

## Inhaltsverzeichnis

1. Datenbank Programmierung
  - 1.1. Preparation
  - 1.2. Setup H2 Database
  - 1.3. Theorie & Einführung
    - 1.3.1. SQL-Datenbanken
    - 1.3.2. Structured Query Language (SQL)
    - 1.3.3. NoSQL-Datenbanken
    - 1.3.4. SQL oder NoSQL?
    - 1.3.5. ORM Objektrelationales Mapping
    - 1.3.6. JPA - Jakarta Persistence API
  - 1.4. Demonstrationen
  - 1.5. Exercises
  - 1.6. Tipps, Patterns & Best Practices

## 1. Datenbank Programmierung

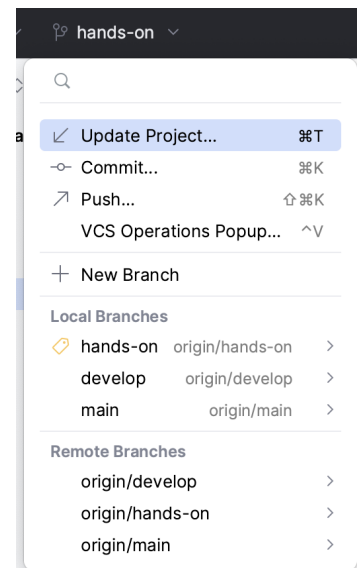
### 1.1. Preparation

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. ...

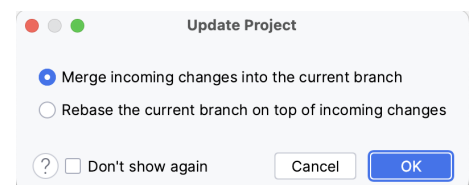
✓ Update Project...

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

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. Setup H2 Database



Grundsätzliches zur H2 Datenbank → [h2database.com](http://www.h2database.com/html/features.html)  
(<http://www.h2database.com/html/features.html>)

1. **PlugIn** installieren (bereits am Kurstag 1 erfolgt)
2. Die **View DB Browser** aktivieren, im Menü **View > Tool Windows > DB Browser** (kann noch am rechten Rand von IntelliJ angesiedelt werden mit Rechtsklick auf das Symbol und *Move to ...*)
3. **Hinzufügen** einer Datenbankverbindung: Klick auf das **+** Plus-Symbol
4. Dazu folgende **Connection Settings** nutzen und Verbindung testen mithilfe des Buttons **Test Connection** :

**DB Navigator - Settings**

Connections Database Browser Navigation Code Editor Code Completion Data Grid Data Editor Execution Engine Op ▾

+ - ↑ ↓ [Icons]

**H2 Database**

**H2 Database**

Database Properties Details Filters

Name  Type Generic ▾

Description

URL

Authentication None ▾

Driver source External library ▾

Driver library  📁

Driver org.h2.Driver ▾

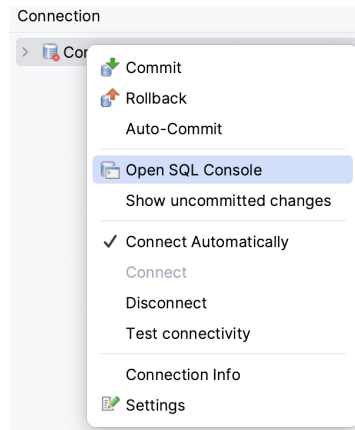
[Reload drivers](#)


☒ Active

Test Connection Info

? Close Apply OK

5. Connection einmal testen mit **Test Connection** , sollte ein *successful* ergeben.
6. Dann kann man eine sogenannte **SQL Console** öffnen, wobei automatisch eine neue Verbindung zur Datenbank aufgebaut wird:



Im unteren Bereich von IntelliJ gibt es neues Icon  **DB Execution Console**, für den Zugriff auf die Inhalte und Ergebnisse von SQL Statements.

## 1.3. Theorie & Einführung

### Allgemeine Inhalte

- ☐ SQL-Datenbanken
- ☐ NoSQL-Datenbanken
- ☐ SQL
- ☐ ORM Objektrelationales Mapping
- ☐ JDBC, JPA
- ☐ Unit-Tests gegen In-Memory-Datenbank (H2)

#### 1.3.1. SQL-Datenbanken

Eine **SQL-Datenbank** ist eine relationale Datenbank, die **Structured Query Language (SQL)** zum Speichern, Abrufen und Bearbeiten von Daten verwendet. SQL-Datenbanken sind die gängigste Art von relationalen Datenbanken und werden von einer Vielzahl von Unternehmen und Organisationen verwendet.

SQL-Datenbanken sind einfach zu benutzen und zu warten und bieten viele Funktionen, die sie für verschiedene Anwendungen geeignet machen. SQL-Datenbanken bieten zum Beispiel folgendes:

- Robuste Datensicherheit
- Skalierbarkeit
- hohe Leistung
- Benutzerfreundlichkeit

und haben anerkannte Eigenschaften, darunter:

- Seit langer Zeit etabliert und weit verbreitet
- Einfach zu bedienen & erlernen
- Vielseitig, geeignet für stark strukturierte Daten, im kleinen wie im grossen Kontext einsetzbar
- Folgen dem → ACID Prinzip und sind daher "zuverlässig"
- Sind skalierbar (vertikal)

#### 1.3.2. Structured Query Language (SQL)

SQL ist eine **Datenbanksprache** zur Definition von Datenstrukturen in relationalen Datenbanken sowie zum Bearbeiten (Einfügen, Verändern, Löschen) und Abfragen von darauf basierenden Datenbeständen.

Vielfach wird hier das Kürzel **CRUD** benutzt, um die grundsätzlichen Datenbankoperationen *Create, Read, Update & Delete* zusammenzufassen, aber die Sprache SQL wird eigentlich und zum besseren Verständnis logisch unterteilt, und zwar im Wesentlichen in

1. **DDL - Data Definition Language**. Befehle zur Definition des Datenbankschemas, d.h. zum Erzeugen, Ändern, Löschen von Datenbanktabellen, sowie zur Definition von Primärschlüsseln und Fremdschlüsseln.

```
create table if not exists persons
(
  'id'          int AUTO_INCREMENT NOT NULL,
  'firstname'   varchar(100)       NOT NULL,
  'lastname'    varchar(100)       NOT NULL,
  PRIMARY KEY ('id')
);

create index TrainNumber_Idx ON Trains (composedNumber)

create view trains as
  select ...
    from locomotives l, wagons w
   where ...
```

SQL

2. **DML - Data Manipulation Language**. Befehle zur Abfrage und Aufbereitung der gesuchten Informationen ( `select`, `join`, ... ), darüber hinaus Befehle zur Datenmanipulation ( `insert`, `update`, `delete` ) und lesendem Zugriff

```
select * from customers c

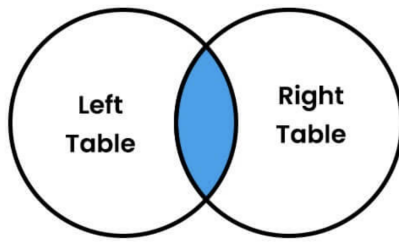
select (p.name) from persons p

select p.address,
       o.delivery
  from persons p,
       orders o
 where p.id = 'Hamburg'
```

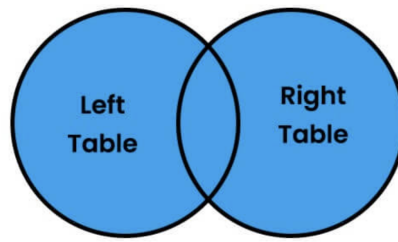
SQL

Zur letztlich gewünschten **Erkenntnisgewinnung** von Daten aus einer Datenbank sind in aller Regel komplexe Abfragen erforderlich. Dazu müssen häufig fachliche zusammenhängende, aber technisch getrennte Daten aus unterschiedlichen Tabellen zusammenführen. Eines der typischsten Mittel, um dies zu erreichen, sind die sogenannten *Joins*, eine spezielle Syntax zur Vereinigung von Daten, z.B.

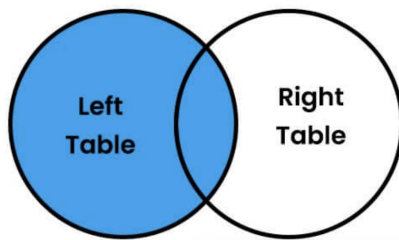
INNER JOIN



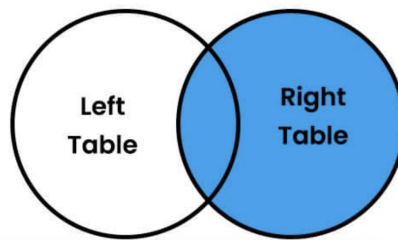
FULL JOIN



LEFT JOIN



RIGHT JOIN



Und noch ein kurzes SQL Beispiel:

```
join
  a.*,
  b.*
from table1 a
join table2 b
  on a.id = b.a_id
```

SQL

In Übung 2 unten findet sich ein Beispiel zum Nachvollziehen dazu.

## Datenbankmodellierung

Ein Datenbankmodell ist die theoretische Grundlage für eine Datenbank und bestimmt, in welcher Struktur Daten in einem Datenbanksystem gespeichert werden. In einem relationale Datenbankmodell werden die Daten tabellenbasiert organisiert. In NoSQL-Datenbanken werden als nicht-tabellenbasiert gespeichert, sondern als Dokumente, Key-Value-Listen oder Graphen.

Es gibt zur **Modellierung** verschiedene Möglichkeiten und Notationen. Das bekannteste ist das Entity-Relationship-Modell und das zugehörige Diagramm ERD. Als Notation wird häufig die Krähenfuß-Notation eingesetzt.

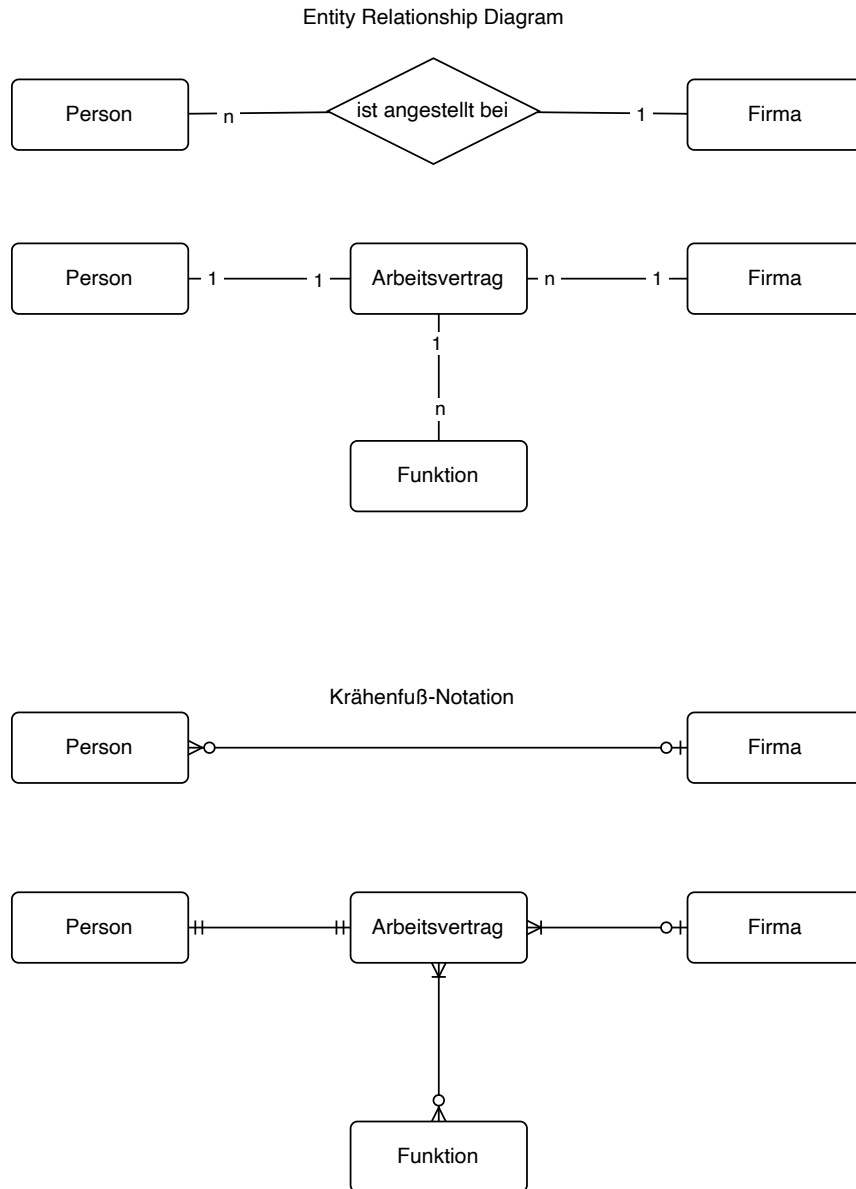


Figure 1. Beispiel Datenbankmodellierung mit Krähenfuß-Notation

oder mit konkreten Attributen:

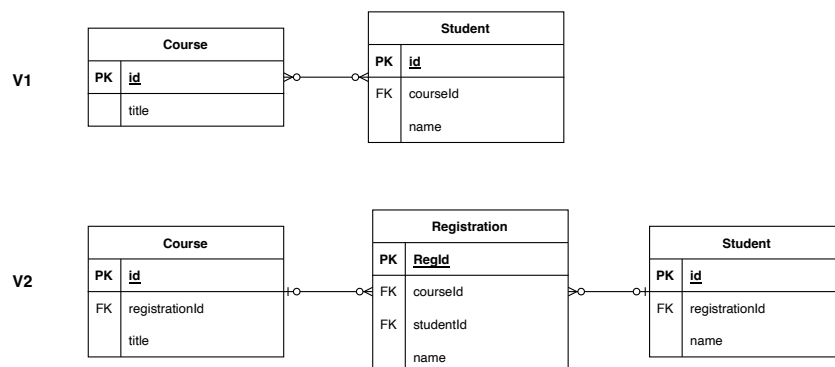


Figure 2. Beispiel Datenbankmodellierung mit Krähenfuß-Notation (n:m)

### 1.3.3. NoSQL-Datenbanken

Eine **NoSQL-Datenbank** (nicht-relationale Datenbank, *Not-Only-SQL*) ist eine nicht-relationale Datenbank, die *nicht* die traditionelle tabellenbasierte Struktur von relationalen Datenbanken verwendet. NoSQL-Datenbanken werden häufig für die Verarbeitung schwach strukturierter Daten und großer Datenmengen verwendet, die sich nicht gut für das relationale Modell eignen.

NoSQL-Datenbanken können in vier Hauptkategorien eingeteilt werden:

1. **Schlüssel-Wert-Datenbank.** Daten als Sammlung von Schlüssel-Wert-Paaren. Der Wert, bei dem es sich um einen einfachen Text oder eine komplizierte Datenstruktur handeln kann, wird mithilfe des Schlüssels nachgeschlagen. Beispiele für Schlüssel-Wert-Speicher sind `DynamoDB` und `Riak`.
2. **Spaltenorientierte Datenbank.** Sie speichern Daten in Spalten statt in Zeilen. Spaltenorientierte Speicher werden häufig für Data Warehousing- und Analyseanwendungen verwendet. Beispiele für spaltenorientierte Speicher sind `Cassandra` und `HBase`.
3. **Dokumenten-Datenbank.** In diesen NoSQL-Datenbanken werden die Daten in "Dokumenten" gespeichert. Dokumente können auf beliebige Weise strukturiert werden, was sie sehr flexibel macht. Beispiele für Dokumentenspeicher sind `MongoDB` und `Couchbase`.
4. **Grafen-Datenbank.** Diese Datenbanken speichern Daten in einer Graphenstruktur, wobei Knoten und Kanten die Daten miteinander verbinden. Graphenspeicher werden häufig für Anwendungen verwendet, die komplexe Beziehungen analysieren müssen. Beispiele für Graphenspeicher sind `Neo4j` und `OrientDB`.

#### 1.3.4. SQL oder NoSQL?

Zu den **Hauptunterschieden** zählen:

- **SQL-Datenbanken** sind relationale Datenbanken. Das bedeutet, dass die Daten in Tabellen organisiert sind und jede Tabelle eine bestimmte **Struktur** mit Beziehungen (Relationen) hat. Die zu speichernden (fachlichen) Daten sollen entsprechend ebenfalls stark strukturiert sein. Beispiele wären u.a.:
  - *Infrastrukturdaten, z.B. Abbildung von Gebäuden oder Städten, oder vom Schienennetz*
  - *Abbildung von fachlichen Einheiten ("Entities" z.B. aus Natur & Umwelt)*
- **NoSQL-Datenbanken** sind nicht-relationale Datenbanken. Das bedeutet, dass die Daten in einer Sammlung von "Elementen" gespeichert werden. Diese Elemente weisen häufig keine spezifische Struktur auf und sind wenig durch Beziehungen miteinander verbunden. Sie eignen sich daher besser für die Speicherung von Daten, die schwach strukturiert sind. Datenbeispiele wären:
  - *Dokumente und deren Inhalt*
  - *Daten aus Sensoren, Monitoring-Daten*
- Ein Hauptunterschied zwischen SQL- und NoSQL-Datenbanken ist die **Skalierung**. SQL-Datenbanken verwenden einen vertikalen Skalierungsansatz, d.h. sie skalieren, indem sie dem Server mehr Leistung hinzufügen. NoSQL-Datenbanken verwenden einen horizontalen Skalierungsansatz, d.h. sie skalieren durch Hinzufügen weiterer Server.
- SQL-Datenbanken sind im Allgemeinen auch **komplexer** als NoSQL-Datenbanken. Das liegt daran, dass SQL-Datenbanken den Regeln von → ACID (Atomarität, Konsistenz, Isolation und Dauerhaftigkeit) folgen müssen, was sie ggf. langsamer und komplizierter machen kann. NoSQL-Datenbanken hingegen sind oft unkomplizierter und können schneller sein, weil sie die ACID Regeln nicht befolgen müssen.

Was ist also **besser**?

Die Entscheidung hängt sehr stark von der Fachlichkeit ab. D.h. also, vor der technischen Abbildung in die Datenbank ist eine Analyse der Daten, deren Struktur, Zweck und Zuverlässigkeit erforderlich.

#### 1.3.5. ORM Objektrelationales Mapping

Objektorientierte Programmiersprachen wie Java kapseln Daten und Verhalten in **Objekten**, hingegen legen relationale Datenbanken Daten in **Tabellen** ab. Die beiden Paradigmen sind grundlegend verschieden. So kapseln Objekte ihren Zustand und ihr Verhalten hinter einer Schnittstelle und haben eine eindeutige Identität. Relationale Datenbanken basieren dagegen auf dem mathematischen Konzept der relationalen Algebra. Dieser konzeptionelle Widerspruch wurde in den 1990er Jahren als object-relational impedance mismatch bekannt.

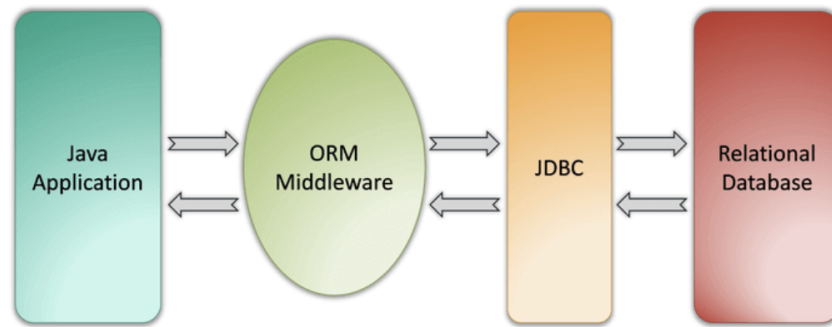


Figure 3. ORM 1

Um dieses Problem zu lösen oder zumindest zu mildern, wurden verschiedene Lösungen vorgeschlagen. Dazu gehört die direkte objektrelationale Abbildung von Objekten auf Relationen. Sie hat den Vorteil, dass einerseits die Programmiersprache selbst nicht erweitert werden muss und andererseits relationale Datenbanken als etablierte Technik in allen Umgebungen als ausgereifte Software verfügbar sind.

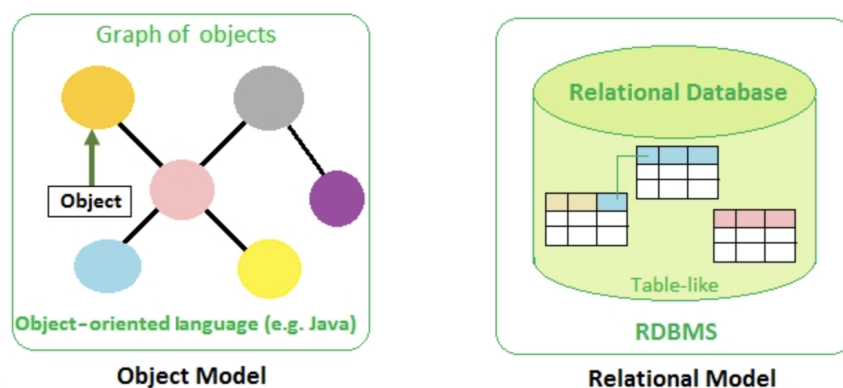


Figure 4. ORM 2

### 1.3.6. JPA - Jakarta Persistence API

Die **Java/Jakarta Persistence API** ist eine Spezifikation von Java. Sie wird benutzt, um Java Objekte in relationalen Datenbanken zu persistieren. Sie agiert dabei als Brücke zwischen den objekt-orientierten Domänenmodellen ( **POJOs** ) und dem **RDBMS** .

Dabei ist sie "nur" eine Spezifikation, sie führt selbst keine Datenbankoperationen aus. Daher ist immer eine Implementierung der API erforderlich, ORM Tools wie **Hibernate** oder **EclipseLink** implementieren die JPA Spezifikation für die Datenpersistenz.



```

1  @Entity ❶
2  @Table(name="Stations") ❷
3  public class Station {
4      @Id ❸
5      @GeneratedValue(strategy=GenerationType.AUTO)
6      private Long id;
7
8      @Column( ❹
9          name="StationName",
10         length=100,
11         nullable=false,
12         unique=false)
13     private String name;
14
15     @Transient ❺
16     private Integer yearOfConstruction;
17
18     @Temporal(TemporalType.DATE)
19     private Date nextFireDrill;
20
21     @Enumerated(EnumType.STRING)
22     private Buildings buildings;
23
24     @OneToMany(mappedBy="Platforms") ❻
25     private Set<Platform> platforms;
26
27     // other fields, getters and setters
28 }

```

- ❶ Markiert eine Datenbank-Entität im Sinne der JPA
- ❷ Tabellename Mehrzahl, Javaklasse Einzahl
- ❸ Primärschlüssel
- ❹ Spalten Metadaten & Constraints
- ❺ wird nicht in der Tabelle gespeichert
- ❻ "One-To-Many" Relation als Liste ( Set )

```

1  @Entity
2  @Table(name="Platforms")
3  public class Platform {
4
5      @Id
6      @GeneratedValue(strategy=GenerationType.AUTO)
7      private Long id;
8
9      @ManyToOne 1
10     @Column(name="StationId")
11     @JoinColumn(name="id", nullable=false)
12     private Station station;
13
14     // ... more fields
15
16     public Platform() {}
17 }

```

<sup>1</sup> siehe → (6): "Rück-Bezug" nach instanzierter Relation

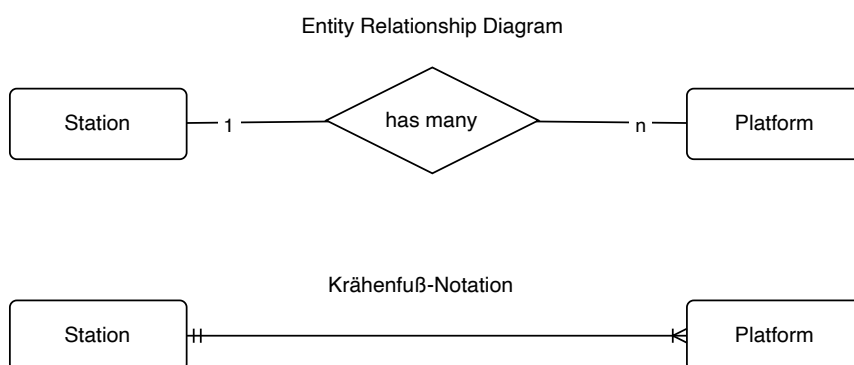


Figure 5. "1:n" Relation im Sourcecode

*to be continued here ...*

## 1.4. Demonstrationen

Die Unit-Tests zur **Demonstration** finden sich hier:

```
src/test/java/de/dhbw/databases/DataSourceDBUnitTests.java
```

Der zugehörige, in den Tests genutzte **Quellcode** findet sich hier:

```
src/main/java/de/dhbw/databases/demo/*.java
```

## 1.5. Exercises

Nutze folgendes Package für deine **Unit-Tests**:

```
src/test/java/de/dhbw/databases/ExerciseTests.java
```


Die im Test benutzten **Implementierungen** gehören in das Package:

```
src/main/java/de/dhbw/databases/exercises/*.java
```

### Übung 1:

Nutze die **SQL Statements** aus der Datei

```
module-databases/dbunit/demonstration.sql
```

um diese einfach selbst einmal *Statement für Statement* gegen die lokale bzw. eigene H2 Datenbank auszuführen. Die Befehle können einfach aus der Datei in die geöffnete/gestartete *Datenbank Console* kopiert werden, dann erscheinen in den Zeilen jeweils *execution* Symbole: 3  `CREATE TABLE IF N...`

### Übung 2:

Nutzt die **SQL Statements** aus der Datei

```
module-databases/dbunit/join-example.sql
```

um diese einfach, *Schritt für Schritt*, selbst einmal gegen die lokale bzw. eigene H2 Datenbank auszuführen.

### Übung 3:

Zwei weitere kleine Übungen:

**a)** Eine Tabelle anlegen mit `create table`

Erzeuge eine **neue Tabelle** `ORDERS` in der Datenbank. Die Tabelle soll folgende Spalten bekommen:

- `id` als Ganzzahl
- `orderNumber` als String

**b)** Daten hinzufügen mit `insert into`

Erzeuge in der Tabelle `ORDERS` einen oder mehrere neue Datensätze mit beliebigen Daten an

## 1.6. Tipps, Patterns & Best Practices

### Das ACID Prinzip

ACID beschreibt die häufig erwünschte Eigenschaften von Transaktionen in Datenbankmanagementsystemen (DBMS). Es steht für `Atomicity`, `Consistency`, `Isolation` und `Durability`. Sie gelten als Voraussetzung für die Verlässlichkeit von Systemen. Demgegenüber steht der Begriff `Eventual Consistency` (gelegentliche Konsistenz) beim Einsatz von NoSQL Datenbanken.