Inhalt

Motivation

Datenstrukturen zur Implementierungen von „Sammlungen“:

- Arrays fester Länge
- Dynamische Arrays
- Verkettete Listen

Wie kann ein Durchlauf durch eine solche Sammlung implementiert werden?

Was uns fehlt:

- eine (für alle Datenstrukturen) einheitliche Möglichkeit, sukzessive auf alle Elemente einer Sammlung zuzugreifen.
- Genauer: Einheitliche Methoden, um
  - festzustellen, ob es noch unbesuchte Elemente in der Sammlung gibt
  - das nächste (noch unbesuchte) Element zu bekommen.

Java-API-Interface Iterator<T>

- Das Interface Iterator<T> der Java-API definiert genau dies:

```
boolean hasNext();
T next();
```

- Wenn wir eine implementierende Klasse
  (zB **class** MyIterator<T> **implements** Iterator<T>)
  hätten, dann könnten wir einen Durchlauf durch die Datenstruktur nach diesem Muster implementieren:

```
Iterator<T> it = new MyIterator<>();
while (it.hasNext())
        doSomething(it.next());
```

Beispiel: Iterator für Dynamische Arrays

```
public class DynArrayIterator<T> implements Iterator<T> {

        private DynArray<T> dArr;
        private int index;

        public DynArrayIterator(DynArray<T> dArr) {
                this.dArr = dArr;
                index = 0;
        }

        public boolean hasNext() {
                return (index < dArr.size());
        }

        public T next() {
                return dArr.get(index++);
        }
}
```

Beispiel: Iterator für 2D-Arrays

```
public class ZeilenIterator2DArray<T> implements Iterator<T> {

    T a[][];
    int i;
    int j;

    public Iterator2DArray(T[][] a) {
        this.a = a;
        this.j = 0;
        this.i = 0;
    }

    public boolean hasNext() {
        return (i >= 0 && i < a.length &&
                j >= 0 && j < a[i].length);
    }
```

Beispiel: Iterator für 2D-Arrays (Forts.)

```
    public T next() {
        T v = a[i][j];

        j++;
        if(j >= a[i].length) {
                i++;
                j = 0;
        }
        return v;
    }
}
```

Nutzung eines Iterators

Beispiel: zeilenweise Ausgabe eines 2D-Arrays

```
Integer[][] arr2 = { {1, 2, 3}, {4, 5}, {6, 7, 8, 9}};

Iterator<Integer> it = new ZeilenIterator2DArray<>(arr2);
while (it.hasNext()) {
      System.out.print(it.next() + " ");
}
System.out.println();
```

liefert die Ausgabe

```
1 2 3 4 5 6 7 8 9
```

(noch kein) Beispiel: Iterator für verkettete Listen

```
public class EVLIterator implements Iterator<T>{
        private ListenElem crs;

        public EVLIterator({EVL<> evl) {
            csr = evl.first;
        }

        public boolean hasNext() {
                return crs != null;
        }

        public T next() {
                T v = crs.value;
                crs = crs.next;
                return v;
        }
}
```

Problem: diese Klasse benötigt Zugriff auf die innere Klasse ListenElem und auf das private Attribut first .

EVLIterator als innere Klasse von EVL<T>

- Problem:
  Zugriff auf die innere Klasse und auf das private Attribut nötig.
- Lösung:
  Definiere die Klasse EVLIterator als innere Klasse in EVL<T>:
  (Konstruktor und Übergabe des EVL-Objektes nicht mehr nötig)

```
class EVLIterator implements Iterator<T>{
      ListenElem crs = first;

      public boolean hasNext() {
            return crs != null;
      }

      public T next() {
            T v = crs.value;
            crs = crs.next;
            return v;
      }
}
```

Nutzung von EVLIterator „von innen“

- Kein Problem: Nutzung innerhalb der Klasse EVL<T>, zB zur Implementierung der Methode toString():

```
public String toString() {
        String s = "";
        Iterator<T> it = new EVLIterator();
        while(it.hasNext()) {
                s += it.next().toString() + " ";
        }
        return s;
}
```

Nutzung von EVLIterator „von außen“

- Nutzung auch außerhalb der Klasse EVL<T> möglich:

```
public static void testIterator() {
        EVL<String> evl = new EVL<>();
        evl.insert("Hallo");
        evl.insert("du");
        evl.insert("da");

        Iterator<String> it = evl.new EVLIterator(); //(*)

        while (it.hasNext())
                System.out.print(it.next() + " ");
        }
```

An der Zeile (∗) erkennbar:

- Der Iterator ist an ein EVL-Objekt „gebunden“!

Zwischenstand - Überblick

- Iterator als „externe“ öffentliche Klasse
  - Beispiele
    - ZeilenIterator2DArray
    - DynArrayIterator
  - Konstruktor benötigt das zu iterierende Objekt als Übergabeparameter
  - Aufruf-Bsp: Iterator <Integer> it = **new** ZeilenIterator2DArray<>(arr2);
- Iterator als innere Klasse einer Datenstruktur
  - Beispiele
    - InternerDynArrayIterator
    - EVLIterator
  - Konstruktor ohne Parameter
  - Iterator ist an ein Objekt der „äusseren“ Klasse gebunden
  - Aufruf-Bsp. (von aussen): Iterator <String> it = evl.**new** EVLIterator();
- Nutzung immer gleich:

```
while (it.hasNext())
    doSomething(it.next());
```

Inhalt

Motivation

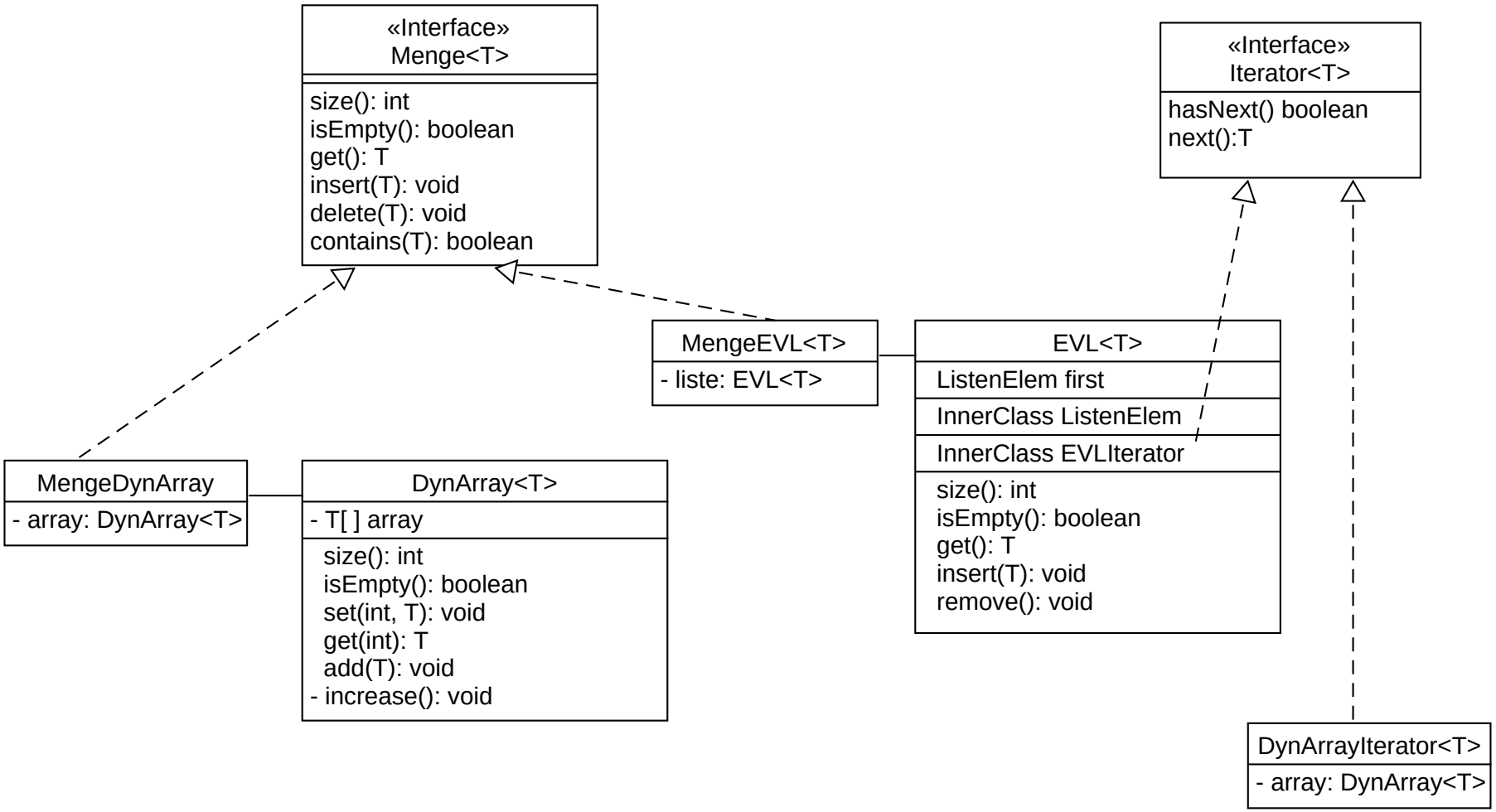
Beispiel: Das **interface** Menge haben wir mithilfe verschiedener Datenstrukturen implementiert:

- mittels eines dynamischen Arrays
- mitttels einer verketteten Liste
- weitere (andere) Implementierungen sind denkbar

was uns fehlt:

- die Möglichkeit, ohne Kenntnis der konkreten Implementierung
- über beliebige Mengen-Objekte zu iterieren

Beispiel Mengen - UML



Beispiel: Vereinigung von Mengen

Was wir wollen:

- eine Methode **static void** merge(Menge<T> a, Menge<T> b)
- die der Menge *a* alle Objekte aus *b* hinzufügt,
- ohne *b* dabei zu verändern.

(Unvollständiger Code!)

```
public static void merge(Menge<T> a, Menge<T> b) {
        // irgendwie einen Iterator fuer b erzeugen
        Iterator<T> it = ...;
        while (it.hasNext()) {
                a.insert(it.next());
}
```

Java-API-Interface Iterable<T>

Was wir brauchen:

- eine Garantie, dass es für Mengen-Objekte - egal wie sie implementiert sind - einen Iterator gibt
- eine Möglichkeit - unabhängig von der Implementierung - einen Iterator für die Menge *b* zu erzeugen

Das Interface `Iterable<T>` der Java-API definiert genau dies:

```
interface Iterable<T> {
        Iterator<T> iterator();
}
```

Mengen iterierbar machen

- Wir verlangen nun, dass jede Implementierung von Menge<T> auch das Interface Iterable <T> implementiert:

```
interface Menge<T> extends Iterable<T> {
    // wie bisher
}
```

- ... und implementieren in den beiden Klassen MengeDynArray und MengeEVL jeweils die Methode Iterator <T> iterator():
  - zB in MengeDynArray:

```
public Iterator<T> iterator() {
        return new DynArrayIterator<>(this.array);
}
```

  - zB in MengeEVL:

```
public Iterator<T> iterator() {
        return liste.new EVLIterator();
}
```

Zurück zum Beispiel: Vereinigung von Mengen

Wir können nun den Code von Folie 5-16 vervollständigen:

```
public static void merge(Menge<T> a, Menge<T> b) {
        // einen Iterator fuer b erzeugen
        Iterator<T> it = b.iterator();
        while (it.hasNext()) {
                a.insert(it.next());
}
```

Weitere Verbesserung

Es fällt auf:
Es ist keine Mühe, auch die Klassen DynArray<T> und EVL<T> das Interface Iterable<T> implementieren zu lassen:

```
public class DynArray<T> implements Iterable<T>{
        // ...
        public Iterator<T> iterator() {
                return new DynArrayIterator<>(this);
        }
}

public class EVL<T> implements Iterable<T>{
        // ...
        public Iterator<T> iterator() {
                return new EVLIterator();
        }
```

Letzer Schritt

Kann dann die Implementierungen von `iterator()` in den Klassen MengeDynArray und MengeEVL anpassen:

- In MengeDynArray:

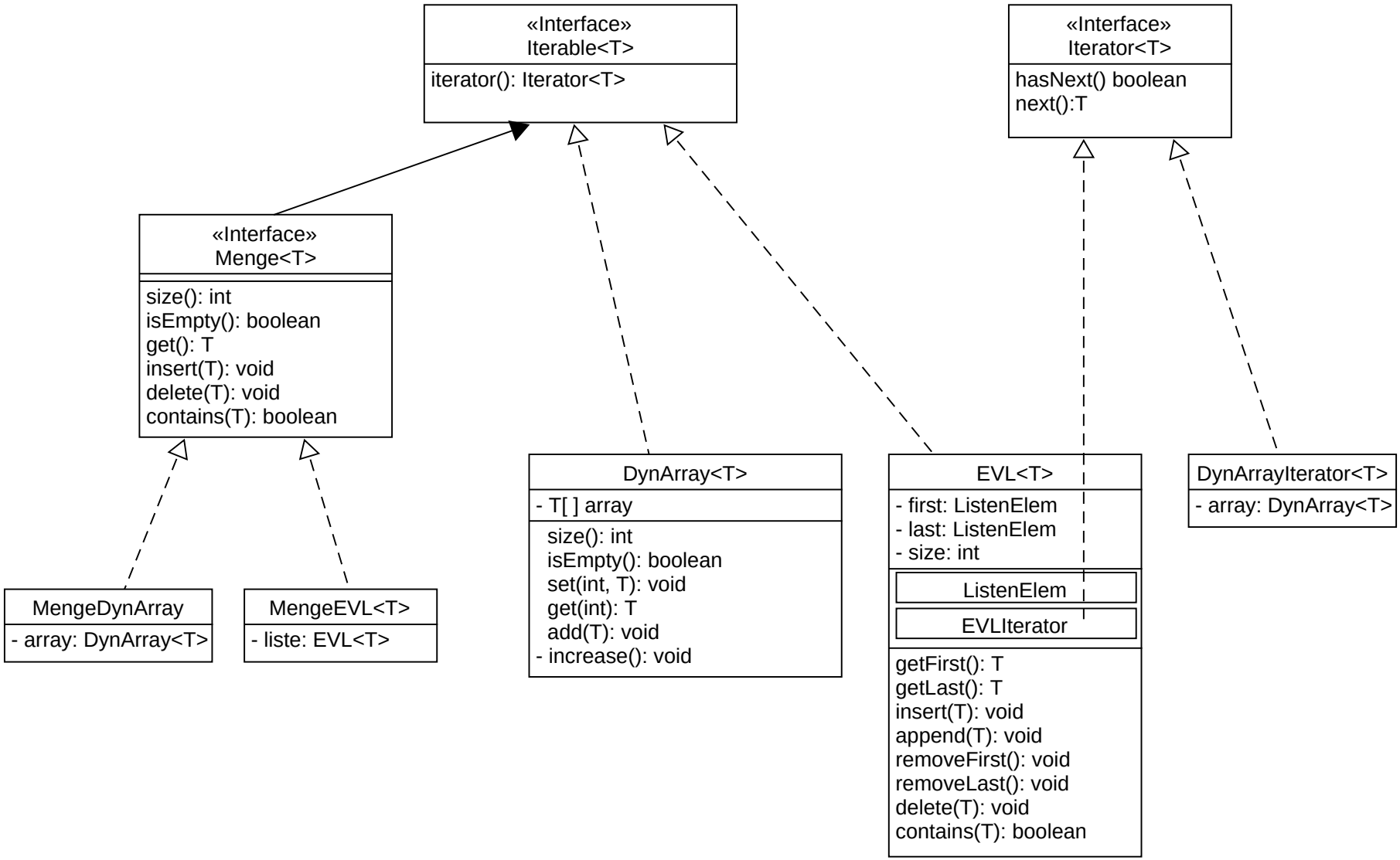
```
public Iterator<T> iterator() {
        return this.array.iterator();
}
```

- In MengeEVL:

```
public Iterator<T> iterator() {
        return this.liste.iterator();
}
```

(Auf die Nutzung in der Methode `merge` hat das keinen Einfluss.)

Beispiel Mengen - UML



Inhalt

Motivation

Die „Standardfloskel“ zum Umgang mit Iteratoren

```
Iterator<T> it = b.iterator();
while (it.hasNext()) {
        doSomething(it.next());
```

kann auch abgekürzt werden zu

```
for (T e: b)
        doSomething(e);
```

- Voraussetzung:
  b ist vom Typ `Iterable<T>` oder von einem Array-Typ.
- Wir lesen „**for**(T e: b)“ als „für jedes e aus b ...“.
  (Daher wird dieser Schleifentyp manchmal auch als „for-each-Schleife“ bezeichnet.)

Erweiterte for-Schleife für Iterable<T>-Objekte

Genauer: Falls *b* von einem Iterable <T>-Typ ist, steht die Konstruktion

```
for (T e: b)
        doSomething(e);
```

für (Pseudocode)

```
for (Iterator<T> i = b.iterator; i.hasNext(); ) {
        T e = i.next();
        doSomething(e);
}
```

Erweiterte for-Schleife für Arrays

Falls *b* von einem Array-Typ (`Typ[ ] b = new Typ[...];`) ist, dann steht

```
for (Typ e: b)
        doSomething(e);
```

für (Pseudocode)

```
for (int i = 0; i < b.length; i++) {
        Typ e = b[i];
        doSomething(e);
}
```

- Konsequenz:
  Falls `Typ` ein simple type ist, wird die Aktion `doSomething()` nur auf einer Kopie von `b[i]` ausgeführt!
- $\rightsquigarrow$ kein schreibender Zugriff auf `b[i]` möglich!
- Faustregel:
  Erweitertes for bei Arrays nur dann anwenden, wenn nur lesend auf die Array-Elemente zugegriffen wird!