Inhaltsverzeichnis

1. Kurstag 2 - Basics 2/2

Allgemeine Inhalte

- ▢ Methoden überladen, überschreiben/übersteuern
- ▢ Collections API
- ▢ Beziehungsarten (Aufruf, Vererbung, Assoziation, ...)
- ▢ Assoziationen (Aggregation vs. Komposition im Code)

Fachlicher Kontext

Mögliche Assoziationen/Relationen mit Kardinalitäten

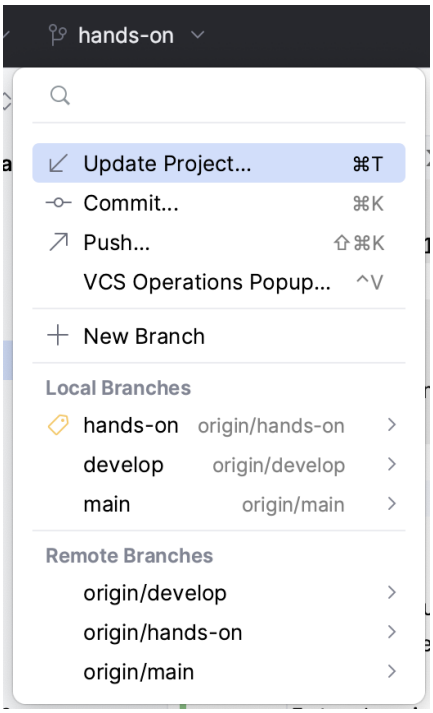
```
+ Zug          1:n      Wagon
+ Zug          1:1      Lokomotive
+ Zug          n:m      Soll-Fahrplan
 (=> Zug 1:n Fahrt 1:n Soll-Fahrplan)

+ Zug          n:m      Strecke
+ Zug          1:n      Fahrt
+ Strecke      1:n      Fahrt
 (=> Zug 1:n Fahrt n:1 Strecke)

+ Fahrt        1:n      Reisende
+ Zug          1:2..3   Flügel
+ Streckennetz 1:n      Strecke
+ Strecke      n:m      Abschnitte
```

1.1. UPDATE

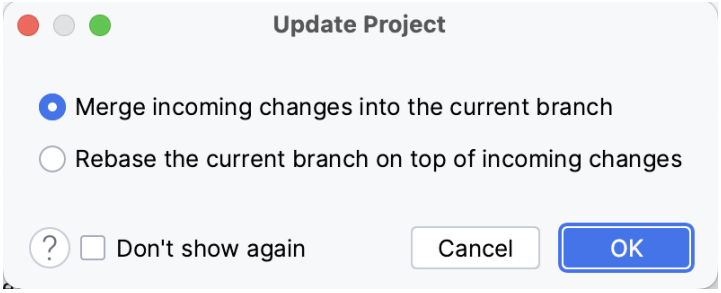
Das **Projekt** bzw. der "*lokale Workspace*", d.h. euer lokales Arbeitsverzeichnis, in dem alle Sourcen liegen, muss als allererstes zum Start in den Tag aktualisiert werden, d.h. ...

→ *"Update Project"*

Das geht am besten mithilfe der IDE im Menü oder über das GIT Icon:



Danach muss - im sich öffnenden Dialog - noch folgendes bestätigt werden: *merge incoming changes into the current branch*



Der Vorgang sollte mit einer Erfolgsmeldung abschließen.

1.2. Recap

Was bisher geschah ...

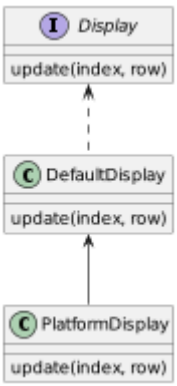
- [ ] Klassen & Objekte
- [ ] Objektvertrag (hashCode, equals)
- [ ] Vererbung & Interfaces
- [ ] Abstrakte Klassen
- [ ] Access Modifier (Sichtbarkeiten, Scopes)

1.3. Überladen & Überschreiben

Beispiel Überladen von Methoden

JAVA

```java
public class PlatformDisplay {

    // multiple methods can be used to
    // update a platforms' display

    public void update() {}

    public void update(String line) {}

    public void update(List<String> lines) {}

}
```



Beispiel Übersteuern von Methoden

JAVA

```java
public interface Display {

    void update(int index, String row);
```

Die zugehörige Annotation im Code ist

```
@Override
```

Sie sollte in jedem Fall genutzt werden, zudem wird sie auch von der IDE vorgeschlagen

Die zugehörigen **Unit-Tests** finden sich hier:

```
/test/de/dhbw/course2/basics/BasicsTest
  .canOverloadMethods()
  .canOverrideMethods1()
  .canOverrideMethods2()
```

1.4. Datentypen

Übersicht über die Datentypen in Java:

Typname	Größe[1]	Wrapper-Klasse	Wertebereich	Beschreibung
boolean	undefiniert[2]	java.lang.Boolean	true / false	Boolescher Wahrheitswert, Boolescher Typ[3]
char	16 bit	java.lang.Character	0 ... 65.535 (z. B. 'A')	Unicode-Zeichen (UTF-16)
byte	8 bit	java.lang.Byte	-128 ... 127	Zweierkomplement-Wert
short	16 bit	java.lang.Short	-32.768 ... 32.767	Zweierkomplement-Wert
int	32 bit	java.lang.Integer	-2.147.483.648 ... 2.147.483.647	Zweierkomplement-Wert
long	64 bit	java.lang.Long	-2^{63} bis $2^{63}-1$, ab Java 8 auch 0 bis $2^{64}-1$[4]	Zweierkomplement-Wert
float	32 bit	java.lang.Float	+/-1,4E-45 ... +/-3,4E+38	32-bit IEEE 754, es wird empfohlen, diesen Wert nicht für Programme zu verwenden, die sehr genau rechnen müssen.
double	64 bit	java.lang.Double	+/-4,9E-324 ... +/-1,7E+308	64-bit IEEE 754, doppelte Genauigkeit

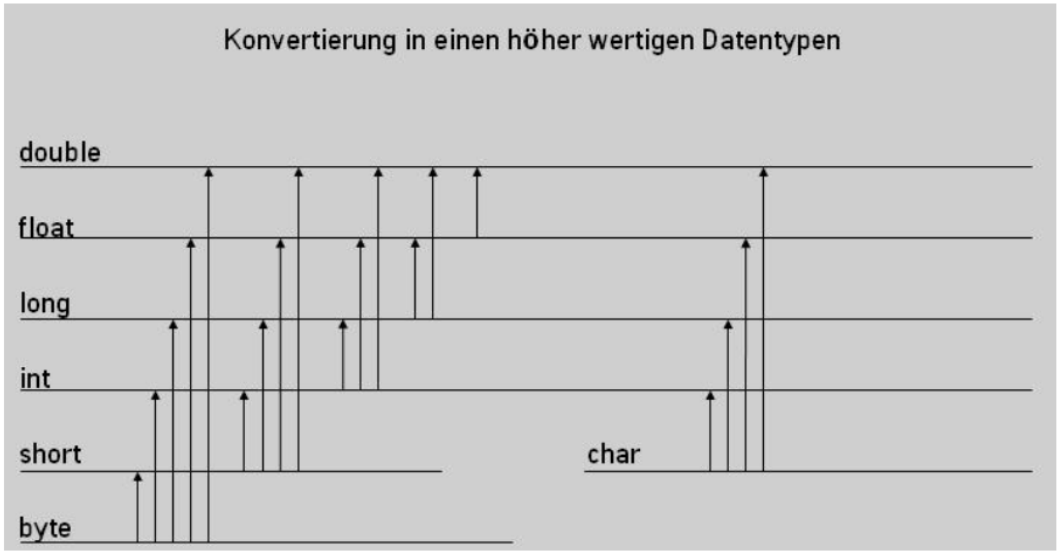
Quelle: → Wikibooks: Datatypes (https://de.wikibooks.org/wiki/Java_Standard:_Primitive_Datentypen)

Quelle: → Oracle: Datatypes (https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html)

1.5. Typumwandlungen

Type-Casting mit primitiven Datentypen

Man unterscheidet zwischen einer **expliziten** und einer **impliziten** Typumwandlung.



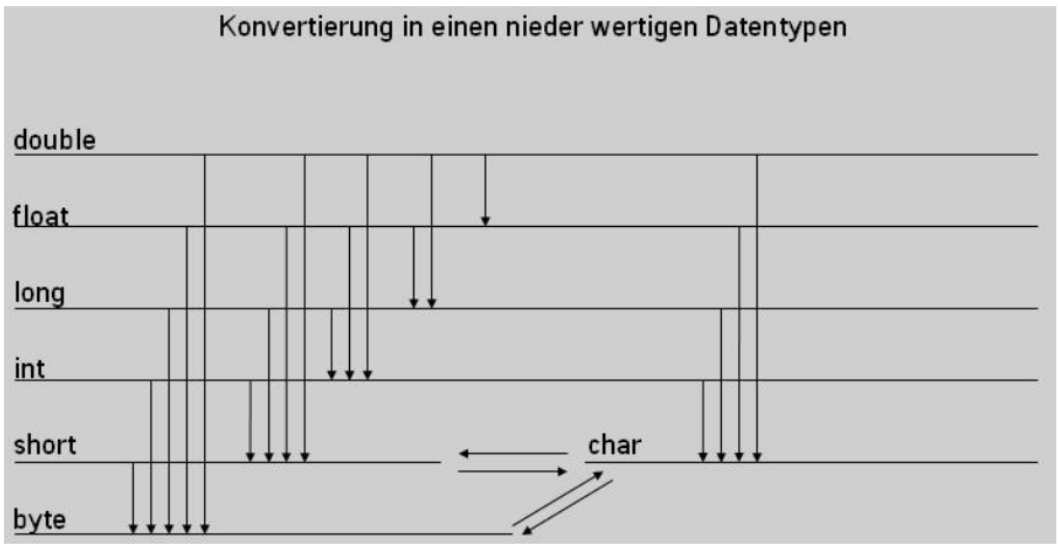


Figure 1. Widening & Narrowing

> *Bei der Umwandlung in "kleinere" Datentypen können Fehler auftreten (Informationsverlust), es findet das sogenannte "Narrowing" automatisch statt. Es entsteht zwar kein Compiler-Fehler, aber z.B. die Zahl wird verfälscht (→ Narrowing (https://docs.oracle.com/javase/specs/jls/se17/html/jls-5.html#jls-5.1.3))*

1.5.1. Implizite Typumwandlung

Die implizite Typumwandlung findet automatisch bei der Zuweisung statt. Dies geht jedoch nur, wenn ein niederwertiger Datentyp in einen höher wertigeren Datentypen umgewandelt wird, also z.B. vom Datentyp `int` in den Datentyp `long`:

```java
int wert1 = 10;
long wert2 = 30;
wert2 = wert1; // automatische Umwandlung
```

1.5.2. Explizite Typumwandlung

Die explizite Umwandlung erfolgt durch den sogenannten `cast`-Operator mit runden Klammern. Hier wird von einem höher wertigeren Datentyp in einen nieder wertigen Datentypen umgewandelt. In welchen Datentyp umgewandelt werden soll, muss bei dem cast Operator explizit angegeben werden.

```java
int wert1 = 10;
float wert2 = 30.5f;
wert1 = (int) wert2; // Umwandlung per 'cast'
```

→ Übung 2

1.6. Statische Klassen & Methoden

Das Schlüsselwort `static` bedeutet im Wesentlichen, das die Attribute oder Methoden nicht an die Erzeugung einer **Instanz** gebunden sind.

Siehe dazu den **Unit-Test** hier

```
src/test/java/de/dhbw/course2/basics/StaticTest.java
```

1.7. Schleifen

Es gibt eine Reihe von Möglichkeiten, über die Einträge von Listen oder Maps zu iterieren. Dazu werden **Schleifen** eingesetzt.

Es drei relevante Schleifen-Typen. Je nach Einsatzszenario entscheidet man sich für eine der drei Schleifen:

- For-(Each)-Schleife
- While-Schleife
- Do-While-Schleife

Hier ein Beispiel für eine häufig genutzte Schleife, die `for-each` Loop:

siehe → de.dhbw.course2.basics.BasicsTest.canLoopWithForEach()

```java
    public void canLoopWithForEach() {
        // given
        List<String> list = List.of("a", "b", "c", "d");

        // when
        for (String item : list) {
            System.out.printf(
                    "Item '%s' at position '%s' \n",
                    item,
                    list.indexOf(item));
        }

        // then
        assertEquals(4, list.size());
    }
```

1.8. Collections Framework

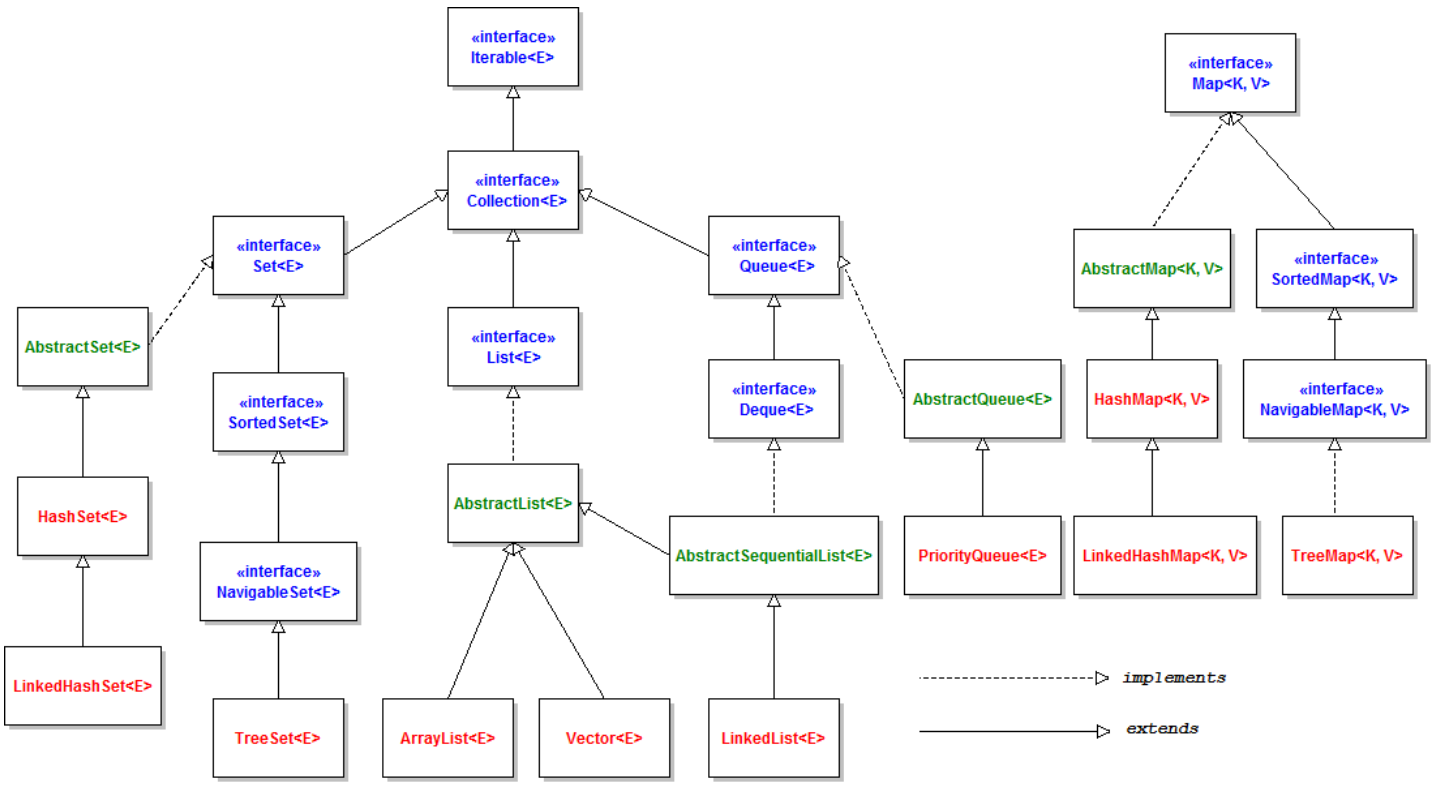
Siehe z.B. → Java Collections Framework auf Wikipedia (https://en.wikipedia.org/wiki/Java_collections_framework)

Die wichtigsten Auswahlkriterien:

- Position von Elementen (*numerischer index*)
- Möglichkeit zur Ordnung von Elementen (z.B. *insertion order*)
- Möglichkeit von `null` Elementen oder Duplikaten (*Set vs. List*)

> *→ Duplikate in Listen? Wann ist ein Listenelement ein Duplikat? Technisch gleich oder logisch gleich?*

- Zugriff auf Elemente anhand eines Schlüssels (*List vs. Map*)



1.8.1. Java Collections API Interfaces

Java collection interfaces are the foundation of the Java Collections Framework. All core collection interfaces are generic. For example `public interface Collection<E>`. The <E> syntax is for Generics and when we declare Collection (*später mehr zum Thema `Generics`*)

1. **Collection interface**. This is the root of the collection hierarchy. A collection represents a group of objects known as its elements. Some basic operations are provided.
2. **Iterator Interface**. Iterator interface provides methods to iterate over the elements of the Collection. Iterators allow the caller to remove elements from the underlying collection during the iteration.
3. **Set Interface** Set is a collection that cannot contain duplicate elements. The Java platform contains three general-purpose Set implementations: HashSet, TreeSet, and LinkedHashSet.
4. **List Interface** List is an ordered collection and can contain duplicate elements. You can access any element by its index. List has a dynamic length. ArrayList and LinkedList are implementation classes of List interface.
5. **Queue Interface** Queue is a collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations. Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner.

6. **Dequeue Interface** A linear collection that supports element insertion and removal at both ends. The name deque is short for "double-ended queue"
7. **Map Interface** Java Map is an object that maps keys to values. A map cannot contain duplicate keys, each key can map to at most one value. The Java platform contains three general-purpose Map implementations: HashMap, TreeMap, and LinkedHashMap.
8. **SortedSet Interface** SortedSet is a Set that maintains its elements in ascending order. Sorted sets are used for naturally ordered sets.
9. **SortedMap Interface** A map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs.

> Übrigens, das ist eine beliebte Prüfungsfrage: *"Welche Eigenschaften hat das Collections-Interface ...?"*

1.8.2. Special Java Collections Classes

Java Collections framework comes with many implementation classes for the interfaces. Most common implementations are:

1. `HashSet` Class
2. `TreeSet` Class
3. `ArrayList` Class
4. `LinkedList` Class
5. `HashMap` Class
6. `TreeMap` Class

1.8.3. Synchronized Wrappers

The **synchronization** wrappers add automatic synchronization (thread-safety) to an arbitrary collection. Each of the six core collection interfaces - Collection, Set, List, Map, SortedSet, and SortedMap - has one static factory method, which return a synchronized (thread-safe) collection backed up by the specified collection.

1.8.4. Unmodifiable Wrappers

Unmodifiable wrappers take away the ability to modify the collection by intercepting all the operations that would modify the collection and throwing an UnsupportedOperationException.

Its main usage are;

- To make a collection immutable once it has been built
- To allow certain clients read-only access to data structures (keep a reference to the backing collection but hand out a reference to the wrapper)
- To avoid ConcurrentModificationException

1.9. Assoziationen

Klassen existieren quasi nie unabhängig, denn Software besteht immer auch vielen Klassen und weiteren Datenstrukturen, die nur gemeinsam funktionieren und somit die Software bilden.

1.9.1. Beziehungsarten

Klassen gehen also mit anderen Klassen viele sogenannte **"Beziehungen"** ein. Dabei gibt es verschiedene Arten von Beziehungen. Die wichtigsten sind:

- **Aufruf** (*dynamische Beziehung zu Laufzeit*), kann innerhalb einer Software sein oder zwischen ganzen Systemen!
- **Vererbung**, **Realisierung** (*statische Beziehung zur Compilezeit*)
- **Besitz** (*statische Beziehung zur Compilezeit*) → wird (neben weiteren Varianten) oft auch als **Assoziation** bezeichnet.

> *Der Grad - also die Anzahl und Qualität - der Summe aller dieser Beziehungen zwischen den Klassen einer Software wird unter dem Begriffspaar **Kopplung & Kohäsion** zusammengefasst ("loose coupling"). Ein allgemeines Qualitätsziel von Software ist bspw., einen niedrigen Grad von Kopplung der Komponenten zu erreichen. Je geringer die Kopplung ist, desto besser ist die Software änderbar!*

⇒ *Einfügen: Erläuterndes Bild von "Kopplung & Kohäsion"*

1.9.2. Aggregation & Komposition

Assoziationen können auch qualitativ und semantisch recht unterschiedlich sein. In diesem Zusammenhang sind Aggregation und Komposition von Bedeutung.

Die Aggregation, sowie auch die Komposition, ist eine Assoziation von **Teilen zu einem Ganzen**. Jedes Teil ist zu dem Ganzen mit einer Assoziation "ist-teil-von" (part-of) verbunden.

Der Unterschied einer Komposition zur Aggregation ist, dass die Teile, die ein Objekt enthält, von dem Ganzen **existentiell abhängig** sind (Beispiel *Gebäude* und *Räume*).

> *Frage: Wie könnte man "existentielle Abhängigkeit" im Code ausdrücken?*

1.9.3. Navigierbarkeit

Sobald Klassen durch Assoziationen verbunden werden, kommt der Aspekt der **Navigierbarkeit** (von Instanz zu Instanz) hinzu, d.h. man überlegt, welche Klasse welche als Attribut enthält und ob auch die assoziierte Klasse etwas über diese Beziehung weiß.

Letztlich drückt sich die Navigierbarkeit darin aus, ob assoziierte Klassen

- als Attribut mit `Typ` → (2) oder
- nur als `Id` in Form eines primitiven Datentyps → (3)

vorhanden sind. Auch die "besitzende" Klasse hat in aller Regel selbst eine ID → (1).

Am Beispiel:

```java
public class Train {
    public long id; (1)
    public Locomotive locomotive; (2)
    public long locomotiveId; (3)
}                           |
                            |
public class Locomotive {   |
    public long id; <-------+
    public Train train; // ist das hier sinnvoll?
}
```

1.9.4. One-to-One-Assoziation

Siehe Unit-Test:

```
de.dhbw.course2.relations.RelationsTest.oneToOne()
```

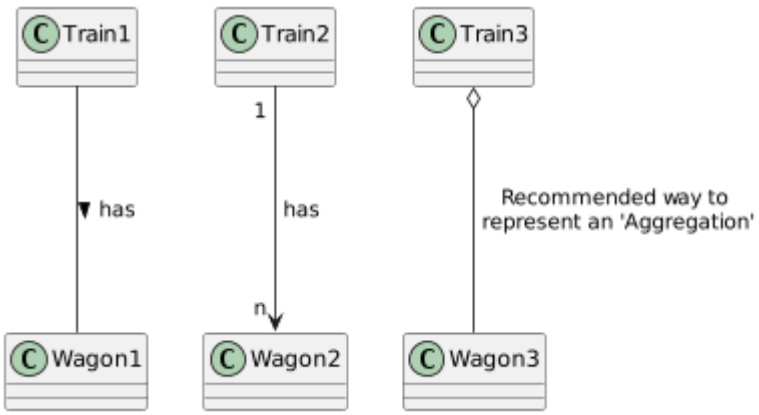
Ein Beispiel

> *Jede natürliche Person besitzt (genau) einen Personalausweis, der Ausweis kann verloren gehen, die Person kann sterben, dann müsste auch der Personalausweis vernichtet werden.*

Es geht hier also oft um die Überlegung zur **Semantik bzw. Qualität** einer Beziehung zwischen zwei abhängigen Objekten!

1.9.5. One-to-Many-Assoziation

Ein Beispiel für eine Eins-zu-Viele Beziehung:



Man beachte immer auch die Richtung der Pfeile, die in aller Regel etwas über die **Lesart** und **Navigierbarkeit** aussagt!

Zugehörige Unit-Tests zur **Demonstration**:

```
de.dhbw.course2.relations.RelationsTest.oneToMany1()
de.dhbw.course2.relations.RelationsTest.oneToMany2()
```

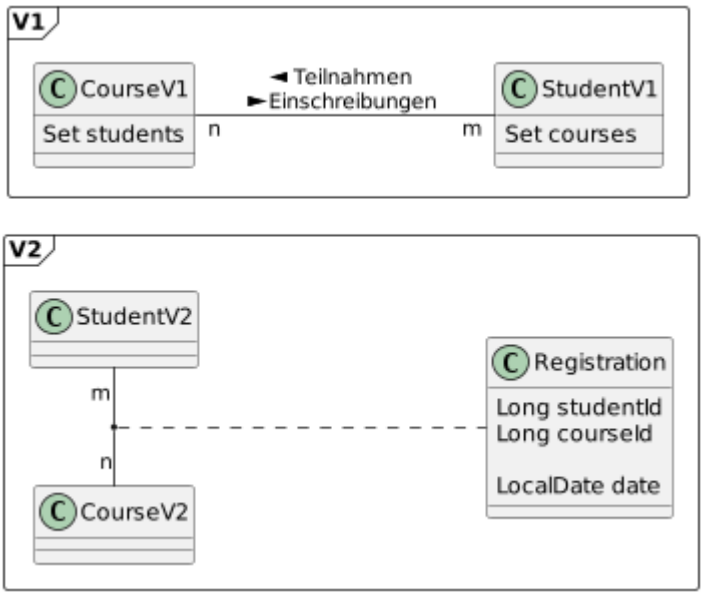
sowie Implementierungen für V1 und V2:

```
de.dhbw.course2.basics.collections.n
```

→ Übung 5

1.9.6. Many-to-Many-Assoziation

Die folgenden zwei Varianten kommen am häufigsten vor:



Implementierungsbeispiele:

```
de.dhbw.course2.basics.collections.n
de.dhbw.course2.basics.collections.nm
```

Zugehörige Unit-Tests:

```
de.dhbw.course2.relations.RelationsTest.manyToMany()
```

Eine weitere Perspektive auf Relationen kommt oft im Umfeld Datenbanken & SQL vor, dazu mehr erst später, am → `Kurstag 7`.

1.10. Übungen

Die **Übungen** sollen in Form von **Unit-Tests** in folgendem *Package* implementiert werden:

```
src/test/java/de/dhbw/course2/exercises/ExerciseTests.java
```

Die **Testobjekte**, also die *echten* Klassen, Interfaces oder anderer Sourcecode sollen getrennt, nämlich hier umgesetzt werden:

```
src/main/java/de/dhbw/course2/exercises
```

Übung 1

Die Übung für `equals()` und `hashCode()` besteht aus 2 Teilen:

1a) Erstelle eine Klasse `News`, die den Inhalt der News als Attribut/Feld vom Typ `String` enthält. Versuche dabei, die zum Feld gehörenden Methoden (`getter` und `setter`) UND einen Konstruktur (optional) mithilfe von *Code Generierung* zu erzeugen. Schreibe dazu einen Test (in der Testmethode `exercise1a()`), in der der hashCode von 2 News-Instanzen *technisch* verglichen wird.

1b) Mache die News mithilfe der Methode `equals()` und `hashCode()` (=Overrides aus der Klasse Object) *logisch* vergleichbar. Schreibe einen Test (in der Testmethode `exercise1b()`), durch den geprüft werden kann, ob 2 News (= 2 Instanzen der Klasse News) mit verschiedenen - vielleicht aber *ähnlichen* Aussagen - inhaltlich gleich sind oder nicht.

Übung 2

Übung zu **Typumwandlung** in

```
src/test/java/de/dhbw/course2/exercises/MoreExerciseTests.java
```

Schreibe je einen Test für

a. Gegeben `char c = '1'` → Umwandlung in `int`

b. Gegeben `int i = 127` → Umwandlung in `byte`

und prüfe das Ergebnis, also den erwarteten Wert, jeweils mithilfe der Assertions-Methode `assertEquals(<expected>, <actual>)`.

Übung 3

Übung zu **Syntax, Klassenmodell** und erforderliche **Methoden**.

Erzeuge eine

1. konkrete Klasse `Person`
2. mit einem Attribut `name`, erstelle dann
3. die zum Attribut gehörende `get` und `set` Methode,
4. erstelle außerdem die `equals()` und `hashCode()` Methoden,
5. zuletzt leite aus dieser konkreten Klasse ein Interface `Mensch` ab

Wenn möglich, nutze für alle diese Schritte `Code-Generation`, die von der IDE angeboten werden (IntelliJ, Tastenkombinationen).

Übung 4

a. **Listen**: Schreibe einen Test, in dem eine konkrete Implementierung von `List` und den Datentyp `Integer` benutzt wird. Befülle diese mit beliebig vielen Einträgen. *Optional: ... und summiere alle Listeneinträge mithilfe einer* `for-each` *Schleife*.

b. **Maps**: Schreibe einen Test, in dem eine konkrete Implementierung von `SortedMap` benutzt wird. Befülle diese mit mindestens 5 Einträgen, nutze dazu `String` sowohl für den Schlüssel (K) also auch für den Wert (V). Überprüfe die Haupt-Charakteristik *"natürliche Sortierung"* dieses Map-Typs.

Übung 5

Erstelle zwei Klassen

- `Course` und
- `Student`

Der Kurs kann von **mehreren** Student:innen besucht werden. Setze diese Beziehung zwischen den beiden Klassen in der Klasse `Course` um.

Überlege hier auch, welcher **Listentyp** sich dafür am besten eignet.

Übung 6 (optional)

Umsetzung des Prinzips **Information Hiding**

In dem folgenden Paket finden sich die Klassen aus der Demonstration:

```
de.dhbw.course2.basics.collections.nm.v2
```

Diese Klassen enthalten aber noch keine Methoden!

Füge die zu den Attributen gehörenden `getter` und `setter` Methoden hinzu und passe auch den zugehörigen Unit-Test entsprechend an. Der findet sich hier:

```
de.dhbw.course2.relations.RelationsTest.manyToMany()
```

Testfragen

Im Modul `/exam` finden sich weitere kleine Übungen für die Inhalte des Kurses 2, und zwar in der dortigen Testklasse:

```
<your-repo>/exam/test/de/dhbw/exam/course2/ExamTest.java
```