



# DATENSTRUKTUREN

OCA Konform



ARRAY

# Array

- **Arrays:** rudimentärste Art, mehrere gleichartige Objekte in Java zu speichern
  - *Elemente werden sequenziell hintereinander in den Hauptspeicher geschrieben*
  - *Zugriff auf ein Element durch Angabe der Index*
  - *Index eines Elementes: Position innerhalb des für den Array reservierten Speicherbereichs*
  - *Index beginnt stets mit 0*
  - *Index des letzten Element eines Arrays mit  $n$  Elementen ist stets  $[n-1]$*

# Array

## ■ Deklaration eines Arrays:

- *Datentyp gefolgt von einer geöffneten und einer geschlossenen eckigen Klammer und dem Bezeichner*

```
private Kunde[] kunden;  
...
```

Datentyp der Elemente

Name des Arrays

- *Bei der Deklaration wird die Größe des Arrays nicht angegeben. Es wird daher zu diesem Zeitpunkt noch kein Speicherplatz für den Array reserviert*

# Array

- Bei der **Instanziierung** des Arrays wird Speicherplatz reserviert
  - *Instanziierung erfolgt mit dem Schlüsselwort new*
  - *In den eckigen Klammern ist die gewünschte Kapazität anzugeben*
  - *Bei der Instanziierung ist zu beachten, dass die Elemente mit dem Standardwert des jeweiligen Datentyps vorbelegt werden:*
    - Ein int-Array wird mit lauter Nullen gefüllt
    - Ein boolean-Array mit false-Werten
    - Arrays für komplexe Datentypen (z. B. Strings und eigene Klassen) mit null-Werten

# Array

- Deklaration und Instanziierung eines Arrays

```
public class Kundenverwaltung {  
    private Kunde[] kunden;  
    ...  
    public Kundenverwaltung(){  
        kunden = new Kunde[42];  
  
        System.out.println(kunden[0]);  
        System.out.println(kunden[41]);  
        System.out.println(kunden[42]);  
    }  
}
```

← Deklaration des Arrays

← Instanziierung mit Kapazität 42  
Alle Elemente werden mit Datentyp-spezifischen Standard-Werten belegt

← null (Standard-Wert)

← `ArrayIndexOutOfBoundsException`

Rechteckiges Ausschneiden

# Array

- Für eine andere Vorbelegung (keine Standardwerte) kann die Instanziierung auch mit einer Initialisierung einhergehen
  - *In geschweiften Klammern wird eine Komma-getrennte Liste von Wertausprägungen angegeben*
  - *Durch Angabe der Initialwerte wird implizit die Kapazität des Arrays festgelegt - die Angabe der Kapazität fällt weg.*
  - *Die Initialisierung kann nur zusammen mit der Instanziierung erfolgen und nicht getrennt in einer späteren Anweisung*

# Array

- Instanziierung eines Arrays mit Initialisierung

```
public class Kundenverwaltung {
```

```
    private Kunde[] kunden;
```

```
    ...
```

```
    public Kundenverwaltung(){
```

```
        kunden = new Kunde[] {new Kunde("Ulf", "Koll"),  
                                new Kunde("Ilse", "Stahl"),  
                                new Kunde("Rita", "Kafka")};
```

Kapazität wird  
implizit durch die  
Initialisierung  
vorgegeben

```
        System.out.println(kunden[0]);  
        System.out.println(kunden[1]);  
        System.out.println(kunden[2]);
```

Instanziierung mit  
Initialisierung

Rechteckiges Ausschneiden



# Array

- Nach der Instanziierung kann die Kapazität eines Arrays nicht mehr verändert werden
- Überblick über die Kapazität mit Hilfe des Attribut `length` möglich
  - *Wichtig wenn eine separate Methode, in der die Größe des Arrays üblicherweise unbekannt ist, alle Elemente des Arrays verarbeiten möchte*  
*Siehe nächste Folie*

# Array

- Attribut length am Beispiel der for-Schleife

```
public class Kundenverwaltung {
```

```
    private Kunde[] kunden;
```

```
    ...
```

```
    public void aktualisiereAlleKunden(){
```

```
        for (int index=0; i<kunden.length; index++)
```

```
            if (kunden[index] != null)
```

```
                kundenSpeicher.aktualisieren(kunden[index]);
```

```
        }
```

```
    }
```

Schleife stoppt, sobald i kein gültiger Index mehr ist, d. h.  $i == kunden.length$

Achtung: Prüfen, ob sich an der Index-Stelle tatsächlich ein Element befindet

pro Schleifendurchlauf wird ein Kunde aktualisiert

# Array

- In Java ist es möglich, Arrays zu verschachteln:
  - *Die Elemente eines Arrays sind dann ebenfalls Arrays*
  - *Man spricht dann von mehrdimensionalen Arrays, da sich die Größe des Arrays bildlich gesehen nicht nur in eine Dimension ausdehnt, sondern in mindestens zwei*
  - *Möglicher Anwendungsfall: ein Schachbrett-Array, das zu jeder Zeile jeweils ein Array mit den dazugehörigen Spielfeldern enthält*

# Array

- „Normale“ und mehrdimensionale Arrays

```
int[] vektor = new int[] {2, 4, 1};    ← „normaler“, 1-dimensionaler Array
```

```
int[][] matrix = new int[][] {{7, 3, 2},  
                               {9, 2, 7},  
                               {0, 3, 3},  
                               {1, 0, 6}}; ← verschachtelter, 2-dimensionaler Array
```

2 Klammer-Paare  
=> 2 Dimensionen

# Array

## ■ Vorteile:

- *Deklaration und Verwendung unmittelbarer Bestandteil der Java-Syntax*
- *Daher nicht nötig Bibliotheken zu importieren*
- *Arrays können beliebige Typen enthalten: primitiven Datentypen, Strings und auch selbst programmierte Klassen*

## ■ Nachteile:

- *Bei Arrays muss man sich selbst um die Kapazität kümmern - im Gegensatz zu Collections*
- *Array voll:*
  - *Es muss es zur Laufzeit mit einer größeren Kapazität neu initialisiert werden und alle Elemente müssen übertragen werden*
- *zu hohe Kapazität und folglich unnötigerweise ein viel zu großer Speicherbereich ebenfalls möglich*

# Array

## ■ Nachteile:

- *Lücken in sortierten Arrays zu schließen ist mit großen Anstrengungen verbunden*
  - fürs Aufrücken muss jedes Folgeelement bewegt werden
- *Arrays haben eine begrenzte eingebaute Funktionalität:*
  - zusätzlicher Programmieraufwand für die Form eines Stapels, einer Warteschlange oder einer Menge

COLLECTIONS



# Collections

- Das **Collections - Framework** ist eine Menge an häufig benötigten Datenstrukturen und dazu passenden Such- und Sortieralgorithmen
- Die Gemeinsamkeiten der Collections befinden sich im Interface Collection
- Bis auf Datenstrukturen zur Realisierung von Mengen wird dieses Interface (oder daraus abgeleitete Interfaces) von allen Klassen des Collections- Frameworks implementiert
- Für den OCA nur der Vollständigkeit halber angegeben. Eher bestandteil des OCP




# Collections

- **Hauptaufgaben:**
  - *Daten effizient zu speichern*
  - *Effizienten Zugriff auf die Daten zu ermöglichen*
- Beides sind konkurrierende Ziele, daher:
  - *Wahl zwischen verschiedene Implementierungen, entweder:*
    - sparsame Speicherung oder
    - schneller Zugriff begünstigen

# Collections

## ■ Die wichtigsten Methoden der Schnittstelle Collection:

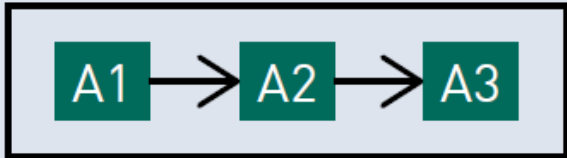
Platzhalter für den Typ der gespeicherten Objekte.  
Wird erst zur Laufzeit bzw. bei der Deklaration definiert („Generics“)



```
public interface Collection<E> extends Iterable<E> {  
    boolean add(E e);           fügt Objekt hinzu (Typ erst zur Laufzeit bekannt, s. o.)  
    boolean remove(Object o);   entfernt ein Objekt  
    int size();                 liefert Anzahl der Objekte zurück (nicht die Kapazität!)  
    boolean isEmpty();          prüft, ob irgendwelche Objekte enthalten sind  
    boolean contains(Object o); prüft, ob ein bestimmtes Objekt enthalten ist  
    void clear();               entfernt alle Objekte der Collection  
    Iterator<E> iterator();     liefert den Iterator zurück (s. nächste Lektion)  
    Object[] toArray();         liefert die Collection als einfachen Array zurück  
}
```


# Collections

## ■ Die wichtigsten Collections:

| Programmieraufgabe  | Besondere Eigenschaften  |
|---|--|
| Artikel im Warenkorb verwalten  | Elemente in sequenziell geordneter Reihenfolge   |
| Skizze  | Interfaces/Klassen im Collections-Framework  |
|  | <p>Interface <code>java.util.List</code><br/>Beispiele für Implementierungen:</p> <ul style="list-style-type: none"><li>• <code>java.util.ArrayList</code> (als Array realisiert)</li><li>• <code>java.util.LinkedList</code> (als Verkettung von Referenzdatentypen realisiert)</li></ul> |

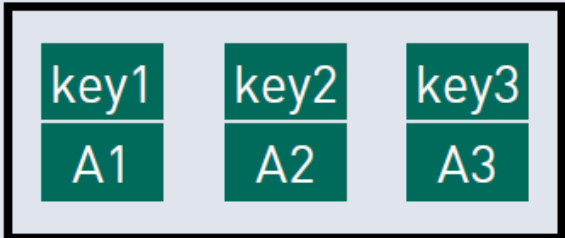
# Collections

## ■ Die wichtigsten Collections:

| Programmieraufgabe   | Besondere Eigenschaften   |
|--|---|
| Verwaltung des Shop-Sortiments   | Keine doppelten Elemente; Reihenfolge egal  |
| Skizze   | Interfaces/Klassen im Collections-Framework   |
|  | Interface <code>java.util.Set</code><br>Beispiele für Implementierungen: <ul style="list-style-type: none"><li>• <code>java.util.TreeSet</code> (als Baum realisiert)</li><li>• <code>java.util.HashSet</code> (als Hash-Tabelle)</li></ul> |

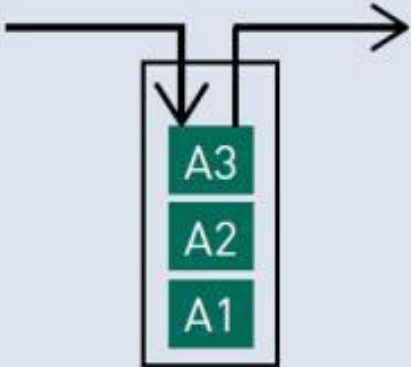
# Collections

## ■ Die wichtigsten Collections:

| Programmieraufgabe   | Besondere Eigenschaften   |
|--|---|
| Kundenverwaltung   | Schneller Zugriff anhand der Kundennummer   |
| Skizze   | Interfaces/Klassen im Collections-Framework   |
|  | <p>Interface <code>java.util.Map</code><br/>Beispiele für Implementierungen:</p> <ul style="list-style-type: none"><li>• <code>java.util.TreeMap</code> (als Baum realisiert)</li><li>• <code>java.util.HashMap</code> (als Hash-Tabelle)</li><li>• <code>java.util.LinkedHashMap</code> (Kombination aus Hash-Tabelle und verketteter Liste)</li></ul> |

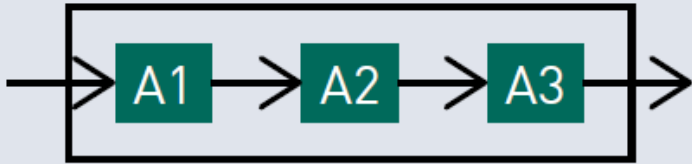
# Collections

## ■ Die wichtigsten Collections:

| Programmieraufgabe   | Besondere Eigenschaften   |
|--|---|
| „Undo“-Funktion im Bestellprozess  | Die Bestellschritte des Benutzers sollen rückgängig gemacht werden können („Last in, First out“)  |
| Skizze   | Interfaces/Klassen im Collections-Framework   |
|  | <p>Interface <code>java.util.Deque</code><br/>Beispielimplementierung:</p> <ul style="list-style-type: none"><li>• <code>java.util.ArrayDeque</code> (als Array realisiert)</li></ul> <p>Interface <code>java.util.List</code></p> <ul style="list-style-type: none"><li>• <code>java.util.Stack</code></li></ul> |

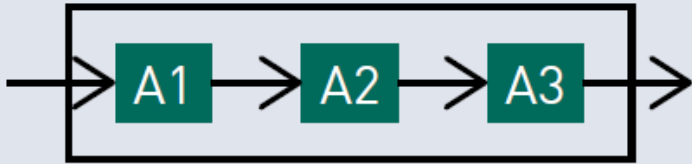
# Collections

## ■ Die wichtigsten Collections:

| Programmiernaufgabe   | Besondere Eigenschaften  |
|---|--|
| Warteschlange für Bestellungen  | Bearbeitung in der Reihenfolge des Eingangs („First in, First out“)  |
| Skizze  | Interfaces/Klassen im Collections-Framework  |
|  | Interface <code>java.util.Queue</code><br>Beispielimplementierung: <ul style="list-style-type: none"><li>• <code>java.util.LinkedList</code> (s. o.)</li></ul> |

# Collections

## ■ Die wichtigsten Collections:

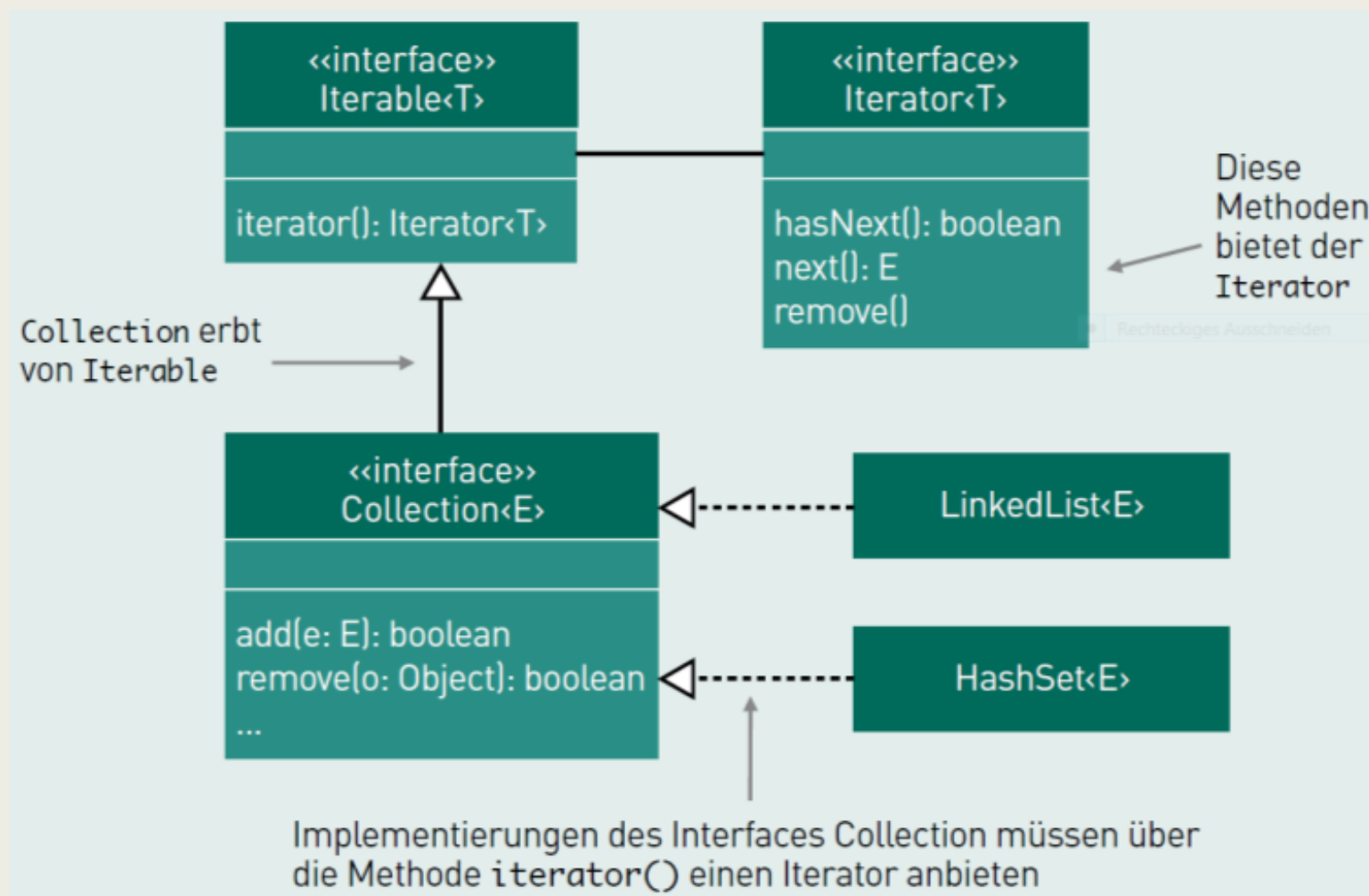
| Programmieraufgabe  | Besondere Eigenschaften  |
|---|--|
| Warteschlange für Bestellungen  | Bearbeitung in der Reihenfolge des Eingangs („First in, First out“)  |
| Skizze  | Interfaces/Klassen im Collections-Framework  |
|  | Interface <code>java.util.Queue</code><br>Beispielimplementierung: <ul style="list-style-type: none"><li>• <code>java.util.LinkedList</code> (s. o.)</li></ul> |



# Iterator

- **Iterator:** Sein Zweck ist es, über alle Elemente in einer beliebigen Collection zu iterieren.
  - *Iterieren bedeutet, alle Elemente, z. B. innerhalb einer Programmschleife, zu durchlaufen und bei Bedarf zu verarbeiten*
  - *Im Collections-Framework hat jede Collection eine eigene Iterator-Implementierung*
    - Sicherstellung mit dem Interface Iterable
    - besteht aus der Methode iterator() und wird vom Interface Collection erweitert

# Collections



# Collections

- Der Iterator besitzt drei Methoden:
- `hasNext()`: prüft, ob es an der aktuellen Position des Iterators noch ein neues Element gibt
- Übergabe mit der Methode `next()`. Der Iterator positioniert sich anschließend selbstständig auf das nächste Element
- `remove()` kann das aktuelle Element, auf das sich der Iterator per `next()` zuvor positioniert hat, aus der Collection entfernt werden

# Collections

## ■ Bereinigung von Datensätzen mit Hilfe eines Iterators:

Die Methode funktioniert mit jeder Collection, die Kunden enthält

```
public void bereinigeAlleKunden(Collection<Kunde> c) {
```

```
    Kunde k = null;
```

```
    for (Iterator<Kunde> it = c.iterator(); it.hasNext(); k = it.next())
```

```
        if (!k.getGeschlecht().equals(""))
```

```
            it.remove();
```

```
}
```

Aufruf der Iterator-Methode zum Löschen des aktuellen Elements

Der Iterator ersetzt die Laufvariable

anstelle von i++ (o. ä.) tritt it.next()

Schleifen-Bedingung erfüllt, solange noch Elemente folgen

# Collections

- Die Methode erwartet als Parameter:
  - *beliebige Collection – d.h. alle Klassen, die das Interface Collection implementieren.*
  - *Einschränkung: Elemente der Collection Objekte der Klasse „Kunde“ oder einer ihrer Unterklassen*
- Methode besteht aus einer for-Schleife:
  - *anstelle einer Laufvariablen verwendet den Iterator*
  - *Bei der Initialisierung wird keine Laufvariable deklariert. Es wird eine Referenz auf den Iterator der Collection erzeugt.*

# Collections

- Abbruchbedingung erfüllt, wenn:
  - *Iterator findet keine weiteren Elemente, d.h. `it.hasNext()` wird zu `false` ausgewertet wird*
  - *Die Angabe einer Schrittweite (z. B. `i++`) ist bei einem Iterator überflüssig*
  - *Der Iterator auf den nächsten Kunden positioniert. Für die Weiterverarbeitung  
Um das Objekt im Rumpf der Schleife komfortabel weiterverarbeiten zu  
können, wird eine Referenz zwischengespeichert*

# Collections

- Nützliches Werkzeug für das Arbeiten mit Collections ist die **erweiterte for-Schleife**:
  - *vereinfacht Durchlaufen von beliebigen Klassen, die das Interface Iterable implementieren (auch Collections)*
- Angaben im Kopf der Schleife:
  - *Deklaration eines Stellvertreters für jedes Collection - Element und die zu verarbeitende Collection selbst*
  - *Argumente werden mit einem Doppelpunkt getrennt*

# Collections

- **Erweiterte for-Schleife** als Ersatz für den Iterator

Stellvertreter-  
Objekt für alle  
Elemente

```
public void bereinigeAlleKunden(Collection<Kunde> c) {  
    for (Kunde k : c)  
        if (!k.getGeschlecht().equals(""))  
            c.remove(k);  
}
```

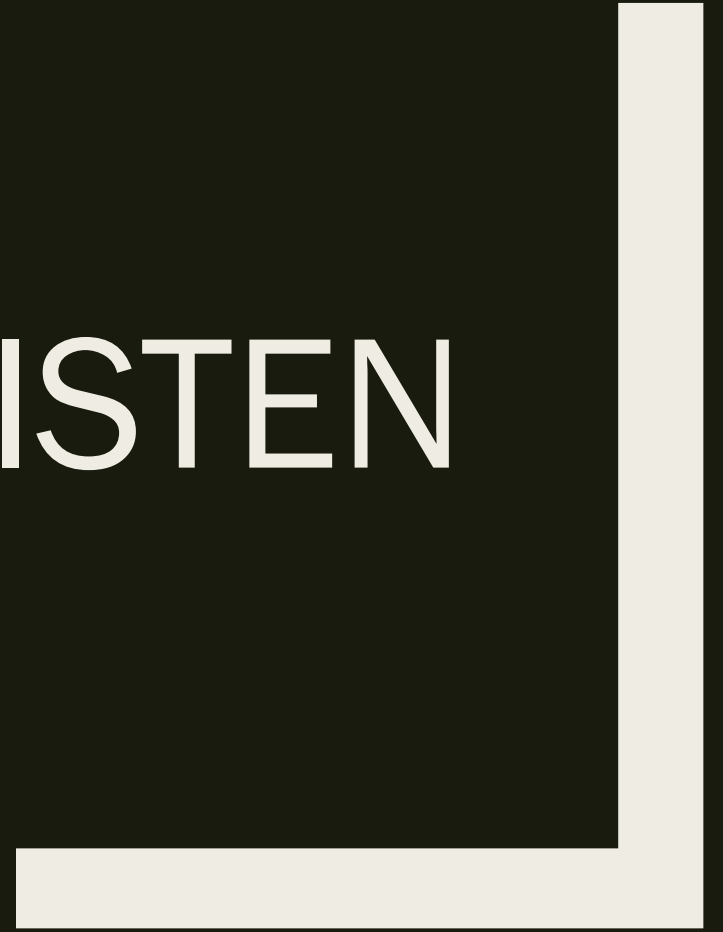
Collection, die durchlaufen werden soll

Rechteckiges Ausschneiden

Anweisungen im Rumpf der Schleife werden für jedes  
einzelne Element (repräsentiert durch k) ausgeführt

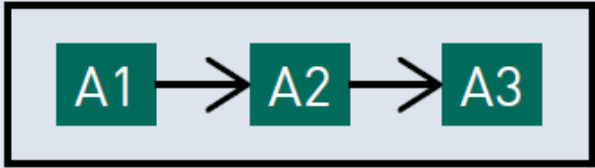


LISTEN



# Listen

- **Liste:** Speichern beliebig vieler Elemente in sequenziell geordneter Reihenfolge

| Skizze  | Interfaces/Klassen im Collections-Framework  |
|---|--|
|  | <p>Interface <code>java.util.List</code><br/>Beispiele für Implementierungen:</p> <ul style="list-style-type: none"><li>• <code>java.util.ArrayList</code> (als Array realisiert)</li><li>• <code>java.util.LinkedList</code> (als Verkettung von Referenzdatentypen realisiert)</li></ul> |

# Listen

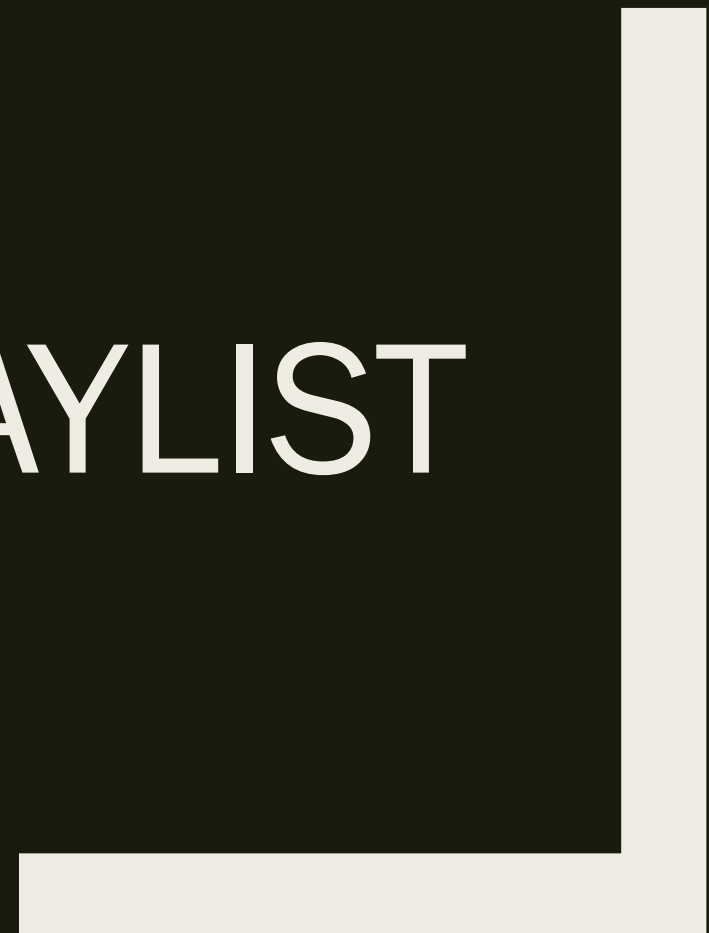
Zusätzliche Methoden - außer der Collections-Interfaces

- `void add(int index, E element)`
  - *Fügt Element vom Typ E an Stelle index ein und verschiebt Rest nach Rechts*
- `boolean addAll(int index, Collection<? extends E> c)`
  - *Fügt ganze Collection an der Stelle index ein*
  - *Elemente der Collection können beliebigen Typs sein*
  - *Typ sollte zumindest in einer Vererbungsbeziehung mit dem Element-Typ der Ziel-Collection (E) stehen*
  - *Liefert true zurück, falls die Operation erfolgreich war*

# Listen

- `E remove(int index)`
  - *Löscht Objekt an der Stelle index*
  - *Gibt Referenz auf das gelöschte Element zurück.*
- `set(int index, E element)`
  - *Tauscht übergebenes Element mit Element an Position index*
- `subList(int fromIndex, int toIndex)`
  - *Liefert Ausschnitt (von fromIndex bis exklusive toIndex)*
- **Achtung:** Liste ist Referenzdatentyp
  - *Veränderungen an der Teil-Liste wirken sich aufs Original aus*

# ARRAYLIST



# ArrayList

- **ArrayList** speichert Elemente intern in einem Array
  - *Zugriff auf Elemente äußerst schnell*
  - *Änderungen am ArrayList aufwendig (nachfolgende Listenelemente müssen ebenfalls verschoben werden)*
  - *Ausnahme: Element wird am Ende der Liste eingefügt bzw. gelöscht*
  - *Potentielle Probleme mit Größe des internen Arrays, siehe Nachteile für Arrays*
  - *ArrayList – komfortabler als herkömmliches Array*

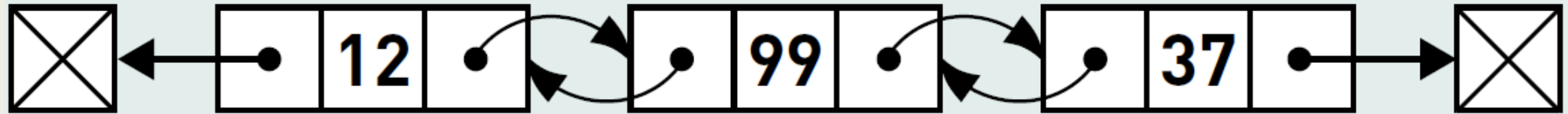
# LINKEDLIST

Nur für die Vollständigkeit



# LinkedList

- **LinkedList** ist als doppelt verkettete Liste realisiert
  - *Liste besteht aus Objekten, die eine Referenz auf ihren Vorgänger und ihren Nachfolger halten*
- Interne Speicherstruktur einer doppelt verketteten Liste



- LinkedList arbeitet ohne einen Index



# LinkedList

- Zugriff auf beliebiges Element:
  - *Im schlimmsten Fall muss ganze Liste durchlaufen werden, um Referenz zu erhalten*
    - Zugriff bei verketteter Liste langsamer, als bei der ArrayList
  - *Deutlich schneller, ein Element an beliebiger Stelle einzufügen oder zu löschen (lediglich Referenzen der Vorgänger und Nachfolger müssen angepasst werden)*

# LinkedList

- Bp. für Methoden der Schnittstellen Collection und List

```
import java.util.LinkedList;  
import java.util.List;
```

← Die benötigten Klassen aus dem Paket  
java.util müssen importiert werden

```
public class Warenkorb {
```

Hier wird die Implementierung einmalig definiert

```
    private float artikelSumme;  
    private List<Artikel> artikelliste = new LinkedList<Artikel>();  
    ...
```

```
    public boolean artikelHinzufuegen(int position, Artikel a){  
        try {  
            artikelliste.add(position, a);  
            artikelSumme += a.getPreis();  
        } catch (IndexOutOfBoundsException ex) {  
            return false; // Position ist ungültig  
        }  
        return true;  
    }  
}
```

← Einsatzbeispiel für  
Methoden der Schnittstelle  
java.util.List

# LinkedList

- Bp. für Methoden der Schnittstellen Collection und List


```
public boolean artikelHinzufuegen(Artikel a){  
    boolean erfolgreich = artikelliste.add(a);  
    if (erfolgreich)  
        artikelSumme += a.getPreis();  
    return erfolgreich;  
}
```

```
public boolean artikelEntfernen(Artikel a){  
    boolean erfolgreich = artikelliste.remove(a);  
    if (erfolgreich)  
        artikelSumme -= a.getPreis();  
    return erfolgreich;  
}
```

```
public void leereWarenkorb(){  
    artikelliste.clear();  
    artikelSumme = 0;  
}
```

```
public int getAnzahlArtikel(){  
    return artikelliste.size();  
}
```

Einsatzbeispiele für die  
Methoden der Schnittstelle  
java.util.Collection

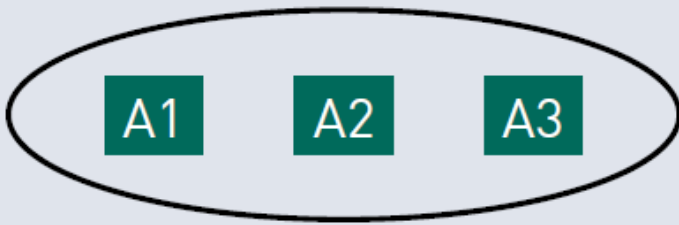


# SETS

Nur für die Vollständigkeit

# Mengen-Datenstrukturen

- Verwaltung zusammengehöriger Elementen.
- Reihenfolge ist unerheblich
- Keine doppelten Elemente

| Skizze   | Interfaces/Klassen im Collections-Framework   |
|--|---|
|  | <p>Interface <code>java.util.Set</code><br/>Beispiele für Implementierungen:</p> <ul style="list-style-type: none"><li>• <code>java.util.TreeSet</code> (als Baum realisiert)</li><li>• <code>java.util.HashSet</code> (als Hash-Tabelle)</li></ul> |

# Mengen-Datenstrukturen

- Implementierungen dieses Interfaces stellen sicher, dass kein Element doppelt vorhanden ist
- Es wird sichergestellt, dass folgende Bedingung nie für ein beliebiges Element-Paar  $x$  und  $y$  gilt:  $x.equals(y) == true$
- Nur Methoden des Collections-Interfaces

# TreeSet

- verwaltet Elemente der Menge intern in einer Baum-Struktur
- Operationen wie add, remove, contains können selbst bei sehr großen Datenmengen noch mit vertretbarem Aufwand durchgeführt werden

# HashSet

- verwaltet Menge mithilfe einer Hash-Tabelle
- konstanter Aufwand bei Operationen sichergestellt (vorausgesetzt die Hash-Funktion besitzt eine gute Streuung)
- Geschwindigkeit beim lesenden Zugriff auf die Elemente steht einem etwas höheren Aufwand beim schreibenden Zugriff gegenüber, da die Kosten zur Berechnung der Hash-Funktion berücksichtigt werden müssen.



# Weitere Collections

- Neben den Listen und Sets gibt es noch weitere Collections.
- Queue
  - *Das Queue Interface stellt eine Datenstruktur nach dem FIFO prinzip dar.*
    - First in, First out. Bsp.: Eine Warteschlange beim Kiosk
  - *Deque bietet Operationen an, womit an beiden Enden der Queue gearbeitet werden kann.*
  - *Verwandt mit der Queue ist der Stack, dieser arbeitet nach dem LIFO prinzip.*
    - Last in, First out.

| Methoden der Queue | Mit Exception | Ohne Exception |
|--------------------|---------------|----------------|
| Einfügen           | add(...)      | offer(...)     |
| Abfragen           | element()     | peek()         |
| Löschen            | remove()      | poll()         |

# Weitere Collections

- Map – Oder auch Assoziativ Speicher
  - *Maps arbeiten mit Key, Value paaren.*
    - Der Key stellt die Assoziation zum abfragen des assoziierten Wertes dar.
      - *Der Key muss eineindeutig sein.*
      - *Die Schlüsselobjekte müssen `>>hashbar<<` sein, also `equals(...)` und `hashCode()` konkret implementieren. Eine besondere Schnittstelle für die Elemente ist nicht nötig.*
    - Die Values können beliebig und auch mehrfach vorhanden sein.