

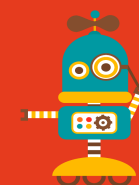


Java Collections

DI Reinhold Buchinger

Creative Commons-Lizenz CC BY-NC-SA 4.0 AT.

Höhere Abteilung für Mechatronik
Höhere Abteilung für Informationstechnologie
Fachschule für Informationstechnik



Ziele

» Java List, Set und Map anwenden können

Übersicht Java Collection

Datenstrukturen

Welche Datenstruktur kennen wir bis jetzt um mehrere Objekte gemeinsam zu speichern?

Array

Nachteile Array

- » hat eine fixe Länge (in Java)
- » ist nicht type-safe
 - » kann nicht sicher sein, welcher Datentyp gespeichert ist
- » stellt selbst keine Methoden für häufige Anwendungsfälle zu Verfügung
 - » z.B. Sortierung, Suche,...
- » ...

Java Collection

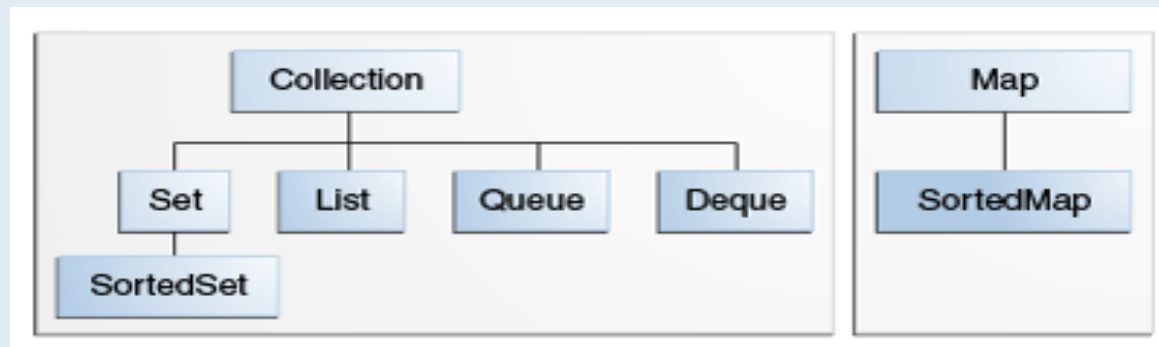
Eine Collection ist...

...eine Gruppe von Objekten.

...ein Interface in der Java API.

Java Collection

- » Java stellt verschiedene Interfaces und konkrete Implementierungen für verschiedene Anwendungsfälle zu Verfügung



- » Eine **Liste** ist eine Datenstruktur ...
 - » deren Elemente immer eine Ordnung haben (z.B. Reihenfolge des Einfügens).
 - » Duplikate erlaubt sind
 - » auf dessen Elemente über einen Index zugegriffen werden kann.
- » Die Java API kennt folgende Implementierungen einer Liste:
 - » [ArrayList](#) (bessere Performance Speicherung/Zugriff)
 - » [LinkedList](#) (bessere Performance bei Manipulationen)
 - » [Vector](#) (synchronized - nur ein Thread gleichzeitig)
 - » [Stack](#) (LIFO - last-in-first-out)

Set

- » Ein Set ist eine Datenstruktur....
 - » die keine Duplikate erlaubt.
 - » nicht unbedingt eine Ordnung haben muss.
- » Die Java API kennt drei konkrete Implementierungen eines Sets:
 - » HashSet (keine Ordnung)
 - » TreeSet (Elemente sind immer geordnet)
 - » LinkedHashSet (Reihenfolge des Einfügens bleibt erhalten)

Map

- » Eine Map ist eine Datenstruktur....
 - » die Schlüssel (keys) zu Werten (values) zuordnet
 - » z.B. Name (key) zu Telefonnummer (value)
 - » bei der alle Schlüssel eindeutig sein müssen
- » Die Java API kennt drei konkrete Implementierungen einer Map (vgl. Sets!):
 - » [HashMap](#) (keine Ordnung)
 - » [TreeMap](#) (Elemente sind immer geordnet)
 - » [LinkedHashMap](#) (Reihenfolge des Einfügens bleibt erhalten)

Generics

- » Eine Collection (z.B. ArrayList) ist so implementiert, dass ein beliebiger Datentyp darin gespeichert werden kann.
- » Im Code wird aber angegeben welchen Datentyp man in der Collection speichern will.
- » Erlaubt eine Überprüfung, ob nur Objekte vom gewünschten Datentyp eingefügt werden, bereits vom Compiler und nicht erst während der Ausführung des Programms.
- » Verhindert somit Fehler während der Programmausführung

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

Codebeispiele

Interfaces

Interface

- » Wiederholung: Was ist ein Interface?
- » Ein Interface besitzt keine Implementierungen, sondern nur Methodenköpfe und Konstanten.
- » Trennt die Definition der Schnittstelle von der Implementierung.

```
public interface RidableAnimal {
    public void rideAnimal();
}
```

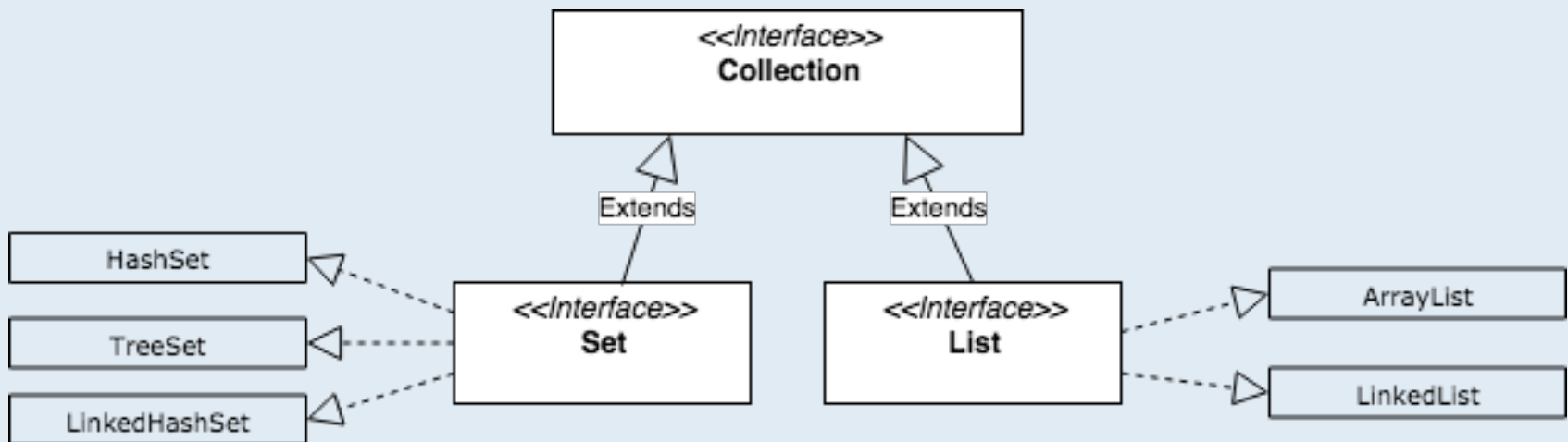
```
public class Horse implements RidableAnimal{

    @Override
    public void rideAnimal() {
        System.out.println("Riding is fun");
    }
}
```

Interfaces & Klassen

- » Collection ist ein Interface
 - » D.h. definiert etliche Methoden, implementiert sie aber nicht.
 - » Es kann kein Objekt davon erzeugt werden.
- » List und Set sind Interfaces und erben von Collection
 - » Besitzen alle Methoden von Collection und evt. noch zusätzliche
 - » Es kann kein Objekt davon erzeugt werden.
- » ArrayList (als Beispiel) implementiert das Interface List
 - » Implementiert alle Methoden des Interfaces (und könnte noch zusätzliche implementieren)
 - » Es kann ein Objekt davon erzeugt werden.

UML Diagram



Interfaces & Klassen - Beispiel

Welchen Datentyp kann die Variable list haben?

? list = **new** ArrayList<>();

Datentyp	Zugriff auf...
ArrayList (Klasse)	Methoden von ArrayList
List (Interface)	Methoden von List
Collection (Interface)	Methoden von Collection

Welcher Datentyp soll verwendet werden und warum?

Interfaces & Klassen - Beispiel 2

```
Collection<String> c = new ArrayList<>();  
c.add("Saturn"); //the method add is defined in the interface "Collection"  
c.get(0); //Error: not method "get"  
  
List<String> l = (List) c; //cast to list  
String s = l.get(0); //the interface "List" has a method "get"  
  
l.ensureCapacity(20); //Error: not method "ensureCapacity"  
  
ArrayList<String> al = (ArrayList) l; //cast  
al.ensureCapacity(20); //the class "ArrayList" has the method "ensureCapacity"
```

Conversion Konstruktor

- » Jede Collection Implementierung besitzt einen Konstruktor mit einem Parameter vom Typ *Collection*.
- » Dieser Konstruktor initialisiert die neue Collection mit allen Elementen der übergebenen Collection.
- » Welche Objekte kann ich einem solchen Konstruktor übergeben?
- » Beispiel:

```
ArrayList<String> list = new ArrayList<>();  
list.add("Semmel");  
list.add("Semmel");  
HashSet<String> set = new HashSet<>(list);
```

Array <-> Collection

- » Umwandlung Collection in Array
 - » toArray() Methode des Collection Interface
- » Umwandlung Array in Collection
 - » Arrays.asList() + evt. Conversion Konstruktor

```
String[] names = new String[]{"Paul", "Hannah", "Anna"};
```

```
List<String> list = Arrays.asList(names);
```

```
Set<String> set = new HashSet<>(Arrays.asList(names));
```

```
String[] namesArray = (String[]) set.toArray();
```

Collection durchlaufen

for-each Schleife

```
Collection<String> collection = new  
HashSet<>(Arrays.asList("Paul", "Hannah", "Anna"));  
  
for (String s: collection) {  
    System.out.println(s);  
}
```

Iterator

- » Ein Iterator muss verwendet werden wenn man Elemente während des Durchlaufs entfernen möchte.

```
Collection<String> collection = new  
HashSet<>(Arrays.asList("Paul", "Hannah", "Anna"));
```

```
Iterator<String> it = collection.iterator();  
while(it.hasNext()) {  
    System.out.println(it.next());  
}
```

Streams

Several white, stylized cloud shapes are scattered across the upper half of the slide, adding a decorative touch to the light blue background.

» Behandeln wir später.


Wrapper Klassen

Primitive Datentypen vs. Objekte

- » In einer Collection können nur Objekte gespeichert werden.
- » Trotzdem läuft der Code wenn primitive Typen (int, float, boolean,...) in Collection gespeichert werden. Warum?
- » **"Auto(un)boxing"**: Java nimmt eine automatische Umwandlung primitiver Typ <-> Objekt vor.
- » Jeder primitive Datentyp besitzt in Java ein dazugehörige Klasse.
- » **"Wrapper-Klassen"**: Sie "verpacken" (engl. "wrap") einen primitiven Datentyp

Wrapper Klassen

Primitiver Datentyp	Wrapper Klasse
<i>boolean</i>	<i>Boolean</i>
<i>byte</i>	<i>Byte</i>
<i>short</i>	<i>Short</i>
<i>int</i>	<i>Integer</i>
<i>long</i>	<i>Long</i>
<i>float</i>	<i>Float</i>
<i>double</i>	<i>Double</i>
<i>char</i>	<i>Character</i>



Fragen?
Anregungen?
Bemerkungen?