

**Fachhochschule  
Dortmund**

University of Applied Sciences and Arts  
Fachbereich Informatik

**Algorithmen und Datenstrukturen**

# **VL03 – LISTEN**

# Inhalt

---

- Einführung
- Lineare Listen
  - Abstrakte Datenstruktur
  - Konkrete Datenstruktur: Implementierung durch Array
  - Konkrete Datenstruktur: Implementierung durch Verkettung
  - Komplexität
- Generics (Typparameter)

# EINFÜHRUNG

# Definitionen

---

- **Abstrakte Datenstruktur:**
  - Beschreibung der wesentlichen Eigenschaften eines Datenverbundes und der zulässigen Operationen auf der Datenstruktur
- **Konkrete Datenstruktur:**
  - Konkrete Art und Weise (Implementierung), wie die Daten im Speicher abgelegt und die Operationen auf ihnen realisiert werden

# Einführung

## Übersicht

Abstrakt

Lineare Listen

Konkret

Verkettung

- Einfach/doppelt
- Zyklisch
- LinkedList<E>

Array

- ArrayList<E>

Kontrollierte  
Zugriffspunkte

Stack

Queue

Priority-Queue

Stack

Queue

Ringpuffer

Lineare Listen

# ABSTRAKTE DATENSTRUKTUR

# Lineare Listen

## Übersicht

Abstrakt

Lineare Listen

Konkret

Verkettung

- Einfach/doppelt
- Zyklisch
- LinkedList<E>

Array

- ArrayList<E>

Kontrollierte  
Zugriffspunkte

Stack

Queue

Priority-Queue

Stack

Queue

Ringpuffer

# Abstrakte Datenstruktur

- **Linear geordnete** Folge von Elementen:
  - Die lineare Ordnung bezieht sich auf die Position in der Liste.
  - Jedes Listenelement besitzt einen **Vorgänger** und einen **Nachfolger**:



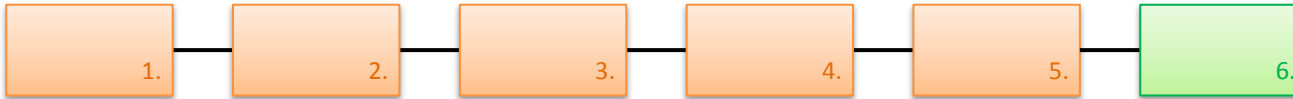
- Offensichtliche Ausnahmen:
  - Das erste Element (**Anfang**) hat keinen Vorgänger.
  - Das letzte Element (**Ende**) hat keinen Nachfolger.
- Offensichtliche Sachverhalte:
  - Eine Liste kann **leer** sein, das heißt sie enthält keine Elemente.
  - In einer Liste mit genau einem Element sind Anfang und Ende identisch.



## Lineare Listen

# Einige wichtige Operationen

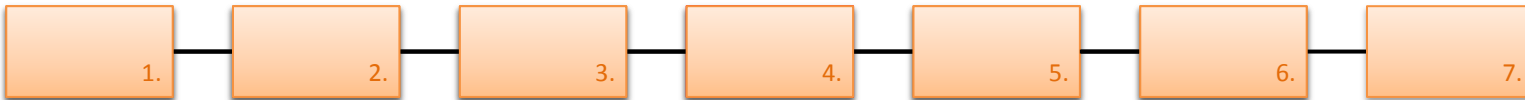
- Element am Ende **anhängen**:



- Element an beliebiger Stelle **einfügen**:



- Element an beliebiger Stelle **entfernen**:



- Angabe einer Stelle:
  - Durch Referenz (vor/hinter einem anderen Element)
  - Durch Index-Position

Lineare Listen

# KONKRETE DATENSTRUKTUR: IMPLEMENTIERUNG DURCH ARRAY

# Lineare Listen

## Übersicht

Abstrakt

Lineare Listen

Konkret

Verkettung

- Einfach/doppelt
- Zyklisch
- LinkedList<E>

Array

- ArrayList<E>

Kontrollierte  
Zugriffspunkte

Stack

Queue

Priority-Queue

Stack

Queue

Ringpuffer

# Implementierung durch Array

- Arrays sind in den meisten Programmiersprachen:
  - Statische Datenstruktur zur sequenziellen Speicherung
  - Länge wird bei der Deklaration (statische Arrays) oder Erzeugung mit `new` oder `malloc()` (dynamische Arrays) festgelegt
  - Länge während der Laufzeit nicht veränderbar
    - In C/C++ können Arrays, die dynamisch mit `malloc()` erzeugt wurden, durch `realloc()` in ihrer Größe angepasst werden, ohne dass Elemente umkopiert werden müssen. Hierzu muss die Plattform eine virtuelle Speicherverwaltung bieten.
  - Arrays implementieren bereits eine Lineare Liste: Vorgänger/Nachfolger ergeben sich durch die Index-Position der Elemente.
- Bewertung:
  - Sortieralgorithmen und Binäre Suche möglich:
    - Jedes Element in  $O(1)$  über Indexposition ansprechbar
  - Hohe Verschiebekosten  $O(n)$  für das Einfügen und Entfernen
  - Anzahl der Elemente begrenzt, ggf. wird Speicher verschwendet

Lineare Listen

# KONKRETE DATENSTRUKTUR: IMPLEMENTIERUNG DURCH VERKETTUNG

# Lineare Listen

## Übersicht

Abstrakt

Lineare Listen

Konkret

Verkettung

- Einfach/doppelt
- Zyklisch
- LinkedList<E>

Array

- ArrayList<E>

Kontrollierte  
Zugriffspunkte

Stack

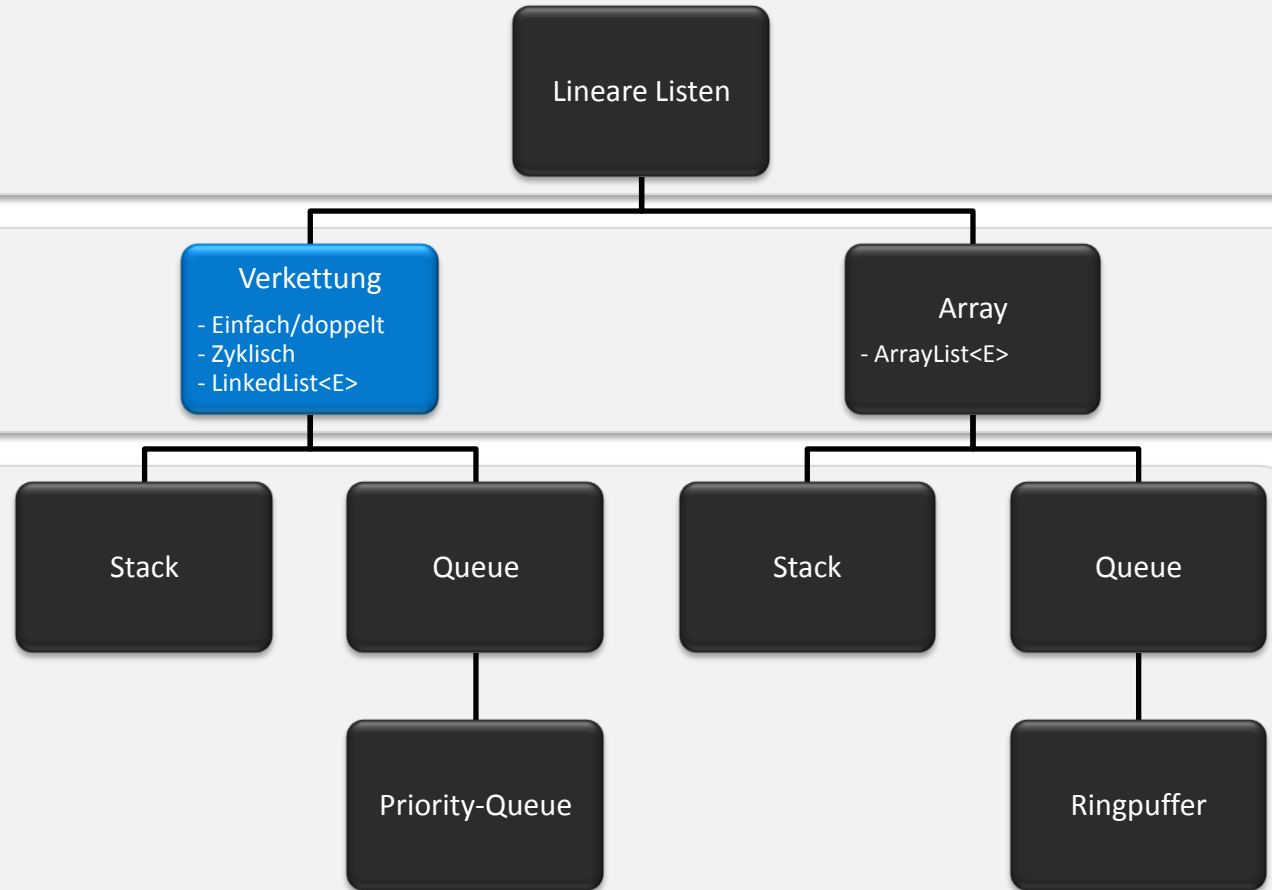
Queue

Priority-Queue

Stack

Queue

Ringpuffer



# Dynamische Datenstrukturen

---

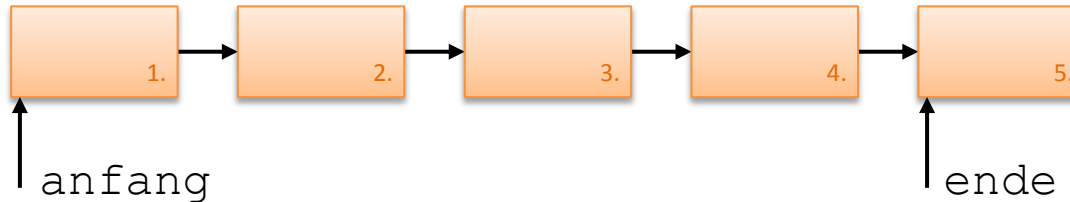
- Dynamische Datenstrukturen können wachsen oder schrumpfen, je nach aktuellem Speicherbedarf.
- Vorteile dynamischer Listen:
  - Elemente können an beliebiger Stelle eingefügt oder entfernt werden, ohne dass andere Elemente verschoben werden müssen.
  - Optimale Speicherauslastung

## Implementierung durch Verkettung

# Grundlegender Aufbau

- Implementierung durch Verkettung:

- Jedes Element enthält einen Zeiger auf das nachfolgende Element:



- Die Liste enthält einen Zeiger auf das erste Element (`anfang`).
- Das Anhängen an eine Liste kann beschleunigt werden, wenn auch ein Zeiger auf das letzte Element verwaltet wird (`ende`).

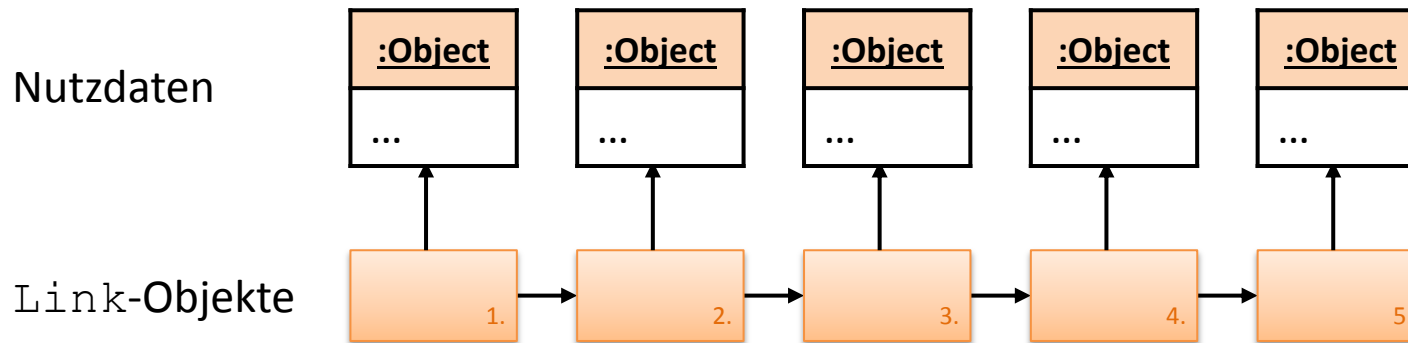
- Listengröße:

- Die Größe ändert sich während der Laufzeit (Einfügen und Löschen).
- Sie kann somit zum Compile-Zeitpunkt nicht bekannt sein.
- Der Compiler kann also den Speicherplatz nicht statisch allokalieren.
- Lösung: Speicherplatz für Listen-Elemente wird dynamisch auf dem Heap belegt und freigegeben!



## Implementierung durch Verkettung

# Link-Objekte



- Beliebige Objekte, die von `Object` erben, sollen verkettet werden.
  - Im Allgemeinen verfügen diese Objekte aber über keinen Zeiger, der einen Nachfolger referenzieren könnte (`naechster`).
  - Idee: wir benutzen besondere `Link-Objekte`, die sich einerseits verketten lassen, andererseits jeweils ein `Nutzdaten-Objekt` referenzieren können. Das letzte Element hat den Nachfolger `null`.


## Implementierung durch Verkettung

**Klasse** Link

```
public class Link
{
    private Object daten;
    Link naechster;

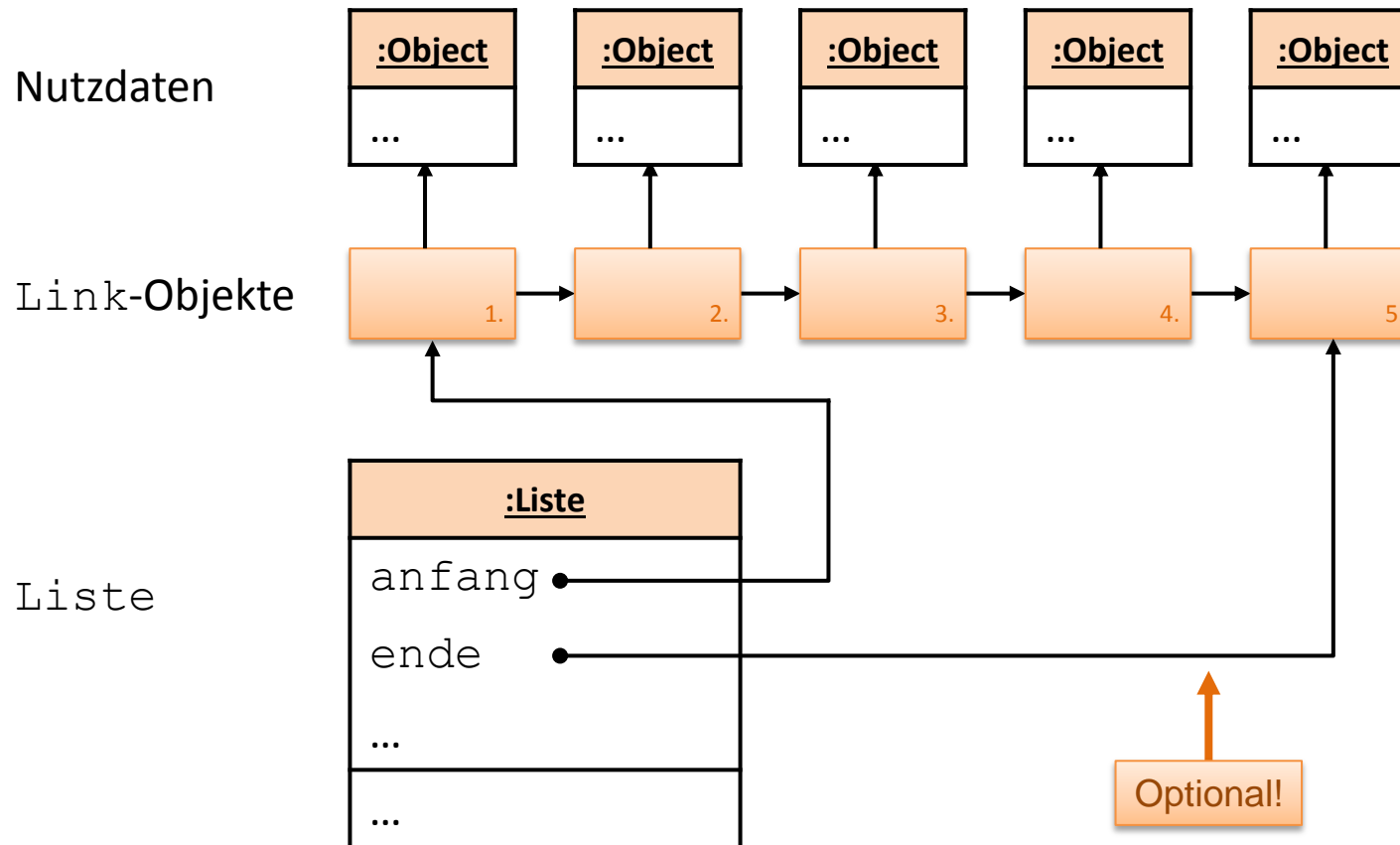
    public Link(Object daten, Link naechster)
    {
        this.daten = daten;
        this.naechster = naechster;
    }

    // Weitere Methoden
    ...
}
```



## Implementierung durch Verkettung

# Drei Bestandteile zum Aufbau verketteter Listen



- Das `Liste`-Objekt verwaltet die einzelnen Elemente und stellt Methoden für Operationen nach außen zur Verfügung.

Implementierung durch Verkettung

## Klasse Liste: Verwaltung der Link-Elemente

```
public class Liste
{
    protected Link anfang;
    // Ggf. weitere Zeiger, z.B. ende

    public Liste()
    {
        // Leere Liste: anfang-Zeiger ist null
    }

    // Operationen
    ...
}
```

## Implementierung durch Verkettung

**Klasse Liste: Einfügen am Anfang**

```
public class Liste
{
    protected Link anfang;
    // Ggf. weitere Zeiger, z.B. ende

    public Liste()
    {
        // Leere Liste: anfang-Zeiger ist null
    }

    public void einfuegenAnfang(Object neuesElement)
    {
        anfang = new Link(neuesElement, anfang);
    }
    ...
}
```

## Implementierung durch Verkettung

**Klasse Liste: Löschen am Anfang**

```
public class Liste
{
    protected Link anfang;
    // Ggf. weitere Zeiger, z.B. ende

    public Liste()
    {
        // Leere Liste: anfang-Zeiger ist null
    }

    public void entferneAnfang()
    {
        if (anfang!=null)
            anfang = anfang.naechster;
    }
    ...
}
```

Kann eine `NullPointerException` werfen, wenn `anfang==null` ist!

## Implementierung durch Verkettung

**Klasse Liste: Liste vollständig leeren**

```
public class Liste
{
    protected Link anfang;
    protected Link ende;    // Optional
    // Ggf. weitere Zeiger, z.B. für "Aktuelles Element"

    public Liste()
    {
        // Leere Liste: anfang- und ende-Zeiger sind null
    }

    public void leeren()
    {
        anfang = ende = null;
    }
    ...
}
```

## Implementierung durch Verkettung

**Klasse Liste: Anhängen ans Ende**

...

// ende-Zeiger beschleunigt Anhängen von  $O(n)$  auf  $O(1)$

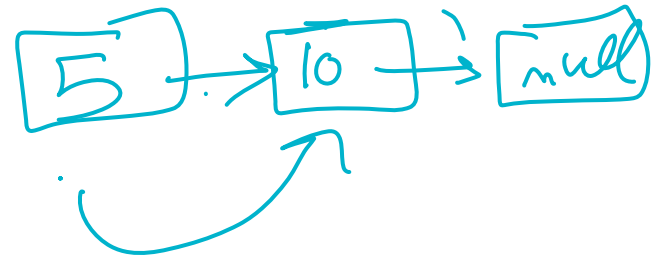
```
public void anhaengenEnde(Object neuesElement)
{
    Link neu = new Link(neuesElement, null);
    if (anfang == null)
    {
        anfang = ende = neu;
    }
    else
    {
        ende = ende.naechster = neu;
    }
}
```

ende = 5

neu = 10

ende = 5

ende.next = null



ende = neu



## Klasse Liste: Iterieren

- Das **Iterieren**, also schrittweises Besuchen aller Elemente, ist eine wichtige Grundoperation verketteter Listen.
- Das Iterieren ist die Basis für viele Operationen, z.B.:
  - Ausgeben aller Elemente
  - Suchen eines Elements
  - Zählen der gespeicherten Elemente
  - ...

## Implementierung durch Verkettung

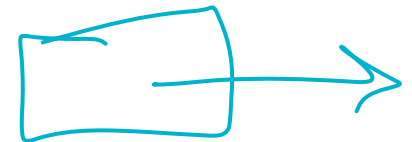
**Klasse Liste: Iterieren**

...

```
public void iterieren()
{
    // Kopie von anfang anlegen, da sonst die Liste
    // zerstört wird!
    Link zeiger = anfang;

    // Wird bei leerer Liste nicht durchlaufen!
    while (zeiger != null)
    {
        ...

        zeiger = zeiger.naechster;
    }
}
```




## Implementierung durch Verkettung

**Klasse Liste: Iterieren/Elemente ausgeben**

...

```
public void ausgeben()  
{  
    Link zeiger = anfang;  
  
    while (zeiger!=null)  
    {  
        System.out.println(zeiger.getDaten());  
  
        zeiger = zeiger.naechster;  
    }  
}
```



## Implementierung durch Verkettung

**Klasse Liste: Iterieren/Element suchen**

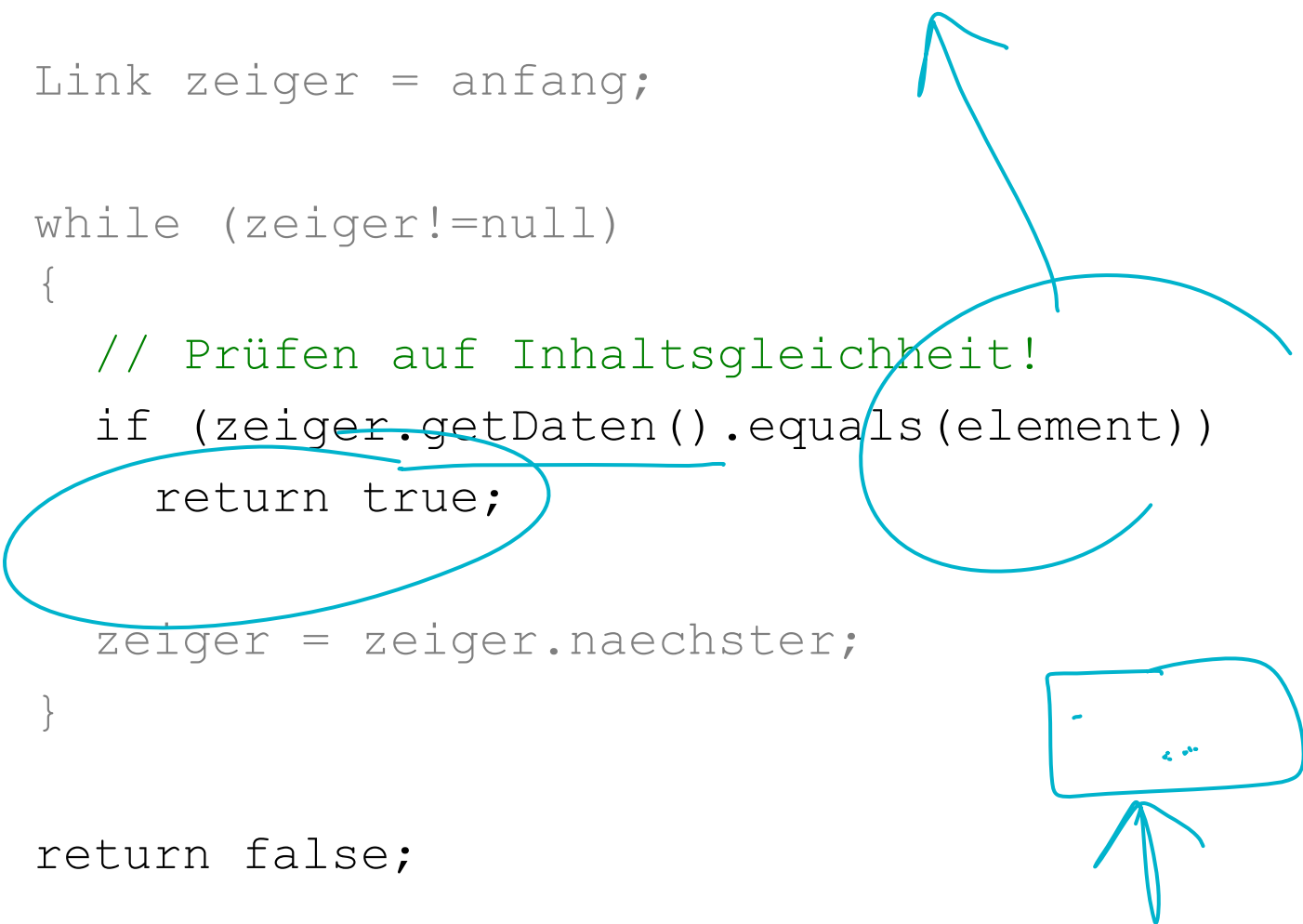
...

```
public boolean suchen(Object element)
{
    Link zeiger = anfang;

    while (zeiger!=null)
    {
        // Prüfen auf Inhaltsgleichheit!
        if (zeiger.getDaten().equals(element))
            return true;

        zeiger = zeiger.naechster;
    }

    return false;
}
```



## Implementierung durch Verkettung

**Klasse Liste: Iterieren/Elemente zählen**

...

```
public int zaehlen()  
{  
    int anzahl = 0;  
    Link zeiger = anfang;  
    while (zeiger != null)  
    {  
        anzahl++;  
        zeiger = zeiger.naechster;  
    }  
    return anzahl;  
}
```



## Implementierung durch Verkettung

# Iterator

- Um die Liste auf diese Art und Weise zu iterieren, benötigt man direkten Zugriff auf die Verkettung, z.B. durch Ableiten einer eigenen Klasse und Hinzufügen von Methoden für jeden Anwendungsfall. Das ist zu aufwändig!
- Die Lösung ist ein Iterator als Abstraktion des Iterierens:
  - Objekt, das Methoden zum Iterieren bereitstellt
  - Kann vom Benutzer der Liste (also von außen) angefordert werden
  - Eigentliches Iterieren findet über den Iterator statt, und nicht über das `Liste`-Objekt:
    - Keine Zusatzklassen für jeden Anwendungsfall erforderlich
    - `Liste`-Objekt bleibt gekapselt

## Implementierung durch Verkettung

# Iterator

- Die Methode `iterator()` liefert einen Iterator zurück:

```
Liste li = new Liste();
```

```
...
```

```
// Iterator anfordern
```

```
ListeIterator it = li.iterator();
```

```
// Iterieren (hier: Ausgabe auf der Konsole)
```

```
while (it.hasNext())
```

```
    System.out.println(it.next());
```

- Die Klasse `ListeIterator` deklariert die folgenden Methoden:

- `boolean hasNext()`

Gibt `true` zurück, wenn noch Elemente übrig sind.

- `Object next()`

Liefert das aktuelle Element zurück und bewegt den Iterator weiter auf den Nachfolger.

| <i>ListeIterator</i>    |
|-------------------------|
|                         |
| + hasNext()<br>+ next() |

## Implementierung durch Verkettung

**Klasse `Liste`: Aktueller Zeiger**

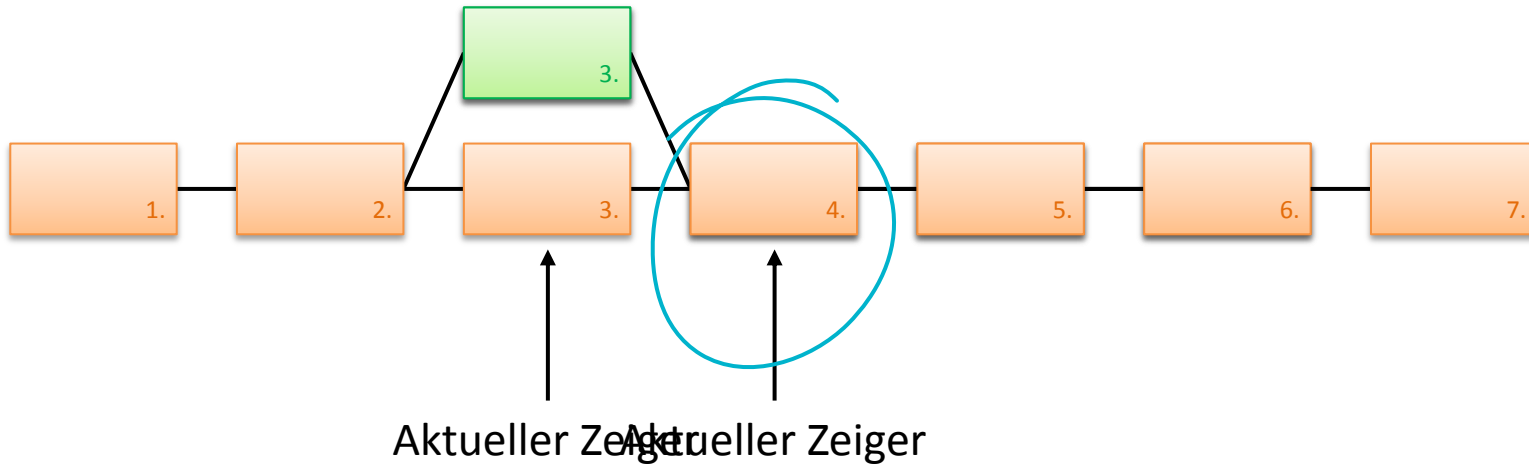
- Wir ergänzen die Liste nun um einen **Aktuellen Zeiger**:
  - Markiert ein bestimmtes Element der Liste
  - Wir fügen gleich Methoden zur Klasse `Liste` hinzu, um ein neues Element vor dem aktuellen Element einzufügen, oder um das aktuelle Element zu entfernen.
- Methoden zur Verwaltung des Aktuellen Zeigers:
  - `void setzeAktuellerZeigerZurueck()`  
Setzt den Aktuellen Zeiger auf den Anfang der Liste zurück.
  - `boolean suchenElement(Object element)`  
Sucht `element`, und setzt den aktuellen Zeiger auf dieses Element.
  - `boolean weitereElemente()`  
Liefert `true` zurück, wenn der Aktuelle Zeiger das Listenende noch nicht erreicht hat (also noch weitere Elemente besucht werden können).
  - `public Link aktuellerZeiger()`  
Liefert das Aktuelle Element zurück.



## Implementierung durch Verkettung

**Klasse Liste: Einfügen in der Mitte**

- Annahme: die Einfügestelle ist durch den Aktuellen Zeiger markiert. Ein neues Element soll davor eingefügt werden:

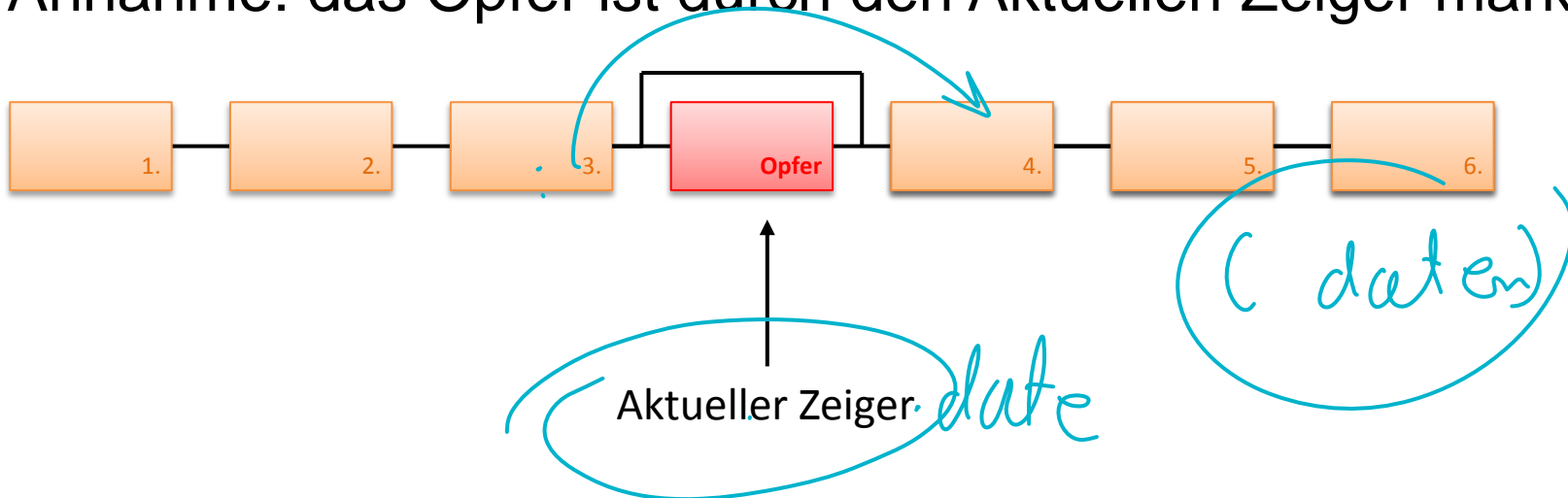


- Das aktuelle Link-Element wird dabei **nicht** verändert! Alle Veränderungen finden am **Vorgänger** des aktuellen Elements statt!

## Implementierung durch Verkettung

Klasse `Liste`: Entfernen in der Mitte

- Annahme: das Opfer ist durch den Aktuellen Zeiger markiert.



- Auch beim Entfernen eines Elements wird das aktuelle `Link`-Element **nicht** verändert! Alle Veränderungen finden am **Vorgänger** des aktuellen Elements statt!
- Dass `.naechster` des Opfers noch auf ein Element der Verkettung zeigt, spielt übrigens keine Rolle: beim Iterieren der Liste wird das Opfer ab sofort einfach übersprungen.

## Implementierung durch Verkettung

**Klasse** `Liste`: **Vorgänger-Problem**

- Sowohl beim Einfügen als auch Entfernen in der Mitte müssen wir Änderungen am **Vorgänger** des Aktuellen Zeigers durchführen. Wir können aber nur zum **Nachfolger** springen!
- Idee: tatsächlich speichern wir in der Klasse `Liste` immer den **Vorgänger** des Aktuellen Zeigers ab. Durch trickreiche Implementierung u.a. der Methode `aktuellerZeiger()` ist dies von außen nicht sichtbar:

```
// Vorgänger von AktuellerZeiger
protected Link vorgaengerAktuellerZeiger;
...
public Link aktuellerZeiger()
{
    return istLeer() ? null :
        (vorgaengerAktuellerZeiger == null) ? anfang :
        vorgaengerAktuellerZeiger.naechster;
}
```

## Implementierung durch Verkettung

**Klasse Liste: Einfügen in der Mitte**

```
// Einfügen vor die aktuelle Zeigerposition
public void einfuegenElement(Object neuesElement)
{
    // Wenn die Liste leer ist, entspricht dies einem Anfügen
    if (istLeer())
    {
        anfuegenElement(neuesElement);
        return;
    }

    Link neu = new Link(neuesElement, aktuellerZeiger());

    // Wenn die Liste nur ein Element hat, muss anfang gesetzt werden
    if (vorgaengerAktuellerZeiger == null)
    {
        anfang = vorgaengerAktuellerZeiger = neu;
        return;
    }

    // Vorgänger-Element von neu zeigt jetzt auf neu
    vorgaengerAktuellerZeiger = vorgaengerAktuellerZeiger.naechster = neu;
}
```

# DOPPELT VERKETTETE LISTEN



## Doppelt verkettete Listen

# Unterschied zur einfachen Verkettung

- Unsere verketteten Listen sind bisher **einfach verkettet**, das heißt jedes Element enthält einen Zeiger auf seinen Nachfolger.
  - Dadurch können die Listen leicht vom Anfang zum Ende iteriert werden.
- **Doppelt verkettete Listen** speichern in jedem Element einen Zeiger auf den Vorgänger und den Nachfolger ab.
  - Ausnahmen: das erste Element hat keinen Vorgänger, das letzte Element hat keinen Nachfolger (die Zeiger sind jeweils `null`).
  - Dadurch können doppelt verkettete Listen in beide Richtungen durchlaufen werden.
  - Aufbau:



## Doppelt verkettete Listen

# Bewertung

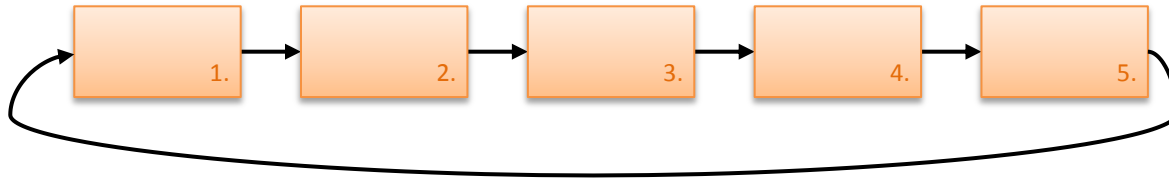
- Vorteile:
  - Die Liste kann effizient vorwärts und rückwärts iteriert werden.
  - Das Entfernen von Elementen ist bei gegebener Position noch einfacher zu implementieren, da der Vorgänger des Opfers stets bekannt ist.
- Nachteile:
  - Bei jeder Operation müssen zwei Zeiger statt einem manipuliert werden.
  - Jedes Element benötigt zusätzlichen Speicherplatz für den zweiten Zeiger.

# ZYKLISCH VERKETTETE LISTEN

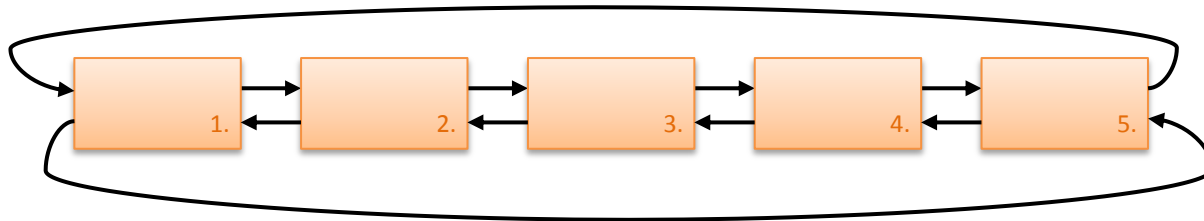


# Zyklisch verkettete Listen

- Einfach und doppelt verkettete Listen können zusätzlich noch zyklisch verkettet sein.
  - Der Nachfolger des letzten Elements ist das erste Element:



- Bei doppelter Verkettung ist der Vorgänger des ersten Elements das letzte Element:



- Dadurch kann eine Liste unendlich oft iteriert werden, z.B. für eine Liste mit rotierenden Aufgaben.

Lineare Listen

**KOMPLEXITÄT**

## Komplexität

## Implementierung durch Array

- Einfügen an gegebener Stelle  $i$ :
  - $n-i$  Elemente müssen verschoben werden
  - Aufwand:  $O(n)$
- Entfernen an gegebener Stelle  $i$ :
  - $n-i-1$  Elemente müssen verschoben werden
  - Aufwand:  $O(n)$
- Anhängen und Entfernen am Ende:
  - $O(1)$

| Operation                       | Zeitaufwand (worst case) |
|---------------------------------|--------------------------|
| Einfügen/Entfernen am Anfang    | $O(n)$                   |
| Einfügen/Entfernen in der Mitte | $O(n)$                   |
| Anhängen/Entfernen am Ende      | $O(1)$                   |
| Element suchen                  | $O(n)$                   |
| Zugriff auf $i$ -tes Element    | $O(1)$                   |

## Komplexität

## Implementierung durch Verkettung

- Einfügen und Entfernen:
  - An einer **vorgegebenen** Position (Zeiger auf das Element bzw. dessen Vorgänger ist vorhanden) in  $O(1)$  möglich
    - Gilt insbesondere für den Anfang und ggf. das Ende (nur mit `ende`-Referenz)
  - An einer **beliebigen** Position nur in  $O(n)$  möglich, da das Element bzw. dessen Vorgänger erst gesucht werden muss
- Wahlfreier Zugriff auf Stelle  $i$ :
  - Aufwand  $O(n)$ , da das  $i$ -te Element erst gesucht werden muss

| Operation                       | Zeitaufwand (worst case)                                      |
|---------------------------------|---|
| Einfügen/Entfernen am Anfang    | <b><math>O(1)</math></b>                                      |
| Einfügen/Entfernen in der Mitte | $O(n)$ , <b><math>O(1)</math> mit Referenz</b>                |
| Anhängen/Entfernen am Ende      | <b><math>O(n)</math></b> , $O(1)$ mit <code>ende</code> -Ref. |
| Element suchen                  | $O(n)$  |
| Zugriff auf $i$ -tes Element    | <b><math>O(n)</math></b>                                      |

Farbliche Darstellung im Unterschied zur Implementierung durch Arrays

## Komplexität

# Abschließende Bewertung verketteter Listen

- Gut geeignet, um Daten dynamisch zu verwalten
- Einfügen und Entfernen von Elementen einfach
- Sequenzielles Iterieren effizient in  $O(n)$ 
  - Arrays sind in der Praxis jedoch noch schneller!
- Zugriff aufwändig, wenn ein bestimmtes Element benötigt wird
  - Liste muss von vorne nach hinten durchlaufen werden
  - Sortieralgorithmen und Binäre Suche benötigen jedoch einen effizienten wahlfreien Zugriff auf beliebige Elemente
  - Somit sind verkettete Listen ungeeignet für sortierte Datenbestände.
  - Effiziente Datenstruktur für sortierte Daten: **gute** Bäume (VL07)

# GENERIC (TYPEPARAMETER)

## Generics

# Problemstellung

- Bei der Verwendung von Bibliotheken gibt es ein prinzipielles Problem: beim Entwickeln weiß man nicht, in welcher Umgebung die Bibliothek später einmal eingesetzt wird.
- Beispiel:
  - Beim Entwickeln einer allgemeinen Klasse für verkettete Listen kann man nicht vorhersehen, welche Art von Objekten einmal gespeichert werden sollen.
- Bisherige Lösung:
  - In Java erben alle Klassen von `Object`!
  - Daher haben wir z.B. die Klasse `Liste` so entworfen, dass Objekte vom Typ `Object` gespeichert werden. Dadurch ist die Liste automatisch für alle Java-Objekte geeignet.

## Generics

# Fehlende Typsicherheit

- Beispiel:

```
Liste li = new Liste();
```

```
li.anfuegenElement(new Integer(1));
```

```
li.anfuegenElement(new Integer(2));
```

```
li.anfuegenElement(new Student("Fritz", "Zimmermann"));
```

```
int i1 = (Integer)li.naechstesElement();
```

```
int i2 = (Integer)li.naechstesElement();
```

```
int i3 = (Integer)li.naechstesElement();
```

- Der letzte Typecast löst eine Exception aus, da das zurückgelieferte Element **nicht** vom Typ `Integer` ist!
- Zur Laufzeit ist keine Unterstützung durch den Compiler möglich.



## Generics

# Typparameter

- Wie können wir Typsicherheit herstellen?
  - Für jeden Datentyp eine eigene Listen-Klasse schreiben (z.B. für `String`, `Integer`, `Student`, ...)?
  - Es ist höchst fehleranfällig und unkomfortabel für den Entwickler, dieselben Algorithmen immer wieder neu zu schreiben.
- Verbesserte Lösung: Klassen verwenden **Typparameter**!
  - **C++**: Template genannt
  - **Java**: **Generics** genannt
- Vorteile:
  - Algorithmen müssen nur einmal entwickelt und programmiert werden.
  - Die Datenstrukturen werden vom Compiler automatisch (und damit ohne Fehler) für unterschiedliche Datentypen angepasst.
  - Die Klassen werden dadurch **generisch**, d.h. universell einsetzbar.

## Generics

# Umrüstung (1/4)

- `Liste.java`:

```
public class Liste
{
    ...
    public Object naechstesElement()
    {
        ...
    }
    ...
    public void an fuegenElement(Object neuesElement)
    {
        ...
    }
}
```

## Generics

## Umrüstung (2/4)

- Hinzufügen eines Typparameters T:
  - In spitzen Klammern
  - Wird innerhalb der Klasse wie jeder andere Datentyp verwendet
  - Wird beim Erzeugen von Objekten durch konkreten Typ ersetzt

```
public class Liste<T>
{
    ...
    public Object naechstesElement()
    {
        ...
    }
    ...
    public void anfuegenElement(Object neuesElement)
    {
        ...
    }
}
```

## Generics

## Umrüstung (3/4)

- Statt `Object` wird nun in der gesamten Klasse `T` verwendet:

```
public class Liste<T>
{
    ...
    public T naechstesElement()
    {
        ...
    }
    ...
    public void an fuegenElement(T neuesElement)
    {
        ...
    }
}
```

## Generics

# Umrüstung (4/4)

- Verwendung der Klasse `Liste<T>`:

```
// Liste nur für Integer
```

```
Liste<Integer> li = new Liste<Integer>();
```

```
li.anfuegenElement(new Integer(1));
```

```
li.anfuegenElement(new Integer(2));
```

```
// Die folgende Zeile compiliert nicht!
```

```
li.anfuegenElement(new Student("Fritz", "Zimmermann"));
```

```
// Kein Typecast mehr erforderlich!
```

```
int i1 = li.naechstesElement();
```

```
int i2 = li.naechstesElement();
```

- Auch Unterklassen von `Integer` wären als Elemente von `Liste<Integer>` zugelassen.

## Generics

## Methoden mit besonderen Typparametern

- Bei statischen Methoden mit Typparametern oder Methoden mit Typparametern, die nicht zur Klasse gehören, müssen diese besonderen Typparameter vor dem Rückgabetyp der Methode angegeben werden:

```
class ClearableList<T> extends Liste<T>
{
    public static <T> void printList(Liste<T> li)
    {
        Iterator<T> iterator = li.iterator();
        ...
    }

    public <E> void compareList(Liste<E> li)
    {
        ...
    }
}
```

## Generics

# Beschränkte Typparameter

- Typparameter können in Java auf eine bestimmte Oberklasse beschränkt werden:

```
public class Liste<T extends Mitarbeiter>
{
    ...
    public double berechneGesamtgehaltProMonat()
    {
        // Aufruf von berechneGehalt eines jeden Elements
        // notwendig zur Berechnung des Gesamtgehalts
        ...
    }
    ...
}
```

- So wird sichergestellt, dass für `T` nur die Klasse `Mitarbeiter` oder Unterklassen eingesetzt werden können.
  - Der Zugriff auf spezielle Methoden und Attribute von `Mitarbeiter` ist so gestattet.

# Lernfragen

- Sie kennen die drei Elemente, die zum Aufbau einer verketteten Liste benötigt werden
- Sie kennen die verschiedenen Methoden auf verketteten Listen und können die zugehörigen Algorithmen an Objektdiagrammen erläutern
- Sie kennen die asymptotische Laufzeit der Methoden von sequenziell und verkettet gespeicherten Linearen Listen
- Sie beherrschen Generics, und können diese sicher einsetzen