# 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


Ist das wartbar?

# Auch wenig Code kann unverständlich sein

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

```java
static boolean foo(int[] a, int v, int l, int u) {
    int p = (u - l) / 2 + l;
    if(u < l || p < 0 || p >= a.length) {
        return false;
    }
    int e = a[p];
    if(e == v) {
        return true;
    }
    return (e > v) ? foo(a, v, l, p-1) : foo(a, v, p+1, u);
}
```

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?

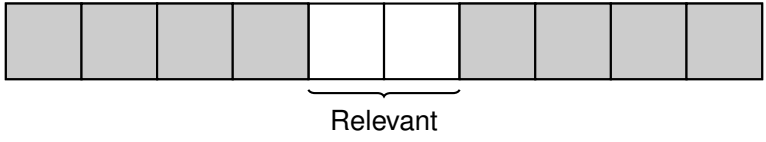
- Software sinnvoll unterteilen in Komponenten oder Module
- Üblicherweise sind nur Teile von großer Software für eine Änderung relevant
- Das Verhalten von diesen Komponenten sollte schnell begreifbar sein
- Welches Stück Code macht was? Was tut es nicht?
- Wenn Sie den Code anderer benutzen oder ändern,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

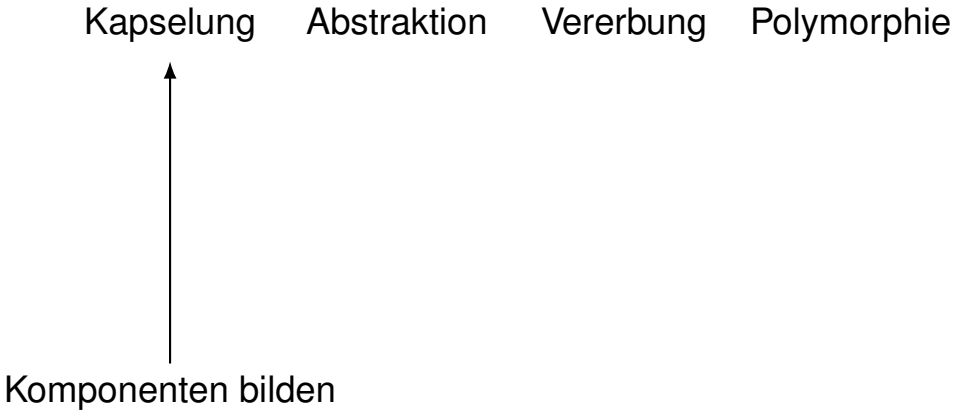
- Software sinnvoll unterteilen in Komponenten oder Module
- Üblicherweise sind nur Teile von großer Software für eine Änderung relevant
- Das Verhalten von diesen Komponenten sollte schnell begreifbar sein
- Welches Stück Code macht was? Was tut es nicht?
- Wenn Sie den Code anderer benutzen oder ändern,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



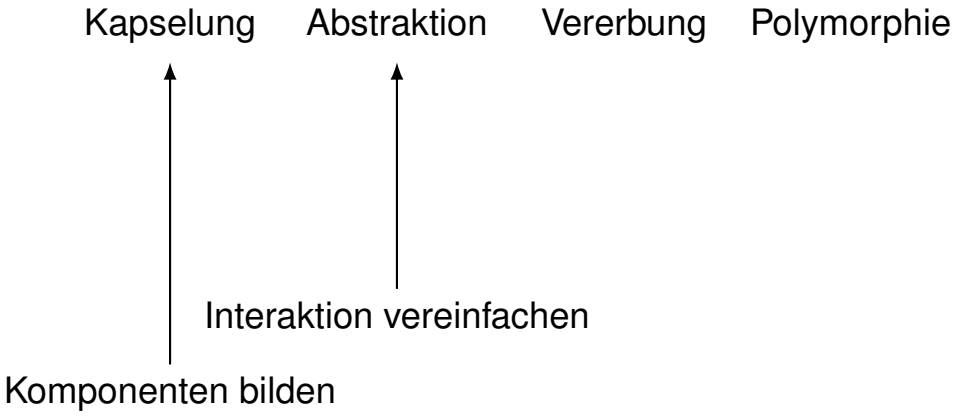
# Die Grundkonzepte der Objektorientierung

Kapselung Abstraktion Vererbung Polymorphie

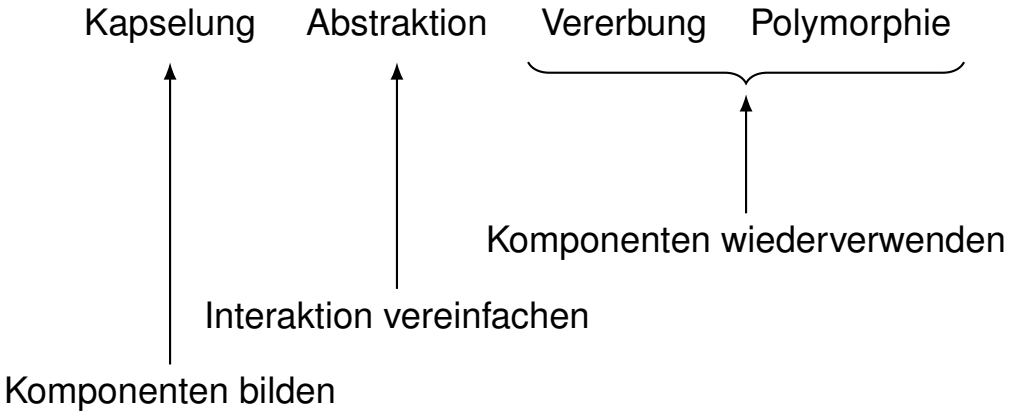
# Die Grundkonzepte der Objektorientierung



# 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!

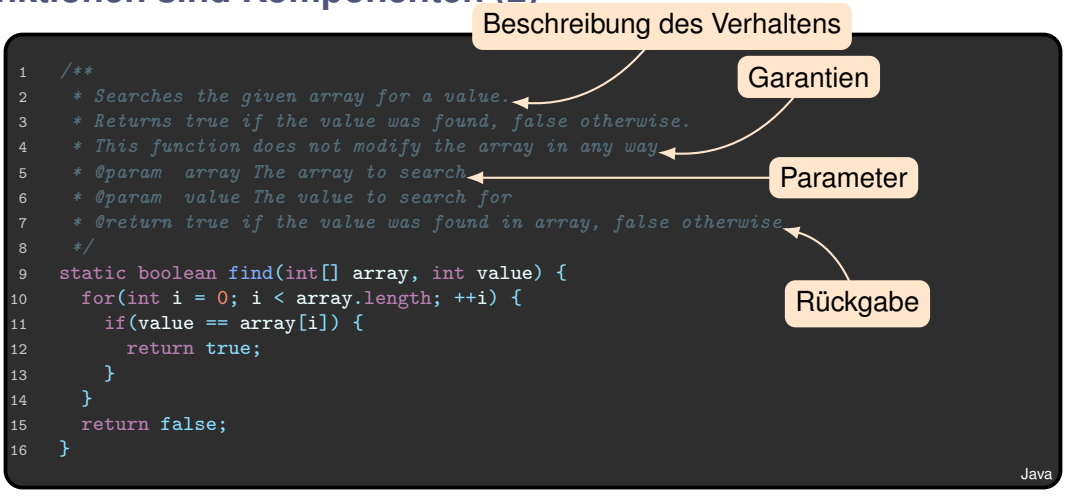
# Funktionen sind Komponenten

- Eine Funktion beschreibt ein Verhalten.

```java
static boolean find(int[] array, int value) {
  for(int i = 0; i < array.length; ++i) {
    if(value == array[i]) {
      return true;
    }
  }
  return false;
}
```

Java

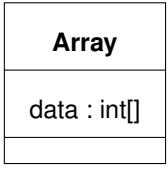
# Funktionen sind Komponenten (2)

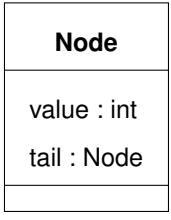


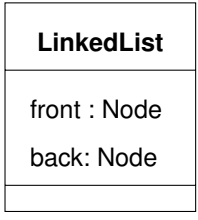

Code muss eigentlich nicht mehr gelesen werden damit man diese Funktion benutzen kann! Implementierung von `find` kann sich ändern, solange die Eigenschaften eingehalten werden.

# Datenstrukturen sind Komponenten

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



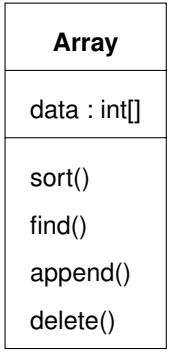
# Klassen

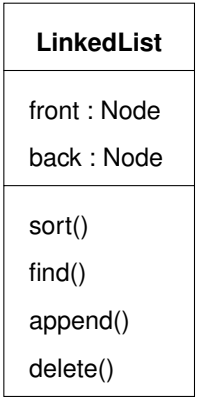
- Das Verhalten von Software ist oft stark abhängig von den Daten und der Art und Weise wie sie gespeichert sind
- Zu einer Datenstruktur gehört also oft auch ein gewisses Verhalten
- Klassen gruppieren einen Zustand und ein Verhalten

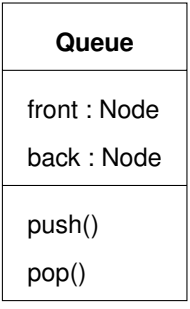
Klassen sind das Grundkonzept der Objektorientierung.

# Klassen

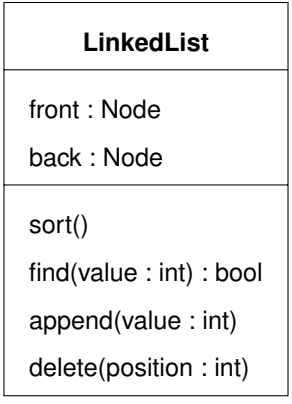
- Klassen gruppieren einen Zustand und ein Verhalten
- Das vereinfacht die Benutzung, weil im Kontext der Daten direkt die möglichen Interaktionen definiert werden



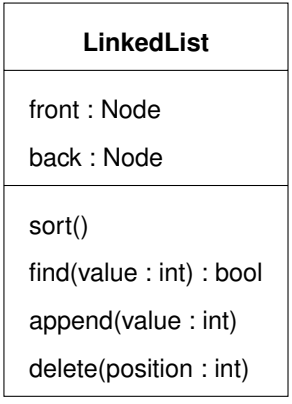
# Klassen (2)




```java
/**
 * A variable length container implemented as a singly
 * linked list
 */
class LinkedList {
  class Node {
    ...
  };

  LinkedList() { ... }

  void    sort() { ... }
  boolean find(int value) { ... }
  void    append(int value) { ... }
  void    delete(int position) { ... }

  Node front;
  Node back;
}
```

Java

# Nomenklatur




- 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)

```java
class LinkedList {
  private class Node {
    ...
  };

  public LinkedList() { ... }

  public void    sort() { ... }
  public boolean find(int value) { ... }
  public void    append(int value) { ... }
  public void    delete(int position) { ... }

  private Node front;
  private Node back;
}
```

Java

private: Details, die für die Benutzung unwichtig sind

public: Wichtig für die Benutzung

Implementierung weiterhin unwichtig für die Benutzung

# Zugriffsbeschränkunen 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

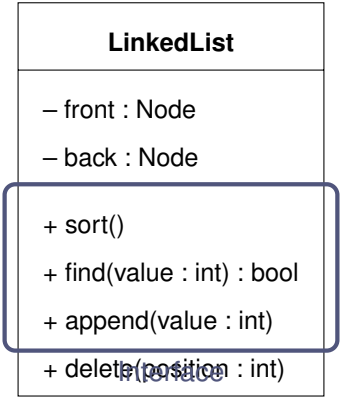
| **LinkedList** |
| --- |
| – front : Node<br>– back : Node |
| + sort()<br>+ find(value : int) : bool<br>+ append(value : int)<br>+ delete(position : int) |

```java
class LinkedList {
  private class Node {
    ...
  };

  public LinkedList() { ... }

  public void    sort() { ... }
  public boolean find(int value) { ... }
  public void    append(int value) { ... }
  public void    delete(int position) { ... }

  private Node front;
  private Node back;
}
```

Java

# Interface

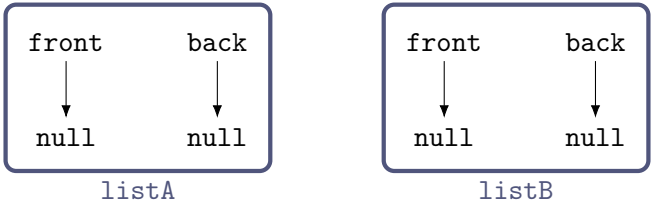
public Methoden und Variablen einer Klasse beschreiben das *Interface*



# 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

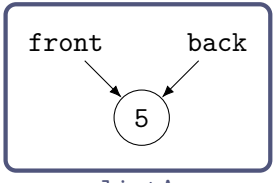
```java
LinkedList listA = new LinkedList();
LinkedList listB = new LinkedList();
```



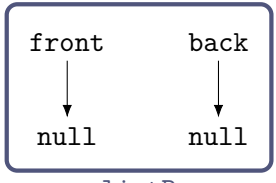
# Objektorientierung

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

```java
LinkedList listA = new LinkedList();
LinkedList listB = new LinkedList();

listA.append(5); // Definierte Interaktion: Anhängen
```




listA



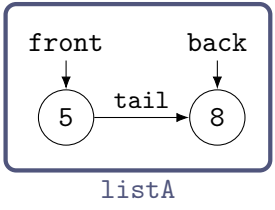

listB

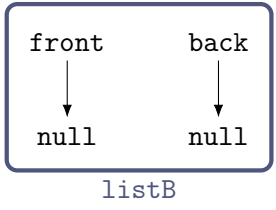
# Objektorientierung

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

```java
LinkedList listA = new LinkedList();
LinkedList listB = new LinkedList();

listA.append(5);
listA.append(8);
```

Java



# Methoden

- Methoden werden „auf einem Objekt“ aufgerufen

```java
LinkedList list = new LinkedList();
list.append(5);
```

Java

Objekt

Methode

Parameter

# Referenzen in Methoden

- Innerhalb der Implementierung von Methoden werden Attribute automatisch richtig referenziert

```java
public class LinkedList {

  public void append(int value) {
    // Hier ist back automatisch die Variable die zum Objekt gehört, auf dem
    // append aufgerufen wird
    back = new Node();
    // Expliziter Zugriff auf Attribute mit this
    this.back = new Node(); // Äquivalent zu oben
  }

  private Node back;
}
```

Java

# Implementierung von append

```java
public class LinkedList {

  public void append(int value) {
    Node element = new Node();
    element.value = value;

    if(front == null) {
      // Liste ist leer
      front = element;
      back = element;
      return;
    }

    back.tail = element;
    back = element;
  }

  private Node back;
}
```

Java

# Konstruktoren

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

```java
public class LinkedList {
  // Create a new empty list
  public LinkedList() { ... } // der default constructor

  // Create a new list with n entries, each with the given value
  public LinkedList(int n, int value) {
    for(int i = 0; i < n; ++i) {
      this.append(value);
    }
  }

  private Node front = null;
  private Node back = null;
}
```

Java

Funktionsname ist immer der Klassenname

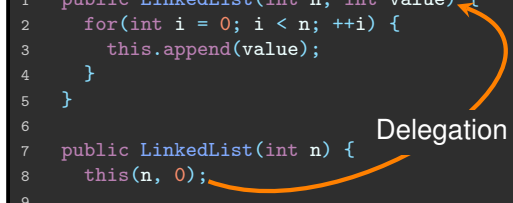
# Delegation von Konstruktoren

- 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*

```java
public LinkedList(int n, int value) {
  for(int i = 0; i < n; ++i) {
    this.append(value);
  }
}

public LinkedList(int n) {
  for(int i = 0; i < n; ++i) {
    this.append(0);
  }
}
```

Java

```java
public LinkedList(int n, int value) {
  for(int i = 0; i < n; ++i) {
    this.append(value);
  }
}

public LinkedList(int n) {
  this(n, 0);

  // Hier darf noch mehr passieren
}
```




Java

# `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*

```java
public class LinkedList {

  public LinkedList() {
    counter += 1;
  }

  static private long counter = 0;
}
```

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)

```java
public class LinkedList {
  // Könnte auch mit einem Konstruktor realisiert werden
  public static createFrom(int[] array) {
    // WICHTIG: static methods arbeiten nicht auf einer Instanz der Klasse.
    // D.h. this ist hier nicht definiert
    LinkedList list = new LinkedList();
    for(int i = 0; i < list.length; ++i) {
      list.append(array[i]);
    }
    return list;
  }
}

// Benutzung
int[] array = {0,1,2,3};
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!

```java
public class LinkedList {
  public void append(int value) {
    ...
  }

  public void append(long value) {
    ...
  }

  public void append(float value) {
    ...
  }
}
```

Java

Drei overloads mit gleichem Namen, aber verschiedenen Parameter-Typen

# 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

```java
private class Node{
  public Node(int value) {
                          // Zwei Variablen mit dem Namen value!

    this.value = value;
  }

  int value = null;
  Node tail = null;
}
```

Java

# Zusammenfassung

Kapselung Abstraktion Vererbung Polymorphie

Nächste Vorlesung

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

# Packages

packages, package private

imports