Generische Klassen und Methoden

CS1016 Programmierung interaktiver Systeme

von Prof. Dr. Weigel

Generische Klassen und Methoden

1. Motivation: Eine ArrayList in Java
2. Generische Klassen und Methoden
3. Collections in Java (Listen, Sets und Maps)

Was ist eine ArrayList?

Datenstruktur zum Speichern von beliebig vielen Inhalten

- Beinhaltet intern einen Array (daher *Array*List)
- Erlaubter Zugriff aber nur über Methoden (Kapselung!)

Typische Methoden auf Listen:

```
void add(int value):
```

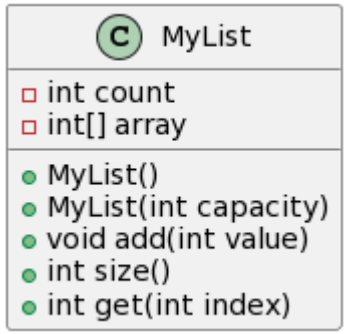
Fügt das Element value dem Ende der Liste hinzu

```
int size():
```

Gibt die Anzahl der Elemente zurück

```
int get(int index):
```

Gibt das Element an der Stelle *index* zurück



Einfache Implementierung einer ArrayList

```
class MyList {
    int count = 0;
    int[] data;
    public MyList(int capacity) {
        data = new int[capacity];
    }

    public void add(int value) {
        if(data.length == count) {
            var newArray = new int[count * 2];
            System.arraycopy(data, 0, newArray, 0, count);
            data = newArray;
        }
        data[count] = value;
        count++;
    }
    // [...]
}
```

Beispiel (JShell):

```
var a = new MyList(3);
a.add(1);
a.data
// → [1, 0, 0]

a.add(2);
a.add(3);
a.data
// → [1, 2, 3]

// Liste erweitert internen Array
a.add(4);
a.data
// → [1, 2, 3, 4, 0, 0]
```

*) Die Variablen *count* und *data* sollten normalerweise auf private gesetzt werden. Der direkte Zugriff dient hier nur der Veranschaulichung.

Einfache Implementierung einer ArrayList

```
class MyList {
    int count = 0;
    int[] data;
    public MyList(int capacity) {
        data = new int[capacity];
    }

    public void add(int value) {
        if(data.length == count) {
            var newArray = new int[count * 2];
            System.arraycopy(data, 0, newArray, 0, count);
            data = newArray;
        }
        data[count] = value;
        count++;
    }
    // [...]
}
```

Problem:
Diese Implementierung einer ArrayList funktioniert nur für *int-Werte*.

Was ist mit...
- double und float
- Strings
- Pokemon
- ...

Idee 1: Liste für die Basisklasse “Object”

```
class MyObjectList {
    int count = 0;
    Object[] data;
    public MyObjectList(int capacity) {
        data = new Object[capacity];
    }

    public void add(Object value) {
        if(data.length == count) {
            var newArray = new Object[count * 2];
            System.arraycopy(data, 0, newArray, 0, count);
            data = newArray;
        }
        data[count] = value;
        count++;
    }
    public Object get(int index) {
        return data[index];
    }
}
```

Beispiel (JShell):

```
var a = new MyObjectList(3);
a.add(123);
a.add(777.77);
a.add("Bla bla");

int first = a.get(0);
// |  Error:
// |  incompatible types:
java.lang.Object cannot be
converted to int

int first = (int)a.get(0);
```

→ **Funktioniert, aber erfordert viele Typumwandlungen**

- ☹ Aufwändig
- ☹ Mögliche Fehlerquelle

Idee 2: Eine Liste pro Datentyp

Integer

```
class MyIntList {
    int count = 0;
    int[] data;
    public MyIntList(int capacity) {
        data = new int[capacity];
    }

    public void add(int value) {
        if(data.length == count) {
            var newArray = new int[count * 2];
            System.arraycopy(data, 0, newArray, 0, count);
            data = newArray;
        }
        data[count] = value;
        count++;
    }

    public int get(int index) {
        return data[index];
    }
}
```

Double

```
class MyDoubleList {
    int count = 0;
    double[] data;
    public MyDoubleList(int capacity) {
        data = new double[capacity];
    }

    public void add(double value) {
        if(data.length == count) {
            var newArray = new double[count * 2];
            System.arraycopy(data, 0, newArray, 0, count);
            data = newArray;
        }
        data[count] = value;
        count++;
    }

    public double get(int index) {
        return data[index];
    }
}
```

String (usw.)

```
class MyStringList {
    int count = 0;
    String[] data;
    public MyStringList(int capacity) {
        data = new String[capacity];
    }

    public void add(String value) {
        if(data.length == count) {
            var newArray = new String[count * 2];
            System.arraycopy(data, 0, newArray, 0, count);
            data = newArray;
        }
        data[count] = value;
        count++;
    }

    public String get(int index) {
        return data[index];
    }
}
```

Aber...

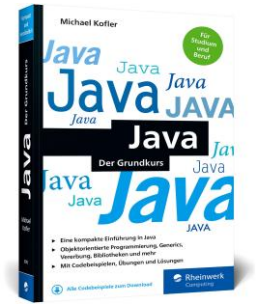
- Sehr viel Copy+Paste. Schlecht wartbar und erweiterbar.
- Was ist mit unseren eigenen Datentypen (z.B. Pokemon)?

Generische Klassen und Methoden

1. Motivation: Eine ArrayList in Java
2. Generische Klassen und Methoden
3. Collections in Java (Listen, Sets und Maps)

Wir programmieren eine allgemeine Klassen, welche nicht direkt vom Datentyp abhängt
→ Der Compiler erzeugt die konkreten Klassen

Vertiefung in



Kapitel 16

Beispiel: Dupel (geordnetes Paar)

Ein Tupel mit genau zwei Elementen (hier: first und second)

```
class MyDupel {
    int first;
    int second;

    MyDupel(int first, int second) {
        this.first = first;
        this.second = second;
    }
}
```

oder immutable:

```
record MyDupel(int first, int second) {}
```

Wie erzeugen wir eine
generische Klasse
für unser Dupel?

1. In der Signatur einen abstrakten Typ benennen
2. Den Typ in der Klasse als normalen Datentyp verwenden

Warum `Integer` und nicht `int`?
Generische Klassen erwarten einen Klassentyp. Primitive Typen wie int, char, boolean, usw. sind keine Klassen und daher nicht erlaubt. Dafür gibt es in Java spezielle Wrapper-Klassen wie Integer, Character und Boolean. Diese werden automatisch von und zu den Primitiven Typen umgewandelt werden.

Syntax: <T> steht für einen generischen Typen "T"

Generische Klasse

```
// T repräsentiert einen beliebigen Datentyp
// T kann überall in der Klasse wie ein Datentyp verwendet werden
// (z.B. für Variablen, Parameter, Rückgabewerte, ...)
class MyDupel<T> {
    T first;
    T second;

    MyDupel(T first, T second) {
        this.first = first;
        this.second = second;
    }
}
```

Beispiel:

```
record MyDupel<T>(T first, T second) {}

var t1 = new MyDupel<Integer>(4, 2);
t1.first      // → 4 (Typ: Integer)
t1.second     // → 2 (Typ: Integer)

var t2 = new MyDupel<String>("Hello", "World");
t2.first      // → "Hello" (Typ: String)
t2.second     // → "World" (Typ: String)
```

☺ Die Klasse unterstützt beliebige Datentypen
☺ Setzen des Datentyps beim Initialisieren mit <...>

Mehr als ein generischer Datentyp

Generische Klasse

```
class KeyValuePair<K, V> {
    K key;
    V value;

    KeyValuePair(K key, V value) {
        this.key = key;
        this.value = value;
    }
}
```

Beispiel:

```
var p1 = new KeyValuePair<Integer, String>
                          (25, "Pikachu");
p1.key      // → 25 (Typ: Integer)
p1.value    // → "Pikachu" (Typ: String)

var p2 = new KeyValuePair<String, Double>
                          (“Pi", 3.14159265359);
p2.key      // → "Pi" (Typ: String)
p2.value    // → 3.14159265359 (Typ: Double)
```

Typeinschränkung mit “<T extends XYZ>” (Interfaces oder Vererbung)

Einschränkung auf Subklassen oder Klassen die ein bestimmtes Interface implementieren

→ Erlaubt das Verwenden von Methoden und Variablen

```
interface Nameable {
    String getName();
}

record Person(String firstname, String lastname)
implements Nameable {
    public String getName() {
        return lastname +", "+ firstname;
    }
}
```

```
public class AddressBook<T extends Nameable> {
    List<T> data = new ArrayList<T>();

    void printNames() {
        for(var d : data)
            System.out.println(d.getName());
    }
}

// JShell Test
var ab = new AddressBook<Person>()
ab.data.add(new Person("Max", "Mustermann"));
ab.data.add(new Person("Erika", "Mustermann"));
ab.printNames();
```

Nur wegen “extends Nameable” nutzbar

Generische Interfaces und Methoden

Generisches Interface

```
enum CompareResult { LESS, EQUAL, GREATER }

interface Comparable<T> {
     CompareResult compare(T other);
}


record MyInt(int value) implements Comparable<MyInt> {
    public CompareResult compare(MyInt other) {
        if(value > other.value)
            return CompareResult.GREATER;
        if(value < other.value)
            return CompareResult.LESS;
        return CompareResult.EQUAL;
    }
}
```

Generische Methode

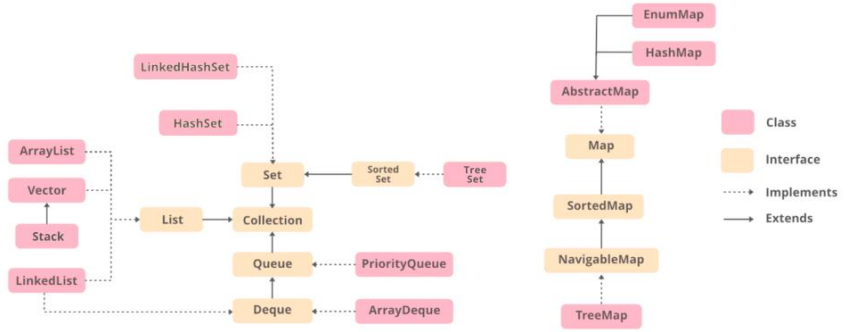
```
abstract class Helper {
    static <T extends Comparable<T>> T max(T a, T b) {
        if(a.compare(b) == CompareResult.GREATER)
            return a;
        else
            return b;
    }
}

// JShell Test
Helper.max(new MyInt(132), new MyInt(777))
```

Generische Klassen und Methoden

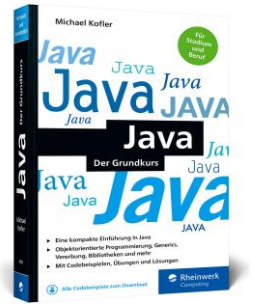
1. Motivation: Eine ArrayList in Java
2. Generische Klassen und Methoden
3. Collections in Java (Listen, Sets und Maps)

Java enthält bereits viele generische Klassen zum Speichern und Organisieren von Daten
→ Häufig “Collections” genannt



[Bildquelle: https://www.geeksforgeeks.org/collections-in-java-2/]

Vertiefung in

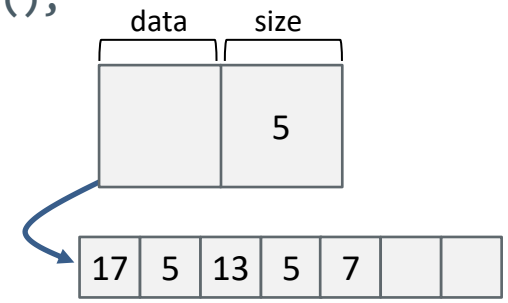


Kapitel 12

List<T>: ArrayList<T> und LinkedList<T>

ArrayList<T>

```
var al = new ArrayList<Integer>();
for(int i=0; i < 1000; i++) {
    al.add(i);
}
al.get(500);
```



Vorteile

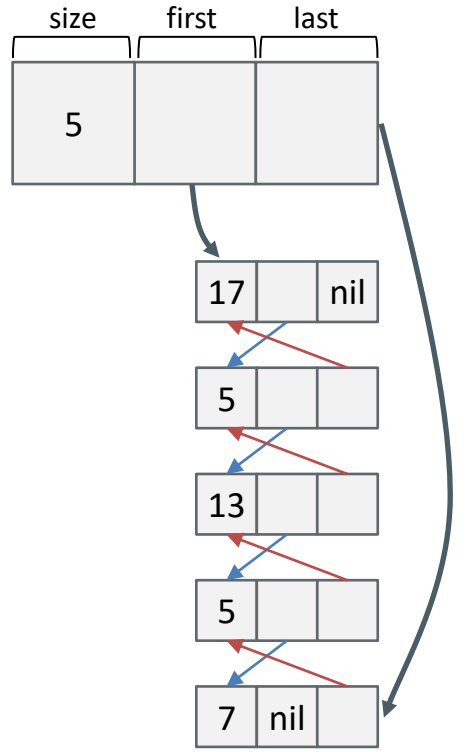
- Ähnlich nutzbar wie Arrays (`get(i)` statt `[i]`)
- Daten liegen nebeneinander im Speicher

Nachteile

- Einfügen am Anfang und in der Mitte ist sehr langsam (kopieren aller nachfolgenden Elemente)

LinkedList<T>

Alle kennen ihre direkten Nachbarn (wie eine Telefonkette)



Vorteile

- Sehr schnelles Einfügen am Anfang und Ende der Liste
- Schnelles Einfügen, wenn einer der Nachbarn bekannt ist

Nachteile

- `get(i)` ist sehr langsam
- Verteilte Daten und Speicher-Overhead

(Wird vertieft im Kurs „Algorithmen und Datenstrukturen“.)

Set<T>: HashSet<T>, …

Ungeordnete Sammlung an Elementen

- Keine bestimmte Reihenfolge
- Keine Duplikate

Methoden:

- **`add(T e)`**
  Fügt e dem Set hinzu
- **`addAll(Set<T> set) / addAll(List<T> list)`**
  Fügt alle Einträge aus dem Set/der Liste dem Set hinzu
- **`clear()`**
  Entfernt alle Einträge
- **`remove(T e)`**
  Entfernt den Eintrag e
- **`size()`**
  Anzahl der Elemente

Beispiel (JShell)

```
var set = new HashSet<String>();
set.add("Bonjour");
set.add("Hallo");
set.add("Hello");
set.add("Bonjour");

var size = set.size();
// size ist 3
for(var e : set)
    System.out.println(e);
// Ausgabe: Hallo, Hello, Bonjour
// (Die Reihenfolge ist beliebig)
```

Map<K, V>: HashMap<K, V>, …

Speichert Werte (values) mit einem Schlüssel (keys)

Jeder key kann nur einmal verwendet werden

→ Mehrfachzuweisung überschreibt letzten Wert

Methoden:

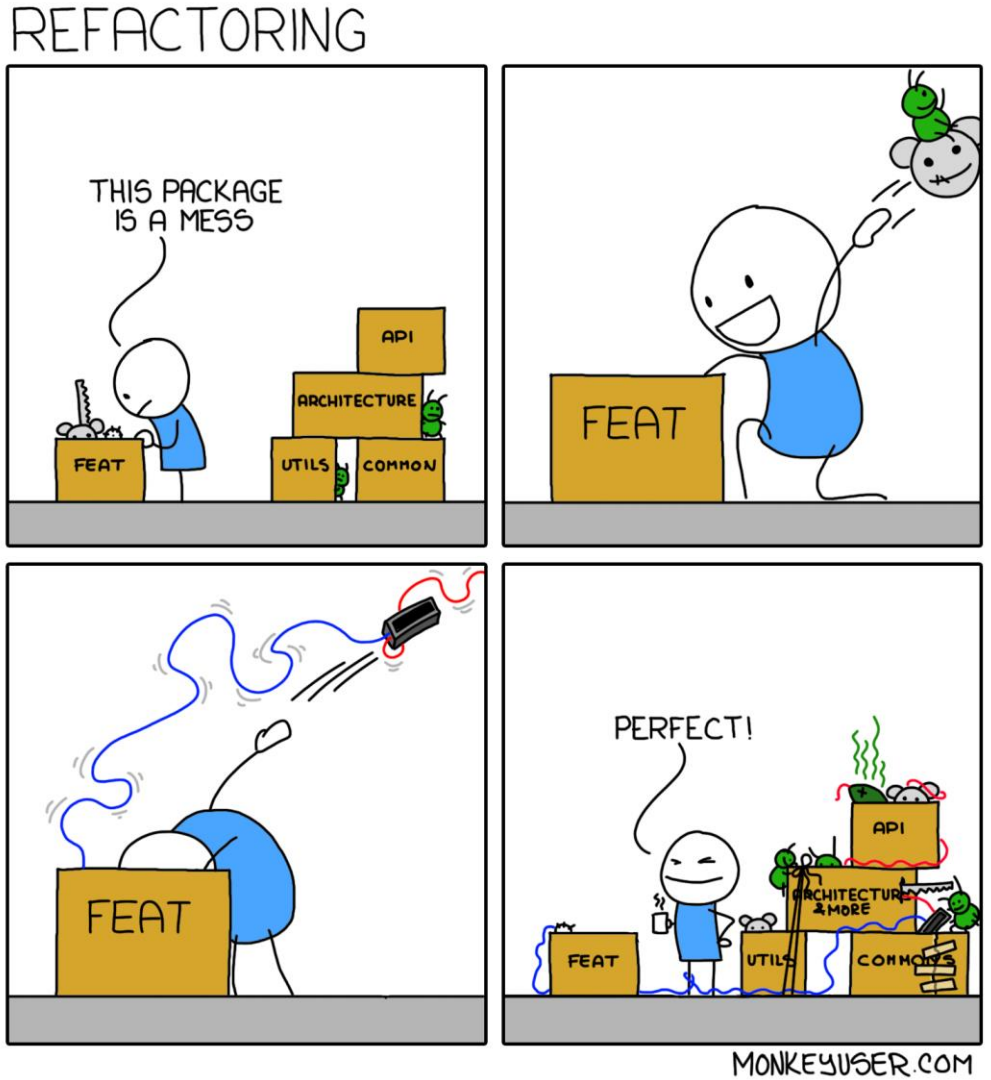
- **`containsKey(K key) / containsValue(V value)`**
  Prüft ob Schlüssel / Wert in der Map existiert
- **`get(k)`**
  Gibt den Wert des Schlüssels zurück (oder null)
- **`size()`**
  Gibt die Anzahl der Schlüssel/Wert Paare zurück
- **`put(K key, V value)`**
  Fügt Wert value mit Schlüssel key hinzu
- **`entrySet() / keySet() / values()`**
  Set mit den Einträgen / Schlüsseln / Werten

Beispiel (JShell)

```
var map = new HashMap<Integer, String>();
map.put(4, "Glumanda");
map.put(10, "Raupy");
map.put(25, "Pikachu");
map.put(52, "Mauzi");

String pokemon1 = map.get(10);
// pokemon1 ==> "Raupy"
String pokemon2 = map.get(151);
// pokemon2 ==> null
for(var e : map.entrySet()) {
    System.out.println(
        e.getKey() + " : " + e.getValue());
}
// Ausgabe in beliebiger Reihenfolge
```

Fragen?



[Bildquelle: https://www.monkeyuser.com/2018/refactoring/]