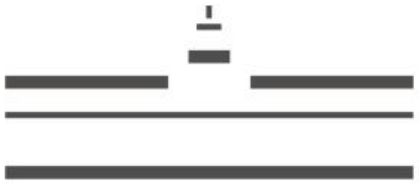


Westfälische
Wilhelms-Universität
Münster

Programmieren in Java

Datenstrukturen

wissen.leben
WWU Münster

Name: Patrick Förster, Thorsten Hovenga

Zentrum für
Informations
Verarbeitung

Gruppierung von Daten

- Datentypen, die eine Menge an Daten gruppieren und auf dieser Gruppierung eine gewisse Logik bereitstellen werden „Datenstrukturen“ genannt
- Bisher wurde zur Gruppierung immer Arrays angewandt
- Bis auf die Index-Funktion gibt es allerdings keinerlei logischer Operationen, mit denen der Array manipuliert werden könnte
- Wenn die zu organisierende Menge dynamisch ist, machen Arrays einem das Leben daher schwer:
    - Arrays sind statisch: Länge einmal festgelegt, nicht mehr änderbar
    - „Mal eben“ ein Element hinzufügen oder entfernen ist aufwändig
    - Zumeist muss der Array-Inhalt kopiert werden
    - Große Arrays ➔ großer Kopieraufwand

Collection-API

- Java bietet mit der Collection-API einen Basissatz an Datenstrukturen
- All diese Strukturen sind dynamisch und damit flexibler als Arrays
- Jede der Strukturen basiert auf einem der folgenden Interfaces
  - **`Collection`**: Interface für Listen bzw. Mengen
  - **`Map`**: Interface für eine Gruppe assoziativ angeordneter Elemente
- Sowohl **`Collection`** als auch **`Map`** sind generisch, d.h. der Datentyp der Inhalte kann generisch spezifiziert werden:

```
Collection<String> list = new LinkedList<String>();
dice("Dies ist ein String der durcheinander wird", list);

➔ [ein, ist, String, Dies, wird, der, durcheinander]
```

- **`LinkedList`** ist eine Implementierung von **`Collection`**
- Die Methode **`dice`** fühlt die List in zufälliger Reihenfolge mit String-Fragmenten getrennt nach dem Leerzeichen aus einem größeren String

Das Interface Collection

```
public interface Collection<E> extends Iterable<E> {
    boolean add(E e);
    boolean remove(Object o);
    boolean contains(Object o);
    int size();
    boolean isEmpty();
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
}
```

Das Interface `Collection`

- Jede Implementierung des **`Collection`**-Interfaces unterstützt das
  - Hinzufügen und Entfernen von Elementen (einzeln oder massenhaft)
  - Überprüfen ob gewissen Inhalte vorhanden sind
  - Abfragen der Größe
  - Umwandeln in einen Array
- Desweiteren erweitert Collection das Interface **`Iterable`**
- Dies schreibt die Methode **`iterator()`** vor
- Alle Klassen, die dieses Interface implementieren können in der **`foreach`**-Schleife genutzt werden:

```
Collection<String> list = new LinkedList<String>();
dice("Dies ist ein String der durcheinander wird", list);
for (String element : list) {
   System.out.println(element);
}
```

Listen

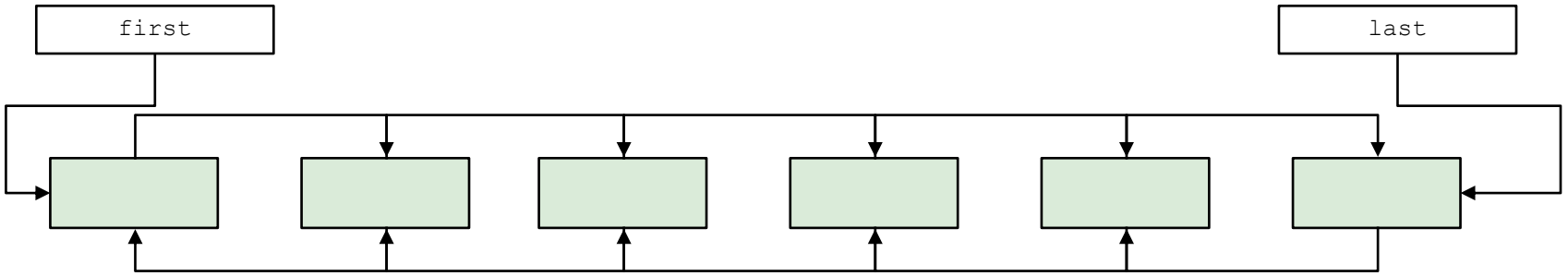
- Listen sind Collections mit einer Ordnungshierarchie
- Auf bestimmte Inhalte innerhalb der Liste kann über einen Index zugegriffen werden

```
List<String> list = new LinkedList<String>();    // füllen mit dice
String fragment = list.get(3);
```

- **`List`** ist ein Interface mit verschiedenen bereits vorhandenen Implementierungen:
  - **`ArrayList`**: das dynamische Pendant zum Array
  - **`Vector`**: siehe ArrayList
  - **`LinkedList`**: Die Elemente sind miteinander verkettet (siehe nächste Folie)
- Sowohl **`ArrayList`** als auch **`Vector`** greifen intern auf Arrays zurück
- Die Unterschiede sind liegen in der Thread-Sicherheit (**`Vector`**) und der Art und Weise wie der interne Array verwaltet wird

LinkedList

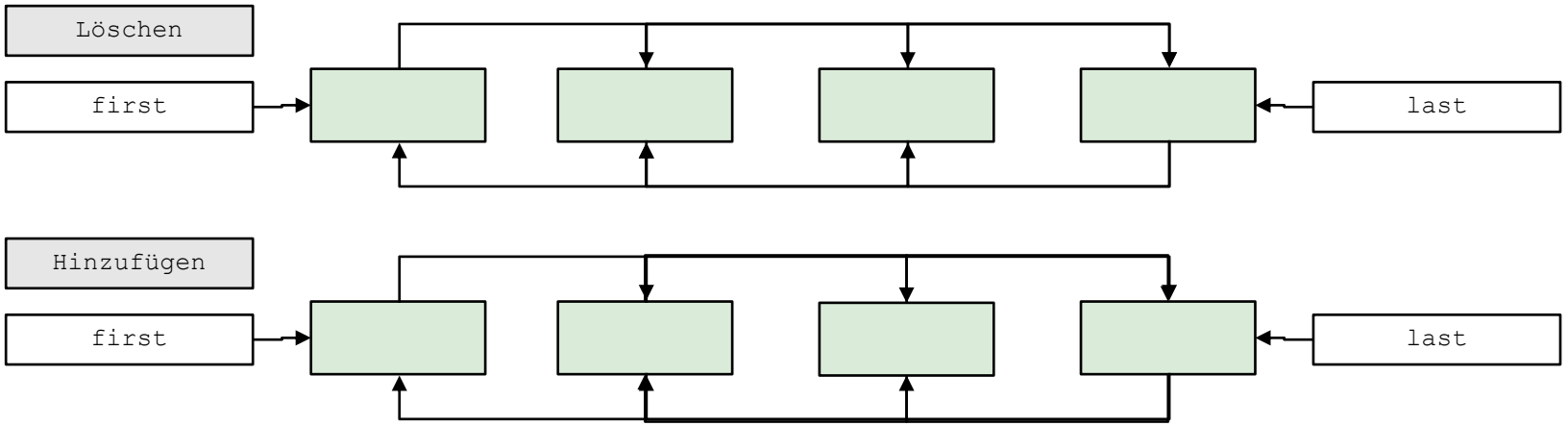
- Schema



- Jedes Element in der Liste wird in einem Node gespeichert
- Jeder Node kennt seinen Vorgänger und seinen Nachfolger
- Der erste und letzte Node sind in der Liste „verlinkt“
- Um ein Element mit einem bestimmten Index zu finden wird diese Kette von Anfang (oder Ende) durchlaufen
- Um also ein Element zu finden benötigt es im schlimmsten Fall n-Suchschritte (n = Anzahl der Elemente)
- **`ArrayList`** und **`Vector`** liefern Elemente über den Index in einem Schritt

LinkedList

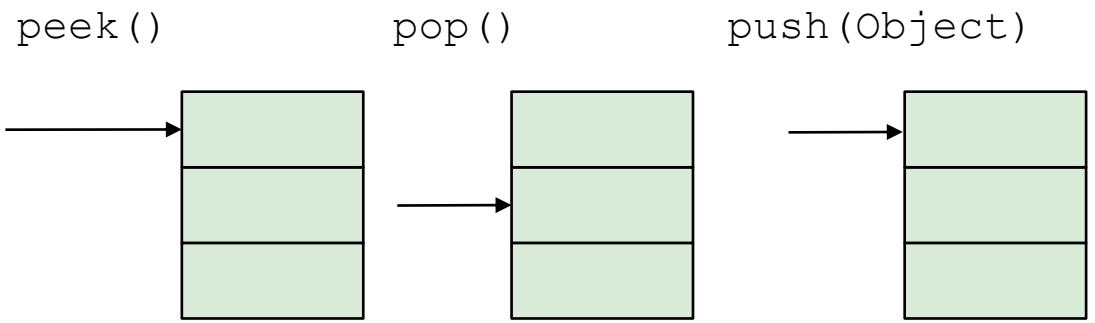
- Indizieren eines Elements in einer Liste langsam
- Das Löschen und Hinzufügen jedoch ist schneller als bei ArrayList/Vector
- Da jeder Node seinen Vorgänger und Nachfolger kennt, lassen sich beide Operationen durch „umbiegen“ der Verlinkung in einem Schritt durchführen
- ArrayList/Node müssen die richtige Stelle zum Löschen finden, den Array unter Umständen kopieren und neufüllen



Stack

- Ein Stack ist bereits im Kapitel „Speichermanagement“ aufgetaucht
- Der Stack-Speicher baute sich Frame für Frame auf einer Seite des Speichers auf
- Mit jedem Methodenende wird der Stack auf genau der Seite wieder abgebaut
- Die Stack-Datenstruktur ist analog aufgebaut
- Auch LIFO (*L*ast-*I*n-*F*irs t-*O*ut) genannt

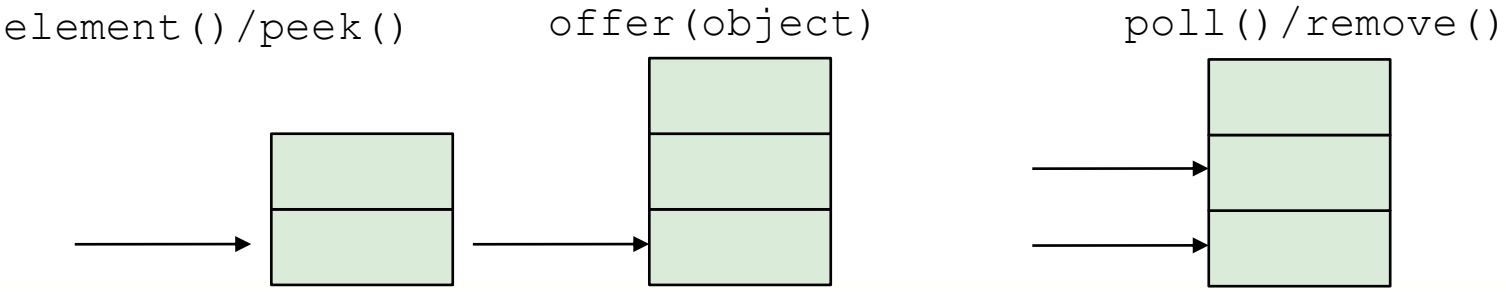
```
boolean empty();
E peek ();
E pop();
E push(E o);
int search(Object o);
```



Queue

- Ähnlich dem Stack
- Eine Queue oder auch Warteschlange genannt wird „hinten“ auf- und „vorne“ abgebaut
- Daher: FIFO (*F*irst-*I*n-*F*irst-*O*ut)

```
E element();
boolean offer(E o);
E peek();
E poll();
E remove();
```

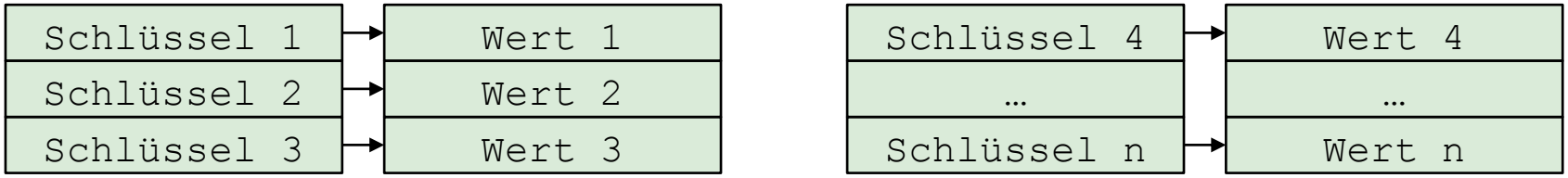


Set

- Wie die Listen enthält der Datentyp eine Menge von Elementen
- Allerdings nicht zwingend „zusammenhängend“
- Zudem gilt: Jedes Element ist in der Menge eindeutig
- Keine doppelte Einträge
- Insb. keine doppelten „null“-Einträge

Map

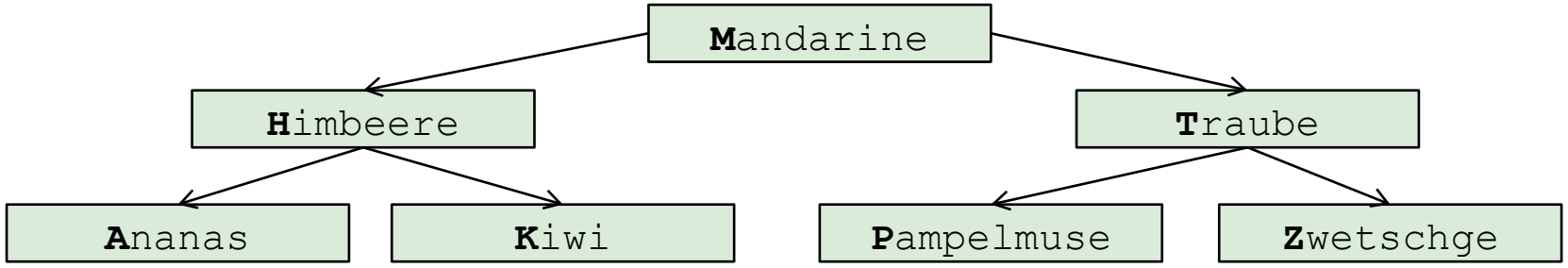
- Der Datentyp Map wird auch als „assoziativer Speicher“ bezeichnet
- Jedes Element in der Map ist eindeutig über einen Schlüssel assoziiert
- Eine Map enthält demnach Schlüssel-Wert Paare



- HashMap:
  - Direkter Zugriff ohne Suche auf Werte über den Schlüssel
  - Schlüssel sind Hashwerte (**`hashCode`**()-Methode)
  - Suche nach einem Schlüssel über den Wert allerdings aufwendig
  - Keine „Rückwärtsassoziation“ zwischen Wert und Schlüssel
  - Unsortiert

Map (2)

- TreeMap:
  - Sortiert nach den Schlüsseln
  - Schlüssel müssen entweder das Interface **`Comparable`** implementieren
  - Oder eine Instanz des Interfaces **`Comparator<T>`**, wobei T dem Datentyp der Schlüssel entsprechen muss, muss die Sortierung übernehmen
  - Einträge sind in einer Baumstruktur angeordnet
  - Zugriff daher nicht „direkt“ wie bei **`HashMap`**, aber immer noch schneller als in einer linear angeordneten Liste



Immutable/Hash/Equals

- Objekte werden als „immutable“ bezeichnet, wenn sie nach ihrer Erzeugung nicht mehr geändert werden können
- Beispiel: **`Strings`**
- Solange ein Objekt sich nicht ändert, muss seine **`hashCode`**-Methode immer den gleichen Wert liefern
- Wenn zwei Objekte unter „equals“ gleich sind, müssen Ihre Hash-Werte übereinstimmen
- Die **`equals`**-Methode wird daher häufig in Abhängigkeit zu Attributwerten definiert
- Hier ist vor allem im Hinblick auf HashMaps und Sets Vorsicht geboten:
  - Da die Eindeutigkeitsforderung auf Basis von **`equals`** überprüft wird, sollten Sets nur unveränderbare Werte enthalten
  - In Hashes werden die Werte über den Hash-Wert eines Schlüssel assoziiert, auch hier sollte der Hash-Wert und damit der Schlüssel nicht veränderbar sein

Iterator

- Zu Anfang wurde erwähnt, das Collections das **`Iterable`**-Interface implementieren
- So sind Collections für die **`foreach`**-Schleife einsetzbar
- Jede Collection bietet mit **`iterator()`** allerdings die Möglichkeit direkt auf einen Iterator zuzugreifen:

```
Iterator<String> iterator = list.iterator();
String string;
while (iterator.hasNext()) {
    string = iterator.next();
    …
    iterator.remove();
}
```

- Vorteil: Die meisten Iterator bieten die Möglichkeit während der Iteration Einträge aus der Collection zu löschen
- In einer **`foreach`**-Schleife nicht möglich: ConcurrentModificationException

Comparator

- Ein Comparator vergleich zwei Elemente gleichen Typs
  - Falls beide Elemente gleich sind (equals) liefert er 0
  - Falls das erste Argument „kleiner“ ist als das zweite eine Zahl < 0
  - Ansonsten > 0
- Es muss dabei immer gelten:

$$sign(compare(x,y)) = -1 * sign(compare(y,x))$$
$$compare(x,y) > 0 \wedge compare(y,z) > 0 \Rightarrow compare(x,z) > 0$$

- Mit Hilfe von Comparatoren und der Hilfs-Klasse Collections können Listen relative einfach sortiert werden:

```
List<String> list = new LinkedList<String>();
dice("dies ist ein string der durcheinander gewürfelt wird", list);
Collections.sort(list, new StringComparator());
```

String-Comparator

```
public static class StringComparator implements Comparator<String> {
    public int compare(String string1, String string2) {
            int len1 = string1.length();
            int len2 = string2.length();
            int lim = Math.min(len1, len2);
            char v1[] = string1.toCharArray();
            char v2[] = string2.toCharArray();
            int k = 0;
            while (k < lim) {
                char c1 = v1[k];
                char c2 = v2[k];
                if (c1 != c2) return c1 - c2;
                k++;
            }
            return len1 - len2;
    }
}
```

Aufgabe 9

- Schreiben Sie eine eigene Implementierung von **`LinkedList`** (Folie 7)
- Nennen sie die Klasse **`MyLinkedList`**
- *Optional*: Legen Sie die Klasse generisch an
- Implementieren Sie zuerst die Methoden
  - **`add(Object(bzw.E) object), remove(Object object)`**
- Die Klasse benötigt dafür eine innere (**`protected/private`**) statische Klasse Namens **`Node`** (siehe nächste Folie) für die Verkettung
- MyLinkedList hat zwei Attribute: **`Node first, Node last`**
- *Tipp*: Zur Ausgabe entweder **`toString()`** überschreiben oder eine eigene Methode entwerfen
- *Tipp*: Wie iteriert man durch die Elemente ohne **`foreach`** oder **`iterator`**?
- Die Klasse soll das **`Collection`**-Interface implementieren (iterator() kann leer implementiert werden „**`return null;`**“)
- *Tipp*: **`implements Collection`** erst hinzufügen, wenn **`add`** und **`remove`** implementiert sind

Aufgabe 9 (Erweitert)

- Erweitern Sie **`MyLinkedList`** zur Liste indem Sie das List-Interface implementieren
- Erweitern Sie Ihre Klasse entweder zu einer Warteschlange oder einem Stack

```
private static class Node<E> {
        E item;
        Node<E> next, prev;

        Node(Node<E> prev, E element, Node<E> next) {
            this.item = element;
            this.next = next;
            this.prev = prev;
        }
    }
```