10 Java Collections

10.1 Philosophisches

Collections sind “Behälter” für Objekte. Früher wurden dazu Arrays verwendet.

Collections haben Vorteile gegenüber Arrays:

- Grösse muss nicht im Voraus bekannt sein
- Einfache Zugriffsmöglichkeiten

Natürlich ist es möglich, auch ohne die Java Collections eigene “Behälter” für Daten zu implementieren. Aber Java Collections stellt viel benötigte Versionen auf eine einfache Weise zur Verfügung.

10.2 Ersatz für arays: ArrayList

ArrayList ist *der* Ersatz für Arrays:

Früher:

```
Agent[] agents ;
agents = new Agent[10] ; // note fixed size
```

Neu:

```
ArrayList<Agent> agents ;
agents = new ArrayList<>() ; // note no size given
```

Eine ArrayList wird dann verwendet, wenn man willkürlich (= mit Positionsindex) auf die Elemente darin zugreifen möchte.

<Agent>, <>: “Generics”, siehe später. Legt fest, was in den Container “hinein darf”.

Beispiel:

```
... main ... {

    ArrayList<Agent> agents ;
    agents = new ArrayList<>() ;

    // generate 10 new agents and ``register'' them:
    for ( int ii=1; ii<=10 ; ii++ ) {
        Agent ag = new Agent() ;
        agents.add( ag ) ;
    }

    // go through all agents and do something
    for ( Agent ag : agents ) {
        ag.doSomeThing() ;
    }

    Agent ag = agents.get(5) ; // get agent number 5

    agents.remove(5) ; // remove agent number 5


}
```

Nach der Initialisierung kann `ArrayList` von der Funktionalität genau wie ein Array verwendet werden (aber mit Syntax `array.get(ii)` und `array.set(ii,content)` etc.)

Eine neu erstellte Collection ist leer. Erst nachdem (z.B. mit `add`) Elemente eingefügt worden sind, kann darauf zugegriffen werden.

10.3 Einführung in “Java generics”

Ohne “generics” ... ... würde es (nur) wie folgt gehen:

```
List persons = new ArrayList() ;
for ( int ii=0 ; ii<10 ; ii++ ) {
    Person aPerson = new Person() ;
    persons.add( aPerson ) ;
    aPerson.setName( Integer.toString(ii) ) ;
}
// (**)
Person pp = (Person) persons.get(5) ; // (#)

for ( Object oo : persons ) { // (##)
    Person person = (Person) oo ; // (***)
    System.out.println ( person.getName() ) ;
}
```

Bem:

- Ohne die Casts (#) bzw. (***) geht es nicht.

  Die List-ohne-Generics enthält sozusagen “allgemeine” Objekte, und man muss sie erst zwangsweise (per Cast) nach `Person` umwandeln.

- Beachte (##). Wenn die Collection nicht “generifiziert” ist, dann kann man nur über Objekte vom (nicht sehr leistungsfähigen) Typ `Object` iterieren.

Hinzufügen eines Objektes vom “falschen” Typ Nehmen wir nun an, jemand fügt an der Stelle (**) die Zeile

```
persons.add( "Meier" ) ;
```

ein. Was passiert?

- Der Compiler schluckt das problemlos (*kein* Compile-time Error).
- Zur Laufzeit wird der neue “add”-Befehl problemlos ausgeführt.
- Das zusätzliche Objekt wird problemlos aus der Collection wieder rausgeholt.
- Erst beim Versuch, es auf `Person` zu casten, bricht das Pgm ab:

```
Exception in thread "main" java.lang.ClassCastException:
java.lang.String cannot be cast to hh_generics.bb_simple.Person
        at hh_generics.bb_simple.Test.main(Test.java:26)
```

Interpretation: Das rausgeholte Objekt ist vom Typ `java.lang.String`, und mit diesem Objekt läßt sich der (***) verlangte Cast nicht durchführen.

[[Was ist schiefgegangen? `"Meier"` ist vom Typ `String`. Da `String` keine Erweiterung von `Person` ist, ist es nicht möglich, dieses Objekt auf `Person` zu casten.]]

Es ist wünschenswert, dass solche Fehler schon beim Kompilieren abgefangen werden.

Stellen Sie sich eine Simulation einer Einwohner-Datenbank vor, und der Laufzeit-Fehler tritt nach 3 Tagen Rechenzeit auf ...

Das erreicht man durch folgende Syntax:

```
List<Person> persons = new ArrayList<>() ;
for ( int ii=0 ; ii<10 ; ii++ ) {
    Person aPerson = new Person() ;
    persons.add( aPerson ) ;
    aPerson.setName( Integer.toString(ii) ) ;
}
```

Nun beschwert sich (bereits!) der Compiler (!) bei der Verwendung von

```
persons.add( "Meier" ) ;
```

wie folgt:

```
The method add(Person) in the type List<Person> is not applicable for
the arguments (String)
```

Interpretation: Das Interface `List<Person>` hat keine Methode mit der Signatur `add(String)`.

Man kann sich das vielleicht vorstellen wie eine Art Schablone, die von Objekten erfüllt werden muss, um in der Collection angenommen werden.

Mit der generifizierten Liste geht auch

```
Person pp = person.get(5) ; // note missing cast
```

sowie (wie bereits gewohnt)

```
for ( Person person : persons ) {
   // note missing cast here:
   person.setName("Mueller") ;
}
```

Nochmal: Ein ‘cast’ bedeutet immer, dass man erst zur Laufzeit herausfindet, ob das Objekt vom richtigen Typ ist.

Ohne ‘cast’ (also mit ‘generics’) wird schon beim Kompilieren geprüft, ob das Objekt vom richtigen Typ ist.

10.4 Arrays

("Vorgänger" von Collections)

10.4.1 Basics

- viele Elemente vom selben Typ
- feste Länge (Anzahl der Elemente) nach Erzeugung zur Laufzeit

```
...
int[] array ; // Deklaration
...
array = new int[10] ; // Erzeugung
...
for ( int ii=0 ; ii<array.length ; ii++ ) { // _NOT_ xxx.length()
    array[ii] = ii ;
}
...
```

Geht auch mit Objekten, nicht nur mit primitiven Typen:

```
...
Person[] anotherArray ;
...
```

Man kann vereinfacht deklarieren, erzeugen, und initialisieren:

```
...
String[] answers = { "true", "false", "32", "maybe", "blue" } ;
...
for ( int ii=0 ; ii<answers.length ; ii++ ) {
    System.out.println ( "ii: " + ii + " answer: " + answer[ii] ) ;
}
...
```

Starten immer bei 0 und gehen bis `array.length-1`.

10.4.2 Arrays of Arrays

```
int[][] aMatrix = new int[4][];
for (int ii = 0; ii < aMatrix.length; ii++) {
    aMatrix[ii] = new int[ii]; //create sub-array
    for (int jj = 0; jj < aMatrix[ii].length; jj++) {
        aMatrix[ii][jj] = ii + jj;
    }
}
for (int ii = 0; ii < aMatrix.length; ii++) {
    for (int jj = 0; jj < aMatrix[ii].length; jj++) {
        System.out.print(aMatrix[ii][jj] + " ");
    }
    System.out.println();
}
```

Bem:

- Die Länge der Sub-Arrays kann später (= zur Laufzeit) festgelegt werden (wie in obigem Beispiel).
- Geht auch mit mehr als 2 Dimensionen.

10.5 Vector

Benimmt sich in etwa so wie eine Collection Klasse, ist aber historisch gesehen älter. M.E. kaum bis gar nicht notwendig bei neuem Code, kommt aber vor bei altem Code.

10.6 Collections (im engeren Sinne)

10.6.1 Das `Collection` interface

```
public interface Collection<E> {
  // Basic Operations
  int size();
  boolean isEmpty();
  boolean contains(Object element);
  boolean add(E element); // Optional
  boolean remove(Object element); // Optional
  Iterator<E> iterator();

  // Bulk Operations
  boolean containsAll(Collection<?> c);
  boolean addAll(Collection<? extends E> c); // Optional
  boolean removeAll(Collection<?> c); // Optional
  boolean retainAll(Collection<?> c); // Optional
  void clear(); // Optional

  // Array Operations
  Object[] toArray();
  T[] toArray(T[] a);
}
```

10.6.2 Iteratoren

Wir hatten bereits

```
for ( Agent ag : agents ) {
   ag.doSomething() ;
}
```

Etwas komplizierter geht auch:

```
for ( Iterator<Agent> it=agents.iterator() ; it.hasNext() ; ) {
    Agent ag = it.next() ;
    ag.doSomeThing() ;
}
```

- Jede `Collection` hat einen `Iterator`; der Aufruf

  ```
  .... ( Iterator<...> it=ccc.iterator() ...
  ```

  ist gültig, solange `ccc` irgendwie von `Collection` abgeleitet ist.

- `it.hasNext()` prüft, ob es noch ein weiteres Element gibt;

- `it.next()` holt das nächste Element aus der Collection und setzt den Iterator eine Position weiter.

Iteratoren haben Vorteile gegenüber der vereinfachten Syntax, wenn man während des Iterierens die Collection verändern will:

```
for ( Iterator<Agent> it=workPop.iterator() ; it.hasNext() ; ) {
   Agent ag = it.next() ;
   if ( ag.getAge > 65 ) {
      it.remove() ; // (*)
   }
}
```

Folgendes geht nur theoretisch:

```
for ( Agent ag : workPop ) {
   if ( ag.getAge > 65 ) {
      workPop.remove( ag ) ; // WRONG!
   }
}
```

Gibt vermutlich einen Laufzeit-Fehler ("fail-fast" bei `ConcurrentModificationException`).

Man soll eine Collection nicht abseits vom Iterator verändern, während man über sie iteriert.

10.6.3 Das `List` Interface

```
public interface List<E> extends Collection<E> { // also ZUSAETZLICH zu Collection!
  // Positional Access
  E get(int index);
  E set(int index, E element); // Optional
  void add(int index, E element); // Optional ; ``insert''
  boolean remove(int index); // Optional
  boolean addAll(int index, Collection<? extends E> c); // Optional

  // Search
  int indexOf(Object o);
  int lastIndexOf(Object o);

  // Iteration
  ListIterator<E> listIterator();
  ListIterator<E> listIterator(int index);

  // Range-view
  List<E> subList(int from, int to);
}
```

10.6.4 `List` Interface mit `LinkedList` Implementierung

Beispiel:

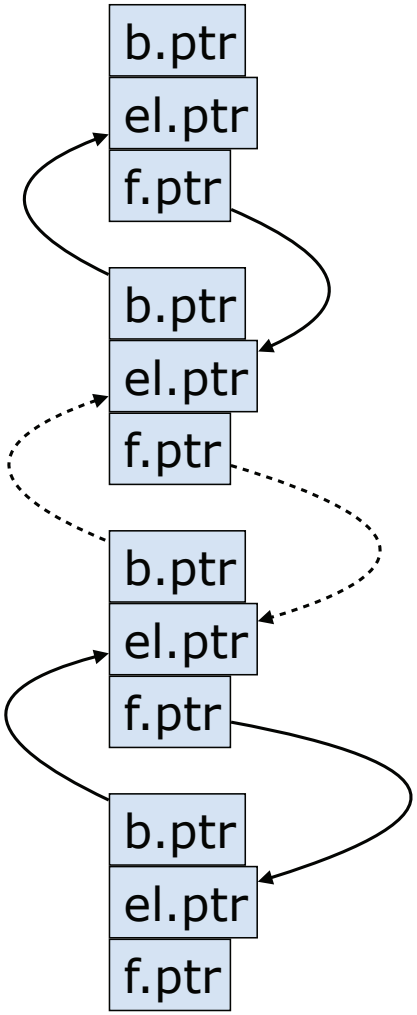
```
... main ... {
    List agents ;
    agents = new LinkedList() ;

    // fill:
    for ( int ii=0 ; ii<10; ii++ ) {
        Agent ag = new Agent() ;
        agents.add( ag ) ;
    }

    ...
```

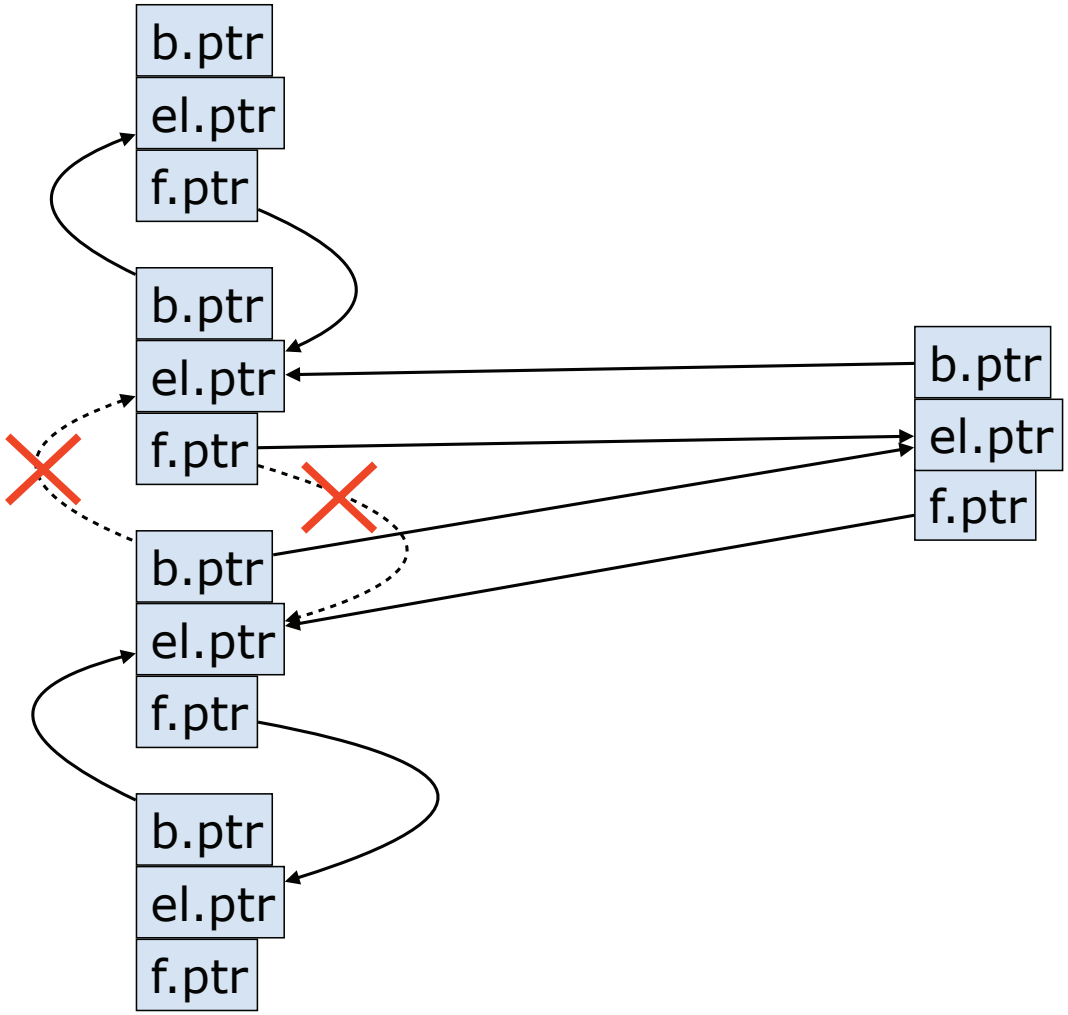
Also genauso wie bei `ArrayList()` Implementierung.

`LinkedList` kann das gleiche wie `ArrayList`. Aber mit anderer Effizienzcharakteristik:

- *Zugriff* ist bei großen `LinkedLists` *sehr* langsam, bei großen `ArrayLists` schnell.
- Hingegen ist Einfügen/Entfernen in der Mitte der Liste bei `LinkedList` sehr schnell, bei großen `ArrayLists` sehr langsam.

[[draw]]


b.ptr
el.ptr
f.ptr
b.ptr
el.ptr
f.ptr
b.ptr
el.ptr
f.ptr
b.ptr
el.ptr
f.ptr


b.ptr
el.ptr
f.ptr
b.ptr
el.ptr
f.ptr
b.ptr
el.ptr
f.ptr
b.ptr
el.ptr
f.ptr
b.ptr
el.ptr
f.ptr

10.7 Indizieren mit beliebigen Objekten statt mit fortlaufenden Zahlen: HashMap

Die ArrayList indiziert mit Integers. Z.B. geht so etwas wie

```
ArrayList<Agent> agents = new ArrayList<>() ;
...
Agent ag = agents.get(5) ;
```

Das Map-Interface ist die Verallgemeinerung davon: Es indiziert mit beliebigen Objekten. Z.B. geht so etwas wie

```
HashMap<String,Agent> agentsMap = new HashMap<>();
...
Agent ag = agentsMap.get("Meier") ;
```

Die Objekte, mit denen man indiziert, werden "key" (Schlüssel) genannt.

ArrayList befüllt man über

```
Agent ag1 = new Agent() ;
agents.add(ag1) ;
```

HashMap befüllt man über

```
Agent ag1 = new Agent() ;
agentsMap.put( "Meier" , ag1 ) ;
```

Durch List iteriert man mit

```
for ( Agent ag : agents ) {
   ag.doSomething() ;
}
```

Fär Map ist die Standard-Methode:

```
for ( Agent ag : agentsMap.values() ) {
                 ^^^^^^^^^

   ag.doSomething() ;
}
```

10.8 Maps

10.8.1 Interface

```
public interface Map<K,V> {
    // Basic operations
    V put(K key, V value);
    V get(K key);
    V remove(K key);
    boolean containsKey(K key);
    boolean containsValue(V value);
    int size();
    boolean isEmpty();
    // Bulk operations
    void putAll(Map<K,V> m);
    void clear();
    // Collection Views
    public Set<K> keySet(); // Set (!) of the keys
    public Collection<V> values(); // all values
    public Set<Map.Entry<K,V>> entrySet(); // (key,value)-pairs. See below
    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

Bem.:

- Die Notation `<K,V>` kommt von "Generics" und steht als Platzhalter für Typen. Z.B. `Map<String,Agent>`.
- Das `Map`-Interface erlaubt höchstens einen Eintrag pro Key.

  Eine multimap (= mehr als einen Eintrag pro Key) gibt es nicht bei Java Collections, kann aber durch Kombination von Map und List emuliert werden.

Ansonsten siehe Javadoc.

Das demonstriert dann am Beispiel auch gleich "Klassen innerhalb von Klassen" (hier: Klasse(ninterface) `Entry` innerhalb von `Map`).

Iteratoren für `Map`

Iteration über die keys:

```
for (Iterator<...> it = m.keySet().iterator(); it.hasNext(); ) {
    if (it.next().isBogus()) {
        it.remove();
    }
}
```

(Das funktioniert, obwohl bei näherem Hinsehen nicht ganz klar: obwohl von der Syntax her nur der `key` entfernt wird, wird tatsächlich der ganze `Entry` entfernt.)

Iteration über die values ist analog:

```
for (Iterator<Agent> it = m.values().iterator(); it.hasNext(); ) {
   Agent ag = it.next() ;
   ag.update() ;
}
```

Häufiger wird man so etwas machen wie:

```
for ( Agent ag : agents.values() ) {
   ag.update() ;
}
```

Man kann auch über die entries iterieren:

```
for ( Iterator<Map.Entry> it = agents.entrySet().iterator() ; it.hasNext() ; ) {
   Map.Entry e = it.next() ;
   System.out.println ( " key: " + e.getKey() +
       " value: " + e.getValue() ) ;
}
```

Das ist (natürlich) dann sinnvoll, wenn man “key” und “value” gleichzeitig benötigt (warum auch immer).