zu schreiben
            while (!ende) {
                        ausgabe.println(…);   // Ausgaben schreiben, exakt wie auf dem
                        ausgabe.print(…);     // Bildschirm mit System.out.print…
                        if ( … )                  // Bedingung prüfen, ob
                                    ende = true;      // Verarbeitung beendet
            }
            ausgabe.close(); // Datei schließen
      }
}
```

Merke!

- Alternative Vorgehensweisen sind möglich: FileInputStream, FileOutputStream, zeichenweises Lesen und Schreiben
- Objekte vom Typ File kennen die Methode exists():
  datei.exists(); // true, wenn die Datei unter dem
  // angegebenen Pfad existiert, false ansonsten
- Wir betrachten hier nur den sequentiellen Zugriff, Java erlaubt aber auch andere Methoden wie z.B. den wahlfreien Dateizugriff
- Die Beispiel-Quellcodefragmente enthalten keinerlei Prüfung auf IO-Fehler – für die Übungen ist dies ebenfalls nicht zwingend nötig

switch-Anweisung

Die Mehrfach-Alternative

- Die Mehrfach-Alternative wird benutzt, um in Abhängigkeit vom Wert eines Ausdrucks direkt zu einer von mehreren möglichen Anweisungen zu verzweigen
- Die Mehrfach-Alternative wird in Java mit switch/case notiert
- Vorteil von switch gegenüber einer if-else-if-else-...-Folge: es müssen nicht alle if-Bedingungen sequentiell durchgetestet werden, bis eine zu true ausgewertet wird, sondern nach der Auswertung des Ausdrucks ist die gewählte Alternative unmittelbar klar (Optimierung durch den Compiler ist **möglich**)
- Nachteil von switch gegenüber einer if-else-if-else-...-Folge: das switch ist weniger mächtig

Syntaxregeln Mehrfach-Alternative – 1

Die Mehrfach-Alternative mit switch ist eine Anweisung

Statement:
- *IfThenStatement*
- *IfThenElseStatement*
- *Block*
- *EmptyStatement*
- *ExpressionStatement*
- *WhileStatement*
- *DoStatement*
- *ForStatement*
- *SwitchStatement*
- ...

SwitchStatement:
- switch (*Expression*) *SwitchBlock*

Syntaxregeln Mehrfach-Alternative – 2

SwitchStatement:
switch (*Expression*) *SwitchBlock*

SwitchBlock:
{ *SwitchBlockStatementGroups*$_{opt}$ *SwitchLabels*$_{opt}$ }

SwitchBlockStatementGroups:
SwitchBlockStatementGroup
SwitchBlockStatementGroups SwitchBlockStatementGroup

SwitchBlockStatementGroup:
SwitchLabels BlockStatements

BlockStatement:
LocalVariableDeclarationStatement
Statement
...

Das ist schon bekannt

SwitchLabels:
SwitchLabel
SwitchLabels SwitchLabel

SwitchLabel:
case *ConstantExpression* :
default :

Ablauf einer switch-Anweisung

Zunächst wird der Ausdruck ausgewertet

```
switch ( <Ausdruck> ) {
    case <KonstanterAusdruck>:
    ...
    case <KonstanterAusdruck>:
    ...
    case <KonstanterAusdruck>:
    default:
}
```

Ist der Wert des Ausdrucks gleich dem Wert von einem dieser konstanten (!) Ausdrücke, ...

```
<Deklaration>
...
<Deklaration>
<Anweisung>
...
<Anweisung>
```

... so wird diese Folge von Deklarationen und Anweisungen ausgeführt

Danach werden weitere cases **ebenso** behandelt (sofern vorhanden)!

```
...
<Deklaration>
...
<Deklaration>
<Anweisung>
...
<Anweisung>
```

Gibt es ein „default“, so werden **auch** dessen Deklarationen und Anweisungen ausgeführt

Regeln für die switch-Anweisung

- Alle Typen der konstanten Ausdrücke müssen zum Typ des switch-Ausdrucks zuweisbar sein
- Keine zwei konstante Ausdrücke dürfen gleich sein
- Der switch-Block darf maximal ein default enthalten
- Prinzipiell werden alle matchenden Label und das default abgearbeitet: die switch-Anweisung ist nicht etwa beendet, wenn ein (erster) passender Ausdruck gefunden wurde

- Bemerkungen
  - Ist der Wert des switch-Ausdrucks gleich einem konstanten Ausdruck, so sagt man „sie matchen“
  - Der default-Fall wird üblicherweise als letzter notiert

Beispiel für eine switch-Anweisung

```
switch ( k ) {
   case 1: System.out.print( "one " );
   case 2: System.out.print( "too " );
   case 3: System.out.println( "many" );
}
```

Beispiel für eine switch-Anweisung

```
switch ( k ) {
  case 1: System.out.print( "one " );
  case 2: System.out.print( "too " );
  case 3: System.out.println( "many" );
}
```

- Läuft die switch-Anweisung mit k == 3, lautet die Ausgabe
  many
- Läuft die switch-Anweisung mit k == 2, lautet die Ausgabe
  too many
- Läuft die switch-Anweisung mit k == 1, lautet die Ausgabe
  one too many

Abbrechen einer switch-Anweisung

- Oft will man keine weitere Alternative durchlaufen, wenn einer der konstanten Ausdrücke matcht
- Nach Abarbeiten der zugehörigen Anweisungsfolge soll der switch-Block verlassen werden
- Das kann durch die Anweisung „break“ am Ende einer Anweisungsfolge erreicht werden
- Bemerkungen
  - Der default-Fall wird nur dann ausgeführt,
    wenn kein anderer Fall matcht oder
    wenn ein Fall matcht, der nicht durch ein break vom default getrennt ist
  - Durch mehrere cases können verschiedene Werte zusammengefasst und durch die gleiche Anweisungsfolge behandelt werden
    Die entsprechenden cases werden dann nacheinander notiert und nur das letzte enthält eine Anweisungsfolge, die durch break abgeschlossen wird

Syntax-Regeln der break-Anweisung

Statement:
- *IfThenStatement*
- *IfThenElseStatement*
- *Block*
- *EmptyStatement*
- *ExpressionStatement*
- *WhileStatement*
- *DoStatement*
- *ForStatement*
- *SwitchStatement*
- *BreakStatement*
- ...

BreakStatement:
break *Identifier*$_{opt}$;

Den optionalen Bezeichner verwenden wir nicht!

Beispiel für eine switch-Anweisung – 1

```
switch ( k ) {
    case 1:  System.out.print( "one " );
             break;
    case 2:  System.out.print( "too " );
             break;
    case 3:  System.out.println( "many" );
}
```

- Läuft die switch-Anweisung mit k == 3, lautet die Ausgabe
  many
- Läuft die switch-Anweisung mit k == 2, lautet die Ausgabe
  too
- Läuft die switch-Anweisung mit k == 1, lautet die Ausgabe
  one

Beispiel für eine switch-Anweisung – 2

Diese Funktion liefert einen Zufallswert im Bereich [0 .. 1[

```
for(int i = 0; i < 100; i++) {
        char c = (char)(Math.random() * 26 + 'a');
        System.out.print(c + ": ");
        switch(c) {
                case 'a':
                case 'e':
                case 'i':
                case 'o':
                case 'u':  System.out.println( "ein Vokal" );
                           break;

                case 'y':
                case 'w':  System.out.println( "in manchen Sprachen ein Vokal" );
                           break;

                default:   System.out.println( "ein Konsonant" );
        }
}
```

Konventionen zur Notation – 1

Es gibt keine einheitliche Konvention, wie switch-Anweisungen notiert werden; mehrere Alternativen sind möglich

- Alle case- und default-Fälle werden gegenüber dem switch um 2 Zeichen (oder 1 Tabulator) eingerückt

  Sind im case oder default Deklarationen oder Anweisungen enthalten, werden diese entweder um weitere 2 Zeichen eingerückt oder auf der Höhe nach dem Doppelpunkt notiert

  Beispiel:

```
<Anweisung>
switch ( <Ausdruck> ) {
    case <KAusdruck>:
        <Anweisung>
        ...
        <Anweisung>
    case <KAusdruck>: <Anweisung>
                      ...
                      <Anweisung>
}
<Anweisung>
```

ODER:

(nicht mischen!)

Konventionen zur Notation – 2

2. Alle case- und default-Fälle werden auf der Höhe des switch notiert

   Sind im case oder default Deklarationen oder Anweisungen enthalten, werden diese entweder um weitere 2 Zeichen eingerückt oder auf der Höhe nach dem Doppelpunkt notiert

   Beispiel:

```
<Anweisung>
switch ( <Ausdruck> ) {
case <KAusdruck>:
    <Anweisung>
    ...
    <Anweisung>
case <KAusdruck>: <Anweisung>
                  ...
                  <Anweisung>
}
<Anweisung>
```

ODER:

(nicht mischen!)

??? Frage

Welche Fragen gibt es?

mehrdimensionale Arrays

Mehrdimensionale Arrays: Matrizen

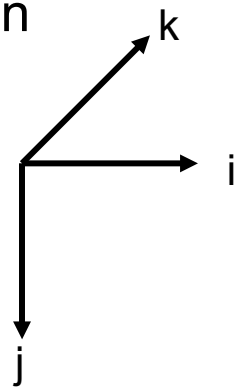
- In der Mathematik und anderen Disziplinen besteht häufig Bedarf an mehrdimensionalen Arrays (also nicht nur Vektoren)
- Matrizen sind ein gebräuchliches Beispiel
- Allgemeine Form einer Matrix:

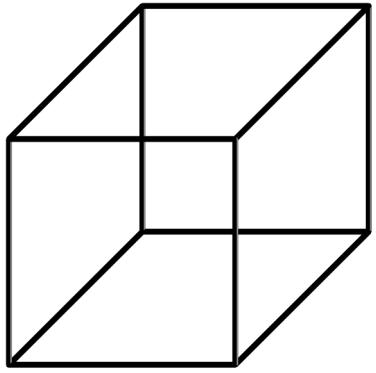
$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

- Geschickte Namensgebung für Java-Programme
  - Konstanten M und N für die maximale Zeilen- und Spaltenzahl
  - Indizes i und j zum Bezeichnen einzelner Elemente, zum Beispiel a_{ij}

Mehrdimensionale Arrays: Mehrdimensionale Gebilde

- Auch drei- oder mehrdimensionale Gebilde bzw. deren Elemente können dargestellt werden
- Elemente eines Würfels können beispielsweise mit a_{ijk} indiziert werden





- Geschickte Namensgebung für Java-Programme
  - Konstanten X, Y und Z für die maximale Zeilen-, Spalten- und Schichtenzahl
  - Indizes i, j und k zum Bezeichnen einzelner Elemente, zum Beispiel a_{ijk}

Beispiel für eine Matrix: Entfernungstabelle

nach / von	Aachen	Augsburg	Baden-Baden	Berlin	Bielefeld	Bonn
Aachen	0	593	409	633	257	91
Augsburg	593	0	258	586	600	502
Baden-Baden	409	258	0	723	522	318
Berlin	633	586	723	0	390	598
Bielefeld	257	600	522	390	0	228
Bonn	91	502	318	598	228	0

Element 3, 4

Beispiel für eine Matrix: Pixel auf einem Schwarz/Weiß-Monitor

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	…	0,1400
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	...	1,1400
2,0	2,1									
3,0										
4,0										
5,0										
6,0										
7,0										
8,0										
...										
1050,0	1050,1	1050,2							...	1050,1400

Angegeben sind die jeweiligen Positionsnummern
Jede Position könnte z.B. einen Boole'schen Wert aufnehmen, der weiß (true) oder schwarz (false) speichert

Deklaration

- Arrays in Java sind immer eindimensional,
  sie können aber geschachtelt werden

- Deklaration
  - <Typ> <Variable1> [][] ... [], <Variable2>, <Variable3> [][] ... [];
    <Typ> [][] ... [] <Variable4>, <Variable5>;
  - int ai[][] ... []; → array of int
    char [][] ... [] as1, as2; → as1 und as2 sind array of char
    float f1[][] ... [], f2, f3[][] ... []; → f1 und f3 sind Arrays,
    f2 ist ein Skalar
  - Es sind „beliebig“ viele Dimensionen zulässig
  - Die Anzahl der Klammerpaare zeigt die Tiefe der Schachtelung an

Initialisierung und Wertzuweisung

- Initialisierung
  - <Typ> [][] ... [] <Variable> = { {...{{...}, {...},...{...}}...} };
  - int [][] matrix = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
  - Die Tiefe der Schachtelung von Deklaration und Initialisierungsausdruck muss übereinstimmen

- Nachträgliches Anlegen
  - <Typ> [][] ... [] <Variable>;
<Variable> = new <Typ>[<Wert1>][<Wert2>]...[<Wert n>];
  - int [][] matrix;
matrix = new int[M][N];
  - Die Anzahl der Klammern muss mit der Schachtelungstiefe des Arrays übereinstimmen

Zugriff auf Komponenten

- <Variable>[<Index1>][<Index2>] ... [<Index n>]
- matrix[i][j]
  Nicht: matrix[i, j] → geschachtelte Arrays!
- Stimmt die Anzahl der Indexklammern mit der Schachtelungstiefe des Arrays überein, so wird als Ergebnis ein Wert vom Komponententyp des Arrays geliefert
- Ist die Anzahl der Indexklammern geringer als die Schachtelungstiefe des Arrays, so wird als Ergebnis ein (mehrdimensionales) Array geliefert, dessen Schachtelungstiefe der Anzahl der fehlenden Klammern entspricht und das den selben Komponententyp wie das ursprüngliche Array hat

  Achtung:
  Vorerst wollen wir solche Teil-Arrays nur auf der rechten Seite von Zuweisungen benutzen
- Ist die Anzahl der Indexklammern größer als die Schachtelungstiefe des Arrays, so meldet der Compiler einen Fehler

Wertzuweisung

- Wie schon bei eindimensionalen Arrays ist die direkte Zuweisung zwar möglich, soll aber zunächst noch vermieden werden
- Alternative
  Elementweises Kopieren der Teil-Arrays
- **Beispiel**

```
float [][][] src = new float[Z][Y][X], dst = new float[Z][Y][X];
int i, j;
for ( i=0; i<Z; i++ )
  for ( j=0; j<Y; j++ )
        System.arraycopy( src[i][j], 0,dst[i][j],0, dst[i][j].length  );
```

Exkurs: Arrays verschiedener Dimensionslänge – 1

- Zum Arraytyp gehört in Java nur der Komponententyp, nicht die Länge der Dimension
- Daher sind unterschiedlich lange Teil-Arrays möglich

- **Beispiel:** Pascal‘sches Dreieck
  int [][] pascal = { { 1 }, { 1, 1 }, { 1, 2, 1 } };
- Erlaubt sind hier nur die folgenden Zugriffe
  pascal[0][0],
  pascal[1][0], pascal[1][1],
  pascal[2][0], pascal[2][1], pascal[2][2]

- Zur Erinnerung:
  Die Länge der einzelnen Teil-Arrays lässt sich mittels “.length“ abfragen; Beispiel:
  pascal[1].length $\rightarrow$ 2
  pascal[2].length $\rightarrow$ 3

Exkurs: Arrays verschiedener Dimensionslänge – 2

- Gegeben sei die folgende Vereinbarung
  ```
  int [][][][] a1, a2;
  ```

- Bekannt ist bereits das folgende Anlegen eines der Arrays
  ```
  a1 = new int[3][3][2][4];
  ```

- Von rechts her dürfen Dimensionen ausgelassen werden, es müssen jedoch alle Schachtelungsebenen aufgeführt werden
  ```
  a2 = new int[3][3][][];
  ```

- Die Länge der fehlenden Dimension kann nachträglich angegeben werden:
  ```
  a2[1][0] = new int[2][4];
  a2[1][1] = new int[5][];
  ```

Exkurs: Arrays verschiedener Dimensionslänge – 3

- Gegeben sei die folgende Vereinbarung

  int [][][][] a;

- Es ist nicht erlaubt, Dimensionslängen anzugeben, wenn übergeordnete Dimensionslängen ausgelassen wurden

  a = new int[3][3][][2]; → Compilerfehler

- Es ist nicht erlaubt, Dimensionen auszulassen

  a = new int[3][4][]; → Compilerfehler

??? Fragen

Welche Fragen gibt es?

Klassen als Datenstruktur

Motivation

- Wir haben Arrays kennen gelernt, die benutzt werden, um eine Folge gleichartiger Elemente zu speichern

  Beispiele: Namen (Strings), Postleitzahlen (int), Matrizenwerte (float)
- Oft ist es jedoch notwendig, eine Menge nicht-gleichartiger Elemente zusammen zu speichern
- **Beispiele:**

  Ein Auto ist charakterisiert durch
  - Das Fabrikat (String)
  - Das Baujahr (int)
  - Die Leistung in kw (float)

  Ein Buch kann beschrieben werden durch
  - Den Namen des Autors (String)
  - Den Titel (String)
  - Den Preis (float)
  - Das Erscheinungsjahr (int)
  - Evtl. weitere Angaben

Klassen und Felder – 1

- Java erlaubt es, so genannte Klassen zu deklarieren, um derartige Ansammlungen von Werten verschiedener Typen zusammenzufassen
- Innerhalb einer Klasse können eine Reihe so genannte Felder (oder: Attribute, englisch: fields) vereinbart werden, welche die verschiedenen Werte aufnehmen können
- **Beispiel**

```
class Book {
        String author;
        String title;
        float price;
        int yearOfPublication;
}
```

Merke!

Merkregeln:

Eine Klasse = Eine Datei

Name Klasse = Name .java Datei

Klassen und Felder – 2

- Klassen kennen wir bereits:
  Unsere bisherigen Programme bestehen aus jeweils einer Klasse
- Die Feld-Vereinbarungen in Klassen entsprechen unseren bereits bekannten globalen Variablen, allerdings ohne das Schlüsselwort "static"

```
class Auto {
    String    fabrikat;
    int       baujahr;
    float     leistung;
}
```

- Die globalen Variablen und die Konstanten, die wir bereits kennen, sind ebenfalls Felder, wir haben sie bisher nur nicht so bezeichnet

Klassen und Objekte – 1

- Klassen-Deklarationen sind benutzerdefinierte (Daten-) Typ-Vereinbarungen
- Benutzerdefinierte Datentypen sind zu verstehen als Gegenstück zu den vordefinierten Datentypen wie z.B. int, float, char und boolean
- Benutzerdefinierte Datentypen können, sobald sie einmal deklariert sind, wie vordefinierte Datentypen verwendet werden, um Variablen dieses Typs zu deklarieren

```
Book myBook;
```

- Die Werte, die eine Klasse annehmen kann, bezeichnet man als Objekte oder Instanzen
- Daher spricht man von der objektorientierten Programmierung (OOP) und von objektorientierten Programmiersprachen (wie zum Beispiel Java)

Klassen und Objekte – 2

- Man sagt, Objekte werden aus Klassen abgeleitet oder instanziiert
- Die Klasse ist ein Muster (oder: Schema, englisch: Template), nach dem alle davon abgeleiteten Objekte angelegt werden
- Alle Felder, die eine Klasse definiert, sind in allen von dieser Klasse abgeleiteten Objekten vertreten

  Beispiel

  - Es kommt nicht vor, dass einem Buch das Feld für den Titel fehlt
  - Der Titel muss aber nicht notwendigerweise mit einem (sinnvollen) Wert belegt sein

Klassen und Objekte – 3

- Objekte, also Werte von Klassen-Typen, werden, ähnlich wie Arrays, durch einen new-Konstruktor gebildet (oder abgeleitet, instanziiert)

```
myBook = new Book();
```

Es werden aber runde Klammern geschrieben und natürlich fehlt die Angabe der Elementanzahl, die bei Arrays angegeben wird

- Diese Art der Zuweisung kann beliebig oft wiederholt werden, sie wird auch für die Initialisierung bei der Deklaration eines Objekts verwendet

```
Book myOtherBook = new Book();
```

Zugriff auf Felder

- Auf die Felder eines Objekts greift man mittels Punkt zu
  Formal:

  <Objektname>.<Feldname>

  Konkret:

  myBook.author
  myBook.title
  myBook.price

- Der Sichtbarkeits- oder Gültigkeitsbereich eines Feldes ist die Klasse (wie bei den bereits bekannten Variablen und Konstanten)
- Alle Feldnamen einer Klasse müssen voneinander verschieden sein (wie globale Variablen und Konstanten)
- Für Feldnamen gelten die gleichen Namenskonventionen wie für globale Variablen

Beispiel – 1

- Es sollen eine Reihe von Büchern verwaltet werden, zum Beispiel für eine Bibliothek

- Die vollständige Deklaration dazu kann wie folgt aussehen

```
class Book {
        String    author, title;      // abkürzende Schreibweise
        int       yearOfPublication;
        boolean   present;
        int       inventaryNumber;
}

static final int N = 100;
...         // in main:
Book [] books = new Book[N]; // hier wird bereits das Array angelegt
```

Beispiel – 2

- Die Initialisierung kann wie folgt in einer Schleife erfolgen

```
...
for ( int i=0; i<N; i++ ) {
  books[i] = new Book();   // hier wird jeweils ein Buchobjekt angelegt
  System.out.print( "Geben Sie den Autor ein: " );
  books[i].author = ... ; // Einlesen über die Tastatur
  System.out.print( "Geben Sie den Titel ein: " );
  books[i].title = ... ; // Einlesen über die Tastatur
  System.out.print( "Geben Sie das Erscheinungsjahr ein: " );
  books[i].yearOfPublication = ... ; // Einlesen über die Tastatur
  books[i].present = true;  // noch nicht verliehen
  books[i].inventoryNumber = i;      // diese Nummer ist eindeutig
}
...
```

- Man könnte die Daten aller belegten Einträge in eine Datei schreiben und daraus auslesen, sobald sie zum ersten Mal eingegeben sind

Default-Initialisierung – 1

- Ist eine lokale Variable vom Typ einer Klasse innerhalb eines Unterprogramms nicht initialisiert, dann meldet der Compiler einen Fehler beim Versuch, lesend darauf zuzugreifen

  Beispiel

```
static void Test () {
    Book myBook;
    System.out.println( myBook.author );
                                        → „variable myBook might not have
                                        been initialized at line ...“
    ...
```

- Ist ein Feld (eine globale Variable) vom Typ einer Klasse innerhalb eines Unterprogramms nicht initialisiert, dann wird ihm der Default-Wert “null“ zugewiesen

  Beispiel

```
static Book myGlobalBook;
static void Test () {
    Book myBook;
    System.out.println( myGlobalBook );      → null
    ...
```

Default-Initialisierung – 2

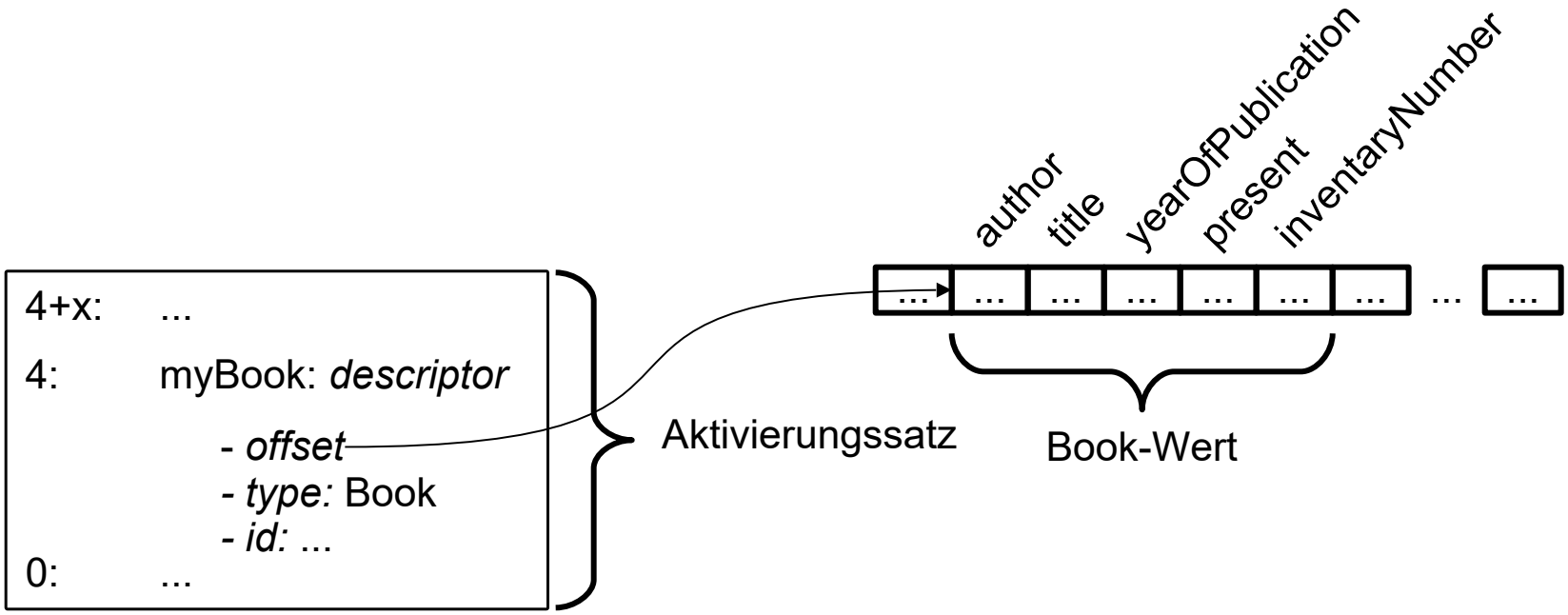
- Die Zuweisung eines Default-Wertes an Felder gilt nicht nur für Objekte, sondern auch für nicht-initialisierte Variablen aller anderen Datentypen

Elementarer Typ	**Default-Wert**
boolean	false
char	‘\u0000’ (null)
byte	(byte) 0
short	(short) 0
int	0
long	0L
float	0.0f
double	0.0d
<Klassentyp>	null
<Arraytyp>	null
String	null

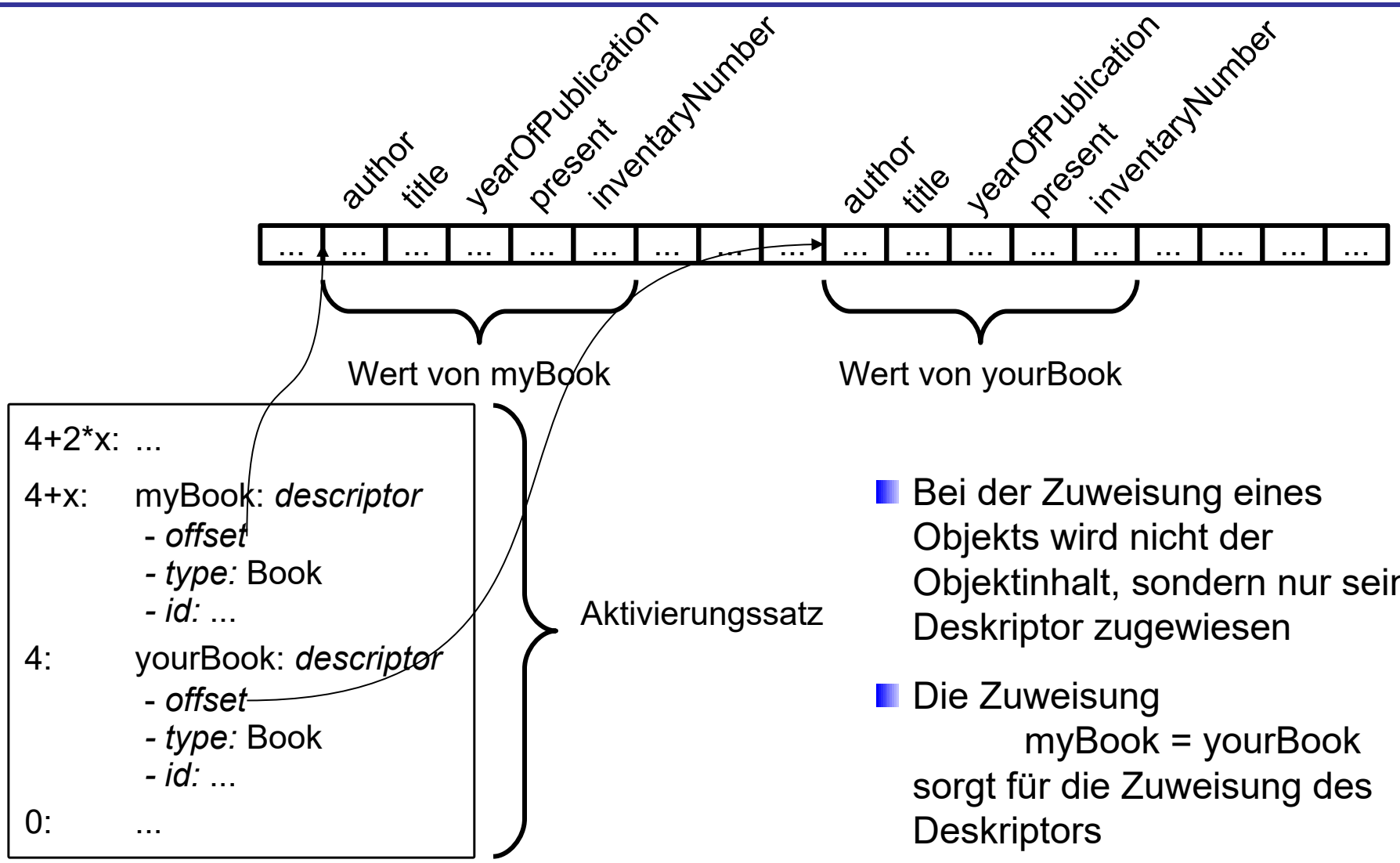
Verwaltung von Objekten

- Objekte werden, ebenso wie Arrays, mittels Deskriptoren verwaltet
- Allerdings ist im Deskriptor für Objekte nicht der Offset und die Länge des Arrays interessant, sondern
  - der Offset des Objekts,
  - der Typ des Objekts (also seine Klasse) und
  - ein eindeutiges Objektkennzeichen

  Die relativen Offsets der Felder darin ergeben sich aus deren Typen



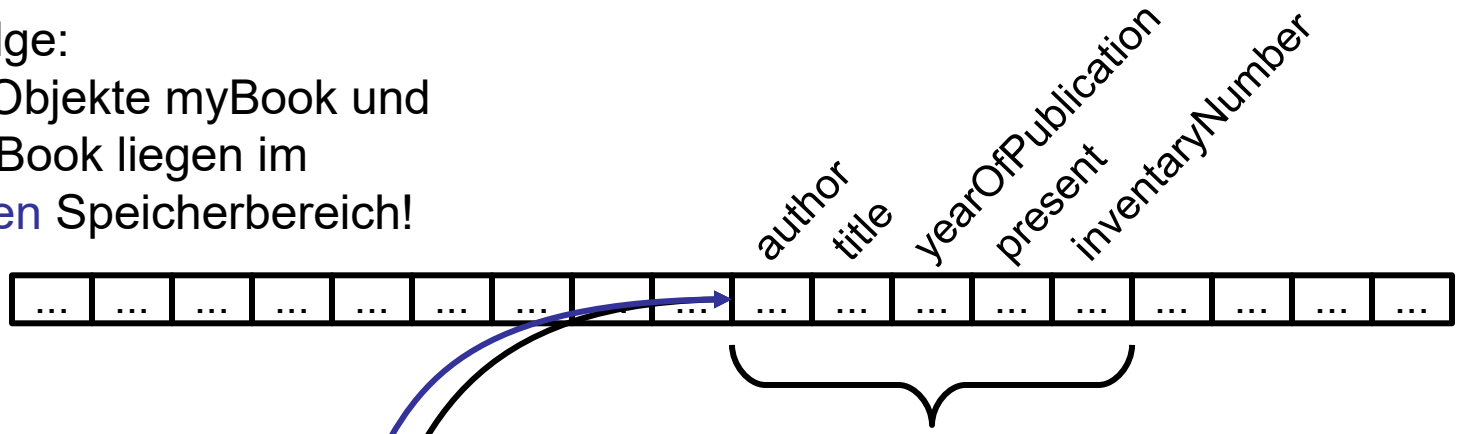
Zuweisung ganzer Objekte – 1



- Bei der Zuweisung eines Objekts wird nicht der Objektinhalt, sondern nur sein Deskriptor zugewiesen
- Die Zuweisung
  myBook = yourBook
  sorgt für die Zuweisung des Deskriptors

Zuweisung ganzer Objekte – 1

Die Folge:
Die Objekte myBook und yourBook liegen im selben Speicherbereich!

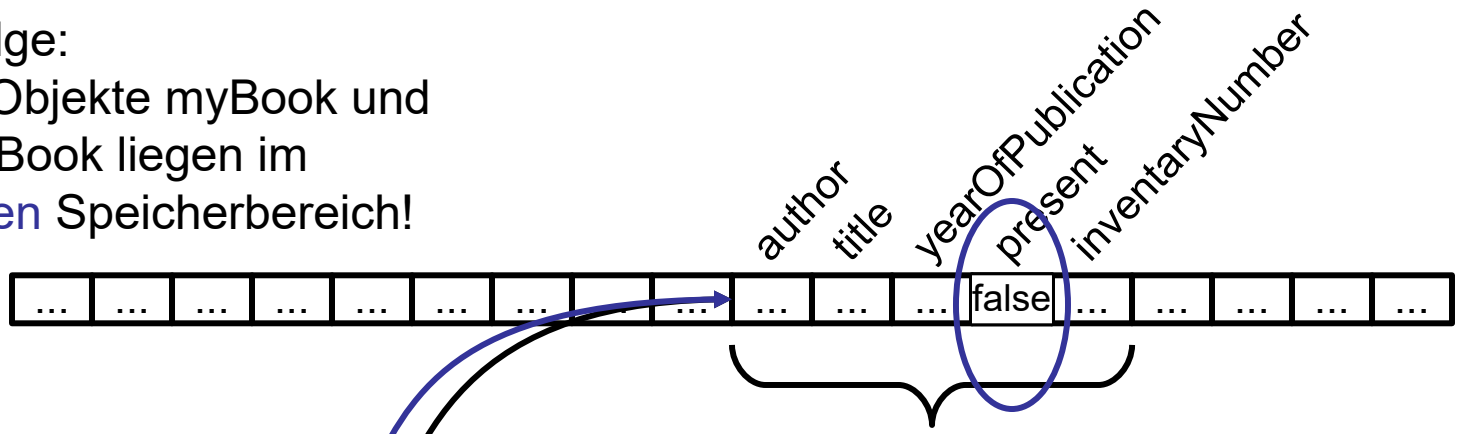



```
4+2*x:  ...
4+x:    myBook: descriptor
          - offset
          - type: Book
          - id: ...
4:      yourBook: descriptor
          - offset
          - type: Book
          - id: ...
0:      ...
```

- Bei der Zuweisung eines Objekts wird nicht der Objektinhalt, sondern nur sein Deskriptor zugewiesen
- Die Zuweisung
  myBook = yourBook
  sorgt für die Zuweisung des Deskriptors

Zuweisung ganzer Objekte – 1

Die Folge:
Die Objekte myBook und yourBook liegen im selben Speicherbereich!




```
4+2*x:  ...
4+x:    myBook: descriptor
          - offset
          - type: Book
          - id: ...
4:      yourBook: descriptor
          - offset
          - type: Book
          - id: ...
0:      ...
```

Beispiel: Die Zuweisung

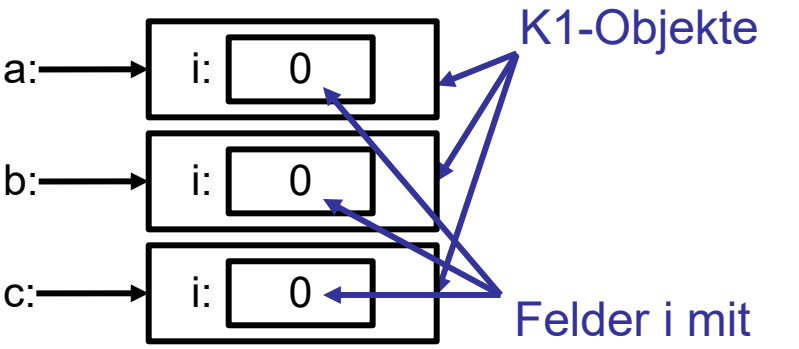
```
yourBook.present = false;
```

bewirkt

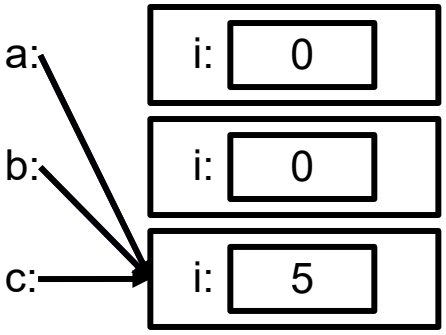
```
myBook.present == false
```

Zuweisung ganzer Objekte – 2

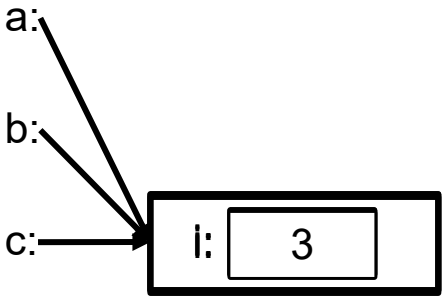
```
class K1 { int  i; }

...

K1      a = new K1(),
        b = new K1(),
        c = new K1();
```




```
c.i = 5;
a = b = c;
```

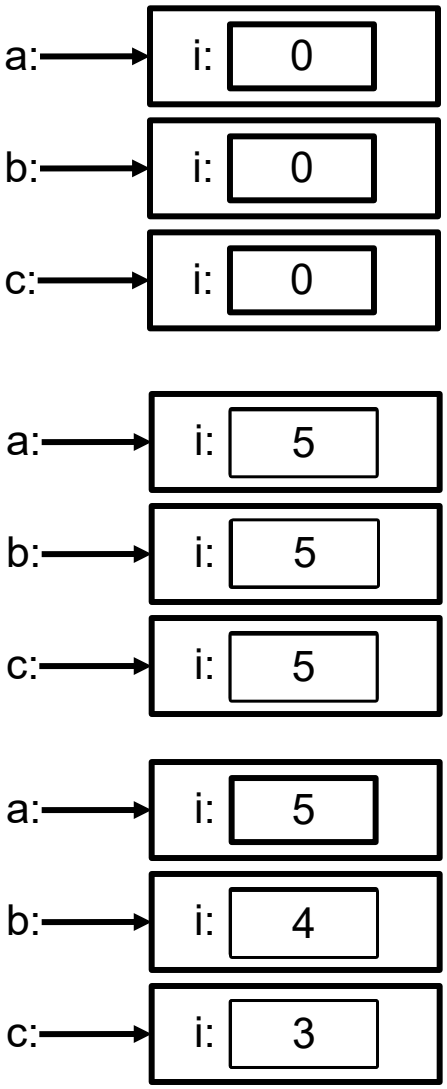



```
b.i = 4;        Folge: a.i == c.i == 4
c.i = 3;        Folge: a.i == b.i == 3
```



Zuweisung einfacher Werte in Objekten

```
class K1 { int  i; }

...

K1          a = new K1(),
            b = new K1(),
            c = new K1();

c.i = 5;

a.i = b.i = c.i;

b.i = 4;

c.i = 3;
```



Semantik Zuweisung von Objekten

- Wie für Arrays gilt auch für Objekte bei der Zuweisung Referenz-Semantik:
  Zugewiesen werden Referenzen auf die jeweiligen Objekte

- Referenzieren zwei (oder mehr) Objekte den gleichen Speicherbereich, spricht man von Aliasierung

- Für Felder in einem Objekt, die einen einfachen Typ haben, gilt Werte-Semantik

Geschachtelte Objekte

Achtung:

- Objekte können Felder enthalten, die einen nicht-einfachen Typ haben, also zum Beispiel ein Array oder Objekte einer Klasse

  Man spricht dann von geschachtelten Objekten
- Ebenso können Arrays als Elementtyp einen nicht-einfachen Typ haben, also ein weiteres Array oder Objekte einer Klasse

  Zweidimensionale Arrays kennen wir bereits
- Für solche geschachtelten Strukturen gilt (rekursiv) Referenz-Semantik bei der Zuweisung

Beispiel für ein geschachteltes Objekt – 1

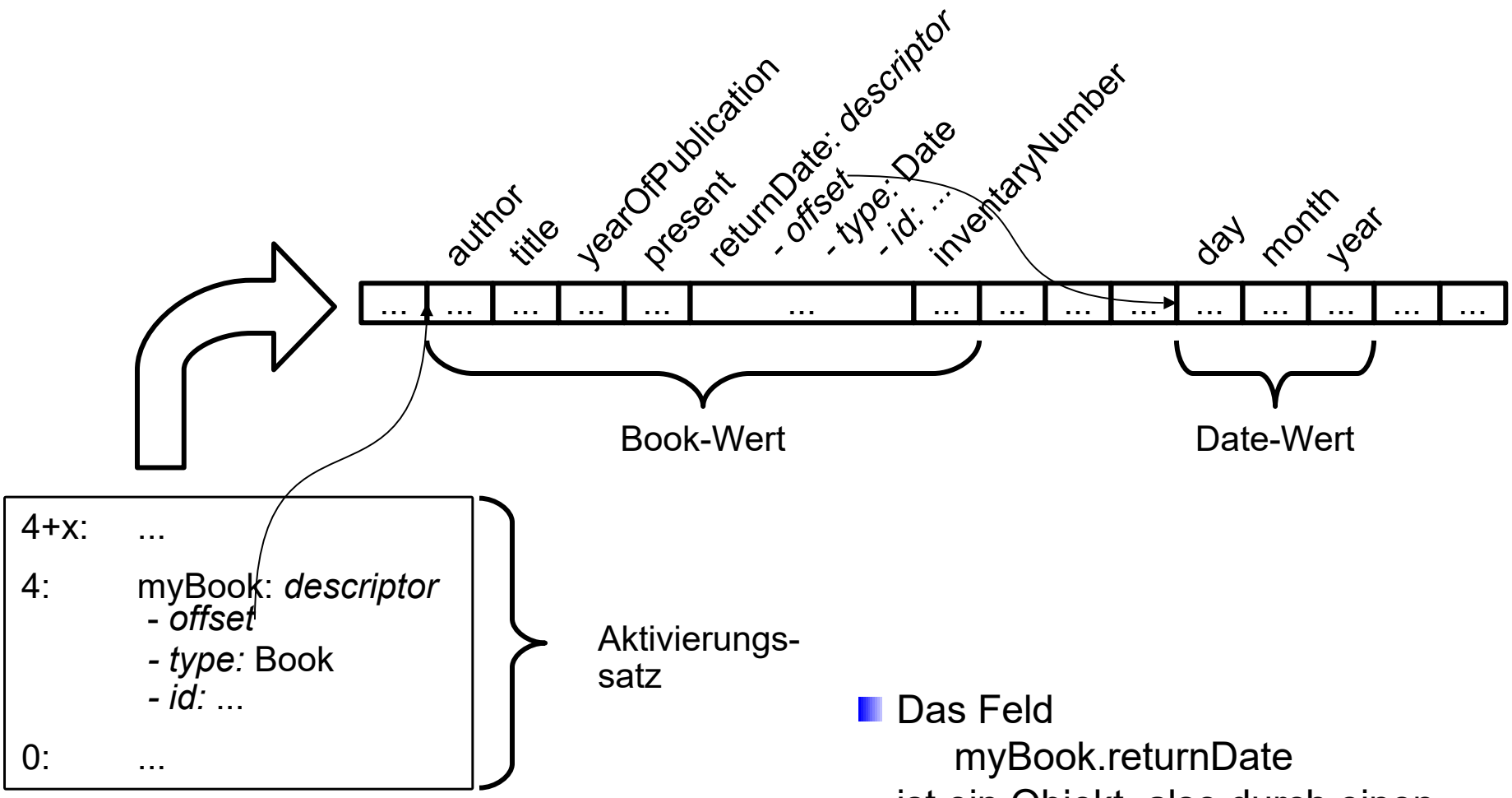
- Wir können der Buch-Klasse ein weiteres Feld hinzufügen für das Datum, an dem es wieder zurückgebracht werden wird, sofern es nicht präsent ist
- Die Deklaration für ein Datum kann wie folgt aussehen

```
class Date {
        int     day;
        int     month;
        int     year;
}
```

- Die erweiterte Buch-Klasse sieht dann wie folgt aus

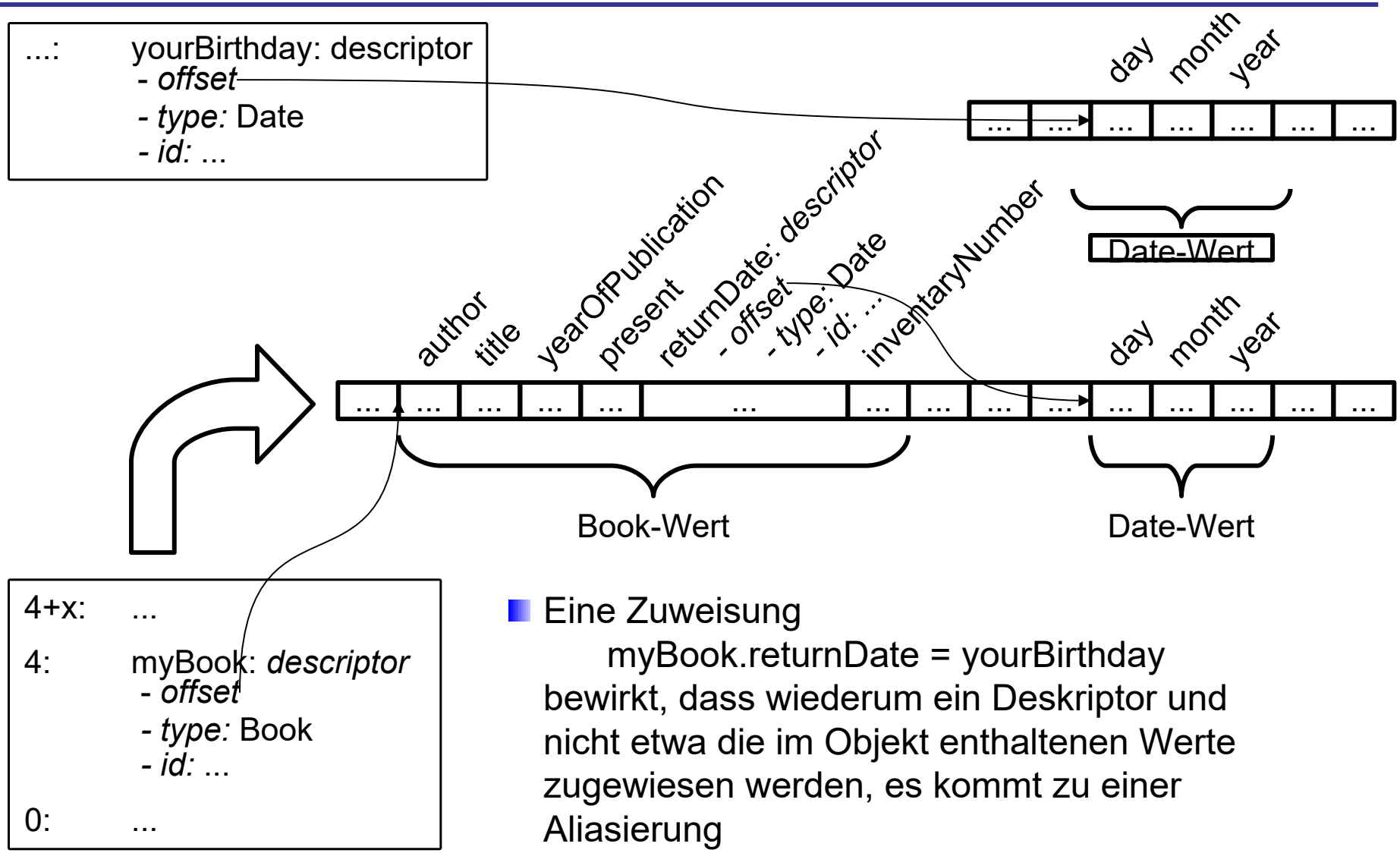
```
class Book {
        String  author;
        String  title;
        int     yearOfPublication;
        boolean present;
        Date    returnDate;
        int     inventaryNumber;
}
```

Beispiel für ein geschachteltes Objekt – 2



- Das Feld
  myBook.returnDate
  ist ein Objekt, also durch einen Deskriptor realisiert

Beispiel für ein geschachteltes Objekt – 3

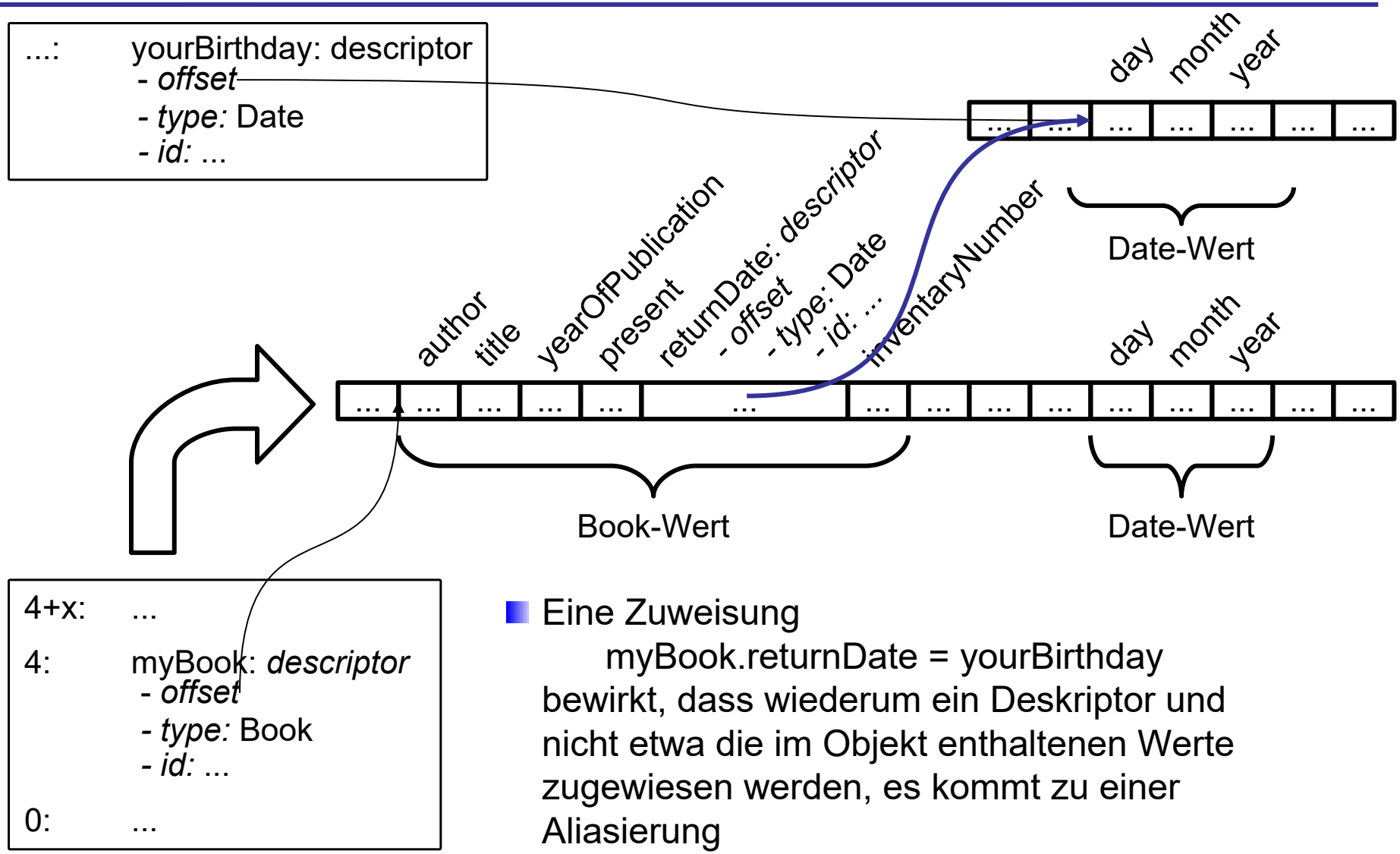


- Eine Zuweisung

  myBook.returnDate = yourBirthday

  bewirkt, dass wiederum ein Deskriptor und nicht etwa die im Objekt enthaltenen Werte zugewiesen werden, es kommt zu einer Aliasierung

Beispiel für ein geschachteltes Objekt – 4



- Eine Zuweisung

  myBook.returnDate = yourBirthday

  bewirkt, dass wiederum ein Deskriptor und nicht etwa die im Objekt enthaltenen Werte zugewiesen werden, es kommt zu einer Aliasierung

Bekannte Objekte: Arrays

- Wir wissen, dass Objekte und Arrays eigentlich durch Deskriptoren dargestellt werden
- Tatsächlich sind in Java Arrays eigentlich (besondere) Objekte

  Sie bieten nur eine besondere Schreibweise, nämlich die Indizierung mittels eckiger Klammern, um auf ihre Elemente zuzugreifen

Bekannte Objekte: Strings

- Auch Strings sind in Java (besondere) Objekte, sie werden ebenfalls durch Deskriptoren dargestellt
- Auch sie bieten eine besondere Schreibweise, nämlich die Notation in doppelten Hochkommata
- Dagegen ist der (schreibende) Zugriff auf einzelne Elemente eines Strings (also einzelne Zeichen) nicht möglich
- Strings und Arrays sind in Java nur deshalb besonders flexibel, weil sie als Objekte realisiert sind
- Hier unterscheidet sich Java deutlich von vielen anderen Programmiersprachen

Nachtrag: Umwandlung String ↔ char [ ]

- Strings und Character-Arrays können einfach ineinander überführt werden
- Die Umwandlung Character-Array → String erfolgt mittels Konstruktor

```
char [] ca = { 'T', 'e', 'x', 't' };
String s = new String( ca );
```

- Die Umwandlung String → Character-Array erfolgt mittels einer Methode der Klasse String

```
String s = "text";
char [] ca;
ca = s.toCharArray();
ca = "dies ist auch ein String".toCharArray();
```

Benutzerdefinierte Konstruktoren – 1

- Wir haben die Erzeugung von Objekten mittels new-Konstruktor kennengelernt

  Beispiel

  ```
  Book myBook;
  myBook = new Book();
  ```

- Korrekt gesagt, ist nicht “new“ der Konstruktor, sondern der Instanziierungs-Ausdruck besteht aus
  - dem Schlüsselwort „new“ plus
  - dem Namen der zu instanziierenden Klasse mit den runden Klammern

Benutzerdefinierte Konstruktoren – 2

- Beim Aufruf des Konstruktors werden alle Felder des neuen Objekts der Klasse Book mit Default-Werten versehen
- Im Beispiel sind das die folgenden Werte

```
class Book {
   String     author;                ← null
   String     title;                 ← null
   int        yearOfPublication;     ← 0
   boolean    present;               ← false
   Date       returnDate;            ← null
   int        inventaryNumber;       ← 0
}
```

Benutzerdefinierte Konstruktoren – 3

- Alternativ ist es möglich, die Default-Werte durch eigene Initialwerte zu ersetzen
- Im Beispiel sind das die folgenden Werte

```
class Book {
    String      author                  = "";
    String      title                   = "";
    int         yearOfPublication;                  ← 0
    boolean     present                 = true;
    Date        returnDate;                         ← null
    int         inventaryNumber;                    ← 0
}
```

Benutzerdefinierte Konstruktoren – 4

- Alternativ ist es auch möglich, bei jeder Instanziierung einer Klasse andere Initialwerte vorzugeben
- Dazu werden eigene Konstruktoren definiert
- Diese können dann mit Parametern versorgt werden, welche für die (jedes Mal unterschiedliche) Initialisierung der Felder der Klasse genutzt werden

Benutzerdefinierte Konstruktoren – 5

- Konstruktoren werden ähnlich definiert wie Methoden, aber mit folgenden Unterschieden
  - Der Name des Konstruktors ist festgelegt:
    Der Konstruktor heißt immer exakt so wie die Klasse, die er instanziieren soll

    Beispiel:
    Heißt die Klasse „Book“, dann heißt auch der Konstruktor „Book“
  - Es wird kein Ergebnistyp angegeben:
    Der Konstruktor liefert immer ein Objekt seiner Klasse als Ergebnis, daher kann die Angabe entfallen

    Beispiel:
    Der Konstruktor der Klasse „Book“ liefert immer ein Objekt der Klasse „Book“ als Ergebnis
  - Ein Konstruktor enthält keine return-Anweisung:
    Beim Konstruktor-Aufruf wird Speicherplatz für das neue Objekt angelegt; Anweisungen im Rumpf des Konstruktors beziehen sich auf die Felder des neu angelegten Objekts, dieses Objekt wird dann als Ergebnis des Aufrufs geliefert

Beispiel Konstruktor-Definition

```
class Book {
    String     author = "";
    String     title = "";
    int        yearOfPublication;
    boolean    present = true;
    Date       returnDate;
    int        inventaryNumber;

    Book () {
        author = "unknown";
        yearOfPublication = 1998;
        returnDate = new Date();
    }
}
```

- Dieser Konstruktor
  - Überschreibt den explizit gesetzten Default-Wert "" für das Feld author,
  - Überschreibt den impliziten Default-Wert 0 für das Feld yearOfPublication,
  - Belegt das Attribut returnDate mit einem neuen Objekt (unbekannten Inhalts)

Konstruktoren mit Parametern – 1

- Wie Methoden können Konstruktoren mit formalen Parametern versehen werden

  Beispiel

```
Book ( String theAuthor, String theTitle, int theYear, int theNumber ) {
    author                  = theAuthor;
    title                   = theTitle;
    yearOfPublication       = theYear;
    inventoryNumber         = theNumber;
}
```

- Der Aufruf dieses Konstruktors erfolgt wiederum mit new und dem Klassennamen, nur werden diesmal aktuelle Parameter übergeben

  Beispiel

```
myBook = new Book( "Henning Mankell", "Mittsommermord",
                                      1997, 512 );
```

Konstruktoren mit Parametern – 2

- Konstruktoren können auch Berechnungen vornehmen

- **Beispiel**
  Basierend auf einer globalen Variable ermittelt der folgende Konstruktor die nächste freie Inventarnummer und vergibt sie an das aktuelle Buch

```
class Book {

    // Vereinbarungen der Felder
    static int freeInventaryNumber = 0;

    Book ( String theAuthor, String theTitle, int theYear ) {
            author  = theAuthor;
            title   = theTitle;
            yearOfPublication = theYear;
            freeInventaryNumber++;
            inventaryNumber   = freeInventaryNumber;
    }

}
```

Merke!

- Abgesehen von den genannten Einschränkungen können Konstruktoren alle Aufgaben übernehmen, die auch Methoden übernehmen können

  Sie sollten aber nach Möglichkeit ausschließlich für die Initialisierung von Feldern der Klasse benutzt werden
- Zu den sinnvollen Aufgaben eines Konstruktors gehört zum Beispiel die Initialisierung geschachtelter Strukturen wie im Beispiel das Datum

- In der Praxis kommt es darauf an, sinnvolle Initialisierungswerte für Felder vorzusehen und im Konstruktor nur Parameter für diejenigen Felder vorzusehen, die erst bei der Instanziierung bekannt sind

??? Fragen

Welche Fragen gibt es?