9. Kapitel

DYNAMISCHE DATENSTRUKTUREN (TEIL 1)

LISTEN

Algorithmen & Datenstrukturen

Prof. Dr. Wolfgang Schramm

Übersicht

Lernziele des Kapitels

- Sie verstehen, was dynamische Datenstrukturen sind und wozu man sie braucht.
- Sie kennen Listen und deren Varianten.
- Sie können Listen nutzen.
- Sie können Listen implementieren mit verschiedenen Mechanismen.
- Sie können den ADT Liste einsetzen für die Realisierung anderer ADT.

Inhalt

- Dynamische Datenstrukturen
- ADT Liste
  - Operationen
  - Regeln & Axiome
- Implementierung der Funktionen des ADT Liste in Java
- Auf dem ADT Liste basierende ADTs
  - ADT Stack
  - ADT Queue
- Listenarten

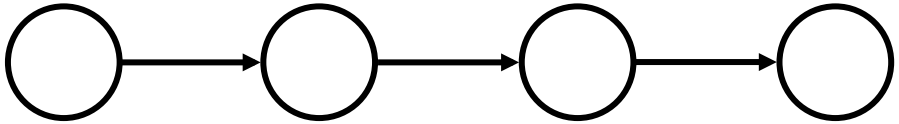
Dynamische Datenstrukturen 1/3

- Bisher: Arbeit auf Strukturen fester Länge (Arrays)
- Oft weiß man nicht im Vorhinein, wieviele Elemente in einer Datenstruktur untergebracht werden müssen ⇒ man braucht Datenstrukturen, die beliebig viele Elemente aufnehmen können.
- Diese Datenstrukturen können wachsen und schrumpfen. Man nennt sie deshalb auch dynamische Datenstrukturen.
  - Die Elemente (Knoten) dieser Strukturen werden in Java zur Laufzeit (→ dynamisch) mittels `new` erzeugt und dann verkettet.
  - Solange Speicher vorhanden ist, können neue Elemente (Knoten) erzeugt werden und an die Datenstruktur angehängt werden.
- Die wichtigsten dynamischen Datenstrukturen sind:
  - Listen
  - Bäume
  - Graphen

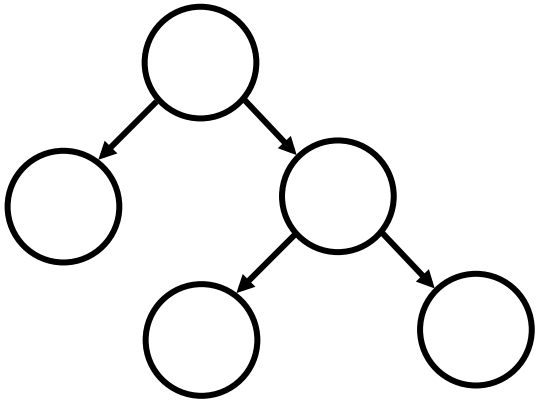
Dynamische Datenstrukturen 2/3

Grundlegende Dynamische Datenstrukturen

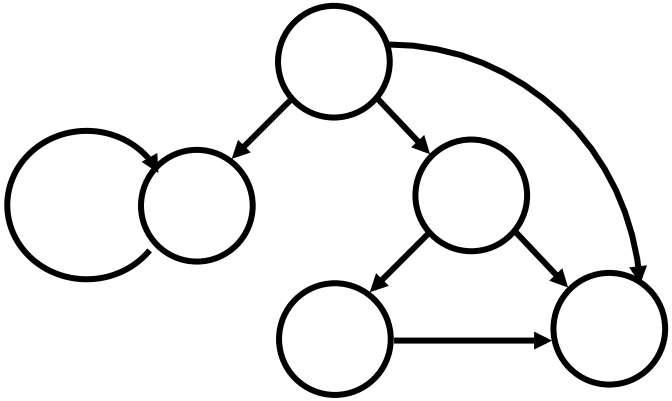
Liste



Baum



Graph



Dynamische Datenstrukturen 3/3

- **Liste** (list)
  - Jeder Knoten außer dem letzten hat genau einen Nachfolger.
  - Merkt man sich in einem Knoten immer nur den Nachfolger, dann handelt es sich um eine Lineare Liste (linked list). Es gibt weitere Listenvarianten.
- **Baum** (tree)
  - Jeder Knoten kann mehrere Nachfolger haben. Jeder Knoten hat aber höchstens einen Vorgänger.
  - Ist die Anzahl der Nachfolger maximal 2, dann nennt man den Baum Binärbaum (binary tree). Bei n Nachfolgern spricht man von n-äern Bäumen (n-ary trees).
- **Graphen** (graph)
  - Jeder Knoten kann mehrere Nachfolger und mehrere Vorgänger haben.

Liste – Operationen 1/2

- Listen sind ein guter Kandidat, um als ADT realisiert zu werden:
  - Werte: sind abhängig vom Elementtyp (ähnlich wie Arrays)
  - **Operationen / Functions (eine Auswahl):**
    - addFirst - fügt ein neues Element am Beginn einer Liste ein

      addFirst: Element × Liste → Liste
    - addLast - hängt ein neues Element am Ende einer Liste an

      addLast : Element × Liste → Liste
    - removeFirst - löscht das erste Element der Liste – Vorbedingung: Liste ≠ ∅

      removeFirst: Liste → Liste
    - getFirst - liefert das erste Element der Liste – Vorbedingung: Liste ≠ ∅

      getFirst: Liste → Element
    - getLast - liefert das letzte Element der Liste – Vorbedingung: Liste ≠ ∅

      getLast: Liste → Element
    - isEmpty - liefert true genau dann, wenn die Liste leer ist

      isEmpty: Liste → boolean

Liste – Operationen 2/2

- **Operationen / Functions (eine Auswahl) - Fortsetzung:**
  - empty – erzeugt eine neue leere Liste

    empty: → Liste
  - contains - sucht ein Element in einer Liste; Ergebnis = true, wenn Element in der Liste, sonst false

    contains: Element × Liste → boolean
  - clear – entfernt alle Elemente aus der Liste

    clear : Liste → Liste
  - size - liefert die Länge der Liste, d.h. die Anzahl der Elemente

    size: Liste → int
  - clone - liefert eine (flache) Kopie der Liste

    clone: Liste → Liste
  - delete – entfernt ein Element aus der Liste, sofern es drin ist

    delete : Element × Liste → Liste

Liste – Axiome

- Axioms (Auswahl)
  - isempty (empty()) = true

    Neu erzeugte Liste ist leer.
  - isempty (addFirst (L, e)) = false

    Sobald ein Element eingefügt wurde,
    ist die Liste nicht mehr leer.
  - getFirst (addFirst (L, e)) = e

    Wird vorne ein Element hinzugefügt,
    so erhalte ich es wieder, wenn ich das vorderste Element hole.
  - removeFirst (addFirst (L, e)) = L

    Wird vorne ein Element hinzugefügt, und anschließend das vorderste Element entfernt, so erhalte ich die ursprüngliche Liste wieder.
  - ...

Lineare Liste – Vorüberlegung zur Implementierung

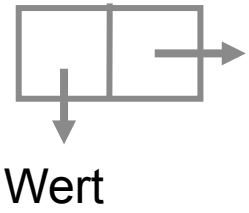
- Lineare Liste in EBNF:
  - Liste → Element Liste | leer.
  - leer = Liste ohne Element (leere Liste)
- Die Datenstruktur hat einen Selbstbezug, d.h. sie ist rekursiv.
- Es gibt unzählige Verwendungszwecke für Listen. Man kann damit Bücher, Werte einer Messreihe oder Wörter einer Eingabe verwalten.
- Annahmen:
  - Für die Beispielprogramme sollen einfache int-Werte verwaltet werden.
  - Man merkt sich nur das Nachfolgerelement.

- Listenelemente = Objekte, die je ein Listenelement repräsentieren.

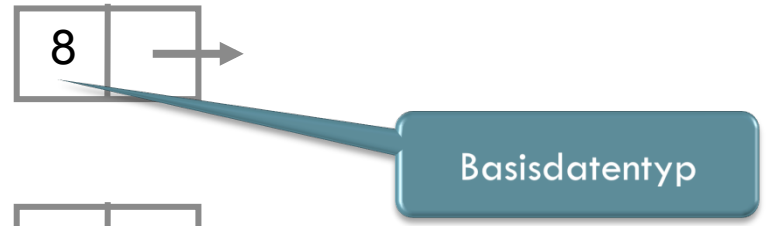
- Für die Implementierung der Liste gibt es verschiedene Varianten.
- Die Listenelemente sind nur für die Verknüpfung zuständig, nicht aber für die konkreten Inhalte, die in der Liste verwaltet werden.

Darstellung von Listen 1/2

Allgemeine Darstellung

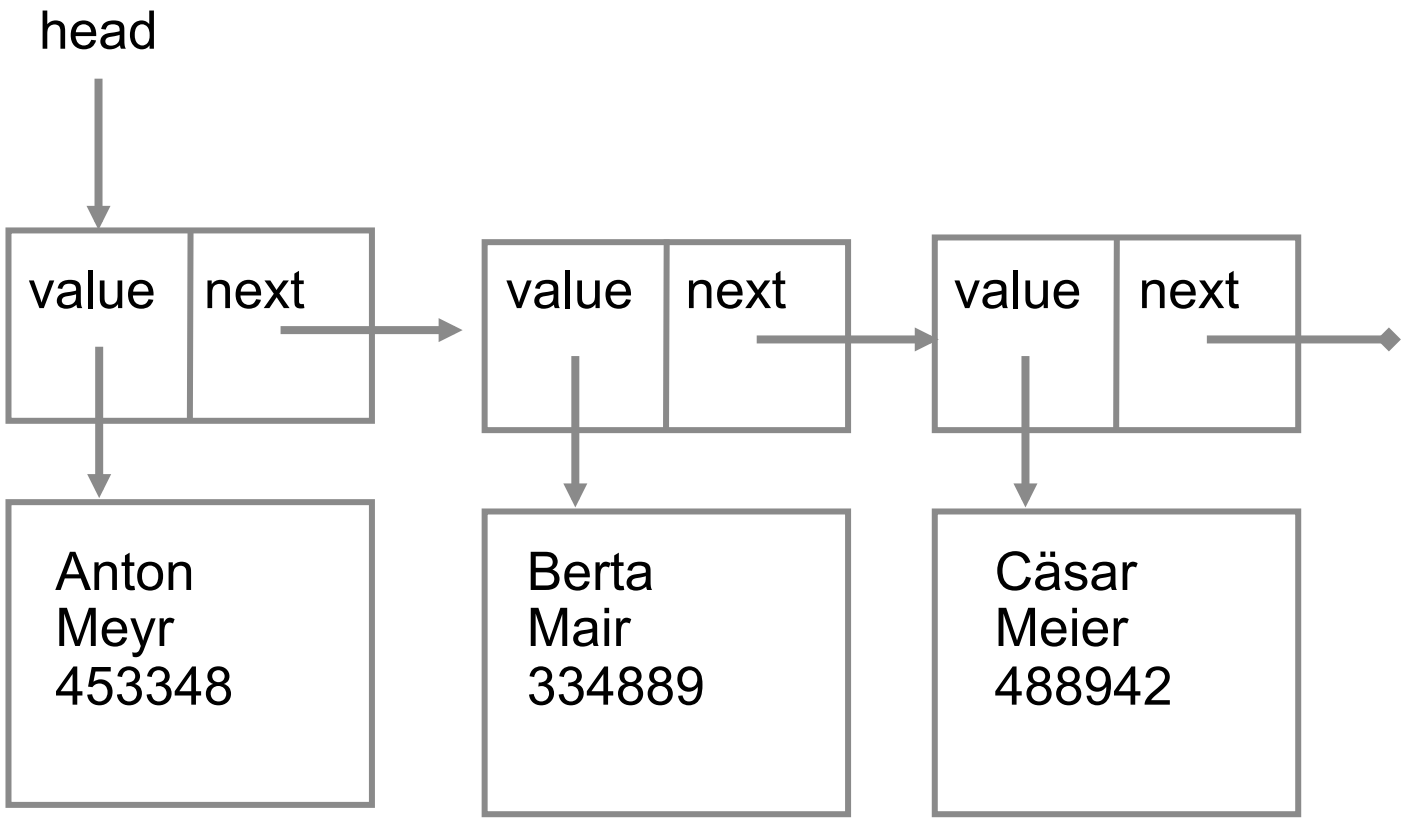


Genauer:
int-Liste

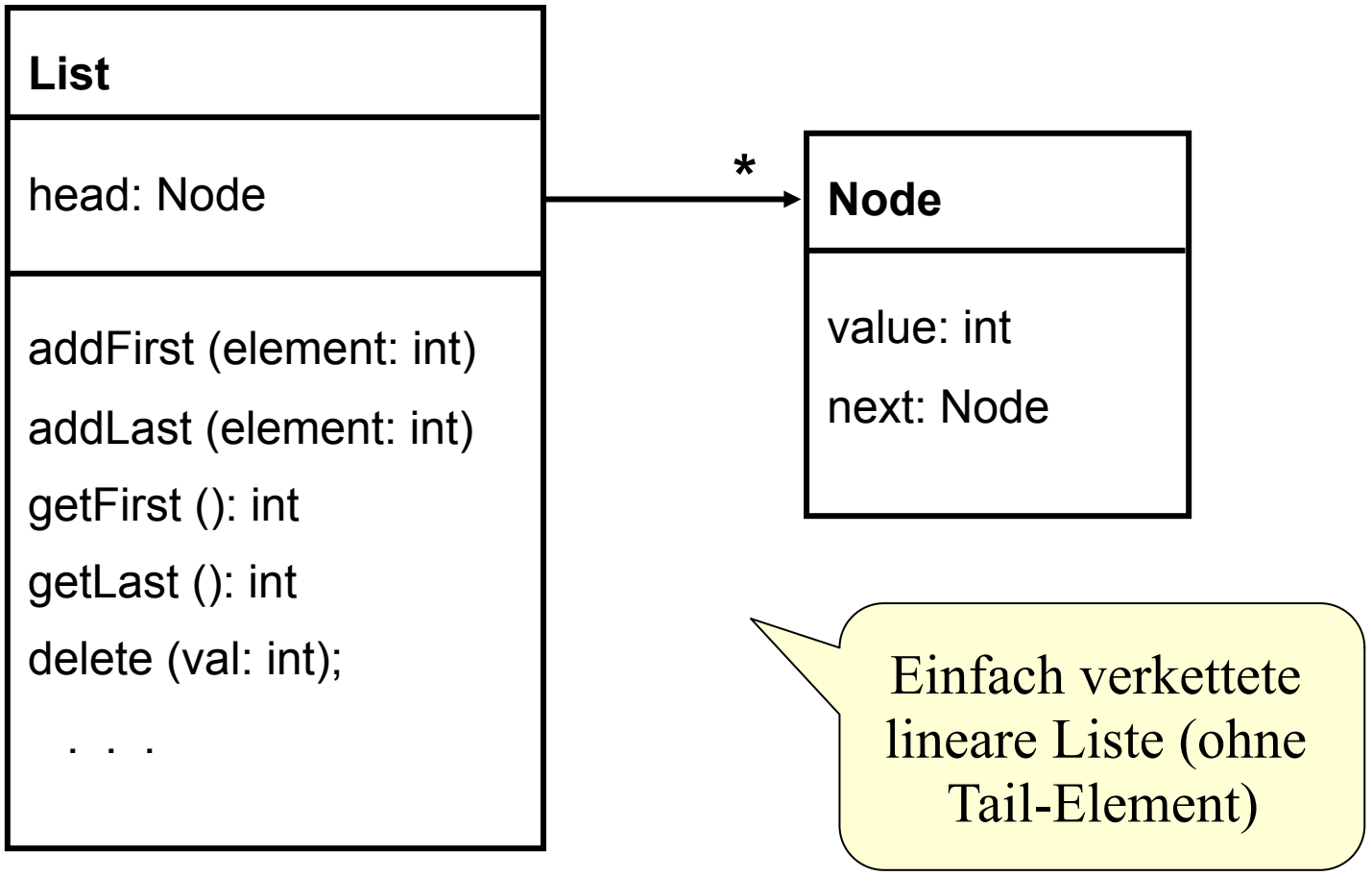


Personen-Liste

Darstellung von Listen 2/2

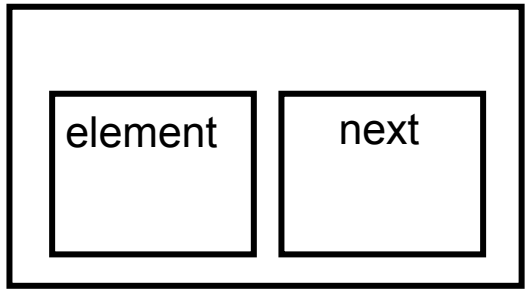


Klassendiagramm für List und Node



Lineare Liste – Implementierung in Java 1/5

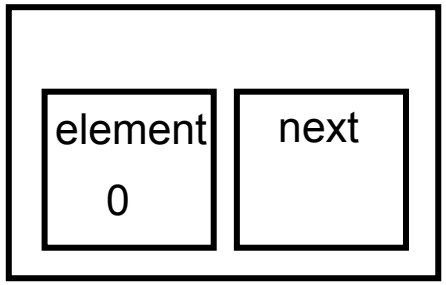
Listenelemente (allgemein)



Objekt der Klasse Node

```
class Node {
  Object element;
  Node next;
}
```

Listenelemente (zur Verwaltung von Objekten vom Typ int)



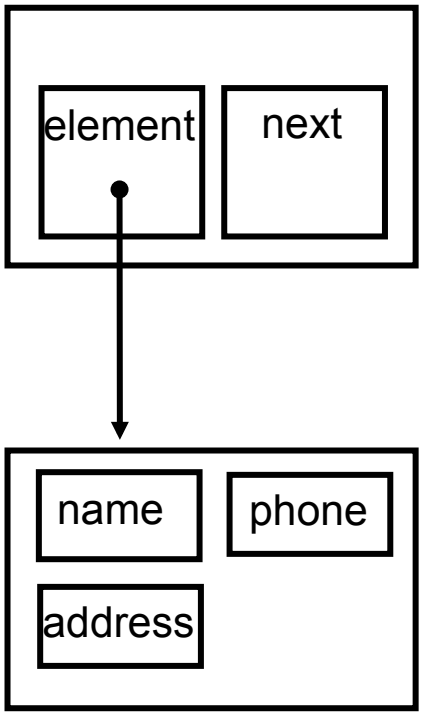
Objekt der Klasse Node

```
class Node {
  int element;
  Node next;
}
```

Lineare Liste – Implementierung in Java 2/5

Listenelemente (mit Objekten von Typ Person)

Objekt der Klasse Node

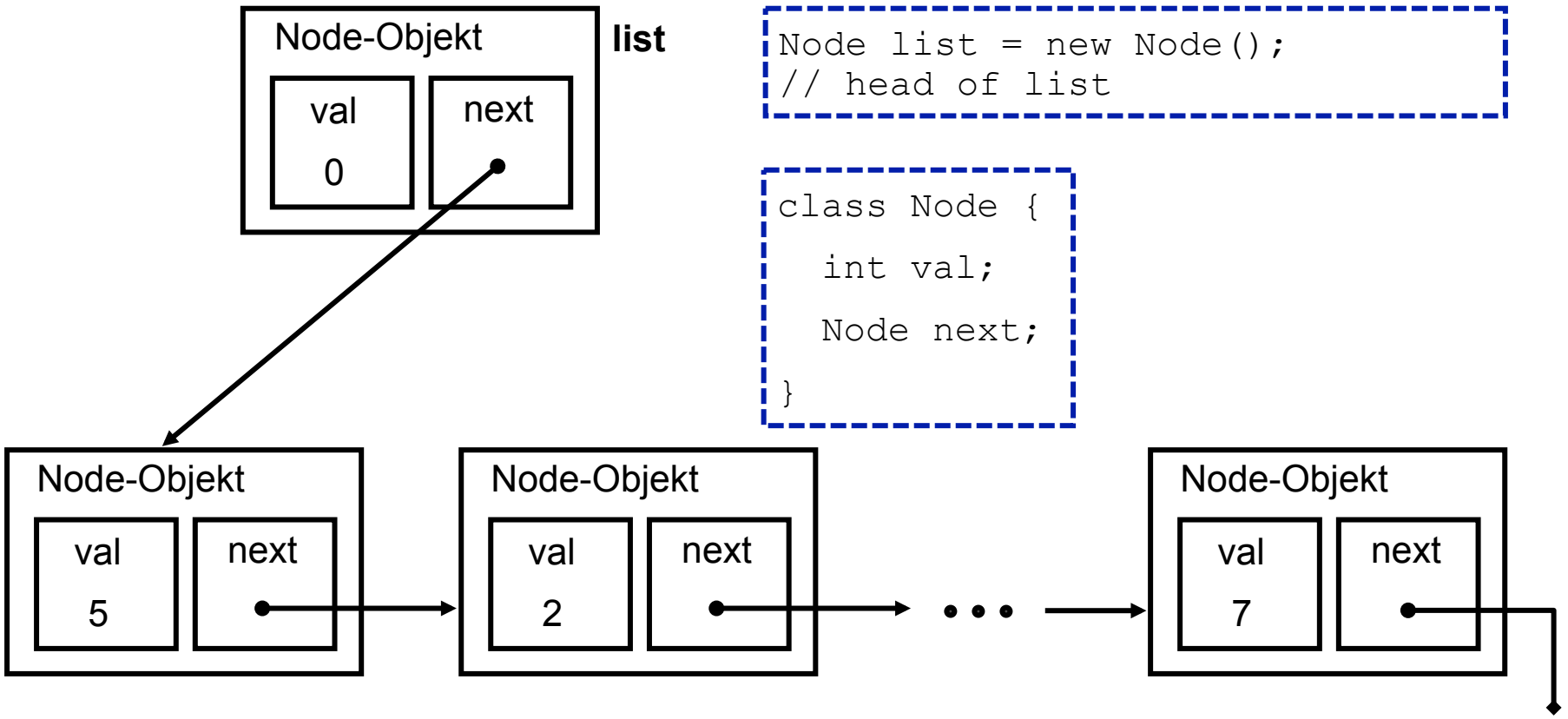


Objekt der Klasse Person

```
class Node {
  Person element;
  Node next;
}
```

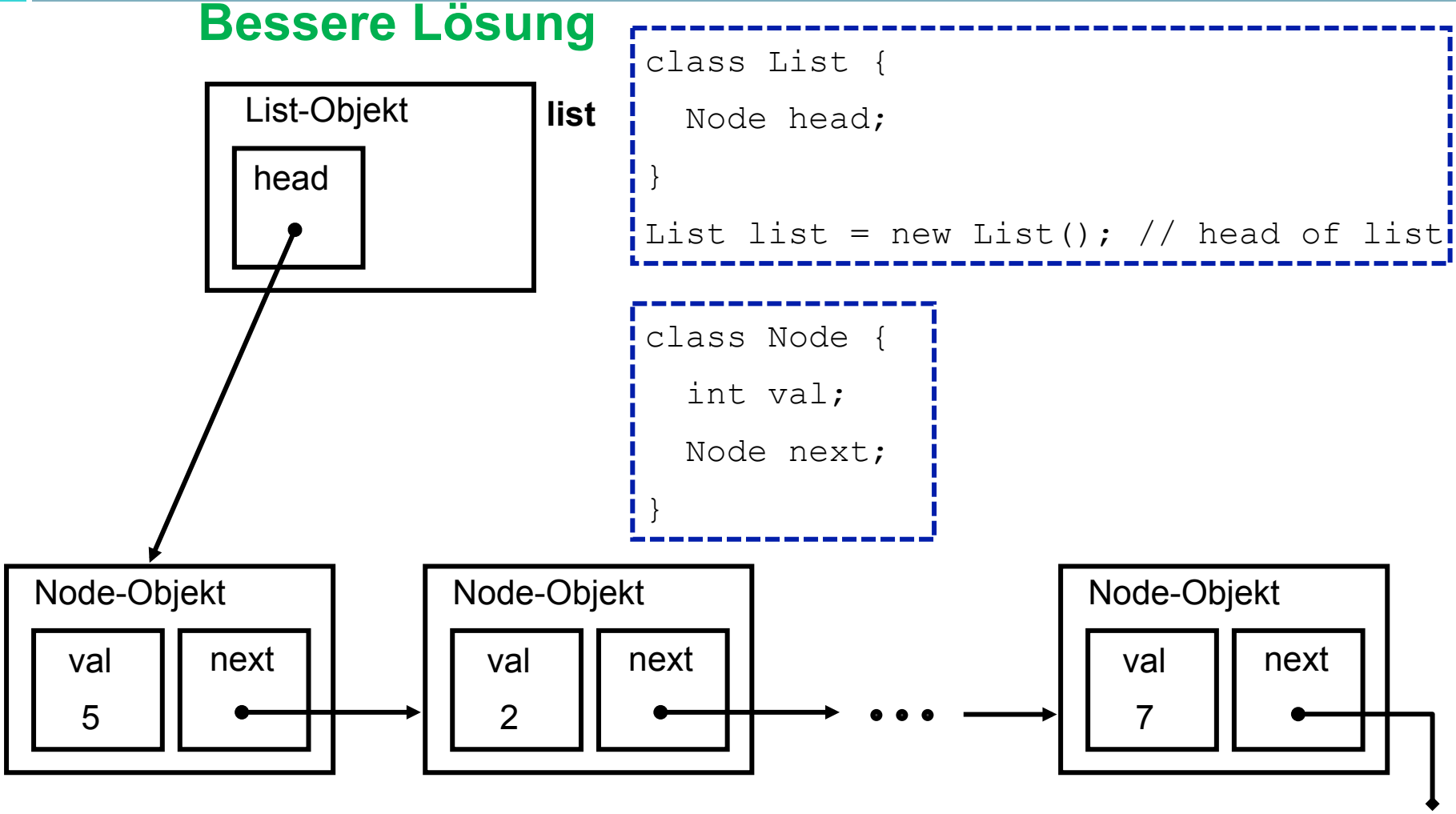
```
class Person {
  String name;
  String address;
  long phone;
}
```

Lineare Liste – Implementierung in Java 3/5



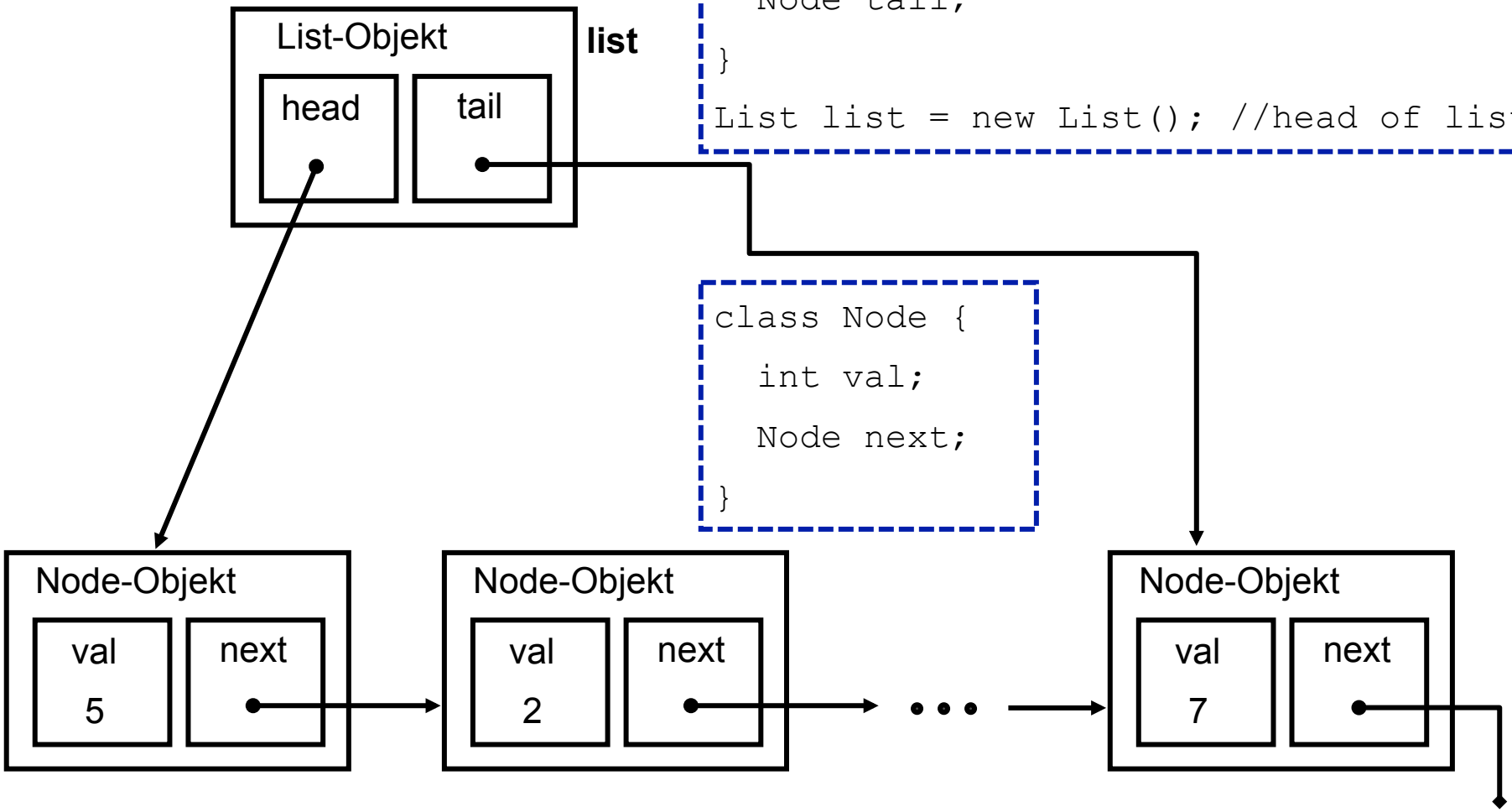
Lineare Liste – Implementierung in Java 4/5

Bessere Lösung



Lineare Liste – Implementierung in Java 5/5


```
class List {
   Node head;
   Node tail;
}
List list = new List(); //head of list
```



Listenarten

- Unsortierte Listen
- Sortierte Listen
- Doppelt verkettete Listen

Was kann man mit Listen machen?

- Stack als Liste
- Queue als Liste

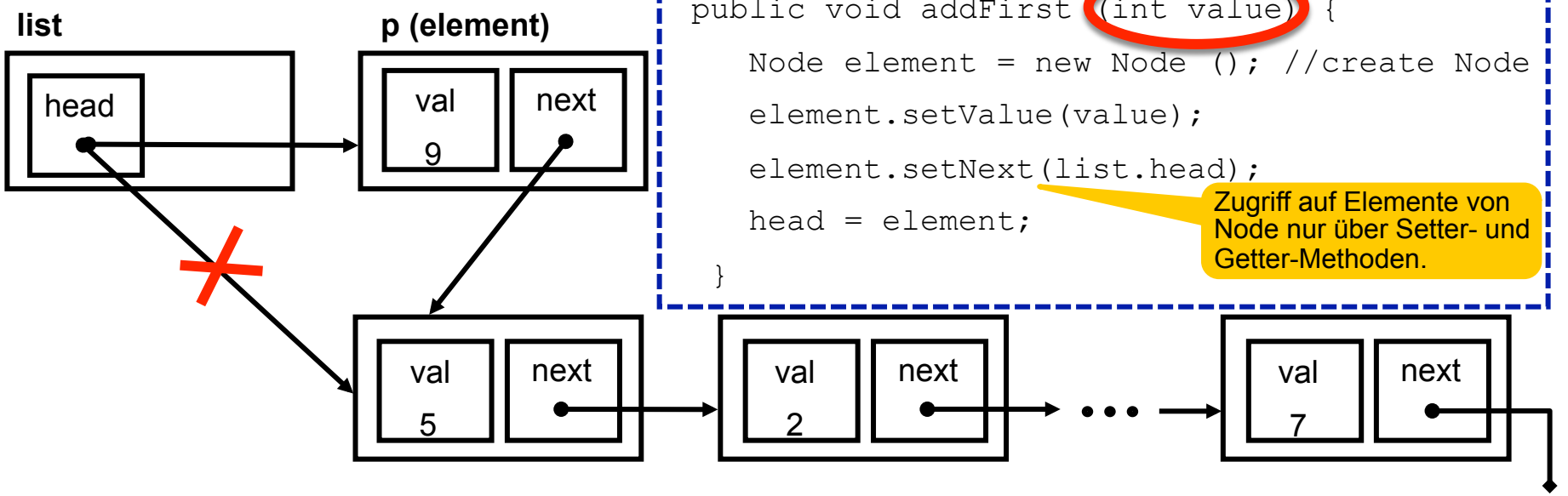
Lineare Liste – Implementierung der Operationen 1/4

- addFirst - fügt ein neues Element am Beginn einer Liste ein

  addFirst: Element × Liste → Liste

  void addFirst (int value)

Hier vereinfachend nur Basistyp – normalerweise werden Objekte übergeben



in main:

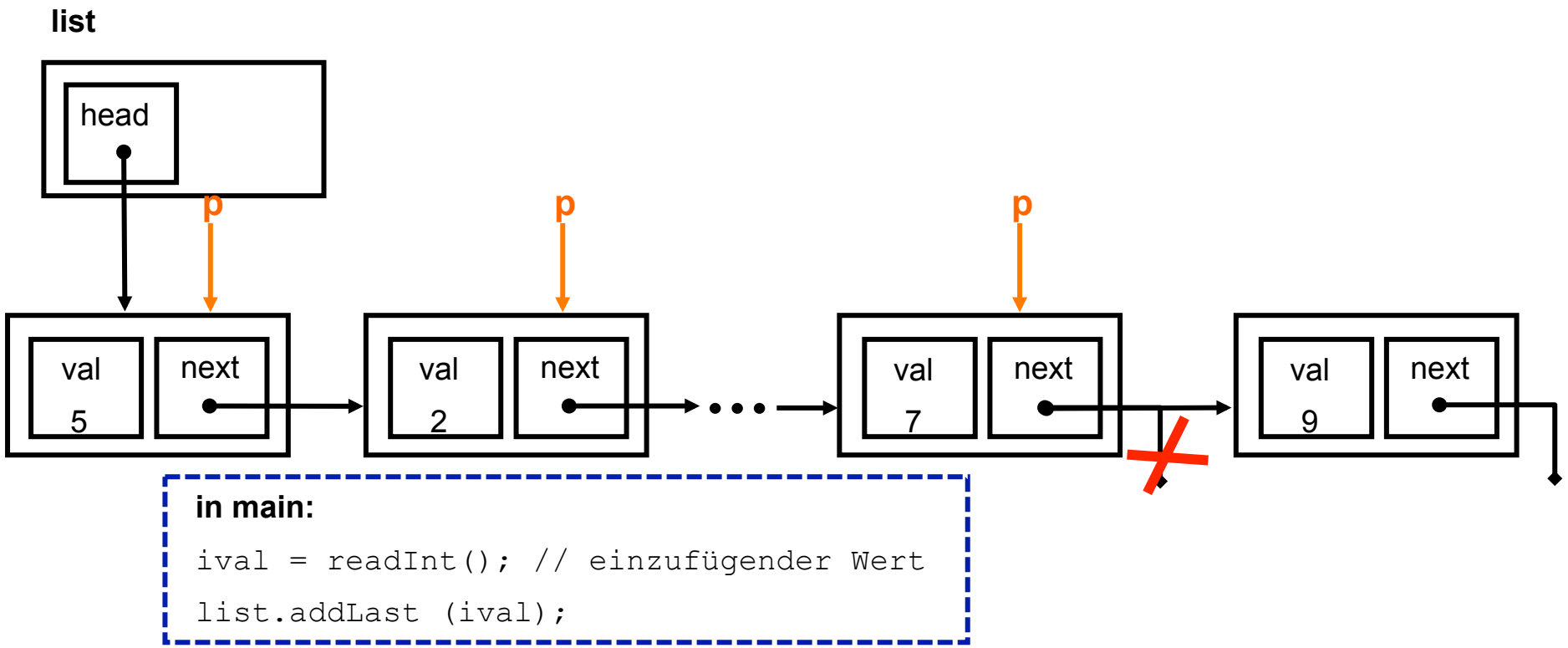
```
ival = readInt(); // einzufügender Wert
list.addFirst (ival);
```

Lineare Liste – Implementierung der Operationen 2/4

- addLast - fügt ein neues Element am Ende einer Liste ein

  addLast : Element × Liste → Liste

  ```
  void addLast (int value)
  ```



Lineare Liste – Implementierung der Operationen 3/4


```
public void addLast (int value) {
  Node element = new Node (); //create Node
  element.setValue(value);
  element.setNext(null);
  // search tail of list
  Node p = head;
  while (p != null && p.getNext() != null) p = p.getNext();
  // append at tail
  if (p == null) // empty list
    head = element;
  else // list contains elements
    p.setNext(element);
 }
```

Lineare Liste – Implementierung der Operationen 4/4


```
public void addLast (int value) {
   Node element = new Node (value, null); //create Node

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

Alternative Implementierung

Lineare Liste als Klasse mit Methoden – Implementierung in Java 1/6


```
class Node {
  int  val;  // normally contains values >= 0
  Node next;

// >>>>>>>>>>    constructors     <<<<<<<<<<<<<<

  public Node (int value, Node n) {
    val = value;
    next = n;
  }

  public Node () {
    val = 0;
    next = null;
  }
```

Lineare Liste als Klasse mit Methoden – Implementierung in Java 2/6


```
// --------  public methods

public void setValue (int val) {
  this.val = val;
}

public void print () {
  print (this.val);
}

public int getValue () {
  return val;
}
```

```
public void setNext (Node n) {
  this.next = n;
}

public Node getNext () {
  return next;
}
```

Lineare Liste als Klasse mit Methoden – Implementierung in Java 3/6


```
class List {
  Node head;

// >>>>>>>>>>    constructors     <<<<<<<<<<<<<<

  public List (int value) {
    head = new Node (value, null);
  }

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

Lineare Liste als Klasse mit Methoden – Implementierung in Java 4/6


```
// ---------------  public methods

  public void addFirst (int val) {

    // inserts val at a new element at head of the list
    // element is created inside method addFirst

    Node n = new Node (val, head);
                // create a new element
    head = n;
  }

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

Lineare Liste als Klasse mit Methoden – Implementierung in Java 5/6


```
public static void main(String[] args) {
    int value;
    char c;
    List list = new List (); // new instance of list
    do {
      println ("insert list element:");
      value = readInt ();
      list.addFirst(value);
      list.addLast (value);

      readLine();
      println ("more elements to insert (y/n)?");
      c = readChar ();
      readLine();
    } while (c == 'y' || c == 'Y');

    list.print();
```

Lineare Liste als Klasse mit Methoden – Implementierung in Java 6/6


```
    do {
      Node p = removeLast();
      if (p != null)
        println ("node " + p.val + " removed");
      else
        println ("list is empty - nothing to remove");

      removeFirst();
      println ("more elements to delete (y/n)?");
      c = readChar ();
      readLine();
    } while (c == 'y' || c == 'Y');

    list.print();
  } // ~main
} // ~class List
```

Was kann man mit Listen machen?

- Man kann sie sortieren
  - Hier gibt es aber bessere Datenstrukturen (bald)
- Stack als Liste
- Queue als Liste

Stack (Kellerspeicher)

- Einfache Datenstruktur mit beschränktem Zugriff auf gespeicherte Elemente.
- Arbeitet nach dem LIFO-Prinzip (Last-In-First-Out): beim Auslesen eines Elements wird das jeweils zuletzt gespeicherte Element zuerst ausgelesen, danach das vorletzte etc.
- Der Kellerspeicher wird auch Stapel genannt: Elemente werden übereinander gestapelt und dann wieder in umgekehrter Reihenfolge vom Stapel genommen.

Stack – Operationen

- Stack als ADT:
  - **Operationen:**
    - push - legt ein neues Element auf den Stapel

      push: Element × Stack → Stack

    - pop – entfernt das oberste Element vom Stapel (und gibt es zurück)

      pop: Stack → Stack

    - top – holt das oberste Element vom Stapel (ohne es zu entfernen)

      top: Stack → Element

    - empty – erzeugt einen leeren neuen Stapel

      empty: → Stack

    - isEmpty - liefert true genau dann, wenn der Stack leer ist

      isEmpty: Stack → boolean

Stack – Axiome

- Axioms (Auswahl)
  - isempty (empty()) = true

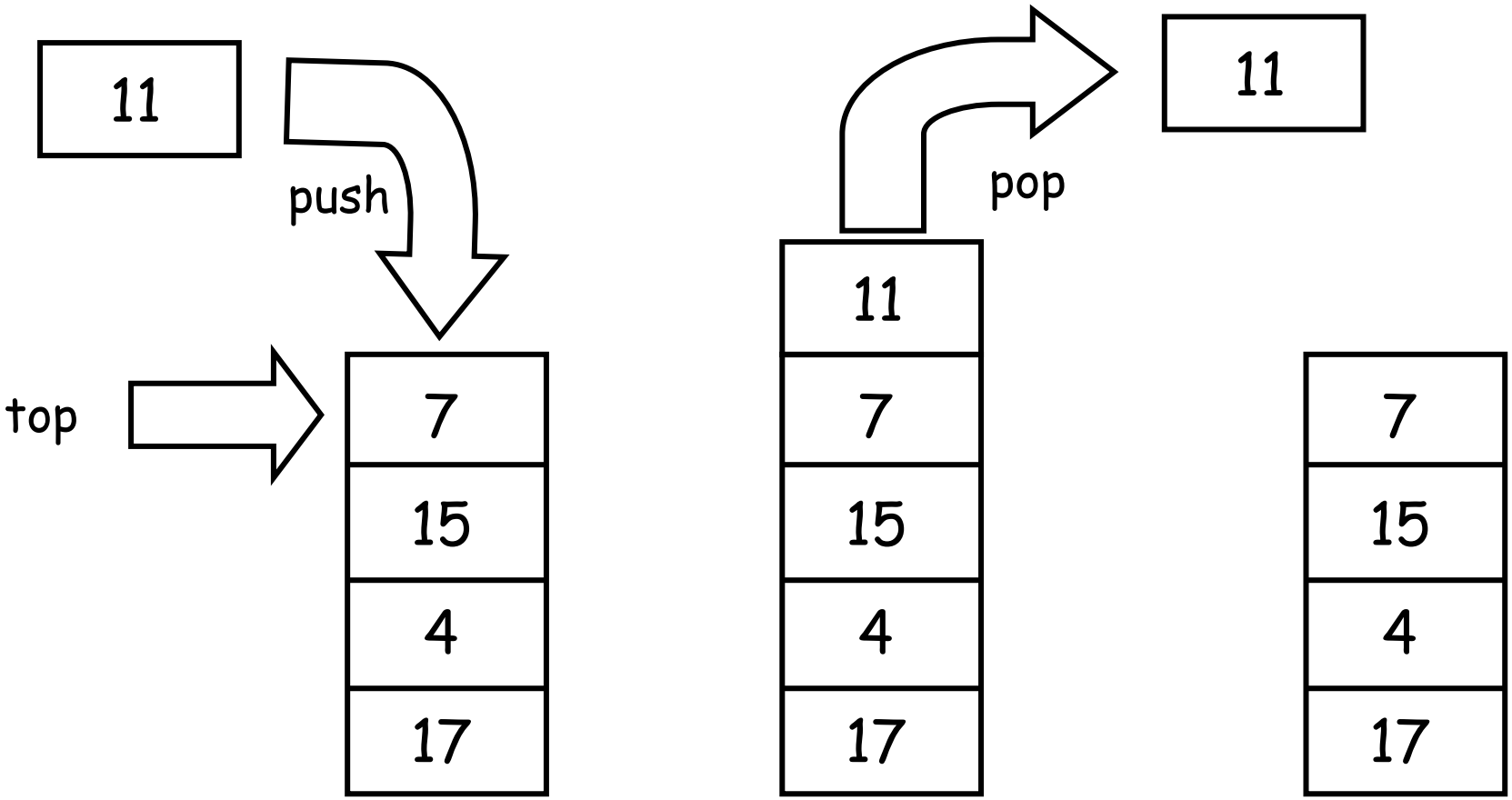
    Neu erzeugter Stack ist leer.
  - isempty (push(S, e)) = false

    Sobald ein Element eingefügt wurde,
    ist der Stack nicht mehr leer.
  - pop (push(S, e)) = S

    Das oberste Element auf dem Stack wird vom Stack genommen.
  - top (push(S, e)) = e

    Das zuletzt auf den Stack gelegte Element wird als erstes wieder vom Stack genommen.

Stack – Arbeitsweise



Stack – Anwendungsbeispiel 1/3

- Auswertung von arithmetischen Ausdrücken – wie in Taschenrechnern oder Interpretern.

- Definition einer Sprache für (einfache) arithmetische Ausdrücke in BNF:

```
expr    →   term "=" .
term    →   "(" term  "+"  term ")" | "(" term "*"  term ")" | number.
number  →   digit | number digit.
```

Beispiel: ((3 + (4 * 5)) * (6 + 7)) =

- Ziel: Auswertung beliebig tief geschachtelter arithmetischer Ausdrücke.
- Wegen der Schachtelung werden dabei noch nicht auswertbare Teilausdrücke im Keller zwischengespeichert.

Stack – Anwendungsbeispiel 2/3

- Schreibweise:
  - push (z_1, ... push (z_n, S)) ≡ z_1 . . . . . z_nS
  - ‖ x + y ‖ Berechnung des Wertes der Addition.
  - ‖ x ‖ Berechnung einer intern verarbeitbaren Zahl aus einer eingelesenen Zahl.

  - Bearbeitungsregeln für den Interpreter
  - Value (e) = eval ⟨ e, empty ⟩ (1)
  - eval ⟨ (e, S ⟩ = eval ⟨ e, (S ⟩ (2)
  - eval ⟨ * e, S ⟩ = eval ⟨ e, * S ⟩ (3)
  - eval ⟨ + e, S ⟩ = eval ⟨ e, + S ⟩ (4)
  - eval ⟨) e, y * x S ⟩ = eval ⟨) e, ‖ x * y ‖ S ⟩ (5)
  - eval ⟨) e, y + x S ⟩ = eval ⟨) e, ‖ x + y ‖ S ⟩ (6)
  - eval ⟨) e, x (S ⟩ = eval ⟨ e, x S ⟩ (7)
  - eval ⟨ =, x empty ⟩ = x (8)
  - eval ⟨ x e, S ⟩ = eval ⟨ e, ‖ x ‖ S ⟩ (9)

Stack – Anwendungsbeispiel 3/3

Bearbeiteter Ausdruck e	Stack	Regel
((3 + (4 * 5)) * (6 + 7)) =	]	(2)
(3 + (4 * 5)) * (6 + 7)) =	( ]	(2)
3 + (4 * 5)) * (6 + 7)) =	(( ]	(9)
+ (4 * 5)) * (6 + 7)) =	3 (( ]	(4)
(4 * 5)) * (6 + 7)) =	+ 3 (( ]	(2)
4 * 5)) * (6 + 7)) =	(+ 3 (( ]	(9)
* 5)) * (6 + 7)) =	4 (+ 3 (( ]	(3)
5)) * (6 + 7)) =	* 4 (+ 3 (( ]	(9)
)) * (6 + 7)) =	5 * 4 (+ 3 (( ]	(5)
)) * (6 + 7)) =	20 (+ 3 (( ]	(7)
) * (6 + 7)) =	20 + 3 (( ]	(6)
) * (6 + 7)) =	23 (( ]	(7)
* (6 + 7)) =	23 ( ]	(3)
(6 + 7)) =	* 23 ( ]	(2)
6 + 7)) =	(* 23 ( ]	(9)
+ 7)) =	6 (* 23 ( ]	(4)
7)) =	+ 6 (* 23 ( ]	(9)
)) =	7 + 6 (* 23 ( ]	(6)
)) =	13 (* 23 ( ]	(7)
) =	13 * 23 ( ]	(5)
) =	299 ( ]	(7)
=	299]	(8)

Stackimplementierung 1/4


```
class Stack {
  private int [] stack;
  private int num  = 0; // number of elements in stack

  public Stack (int size) { // stack with user defined size
     stack = new int [size];
  }

  public Stack () { // stack with standard size
     stack = new int [100];
  }
```

Stackimplementierung 2/4


```
public void push (int obj) {
   if (num == stack.length-1)
     println("stack overflow");
   else
     stack[num++] = obj;
 }

 public int pop () {
   if (isEmpty()) {
     println("stack underflow");
     return -1;
   }
   else
     return stack[--num];
 }
```

Stackimplementierung 3/4


```
public int top () {
   if (isEmpty()) {
     println("stack is empty");
     return -1;
   }
   else
     return stack[num-1];
 }

 public boolean isEmpty () {
   return num == 0;
 }
} // ~class Stack
```

Stackimplementierung 4/4


```
class ArrayStack {
  public static void main(String[] args) {

    Stack stack = new Stack (50);
    stack.push(1);
    stack.push(2);
    stack.push(3);

    while (!stack.isEmpty()) {
      int i = stack.pop();
      print(i);
    }
  }
}
```

Alternative Stackimplementierung 1/3


```
class Stack {
  private List stack;    // stack size is not limited !

    public Stack () {
    stack = new List();
  }

  public int pop () {
    if (isEmpty()) {
      println("stack underflow");
      return -1;
    }
    else
      return stack.removeFirst();
  }
```

Verwendung des ADT List

Verwendung der Methode removeFirst des ADT List

Alternative Stackimplementierung 2/3


```
  public void push (int obj) {
    stack.addFirst(obj);
  }

  public int top () {
    if (isEmpty()) {
      println("stack is empty");
      return -1;
    }
    else return stack.getFirst();
  }

  public boolean isEmpty () {
    return stack.isEmpty();  // calls isEmpty of class List
  }
}  // end of class ListStack
```

Verwendung der Methode addFirst des ADT List

Weiterreichen des Aufrufs an isEmpty des ADT List

Alternative Stackimplementierung 3/3


```
class ListStack {
  public static void main(String[] args) {

    Stack stack = new Stack (); // no size parameter necessary
    stack.push(1);
    stack.push(2);
    stack.push(3);

    while (!stack.isEmpty()) {
      int i = stack.pop();
      print(i);
    }
  }
}
```

Queue (Schlange)

- Einfache Datenstruktur mit beschränktem Zugriff auf gespeicherte Elemente.
- Arbeitet nach dem FIFO-Prinzip (First-In-First-Out): wie in einer Warteschlange, bei der man sich hinten anstellt (Wer zuerst kommt mahlt zuerst).
- Die Spezifikation von Queue und Stack sehen auf den ersten Blick sehr ähnlich aus. Das Verarbeitungsprinzip ist aber vollständig entgegengesetzt.

Queue – Operationen

- Queue als ADT:
  - **Operationen:**
    - enter - fügt ein neues Element in die Queue ein

      enter: Element × Queue → Queue
    - leave – entfernt das erste Element aus der Queue und gibt es zurück

      leave: Queue → Queue
    - front – nimmt das erste Element aus der Queue(ohne es zu entfernen)

      front: Queue → Element
    - empty – erzeugt eine leere Queue

      empty: → Queue
    - isEmpty - liefert true genau dann, wenn die Queue leer ist

      isEmpty: Queue → boolean

Queue – Axiome

- Axioms (Auswahl)
  - isempty (empty()) = true

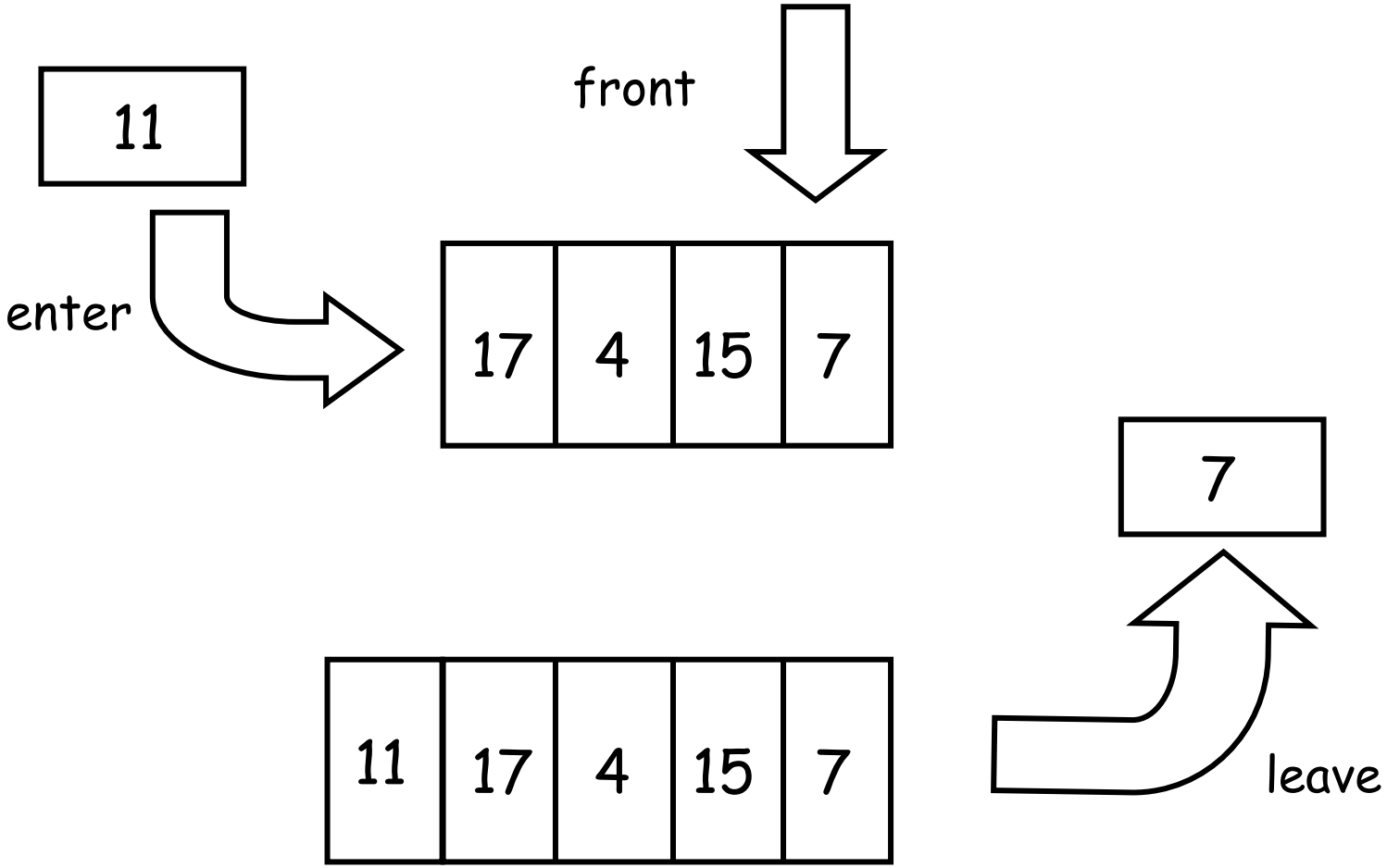
    Neu erzeugte Queue ist leer.
  - isempty (enter (Q, e)) = false

    Sobald ein Element eingefügt wurde,
    ist die Queue nicht mehr leer.
  - leave (enter (enter (Q, e_1), e_2)) = enter (leave (enter (Q, e_1)), e_2)

    Die Elemente werden in der Reihenfolge aus der Queue genommen, in welcher sie eingefügt wurden.
  - front (enter (enter (Q, e_1), $e_{2)}$) = front (enter (Q, e_1))

    Ein Elemente wird unabhängig davon, was nach ihm eingefügt wird, aus der Queue genommen.

Queue– Arbeitsweise



Queue– Anwendungsbeispiele

- Prozessverwaltung in Betriebssystemen
- Speicherverwaltung von Warteschlangen bei Druckaufträgen
- Nachrichtenpuffer in der Kommunikationssoftware

Queueimplementierung 1/5


```
class Queue {  // implemented by an array
  private int queue [];  // elements
  private int l = 0, u = 0;  //lower and upper pointer

  public Queue (int size) { // queue with user defines size
    queue = new int [size+1]; // one element for organizatorial reason
  }

  public Queue () { // queue with standard size queue contains max.
                    //  100 elements
     queue = new int [101];
  }
```

Queueimplementierung 2/5


```
public void enter (int obj) { // appends a new element at rear
                              // of the queue


   if ((queue.length - l + u) % queue.length == queue.length-1)
     println("queue overflow");
   else {
     queue[u] = obj;
     u = (u+1) % queue.length;
   }
 }
```

Queueimplementierung 3/5


```
public int leave () { // takes elements from front of the queue
  if (isEmpty()) {
    println("queue underflow");
    return -1;
  }
  else {
    int i = queue [l];
    queue[l] = -1;
    l = (l+1) % queue.length;
    return i;
  }
}
```

Queueimplementierung 4/5


```
public int front () {
   if (isEmpty()) {
     println("queue is empty");
     return -1;
   }
   else
     return queue[l];
 }

 public boolean isEmpty () {
   return l == u;
 }

} // end of class Queue
```

Queueimplementierung 5/5


```
class ArrayQueue {
  public static void main(String[] args) {

    Queue queue = new Queue (3);
    queue.enter(1);
    queue.enter(2);
    queue.enter(3);

    while (!queue.isempty()) {
      int i = queue.leave();
      print(i);
    }
  }
}
```

Alternative Queueimplementierung 1/2


```
class Queue {
  private List queue = null;  // elements

  public Queue () { // new queue
    queue = new List ();
  }

  public void enter (int obj) { // appends a new element at
                                // rear of the queue
    queue.addLast(obj);
  }
```

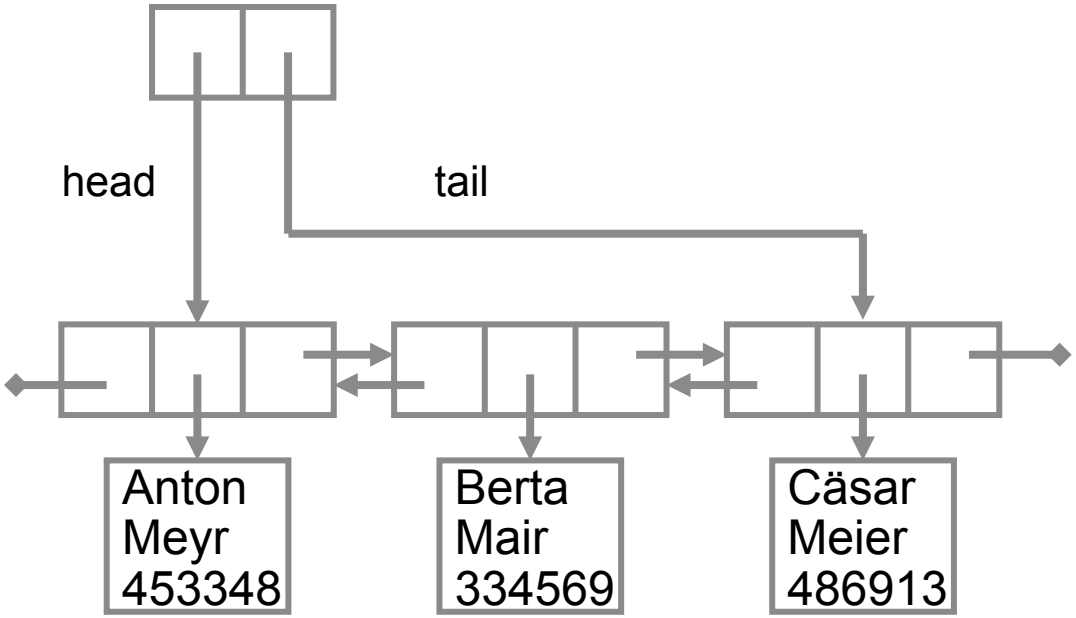
Alternative Queueimplementierung 2/2


```
public int leave () { // takes elements from front of the queue
    if (isEmpty()) {
      println("queue underflow");
      return -1;
    }
    else return queue.removeFirst();
  }

  public boolean isEmpty () {
    return queue.isEmpty();
  }
}  // ~Queue

//  main programm has not to be changed
```

Doppelt verkettete lineare Liste 1/3

- ADT-Sicht:
  - identisch mit allgemeiner Sicht
- Benutzersicht:
  - schneller in der Ausführung
- Entwicklersicht:
  - mehr Aufwand ☹

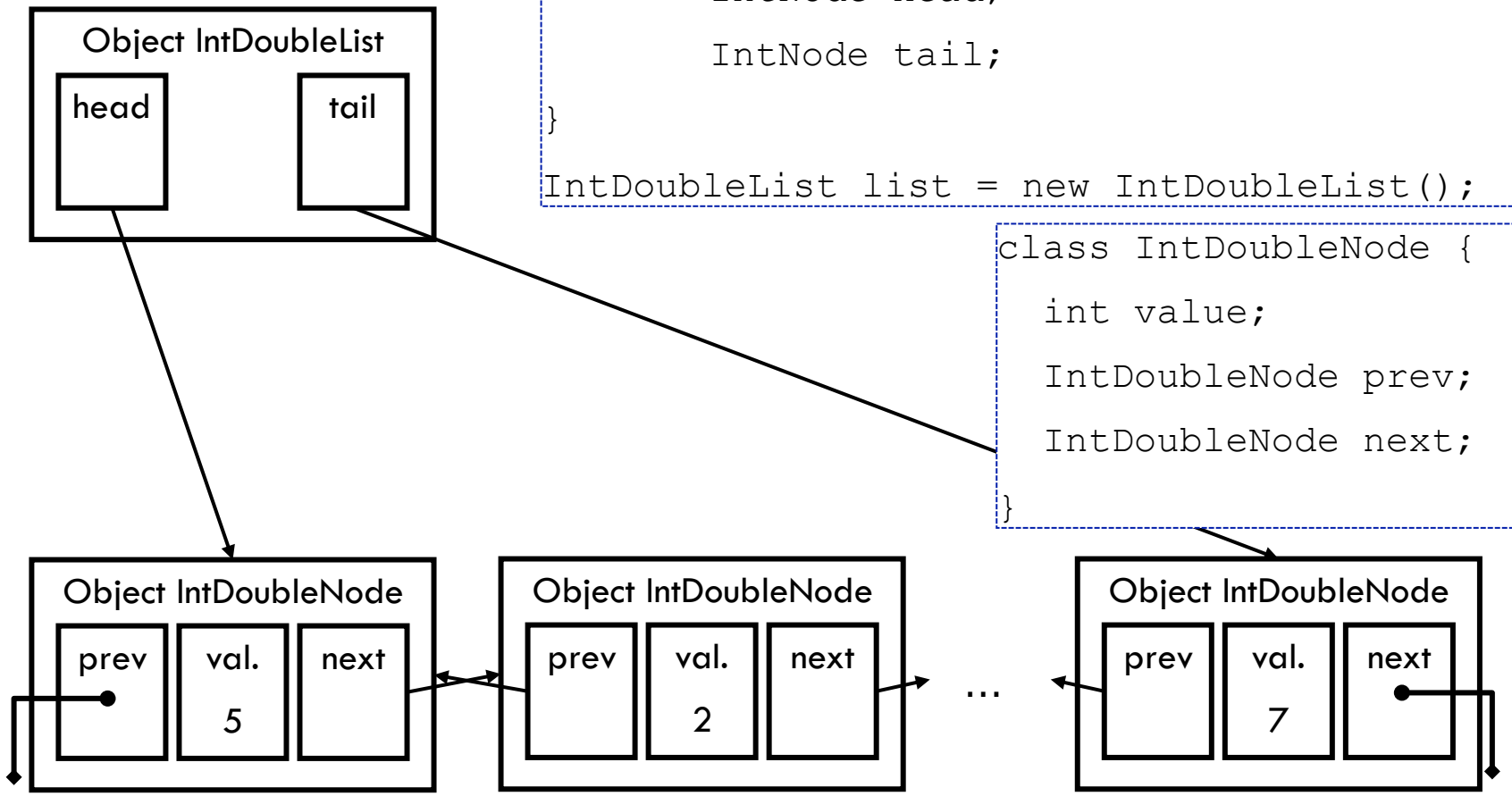
Doppelt verkettete lineare Liste 2/3



Doppelt verkettete lineare Liste 3/3


```
class IntDoubleList {
        IntNode head;
        IntNode tail;
}
IntDoubleList list = new IntDoubleList();
```

```
class IntDoubleNode {
  int value;
  IntDoubleNode prev;
  IntDoubleNode next;
}
```



Doppelt verkettete Liste - Implementierung


```
class DListElem {
  private DListElem prev;
  private int value;
  private DListElem next;
  // Konstruktor
  DListElem (
       DListElem prev,
       int val,
       DListElem next) {
    this.prev = prev;
    value = val;
    this.next = next;
  }
}
```

```
class DList {
  private DListElem head;
  private DListElem tail;
  // Konstruktor
  public DList () {
    head = null;
    tail = null;
  }
// weitere Funktionen
// ...
}
```

Weitere (spezielle) Listen-Operationen

- setAt — List × int × Element → List
  fügt Element an der i-ten Stelle ein
- getAt — List × int → Element
  liefert das Element an der i-ten Stelle
- removeAt — List × int → List
  entfernt das Element an der i-ten Stelle

Laufzeitverhalten für Listen

Operation	**einfach**	**doppelt verkettet**
`addFirst()`	$O(1)$	$O(1)$
`addLast()`	$O(n)$ / $O(1)$	$O(n)$ / $O(1)$
…	…	…
`contains()`	$O(n)$	$O(n)$
`removeLast()`	$O(n)$	$O(n)$ / $O(1)$
`removeFirst()`	$O(1)$	$O(1)$
`getAt()`	$O(n)$	$O(n)$

Zusammenfassung

- Listen sind der einfachste dynamische Datentyp.
- Es gibt unterschiedliche Listenarten und –varianten.
- Das Java Collection Framework stellt eine ganze Reihe von Listentypen zur Verfügung.
- Die vorgestellte Listenimplementierung in Java ist stark vereinfachend – es werden nur einfache Datentypen (int) in den Listenelementen verwaltet.
- Typischer Weise werden in Listen komplexe Objekte verwaltet.
- Der ADT Liste kann dazu verwendet werden andere ADTs, wie Stack oder Queue, zu realisieren.