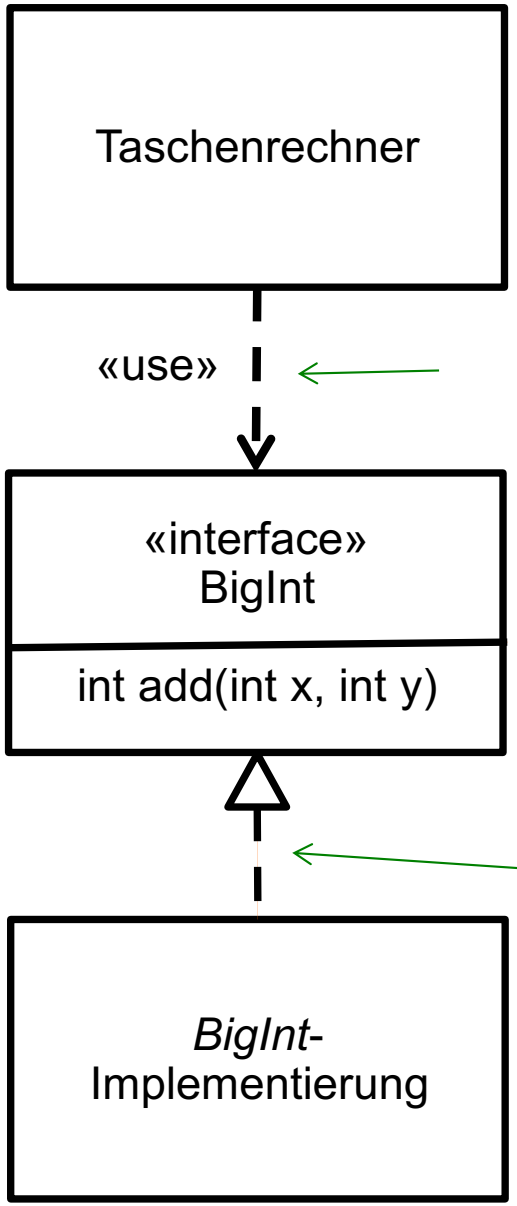
Interfaces und der ADT „Lineare Liste“

- Interfaces
  - Motivation
  - Hardware
  - Software
- ADT Lineare Liste
  - Implementierungsmöglichkeit
  - Java-Interface
  - Implementierung der Methoden
  - Doppelt verkettete Listen
- Interfaces und Klassen
  - Verschiedene Klassen implementieren das gleiche Interface
  - Eine Klasse implementiert verschiedene Interfaces
- Eclipse-Bedienung

Motivation

- Wir bauen mittlerweile unsere Programme aus mehreren Komponenten (= Klassen) auf. Beispiele:
  - Die *BigInt*s benutzen *Liste*n (könnten sie zumindest)
  - Ein Taschenrechner benutzt die *BigInt*
- Damit die Aufteilung funktioniert müssen wir festlegen, wie diese Komponenten zusammenarbeiten,
  wir müssen also ihre *Schnittstellen* (engl. *Interfaces*) zueinander festlegen
  (man könnte auch sagen „ihre *Verbindungsstellen* miteinander“)
- Wie diese Schnittstellen realisiert werden, ist uninteressant
  - Einen Benutzer interessiert nicht, wie der Taschenrechner funktioniert
  - Den Taschenrechner interessiert nicht, wie die *BigInt*s funktionieren
  - Die *BigInt*s interessiert nicht, wie die *Liste*n funktionieren

Veranschaulichung (UML)

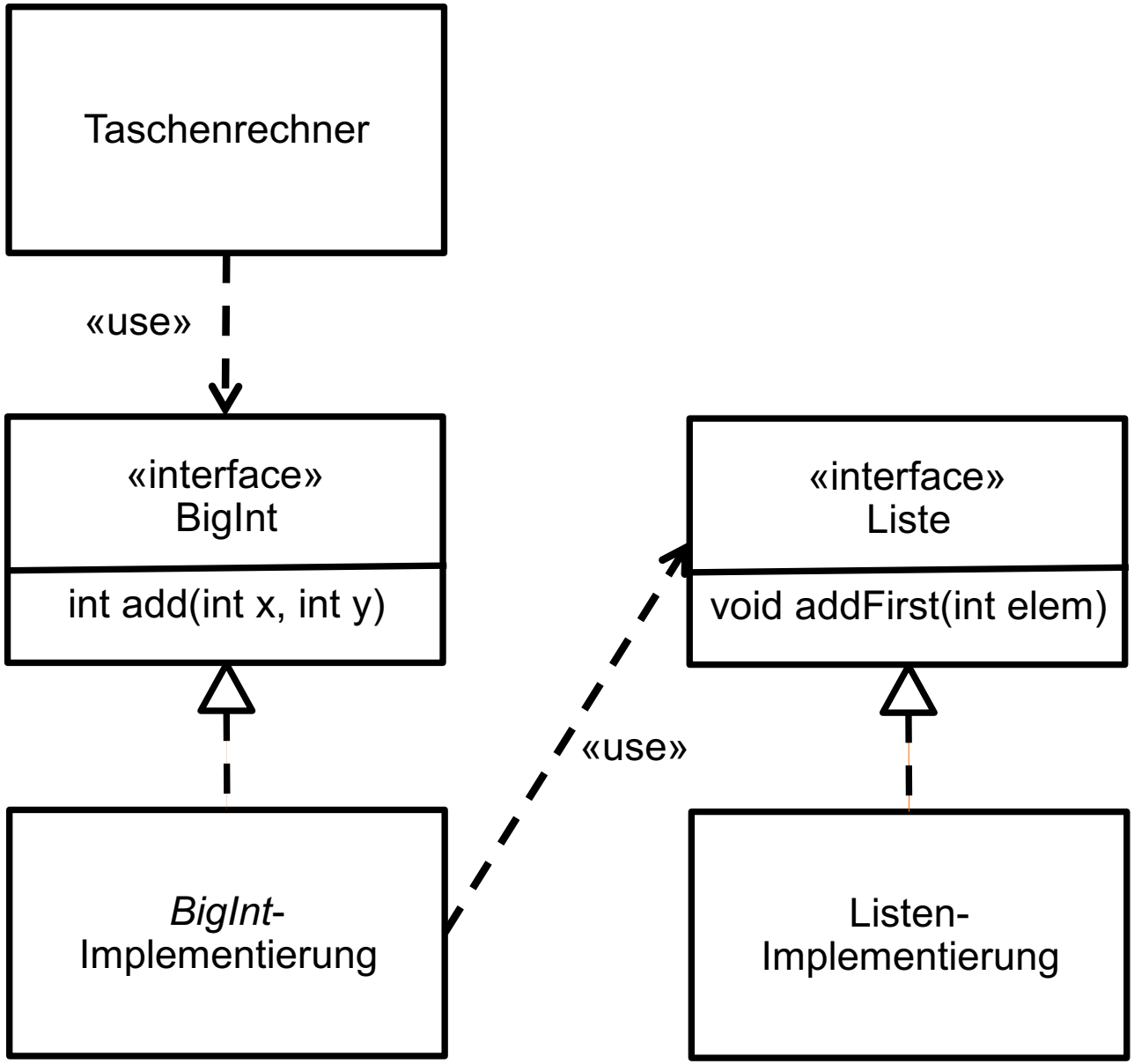


Der „Taschenrechner“ benutzt die Schnittstelle

Schnittstelle „BigInt“ mit einer Methode, ...

... die von der „BigInt-Implementierung“ angeboten wird

Veranschaulichung (UML)



Motivation

- Als Beispiele schauen wir uns zunächst Schnittstellen zwischen Hardware-Komponenten an (solche Schnittstellen kennen wir alle!)

Beispiele für (Hardware-)Schnittstellen

Beispiele für Schnittstellen

Warum definieren wir überhaupt Schnittstellen?

Definition einer Schnittstelle als Java-Interface

```java
public interface MP3Player {

    /**
     * Wählt eine neue Playlist mit dem angegebenen Namen aus.
     * Wenn der angegebene Name nicht existiert, dann wird der Aufruf ignoriert.
     */
    void selectPlaylist(String playlistName);

    /**
     * Spielt das Lied an der angegebenen Position aus der aktuellen Playlist ab Beginn.
     * Wenn keine Playlist ausgewählt ist oder die Position nicht existiert, wird der Aufruf ignoriert.
     * @param songPosition Die erste Position hat die Position 0
     */
    void play(int songPosition);

    /**
     * Spielt dasjenige Lied, welches in der aktuellen Playlist auf das aktuelle folgt.
     * Wird gerade das letzte Lied der aktuellen Playlist gespielt, wird das aktuelle Lied abgebrochen,
     * aber kein neues Lied begonnen.
     * Wenn gerade kein Lied gespielt wird, wird der Aufruf ignoriert.
     * @return Name des neuen Lieds, sofern vorhanden, sonst undefiniert
     */
    String playNext();

    /**
     * Stoppt das Abspielen eines Titels.
     * Wird gerade kein Lied abgespielt, dann wird der Aufruf ignoriert.
     */
    void stop();

    /**
     * Versucht, die Lautstärke auf den angegebenen Wert einzustellen.
     * Wird ein ungültiger Wert übergeben, dann wird der nächstliegende gültige Wert als neue Lautstärke eingestellt.
     * @param volume Wertebereich: 0..100 inklusive
     * @return die neu eingestellte Lautstärke
     */
    int setVolume(int volume);

}
```

Definition einer Schnittstelle als Java-Interface

```
public interface MP3Player {

    /**
     * Wählt eine neue Playlist mit dem angegebenen Namen aus.
     * Wenn der angegebene Name nicht existiert, dann wird der Aufruf ignoriert.
     */
    void selectPlaylist(String playlistName);

    /**
     * Spielt das Lied an der angegebenen Position aus der aktuellen Playlist ab Beginn.
     * Wenn keine Playlist ausgewählt ist oder die Position nicht existiert, wird der Aufruf ignoriert.
     * @param songPosition Die erste Position hat die Position 0
     */
    void play(int songPosition);

    /**
     * Spielt dasjenige Lied, welches in der aktuellen Playlist auf das aktuelle folgt.
     * Wird gerade das letzte Lied der aktuellen Playlist gespielt, wird das aktuelle Lied abgebrochen,
     * aber kein neues Lied begonnen.
     * Wenn gerade kein Lied gespielt wird, wird der Aufruf ignoriert.
     * @return Name des neuen Lieds, sofern vorhanden, sonst undefiniert
     */
    String playNext();

    /**
     * Stoppt das Abspielen eines Titels.
     * Wird gerade kein Lied abgespielt, dann wird der Aufruf ignoriert.
     */
    void stop();

    /**
     * Versucht, die Lautstärke auf den angegebenen Wert einzustellen.
     * Wird ein ungültiger Wert übergeben, dann wird der nächstliegende gültige Wert als neue Lautstärke eingestellt.
     * @param volume Wertebereich: 0..100 inklusive
     * @return die neu eingestellte Lautstärke
     */
    int setVolume(int volume);

}
```

angegeben werden nur Methodenköpfe (Signaturen)

Die Wirkung der Methoden wird textuell beschrieben

Definition einer Schnittstelle als Java-Interface

```java
public interface BigInteger {

    /**
     * Addiert die als Parameter übergebene Zahl vom Typ BigInteger zu der im Objekt
     * vorliegenden Zahl hinzu und ersetzt diese mit dem Ergebnis. Das Ergebnis der Addition
     * wird also anstelle der zuvor vorliegenden Zahl im Objekt gespeichert (so dass man damit
     * weiterarbeiten kann). Das als Parameter übergebene Objekt bleibt unverändert.
     */
    public void add( BigInteger number );

    /**
     * Multipliziert die als Parameter übergebene Zahl vom Typ BigInteger mit der im Objekt
     * vorliegenden Zahl und ersetzt diese mit dem Ergebnis. Das Ergebnis der Multiplikation
     * wird also anstelle der zuvor vorliegenden Zahl im Objekt gespeichert (so dass man damit
     * weiterarbeiten kann). Das als Parameter übergebene Objekt bleibt unverändert.
     */
    public void times(BigInteger number);

    /**
     * Liefert die Anzahl der Ziffern der vorliegenden Zahl ohne führende Nullen.
     */
    public int length();

    /**
     * Liefert das Feld in dem die Ziffern der vorliegenden Zahl gespeichert wurden.
     */
    public int[] getDigits();

    /**
     * Liefert die vorliegende Zahl als String.
     */
    public String toString();

    /**
     * Liefert genau dann true, wenn beide Objekte die gleiche Zahl repräsentieren, sonst false.
     */
    public boolean equals(BigInteger number);

    /**
     * Liefert genau dann true, wenn die vorliegende Zahl größer ist als der Parameter number, sonst false.
     */
    public boolean greaterThan(BigInteger number);

}
```

Zusammenfassung

- Eine Schnittstelle zu einem "Ding" definiert nur, *was* ich *damit tun* kann
- Sie legt *nicht* fest, *wie* dieses "Ding" arbeitet / wie es implementiert wird

→Dafür braucht man Klassen, die die Schnittstelle *implementieren*

Theorie und Programmiersprache zusammen

- Wir wissen:

Ein ADT
- hat eine *Menge von Werten*
- hat *Operationen* auf diesen Werten
  die nur über eine *Schnittstelle* zugänglich sind,
- definiert *Regeln* über die Wirkung der Operationen auf den Werten
- ist *gekapselt* (hat eine verborgene Implementierung).

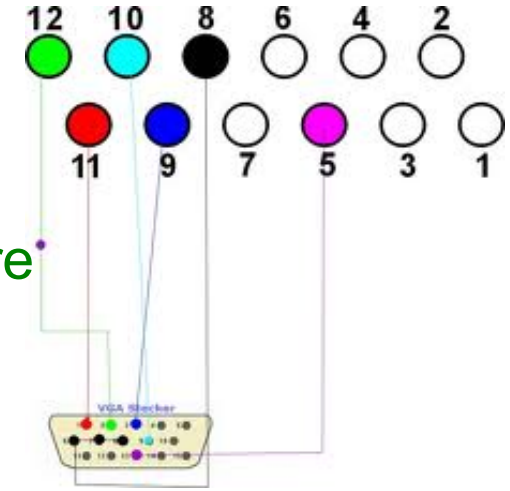
- Es gibt ein geeignetes Java-Konstrukt, um ADTs zu *spezifizieren* (nicht: zu implementieren):

  das ***Interface***

Wir brauchen also drei Dinge

Die Schnittstelle selbst;
in Java: eine Interface–Definition
im ADT: die Signaturen der Operationen

Die Information, welcher Pin was bedeutet;
in Java: Kommentare zu den Methoden im Interface mittels *JavaDoc*
im ADT: Regeln



Ein Gerät, das die Schnittstelle anbietet;
in Java: eine Klasse
im ADT: gekapselte Menge von Werten

Definition einer Schnittstelle: ein Java-Interface

```
public interface List {

//  List create();

    void addFirst(int elem);

    void addLast(int elem);

    void removeFirst();

    int getFirst();

    int getLast();

    int getAt(int index);

    boolean isEmpty();

    boolean contains(int elem);

    void clear();

    int size();

    List clone();

    void delete(int elem);

    ListNode getHead();

}
```

Um die *create*-Methode (oder eine beliebige andere Methode) aus dem *List*-Interface aufzurufen benötigen wir ein Objekt, das dieses Interface implementiert, d.h. dessen Methoden zur Verfügung stellt.

-> Das geht nicht: Wir müssten *create* aufrufen, um ein solches Objekt zu erstellen.

-> Wir realisieren *create* durch einen Konstruktor einer Klasse, die das Interface implementiert.

-> Der Konstruktor heißt so, wie diese Klasse.

-> Den Namen der Klasse kennen wir noch nicht!

-> Wir können keinen Konstruktor oder eine äquivalente Methode im Interface angeben!!!

Interfaces in Java

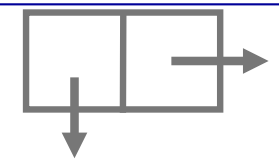
- Ein Interface kann beliebig viele nicht-statische Methoden enthalten
  - Es werden nur die Methoden-Köpfe angegeben
  - Abgesehen vom Rückgabetyp entspricht das einer *Signatur*
- Interfaces können keine Konstruktoren haben (s. Vorne)
- Interfaces können keine statischen Methoden haben
- Interfaces sind in Java vollwertige Typ-Angaben
  - Ihr Name kann (fast) überall dort benutzt werden, wo auch die Namen elementarer Datentypen und von Klassen benutzt werden können
- Um damit arbeiten zu können benötigen wir eine Klasse, die das Interface implementiert
  - Die Klasse muss alle Methoden des Interface‘ anbieten
  - Alle diese Methoden müssen öffentlich (*public*) sein
  - Sonst gibt es für die Klasse keine Einschränkungen bzgl. Attributen, Konstruktoren, (statischen oder nicht-statischen) Methoden, ...

Überlegungen zur Implementierung des Listen-Interface‘

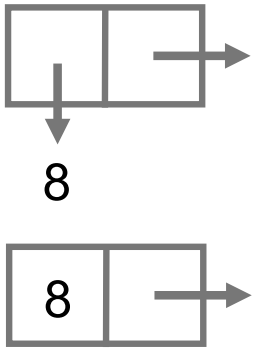
- Für die Implementierung der Liste (in einer Klasse) gibt es verschiedene Varianten
  - Sie haben bereits eine Implementierung auf Basis eines Arrays entwickelt (oder tun das gerade)
- Wir betrachten hier eine Implementierung mit einzelnen Listenelementen, d.h. Objekten, die je ein Listenelement repräsentieren
- Die Listenelemente sind nur für die Verknüpfung zuständig, nicht aber für die konkreten Inhalte, die in der Liste verwaltet werden
- Als Werte bringen wir zunächst nur ganze Zahlen (*int*) unter
  - Man kann natürlich beliebige andere Typen vorsehen:
    Es ändert sich dann nur der Typ des Attributs *value*
  - Die Listenoperationen ändern sich nicht!

Darstellung von Listen

allgemeine Darstellung

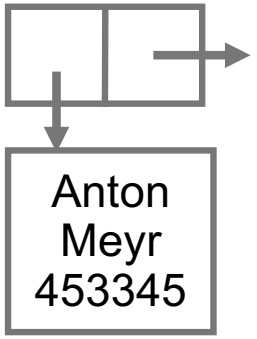


Konkretes Beispiel:
int-Liste




```
public class ListNode {

    private int value;
    private ListNode next;

}
```

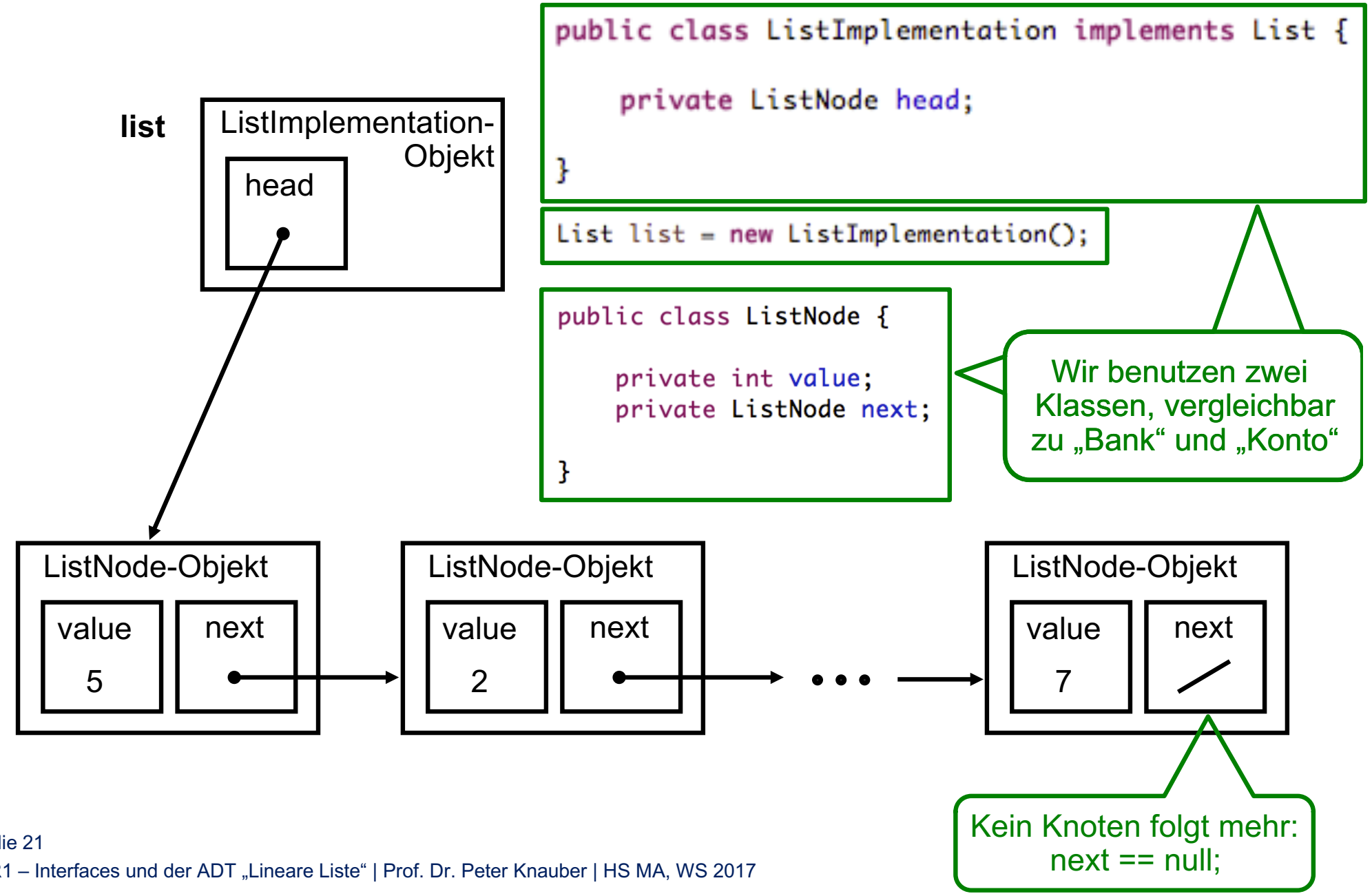
anderes Beispiel:
Liste von Personen
mit ihrer
Telefonnummer




```
public class ListNode {

    private Person value;
    private ListNode next;

}
```

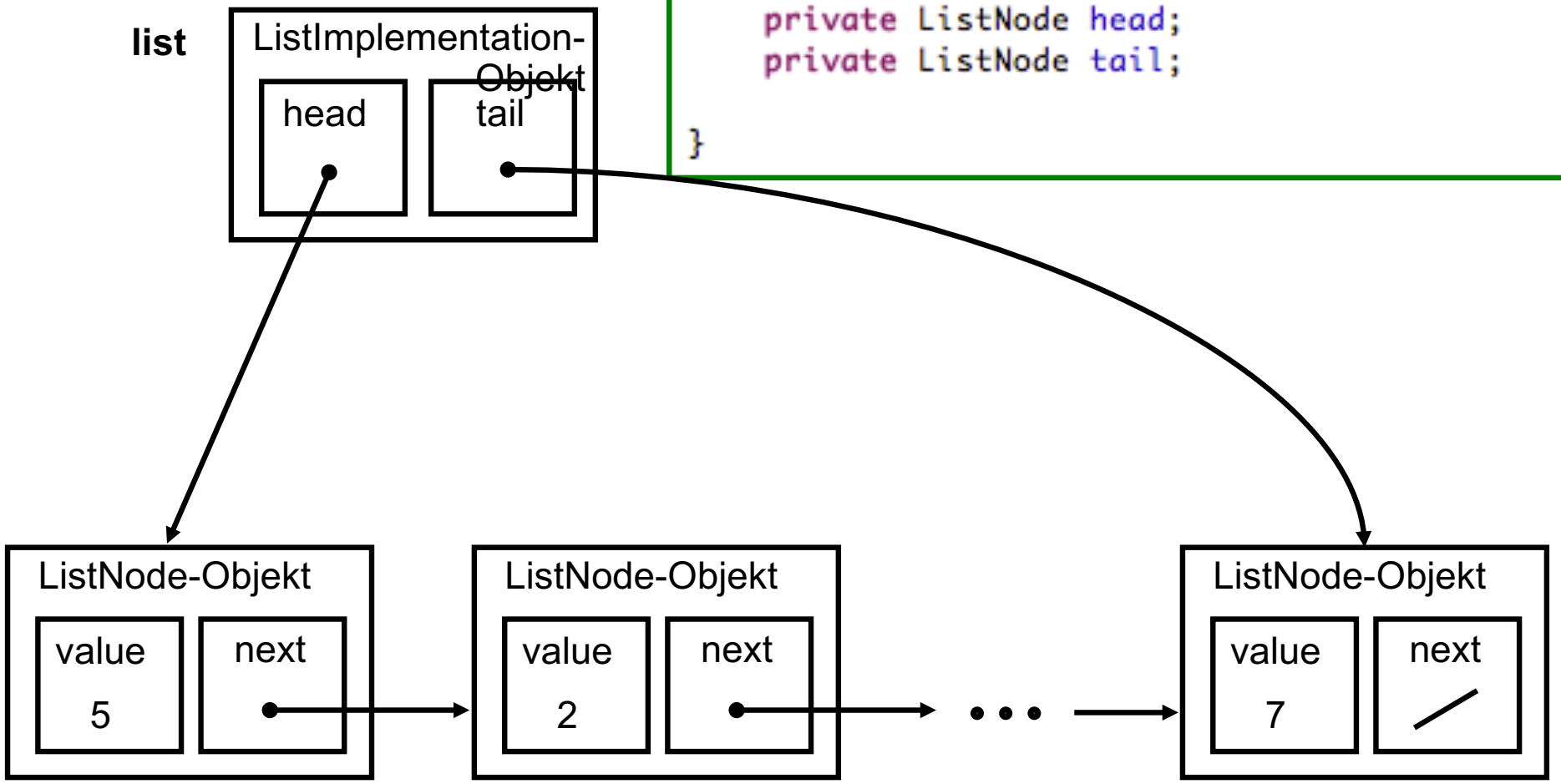
```
public class Person {
    private String firstName;
    private String lastName;
    private long phoneNumber;
}
```

Lineare Liste: einfach verkettet



Lineare Liste: einfach verkettet mit Tail-Zeiger

```
public class ListImplementation implements List {

    private ListNode head;
    private ListNode tail;

}
```



Listenarten

Es gibt diverse Listenarten, optimiert für verschiedene Aufgaben, z.B.

- Sortierte / unsortierte Listen
- Listen mit Tail-Zeiger
- Doppelt verkettete Listen
- Array-Listen
- etc.

Was kann man mit Listen machen?

- Beliebige Mengen beliebiger Objekte speichern
- Stack als Liste implementieren
- Queue als Liste implementieren
- Set als Liste implementieren
- etc.

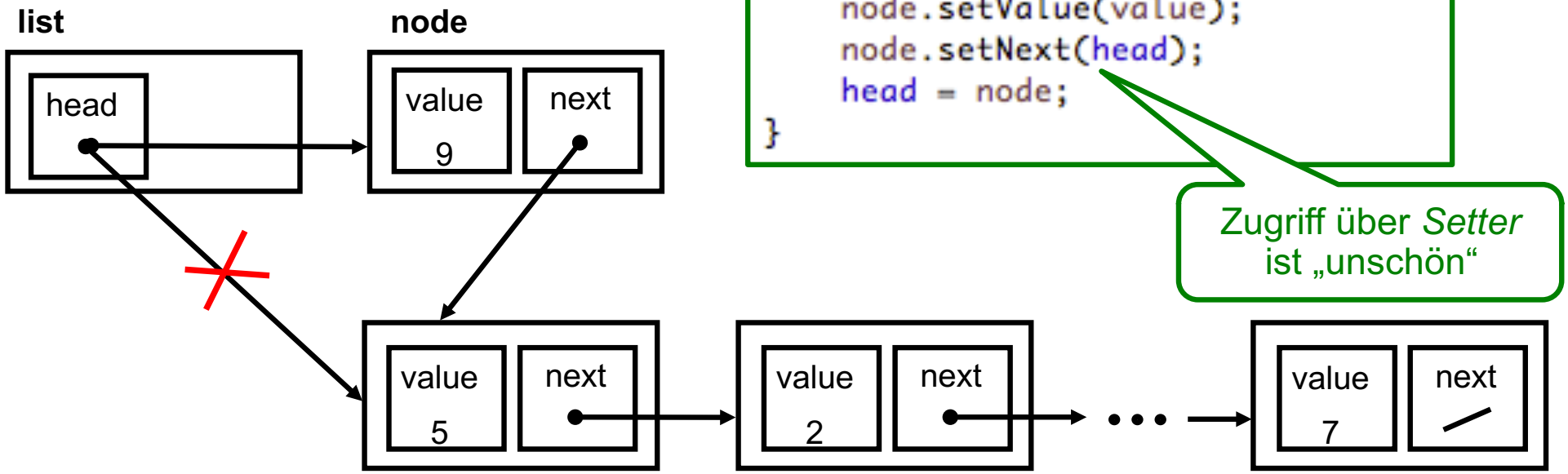
Lineare Liste: Implementierung der Operationen 1

- addFirst: Element × Liste → Liste
  - fügt ein neues Element am Beginn einer Liste ein

```
void addFirst(int elem);
```

```
public void addFirst(int value) {
    ListNode node = new ListNode();
    node.setValue(value);
    node.setNext(head);
    head = node;
}
```

Zugriff über *Setter* ist „unschön“




```
list.addFirst(9);
```

Z.B. in *main* wird ein neues Element hinzugefügt

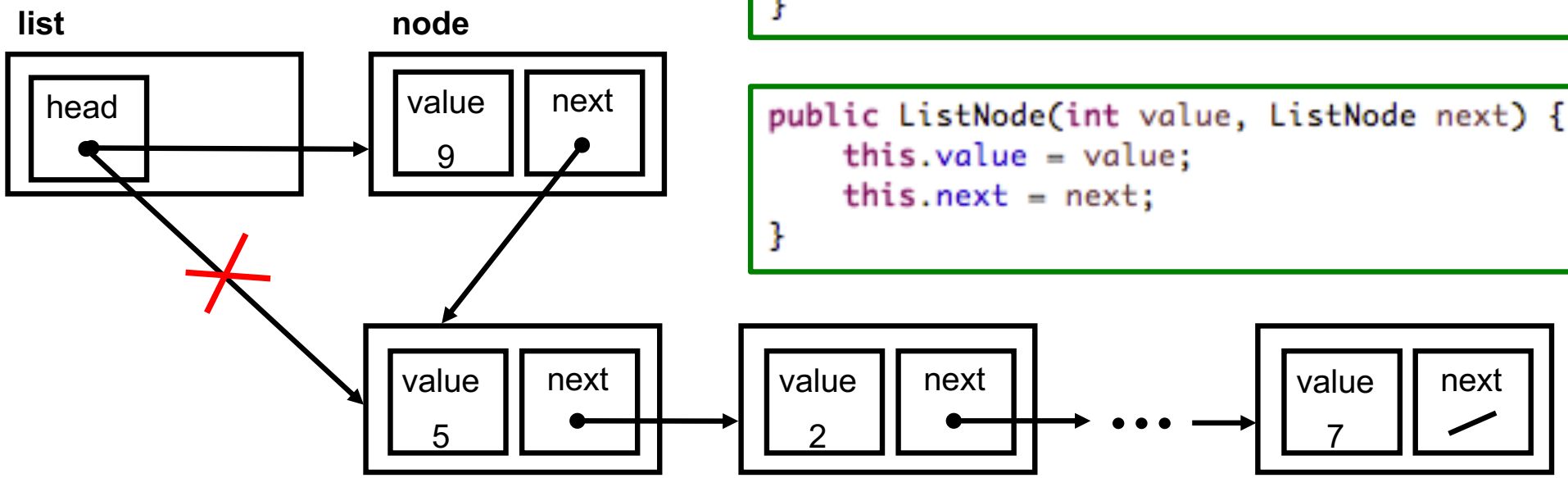
Lineare Liste: Implementierung der Operationen 1

- addFirst: Element × Liste → Liste
  - fügt ein neues Element am Beginn einer Liste ein

```
void addFirst(int elem);
```

```
public void addFirst(int value) {
    ListNode node = new ListNode(value, head);
    head = node;
}
```

```
public ListNode(int value, ListNode next) {
    this.value = value;
    this.next = next;
}
```



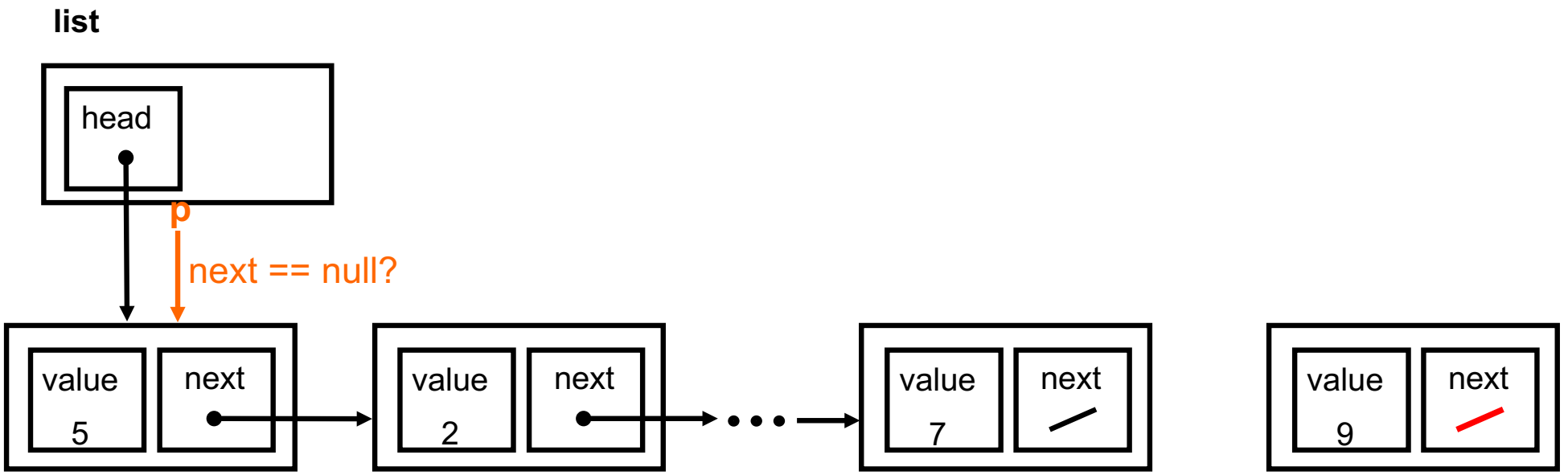

```
list.addFirst(9);
```

Z.B. in *main* wird ein neues Element hinzugefügt

Lineare Liste: Implementierung der Operationen 2

- addLast: Element × Liste → Liste
  - fügt ein neues Element am Ende einer Liste ein

```
void addLast(int elem);
```



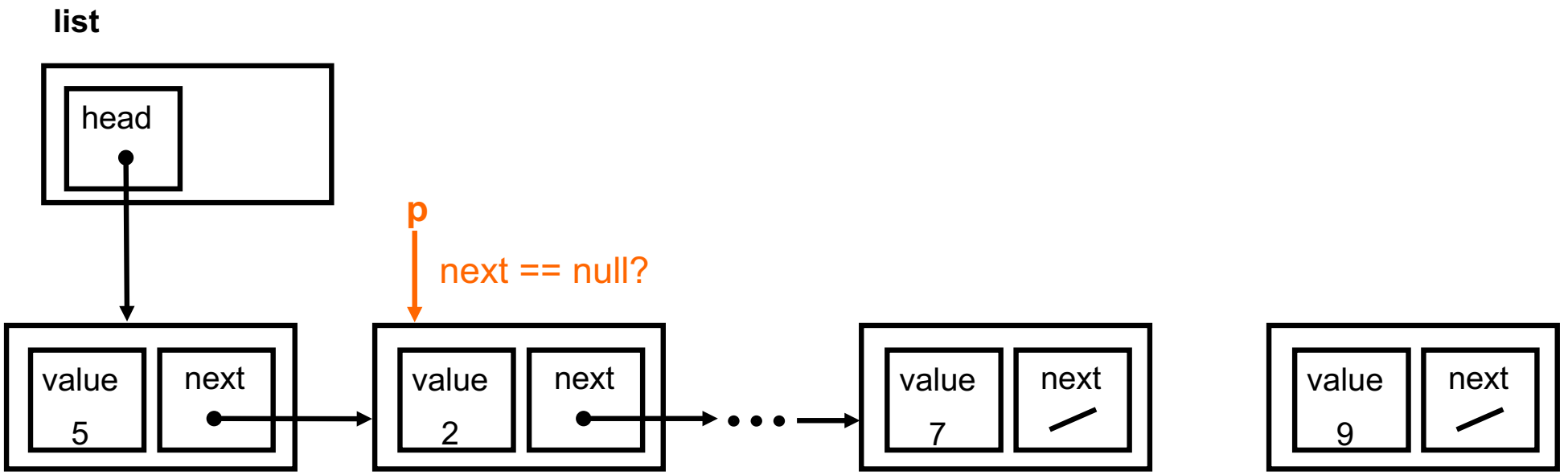

```
list.addLast(9);
```

In *main* wird ein neues Element hinzugefügt

Lineare Liste: Implementierung der Operationen 2

- addLast: Element × Liste → Liste
  - fügt ein neues Element am Ende einer Liste ein

```
void addLast(int elem);
```




```
list.addLast(9);
```

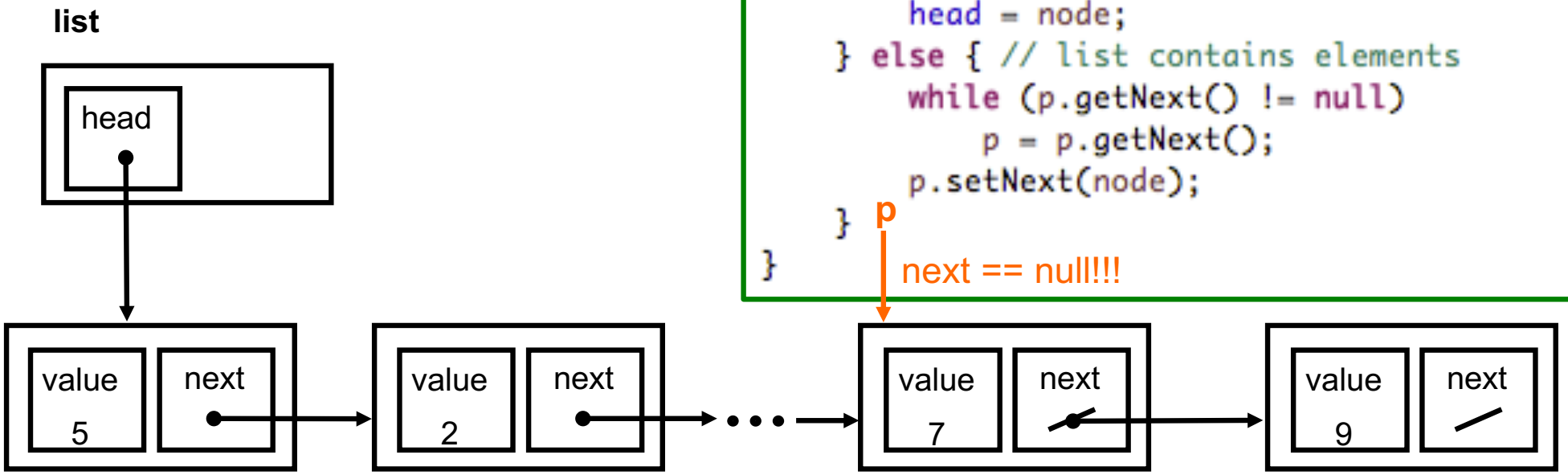
In *main* wird ein neues Element hinzugefügt

Lineare Liste: Implementierung der Operationen 2

- addLast: Element × Liste → Liste

```
void addLast(int elem);
```

```
public void addLast(int value) {
    ListNode node = new ListNode(value, null);

    // search tail of list
    ListNode p = head;
    if (p == null) { // empty list
        head = node;
    } else { // list contains elements
        while (p.getNext() != null)
            p = p.getNext();
        p.setNext(node);
    }
}
```



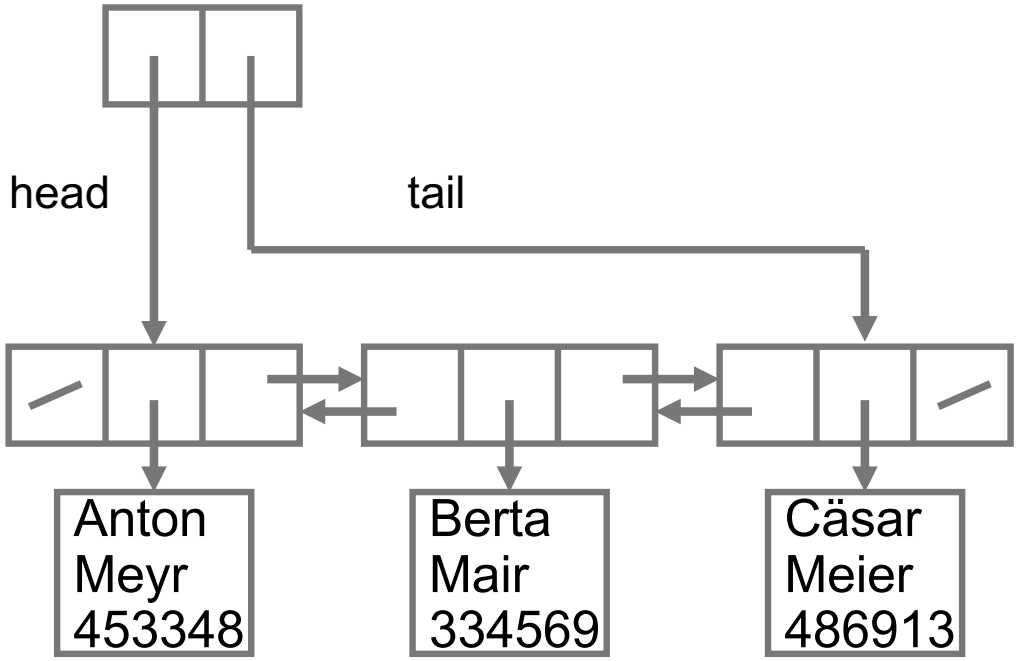

```
list.addLast(9);
```

In *main* wird ein neues Element hinzugefügt

Lineare Liste: Implementierung der Operationen 3

```
public ListImplementation() {
    head = null;
}

public boolean isEmpty() {
    return head == null;
}

public int size() {
    if (head == null) { // empty list
        return 0;
    } else { // list contains elements
        int size = 1;
        ListNode p = head;
        while (p.getNext() != null) {
            size++;
            p = p.getNext();
        }
        return size;
    }
}
```

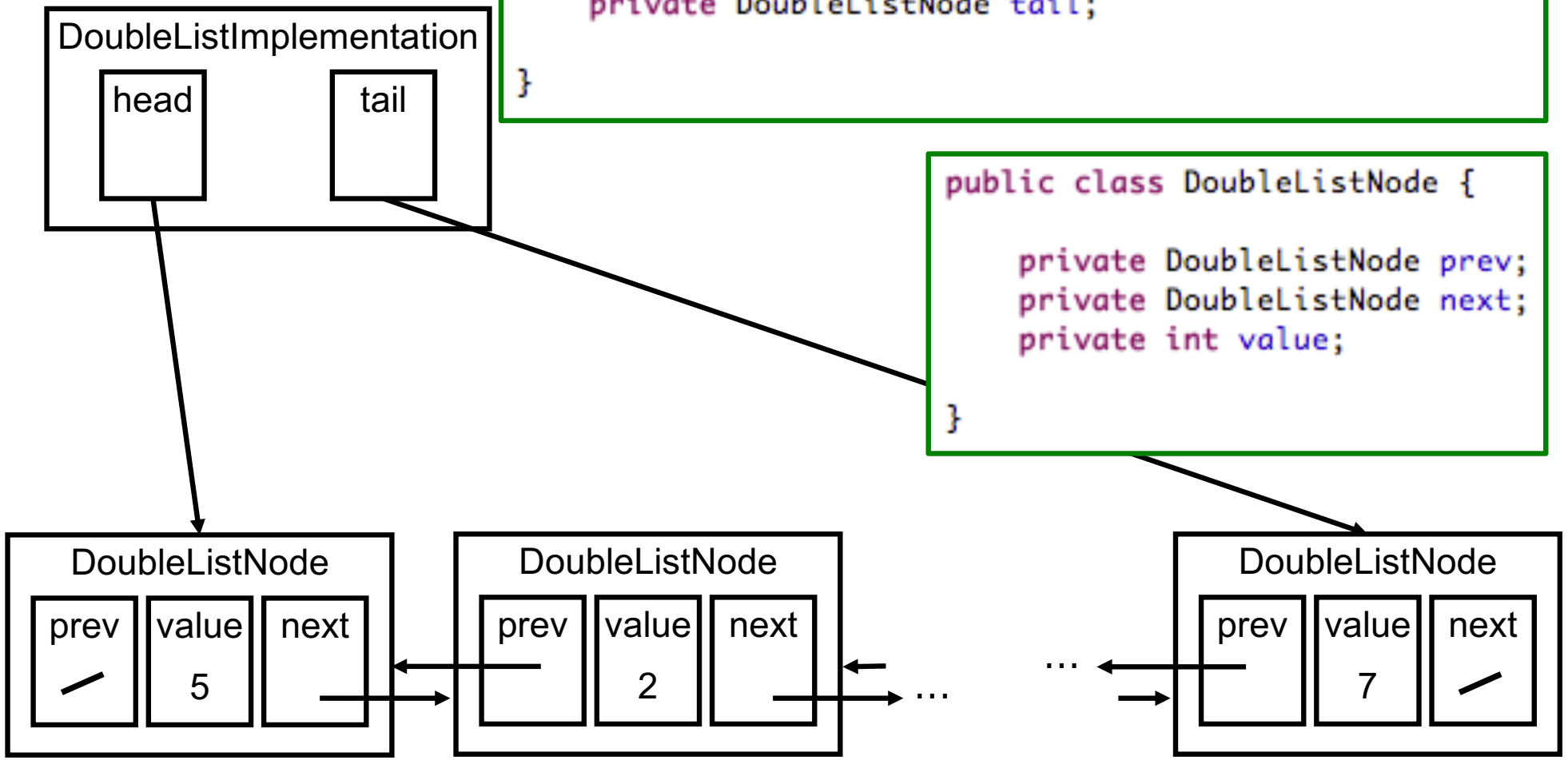
Doppelt verkettete lineare Liste 1/2



Doppelt verkettete lineare Liste 2/2

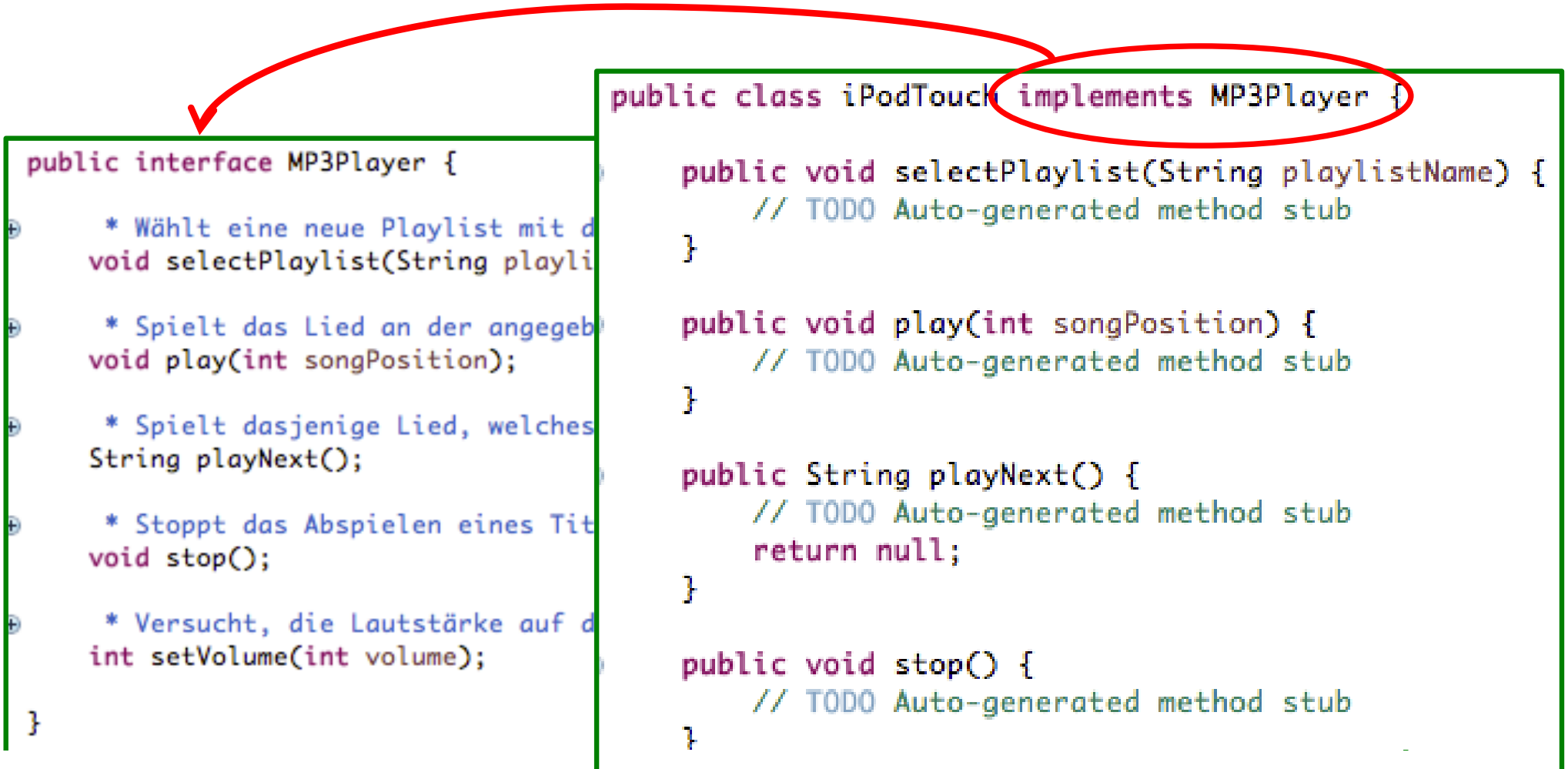
```
public class DoubleListImplementation implements List {

    private DoubleListNode head;
    private DoubleListNode tail;

}
```

```
public class DoubleListNode {

    private DoubleListNode prev;
    private DoubleListNode next;
    private int value;

}
```



Die Realisierung / Implementierung einer Schnittstelle

Man sagt:
Eine *Klasse* ***implementiert*** ein *Interface*



```
public interface MP3Player {

    /**
     * Wählt eine neue Playlist mit d
     */
    void selectPlaylist(String playli

    /**
     * Spielt das Lied an der angegeb
     */
    void play(int songPosition);

    /**
     * Spielt dasjenige Lied, welches
     */
    String playNext();

    /**
     * Stoppt das Abspielen eines Tit
     */
    void stop();

    /**
     * Versucht, die Lautstärke auf d
     */
    int setVolume(int volume);

}
```

```
public class iPodTouch implements MP3Player {

    public void selectPlaylist(String playlistName) {
        // TODO Auto-generated method stub
    }

    public void play(int songPosition) {
        // TODO Auto-generated method stub
    }

    public String playNext() {
        // TODO Auto-generated method stub
        return null;
    }

    public void stop() {
        // TODO Auto-generated method stub
    }
```

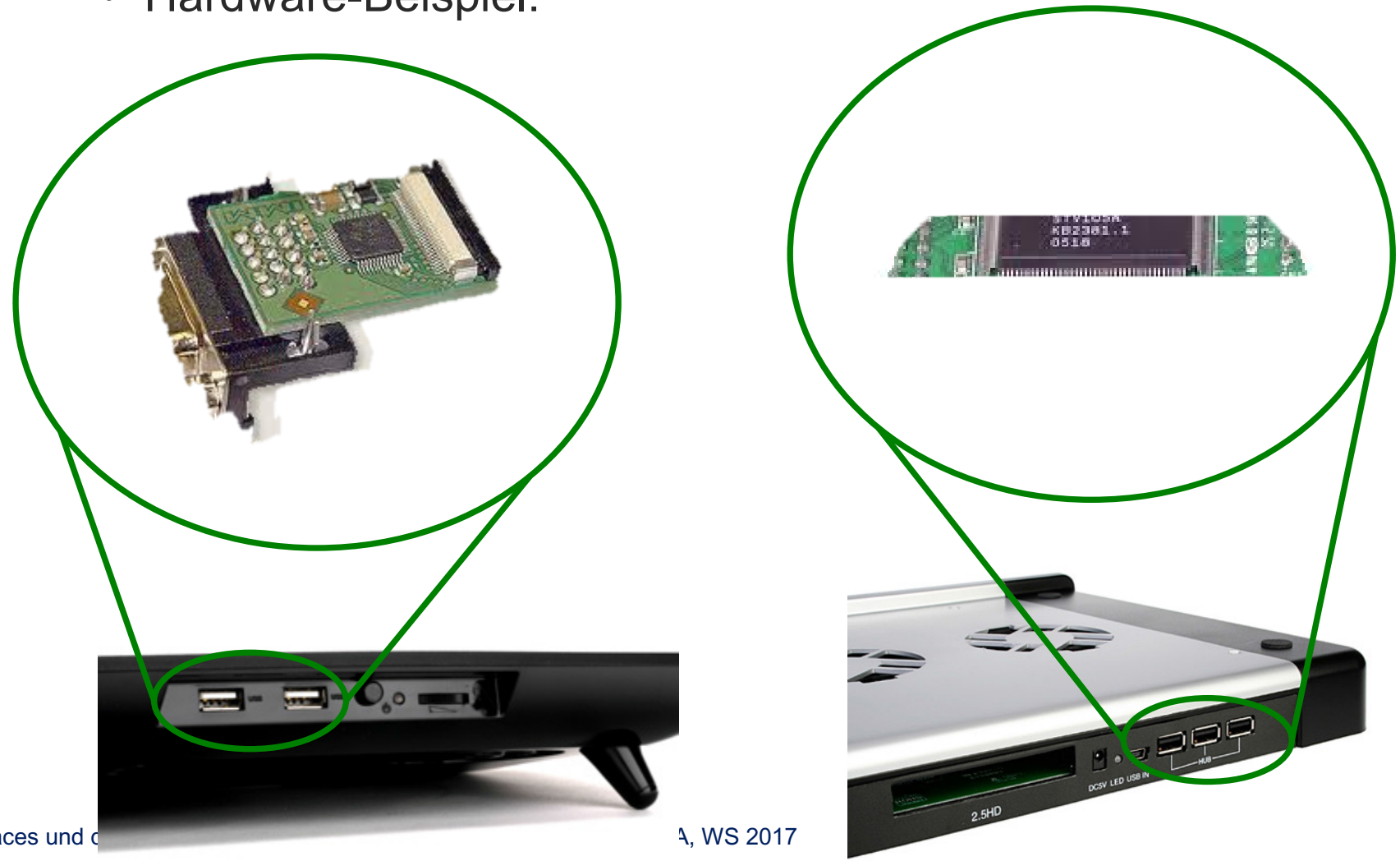
Bemerkungen

Man sagt:
Eine *Klasse* ***implementiert*** ein *Interface*

Mehrere Klassen können dasselbe Interface bedienen, so wie (vermutlich) alle Notebooks USB-Schnittstellen zur Verfügung stellen

Mehrere Möglichkeiten, eine Implementierung zur Verfügung zu stellen

- Es gibt viele Möglichkeiten, *wie* eine Komponente (Klasse) eine Schnittstelle zur Verfügung stellen kann
- Hardware-Beispiel:



Mehrere Möglichkeiten, eine Implementierung zur Verfügung zu stellen

- Es gibt viele Möglichkeiten, *wie* eine Komponente (Klasse) eine Schnittstelle zur Verfügung stellen kann
- Software-Beispiel:

```
public class iPodTouch implements MP3Player { . . .

public class AGPTEK16GB implements MP3Player { . . .

public class SonyNW_A35HN implements MP3Player { . . .
```

Interfaces müssen vollständig implementiert werden

Eine Klasse gibt an, dass sie ein Interface implementiert

→Sie *muss* dafür *alle* Methoden zur Verfügung stellen, die das Interface fordert (sonst wäre es keine vollständige Implementierung ≡ wie wenn bei der Hardware-Komponente einige Pins fehlen würden)

→Diese Methoden müssen alle *öffentlich sichtbar* (*public*) sein

```
public interface MP3Player {

    * Wählt eine neue Playlist mit d
    void selectPlaylist(String playli

    * Spielt das Lied an der angegeb
    void play(int songPosition);

    * Spielt dasjenige Lied, welches
    String playNext();

    * Stoppt das Abspielen eines Tit
    void stop();

    * Versucht, die Lautstärke auf d
```

```
public class iPodTouch implements MP3Player {

    public void selectPlaylist(String playlistName) {
        // TODO Auto-generated method stub
    }

    public void play(int songPosition) {
        // TODO Auto-generated method stub
    }

    public String playNext() {
        // TODO Auto-generated method stub
        return null;
    }
```

Erzeugen und Benutzen mehrerer *verschiedener* CD-Spieler

```
void twoDifferentPlayers() {

    MP3Player apple = new iPodTouch();
    MP3Player sony = new SonyNW_A35HN();

    apple.selectPlaylist("Metal");
    apple.play(3);
    int realVolume = apple.setVolume(100);

    sony.selectPlaylist("Best of Pink Floyd");
    sony.play(25);
    String songName = sony.playNext();

}
```

Bemerkungen

Man sagt:
Eine *Klasse* ***implementiert*** ein *Interface*

Eine Klasse kann gleichzeitig *mehrere* Interfaces implementieren (so wie ein Notebook gleichzeitig mehrere Schnittstellen zur Verfügung stellt)

Bemerkungen

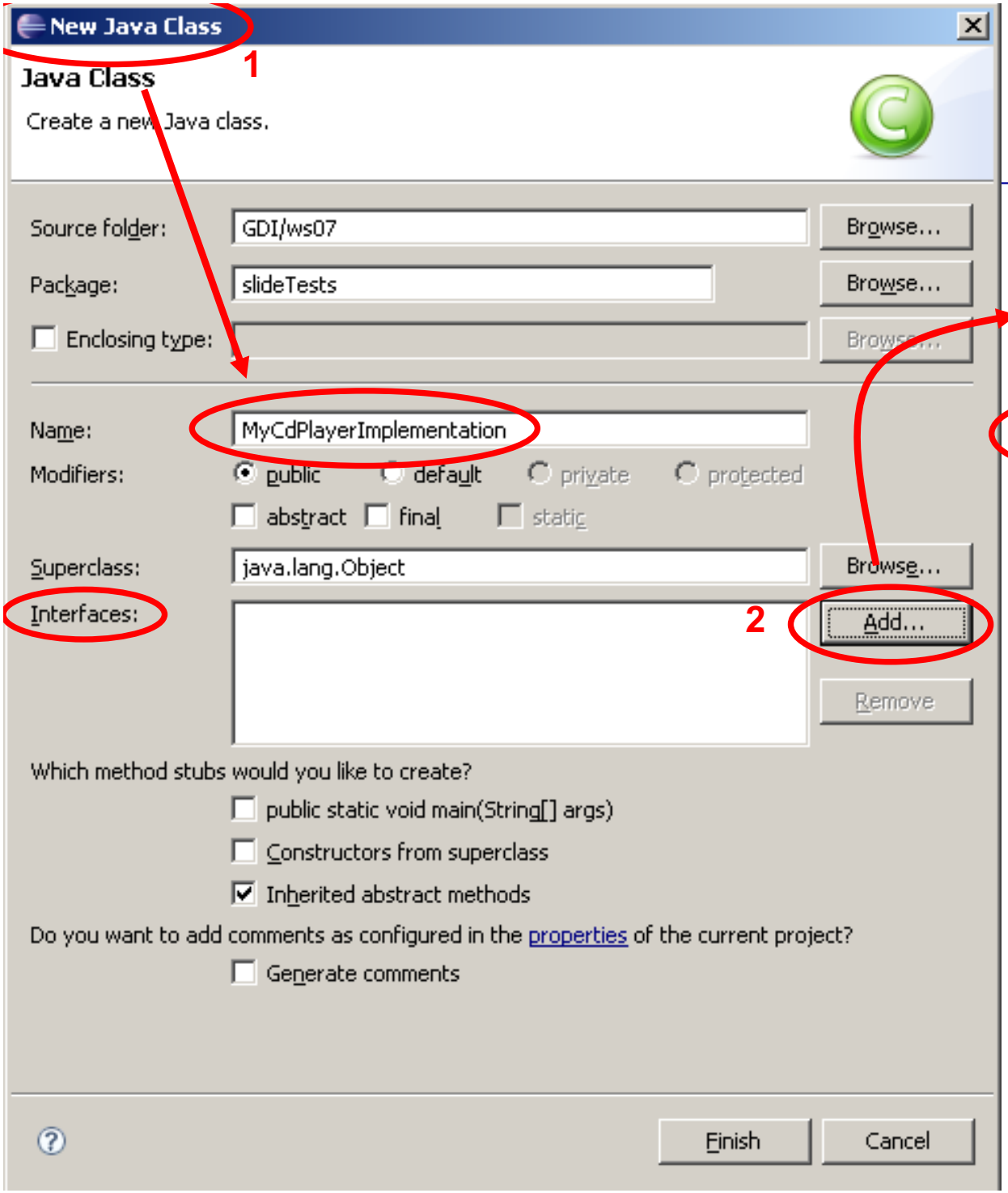
Man sagt:
Eine *Klasse* ***implementiert*** ein *Interface*

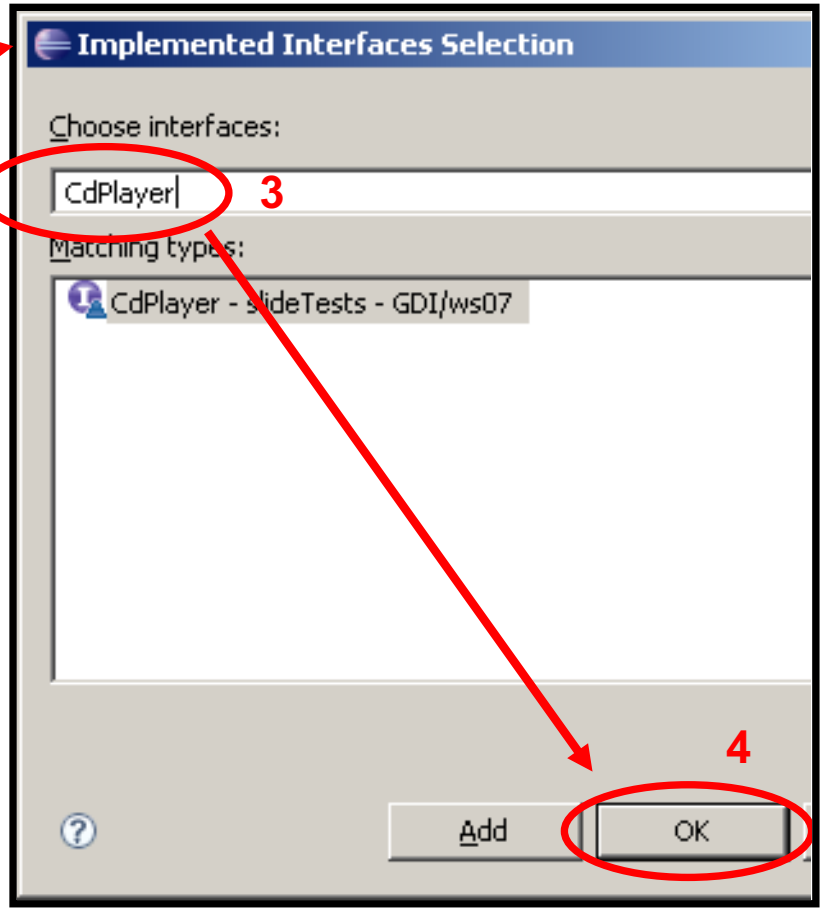
Eine Klasse kann gleichzeitig *mehrere* Interfaces implementieren

Sie muss dann *alle Methoden aller* Interfaces implementieren

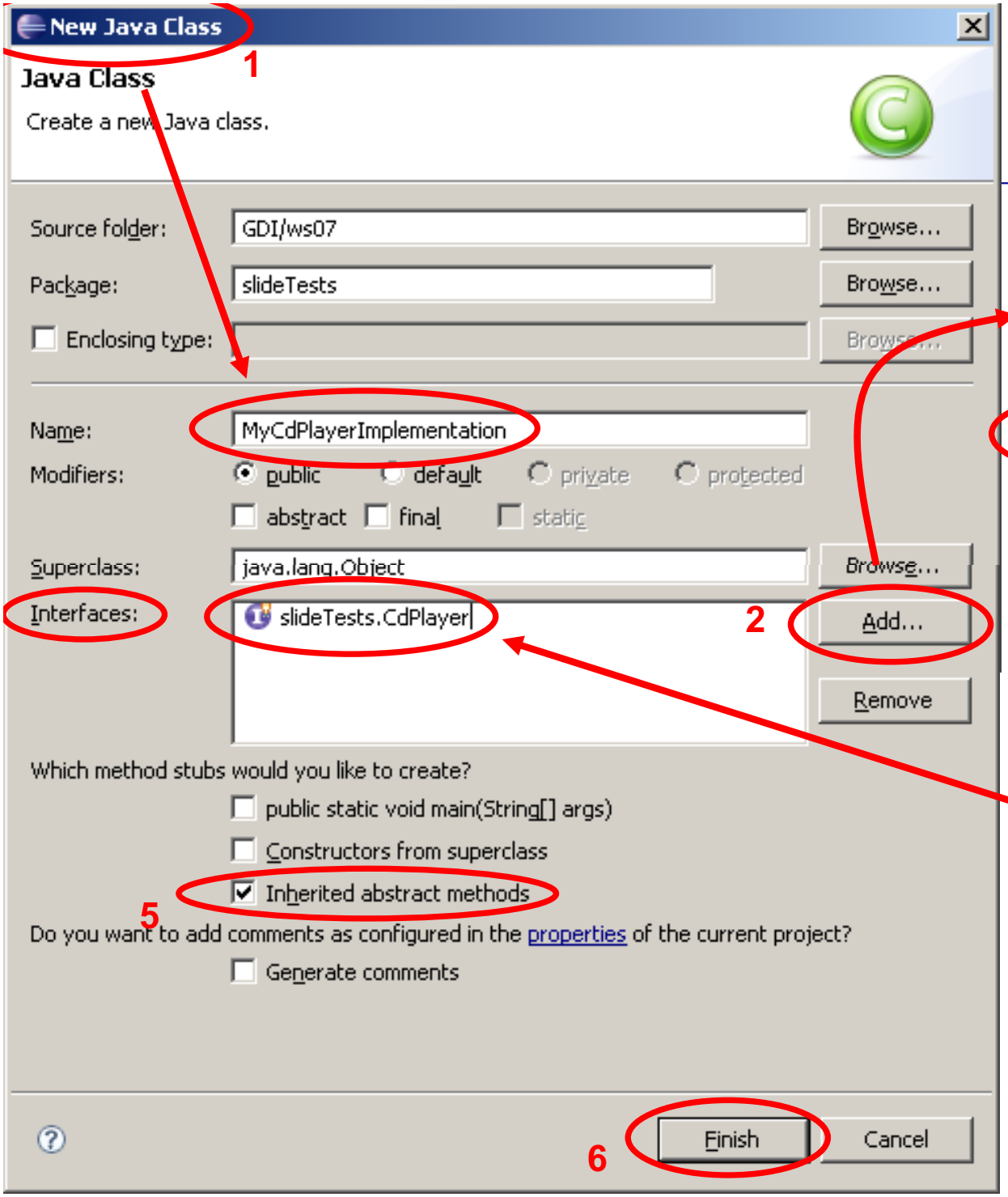
```
public class iPhone implements MP3Player, PodcastPlayer, Phone {

    public void call(long phoneNumber) {
        // TODO Auto-generated method stub
    }

    public void downloadPodcast(String url) {
        // TODO Auto-generated method stub
    }

    public void selectPlaylist(String playlistName) {
```

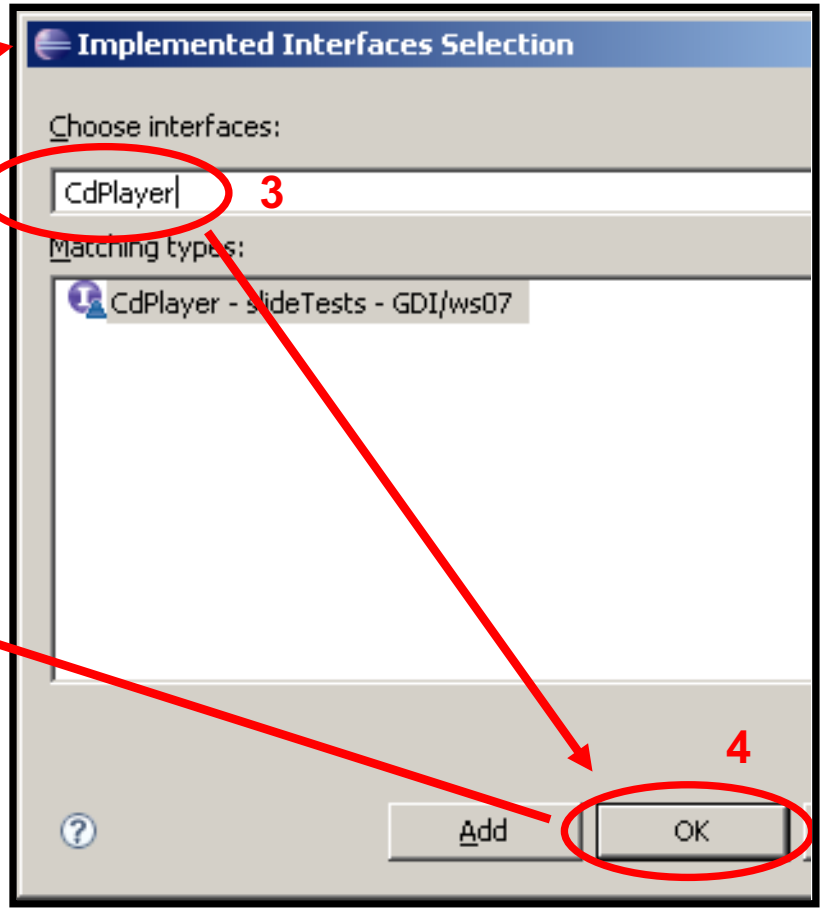
Die Erzeugung in Eclipse (Variante 1)

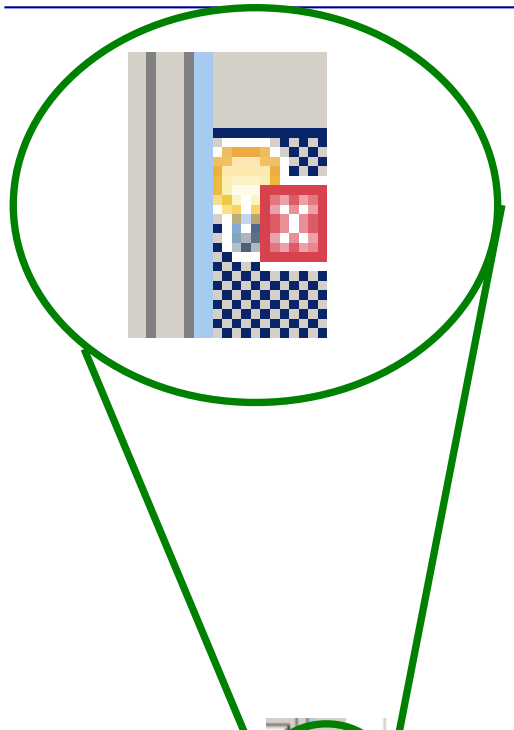


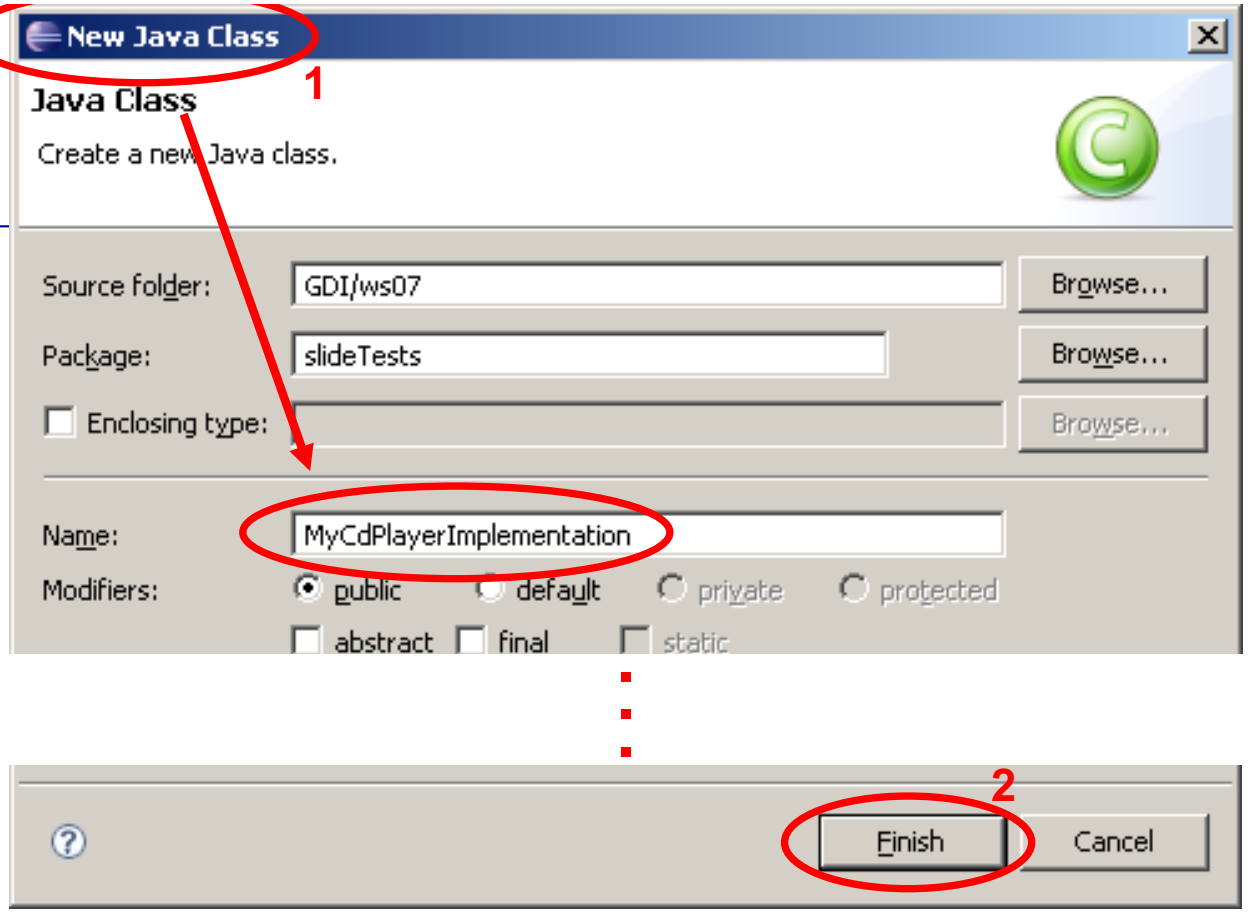


Die Erzeugung in Eclipse V1









3

Klick mit der linken Maustaste

```
public class MyCdPlayerImplementation implements CdPlayer {

}
```

```
public class MyCdPlayerImplementation implements CdPlayer {

}
```

4

- Add unimplemented methods
- Make type 'MyCdPlayerImplementation' abstract
- Rename in file (Ctrl+2, R direct access)

6 method(s) to implement:

- slideTests.CdPlayer.eject()
- slideTests.CdPlayer.pause()
- slideTests.CdPlayer.play()
- slideTests.CdPlayer.playNext()
- slideTests.CdPlayer.playPrevious()
- slideTests.CdPlayer.stop()

Die Erzeugung in Eclipse (Variante 2)

Das Ergebnis (ohne dass wir tippen/kopieren müssen)

```
public class MyCdPlayerImplementation implements CdPlayer {
    public void eject() {
    }
    public void pause() {
    }
    public void play() {
    }
    public void playNext() {
    }
    public void playPrevious() {
    }
    public void stop() {
    }
}
```