252-0027
Einführung in die Programmierung

9. Rekursive Klassen: Verkettete Liste

Manuela Fischer, Malte Schwerhoff

Departement Informatik
ETH Zürich

Rekursion in …

- … **EBNF:** Name kommt auf LHS und RHS vor

```
<number>  ←  ( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 ) [ <number> ]
```

- … **Mathematik:** Definition bezieht sich auf sich selbst

$$n! = \begin{cases} 1, & \text{if } n \leq 1 \\ n \cdot (n-1)!, & \text{otherwise} \end{cases}$$

- … **Java-Methoden:** Methode ruft sich selbst auf

```java
public static void f() {
  f();
}
```

- … **Java-Klassen:** Klasse besitzt Attribut vom gleichen Typ

```java
class A {
  A a;
}
```

9. Rekursive Klassen: Verkettete Liste

9.1 Knoten einer verketteten Liste

9.2 Einschub: Innere Klassen

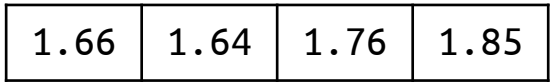
9.3 Listen-Operationen

9.4 Rekursive Methoden

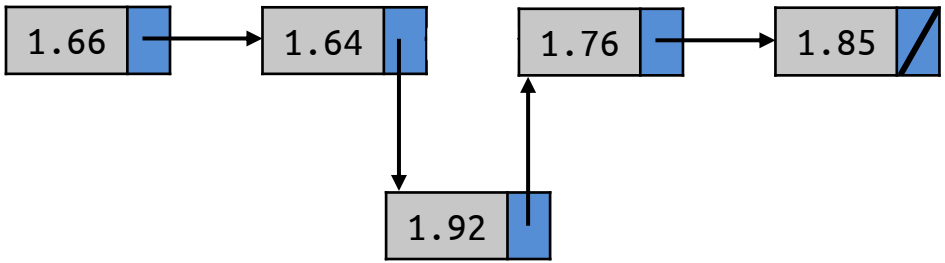
9.5 Ausblick: Java Lists

Beliebig grosse Datenstrukturen?

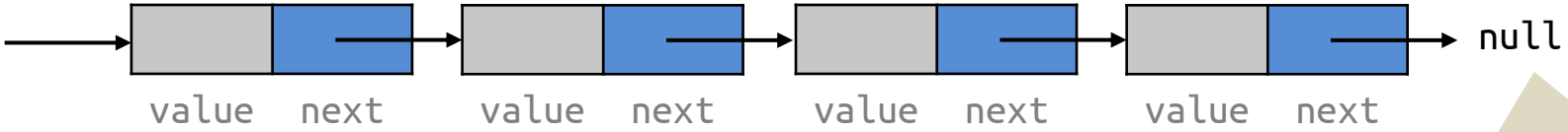
- Array
  - Fixe Grösse, die beim Erstellen bekannt sein muss
  - Einfügen eines Elements erfordert Kopie von allen Elementen



- **Verkettete Liste** («**linked list**»)
  - Einfügen eines Elements an beliebiger Stelle möglich



Verkettete Liste



- **Knoten** («**node**») speichert:
  - Element/Wert (hier vom Typ `int`) und
  - Verweis («link») auf nächsten Knoten

Rekursion stoppt bei `null`-Referenz.

```
class ListNode {
  int value;
  ListNode next;
}
```

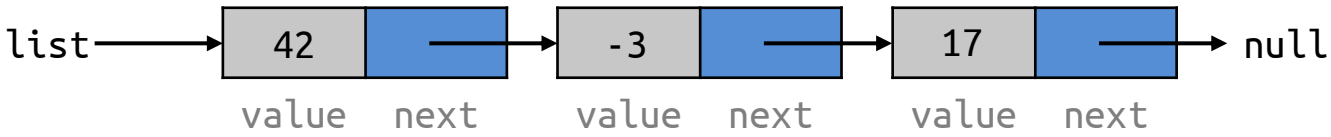
Rekursive Klasse: besitzt eine Referenz auf sich selbst!

- Liste ist gegeben als Verweis auf ersten Knoten

Beispiel

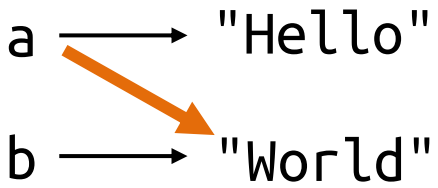
```
class ListNode {
  int value;
  ListNode next;
}
```

```
ListNode list = new ListNode();
list.value = 42;
list.next = new ListNode();
list.next.value = -3;
list.next.next = new ListNode();
list.next.next.value = 17;
```



Erinnerung: Zuweisung von Referenzen

```
String a = "Hello";
String b = "World";
```



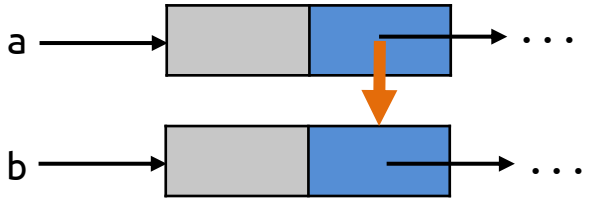

```
a = b;
```

Linke Seite der Zuweisung:
Umhängen des Zeigers a.

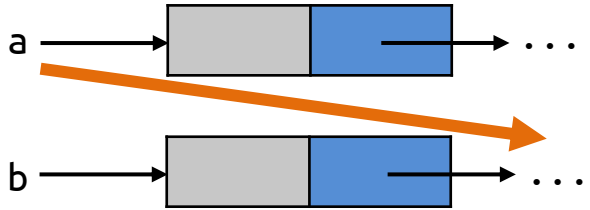
Rechte Seite der Zuweisung:
Umhängen auf das Objekt,
auf das b zeigt.

Erinnerung: Zuweisung von Referenzen

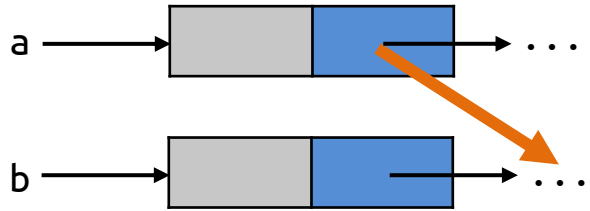
- **`a.next = b;`**
  - next-Verweis von a zeigt nun auf b



- **`a = b.next;`**
  - a zeigt nun auf das gleiche Objekt wie der next-Verweis von b



- **`a.next = b.next;`**
  - next-Verweis von a zeigt nun auf das gleiche Objekt wie der next-Verweis von b



Konstruktoren für `ListNode`

```
class ListNode {
  int value;
  ListNode next;

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

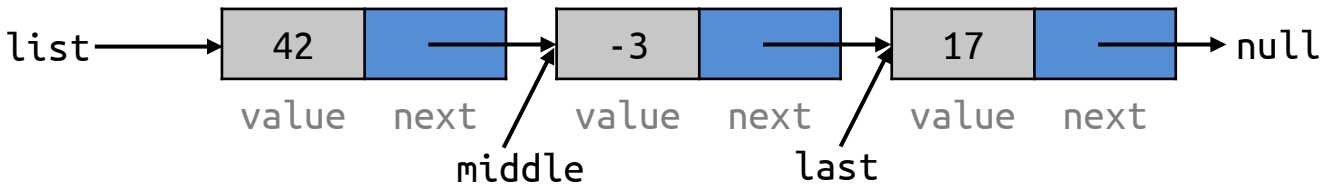
Default-Konstruktor eher nicht erwünscht.

Beispiel nochmals!

```
ListNode last = new ListNode(17);
ListNode middle = new ListNode(-3, last);
ListNode list = new ListNode(42, middle);
```

```
ListNode list = new ListNode(42,
                  new ListNode(-3,
                    new ListNode(17)));
```

```
class ListNode {
  int value;
  ListNode next;

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

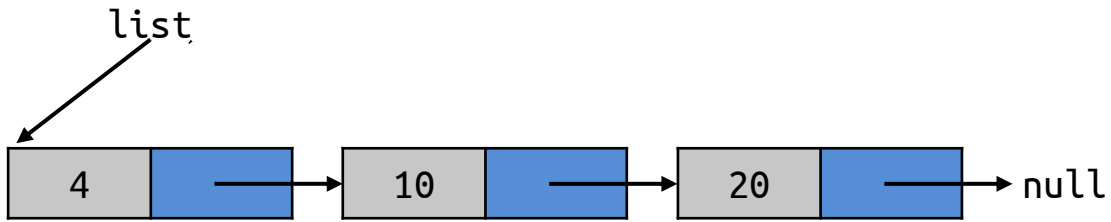


Beispiel 1: Element 30 am Ende einfügen

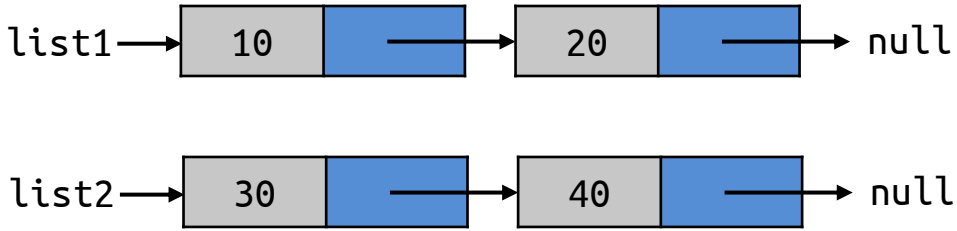

```
list.next.next = new ListNode(30);
```

Beispiel 2: Element 4 am Anfang einfügen

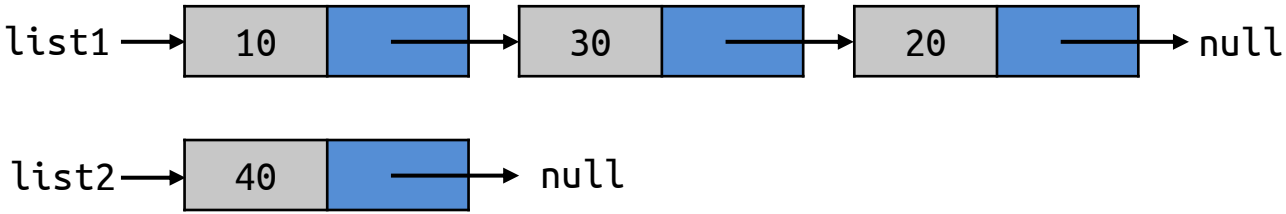
```
list = new ListNode(4, list);
```



Beispiel 3: Listen verändern

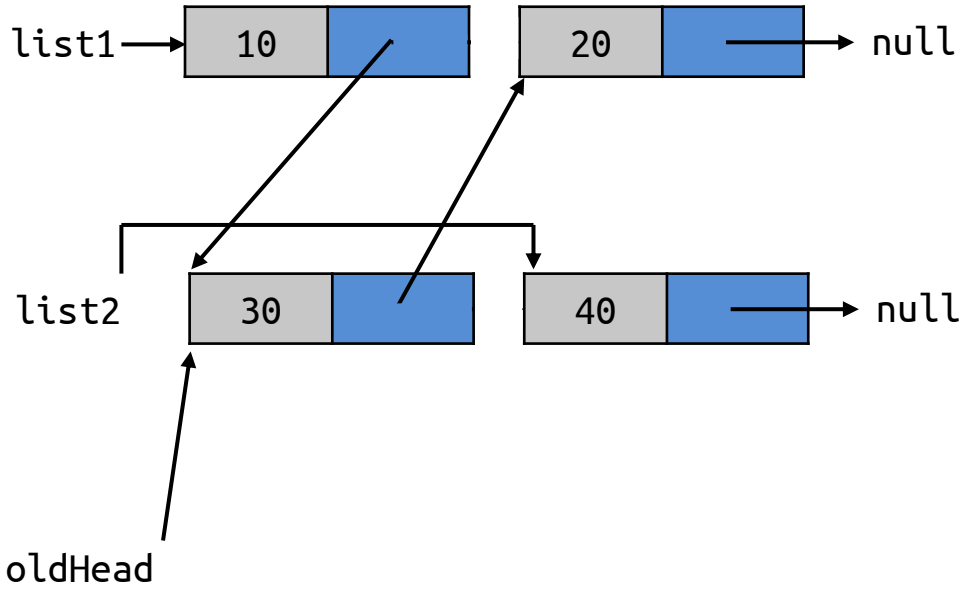


Lösung auf nächster Folie

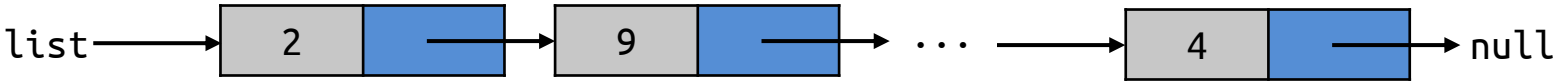


Beispiel 3 fortgesetzt

```
ListNode oldHead = list2;
list2 = list2.next;
oldHead.next = list1.next;
list1.next = oldHead;
```



Alle Elemente ausgeben?

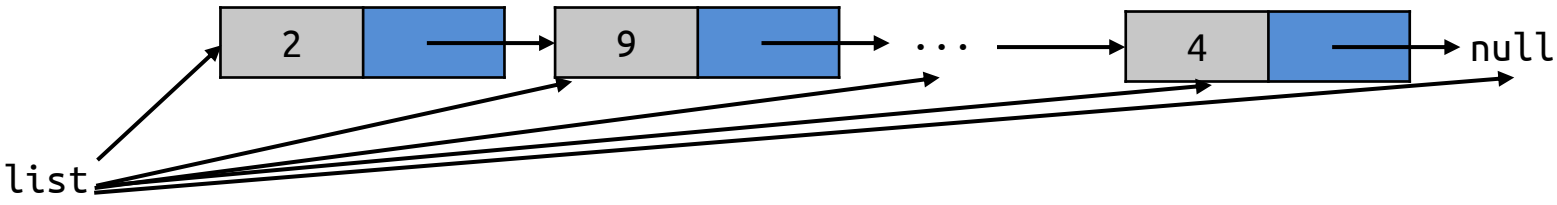


Pseudo-Code:

- Solange es noch Knoten gibt:
  - Gebe Wert des Knotens aus
  - Gehe zum nächsten Knoten

```
while (list != null) {
  System.out.print(list.value + " ");
  list = list.next;
}
```

Alle Elemente ausgeben: Animation

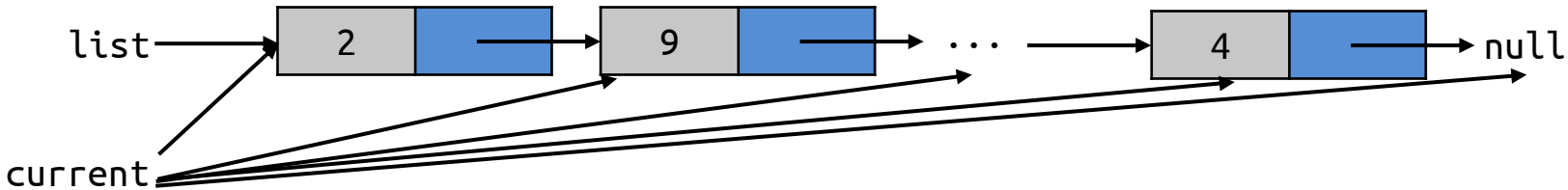


Output: `2 9 ... 4`

Korrekt, aber Zugang zu Liste verloren!

```
while (list != null) {
  System.out.print(list.value + " ");
  list = list.next;
}
```

Alle Elemente ausgeben: Lösung



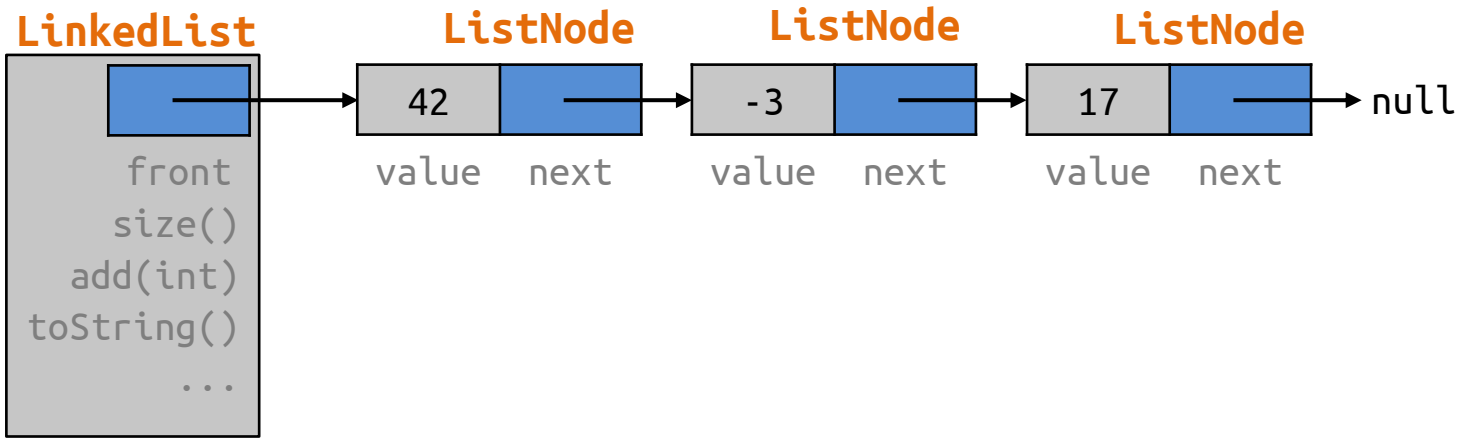
Separate Laufvariable `current` zur Iteration

```
ListNode current = list;
while (current != null) {
  System.out.print(current.value + " ");
  current = current.next;
}
```

Invariante: Alle Werte bis und ohne `current` wurden ausgegeben.

Klasse `LinkedList`

- Knoten sollten nicht von aussen direkt zugreifbar/änderbar sein
  - Klasse `LinkedList` versteckt Knoten
  - Klienten arbeiten nur mit `LinkedList` und angebotenen Methoden, nicht mit `ListNode`
  - Klasse `LinkedList` enthält Referenz `front` auf den ersten Knoten



Klasse LinkedList: 1. Variante

```
class ListNode {
  int value;
  ListNode next;
}
```

```
public class LinkedList {
  private ListNode front;

  public LinkedList() {
    front = null;
  }
  // public methods
}
```

ListNode hat eigentlich ausserhalb von LinkedList keine "Daseinsberechtigung": LinkedList ist der einzige Klient! Alternative: Innere Klasse.

9. Rekursive Klassen: Verkettete Liste

9.1 Knoten einer verketteten Liste

9.2 Einschub: Innere Klassen

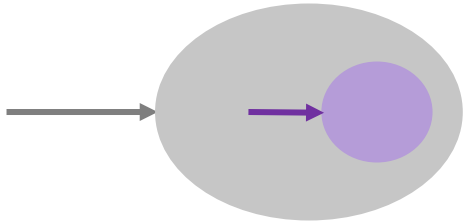
9.3 Listen-Operationen

9.4 Rekursive Methoden

9.5 Ausblick: Java Lists

Verschachtelte Klassen («nested classes»)

```
public class OuterClass {
   ...
   private class InnerClass {

   }
}
```



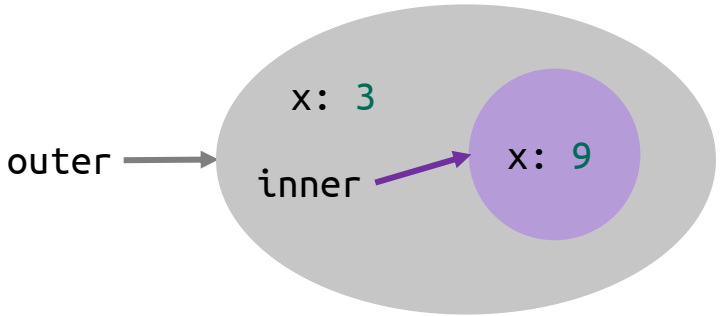
innere Klasse («**inner class**») innerhalb einer anderen Klasse,
der **äusseren Klasse** («**outer class**» / «**enclosing class**»), definiert

- Innere Klasse (nur) innerhalb der äusseren Klasse sichtbar
- Instanzen der inneren Klasse existieren nur innerhalb eines Objekts der äusseren Klasse
- Äussere Klasse in der inneren Klasse sichtbar: Zugriff via **`OuterClass.this.name`**

Innere Klasse kann auch andere Modifier haben, für uns immer non-static und `private` mit `private` Attributen/Methoden.

Beispiel

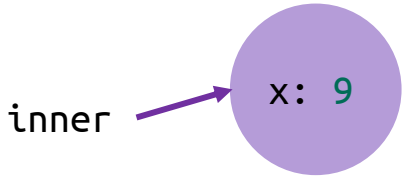
```
class OuterClass {
  private int x = 3;
  String foo() {
    InnerClass inner = new InnerClass(9);
    return inner.bar();
  }
  private class InnerClass {
    private int x;
    private InnerClass(int a) { x = a; }
    private String bar() {
      return x + " " + OuterClass.this.x;
    }
  }
}
```




```
OuterClass outer = new OuterClass();
System.out.println(outer.foo()); // outputs 9 3
```

Achtung: Nicht in statischer Methode!

```
public class OuterClass {
  public static void main(String[] args) {
    InnerClass inner = new InnerClass(9);
  }
  private class InnerClass {
    ...
  }
}
```

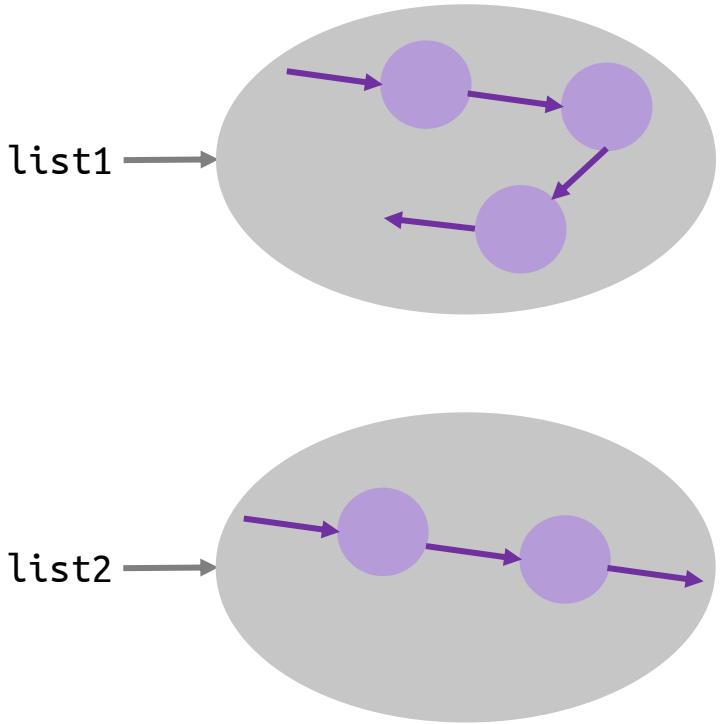


COMPILER ERROR: No enclosing instance of type OuterClass

- Innere Klassen dürfen nur innerhalb eines äusseren Objekts instanziiert werden!

Klasse LinkedList: 2. Variante

```
public class LinkedList {

   private class ListNode {
     private int value;
     private ListNode next;
     ...
   }

   private ListNode front;

   public LinkedList() {
     front = null;
   }
   // methods
}
```



9. Rekursive Klassen: Verkettete Liste

9.1 Knoten einer verketteten Liste

9.2 Einschub: Innere Klassen

9.3 Listen-Operationen

9.4 Rekursive Methoden

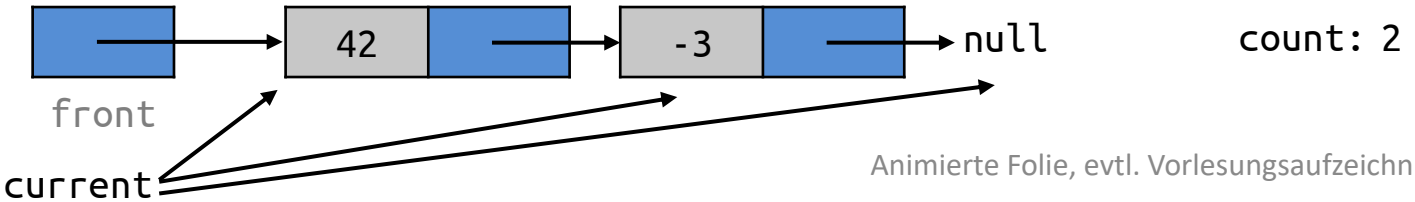
9.5 Ausblick: Java Lists

LinkedList.size()

- Gibt Anzahl Elemente der Liste zurück

```
public int size() {
  int count = 0;
  ListNode current = front;
  while (current != null) {
    count += 1;
    current = current.next;
  }
  return count;
}
```

Invariante: count ist Anzahl Knoten bis und ohne current.



Alternative: `size`-Attribut

```
public int size() {
  int count = 0;
  ListNode current = front;
  while (current != null) {
    count += 1;
    current = current.next;
  }
  return count;
}
```

Methode: neu berechnen bei jeder Anfrage!

```
private int size;
public int size() { return size; }
```

Attribut: Mitführen eines Zählers, Ändern bei add/remove-Operationen

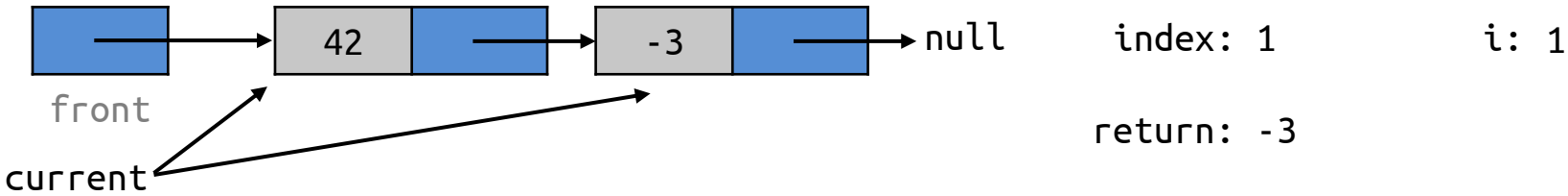
LinkedList.get(int index)

Wir müssen über die ersten index Elemente iterieren.

- Gibt Wert des index-ten Knoten (0-basiert) der Liste zurück

Invariante: current ist Element mit Index i.

```
public int get(int index) {
  // assume 0 <= index && index < size()
  ListNode current = front;
  for (int i = 0; i < index; i++) {
    current = current.next;
  }
  return current.value;
}
```

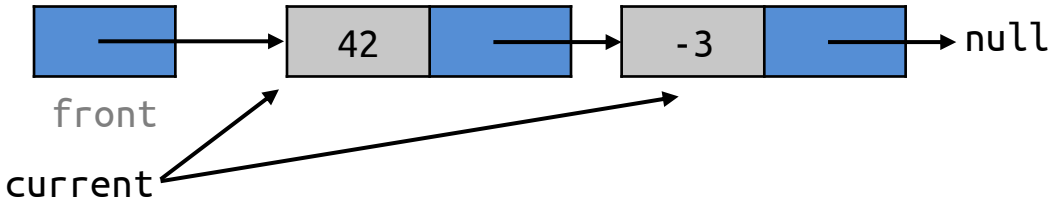


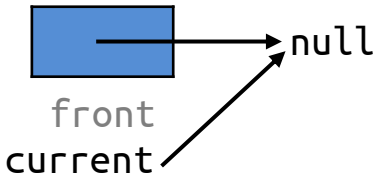
LinkedList.back(): Fehlerhafte Version

- Gibt (Verweis auf) letzten Knoten der Liste zurück

Achtung Edge Case leere Liste: NullPointerException

```
private ListNode back() {
  ListNode current = front;
  while (current.next != null) { // while node after current one != null
    current = current.next;
  }
  return current;
}
```





LinkedList.back(): Lösung

- Gibt (Verweis auf) letzten Knoten der Liste zurück

```
private ListNode back() {
  ListNode current = front;
  while (current != null && current.next != null) {
    current = current.next;
  }
  return current;
}
```

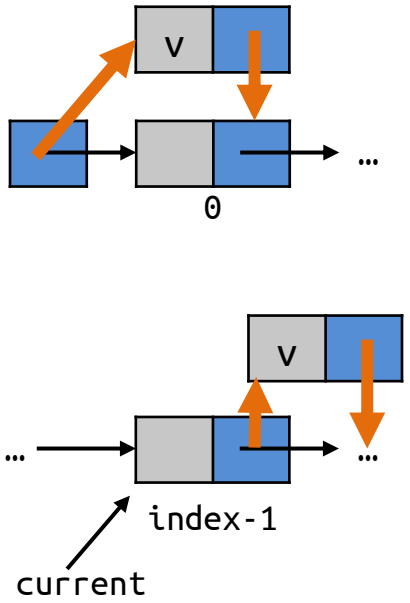
Kurzschlussauswertung!

Alternative: Zeiger auf das letzte Element in Attribut abspeichern und mitführen...

LinkedList.add(int value, int index)

- Fügt Knoten mit Wert `value` als Element mit Index `index` der Liste hinzu

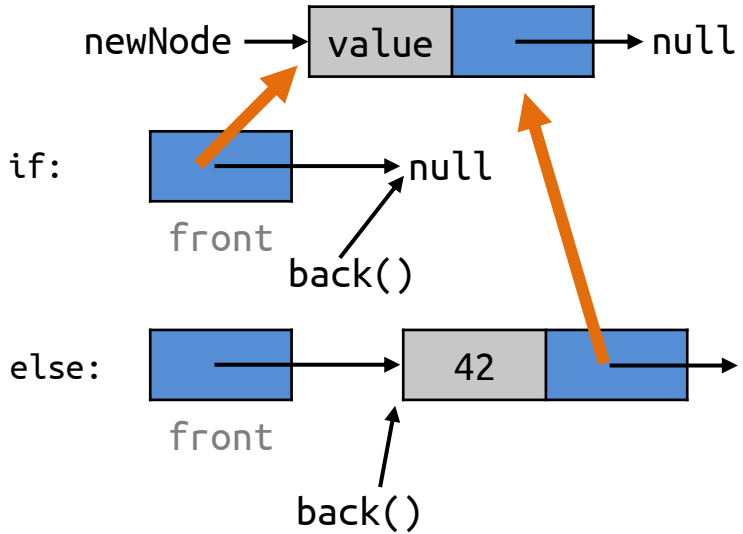
```
public void add(int value, int index) {
  // assume 0 <= index && index < size()
  if (index == 0) {
    front = new ListNode(value, front);
  } else {
    ListNode current = front;
    for (int i = 0; i < index - 1; i++) {
      current = current.next;
    }
    current.next = new ListNode(value, current.next);
  }
}
```



LinkedList.addBack(int value)

- Fügt Knoten mit Wert `value` als letztes Element der Liste hinzu

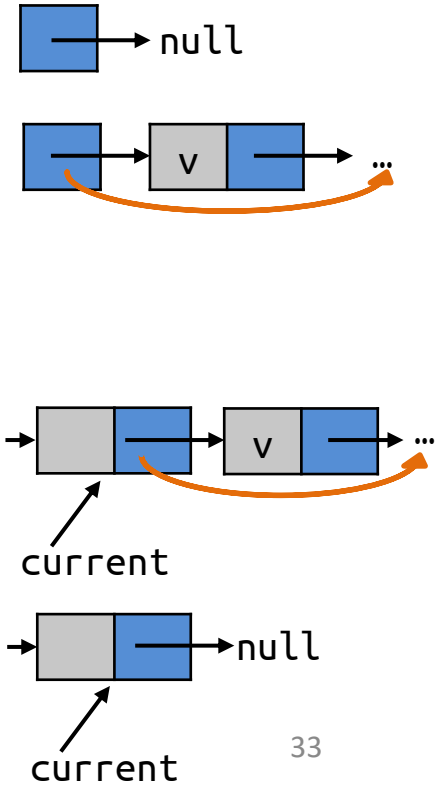
```
public void addBack(int value) {
  ListNode newNode = new ListNode(value);
  if (back() == null) { // if list is empty
    front = newNode;
  } else {
    back().next = newNode;
  }
}
```



LinkedList.remove(int value)

- Entfernt ersten Knoten mit Wert value von der Liste

```
public void remove(int value) {
  if (front == null) return; // nothing to do if list is empty
  if (front.value == value) { // first element has value
    front = front.next; // remove first element
    return;
  }
  ListNode current = front;
  while (current.next != null && current.next.value != value) {
    current = current.next;
  }
  if (current.next != null) { // if value found
    current.next = current.next.next;
  } // if value not found: do nothing
}
```

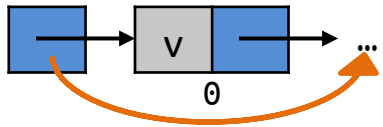


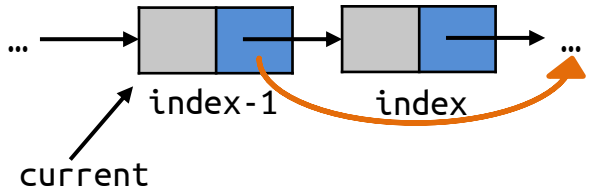
Animierte Folie, evtl. Vorlesungsaufzeichnung anschauen

LinkedList.removeAt(int index)

- Entfernt Knoten an Index index von der Liste

```
public void removeAt(int index) {
  // assume 0 <= index && index < size()
  if (index == 0) { // remove first element
    front = front.next;
  } else {
  ListNode current = front;
  for (int i = 0; i < index - 1; i++) {
    current = current.next;
  }
  current.next = current.next.next;
}
```

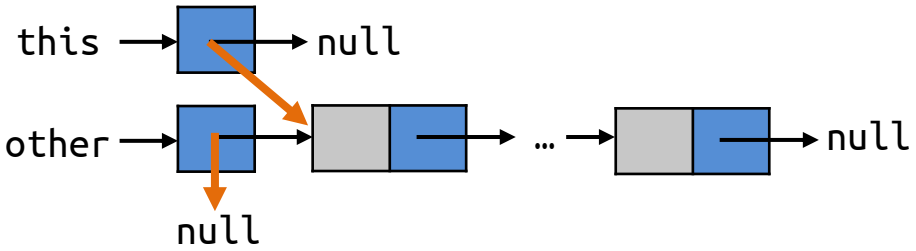


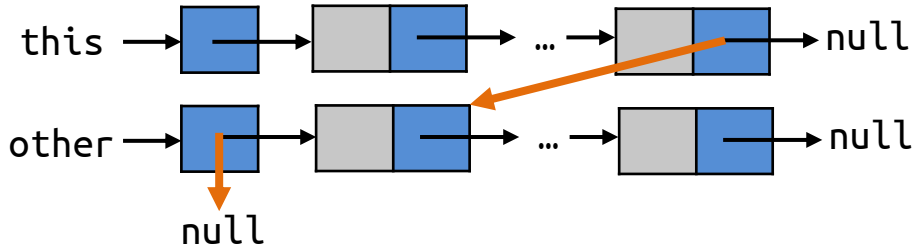


LinkedList.merge(LinkedList other)

- Hängt die Liste other an diese Liste an

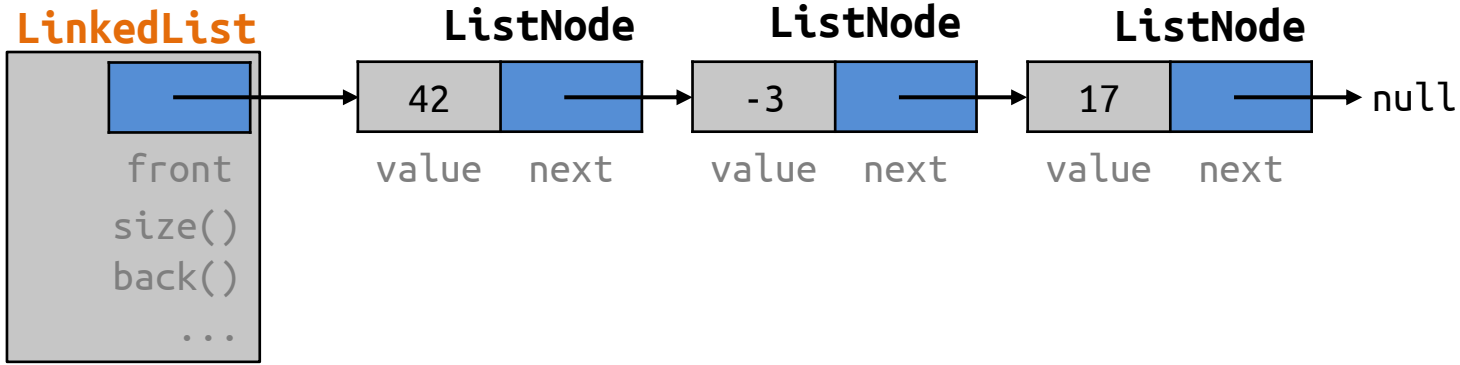
```
public void merge(LinkedList other) {
  if (front == null) {
    front = other.front; // make this list to other list
  } else {
    back().next = other.front; // append other list to back()
  }
  other.front = null; // make other list empty
}
```





Animierte Folie, evtl. Vorlesungsaufzeichnung anschauen

Objektinvarianten von `LinkedList`



- `size()` gleich Anzahl der erreichbarer Knoten
- `back()` erreichbar
- `back() == null` ⇒ `front == null`
- `size() == 0` ⇒ `(front == null && back() == null)`
- Keine Zyklen: jeder erreichbare Knoten ist nur genau einmal erreichbar.
- …

Typische Anwendung: Sortierte Liste

- Elemente der Liste aufsteigend sortiert
  - Macht Suche einfacher/effizienter…
  - …aber Einfügen schwieriger!

Zusätzliche Objektinvariante!

Klasse SortedLinkedList

```
public class SortedLinkedList {
  // Invariant: elements sorted in increasing order
  ListNode front;

  public SortedLinkedList() {
    front = null;
  }

  // methods

}
```

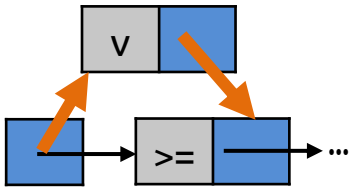
SortedLinkedList.add(int value)

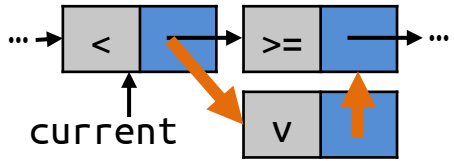
- fügt Knoten mit Wert `value` der Liste hinzu, so dass Liste sortiert bleibt

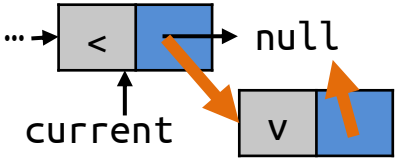
```
public void add(int value) {
  if (front == null || value <= front.value) {
    front = new ListNode(value, front);
  } else {
    ListNode current = front;
    while (current.next != null && current.next.value < value) {
      current = current.next;
    } // either current.next == null or current.next.value >= value
    current.next = new ListNode(value, current.next);
  }
}
```

Invariante: Alle Werte bis und mit `current` sind kleiner als `value`

`current.next` erster Wert `>= value`







Animierte Folie, evtl. Vorlesungsaufzeichnung anschauen

9. Rekursive Klassen: Verkettete Liste

9.1 Knoten einer verketteten Liste

9.2 Einschub: Innere Klassen

9.3 Listen-Operationen

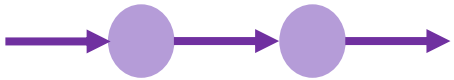
9.4 Rekursive Methoden

9.5 Ausblick: Java Lists

Übersicht Rekursive Strukturen: Liste und Baum

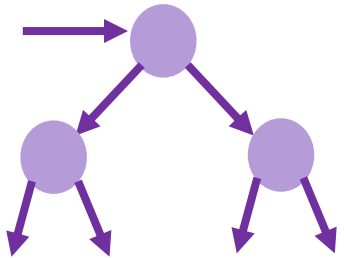
```
class ListNode {
  int value;
  ListNode next;
}

class LinkedList {
  ListNode front;
}
```

Ein rekursiver Branch!



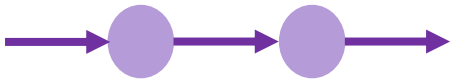
```
class BinaryTreeNode {
  int value;
  BinaryTreeNode left;
  BinaryTreeNode right;
}

class BinaryTree {
  BinaryTreeNode root;
}
```

Zwei rekursive Branches!



Hier alle Klassen/Attribute sichtbar, um Code minimal zu halten!

Rekursive Strukturen und Rekursion

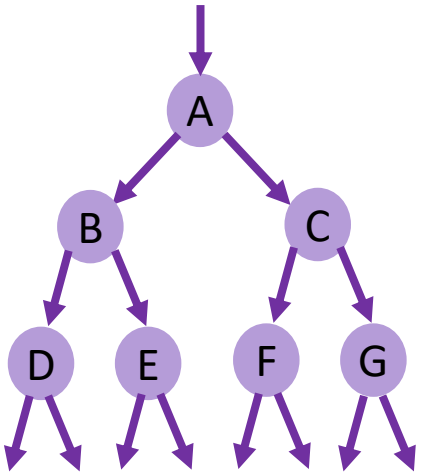


Rekursive Strukturen lassen sich relativ einfach mit Rekursion traversieren!

```
class LinkedList {
  ...
  void visit() {
    if (front != null) front.visit();
  }
}
```

```
class ListNode {
  ...
  void visit() {
    System.out.println(value);
    if (next != null) next.visit();
  }
}
```

Rekursive Strukturen und Rekursion



A, B, D, E, C, F, G

```
class BinaryTreeNode {
  ...
  void visit() {
    System.out.println(value);
    if (left != null) left.visit();
    if (right != null) right.visit();
  }
}
```

Insbesondere fürs Traversieren von rekursiven Strukturen mit mehreren rekursiven Branches bietet sich Rekursion an!

Rekursion: Grösse der `LinkedList`

```
class LinkedList {
  ...
  int size() {
    if (front != null) {
      return front.size();
    } else {
      return 0;
    }
  }
}
```

```
class ListNode {
  ...
  int size() {
    if (next != null) {
      return 1 + next.size();
    } else {
      return 1;
    }
  }
}
```

Rekursion: Einfügen in LinkedList

```
class LinkedList {
  ...
  void addBack(int v) {
    if (front != null) {
      front.addBack(v);
    } else {
      front = new ListNode(v);
    }
  }
}
```

```
class ListNode {
  ...
  void addBack(int v) {
    if (next != null) {
      next.addBack(v);
    } else {
      next = new ListNode(v);
    }
  }
}
```

9. Rekursive Klassen: Verkettete Liste

9.1 Knoten einer verketteten Liste

9.2 Einschub: Innere Klassen

9.3 Listen-Operationen

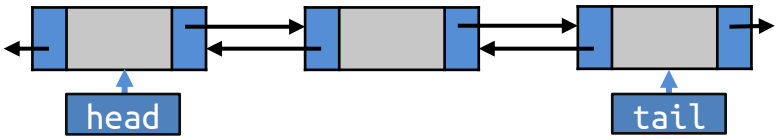
9.4 Rekursive Methoden

9.5 Ausblick: Java Lists

Ausblick: `java.util.LinkedList`

Verkettete Liste bereits implementiert für Sie in `java.util.LinkedList`

- Importieren mit `import java.util.LinkedList;`



Doppelt verkettete Liste

- Knoten mit Zeiger auf Vorgänger und Nachfolger
- Liste mit Zeiger auf Anfang (`head`) und Ende (`tail`) der Liste
- Details folgen später

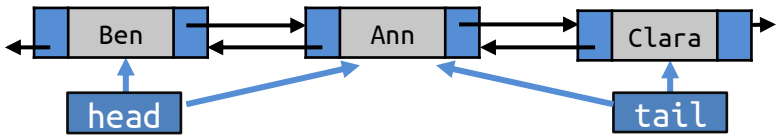
Beispiel: java.util.LinkedList

Bereits implementiert für Sie in `java.util.LinkedList`

- Importieren mit `import java.util.LinkedList;`

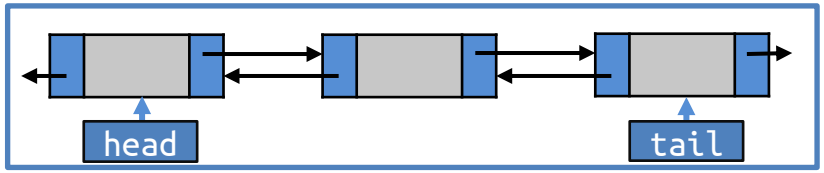
Typ der Elemente, mehr dazu später!

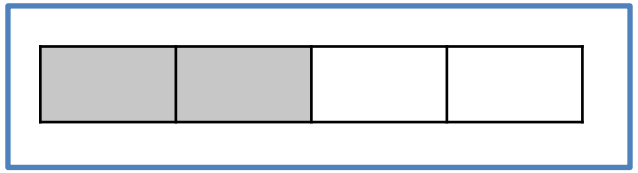
```
LinkedList<String> names = new LinkedList<String>();
names.addFirst("Ann");                          // "Ann"
names.addFirst("Ben");                          // "Ben", "Ann"
names.addLast("Clara");                         // "Ben", "Ann", "Clara"
System.out.println(names.getFirst());           // outputs "Ben"
System.out.println(names.removeFirst());        // "Ann", "Clara"; outputs "Ben"
System.out.println(names.getFirst());           // outputs "Ann"
System.out.println(names.removeLast());         // "Ann"; outputs "Clara"
```



Animierte Folie, evtl. Vorlesungsaufzeichnung anschauen

Vergleich von Java Lists





`LinkedList`

- Doppelt verkettete Liste
- Index-basierter Zugriff ineffizient
- Effizientes **Einfügen/Löschen am Anfang und Ende** der Liste
- Geeignet für Implementation von Stack + Queue

`ArrayList`

- Dynamisches (wachsendes) Array: Grösse kann sich zur Laufzeit ändern
- Einfügen/Löschen am Anfang ineffizient; am Ende schnell in Praxis
- **Random Access**: effizienter Index-basierter Zugriff

Theorie dazu in A&D. Mehr Details zu Implementation und Verwendung später im Kapitel zu *Generics*.

Beispiel: `java.util.ArrayList`

Wachsendes Array bereits implementiert für Sie in `java.util.ArrayList`

- Importieren mit `import java.util.ArrayList;`

```
ArrayList<String> names = new ArrayList<String>();
names.add("Ann");                          // "Ann"
names.add("Ben");                          // "Ann", "Ben"
names.add("Clara");                        // "Ann", "Ben", "Clara"
System.out.println(names.get(1));          // outputs "Ben"
System.out.println(names.indexOf("Ann"));  // outputs 0
System.out.println(names.indexOf("Dan"));  // outputs -1
names.removeLast();                        // "Ann", "Ben"
System.out.println(names.toString());      // outputs [Ann, Ben]
```

Animierte Folie, evtl. Vorlesungsaufzeichnung anschauen

Ausblick: Java Collections

Mehr Informationen folgen!

`ArrayList`, `LinkedList` und noch viele andere Datenstrukturen mit zahlreicher Funktionalität (`import java.util.Collections;`)

- Sortieren: `Collections.sort`
- Binäre Suche: `Collections.binarySearch`
- Minimum & Maximum: `Collections.min` & `Collections.max`
- Elemente ersetzen: `Collections.replaceAll`
- Elemente swappen: `Collections.swap`
- Auf gemeinsame Elemente überprüfen: `Collections.disjoint`
- Umkehren: `Collections.reverse` & `Collections.reverseOrder`
- `...`

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Collections.html

`LinkedList` mit Zahlen

```
LinkedList<int> heights = new LinkedList<int>();
```

```
Syntax error, insert "Dimensions" to complete ReferenceType
```

Problem in Java:

- Java Collections funktionieren nur mit Referenztypen
- Primitive Typen (wie `int`) sind keine Referenztypen…

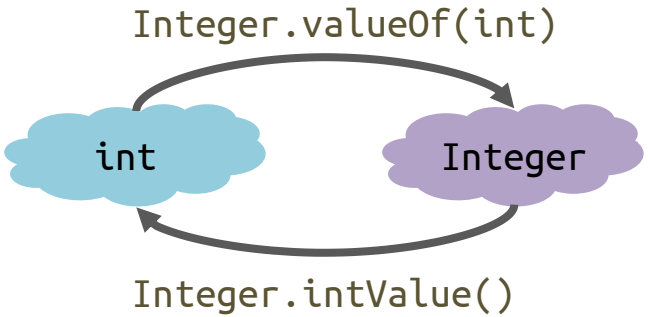
Lösung in Java: Wrapper-Klassen («wrapper classes»)

- Pro primitivem Typ eine Klasse (Referenztyp) in `java.lang`, die Wert dieses Typs intern speichert
  - `Integer`, `Boolean`, `double`, usw.

```
class Integer {
  int value;

  …
}
```

Boxing and Unboxing

- Mindestfunktionalität: Wechsel `int` ⟷ `Integer` (analog für `boolean`, …)

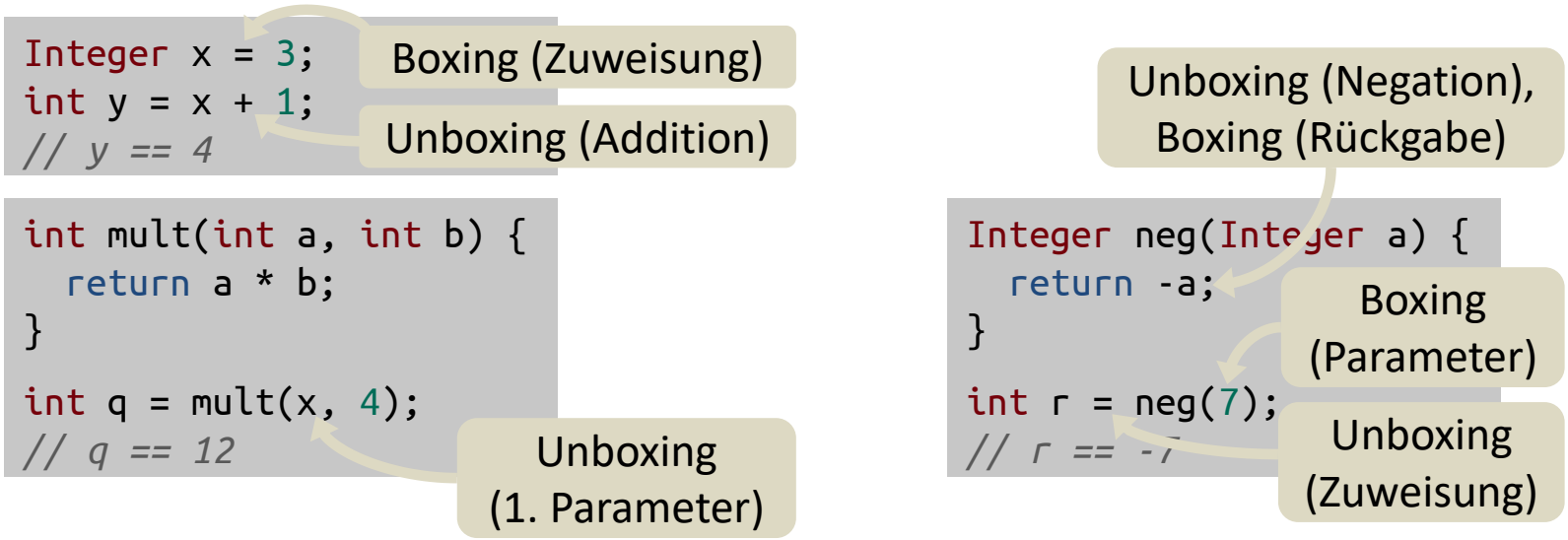



```
Integer x = Integer.valueOf(3);
int y = x.intValue(); // y == 3
```

- **Boxing**: Wechsel primitiver Wert zu Wrapper-Objekt (`valueOf(…)`)
- **Unboxing**: Wechsel Wrapper-Objekt zu primitivem Wert (`intValue()`)

Autoboxing

Auto(un)boxing = Automatisches (Un-)Boxen, falls `int` vorhanden, aber `Integer` verlangt, bzw. andersherum. Beispiele:



Vorsicht: Autoboxing und Type Conversions

```
int i = 1;
double d = 3;  // OK
double e = i;  // OK
```

	int	long	double
int			
long			
double			

Implizite Konvertierung

Implizite Konvertierung, pot. mit Genauigkeitsverlust

Explizite Konvertierung

Keine Typkonversionen in Kombination mit Boxing. Nur passender Wrapper-Typ erlaubt!

```
Integer i = 1;
Double d = 3;  // COMPILER ERROR
Double e = i;  // COMPILER ERROR
```

Keine implizite Konvertierung von `Integer` nach `Double`.

Gleichheit von Wrapper-Objekten

- Bekannt
  - Primitive Werte können mit `p1 == p2` verglichen werden
  - Objekte sollten mit `o1.equals(o2)` verglichen werden
    (denn `o1 == o2` vergleicht Speichadressen, nicht den Objektinhalt)
- Wrapper-Objekte sind daher auch mit `equals()` zu vergleichen

```
jshell> Integer.valueOf(200) == Integer.valueOf(200)
$18 ==> false

jshell> Integer.valueOf(200).equals(Integer.valueOf(200))
$19 ==> true
```

Java Lists und Autoboxing

```
ArrayList<Integer> data  = new ArrayList<Integer>();

data.add(10);
data.add(11);

int first = data.get(0);
Integer second = data.get(1);
System.out.println(first + second);
```

Boxing (Parameter)

Unboxing (Zuweisung)

Unboxing (Addition)

```
21
```

- Elementtyp (hier `Integer`) muss Referenztyp sein
- Mehr zu Java Lists und Collections später, in Kapitel zu *Collections & Generics*