

# Collections

## SS 2019

# Listen

- ▶ Eine Liste ist eine
  - ▶ zusammenhängende Folge
  - ▶ einer variablen Anzahl
  - ▶ beliebig im Speicher abgelegter Datenelemente
    - ▶ Die Realisierung erfolgt über dyn. Speicherzuweisung für jedes Listenelement
  - ▶ auf Basis von Objekten
- ▶ Jedes Element enthält Referenz auf Vorgänger und Nachfolger
- ▶ Listenelemente werden zur Laufzeit des Programms definiert
- ▶ Die Größe der Liste kann zur Laufzeit steigen und sinken
- ▶ eine Größenänderung während des Programmlaufs ist einfach möglich

# Verkettete Listen

## Übliche Operationen auf Listen

- ▶ Hinzufügen von Elementen am Anfang/Ende
- ▶ Entfernen eines Elements am Anfang/Ende
- ▶ Zugriff auf Elemente der Liste (Anfang/Ende/Position)
- ▶ Berechnen der Länge der Liste
- ▶ Prüfen auf leere Liste
- ▶ Listendurchlauf (und ggf. Ausgabe/Suche)

# Verkettete Listen - Methoden

Die Java-Klassenbibliothek **java.util.LinkedList**

<http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html>

stellt verschiedene Methoden zur Verfügung (Auswahl):

- ▶ `LinkedList()` -> erzeugt eine neue Liste
- ▶ `add()` / `addLast()` -> fügt ein Listenelement am Ende ein
- ▶ `addFirst()` -> fügt ein Element am Anfang ein
- ▶ `clear()` -> entfernt alle Listenelemente
- ▶ `get()` / `getFirst()` / `getLast()` -> liefert die Werte der Elemente
- ▶ `indexOf()` -> liefert erste Position eines Wertes
- ▶ `lastIndexOf()` -> liefert letztes Auftreten eines Wertes
- ▶ `size()` -> liefert die Anzahl Elemente

# Verkettete Liste - Java

Erzeugen einer leeren Liste:

```
// LinkedList wird erzeugt  
LinkedList list = new LinkedList();
```

Hinzufügen von Elementen:

```
list.addFirst(10);  
list.addFirst(12);  
list.addLast(23);
```

# Durchlaufen einer Liste mittels Iteratoren

Die Klassen für Listen in der Java-Bibliothek erlauben den Durchlauf von Listen mittels sogenannter **Iteratoren**.

Ein Iterator ist ein Objekt, von dem man sich die Elemente der Liste eines nach dem anderen zurückgeben lassen kann.

Die Methode **iterator()** erzeugt einen Iterator für eine Liste.

Der erste Aufruf der Methode **next()** auf dem Iterator liefert das erste Element der Liste zurück, der zweite Aufruf liefert das zweite Element, usw.

Mit der Methode **hasNext()** stellt der Iterator fest, ob noch weitere Elemente kommen.

```
Iterator<Integer> it = list.iterator();
```

# Iteratoren - Beispiel

```
public static void main(String[] args) {

    LinkedList<Integer> list = new LinkedList();

    list.addFirst(new Integer(3));    // <3>
    list.addLast(new Integer(12));    // <3,12>
    list.addFirst(new Integer(72));   // <72,3,12>

    Iterator<Integer> it = list.iterator();
    while (it.hasNext()) {
        Integer k = it.next();
        System.out.println(k.intValue());
    }
}

/* Kurzform:
*
*   for (Integer k : list) {
*       System.out.println(k.intValue());
*   }
*/
```

# Beispiel – verkettete Listen

## ► Rumpfprogramm

```
1      package listenoperationen;
2
3      import java.util.LinkedList;
4      import java.util.Iterator;
5
6      public class Listenoperationen
7      {
8          public static void main(String[] args)
9          { ... }
```

## ► 2 leere Listen erzeugen

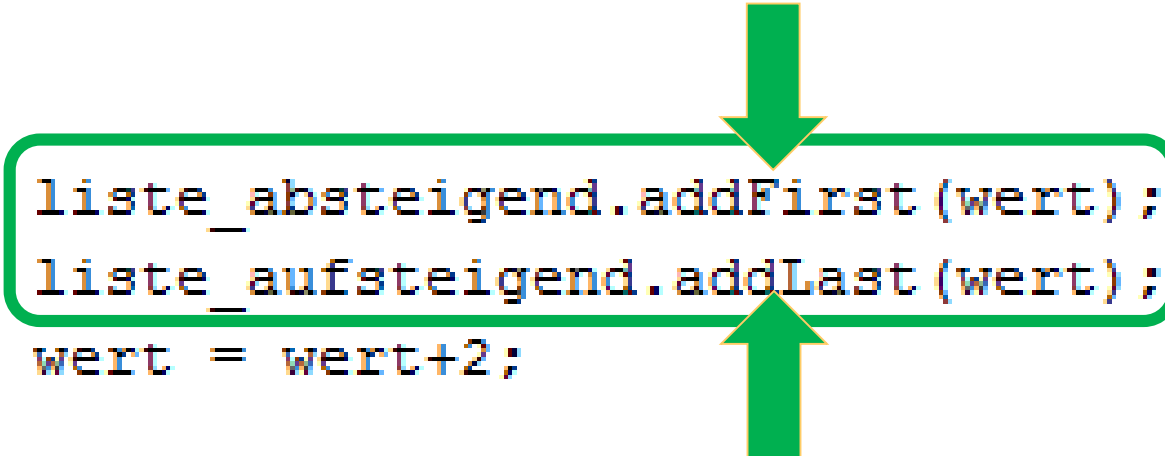
```
LinkedList liste_aufsteigend = new LinkedList();
LinkedList liste_absteigend = new LinkedList();
```



# Beispiel – verkettete Liste

- ▶ Elemente in die Liste eintragen  
(alle ungeraden Zahlen zwischen 1 und 21)

```
int wert = 1;
do
{
    liste_absteigend.addFirst(wert);
    liste_aufsteigend.addLast(wert);
    wert = wert+2;
}
while (wert<=21);
```



# Beispiel – verkettete Liste

## ▶ Listen komplett ausgeben

```
Iterator position1 = liste_aufsteigend.iterator();  
Iterator position2 = liste_absteigend.iterator();
```

```
System.out.println("Werte 1. Liste\tWerte 2. Liste");
```

```
int inhalt;
```

```
while(position1.hasNext() || position2.hasNext())
```

```
{
```

```
    inhalt = (int) position1.next();
```





```
    System.out.print(inhalt);
```

```
    inhalt = (int) position2.next();
```

```
    System.out.printf("\t\t");
```

```
    System.out.println(inhalt);
```

```
}
```



Werte 1. Liste	Werte 2. Liste
1	21
3	19
5	17
7	15
9	13
11	11
13	9
15	7
17	5
19	3
21	1



# Beispiel – verkettete Liste

- Wert an einer bestimmten Position ausgeben

```
int index = 3;  
System.out.printf("Der Wert an Position %d ist: ", index);  
System.out.println(liste_aufsteigend.get(index));
```




Ausgabe: Der Wert an Position 3 ist: 7

# Beispiel – verkettete Liste

- ▶ Wert an bestimmter Position in der Liste einfügen

`Liste_aufsteigend.add(10, 9999);`



```
1  
3  
5  
7  
9  
11  
13  
15  
17  
19  
9999  
21  
BUILD SUCCESSFUL
```

# Beispiel – verkettete Liste

- ▶ Bestimmen Wert in der Liste suchen (erstmaliges und letztmaliges Auftreten)

```
//Bestimmen Wert in der Liste suchen (1. Auftreten)
int wo = liste_aufsteigend.indexOf(13);
System.out.printf("\nDer gesuchte Wert steht erstmals an Position %d", wo);
//Bestimmen Wert in der Liste suchen (letztes Auftreten)
liste_aufsteigend.add(9,13);
wo = liste_aufsteigend.lastIndexOf(13);
System.out.printf("\nDer gesuchte Wert steht letztmalig an Position %d", wo);
```

```
Der gesuchte Wert steht erstmals an Position 6
Der gesuchte Wert steht letztmalig an Position 9
```

# LinkedList

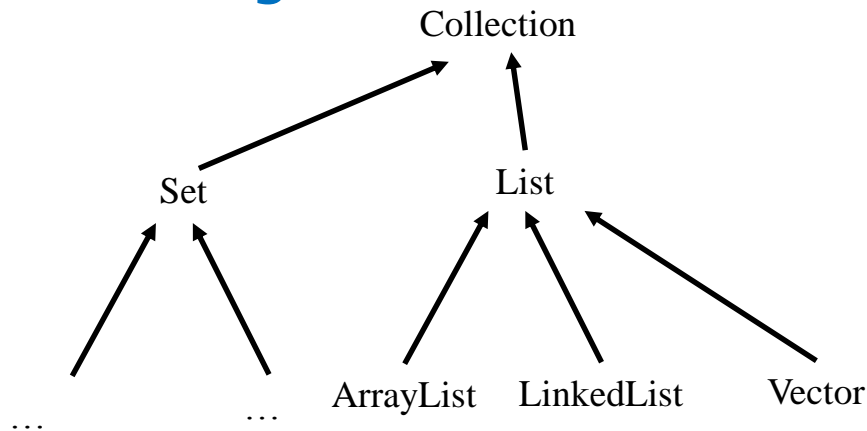
- ▶ volle Flexibilität,
- ▶ beliebiges Einfügen, Umsortieren, Entfernen,
- ▶ langsamer Zugriff, besonders auf ganze Folgen von Elementen

# ArrayList

- ▶ Benutzung analog zu LinkedList, vgl. API
- ▶ Anzahl der Elemente muss bei Deklaration nicht bekannt sein
- ▶ Hat Methoden um die Länge zu handhaben (`capacity` kann verändert werden)
- ▶ Anhängen und Löschen (hinten)
- ▶ aufwändig: Einfügen an beliebiger Stelle, Umsortieren.

# LinkedList und ArrayList

- ▶ sind Beispiele für generische Datentypen (Generics), da man damit Listen beliebiger Typen generieren kann.
  - ▶ Jedes Element muss jedoch vom selben Typ sein
- ▶ Sind Teil des sogenannten Collection-Frameworks





# Collection Framework

- ▶ **Helferklasse:** `java.util.Collections`
  - ▶ Hilft u.a. beim Sortieren
  - ▶ `Collections.sort(liste);`
  - ▶ Die `sort()`-Methode
    - ▶ erwartet eine Liste als Eingabe
    - ▶ sortiert alle Elemente in dieser Liste anhand ihrer `compareTo()`-Methoden.
      - ▶ Diese Methode muss für alle Elemente implementiert sein
      - ▶ Listenelementtyp implementiert das Comparable-Interface.
      - ▶ die Elemente werden sortiert, wie es durch die `compareTo()`-Methode vorgegeben wird

# Sortieren in einer ArrayList – Beispiel

```
package model;  
import java.util.ArrayList;  
import java.util.Collections;
```

```
public class Manager {
```

```
    private ArrayList<Group> grouplist;  
    private ... ;
```

```
    public manager() {  
        grouplist = new ArrayList<>();  
        ...  
    }
```

```
    public void addGroup(Group group) {  
        grouplist.add(group);  
        Collections.sort(grouplist);  
    }
```

```
}
```

Die Liste aller  
(Produkt-)Gruppen

Konstruktor,  
erzeugt neues  
Manager-Objekt.

Fügt eine Gruppe hinzu und  
sortiert die Gruppenliste.  
Sortierung anhand compareTo-  
Methode von Group.

# Sortieren in einer ArrayList – Beispiel Klasse Group

```
public class Group implements Comparable<Group> {
```

```
    private String title;
```

```
    private ArrayList<Product> products;
```

```
    public Group(String title) {
```

```
        this.title = title;
```

```
        products = new ArrayList<Product>();
```

```
    }
```

```
    public void addProduct(Product product) {
```

```
        products.add(product);
```

```
        Collections.sort(products);
```

```
    }
```

```
    @Override
```

```
    public int compareTo(Group group) {
```

```
        return this.title.compareTo(group.title);
```

```
    }...
```

Die eindeutige  
Bezeichnung der  
Gruppe

Die Liste von  
Produkten dieser  
Gruppe

Konstruktor  
erzeugt  
eine Gruppe  
mit der  
übergeb.  
Bezeichnung

Fügt ein Produkt zu  
dieser Gruppe hinzu  
und sortiert die  
Produktliste der  
Gruppe. Sortierung  
anhand compareTo-  
Methode der Klasse  
Product

Vergleichsfunktion  
zweier Gruppen anhand  
der Bezeichnung für die  
Sortierung der Gruppen.

# ArrayList - Beispiel Klasse Group

```
public class Group implements Comparable<Group> {  
    private String title;  
    private ArrayList<Product> products;  
    public Group(String title) { ... }  
    public void addProduct(Product product) { ... }  
    @Override  
    public int compareTo(Group group) { ... }  
  
    public ArrayList<Product> getProducts() {  
        return products;  
    }  
    public String getTitle() {  
        return title;  
    }  
    public boolean isEmpty() {  
        return products.isEmpty();  
    }  
    public boolean removeProduct(Product product) {  
        return product.remove(product);  
    }  
}
```

Gibt die Liste von Produkten dieser Gruppe zurück.

Gibt zurück, ob die Gruppe Produkte enthält oder nicht.

Entfernt ein Produkt aus der Liste von Produkten in dieser Gruppe.

# Weitere nützliche Klasse: Hashtable

- ▶ **Objekte der Klasse** `java.util.Hashtable`
  - ▶ ermöglichen die Speicherung von Datenpaaren aus einem Schlüssel und einem zugeordneten Wert
  - ▶ und den effizienten Zugriff auf den Wert über den Schlüssel
- ▶ **Struktur entspricht einer key-value-Tabelle**
- ▶ **Die Klassen der Objekte, die in die Tabelle als Schlüssel eingetragen werden sollen, implementieren die Methoden** `equals` **und** `hashCode`
  - ▶ Mit Hilfe von `equals` erfolgt intern der Eintrag und der Zugriff auf Schlüssel
  - ▶ `hashCode` (von `Object`) liefert einen ganzzahligen Wert, den so genannten Hashcode, der für die Speicherung von Objekten in Hashtabellen gebraucht wird.