



Effiziente verkettete Listen

Annabelle Klarl

Zentralübung zur Vorlesung

„Einführung in die Informatik: Programmierung und Softwareentwicklung“

<http://www.pst.ifi.lmu.de/Lehre/wise-12-13/infoeinf>



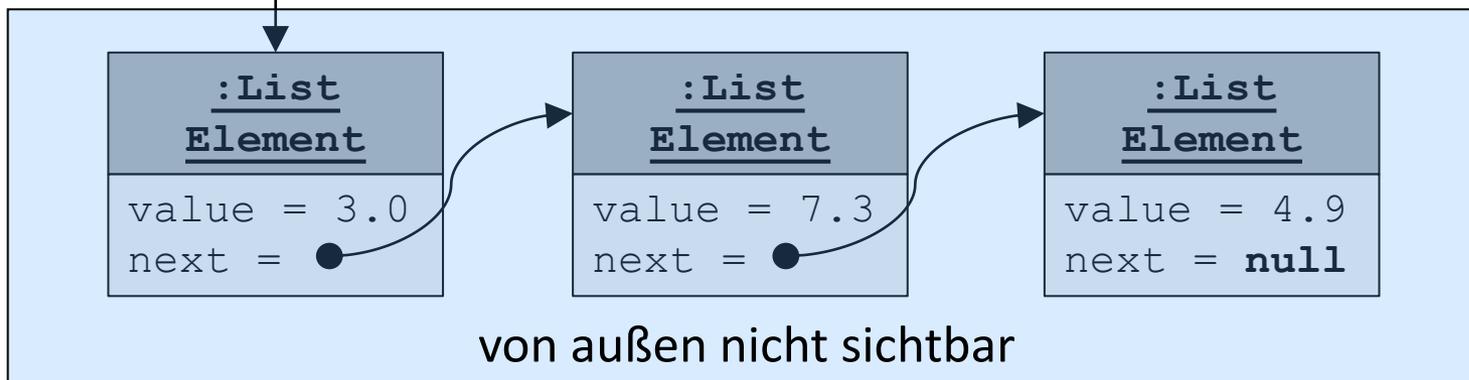
Verkettete Listen: Wiederholung

Eine verkettete Liste speichert die Listenelemente als Kette, wobei jedes Listenelement seinen Nachfolger kennt.

z.B. Repräsentation von `list=<3.0, 7.3, 4.9>`



Zugriff auf Listenelemente nur über `list`
z.B. Wert hinzufügen: `list.addFirst(1.1)`
z.B. Wert auslesen: `list.get(2)`

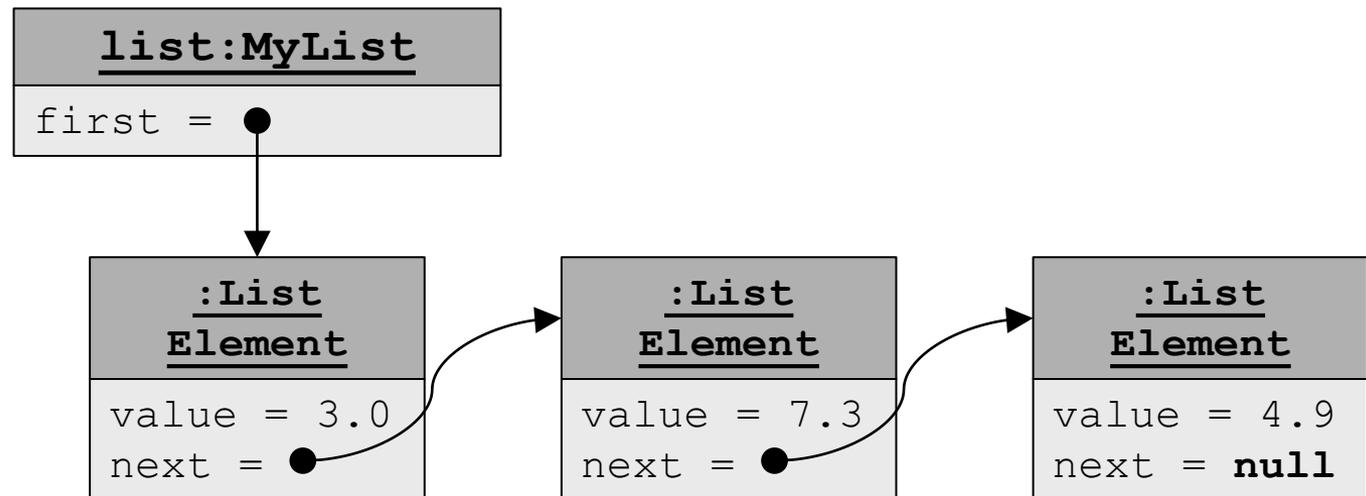




Verkettete Listen: `addFirst` Wiederholung (I)

```
list.addFirst(1.1)
```

=> Hinzufügen von 1.1 am **Anfang** der Liste `list=<3.0, 7.3, 4.9>`





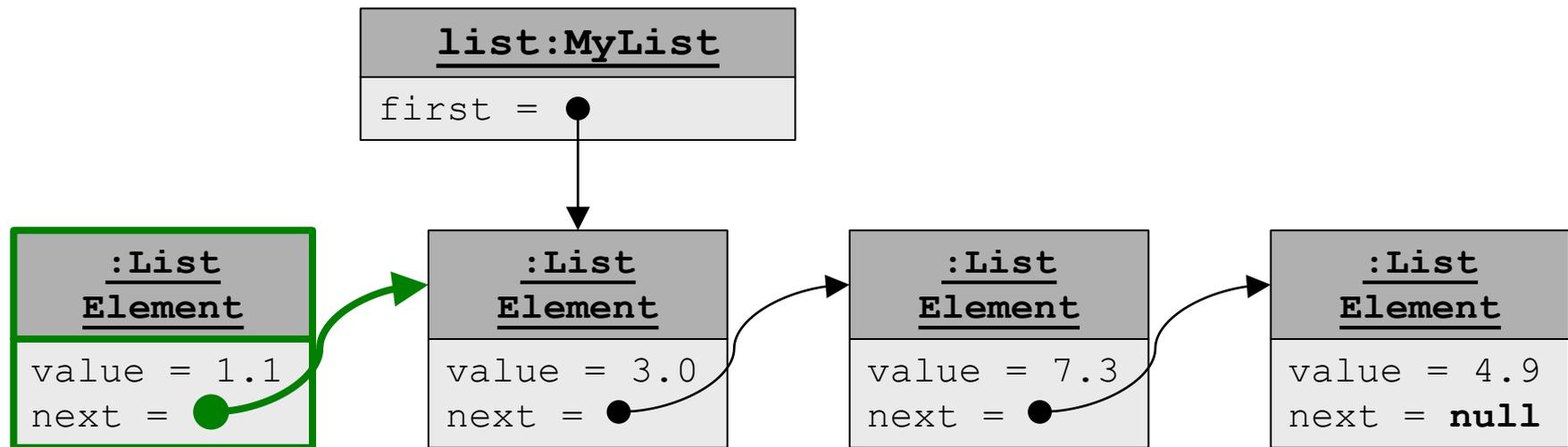
Verkettete Listen: addFirst Wiederholung (II)

```
list.addFirst(1.1)
```

=> Hinzufügen von 1.1 am **Anfang** der Liste `list=<3.0, 7.3, 4.9>`

Schritt 1: Erzeugen eines neuen `ListElement` mit

- `value = 1.1`
- `next = Zeiger auf ListElement für 3.0`



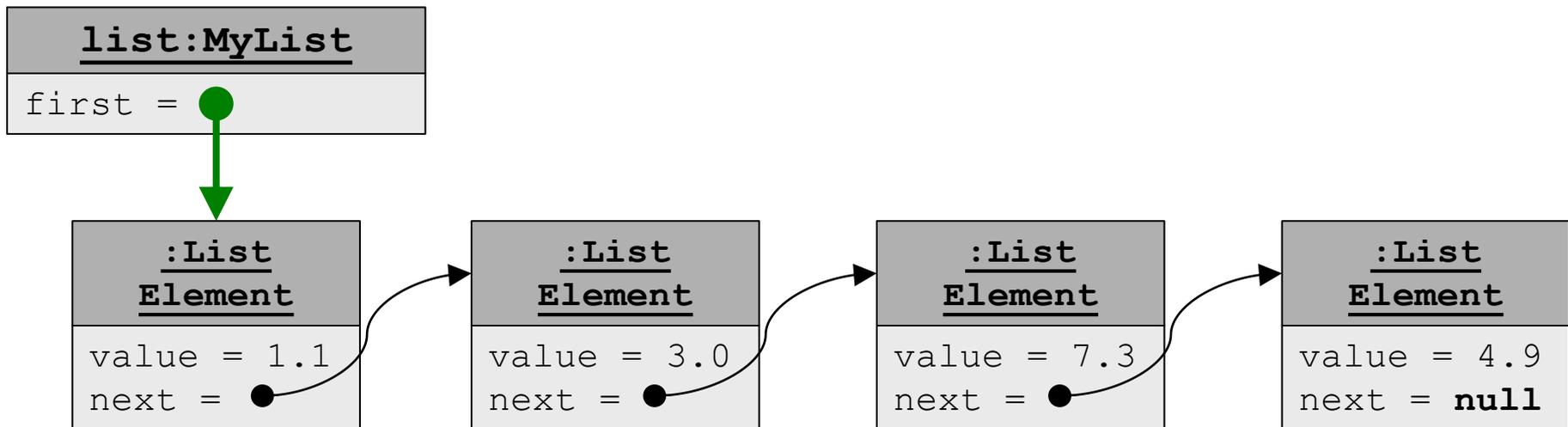


Verkettete Listen: `addFirst` Wiederholung (III)

```
list.addFirst(1.1)
```

=> Hinzufügen von 1.1 am **Anfang** der Liste `list=<3.0, 7.3, 4.9>`

Schritt 2: Speichern des neuen `ListElement` für 1.1 als erstes Element in der Liste, d.h. Zeiger `first` ändern





Verkettete Listen: `addFirst` Wiederholung (IV)

```
list.addFirst(1.1)
```

=> Hinzufügen von 1.1 am **Anfang** der Liste `list=<3.0, 7.3, 4.9>`

Implementierung von `addFirst` in der Klasse `MyList`:

```
public void addFirst(double d) {  
    this.first = new ListElement(d, this.first);  
}
```

Schritt 2

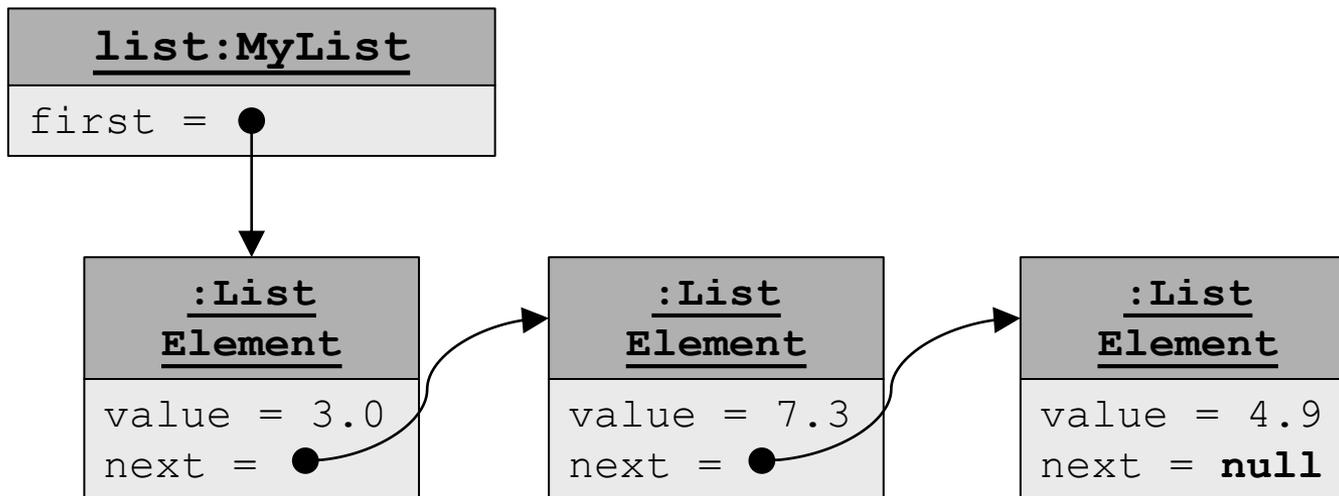
Schritt 1



Verkettete Listen: addLast (I)

```
list.addLast(1.1)
```

=> Hinzufügen von 1.1 am **Ende** der Liste `list=<3.0, 7.3, 4.9>`





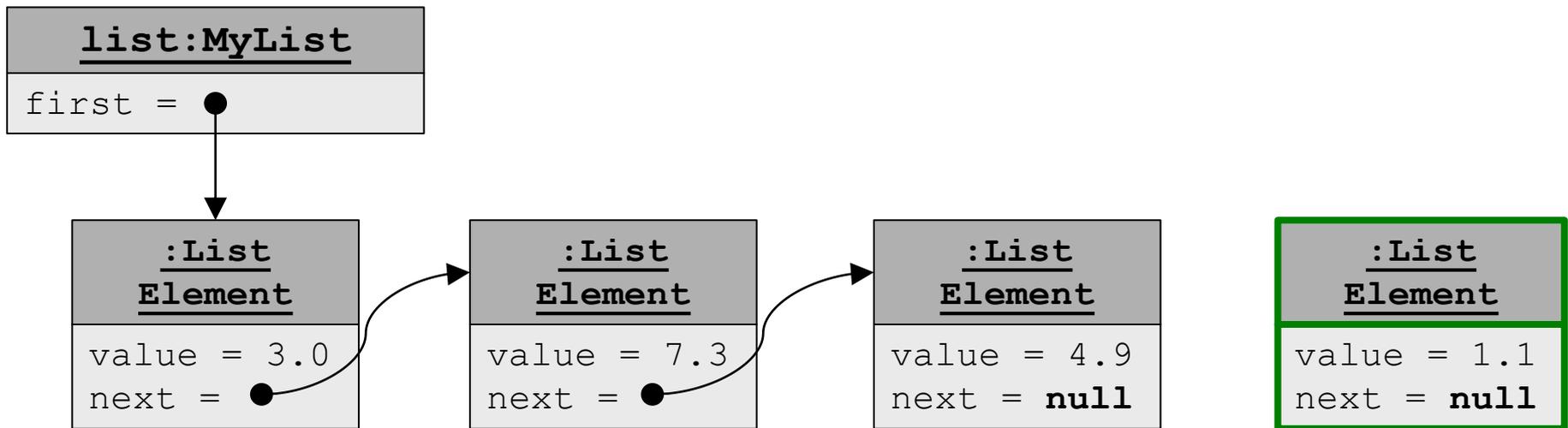
Verkettete Listen: addLast (II)

```
list.addLast(1.1)
```

=> Hinzufügen von 1.1 am **Ende** der Liste `list=<3.0, 7.3, 4.9>`

Schritt 1: Erzeugen eines neuen `ListElement` mit

- `value = 1.1`
- `next = null`



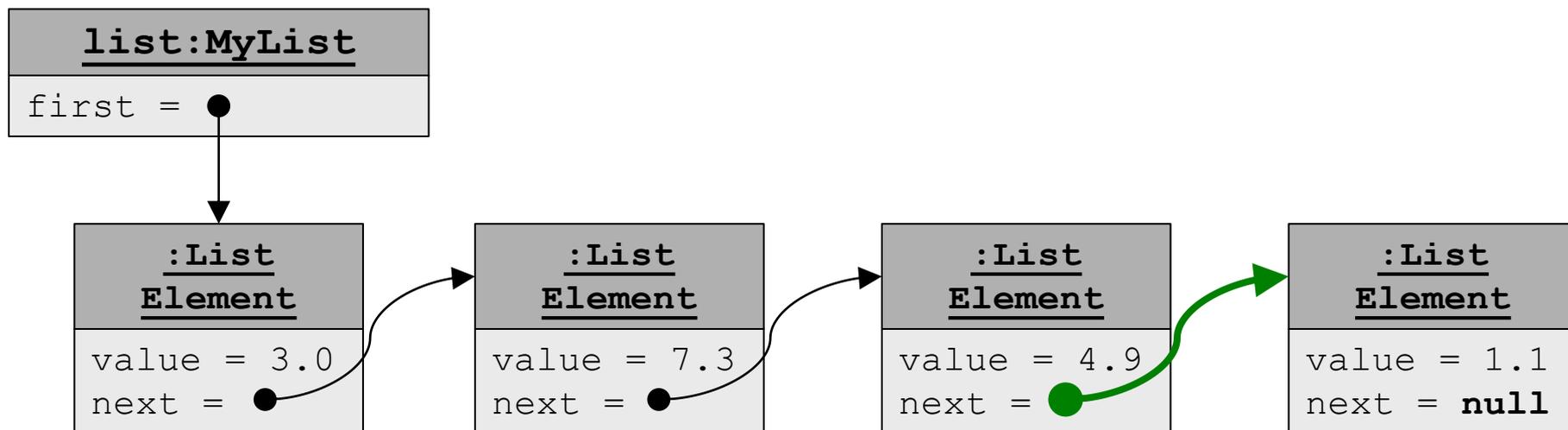


Verkettete Listen: addLast (III)

```
list.addLast(1.1)
```

=> Hinzufügen von 1.1 am **Ende** der Liste `list=<3.0, 7.3, 4.9>`

Schritt 2: Speichern des neuen `ListElement` für 1.1 als letztes Element in der Liste, d.h. die **gesamte** Liste bis zum letzten Element durchlaufen und dessen Zeiger `next` ändern ☹️





Verkettete Listen: addLast (IV)

```
list.addLast(1.1)
```

=> Hinzufügen von 1.1 am **Ende** der Liste `list=<3.0, 7.3, 4.9>`

Implementierung von `addLast` in der Klasse `MyList`:

```
public void addLast(double d) {  
    ListElement newElement = new ListElement(d);  
  
    ListElement e = this.first;  
    while (e.getNext() != null) {  
        e = e.getNext();  
    }  
    e.setNext(newElement);  
}
```

Schritt 1

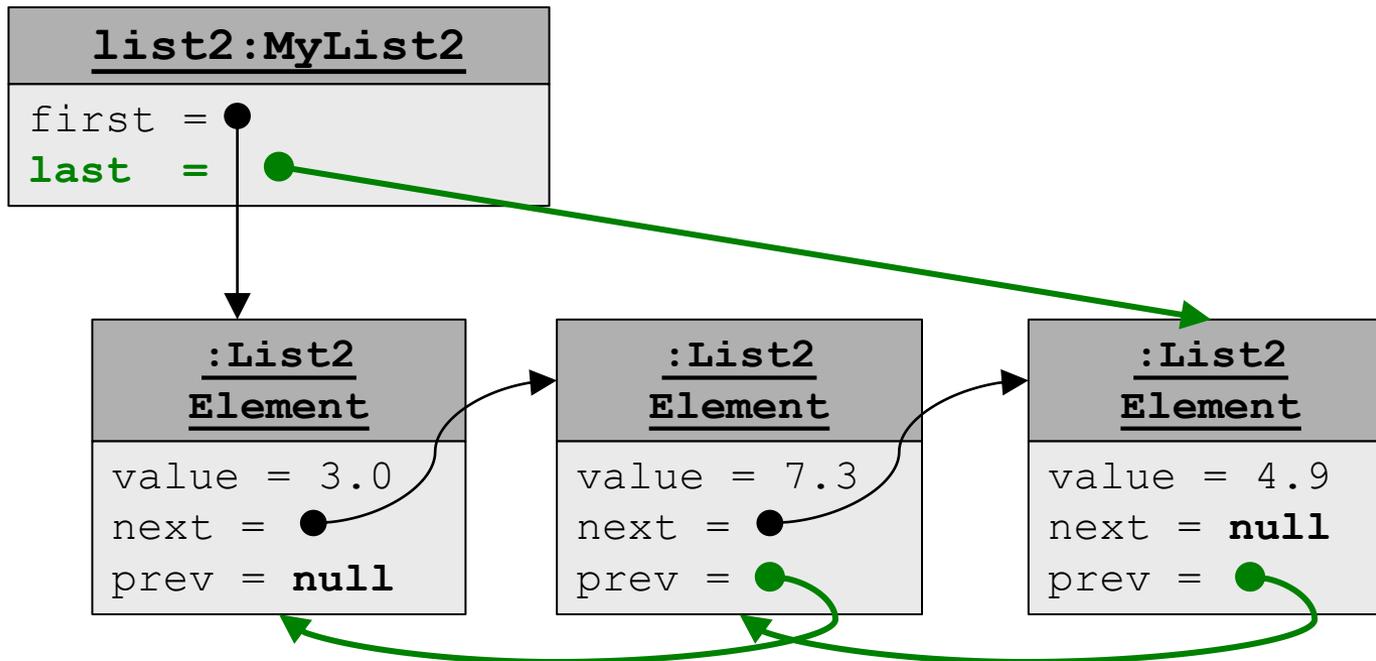
Schritt 2 ☹️



Doppelt verkettete Listen

Eine doppelt verkettete Liste speichert die Listenelemente als Kette, wobei jedes Listenelement seinen Nachfolger **und Vorgänger** kennt.

z.B. Repräsentation von `list2=<3.0,7.3,4.9>`

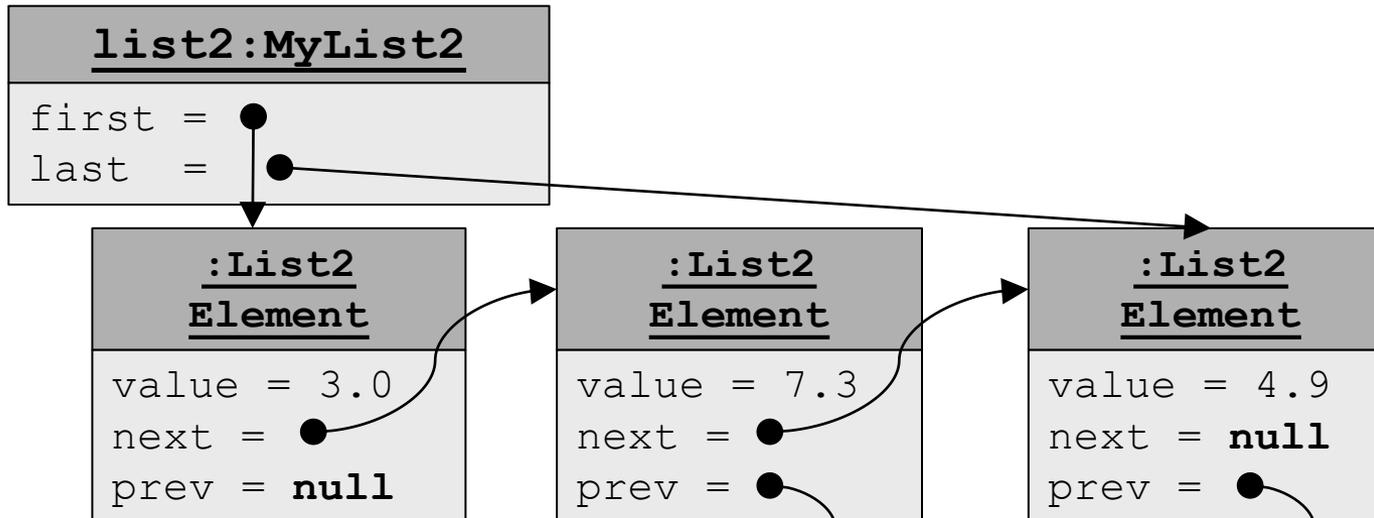




Doppelt verkettete Listen: addLast (I)

```
list2.addLast(1.1)
```

=> Hinzufügen von 1.1 am **Ende** der Liste `list2=<3.0, 7.3, 4.9>`





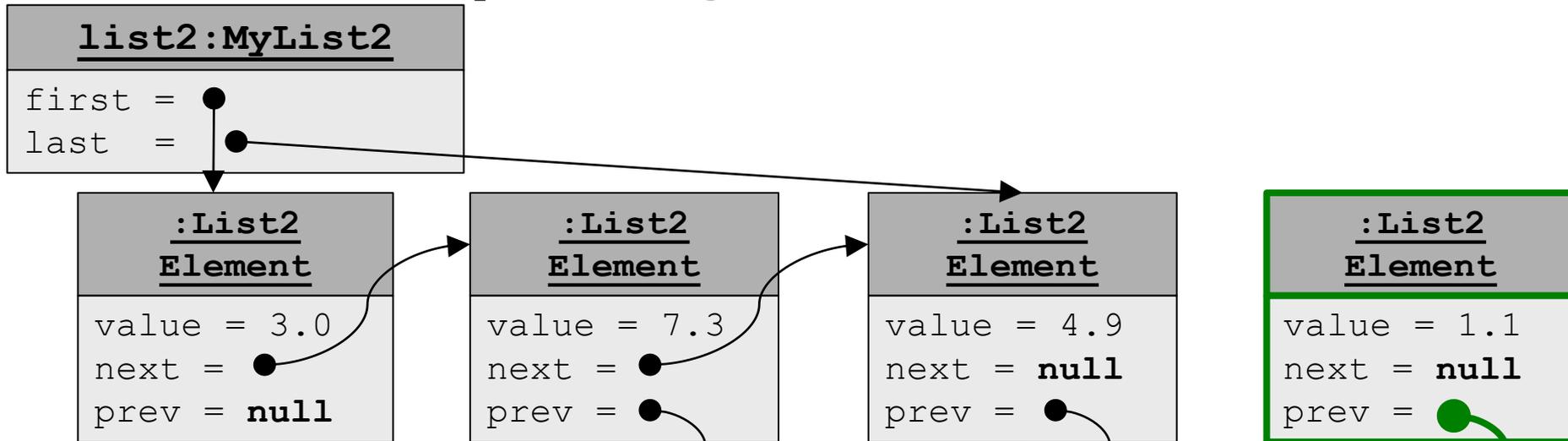
Doppelt verkettete Listen: addLast (II)

`list2.addLast(1.1)`

=> Hinzufügen von 1.1 am **Ende** der Liste `list2=<3.0, 7.3, 4.9>`

Schritt 1: Erzeugen eines neuen `List2Element` mit

- `value = 1.1`
- `next = null`, `prev =` Zeiger auf `List2Element` für 4.9





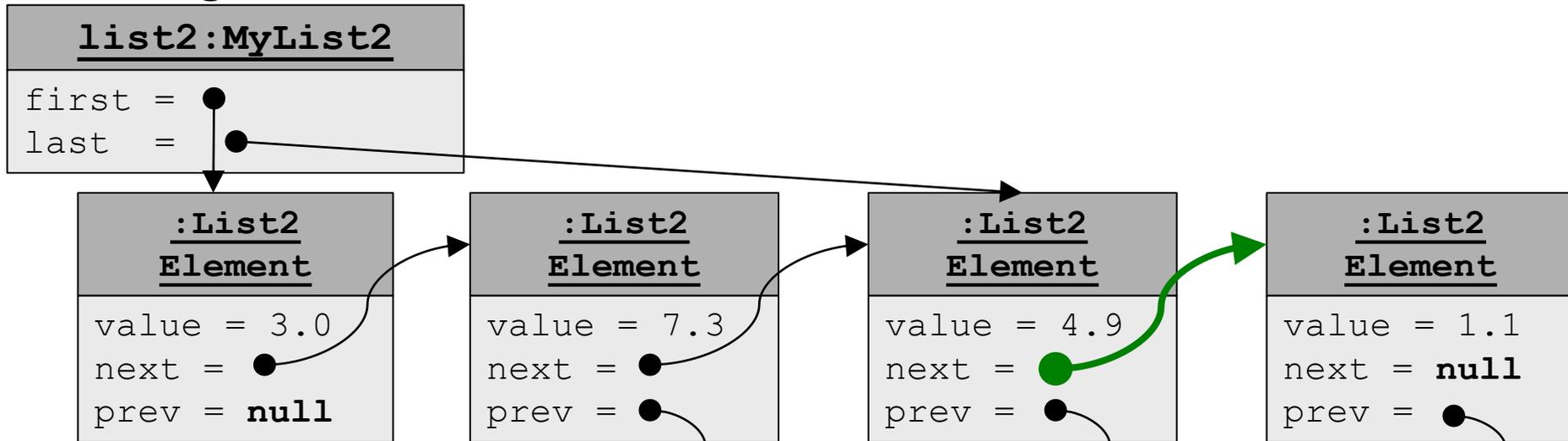
Doppelt verkettete Listen: addLast (III)

`list2.addLast(1.1)`

=> Hinzufügen von 1.1 am **Ende** der Liste `list2=<3.0, 7.3, 4.9>`

Schritt 2: Speichern des neuen `List2Element` für 1.1 als
Nachfolger des `List2Element` für 4.9

d.h. Zeiger `next` des `List2Element` für 4.9 ändern



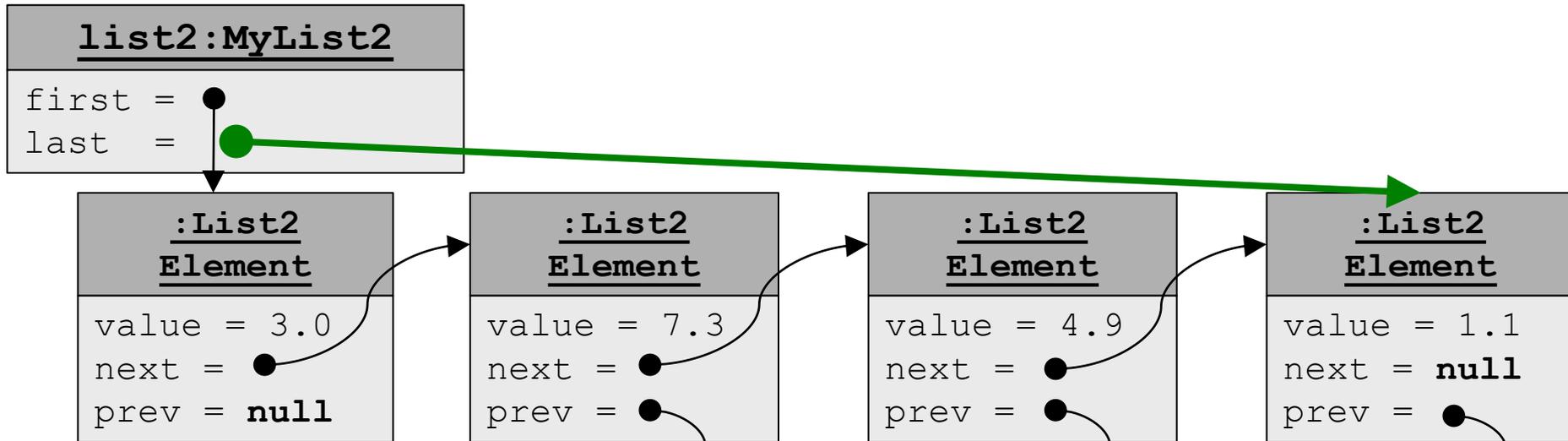


Doppelt verkettete Listen: addLast (IV)

`list2.addLast(1.1)`

=> Hinzufügen von 1.1 am **Ende** der Liste `list2=<3.0, 7.3, 4.9>`

Schritt 3: Speichern des neuen `List2Element` für 1.1 als letztes Element in der Liste, d.h. Zeiger `last` ändern





Doppelt verkettete Listen: addLast (V)

```
list2.addLast(1.1)
```

=> Hinzufügen von 1.1 am **Ende** der Liste `list2=<3.0, 7.3, 4.9>`

Implementierung von `addLast` in der Klasse `MyList2`:

```
public void addLast(double d) {  
    List2Element newElement =  
        new List2Element(d);  
    newElement.setPrev(this.last);  
  
    this.last.setNext(newElement);  
    this.last = newElement;  
}
```

Schritt 1

Schritt 2

Schritt 3



Doppelt verkettete Listen in Java: `LinkedList`

Zeitkomplexität von `addFirst`

- für einfach verkettete Listen: $O(1)$
- für doppelt verkettete Listen: $O(1)$

Zeitkomplexität von `addLast`

- für einfach verkettete Listen: $O(n)$ mit n = Länge der Liste
- für doppelt verkettete Listen: $O(1)$

Es gibt in der Java-Bibliothek eine Standard-Implementierung für verkettete Listen `LinkedList`, die wegen obiger Zeitkomplexität als doppelt verkettete Liste umgesetzt wurde.



Verwendung von `LinkedList`

Normalerweise implementiert man verkettete Listen nicht selbst, sondern verwendet die Standard-Implementierung `LinkedList`.

■ Konstruktor: z.B.

```
LinkedList<Double> list = new LinkedList<Double>();
```

■ Methoden:

- `list.addFirst(...)`
- `list.removeFirst(...)`
- `list.addLast(...)`
- `list.removeLast(...)`
- `list.size()`
- `list.get(i)`
- ... und viele mehr (siehe Java API)



Angeben des Typs von Listenelementen (I)

Eine Liste soll nur Werte des gleichen Typs speichern können:

- bei eigener Implementierung:
 - z.B. fester Typ `double` für das Attribut `value` in `ListElement`
 - z.B. fester Typ `Figur` für das Attribut `value` in `FigurListElement`
- bei Verwendung von `LinkedList`:
 - z.B. `LinkedList<Double> doubleListe =
new LinkedList<Double>();`
 - z.B. `LinkedList<Figur> figurListe =
new LinkedList<Figur>();`

!!Innerhalb der spitzen Klammern darf nur ein Klassenname stehen!!
(keine Grunddatentypen wie `double`) !!



Angeben des Typs von Listenelementen (II)

- Alle Methoden sind für die Liste `doubleListe` auf Werte vom Typ `Double` festgelegt:
 - z.B. **richtig**: `doubleListe.addFirst(1.0);`
 - falsch**: `doubleListe.addFirst(1);`
`doubleListe.addFirst("text");`
 - z.B. **richtig**: `Double d = doubleListe.get(0);`
 - falsch**: `Figur f = doubleListe.get(0);`
- Alle Methoden sind für die Liste `figurListe` auf Werte vom Typ `Figur` festgelegt:
 - z.B. **richtig**: `figurListe.addFirst(new Figur(...));`
 - falsch**: `figurListe.addFirst("text");`
 - z.B. **richtig**: `Figur f = figurListe.get(0);`
 - falsch**: `Double d = figurListe.get(0);`



Durchlaufen einer verketteten Liste: Beispiel

```
double summe = 0.0;
for (int i = 0; i < list2.size(); i++) {
    summe = summe + list2.get(i);
}
```

Für jedes `list2.get(i)` muss die Liste von vorne durchgelaufen werden! ☹️

list2:MyList2

first = ●
last = ●

:List2
Element

value = 3.0
next = ●
prev = null

:List2
Element

value = 7.3
next = ●
prev = ●

:List2
Element

value = 4.9
next = null
prev = ●

von außen nicht sichtbar



Durchlaufen einer LinkedList: Beispiel

```
LinkedList<Double> list = new LinkedList<Double> ();  
... // Hinzufügen der Elemente
```

```
double summe = 0.0;  
Iterator<Double> iterator = list.iterator();  
while (iterator.hasNext()) {  
    summe = summe + iterator.next();  
}
```

Kurzform:

```
double summe = 0.0;  
for (Double d : list) {  
    summe = summe + d;  
}
```

Der Iterator merkt sich,
welches Element in der Liste er
als nächstes zurückgeben muss.