

Objektorientiertes Programmieren

Prof. Dr. Christian Becker

Universität Stuttgart, Institut für Parallele und Verteilte Systeme

15. Mai 2025



Wieso entwickeln wir Software?

- ▶ Software wird geschrieben um **ein Problem zu lösen**
- ▶ Das Problem bzw. die Aufgabe der Software **ändert sich** häufig
(Kunde will etwas anderes, Problem ist anders als angenommen, ...)
- ▶ Im Schnitt ist der Aufwand eine Software weiterzuentwickeln (am Leben zu halten) deutlich höher als sie initial zu schreiben

Die echte Welt ist riesig

- ▶ Linux: ca. 40 Millionen Zeilen Code
- ▶ SAP: ca. 238 Millionen Zeilen Code
- ▶ Microsoft Word: ca. 30 Millionen Zeilen Code

Software sollte also **wartbar** sein

```
(globalThis.webpackChunk=globalThis.webpackChunk||[]).push([["vendors-node_modules_github_arianotify-polyfill_ariaNotify-polyfill_js-node_modules_github_mi-3abb8f"],{48359:()=>{if(!("ariaNotify"in Element.prototype)){let e=`${Date.now()}`;try{e=crypto.randomUUID()}catch{}let t=Symbol(),o=`live-region-${e}`;let Message=class Message{element;message;priority="none";interrupt="none";get #e(){return"all"}===this.interrupt||"pending"===this.interrupt}constructor({element:e,message:t,priority:o="none",interrupt:i="none"}){this.element=e,this.message=t,this.priority=o,this.interrupt=i}matches(e){return this.element===e.element&&this.priority===e.priority&&this.interrupt===e.interrupt}#t(){return this.element.isConnected&&this.element.closest("[inert]")&&(this.element.ownerDocument.querySelector(":modal")?.contains(this.element)??!0)}async announce(){if(!this.#t())return;let e=this.element.closest("dialog")||this.element.getRootNode();(!e||e instanceof Document)&&(e=document.body);let i=e.querySelector(o);this.#e&&i&&(i.remove(),i=null),i||(i=document.createElement(o),e.append(i)),await new Promise(e=>setTimeout(e,250)),i.handleMessage(t,this.message)}};let i=new class MessageQueue{#o=[];#i;enqueue(e){let{priority:t,interrupt:o}=e;if(("all"===o||"pending"===o)&&(this.#o=this.#o.filter(e=>e.matches(e))),important===t){let t=this.#o.findLastIndex(e=>important===e.priority);this.#o.splice(t+1,0,e)}else this.#o.push(e);this.#i||this.#l()}async #l(){this.#i=this.#i.announce(),this.#l()}};let LiveRegionCustomElement=class LiveRegionCustomElement{#n=this.attachShadow({mode:"closed"});connectedCallback(){this.ariaLive="polite",this.#n.style.marginLeft="-1px",this.style.marginTop="-1px",this.style.position="absolute",this.style.height="1px",this.style.overflow="hidden",this.style.clipPath="rect(0 0 0 0)",this.style.overflowWrap="normal",this.#n.handleMessage(e=null,o="")}{t===e&&(this.#n.textContent=o&&(o+="\xa0"),this.#n.textContent=o)}};customElements.define(o,LiveRegionCustomElement),Element.prototype.ariaNotify=function(e,{priority:t="none",interrupt:o="none"}={}){i.enqueue(new Message({element:this,message:e,priority:t,interrupt:o})))}}},70170:(e,t,o=>{"use strict";function i(e,t=0,{start:o=!0,middle:l=!0,once:n=!1})={}{let r,s=o,a=0,c=!1;function u(...i){if(c)return;let p=Date.now()-a;a=Date.now(),o&&l&&p>=t&&(s=!0),s?(s=!1,e.apply(this,i),n&&u.cancel()):(!l&&p<t||l)&&(clearTimeout(r),r=setTimeout(()=>{a=Date.now(),e.apply(this,i),n&&u.cancel()},l?t-p:t))}return u.cancel=()=>{clearTimeout(r),c=!0},u}function l(e,t=0,{start:o=!1,middle:n=!1,once:r=!1})={}{return i(e,t,{start:o,middle:n,once:r})}o.d(t,{#n:()=>l,s:()=>l}),31196:e=>{e.exports={polyfill:function({var e,t=window,o=document;if(!("scrollBehavior"in o.document.documentElement.style)||!0===t.__forceSmoothScrollPolyfill__){var i=t.HTMLElement||t.Element,l={scroll:t.scroll||t.scrollTo,scrollBy:t.scrollBy,elementScroll:i.prototype.scroll||s,scrollIntoView:i.prototype.scrollIntoView},n=t.performance&&t.performance.now?t.performance.now.bind(t.performance):Date.now,r=(e=t.navigator.userAgent,RegExp("MSIE |Trident/|Edge/").test(e))?!1:0;t.scroll=t.scrollTo=function(){if(void 0===arguments[0]){if(!0===a(arguments[0])){l.scroll.call(t,void 0===arguments[0].left?arguments[0].left:"object"!==typeof arguments[0]?arguments[0]:t.scrollX||t.pageXOffset,void 0===arguments[0].top?arguments[0].top:void 0===arguments[1]?arguments[1]:t.scrollY||t.pageYOffset);return}p.call(t,o.body,void 0===arguments[0].left?~~arguments[0].left:t.scrollX||t.pageXOffset,void 0===arguments[0].top?~~arguments[0].top:t.scrollY||t.pageYOffset)}},t.scrollBy=function(){if(void 0===arguments[0]){if(a(arguments[0])){l.scrollBy.call(t,void 0===arguments[0].left?arguments[0].left:"object"!==typeof arguments[0]?arguments[0]:0,void 0===arguments[0].top?arguments[0].top:void 0===arguments[1]?arguments[1]:0);return}p.call(t,o.body,~~arguments[0].left+(t.scrollX||t.pageXOffset),~~arguments[0].top+(t.scrollY
```

Ist das wartbar?

Auch wenig Code kann unverständlich sein

- ▶ Was macht dieses Stück Code?
- ▶ Können Fehler auftreten? ... Sicher?

```
1  static boolean foo(int[] a, int v, int l, int u) {  
2      int p = (u - l) / 2 + 1;  
3      if(u < l || p < 0 || p >= a.length) {  
4          return false;  
5      }  
6      int e = a[p];  
7      if(e == v) {  
8          return true;  
9      }  
10     return (e > v) ? foo(a, v, l, p-1) : foo(a, v, p+1, u);  
11 }
```

Java

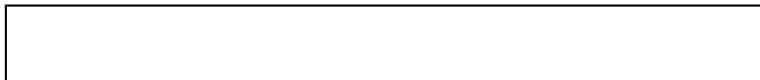
Keine Ahnung! Und das sind nur 10 Zeilen...

Problemverständnis

- ▶ Menschen sind faul, Programmierer auch.
- ▶ Software ist riesig und kompliziert
- ▶ Man möchte so wenig (Code) wie möglich lesen müssen um zu verstehen was ein Stück Software tut
- ▶ Leider wird man aber immer Code von anderen Leuten benutzen müssen
- ▶ Andere werden Ihren Code benutzen müssen...
- ▶ Schreiben Sie Ihren Code so, dass der nächste Idiot ihn versteht.
- ▶ Sie sind meistens selbst der nächste Idiot

Was bedeutet Wartbarkeit?

- ▶ Software sinnvoll unterteilen in Komponenten oder Module
- ▶ Üblicherweise sind nur Teile von großer Software für eine Änderung relevant
- ▶ Das Verhalten von diesen Komponenten sollte schnell begreifbar sein
- ▶ Welches Stück Code macht was? Was tut es nicht?
- ▶ Wenn Sie den Code anderer benutzen oder ändern, möchten Sie klare Regeln auf die Sie sich verlassen können



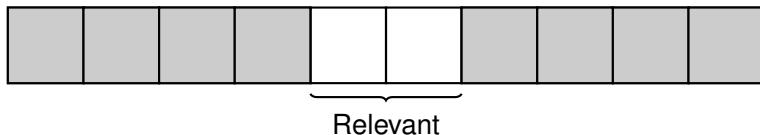
Was bedeutet Wartbarkeit?

- ▶ Software sinnvoll unterteilen in Komponenten oder Module
- ▶ Üblicherweise sind nur Teile von großer Software für eine Änderung relevant
- ▶ Das Verhalten von diesen Komponenten sollte schnell begreifbar sein
- ▶ Welches Stück Code macht was? Was tut es nicht?
- ▶ Wenn Sie den Code anderer benutzen oder ändern, möchten Sie klare Regeln auf die Sie sich verlassen können



Was bedeutet Wartbarkeit?

- ▶ Software sinnvoll unterteilen in Komponenten oder Module
- ▶ Üblicherweise sind nur Teile von großer Software für eine Änderung relevant
- ▶ Das Verhalten von diesen Komponenten sollte schnell begreifbar sein
- ▶ Welches Stück Code macht was? Was tut es nicht?
- ▶ Wenn Sie den Code anderer benutzen oder ändern, möchten Sie klare Regeln auf die Sie sich verlassen können



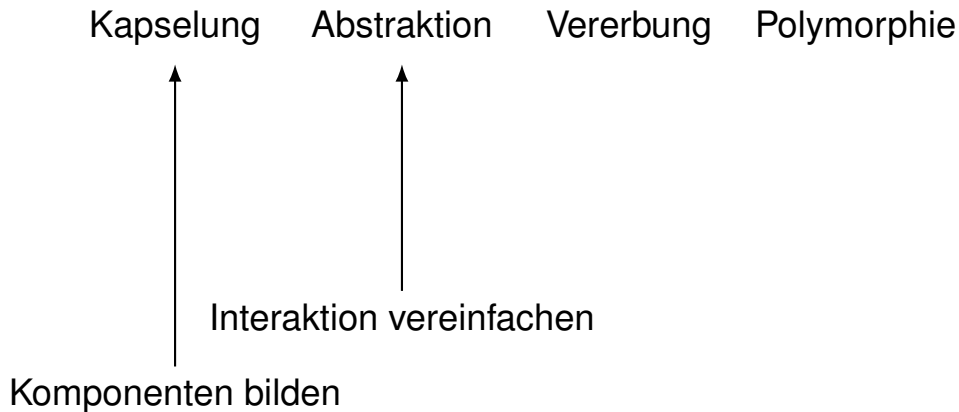
Die Grundkonzepte der Objektorientierung

Kapselung Abstraktion Vererbung Polymorphie

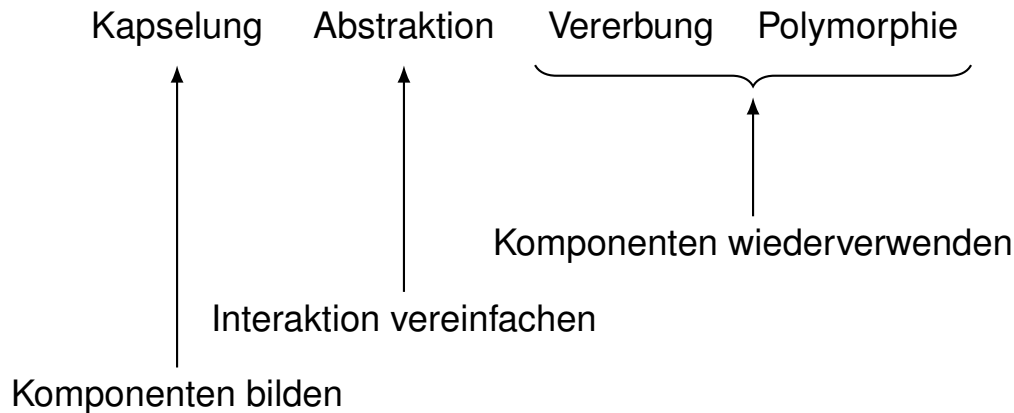
Die Grundkonzepte der Objektorientierung



Die Grundkonzepte der Objektorientierung



Die Grundkonzepte der Objektorientierung



Kapselung: Unterteilen von Software in Komponenten

- ▶ Wie können Komponenten aussehen?
- ▶ Einige Möglichkeiten Komponenten zu bilden haben Sie schon kennengelernt!

Funktionen sind Komponenten

- Eine Funktion beschreibt ein Verhalten.

```
1 static boolean find(int[] array, int value) {  
2     for(int i = 0; i < array.length; ++i) {  
3         if(value == array[i]) {  
4             return true;  
5         }  
6     }  
7     return false;  
8 }
```

Java

Funktionen sind Komponenten (2)

Beschreibung des Verhaltens

Garantien

Parameter

Rückgabe

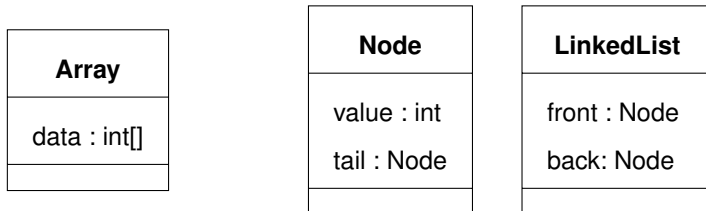
```
1  /**
2   * Searches the given array for a value.
3   * Returns true if the value was found, false otherwise.
4   * This function does not modify the array in any way
5   * @param array The array to search
6   * @param value The value to search for
7   * @return true if the value was found in array, false otherwise
8   */
9  static boolean find(int[] array, int value) {
10     for(int i = 0; i < array.length; ++i) {
11         if(value == array[i]) {
12             return true;
13         }
14     }
15     return false;
16 }
```

Java

Code muss eigentlich nicht mehr gelesen werden damit man diese Funktion benutzen kann! Implementierung von `find` kann sich ändern, solange die Eigenschaften eingehalten werden.

Datenstrukturen sind Komponenten

- ▶ Eine Datenstruktur beschreibt einen Zustand
- ▶ Zum Beispiel ein Integer-Array beschreibt eine Menge von Zahlen

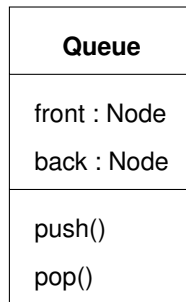
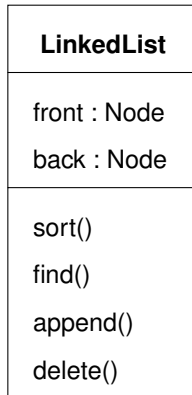
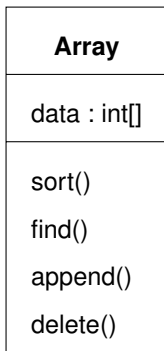


- ▶ Das Verhalten von Software ist oft stark abhängig von den Daten und der Art und Weise wie sie gespeichert sind
- ▶ Zu einer Datenstruktur gehört also oft auch ein gewisses Verhalten
- ▶ Klassen gruppieren einen Zustand und ein Verhalten

Klassen sind das Grundkonzept der Objektorientierung.

Klassen

- ▶ Klassen gruppieren einen Zustand und ein Verhalten
- ▶ Das vereinfacht die Benutzung, weil im Kontext der Daten direkt die möglichen Interaktionen definiert werden



Klassen (2)

LinkedList
front : Node back : Node
sort() find(value : int) : bool append(value : int) delete(position : int)

```
1  /**
2   * A variable length container implemented as a singly
3   * linked list
4   */
5  class LinkedList {
6      class Node {
7          ...
8      };
9
10     LinkedList() { ... }
11
12     void    sort() { ... }
13     boolean find(int value) { ... }
14     void    append(int value) { ... }
15     void    delete(int position) { ... }
16
17     Node front;
18     Node back;
19 }
```

Java

LinkedList
front : Node back : Node
sort() find(value : int) : bool append(value : int) delete(position : int)

- ▶ Variablen in einer Klasse nennt man *member variables* oder Attribute (attributes)
- ▶ Funktionen in einer Klasse nennt man *Methoden* (methods)
- ▶ Diese Nomenklatur unterscheidet sich von Sprache zu Sprache leicht, gemeint ist immer das gleiche

Zugriffsbeschränkungen

- ▶ Wir wollen weiterhin so wenig Code wie möglich lesen müssen.
- ▶ Unterteilen in Implementierungsdetails (private) und öffentlichen Zustand / öffentliches Verhalten (public)

```
1  class LinkedList {  
2      private class Node {  
3          ...  
4      };  
5  
6      public LinkedList() { ... }  
7  
8      public void sort() { ... }  
9      public boolean find(int value) { ... }  
10     public void append(int value) { ... }  
11     public void delete(int position) { ... }  
12  
13     private Node front;  
14     private Node back;  
15 }
```

Java

private: Details, die für die Benutzung unwichtig sind

public: Wichtig für die Benutzung

Implementierung weiterhin unwichtig für die Benutzung

Zugriffsbeschränkungen sind Garantien

Wir unterscheiden "versprochenes externes Verhalten"(public) und internes Verhalten (Implementierungsdetails) (private)

Private

- ▶ Private garantiert, dass niemand außer diejenige, die die Klasse programmiert darauf zugreift
- ▶ Private sollte der Standard sein

Public

- ▶ Public garantiert, dass sich das von außen beobachtbare Verhalten einer Methode nicht ändern wird (Stabilität der Interaktionspunkte)
- ▶ Nutzerin kann sich auf eine Beschreibung verlassen und muss nicht die ganze Methode lesen

Schnittstellen / Interfaces

- ▶ public Methoden und Variablen einer Klasse beschreiben das *Interface*
- ▶ Das Interface ist das, was man verstehen muss um richtig mit einer Klasse interagieren zu können!
- ▶ Designziel: Ein möglichst einfaches Interface für ein gegebenes Problem (Das ist ein schwieriges Problem, unterschätzen Sie es nicht)
- ▶ Best Practice: **Interface besteht nur aus Methoden, alle Attribute sind private**

Graphische Modellierung (UML)

- ▶ + bedeutet public
- ▶ – bedeutet private

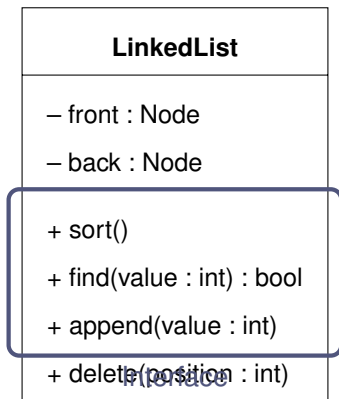
LinkedList
<ul style="list-style-type: none">– front : Node– back : Node
<ul style="list-style-type: none">+ sort()+ find(value : int) : bool+ append(value : int)+ delete(position : int)

```
1  class LinkedList {
2      private class Node {
3          ...
4      };
5
6      public LinkedList() { ... }
7
8      public void    sort() { ... }
9      public boolean find(int value) { ... }
10     public void    append(int value) { ... }
11     public void    delete(int position) { ... }
12
13     private Node front;
14     private Node back;
15 }
```

Java

Interface

public Methoden und Variablen einer Klasse beschreiben das *Interface*

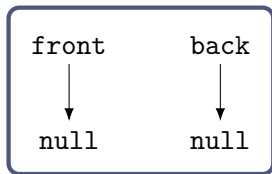


Instanzen und Objekte

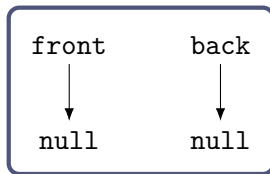
- ▶ Klassen beschreiben Daten und Verhalten
- ▶ Klassen kann man als Bauplan verstehen
- ▶ Klassen beschreiben einen Datentyp, ähnlich wie `int`, `float`, `double`, ...
- ▶ Eine Variable vom Typ T ist eine *Instanz* der Klasse T
- ▶ Java: `new` keyword notwendig um eine Objektinstanz zu erstellen

```
1 LinkedList listA = new LinkedList();  
2 LinkedList listB = new LinkedList();
```

Java



`listA`



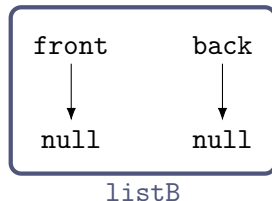
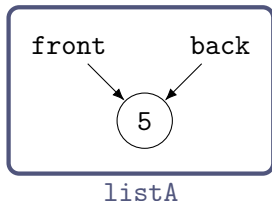
`listB`

Objektorientierung

- ▶ Ähnlich wie man zwei ints addieren kann, kann man mit Instanzen von Klassen (Objekten) interagieren
- ▶ Wichtig: Objekte haben jeweils eigene Variablen (hier front und back)

```
1 LinkedList listA = new LinkedList();  
2 LinkedList listB = new LinkedList();  
3  
4 listA.append(5); // Definierte Interaktion: Anhängen
```

Java

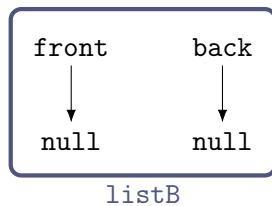
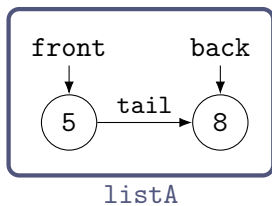


Objektorientierung

- ▶ Ähnlich wie man zwei `ints` addieren kann, kann man mit Instanzen von Klassen (Objekten) interagieren
- ▶ Wichtig: Objekte haben jeweils eigene Variablen (hier `front` und `back`)

```
1 LinkedList listA = new LinkedList();  
2 LinkedList listB = new LinkedList();  
3  
4 listA.append(5);  
5 listA.append(8);
```

Java



- ▶ Methoden werden „auf einem Objekt“ aufgerufen

```
1  LinkedList list = new LinkedList();  
2  list.append(5);
```

Java

Objekt

Methode

Parameter

Referenzen in Methoden

- Innerhalb der Implementierung von Methoden werden Attribute automatisch richtig referenziert

```
1 public class LinkedList {  
2  
3     public void append(int value) {  
4         // Hier ist back automatisch die Variable die zum Objekt gehört, auf dem  
5         // append aufgerufen wird  
6         back = new Node();  
7         // Expliziter Zugriff auf Attribute mit this  
8         this.back = new Node(); // Äquivalent zu oben  
9     }  
10  
11     private Node back;  
12 }
```

Java

Implementierung von append

```
1  public class LinkedList {
2
3      public void append(int value) {
4          Node element = new Node();
5          element.value = value;
6
7          if(front == null) {
8              // Liste ist leer
9              front = element;
10             back = element;
11             return;
12         }
13
14         back.tail = element;
15         back = element;
16     }
17
18     private Node back;
19 }
```

Java



Konstruktoren

- ▶ *Konstruktoren* beschreiben, wie eine neue Instanz erstellt wird
- ▶ Sie sind eine besondere Art von Methode
- ▶ Konstruktoren können auch Argumente entgegen nehmen
- ▶ Der Konstruktor ohne Argumente heißt *default constructor*

```
1 public class LinkedList {  
2     // Create a new empty list  
3     public LinkedList() { ... } // der default constructor  
4  
5     // Create a new list with n entries, each with the given value  
6     public LinkedList(int n, int value) {  
7         for(int i = 0; i < n; ++i) {  
8             this.append(value);  
9         }  
10    }  
11  
12    private Node front = null;  
13    private Node back = null;  
14 }
```

Funktionsname ist immer der Klassenname

Java

Delegation von Konstruktoren

- ▶ Wenn man mehrere Konstruktoren definiert, kommt es oft vor, dass sich ihr Verhalten überschneidet
- ▶ Man kann mit einem Konstruktor das Verhalten eines anderen Konstruktors erweitern. Das nennt man *Delegation*

```
1 public LinkedList(int n, int value) {  
2     for(int i = 0; i < n; ++i) {  
3         this.append(value);  
4     }  
5 }  
6  
7 public LinkedList(int n) {  
8     for(int i = 0; i < n; ++i) {  
9         this.append(0);  
10    }  
11 }
```

Java

```
1 public LinkedList(int n, int value) {  
2     for(int i = 0; i < n; ++i) {  
3         this.append(value);  
4     }  
5 }  
6  
7 public LinkedList(int n) {  
8     this(n, 0);  
9  
10    // Hier darf noch mehr passieren  
11 }
```

Delegation

static variables: Klassenvariablen

- ▶ Manchmal müssen sich alle Instanzen einer Klasse eine Variable teilen
- ▶ Zum Beispiel ein Zähler, wie viele Instanzen es insgesamt erstellt wurden, Maximale Länge einer Liste, ...
- ▶ Dafür benutzt man das keyword `static`
- ▶ Man nennt diese Variablen *Klassenvariablen*

```
1 public class LinkedList {  
2  
3     public LinkedList() {  
4         counter += 1;  
5     }  
6  
7     static private long counter = 0;  
8 }
```

Java

static methods: Statische Methoden

- ▶ Methoden die ein Verhalten beschreiben, das nur logisch zur Klasse gehört aber nicht direkt ein Objekt bearbeiten, können auch statische Methoden sein
- ▶ Werden zum Beispiel oft für sogenannte *Factories* benutzt (dazu später mehr)

```
1 public class LinkedList {
2     // Könnte auch mit einem Konstruktor realisiert werden
3     public static createFrom(int[] array) {
4         // WICHTIG: static methods arbeiten nicht auf einer Instanz der Klasse.
5         // D.h. this ist hier nicht definiert
6         LinkedList list = new LinkedList();
7         for(int i = 0; i < list.length; ++i) {
8             list.append(array[i]);
9         }
10        return list;
11    }
12 }
13
14 // Benutzung
15 int[] array = {0,1,2,3};
16 LinkedList list = LinkedList.createFrom(array);
```

Java

Überladene Methoden (method overloading)

- ▶ Methoden die gleich heißen aber verschiedene Parameter entgegen nehmen
- ▶ Wir haben schon overloading für Konstruktoren gesehen, das geht aber allgemein für alle Methoden
- ▶ Overloading ist nicht in allen Sprachen möglich (z.B. Rust verbietet overloading)
- ▶ Wichtig: Die Parameter müssen sich unterscheiden!

```
1 public class LinkedList {  
2     public void append(int value) {  
3         ...  
4     }  
5  
6     public void append(long value) {  
7         ...  
8     }  
9  
10    public void append(float value) {  
11        ...  
12    }  
13 }
```

Drei overloads mit
gleichem Namen,
aber verschiede-
nen Parameter-
Typen

Java

Shadowing von Variablen

- ▶ In einer Methode (z.B. einem Konstruktor) kann es vorkommen, dass ein Parameter den gleichen Namen hat, wie ein Attribut
- ▶ In diesem Fall sagt man, der Parameter *shadowed* (also überlagert) das Attribut
- ▶ Um auf das Attribut zuzugreifen, muss man `this` verwenden

```
1 private class Node{
2     public Node(int value) {
3         // Zwei Variablen mit dem Namen value!
4
5         this.value = value;
6     }
7
8     int value = null;
9     Node tail = null;
10 }
```

Java

Kapselung Abstraktion Vererbung Polymorphie

⏟
Nächste Vorlesung

- ▶ Klassen sind Vorlagen um Objekte zu erzeugen (instanziiieren)
- ▶ Klassen erlauben die Konstruktion eigener Datentypen
- ▶ Objekt kapselt Verhalten und Zustand hinter einem Interface und besitzt eine eindeutige Identität

Packages

packages, package private

imports