Propädeutikum
Programmierung in der Bioinformatik

Java – Collections

Thomas Mauermeier

15.01.2019

Ludwig-Maximilians-Universität München

Collections?

Was *ist* eine Collection?

- "Container" für mehrere Objekte des selben Typs
  - *ähnlich* zu Array, funktionieren aber intern anders
  - Collection die ihr schon kennt: `ArrayList`

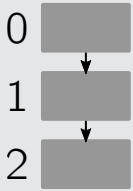
Was *kann* eine Collection?

- Bietet sehr viel mehr als ein einfacher Array:
  - einfaches Bearbeiten, Sortieren, Suchen, etc.
  - spezialisiertere Datenstrukturen (z.B. Mengen, Listen, ...)
  - dynamische Größe: Kein Zwang vorher die Größe zu definieren

Collections!

`List`

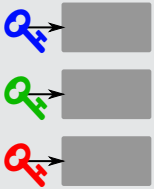
- Referenziert Inhalt mittels Position



- z.B. Klasse `ArrayList`

`Map`

- Referenziert Inhalt mittels “Keys”



- z.B. Klasse `HashMap`

`Set`

- Modelliert mathematische Menge

- z.B. Klasse `HashSet`

Erstellen eines (generischen) Collection-Objekts

Da Collections **generische** Typen sind, haben ihre Konstruktoren eine andere Syntax:

Beispiel: `ArrayList<T>`

Angabe des Typparameters

```
private ArrayList<Integer> myList = new ArrayList<Integer>();

private ArrayList<Integer> myList = new ArrayList<>();
```

Diamantoperator

- **Typparameter**: Typ der Objekte mit denen Collection befüllt werden darf
- **Diamantoperator**: Typparameter darf auf der *rechten* Seite ausgelassen werden

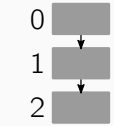
Exkurs: Was sind Generics?

```java
public class NonGenericTupel {
    private Object value1, value2;

    public NonGenericTupel(Object v1,
        ↪ Object v2) {
        this.value1 = v1;
        this.value2 = v2;
    }
    public void setValue1(Object v1) {
        this.value1 = v1;
    }
    // Achtung: Returned immer Object
    // Evtl. Typecasting noetigk
    public Object getValue1() {
        return value1;
    }
}
```

```java
public class GenericTupel<T> {
    private T value1, value2;

    public GenericTupel(T v1, T v2) {
        this.value1 = v1;
        this.value2 = v2;
    }
    public void setValue1(T v1) {
        this.value1 = v1;
    }
    // Bequemer: Returned nun Typ T!
    // Kein Typecasting mehr noetig!
    public T getValue1() {
        return value1;
    }
}
```

List-Klasse: ArrayList<T>

- **Indexbasiert**: Indexierte "Fächer" für Objekte vom Typ `T`
- **Ordered (insertion order)**: Inhalte bleiben am abgelegten Index
- **Duplikate erlaubt**: Gleiche Elemente dürfen mehrfach auftauchen



Methode	Funktion
`add(T e)`	Fügt Element e (vom Typ `T`) ans Ende der Liste an
`remove(Object e)`	Entfernt das **erste** Vorkommen von e aus der Liste
`remove(int i)`	Entfernt das **i**'te Element aus der Liste
`get(int i)`	Returned das **i**'te Element aus der Liste
`contains(Object e)`	Überprüft ob e in der Liste ist
`size()`	Returned die Anzahl der Elemente in der Liste
`isEmpty()`	Überprüft ob die Liste leer ist

$\Rightarrow$ natürlich noch einiges mehr: siehe API

Set-Klasse: `HashSet<T>`

- **Modelliert Menge**: jedes Element darf nur einmal vorkommen
- **Unordered**: Reihenfolge der Elemente kann sich ändern
  - `TreeSet`: geordnet nach *natural order*
  - `LinkedHashSet`: geordnet nach *insertion order*

Methode	Funktion
`add(T e)`	Fügt Element e (vom Typ `T`) zur Menge hinzu
`remove(Object e)`	Entfernt Element e aus der Menge
`contains(Object e)`	Überprüft ob e in der Menge ist
`size()`	Returned die Anzahl der Elemente in der Menge
`isEmpty()`	Überprüft ob die Menge leer ist

$\Rightarrow$ natürlich noch einiges mehr: siehe API

Iterieren über Collections

mit einem **Iterator**:

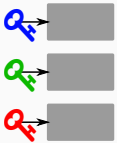
```
public class IteratorDemo {
  public static void main(String[]
      ↪ args) {
    ArrayList<String> names = new
        ↪ ArrayList<>();

    // Namen zu "names" hinzufuegen

    Iterator<String> itr =
        ↪ names.iterator();
    while (itr.hasNext()) {
      String name = itr.next();
      System.out.println(name);
    }
  }
}
```

mit einem **for-each**-Konstrukt:

```
public class ForEachDemo {
  public static void main(String[]
      ↪ args) {
    ArrayList<String> names = new
        ↪ ArrayList<>();

    // Namen zu "names" hinzufuegen

    for (String name : names) {
      System.out.println(name);
    }
  }
}
```

Map-Klasse: `HashMap<K,V>`

- **Key-Value Pairs**: Bildet key-Objekte auf einem value-Objekt ab
  - **Wichtig**: Key muss unique sein!
- **Modelliert Funktion**: sozusagen eine Abbildung $f : K \to V$
- **Unordered**: Reihenfolge der Elemente kann sich ändern
  - `TreeMap`: geordnet nach *natural order*
  - `LinkedHashMap`: geordnet nach *insertion order*



Initialisierung einer `HashMap<K,V>`

```
private HashMap<Integer, String> myMap = new HashMap<>();
                Key      Value
```

Methode	Funktion
`put(K key, V val)`	Assoziiert den Key `key` mit dem Value `val` in der Map
`get(Object key)`	Returned den Value, der mit `key` assoziiert ist
`containsKey(K key)`	Überprüft ob der Key `key` in der Map existiert
`containsValue(V val)`	Überprüft ob ein Key auf den Value `val` abbildet
`size()`	Returned die Anzahl der Key-Value Paare in der Map
`isEmpty()`	Überprüft ob die Map leer ist
`keySet()`	Returned eine `Collection` mit den Keys der Map
`values()`	Returned eine `Collection` mit den Values der Map
`entrySet()`	Returned ein `Set` mit den Key-Value Paaren

$\Rightarrow$ natürlich noch einiges mehr: siehe API

```
public class ColorMap {
   public static void main(String[] args) {
       HashMap<String, String> colorMap = new HashMap<>();
       colorMap.put("black","000000");
       colorMap.put("white","FFFFFF");
       colorMap.put("red","FF0000");
       colorMap.put("magenta","FF00FF");

       colorMap.get("red");
       // returned String "FF0000"
   }
}
```

Bulk Operations

`Collection`-Methoden um diese “in bulk” also in der Masse zu bearbeiten:

Methode	Funktion
`containsAll(Collection c)`	Vergleicht ob `this` alle Elemente aus `c` enthält
`addAll(Collection c)`	Fügt zu `this` alle Elemente aus `c` hinzu
`removeAll(Collection c)`	Analog zu `addAll` aber Elemente werden entfernt
`retainAll(Collection c)`	Behalte in `this` alle Elemente aus `c`, entferne Rest
`clear()`	Gesamte Collection leeren

`HashMap`, `HashSet`... was sind diese Hashes eigentlich?

Hash-Algorithmus

Algorithmus der eine (große) Eingabe **möglichst eindeutig** auf einen **möglichst kleinen** Ausgabewert abbildet

Wozu?

⇓

Hash-basierte Datenstrukturen

Ermöglichen vergleichsweise effiziente Operationen z.B. beim Zugriff oder der Suche
z.B. hashed `HashMap` seine Keys um schnelle Zugriffe auf Values zu ermöglichen

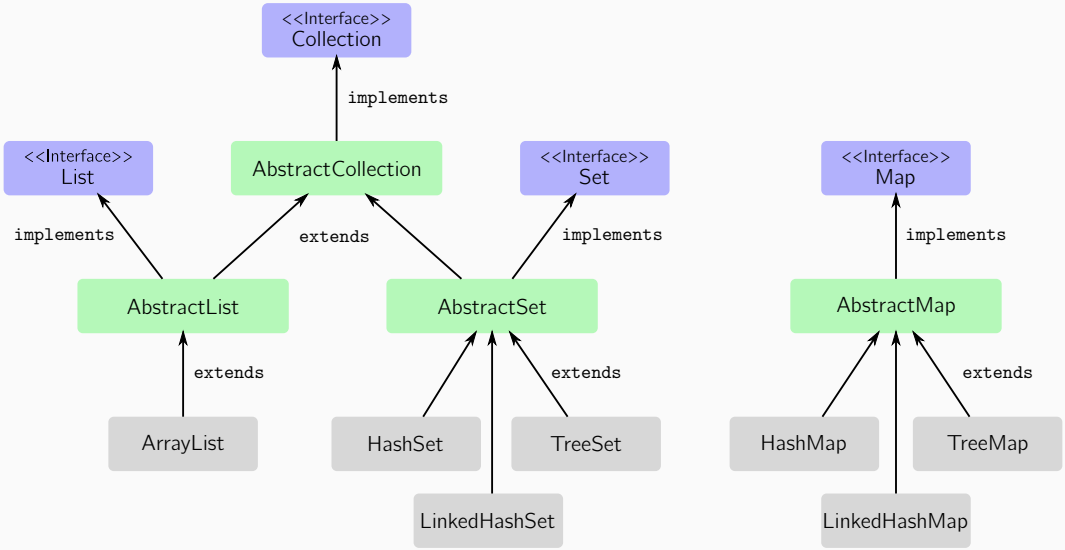
Der `hashCode()`-Contract

- Wenn `hashCode` mehrmals auf ein und dem selben Objekt aufgerufen wird muss es den selben Hash liefern
- Wenn Objekte laut `equals` gleich sind müssen sie den selben Hash liefern
- Wenn Objekte laut `equals` *ungleich* sind müssen sie allerdings nicht *gezwungenermaßen* unterschiedliche Hashes liefern (wäre aber gut)

Liegt Verletzung dieses "Vertrags" vor, sollte `hashCode()` überschrieben werden. z.B.:

```
public class Employee {
    int employeeId;
    String name;
    Department dept;
    @Override
    public int hashCode() {
        // Objects.hash (siehe API) sehr hilfreiche Methode beim Hashing
        return Objects.hash(employeeId, name, dept);
    }
}
```

Vereinfachter Ausschnitt aus der Klassenhierarchie



Exkurs: Interfaces

Interface

- “Alternative” zur Mehrfachvererbung
- Klasse kann also mehrere Interfaces implementieren
  - Keyword: `implements`
- Methoden in Interfaces existieren **nur als Signatur** (implizit abstrakt)
- Interfaces besitzen **keinen** Konstruktor
- Klassen die Interfaces implementieren müssen diese Methoden **überschreiben**

```
public interface Figur {
    public int flaeche();
    public int hoehe();
}
```

```
public class Rchteck implements Figur {
    private int a, b;
    public Rchteck(int a, int b) {
        this.a = a;
        this.b = b;
    }
    @Override
    public int flaeche() {
        return a*b;
    }
    @Override
    public int hoehe() {
        return a;
    }
}
```

```
public class Dreieck implements Figur {
    private int a, b;
    public Dreieck(int a, int b) {
        this.a = a;
        this.b = b;
    }
    @Override
    public int flaeche() {
        return (a*b)/2;
    }
    @Override
    public int hoehe() {
        return (int) sqrt((a*a)+(b*b));
    }
}
```

Exkurs: Abstrakte Klassen und Methoden

Abstrakte Klassen

- Keyword: `abstract`
- Klasse kann nur von **einer** abstrakten Klasse erben
  - Keyword: `extends` (wie normale Vererbung)
- Abstrakte Klassen besitzen **keinen** Konstruktor
- Methoden in abstrakten Klassen können abstrakt **oder** implementiert sein
- Klassen die nicht-abstrakte Methoden erben müssen diese **nicht** überschreiben

Abstrakte Methoden

- Keyword: `abstract`
- Existieren **nur als Signatur** (diesmal explizit abstrakt $\rightarrow$ Keyword)
- Abstrakte Methoden die geerbt werden **müssen** überschrieben werden

```
public abstract class Figur {
    public abstract int flaeche();
    public abstract int hoehe();
    public void eigenschaft() { System.out.println("Ich bin eine Figur"); }
}
```

```
public class Rchteck extends Figur {
    private int a, b;
    public Rchteck(int a, int b) {
        this.a = a;
        this.b = b;
    }
    @Override
    public int flaeche() {
        return a*b;
    }
    @Override
    public int hoehe() {
        eigenschaft();
        return a;
    }
}
```

```
public class Dreieck extends Figur {
    private int a, b;
    public Dreieck(int a, int b) {
        this.a = a;
        this.b = b;
    }
    @Override
    public int flaeche() {
        return (a*b)/2;
    }
    @Override
    public int hoehe() {
        eigenschaft();
        return (int) sqrt((a*a)+(b*b));
    }
}
```

Einfache Sortierung mit dem Interface `Comparable<T>`

- Klasse die `Comparable<T>` implementiert hat eine **natürliche Ordnung**
- Notwendig um `Collection.sort()` zu verwenden
- Überschreiben von `compareTo(T obj)` so dass Return Value folgendes bedeutet:
  - value < 0: `this` kleiner als `obj`
  - value $== 0$: `this` gleich `obj`
  - value < 0: `this` größer als `obj`

```
public class EinObjekt implements Comparable<EinObjekt> {
    final int id;
    public EinObjekt(int id) {
        this.id = id;
    }
    @Override
    public int compareTo(EinObjekt o) {
        return Integer.compare(this.id, o.id);
    }
}
```

```
public class ComparableDemo {
    public static void main(String[] args) {
        ArrayList<EinObjekt> myList = new ArrayList<>();
        // Objekte rueckwaerts hinzufuegen
        for (int i = 14; i >= 0; i--) {
            EinObjekt eo = new EinObjekt(i);
            myList.add(eo);
        }
        for (EinObjekt eo : myList) {
            System.out.print(eo.id + " ");
        }
        // Output: 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
        Collections.sort(myList);
        System.out.println();
        for (EinObjekt eo : myList) {
            System.out.println(eo.id + " ");
        }
        // Output: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
    }
}
```

Abweichende Sortierung: Interface `Comparator<T>`

- Idee: Objekt das man `Collection.sort()` übergibt um Sortierung zu ändern
- D.h. neue Comparator-Klasse schreiben, die `Comparator<T>` implementiert
- Überschreiben von `compare(T obj1, T obj2)`, analog zu `compareTo`

```
public class EinObjektComparator implements Comparator<EinObjekt> {
    boolean desc = false;
    public EinObjektComparator(boolean desc) {
        this.desc = desc;
    }
    @Override
    public int compare(EinObjekt eo1, EinObjekt eo2) {
        if (this.desc) {
            return Integer.compare(eo2.id, eo1.id);
        } else {
            return Integer.compare(eo1.id, eo2.id);
        }
    }
}
```

```
public class ComparableDemo {
    public static void main(String[] args) {
        ArrayList<EinObjekt> myList = new ArrayList<>();
        // Objekte rueckwaerts hinzufuegen
        for (int i = 14; i >= 0; i--) {
            EinObjekt eo = new EinObjekt(i);
            myList.add(eo);
        }
        for (EinObjekt eo : myList) {
            System.out.print(eo.id + " ");
        }
        // Output: 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
        Collections.sort(myList,new EinObjektComparator(false));
        // Output waere: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
        Collections.sort(myList,new EinObjektComparator(true));
        // Output waere: 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
    }
}
```