UNIKASSEL
VERSITÄT

SOFTWARE ENGINEERING
UNIVERSITÄT KASSEL

Vom Modell zum Code: Implementierungsdetails

Jens Kosiol

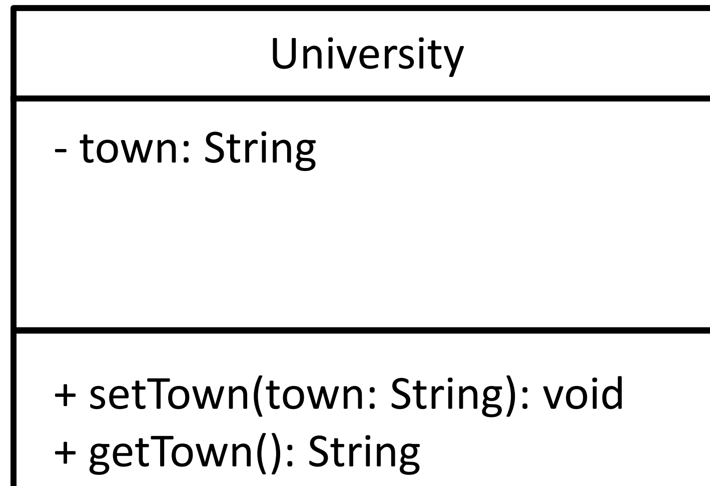
Wintersemester 23/24

(Foliensatz basiert teilweise auf Folien von Prof. Dr. Gabriele Taentzer)

Überblick

- Datenkapselung – Getter- und Setter-Methoden
- Das Java Collection Framework
- Referentielle Integrität

Datenzugriff – Getter und Setter

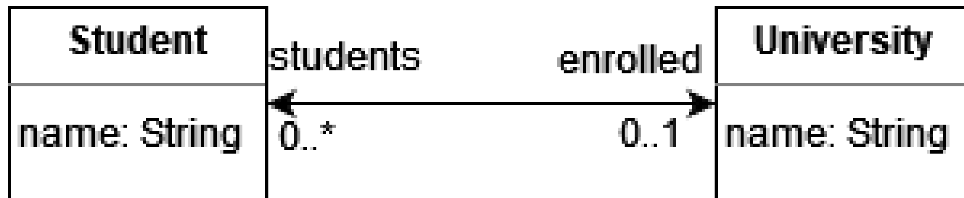



```
class University {
   private String town;

   public void  setTown(String town) {
     if (town == null){
       throw new IllegalArgumentException(
          "University town must not be null!");
     }
     this.town = town;
   }

   public String getTown() {
     return town;
   }
}
```

UNIKASSEL VERSITÄT

SOFTWARE ENGINEERING
UNIVERSITÄT KASSEL

Implementierung von Assoziationen (grundlegend)



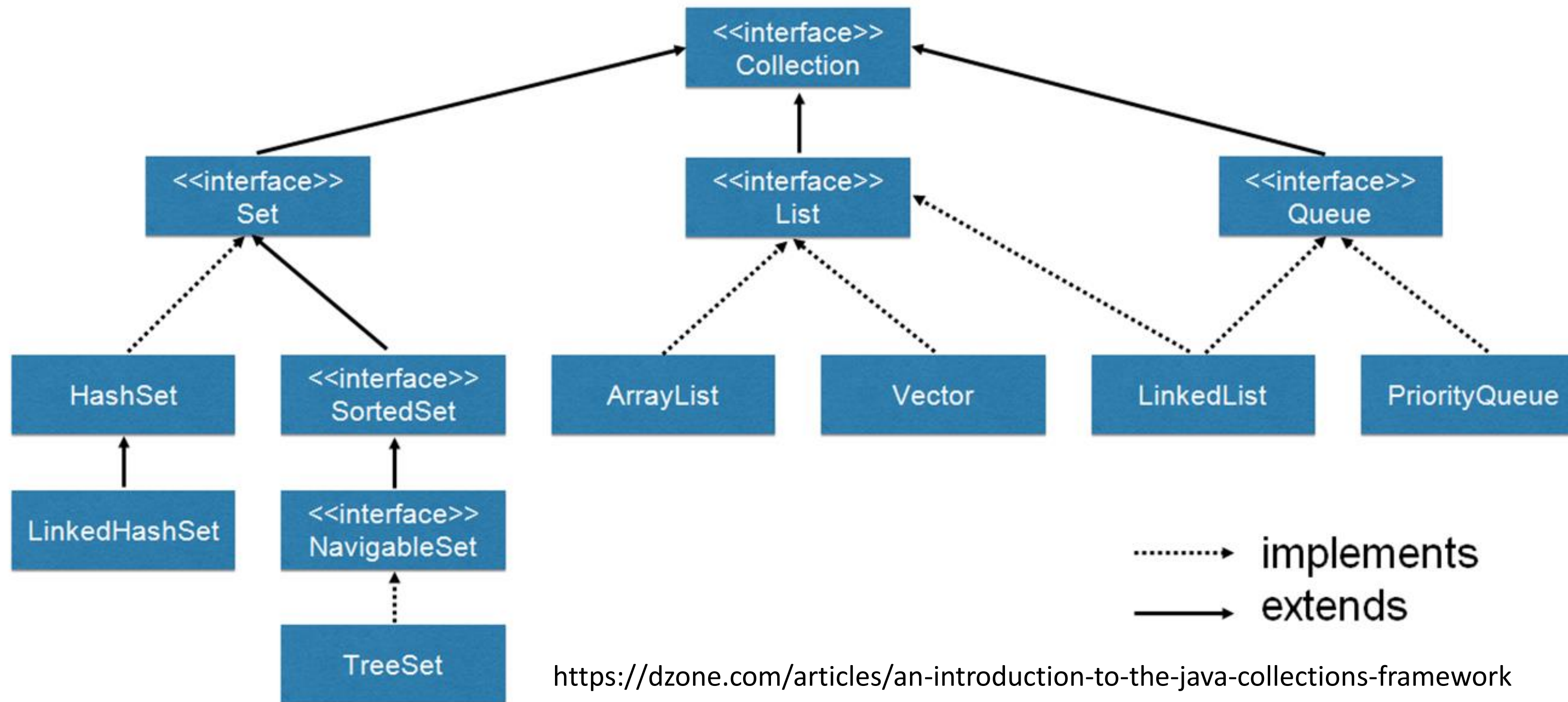

```
import java.util.Collection;
import java.util.ArrayList;

public class Student {
   private String name;
   private University enrolled;
}

public class University {
   private String name;
   private Collection<Student> students
      = new ArrayList<Student>();
}
```

Das Java Collections Framework

Das Java Collections Framework (JCF) bietet Implementierungen der gängigsten (zusammengesetzten) Datenstrukturen.

- Liegt im Paket `java.util`
- Zweck:
  - Reduzierter Programmieraufwand, da die gängigen Datenstrukturen bereits implementiert sind
  - Gute Performanz, da bei der Implementierung in JCF auf Performanz besonders geachtet wurde
- Datenstrukturen für die Gruppierung von Objekten:
  - `Collection` für Objektbehälter
  - Map für Zuordnungen von Schlüsseln zu Werten

Collection Schnittstellenklassen (Ausschnitt)



https://dzone.com/articles/an-introduction-to-the-java-collections-framework

Collections Schnittstellenklassen

Eine Collection ist ein Behälter für Objekte. Die im Folgenden aufgeführten Schnittstellen Set, SortedSet und List sind von Collection abgeleitet.

- Set: Ein Behälter für Objekte mit Mengencharakter. Duplikate sind nicht erlaubt.
- SortedSet: Analog zu Set mit dem Unterschied, dass die Elemente sortiert werden (in Reihenfolge der Einfügung).
- List: Ein Objektbehälter, in dem Objekte über Integer-Werte indiziert werden. Duplikate sind erlaubt. Ein neues Element kann an eine beliebige Stelle in einer List positioniert werden.
- Queue: Ein Objektbehälter, in den immer vorn oder immer hinten eingefügt und vorn herausgenommen wird (LIFO (Stack) und FIFO (Queue)).

Welche Implementierungsklassen gibt es?

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

[Quelle: https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html]

Implementierungsstrukturen

- `Array`…: größenveränderlicher Array
  - Operationen `size`, `isEmpty`, `get`, `set` laufen in konstanter Zeit.
  - Hinzufügen (add) von Elementen läuft in amortisiert konstanter Zeit (d.h., das Hinzufügen von n Elementen ist in $O(n)$).
- `Linked`…: Elemente intern (doppelt) verlinkt
  - Operationen `get, set` und add laufen in linearer Zeit.
  - Lokales Entfernen und Hinzufügen von Elementen (während einer Iteration) funktioniert in konstanter Zeit.
- `Hash`…: benutzt intern eine Hashing-Struktur
  - ca. gleiche Zeit zum Auffinden jedes Elements
- `Tree`…: benutzt intern eine Rot-Schwarz-Baum Struktur
  - Operationen `get`, `put` und `remove` laufen in logarithmischer Zeit.
  - Benötigt Ordnungsstrukur auf den Elementen.

Zentrale Operationen einer Collection

Typ: `Collection<E>` – iterierbare Ansammlung von Objekten vom Typ E

- `boolean add(E e)` – fügt Element e hinzu (für Arrays optional mit Index)
- `boolean contains(E e)` – prüft, ob Element e enthalten ist
- `boolean remove(E e)` – entfernt Element e
- `Iterator<E> iterator()` – gibt einen Iterator auf die Collection zurück

Initialisierung einer Collection

In der Deklaration einen möglichst abstrakten (Interface-)Typen verwenden

Konkrete (Klassen-)Typen nur in der Erzeugung von Objekten (rechte Seite von Zuweisungen) verwenden

```
Collection<String> names = new HashSet<String>();
Queue<String> tasks = new PriorityQueue<String>();
```

Iteration über Collections per Foreach-Schleife

- Über Arrays und Collections kann per Foreach-Schleife iteriert werden
- Man erhält Zugriff auf die einzelnen Elemente, aber keinen Zugriff auf das Array/die Collection selbst!

```
Collection<Integer> intSet = new HashSet<Integer>();
intSet.add(1);
intSet.add(2);
  for (int i:intSet)
    System.out.println(i);
```

Gleichzeitiges Iterieren und Ändern per Iterator

Für jede Collection kann man sich eine Instanz der Klasse Iterator erstellen.

- Ein Iterator bietet die Möglichkeit, eine Collection zu durchlaufen, bietet aber keine Garantie für die Reihenfolge, in der durchlaufen wird.
- Ein Iterator bietet die Methoden `hasNext()`, `next()`, `remove()` und `forEachRemaining()` an.

```
Collection<Integer> intSet = new HashSet<Integer>();
intSet.add(1);
intSet.add(2);
Iterator it = intSet.iterator();
while (it.hasNext()) {
  if (((int) it.next() % 2) == 1)
    it.remove();
}
```

Iterieren über eine Menge von Integern und Entfernen der ungeraden.

Aufbauen einer Liste

```
// Initialisieren und über Operation add aufbauen
List<Integer> numberList1 = new ArrayList<Integer>();
numberList1.add(1);
numberList1.add(2);

// Liste fester Größe
List<Integer> numberList2 = new Arrays.asList(1,2,3);

// Liste mit dynamischer Größe
List<Integer> numberList3 = new ArrayList<Integer>(Arrays.asList(1,2,3));

// Factory Methode der Listenschnittstelle (aber immutable)
List<Integer> numberList4 = List.of(1,2,3,4);
```

Iterieren über eine Liste per ListIterator

Die Listenschnittstelle bietet zusätzlich die Möglichkeit, einen `ListIterator` zu instanziieren.

- Während ein `ListIterator` eine Liste durchläuft, ist seine Position immer *zwischen* zwei Elementen.
- Zusätzliche Methoden (teilweise optional)
  - `hasPrevious()` und `previous()`: Unterstützung für Navigation durch Liste in umgekehrter Reihenfolge
  - `add(E e)`: Fügt das neue Element e zwischen `previous()` und `next()` ein (optional)
  - `nextIndex()` und `previousIndex()`: Zugriff auf aktuelle Position in der Liste
  - `set(E e)`: Ersetzt das Element, das zuletzt durch `next()` oder `previous()` ausgegeben wurde, durch e (optional)

Unterstützung für Lambda-Ausdrücke

In neueren Versionen von Java gibt es immer mehr Funktionen, die Lambda-Ausdrücke als Eingabewerte unterstützen:

```
Collection<Integer> intSet = new HashSet<Integer>();
intSet.add(1);
intSet.add(2);
intSet.removeIf(i -> ((int) i % 2) == 1);
```

`removeIf` als kompaktere Syntax zum Entfernen von Elementen

```
Collection<Integer> intSet = new HashSet<Integer>();
intSet.add(1);
intSet.add(2);
Iterator it = intSet.iterator();
it.forEachRemaining(System.out::println);
```

`forEachRemaining` unterstützt Lambda-Ausdruck als Eingabewert (Methodenreferenz :: statt `i -> System.out.println(i)`)

Datenzugriff Collections

```
import java.util.Collection;
import java.util.ArrayList;

public class University {
   private Collection<Student> students
      = new ArrayList<Student>();

   public void addStudent (Student student) {
      students.add(student);
    }

   public void removeStudent(Student student) {
      students.remove(student);
   }

   public ArrayList<Student> getStudents() {
      return students;
   }
}
```

Verhindert diese Implementierung unkontrollierten Zugriff auf die Liste der Studenten?

```
University sRUni = new University();
// …
sRUni.getStudents().clear();
```

Der Rückgabewert von `getStudents` ist vom Typ `Collection<Student>` mit öffentlichen Methoden `add`, `clear`, `remove`, …

Datenzugriff Collections – Alternative

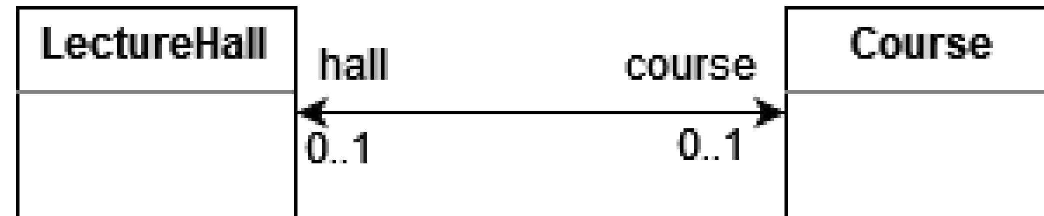
```
import java.util.Collection;
import java.util.Collections;
import java.util.ArrayList;

public class University {
    private Collection<Student> students
        = new ArrayList<Student>();

    public boolean addStudent (Student student) {
        // ggf. Parameterprüfung
        return students.add(student);
     }

    public boolean removeStudent(Student student) {
        return students.remove(student);
    }

    public List<Student> getStudents() {
        return Collections.unmodifiableList(students);
    }
}
```

Die Methode `unmodifiableList` aus der Klasse `Collections` gibt eine nicht bearbeitbare Sicht der Liste, auf der sie aufgerufen wird, zurück.

U N I K A S S E L
V E R S I T Ä T

SOFTWARE ENGINEERING
UNIVERSITÄT KASSEL

Referentielle Integrität




```java
public class LectureHall {
  private Course course;
  public void setCourse(Course course) {
     this.course = course;
  }
  public Course getCourse() {
     return course;
  }
}
```

```java
public class Course {
  private LectureHall hall;
  public void setLectureHall(LectureHall hall) {
     this.hall = hall;
  }
  public LectureHall getLectureHall() {
     return hall;
  }
}
```

Was kann schiefgehen?

- `course.hall` verweist nach Aufruf von `setCourse` nicht unbedingt auf die `LectureHall` auf der `setCourse` aufgerufen wurde (und analog für `hall.course`).
- Der ehemalige `course` der `LectureHall`, auf der `setCourse` aufgerufen wurde, verweist noch auf diese `LectureHall` (und analog für `hall` und `Course`).

Referentielle Integrität

```
public class LectureHall {
  private Course course;
  public void setCourse(Course course) {
    Course oldValue = this.course;
    this.course = course;
    oldValue.setLectureHall(null);
    course.setLectureHall(this);
  }
  public Course getCourse() {
    return course;
  }
}
```

```
public class Course {
  private LectureHall hall;
  public void setLectureHall(LectureHall hall) {
    LectureHall oldValue = this.hall;
    this.hall = hall;
    oldValue.setCourse(null);
    hall.setCourse(this);
  }
  public LectureHall getLectureHall() {
    return hall;
  }
}
```

Was geht hier noch schief?

Referentielle Integrität

```java
public class LectureHall {
  private Course course;
  public void setCourse(Course course) {
    if (this.course != course) {
      Course oldValue = this.course;
      this.course = course;
      oldValue.setLectureHall(null);
      course.setLectureHall(this);
    }
  }
  public Course getCourse() {
    return course;
  }
}
```

```java
public class Course {
  private LectureHall hall;
  public void setLectureHall(LectureHall hall) {
    if (this.hall != hall) {
      LectureHall oldValue = this.hall;
      this.hall = hall;
      oldValue.setCourse(null);
      hall.setCourse(this);
    }
  }
  public LectureHall getLectureHall() {
    return hall;
  }
}
```

Was kann hier noch schiefgehen?

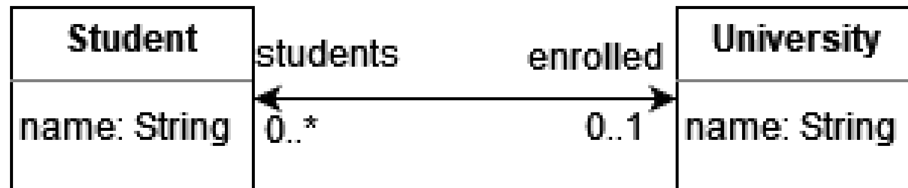
UNIKASSEL
VERSITÄT

Referentielle Integrität

```
public class LectureHall {
  private Course course;
  public void setCourse(Course course) {
     if (this.course != course) {
        Course oldValue = this.course;
        this.course = course;
        if (oldValue != null) {
           oldValue.setLectureHall(null);
        }
        if (course != null) {
           course.setLectureHall(this);
        }
     }
  }
  public Course getCourse() {
     return course;
  }
}
```

```
public class Course {
  private LectureHall hall;
  public void setLectureHall(LectureHall hall) {
     if (this.hall != hall) {
        LectureHall oldValue = this.hall;
        this.hall = hall;
        if (oldValue != null) {
           oldValue.setCourse(null);
        }
        if (hall != null) {
           hall.setCourse(this);
        }
     }
  }
  public LectureHall getLectureHall() {
     return hall;
  }
}
```

Referentielle Integrität II




```java
public class Student {

   private University enrolled;

   public void setUniversity(University university) {
      if (this.enrolled != university) {
         University oldValue = this.enrolled;
         this.enrolled = university;
         if (oldValue != null) {
            oldValue.removeStudent(this);
         }
         if (university != null) {
            university.addStudent(this);
         }
      }
   }

   public University getUniversity() {return university};
}
```

```java
public class University{

   private Set<Student> students = new LinkedHashSet<Student> ();

   public boolean addStudent(Student student) {
      boolean changed = students.add(student);
      if (changed && student != null) {
         student.setUniversity(this);
      }
      return changed;
   }

   public boolean removeStudent(Student student) {
      boolean changed = students.remove(student);
      if (changed && student != null) {
         student.setUniversity(null);
      }
      return changed;
   }

   public List<Student> getStudents() {
      return Collections.unmodifiableSet(students);
   }
}
```