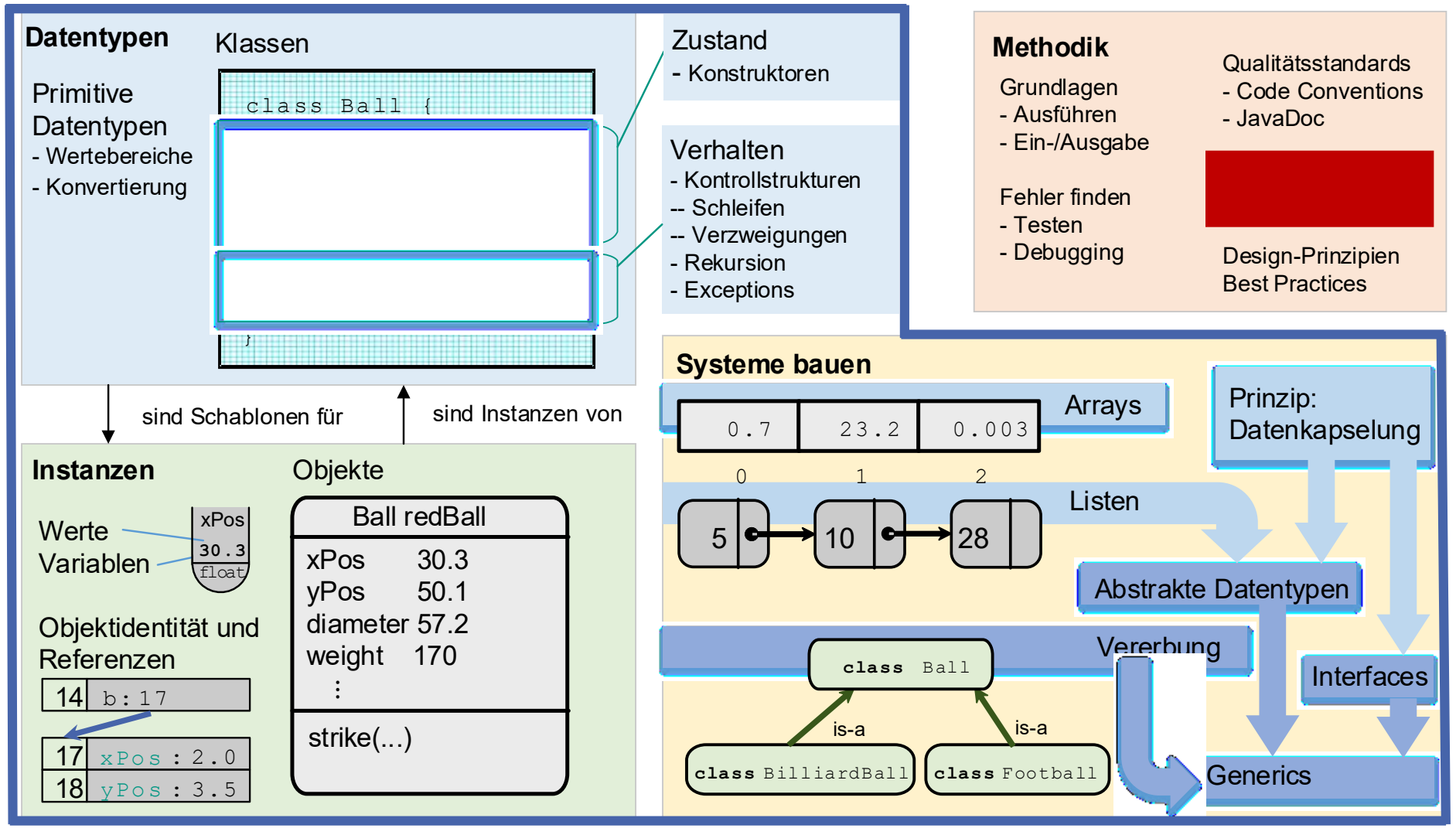
Vorlesung Programmieren

12. Java-API

PD Dr. rer. nat. Robert Heinrich

Vorlesungsüberblick: Objekt-orientiertes Programmieren in Java



Lernziele

Java-API

- Sie wissen, dass Sie in der Java-API häufig benötigte Funktionalität finden
- Sie haben einen Überblick über häufig verwendete und hilfreiche Klassen

Quelle: http://phdcomics.com

Java-API

API: Application Programming Interface

- Sammlung von Klassen / Paketen für häufig benötigte Funktionalität
- »Das Rad nicht immer wieder neu erfinden.«
- Ermöglicht Java-Programmierung auf höherer Ebene

Klassen, die Sie bereits kennen:

- `Object`, `String`, `Math`, `Enum`, `Comparable`, Wrapper-Klassen, ...

Beschreibung/Dokumentation unter https://docs.oracle.com/en/java/javase/11/docs/api/java.base/module-summary.html (Basispakete im Modul java.base) und https://docs.oracle.com/en/java/javase/11/docs/api/index.html (ganze API)

Wichtige Pakete:

- `java.lang` Basisfunktionalität (`Object`, `String`, `Math`, `Enum`, ...)
- `java.util` Java Collections Framework / Zeit- und Datumsfunktionen, ...
- `java.io` Ein- und Ausgabe

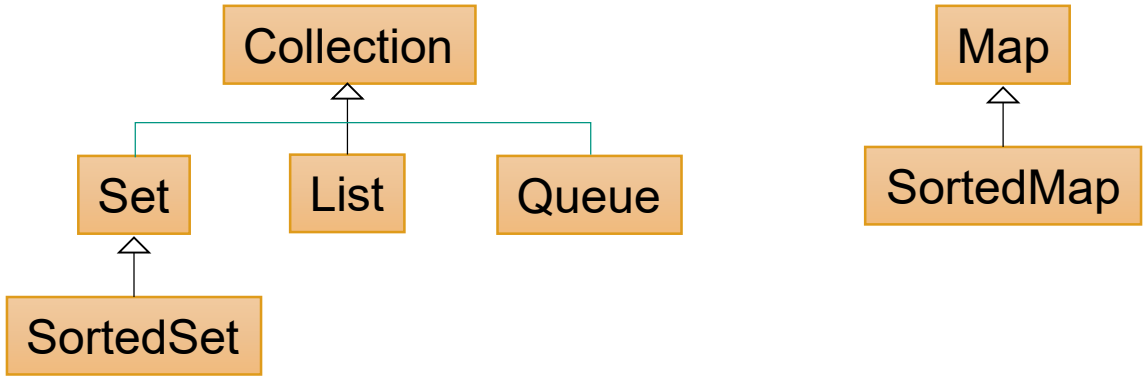
Das Java Collections Framework

Collection = Sammlung von Objekten

Java Collection Framework:

- Einheitliche Architektur zur Repräsentation und Manipulation von Collections
- Abstraktion von der Implementierung
- Verringerung des Programmieraufwands
- 14 generische Interfaces mit jeweils mehreren Implementierungen

Hauptinterfaces:



Das Interface `Collection<E>`

- Alle Collection-Interfaces sind generisch:
  ```
  public interface Collection<E> {…}
  ```
- `Collection<E>` trifft keine Aussage darüber, ob
  - die Elemente der Collection geordnet sind
  - die Collection Duplikate enthält
- `Collection<E>` wird verwendet, wenn möglichst wenig Einschränkungen gelten sollen bzw. bekannt sind.
- `Set`, `List` und `Queue` sind spezifischere Collections

Das Interface `Collection<E>`

```
public interface Collection<E> extends Iterable<E> {
  int size();
  boolean isEmpty();
  boolean contains(Object element);
  boolean add(E element);
  boolean remove(Object element);
  Iterator<E> iterator();
  boolean containsAll(Collection<?> c);
  boolean addAll(Collection<? extends E> c);
  boolean removeAll(Collection<?> c);
  void clear();
  Object[] toArray();
  <T> T[] toArray(T[] a);
  boolean retainAll(Collection<?> c);
  ...
}
```

Sub-Interfaces von `Collection<E>`

`List<E>`

- geordnete Collection; kann Duplikate enthalten; Zugriff auf Elemente mittels Index
- Implementierungen: u.a. `ArrayList`, `LinkedList`

`Set<E>`

- keine Duplikate; modelliert mathematische Menge
- Implementierungen: u.a. `HashSet`, `TreeSet`

`SortedSet<E>`

- keine Duplikate; aufsteigend sortiert
- Implementierungen: u.a. `TreeSet`

Die Klasse `Collections`

Die Klasse `Collections` enthält statische Methoden zum Umgang mit einer `Collection`:

```
public static boolean disjoint(Collection<?> c1, Collection<?> c2){};
// true, falls kein Element sowohl in c1 als auch in c2 enthalten ist
public static int frequency(Collection<?> c, Object o){};
// Anzahl der Elemente in c, die gemäß equals() identisch zu o sind
public static void reverse(List<?> l){};
// Kehrt die Reihenfolge der Elemente in l um
public static <T> boolean replaceAll(List<T> l, T oldV, T newV){};
// Ersetzt jedes zu oldV identische Element in l durch newV
public static <T extends Comparable<? super T>> void sort(List<T> l){};
// Sortiert l gemäß der Methode compareTo des Typs T
public static <T> void sort(List<T> list, Comparator<? super T> c){};
// Sortiert l gemäß Comparator c
```

Das Interface `Map<K,V>`

Modelliert eine mathematische Funktion von K nach V (K: Schlüssel, V: Werte)

Funktionseigenschaft (linkstotal und rechtseindeutig):

- Eine Map enthält keinen Schlüssel mehrmals
- Jeder Schlüssel bildet auf höchstens einen Wert ab

```
public interface Map<K,V> {
  V put(K key, V value);
  V get(Object key);
  V remove(Object key);
  boolean containsKey(Object key);
  boolean containsValue(Object value);
  int size();
  boolean isEmpty();
  void putAll(Map<? extends K, ? extends V> m);
  void clear();
  public Set<K> keySet();
  public Collection<V> values();
}
```

Map<K,V>: Beispiel

Map<K,V> wird unter anderem von TreeMap<K,V> implementiert

Beispiel:

```
TreeMap<String,Integer> mountains = new TreeMap<String,Integer>();
mountains.put("Mount Everest", 8848);
mountains.put("K2", 8611);
mountains.put("Lhotse", 8516);

System.out.println(mountains.containsKey("Lhotse")); // Ausgabe: true
System.out.println(mountains.get("K2"));             // Ausgabe: 8611
System.out.println(mountains.get("Zugspitze"));      // Ausgabe: null

mountains.remove("K2");

System.out.println(mountains.size());            // Ausgabe: 2
System.out.println(mountains.values());          // Ausgabe: [8848, 8516]
```

Das Paket `java.io`

`java.io` enthält Klassen zur Ein- und Ausgabe durch Datenströme und Dateien

Die wichtigsten Klassen zur Ein- und Ausgabe sind:

- Byte-basiert (*byte streams*):
  - `InputStream` und `OutputStream`
  - `FileInputStream` und `FileOutputStream`
- Zeichen-basiert (*character streams*):
  - `Reader` und `Writer`
  - `InputStreamReader` und `OutputStreamWriter`
  - `FileReader` und `FileWriter`
- Mit Puffer (Zwischenspeicher):
  - `BufferedInputStream` und `BufferedOutputStream`
  - `BufferedReader` und `BufferedWriter`

Ein- und Ausgabe in Java

Beispiel 1: Lesen von Zeichen der Standardeingabe in einen String

```
String s = new String();
InputStreamReader isr = new InputStreamReader(System.in);
int c = isr.read();
while (c != -1) {
  s += (char) c;
  c = isr.read();
}
```

Beispiel 2: Schreiben von Zeichen eines Strings auf die Standardausgabe

```
OutputStreamWriter osw = new OutputStreamWriter(System.out);
for (int i = 0; i < s.length(); i++) {
  osw.write(s.charAt(i));
}
osr.flush();
```

Ein- und Ausgabe in Java

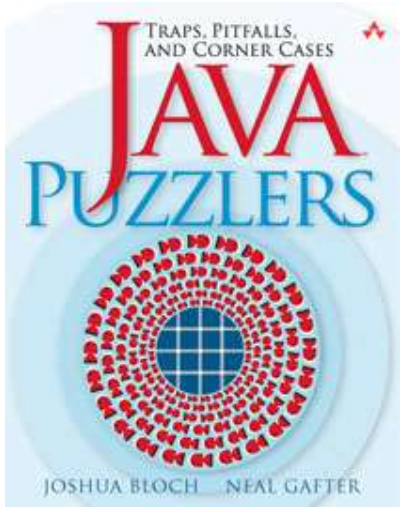
Die Klassen in `java.io` realisieren das Entwurfsmuster *Decorator*, d.h. sie ermöglichen es, zusätzliche Funktionalität zur Laufzeit zu einzelnen Objekten hinzuzufügen

Beispiel:

```
OutputStream os = new FileOutputStream("file.txt");
// Byte-weises schreiben in Datei
OutputStream bos = new BufferedOutputStream(os);
// … mit Ausgabepuffer
OutputStream dbos = new DeflaterOutputStream(bos);
// … und mit Kompression der Ausgabe (ZIP)
```

Java Puzzlers

Bloch, Joshua, and Neal Gafter. *Java puzzlers: traps, pitfalls, and corner cases*. Pearson Education, 2005.



Video: Java Puzzlers

- URL: http://www.youtube.com/watch?v=V1vQf4qyMXg&t=4m14s
- Eine der Botschaften: *Strange and terrible methods lurk in the libraries… beware!*
- Nicht einfach API-Methoden verwenden, die vom Namen her zu passen scheinen, sondern Dokumentation lesen

Zusammenfassung

API: Application Programming Interface

- Sammlung von Klassen / Paketen für häufig benötigte Funktionalität
- Ermöglicht Java-Programmierung auf höherer Ebene

Häufig verwendete und hilfreiche Klassen

- `Object`, `String`, `Math`, `Enum`, `Comparable`, Wrapper-Klassen, ...
- Java Collections

Beschreibung / Dokumentation unter https://docs.oracle.com/en/java/javase/11/docs/api/java.base/module-summary.html (Basispakete im Modul java.base) und https://docs.oracle.com/en/java/javase/11/docs/api/index.html (ganze API)

Wichtige Pakete:

- `java.lang` Basisfunktionalität (`Object`, `String`, `Math`, `Enum`, ...)
- `java.util` Java Collections Framework / Zeit- und Datumsfunktionen, ...
- `java.io` Ein- und Ausgabe