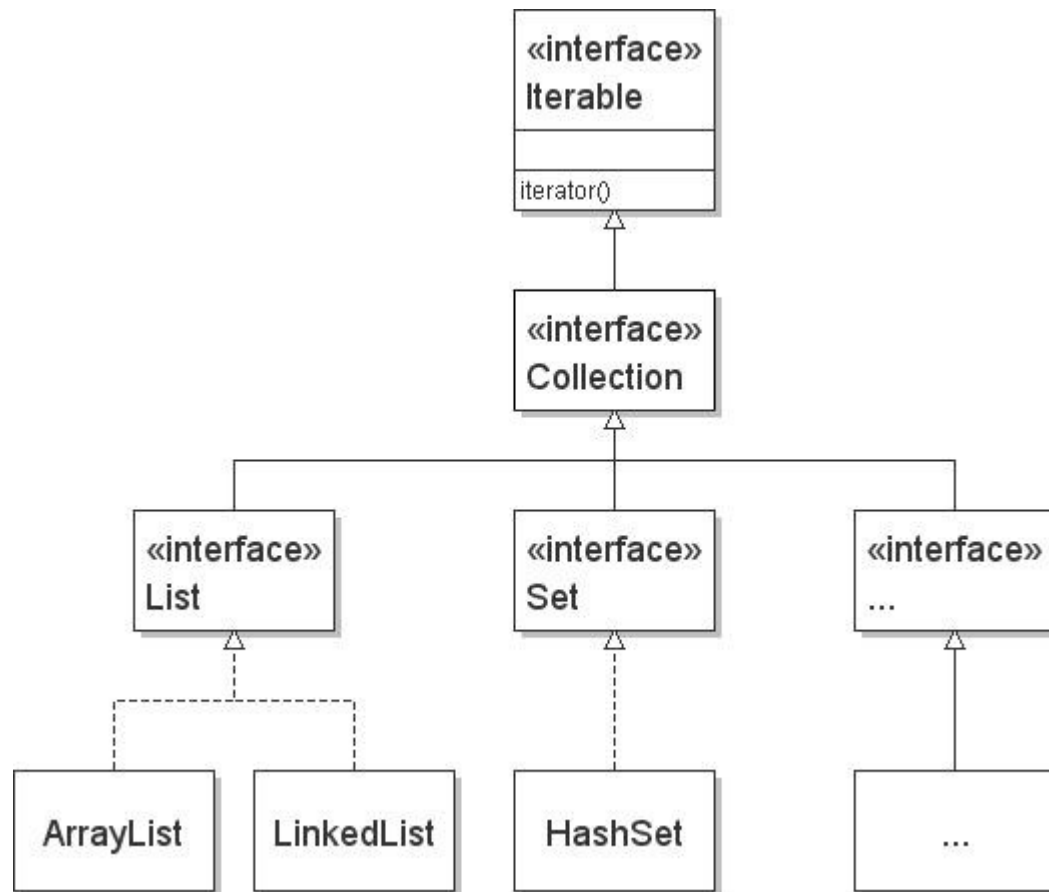


Collections

- Alle möglichen Programme hantieren mit Daten
- Je nach Datenmenge und Anwendungsfall bieten sich verschiedene Datenstrukturen an.
- Dabei liefern Collections alles von effizientem Zugriff bis speichereffiziente Datenstrukturen

Collections



Collections

- **Interface Collection** ist die Mutter aller iterierbaren Datenstrukturen
- **List**: jedes Element hat einen Index
- **Queue**: Warteschlange (z.B. FIFO)
- **Set**: Elemente nur genau einmal gespeichert

Wichtigste Methoden

//Hinzufügen von Elementen

boolean add(Object o)/**boolean** addAll(Collection c)

//Entfernen von Elementen

boolean remove(Object o)/**boolean** removeAll
(Collection c)

//gibt ein Array mit allen Elementen zurück

Object [] toArray()

//enthält Element xy

boolean contains(Object o)

//gibt die Anzahl der Elemente zurück

int size()

List

- **ArrayList**: Array variabler Länge
- **LinkedList**: verlinkte Liste, optimiert für schnelles einfügen und löschen (Queue)
- **Vector**: wie eine ArrayList, aber langsamer

List

- Neben den Methoden von Collections:
size(), add(), contains(), toArray(), remove()

```
//gibt das Element an der Stelle index zurück  
Object get(int index)
```

```
//ersetzt das Object an der Stelle index und  
gibt das alte Object zurück  
Object set(int index, Object obj)
```

```
//entfernt alle Elemente aus der Liste  
void clear()
```

ArrayList Beispiel

```
//erstelle eine Liste mit Strings
```

```
ArrayList<String> list = new ArrayList<String>();
```

```
//füge ein Element hinzu
```

```
list.add("Ein neues Element");
```

```
//Gib alle Elemente der Liste aus
```

```
for (int i = 0; i < list.size(); i++) {
```

```
    System.out.println(list.get(i));
```

```
}
```

Set

- Cool, wenn man viele (gleiche) Daten hat
- Enthält keine Duplikate
- Keine Indices, ungeordnet
- Zugriff auf die Daten über „Iterator“ oder „For each“

Set-Varianten

- **HashSet**: Hash-basierte Datenstruktur, unsortiert, schneller Zugriff bei `add()`, `remove()`, `contains()`, `size()`
- **TreeSet**: balancierte Baumstruktur, sortiert, schneller Zugriff bei `add()`, `remove()`, `contains()`, aber langsamer als `HashSet`

Iterator

- Erlaubt es eine Collection in einer Schleife zu durchlaufen

//gibt das Nächste Element zurück

Object next()

//fragt, ob es ein nächstes Element gibt

boolean hasNext()

//entfernt das letzte mit next-zurückgegebene Element

void remove()

Set und Iterator

//HashSet mit String-Elementen anlegen

```
HashSet<String> set = new HashSet<String>();
```

//Element hinzufügen

```
set.add("neues Element");
```

//Iterator erstellen und Elemente ausgeben

```
Iterator<String> it = set.iterator();
```

```
while(it.hasNext()) { //solange es ein nä.Ele gibt
```

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

```
}
```

Klasse Collections

- Klasse mit statischen Methoden
- Methoden zum
 - Sortieren
 - MinimumWert
 - MaximumWert
 - Mischen (Shuffle)
 - ...

Collections

```
ArrayList<String> list = new ArrayList<String>();
```

```
//fülle die Liste mit Elementen
```

```
for (int i = 0; i < sportArray.length; i++) {  
    list.add(sportArray[i]); //String array  
}
```

```
//sortiere die Liste
```

```
Collections.sort(list);
```

```
//gib die Liste aus
```

```
for (int i = 0; i < list.size(); i++) {  
    System.out.println(list.get(i));  
}
```

Interface Comparable

- Die Elemente in einer Liste müssen vergleichbar sein! Dafür gibt es das Interface Comparable mit der `compareTo(Object other)` - Methode, die überschrieben werden muss

Rückgabewert dieser Methode ist:

- 0: other hat den gleichen Wert
- 1: other ist größer
- -1: other ist kleiner

Interface Comparable

```
public class Project implements Comparable<Project>{  
    private String name;  
    private int prio;  
  
    @Override  
    public int compareTo(Project o) {  
        if (prio < o.getPrio()) //other ist größer  
            return 1;  
        else if (prio > o.getPrio()) //other ist kleiner  
            return -1;  
        else  
            return name.compareTo(o.getName());  
    }  
} //es gibt hier noch die Getter
```

HashMap

- Eine weitere Familie von Datenstrukturen sind assoziative Arrays (Maps)
- Zwei Objekt-Typen werden dabei miteinander verknüpft
- Möchte man wissen wie oft welches Buch vorhanden ist, kann man den Buchtitel (String) und die Anzahl der Bücher (Integer) in einer Map speichern

HashMap

- Dabei speichert die Map die Bücher als Schlüssel und ordnet die Anzahl (Wert) den Büchern zu.

//HashMap <Key, Value>

- HashMap <String, Integer> bookCollection;
- Der Schlüssel wird dabei als Hashwert gespeichert

Beispiel HashMap

```
HashMap<Project, Termin> map = new HashMap<>();
```

```
Project project = new Project("Java Project", 7);
```

```
Termin termin = new Termin(01, 05, 2016);
```

```
//den Termin und das Projekt in die HashMap einfügen  
map.put(project, termin);
```

```
//die Projecte als Set aus der HashMap holen  
Set<Project> set = map.keySet();
```

```
//Gib alle Termine aus  
for (Project project : set) {  
    System.out.println("Termin:" + map.get(project));  
}
```