

Objekte und Klassen



Motivation

Wie würde man ein Datum speichern (z.B. 13. November 2002)?

3 Variablen

```
int day;  
String month;  
int year;
```

Unbequem, wenn man mehrere Exemplare davon braucht:

```
int day1;  
String month1;  
int year1;  
int day2;  
String month2;  
int year2;  
...
```

Idee: die 3 Variablen zu einem eigenen Datentyp zusammenfassen



Datentyp Klasse

Speicherung verschiedenartiger Werte unter einem gemeinsamen Namen

Deklaration

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

Felder der Klasse Date

Verwendung als Typ

```
Date x, y;
```

Erzeugung

```
x = new Date();
```

Zugriff

```
x.day = 13;  
x.month = "November";  
x.year = 2002;
```



Date-Variablen sind Zeiger auf Objekte



Objekte

Objekte einer Klasse müssen vor ihrer ersten Benutzung erzeugt werden

```
Date x, y;
```

reserviert nur Speicher für die Zeigervariablen



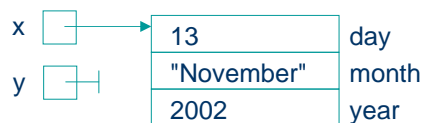
```
x = new Date();
```

erzeugt ein *Date*-Objekt und weist seine Adresse *x* zu



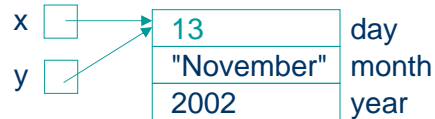
Eine Klasse ist wie eine Schablone, von der beliebig viele Objekte erzeugt werden können.

```
x.day = 13;  
x.month = "November";  
x.year = 2002;
```



Zuweisungen

`y = x;`



Zeigerzuweisung!

`y.day = 20;`



ändert auch x.day!

Zuweisungen sind erlaubt, wenn die Typen gleich sind

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

```
class Address {  
    int number;  
    String street;  
    int zipCode;  
}
```

```
Date d1, d2 = new Date();  
Address a1, a2 = new Address();
```

```
d1 = d2;    // ok, gleiche Typen  
a1 = a2;    // ok, gleiche Typen  
d1 = a1;    // verboten: verschiedene Typen trotz gleicher Struktur!
```



Vergleiche

Zeigervergleich

```
x == y  
x != y
```

vergleicht nur Zeiger

```
x < y  
x <= y  
x > y  
x >= y
```

nicht erlaubt !

Wertvergleich muss mittels Vergleichsmethode selbst implementiert werden

```
static boolean equalDate (Date x, Date y) {  
    return x.day == y.day && x.month.equals(y.month) && x.year == y.year;  
}
```

Anmerkung:

Wir werden später Methoden `boolean equals(Object o)` und `int compareTo(Object o)` kennen lernen !



Wo werden Klassen deklariert

Auf äußerster Ebene eines Programms (einer Datei)

```
class C1 {  
    ...  
}
```

```
class C2 {  
    ...  
}
```

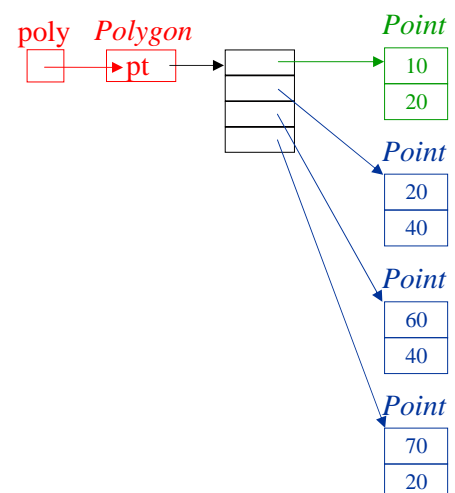
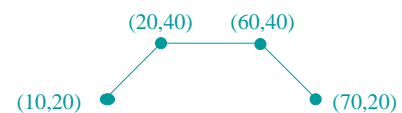
```
public class MainProgram {  
    public static void main (String[] arg) {  
        ...  
    }  
}
```



Beispiel: Polygone

```
class Point { int x, y; }  
class Polygon { Point[] pt; }
```

```
class Program {  
    Polygon poly = new Polygon();  
    poly.pt = new Point[4];  
    Point p = new Point(); p.x = 10; p.y = 20;  
    poly.pt[0] = p;  
    p = new Point(); p.x = 20; p.y = 40;  
    poly.pt[1] = p;  
    p = new Point(); p.x = 60; p.y = 40;  
    poly.pt[2] = p;  
    p = new Point(); p.x = 70; p.y = 20;  
    poly.pt[3] = p;  
    ...  
}
```



Kombination von Klassen mit Arrays

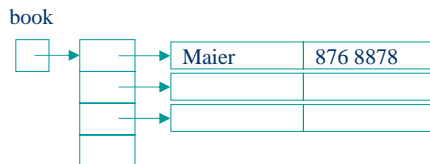
Beispiel: Telefonbuch

	name	phone
0	Maier	876 8878
	Mayr	543 2343
	Meier	656 2332
99		

zweidimensionales Array
kann hier nicht verwendet werden

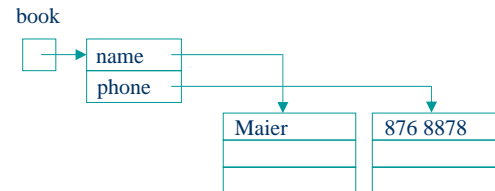
Array von Objekten

```
class Person {
    String name;
    int phone;
}
Person[] book = new Person[100];
```



Objekt bestehend aus 2 Arrays

```
class PhoneBook {
    String[] name;
    int[] phone;
}
PhoneBook book = new PhoneBook();
book.name = new String[100];
book.phone = new int[100];
```



Implementierung

```
class Person {
    String name;
    int phone;
}
```

```
class PhoneBook {
    Person[] book = new Person[1000];
    int nEntries = 0; // current number of entries in book

    void enter (String name, int phone) {
        if (nEntries >= book.length) Out.println("--- phone book full");
        else {
            book[nEntries] = new Person();
            book[nEntries].name = name;
            book[nEntries].phone = phone;
            nEntries++;
        }
    }

    int lookup (String name) {
        int i = 0;
        while (i < nEntries && !name.equals(book[i].name)) i++;
        // i >= nEntries || name.equals(book[i].name)
        if (i < nEntries) return book[i].phone; else return -1;
    }
}
```



Implementierung (Fortsetzung)

```
...
public static void main (String[] arg) {
    PhoneBook privBook = new PhoneBook();
    //----- read the phone book from a file
    In.open("phonebook.txt");
    String name = In.readName();
    int phone;
    while (In.done()) {
        phone = In.readInt();
        privBook.enter(name, phone);
        name = In.readName();
    }
    In.close();
    //----- search in the phone book
    for (;;) {
        Out.print("Name: "); name = In.readName();
        if (!In.done()) break;
        phone = privBook.lookup(name);
        if (phone > 0) Out.println("phone number = " + phone);
        else Out.println(name + " unknown");
    }
} // end PhoneBookSample
```



Methoden mit mehreren Rückgabewerten

Java-Funktionen haben nur 1 Rückgabewert

Will man mehrere Rückgabewerte, muss man sie zu einer Klasse zusammenfassen

Beispiel: Umrechnung von Sekunden auf Std, Min, Sek

```
class Time {
    int h, m, s;
}

class Program {

    static Time convert (int sec) {
        Time t = new Time();
        t.h = sec / 3600; t.m = (sec % 3600) / 60; t.s = sec % 60;
        return t;
    }

    public static void main (String[] arg) {
        Time t = convert(10000);
        Out.println(t.h + ":" + t.m + ":" + t.s);
    }
}
```

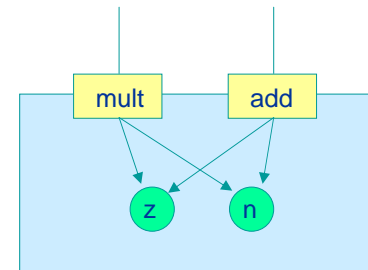


Klasse = Daten + Methoden

Beispiel: Bruchzahlenklasse

```
class Fraction {  
    int z; // Zähler  
    int n; // Nenner  
  
    void mult (Fraction f) {  
        this.z = this.z * f.z;  
        this.n = this.n * f.n;  
    }  
  
    void add (Fraction f) {  
        this.z = this.z * f.n + f.z * this.n;  
        this.n = this.n * f.n;  
    }  
}
```

f1.mult(f2);



geschlossener Baustein

- *mult* und *add* sind lokal zu *Fraction* (können auf *Fraction*-Objekte angewendet werden)
- *this* bezeichnet "dieses Objekt", auf das die Operation angewendet wird
- Methoden hier ohne *static* deklariert (siehe später)



Aufruf von Methoden

```
Fraction a = new Fraction();  
a.z = 1;  
a.n = 2;                                     // a == 1/2  
  
Fraction b = new Fraction();  
b.z = 3;  
b.n = 5;                                     // b == 3/5  
  
a.mult(b);
```

Auf das Objekt *a* wird die Operation *mult* angewendet (mit Parameter *b*)

Man sagt:

- *a* bekommt die Meldung (message) *mult*
- *a* ist der Empfänger der Meldung *mult*

Was passiert dabei?



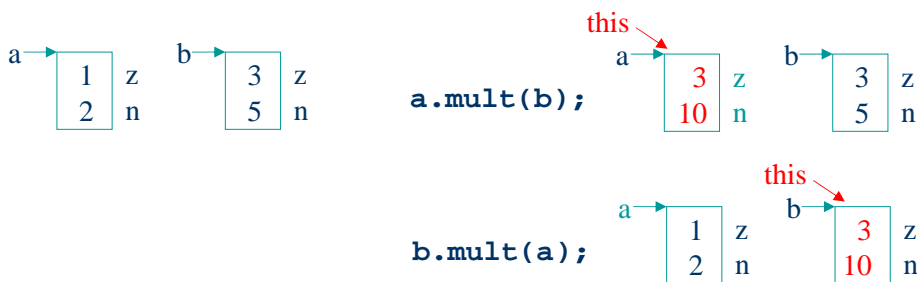
Aufruf von Methoden

`a.mult(b);`

```
void mult (/*Fraction this, */ Fraction f) {  
    this.z = this.z * f.z;  
    this.n = this.n * f.n;  
}
```

Was passiert?

- Parameterübergabe:
 `this = a;` (`this` ist ein versteckter Parameter jeder Methode)
 `f = b;`
- Es wird die `mult`-Methode der Klasse von `a` aufgerufen



Weglassen von this

```
class Fraction {  
    int z, n;  
  
    void mult (Fraction f) {  
        z = z * f.z;  
        n = n * f.n;  
    }  
  
    void add (Fraction n) {  
        z = z * n.n + this.n * n.z;  
        this.n = this.n * n.n;  
    }  
}
```

`z` und `n` sind eindeutig.
Compiler fügt `this` automatisch ein

`n` wäre nicht eindeutig.
Qualifikation mit `this` nötig

`this` kann weggelassen werden, wenn der restliche Name eindeutig ist



Konstruktoren

Spezielle Methoden, die beim Erzeugen eines Objekts automatisch aufgerufen werden

```
class Fraction {
    int z, n;
    Fraction (int z, int n) {
        this.z = z; this.n = n;
    }
    Fraction () {
        z = 0; n = 1;
    }
    void mult (Fraction f) {...}
    void add (Fraction f) {...}
}
```

- dienen zur Initialisierung eines Objekts
- heißen wie die Klasse
- ohne Funktionstyp oder void
- können Parameter haben
- können überladen werden

Überladene
Konstruktoren

Aufruf

```
Fraction f = new Fraction(3, 5);
Fraction g = new Fraction();
```

- legt neues *Fraction*-Objekt an
- ruft den Konstruktor auf



Expliziter Konstruktoraufruf

- Bei überladenen Konstruktoren kann ein Konstruktor von einem anderen Konstruktor „explizit“ aufgerufen werden
- Erfolgt über **this(...)** – Aufruf mit entsprechenden Parametern für den überladenen Konstruktor
- Darf nur als erste Anweisung eines Konstruktors verwendet werden

```
class Fraction {
    int z, n;
    Fraction (int z, int n) {
        this.z = z; this.n = n;
    }
    Fraction () {
        this(0, 1);
    }
    Fraction (int z) {
        this(z, 1);
    }
    void mult (Fraction f) {...}
    void add (Fraction f) {...}
}
```

expliziter Aufruf des Konstruktors
Fraction (int z, int n)



Destruktor

- Speicherplatz von Objekten wird durch Garbage Collector frei gegeben, brauchen daher im Normalfall keinen Destruktor
- Klassen können aber auch eine `finalize`-Methode implementieren, die insbesondere Nicht-Java-Ressourcen frei geben.
- Wird aufgerufen bevor das Objekt vom Garbage Collector zersört wird bzw. das Programm beendet wird

```
public class ProcessFile {
    private Stream file;

    public ProcessFile(String path) {
        file = new Stream(path);
    }

    // ...

    protected void finalize() throws Throwable {
        closeFile();
    }

    void closeFile() {
        if (file != null) {
            file.close();
            file = null;
        }
    }
}
```



static

```
class Window {
    int x, y, w, h; // Objektfelder (in jedem Window-Objekt vorhanden)
    static int border; // Klassenfeld (nur einmal pro Klasse vorhanden)
    void redraw () {...} // Objektmethode (auf Objekte anwendbar)
    static void setBorder (int n) {border = n;} // Klassenmethode (auf Klasse Window anwendbar)
    Window(int w, int h) {...} // Objektkonstruktor (zur Initialisierung von Objekten)
    static { /*read border from config file */} // Klassenkonstruktor (zur Initialisierung der Klasse)
}
```

Klasse Window

border
setBorder
Klassenkonstruktor

Window-Objekt

x
y
w
h
redraw Window()

Window-Objekt

x
y
w
h
redraw Window()

Window-Objekt

x
y
w
h
redraw Window()

- Objektmethoden haben Zugriff auf Klassenfelder (**redraw** kann auf **border** zugreifen)
- Klassenmethoden haben keinen Zugriff auf Objektfelder (**setBorder** kann nicht auf **x** zugreifen)



Was geschieht wann?

Beim Laden der Klasse *Window*

- Klassenfelder werden angelegt (`border`)
- Klassenkonstruktor wird aufgerufen

Beim Erzeugen eines *Window*-Objekts

- Objektfelder werden angelegt (`x`, `y`, `w`, `h`)
- Objektkonstruktor wird aufgerufen

Zugriffe

Zugriff auf `static`-Elemente über den Klassennamen

- `Window.border = ...; Window.setBorder(3);`
- Methoden der Klasse `Window` können Klassennamen weglassen (`border = ...; setBorder(3);`)
- Klassenkonstruktor wird nie explizit aufgerufen

Zugriff auf `nonstatic`-Elemente über einen Objektname

- `Window win = new Window(100, 50);`
`win.x = ...; win.redraw();`
- Methoden der Klasse `Window` können auf eigene Elemente direkt zugreifen (`x = ...; redraw();`)



Grafische Notation für Klassen

UML-Notation (Unified Modeling Language)

Fraction	<i>Klassenname</i>
int z int n	<i>Felder</i>
void mult (Fraction f) void add (Fraction f)	<i>Methoden</i>

Vereinfachte Form

Fraction	<i>falls weniger Details gewünscht oder nötig</i>
z n	
mult(f) add(f)	



Beispiel: Stack und Queue

Stack (Stapel, Kellerspeicher)

push(x); fügt x hinten an den Stack an
x = pop(); entfernt und liefert hinterstes Stackelement

```
push(3);  3
push(4);  3 4
x = pop(); 3 // x == 4
y = pop(); // y == 3
```

LIFO-Datenstruktur
(last in first out)



Queue (Puffer, Schlange)

put(x); fügt x hinten an die Queue an
x = get(); entfernt und liefert vorderstes Queueelement

```
put(3);  3
put(4);  3 4
x = get(); 4 // x == 3
y = get(); // y == 4
```

FIFO-Datenstruktur
(first in first out)



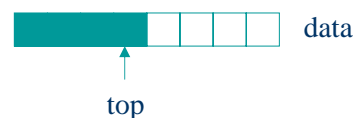
Klasse Stack

```
class Stack {
    int[] data;
    int top;

    Stack (int size) {
        data = new int[size]; top = -1;
    }

    void push (int x) {
        if (top >= data.length - 1)
            Out.println("-- overflow");
        else
            data[++top] = x;
    }

    int pop () {
        if (top < 0) {
            Out.println("-- underflow"); return 0;
        } else
            return data[top--];
    }
}
```



Benutzung

```
Stack s = new Stack(10);
s.push(3);
s.push(6);
int x = s.pop() + s.pop(); // x == 9
```



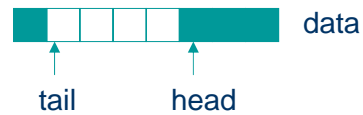
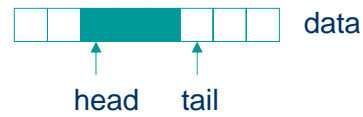
Klasse Queue

```
class Queue {
    int[] data;
    int head, tail;

    Queue (int size) {
        data = new int[size]; head = 0; tail = 0;
    }

    void put (int x) {
        if ((tail+1) % data.length == head)
            Out.println("-- overflow");
        else {
            data[tail] = x;
            tail = (tail+1) % data.length;
        }
    }

    int get () {
        if (head == tail) {
            Out.println("-- underflow"); return 0;
        } else {
            int x = data[head];
            head = (head+1) % data.length;
            return x;
        }
    }
}
```



Benutzung

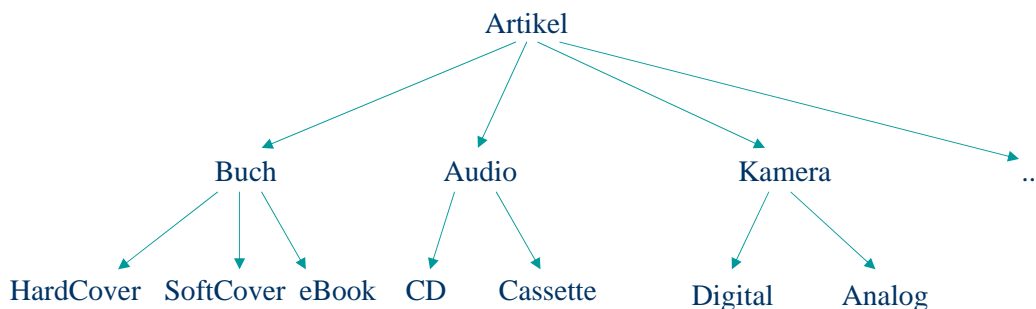
```
Queue q = new Queue(10);
q.put(3);
q.put(6);
int x = q.get(); // x == 3
int y = q.get(); // y == 6
```



Klassifikation

Dinge der realen Welt lassen sich oft klassifizieren

z.B. Artikel eines Web-Shops



Man beachte

- Ein *eBook* hat alle Eigenschaften eines *Buchs*; zusätzlich hat es ...
Ein *Buch* hat alle Eigenschaften eines *Artikels*; zusätzlich hat es ...
- *CD* und *Cassette* lassen sich gleichermaßen als *Audio* behandeln
Buch, *Audio* und *Kamera* lassen sich gleichermaßen als *Artikel* behandeln

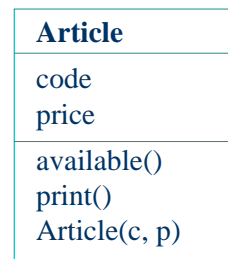
Vererbung



Vererbung

```
class Article {
    int code;
    int price;
    boolean available() {...}
    void print() {...}
    Article(int c, int p) {...}
}
```

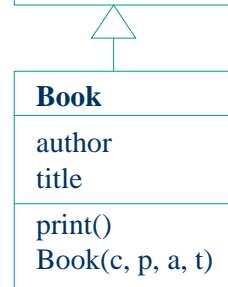
Oberklasse
Basisklasse



```
class Book extends Article {
    String author;
    String title;
    void print() {...}
    Book(int c, int p,
        String a, String t) {...}
}
```

Unterklasse

erbt: code, price, available, print
ergänzt: author, title, Konstruktor
überschreibt: print



Wenn keine Oberklasse angegeben wird, ist sie *Object*

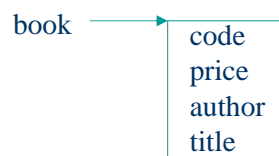


Überschreiben von Methoden

```
class Article {
    ...
    void print() {
        Out.print(code + " " + price);
    }
    Article(int c, int p) {
        code = c; price = p;
    }
}
```

Benutzung

```
Book book = new Book(code, price, author, title);
⇒ erzeugt Book-Objekt
⇒ Book-Konstruktor
⇒ Article-Konstruktor (code = c; price = p);
author = a; title = t;
```



```
class Book extends Article {
    ...
    void print() {
        super.print();
        Out.print(" " + author + ": " + title);
    }
    Book(int c, int p, String a, String t) {
        super(c, p);
        author = a; title = t;
    }
}
```

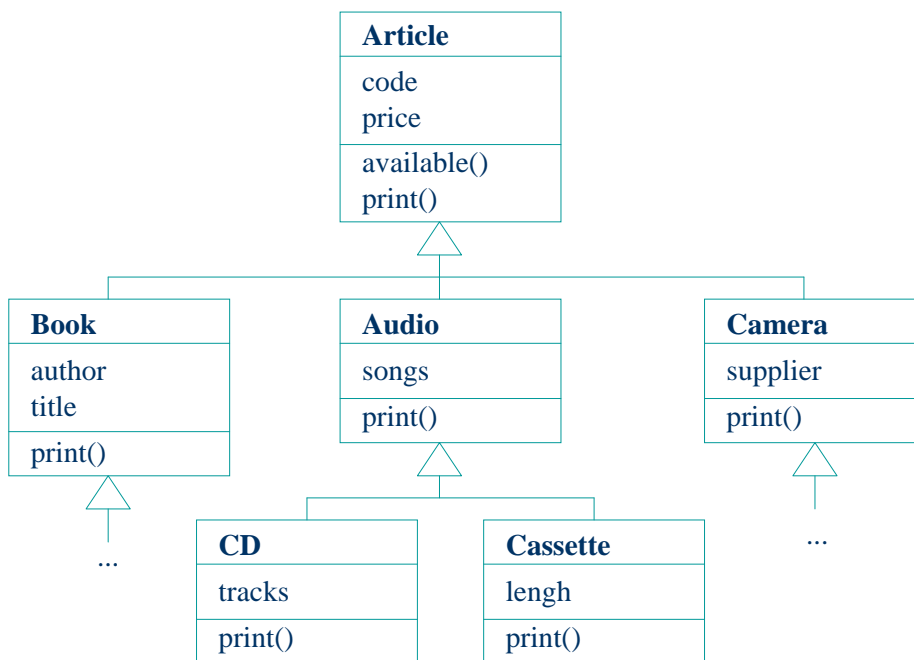
```
book.print();
⇒ print aus Book
⇒ print aus Article
⇒ Out.print(...);
```

code price
author: title

Ausgabe: code price author: title



Klassenhierarchien



Jedes Buch ist ein Artikel
Aber: nicht jeder Artikel ist ein Buch

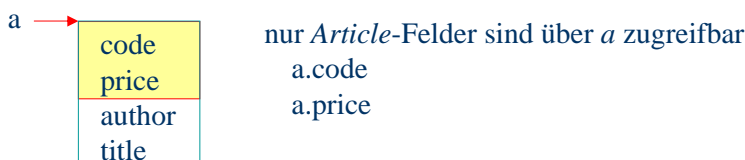


Kompatibilität zwischen Klassen

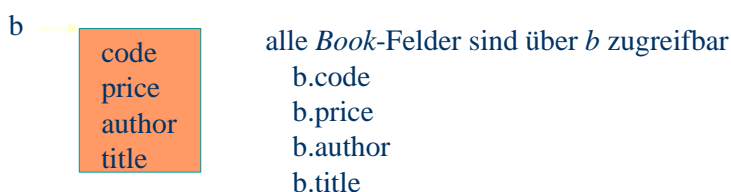
Unterklassen sind Spezialisierungen ihrer Oberklassen

Book-Objekte können Article-Variablen zugewiesen werden

```
Article a = new Book(code, price, author, title);
```

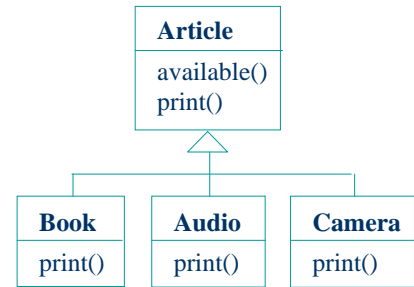
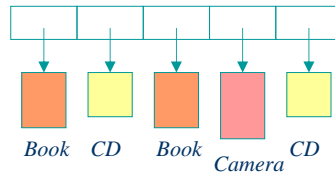


```
if (a instanceof Book) // Laufzeittypstest
    Book b = (Book) a; // Typumwandlung mit Laufzeittypprüfung
```



Heterogene Datenstruktur

Article[] a;



Alle Varianten können als Artikel behandelt werden

```
void printArticles() {
    for (int i = 0; i < a.length; i++) {
        if (a[i].available()) {
            a[i].print();
        }
    }
}
```

ruft *available()* aus *Article* auf

ruft je nach Artikelart das *print()* aus *Book*, *CD* oder *Camera* auf

Dynamische Bindung

obj.print() ruft die *print*-Methode des Objekts auf, auf das *obj* gerade zeigt

