

- Bisher: Zwei Möglichkeiten zur Verwaltung von vielen Daten gleichen Typs:
  - Arrays (zur Verfügung gestellt von Java)
  - Verkettete Listen (selbst implementiert)
- Die Java-Klassenbibliothek beinhaltet sehr viele solcher Datenstrukturen
  - Unter anderem: doppelt verkettete Liste
- Massiver Gebrauch von Interfaces, Vererbung, Generics

## Collections

---

Auf den folgenden Folien wird von einem Interface `Collection` die Rede sein, aber auch von Klassen/Objekten, die dieses Interface implementieren. Auch derartige Klassen bzw. deren Objekte werden im folgenden als `Collection` bezeichnet. Wenn das Interface gemeint ist, wird der Name jedoch in der Formatierung für Code verwendet: `Collection` (Interface) vs. `Collection` (Klasse/Objekt).

# Das Interface `Collection`

- “Oberstes” Interface in der Hierarchie von Typen für Collections
- Eine Collection repräsentiert eine Gruppe von Elementen/Objekten
  - Manche Arten von Collection erlauben Duplikate, andere nicht
  - Manche Arten von Collection sind geordnet, andere nicht
- Das Interface `Collection` wird von keiner Klasse der Java-Klassenbibliothek direkt implementiert
  - Stattdessen: Ableiten von “Unterinterfaces”, die dann implementiert werden

## Wichtige Methoden des Interface `Collection`

**`boolean add(E e)`** Garantiert, dass nach dem Aufruf `e` in der Collection enthalten ist

**`void clear()`** Entfernt alle Elemente aus der Collection

**`boolean contains(Object o)`** Gibt zurück, ob das Objekt `o` in der Collection vorhanden ist

**`boolean remove(Object o)`** Entfernt ein Vorkommen des Objekts `o` aus der Collection, falls vorhanden

**`int size()`** Gibt die Anzahl an Elementen in der Collection zurück

**`Iterator<E> iterator()`** Gibt einen Iterator über die Elemente der Collection zurück (später mehr)

**`Object[] toArray()`** Gibt ein Array zurück, das alle Elemente der Collection enthält

# Das Interface `List`

- Geordnete Struktur (beachte: geordnet  $\neq$  sortiert)
- Geänderte Semantik einiger Methoden von `Collection`
- Ergänzt um einige neue Methoden

## Wichtige Methoden des Interface `List`

`boolean add(E e)` Fügt `e` ans Ende der Liste an

`void add(int index, E e)` Fügt `e` an der Position `index` in die Liste ein

`E get(int index)` Gibt das Element an Position `index` zurück

`boolean remove(Object o)` Entfernt das erste Vorkommen des Objekts `o` aus der Liste, falls vorhanden

# Die Klasse `LinkedList`

- Implementiert das Interface `List` (und noch ein paar andere)
- Erlaubt das Erzeugen von und Arbeiten mit doppelt verketteten Listen



## Aufgabe 1

- Schaue Dir die Dokumentation der Klasse `LinkedList` an (Link) und beantworte folgende Fragen:
  - Welche Interfaces werden von der Klasse `LinkedList` implementiert?
  - Hat die Klasse `LinkedList` eine Oberklasse, und wenn ja, welche?
  - Was ist der Unterschied zwischen den Methoden `peekFirst()` und `getFirst()`? (Internetrecherche erlaubt!)

## Aufgabe 2

- Schreibe ein Programm, das ein Objekt der Klasse `LinkedList` erzeugt
  - Füge 100 ganzzahlige Zufallszahlen aus dem Intervall  $[0; 1000)$  in die Liste ein
  - Lasse die Anzahl der Elemente in der Liste ausgeben
  - Prüfe, ob die Zahl 123 in der Liste vorhanden ist
  - Entferne das erste und das letzte Element der Liste und gib jeweils seinen Wert aus
  - Lasse erneut die Anzahl der Elemente in der Liste ausgeben

# Die Klasse `ArrayList`

- Implementiert das Interface `List` (und noch ein paar andere)
- Verwendet als interne Datenstruktur ein Array
  - Damit sind einige Operationen schneller als bei `LinkedList` (insbesondere `E get(int index)`), andere dafür langsamer

# Iteratoren

---

- Interface `Iterator`
  - Erlaubt das Iterieren über alle Elemente einer Collection
  - Beinhaltet die Methoden `hasNext()`, `next()` und `remove()`
- Interface `ListIterator`
  - Unterinterface für die Arbeit mit verketteten Listen
  - Beinhaltet zusätzliche Methoden zum Hinzufügen und Ändern von Elementen
  - Erlaubt auch die Iteration in umgekehrter Richtung

- Ein ListIterator zeigt nicht direkt auf ein Element der Liste, sondern steht “zwischen” den Elementen:
  - Für eine Liste der Länge  $n$  sind also  $n + 1$  Positionen des Iterators möglich

## Mehr zum 'ListIterator'

- Ein ListIterator zeigt nicht direkt auf ein Element der Liste, sondern steht "zwischen" den Elementen:
  - Für eine Liste der Länge  $n$  sind also  $n + 1$  Positionen des Iterators möglich



**Abbildung 1:** Veranschaulichung der Positionen bei einer Liste aus 3 Elementen

## Beispielcode (1)

```
LinkedList<Integer> list = new LinkedList<>();  
list.add(new Integer(1));  
list.add(new Integer(2));  
list.add(new Integer(3));  
  
ListIterator<Integer> iter = list.listIterator();  
while ( iter.hasNext() ) {  
    System.out.println(iter.next());  
}  
System.out.println(iter.previous());
```

- Was gibt das Programm aus?



## Beispielcode (2)

Für rein lesenden Zugriff kann auch eine `foreach`-Schleife verwendet werden:

```
LinkedList<Integer> list = new LinkedList<>();  
list.add(new Integer(1));  
list.add(new Integer(2));  
list.add(new Integer(3));  
  
for ( Integer i : list ) {  
    System.out.println(i);  
}
```

## Aufgabe 3

- Schaue Dir die Dokumentation zum `ListIterator` an (Link)
- Erweitere Dein Programm aus Aufgabe 2 um (mindestens) einen `ListIterator` (vgl. Codebeispiel)
  - Iteriere über die komplette Liste und gib die Werte aus
  - Ersetze mithilfe eines Iterators das an Index 50 gespeicherte Element aus der Liste mit einem neuen Element
  - Implementiere eine Methode `void entferne(int s, int e)`, die mithilfe eines Iterators alle Elemente von Index `s` bis Index `e` aus der Liste entfernt