Name:	
Klasse:	
Datum:	

Schuljahr: 2022/23

2 JDBI und Objekte

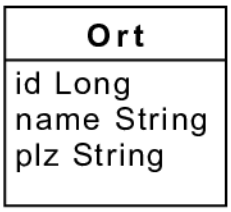
Im nächsten Schritt sollen die Objekte einer Klasse direkt auf die Datenbank gemappt werden.

2.1 Eclipse-Projekt

Importieren Sie das Eclipse-Projekt `eclipse_db_jdbi_02.zip`.
Voraussetzung: OpenJDK 17 (☞Installationsanleitung) und ein aktuelles Eclipse EE (☞Installationsanleitung), z. B. 2022-09 oder neuer.

2.2 Eine Klasse, eine Tabelle

In diesem Beispiel wird der Zugriff auf eine Klasse gezeigt, hier die Klasse `Ort`.



Hier müssen folgende Java-Klassen erstellt werden:

`Ort` Repräsentiert die Daten eines Datensatzes.

`OrtDAO` Regelt das Laden, Speichern, ... über entsprechende SQL-Befehle

Dazu wird zuerst eine Klasse `Ort` erstellt. Die entsprechenden getter/setter etc. werden hier mit „lombok“ generiert. Die Klasse muss dabei die Spalten der Tabelle `ort` abdecken, damit ein kompletter Zugriff möglich ist.
Wichtig ist, dass die Klasse einen Leerkonstruktor hat, der mit `@NoArgsConstructor` automatisch angelegt wird.

```
@Data
@NoArgsConstructor
public class Ort {

   @ColumnName("o_id")
   private Long id;

   @ColumnName("o_name")
   private String name = "";

   @ColumnName("o_plz")
   private String plz = "";

   @ConstructorProperties({ "o_id", "o_name", "o_plz" })
   public Ort(final Long id, final String name, final String plz) {
      this.id = id;
      this.name = name;
      this.plz = plz;
   }

   public Ort(final String name, final String plz) {
      this.name = name;
      this.plz = plz;
   }
}
```

JDBI kann mit unterschiedlichen Verfahren die Daten aus der Tabelle in die Klasse „schieben“.

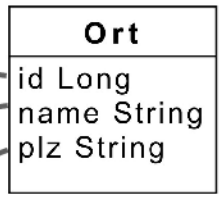
Hier zwei Varianten, von den vielen, die es gibt:

`BeanMapper`

Bei BeanMapper nimmt JDBI die Spalte, sucht in dem Objekt das entsprechende Feld und setzt den Wert. Wird nichts angegeben, so muss der Spaltenname dem Variablennamen genau entsprechen. Dies kann aber zu Problemen führen, wenn Abfragen über mehrere Tabellen erfolgen, da beispielsweise `id` oder `name` öfters vorkommen kann. Geschickter ist es, den Namen über die `@ColumnName` konkret festzulegen. Mit dem Kürzel `o` für Ort, der auch als Alias bei der SQL-Abfrage verwendet wird, können so Konflikte von vornherein vermieden werden.

```
SELECT o.id AS o_id, o.name AS o_name, o.plz AS o_plz
       FROM ort o;
```

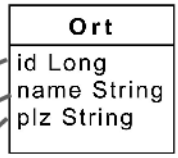
```
@ColumnName("o_id")
private Long id;

@ColumnName("o_name")
private String name;

@ColumnName("o_plz")
private String plz;
```



`ConstructorMapper`

Beim ConstructorMapper wird das Objekt mit dem Konstruktoraufruf erzeugt. Dabei müssen alle Parameter des Konstruktors in der Spaltenabfrage enthalten sein. Auch hier können die Namen entsprechend mit `@ConstructorProperties({ "o_id", "o_name", "o_plz" })` angepasst werden.

Wichtig, es darf pro Klasse die Annotation `@ConstructorProperties` nur einmal verwendet werden.




```
@ConstructorProperties({ "o_id", "o_name", "o_plz" })
public Ort(final Long id, final String name, final String plz) {
   this.id = id;
   this.name = name;
   this.plz = plz;
}
```

Notiz:

2.3 DAO – Data Access Object

Im nächsten Schritt muss festgelegt werden, welche Funktionalität mit der Klasse `Ort` durchgeführt werden soll. Dazu wird das Entwurfsmuster DAO – Data Access Object verwendet.

Dabei wird ein Interface mit den Methoden definiert, welches zusätzlich mit Annotationen versehen wird. JDBI erzeugt daraus automatisch eine entsprechende Klasse.

```
12 public interface OrtDAO {
13
14    @SqlUpdate("""
15          CREATE TABLE IF NOT EXISTS ort (
16             id      INT         PRIMARY KEY AUTO_INCREMENT,
17             name    VARCHAR(50),
18             plz     VARCHAR(10)
19          ) ENGINE=InnoDB DEFAULT CHARSET=utf8
20          """)
21    void createTable();
22
23    @SqlQuery("SELECT * FROM ort WHERE id=:id")
24    @RegisterBeanMapper(Ort.class)
25    Ort findById(@Bind("id") int id);
26
27    @SqlQuery("SELECT * FROM ort")
28    @RegisterBeanMapper(Ort.class)
29    List<Ort> getAll();
30
31    @SqlUpdate("""
32          INSERT INTO ort (name, plz)
33             VALUES (:name, :plz)
34          """)
35    @GetGeneratedKeys("id")
36    long insert(@BindBean Ort ort);
37
38    @SqlUpdate("""
39          INSERT INTO ort (name, plz)
40             VALUES (:name, :plz)
41          """)
42    @GetGeneratedKeys("id")
43    long insert(@Bind("name") String name, @Bind("plz") String plz);
44 }
```

Dazu werden folgende Methoden definiert:

`createTable()`	Wird aufgerufen, wenn die Tabelle `ort` erstellt werden soll. Der entsprechende SQL-Befehl wird mit `@SqlUpdate` festgelegt.
`findById()`	Wird aufgerufen, wenn ein Ort anhand seiner ID ermittelt werden soll. Mit `@SqlQuery` wird dabei der entsprechende `select`-Befehl definiert. Zusätzlich wird mit `@Bind` festgelegt, dass der Parameter (hier `id`) für den Spaltenwert (hier `:id`) verwendet werden soll. Damit JDBI weiß, welche Spalte auf welches Feld gemappt werden soll, muss zusätzliche ein Mapper mit `@RegisterBeanMapper` registriert werden.
`getAll()`	Wird aufgerufen, wenn eine Liste aller Orte ermittelt werden soll. Auch hier wird mit `@SqlQuery` der entsprechende `select`-Befehl definiert.
`insert(Ort ...)`	Wird aufgerufen, wenn ein Ort-Objekt (mit `@BindBean`) gespeichert werden soll. Die Annotation `@GetGeneratedKeys` sorgt dafür, dass der erzeugte Primary Key als Rückgabewert verwendet wird.
`insert(name, plz)`	Wird aufgerufen, wenn ein neuer Ort erzeugt werden soll, bei dem Name und PLZ angegeben werden. Auch hier wird mit `@Bind` der Wert entsprechend dem Spaltenwert zugeordnet.

Aufruf

```
17      final Jdbi jdbi = Jdbi.create("jdbc:mariadb://localhost:3306/bsp", "bsp", "");
18
19      System.out.println("Connect bsp ...");
20
21      jdbi.installPlugin(new SqlObjectPlugin());
22      jdbi.installPlugin(new GuavaPlugin());
23      final Handle handle = jdbi.open();
24
25      final OrtDAO ortDao = handle.attach(OrtDAO.class);
26
27      ortDao.createTable();
28
29      final Ort o1 = new Ort("München", "12345");
30      long pk = ortDao.insert(o1);
31      System.out.println("inserted PK = " + pk);
32
33      pk = ortDao.insert("Haimhausen", "85778");
34      System.out.println("inserted PK = " + pk);
35
36      System.out.println("##########################");
37      final List<Ort> list = ortDao.getAll();
38      list.stream()
39            .forEach(e -> System.out.println(e));
40      System.out.println("##########################");
41      final Ort o = ortDao.findById(1);
42      System.out.println(o);
```

Ausgabe

```
Connect bsp ...
inserted PK = 1
inserted PK = 2
##########################
Ort(id=1, name=München, plz=12345)
Ort(id=2, name=Haimhausen, plz=85778)
##########################
Ort(id=1, name=München, plz=12345)
```

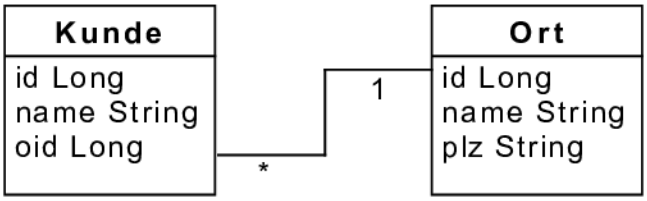
Aufgabe JDBI-02-1

Analysieren Sie das Beispiel und probieren Sie es aus.

Notiz:

2.4 Ort - Kunde: 1–n Beziehung

Im nächsten Schritt wird der `Kunde` hinzugefügt, der an einem bestimmten Ort wohnt.



In der Datenbank wird hier die Verknüpfung mit der `id` durchgeführt. Dies ist aber in der Klasse bzw. in dem daraus erzeugten Objekt nicht sinnvoll, da man hier beim Kunden gleich auf das entsprechende Ort-Objekt zugreifen will. Daher muss beim Einlesen aus der `id` das entsprechende Ort-Objekt erzeugt werden.

```
12 @Data
13 @NoArgsConstructor
14 public class Kunde implements Comparable<Kunde> {
15
16    @ColumnName("k_id")
17    private Long id;
18
19    @ColumnName("k_name")
20    private String name = "";
21
22    @Nested
23    private Ort ort;
24
25    public Kunde(final Long id, final String name) {
26       this.id = id;
27       this.name = name;
28    }
29
30    @ConstructorProperties({ "k_id", "k_name", "o_id", "o_name", "o_plz" })
31    public Kunde(final Long id, final String name, final Long oid, final String oname,
32          final String oplz) {
33       this.id = id;
34       this.name = name;
35       if (oid != null) {
36          ort = new Ort(oid, oname, oplz);
37       }
38    }
39
40    public Kunde(final String name) {
41       this.name = name;
42    }
43
44    public Kunde(final String name, final Ort ort) {
45       this.name = name;
46       this.ort = ort;
47    }
48
49    @Override
50    public int compareTo(final Kunde k) {
51       return name.compareTo(k.getName());
52    }
```

Damit JDBI erkennt, dass der Ort aus einem anderen Bereich kommt, wird das Feld `ort` mit der Annotation `@Nested` versehen. Alle anderen Felder, ... werden wie bei der Klasse `ort` behandelt. Damit der Kunde auch nach dem Namen sortiert werden kann, wird das Interface `Comparable<Kunde>` eingebunden und über die Methode `compareTo()` implementiert.

Notiz:

RowMapper

Der `RowMapper` wird bei jeder Zeile im `ResultSet` aufgerufen, um die Daten der Spalten im Kunden-Objekt zu speichern. Hier können auch entsprechende Bedingungen/Abfragen eingebaut werden, um beispielsweise eine Konvertierung durchzuführen.

```
11 public class KundeMapper implements RowMapper<Kunde> {
12
13    @Override
14    public Kunde map(final ResultSet rs, final StatementContext ctx) throws SQLException {
15       // debug - Spaltenüberschriften ausgeben
16       // final ResultSetMetaData meta = rs.getMetaData();
17       // for (int i = 1, n = meta.getColumnCount(); i <= n; i++) {
18       // System.out.print(meta.getColumnLabel(i) + " ");
19       // }
20       // System.out.println();
21       // ------------
22
23       final Kunde k = new Kunde();
24       k.setId(rs.getLong("k_id"));
25       k.setName(rs.getString("k_name"));
26
27       final Long oid = rs.getLong("k_oid");
28       // ein null-Wert in der Spalte wird bei getLong() mit '0' umgesetzt - Zähler
29       // beginnt bei '1'
30       if (oid != 0) {
31          k.setOrt(new Ort(oid, rs.getString("o_name"), rs.getString("o_plz")));
32       }
33       return k;
34    }
35 }
```

Die Methode `map` wird bei jeder Zeile der Ergebnismenge aufgerufen. Damit man die erzeugten Spaltennamen für Debug-Zwecke sieht, kann man den entsprechenden Code einkommentieren.

Im ersten Schritt wird ein leeres Kunden-Objekt erzeugt und dann die drei Spalten aus dem Result-Set gelesen. Die ersten beiden können direkt zugewiesen werden. Bei der `oid` muss erst geprüft werden, ob hier überhaupt ein Wert vorhanden ist oder in der Spalte ein `null`-Wert eingetragen ist. Die Methode `getLong()` wandelt einen evtl. `null`-Wert in „0“ um. Da die `AUTO_INCREMENT` Funktion immer bei „1“ zu zählen anfängt, bedeutet der „0“ Wert, dass keine Verknüpfung vorhanden ist.

Ist eine Verknüpfung vorhanden, so wird das entsprechende `Ort`-Objekt erzeugt und dem Kunden zugewiesen.

Notiz:

KundeDAO

Das DAO verhält sich sehr ähnlich wie beim Ort. Der wesentliche Unterschied liegt hier bei der Methode `getAll()`.

```
13 public interface KundeDAO {
14
15    @SqlUpdate("""
16          CREATE TABLE IF NOT EXISTS kunde (
17            id      INT         PRIMARY KEY AUTO_INCREMENT,
18            name    VARCHAR(50),
19            oid     INT
20          ) ENGINE=InnoDB DEFAULT CHARSET=utf8
21          """)
22    void createTable();
23
24    @SqlQuery("SELECT * FROM kunde WHERE id=:id")
25    @RegisterBeanMapper(Kunde.class)
26    Kunde findById(@Bind("id") int id);
27
28    @SqlQuery("""
29          SELECT k.id AS k_id, k.name AS k_name, k.oid AS k_oid,
30                 o.name AS o_name, o.plz AS o_plz
31             FROM kunde k LEFT JOIN ort o ON k.oid=o.id
32          """)
33    @RegisterRowMapper(KundeMapper.class)
34    List<Kunde> getAll();
35
36    @SqlUpdate("""
37          INSERT INTO kunde (name, oid)
38             VALUES (:name, :ort.id)
39          """)
40    @GetGeneratedKeys("id")
41    long insert(@BindBean Kunde user);
42 }
```

Die SQL-Abfrage bei der Methode `getAll()` ist ein `LEFT JOIN`, der Kunde und Ort verknüpft – es sollen ja auch die Kunden angezeigt werden, die keinem Ort zugewiesen sind.

Für diese Abfrage muss über `@RegisterRowMapper` der Mapper auf die Klasse `KundeMapper` gesetzt werden.

Werte für Kunde und Ort erzeugen

Damit auch eine Verknüpfung abgefragt werden kann, müssen im ersten Schritt einige Daten hinzugefügt werden.

```
19       final Jdbi jdbi = Jdbi.create("jdbc:mariadb://localhost:3306/bsp", "bsp", "");
20
21       System.out.println("Connect bsp ...");
22
23       jdbi.installPlugin(new SqlObjectPlugin());
24       jdbi.installPlugin(new GuavaPlugin());
25       final Handle handle = jdbi.open();
26
27       final KundeDAO kundeDao = handle.attach(KundeDAO.class);
28       final OrtDAO ortDao = handle.attach(OrtDAO.class);
29
30       handle.execute("DROP TABLE IF EXISTS kunde");
31       handle.execute("DROP TABLE IF EXISTS ort");
32       kundeDao.createTable();
33       ortDao.createTable();
34
35       IntStream.rangeClosed(1, 5)
36             .forEach(o -> handle.execute("""
37                   INSERT INTO ort (id, name, plz) VALUES (?, ?, ?)
38                   """, o, "o_" + o, "p_" + o));
39       final String sql = "INSERT INTO kunde (id, name, oid) VALUES (?, ?, ?)";
40       handle.execute(sql, 1, "k_1", 1);
41       handle.execute(sql, 2, "k_2", 3);
42       handle.execute(sql, 3, "k_3", 5);
43       handle.execute(sql, 4, "k_4", null);
44
45       System.out.println("Ort -----------------");
46       final List<Map<String, Object>> rso = handle.createQuery("SELECT * FROM ort")
47             .mapToMap()
48             .list();
49       rso.stream()
50             .forEach(e -> System.out.println(e));
```


```
51      System.out.println("Kunde -----------------");
52      final List<Map<String, Object>> rsu = handle.createQuery("SELECT * FROM kunde")
53            .mapToMap()
54            .list();
55      rsu.stream()
56            .forEach(e -> System.out.println(e));
57    }
```

Ausgabe

```
Connect bsp ...
Ort -----------------
{id=1, name=o_1, plz=p_1}
{id=2, name=o_2, plz=p_2}
{id=3, name=o_3, plz=p_3}
{id=4, name=o_4, plz=p_4}
{id=5, name=o_5, plz=p_5}
Kunde -----------------
{id=1, name=k_1, oid=1}
{id=2, name=k_2, oid=3}
{id=3, name=k_3, oid=5}
{id=4, name=k_4, oid=null}
```

1-n Abfrage ausführen

```
18      final Jdbi jdbi = Jdbi.create("jdbc:mariadb://localhost:3306/bsp", "bsp", "");
19
20      System.out.println("Connect bsp ...");
21
22      jdbi.installPlugin(new SqlObjectPlugin());
23      jdbi.installPlugin(new GuavaPlugin());
24      final Handle handle = jdbi.open();
25
26      final KundeDAO kundeDao = handle.attach(KundeDAO.class);
27      final OrtDAO ortDao = handle.attach(OrtDAO.class);
28
29      System.out.println("##########################");
30      final List<Kunde> list = kundeDao.getAll();
31      list.stream()
32            .forEach(e -> System.out.println(e));
33
34      final Kunde u = new Kunde("Franz", ortDao.findById(1));
35      u.setId(kundeDao.insert(u));
36      System.out.println("----------------------");
37      System.out.println(u);
```

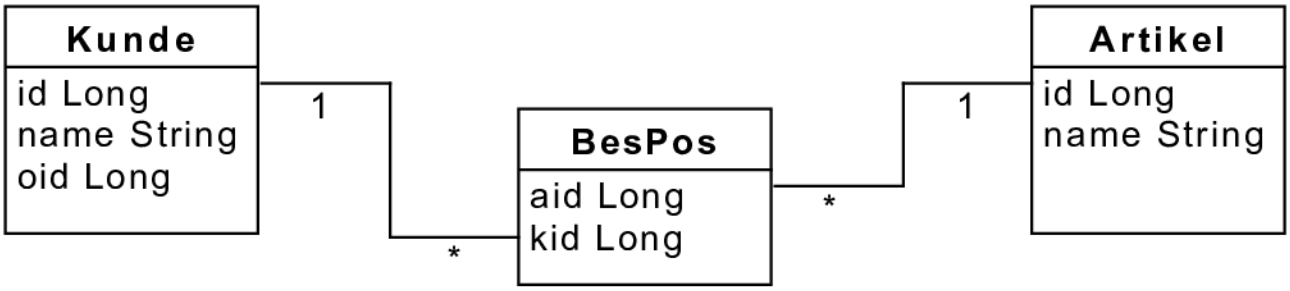
Ausgabe

```
Connect bsp ...
##########################
Kunde(id=1, name=k_1, ort=Ort(id=1, name=o_1, plz=p_1))
Kunde(id=2, name=k_2, ort=Ort(id=3, name=o_3, plz=p_3))
Kunde(id=3, name=k_3, ort=Ort(id=5, name=o_5, plz=p_5))
Kunde(id=4, name=k_4, ort=null)
----------------------
Kunde(id=5, name=Franz, ort=Ort(id=1, name=o_1, plz=p_1))
```

Notiz:

2.5 Kunde - Artikel: n–m Beziehung

Im nächsten Schritt wird die Klasse `Artikel` und die dazugehörigen Bestellpositionen hinzugefügt. n–m Beziehungen müssen in der Datenbank über eine Zwischentabelle gelöst werden. Bei Objekten ist dies nicht notwendig.



Artikel

Die Klasse `Artikel` ist wie die Klasse `Ort` aufgebaut.

```
@Data
@NoArgsConstructor
public class Artikel implements Comparable<Artikel> {

   @ColumnName("a_id")
   private Long id;

   @ColumnName("a_name")
   private String name;

   @ConstructorProperties({ "a_id", "a_name" })
   public Artikel(final Long id, final String name) {
      this.id = id;
      this.name = name;
   }

   public Artikel(final String name) {
      this.name = name;
   }

   @Override
   public int compareTo(final Artikel o) {
      return name.compareTo(o.getName());
   }
}
```

Genauso wie die Klasse `ArtikelDAO`.

```
public interface ArtikelDAO {

   @SqlUpdate("""
         CREATE TABLE IF NOT EXISTS artikel (
            id      INT         PRIMARY KEY AUTO_INCREMENT,
            name    VARCHAR(50)
         ) ENGINE=InnoDB DEFAULT CHARSET=utf8
         """)
   void createTable();

   @SqlQuery("SELECT * FROM artikel WHERE id=:id")
   @RegisterBeanMapper(Artikel.class)
   Artikel findById(@Bind("id") int id);

   @SqlQuery("SELECT * FROM artikel")
   @RegisterBeanMapper(Artikel.class)
```

```
28    List<Artikel> getAll();
29
30    @SqlUpdate("""
31          INSERT INTO artikel (name)
32             VALUES (:name)
33          """)
34    @GetGeneratedKeys("id")
35    long insert(@BindBean Artikel artikel);
36
37    @SqlUpdate("""
38          INSERT INTO artikel (name)
39             VALUES (:name)
40          """)
41    @GetGeneratedKeys("id")
42    long insert(@Bind("name") String name);
43 }
```

BesPos

Die Klasse `BesPos` hat das selbe Schema.

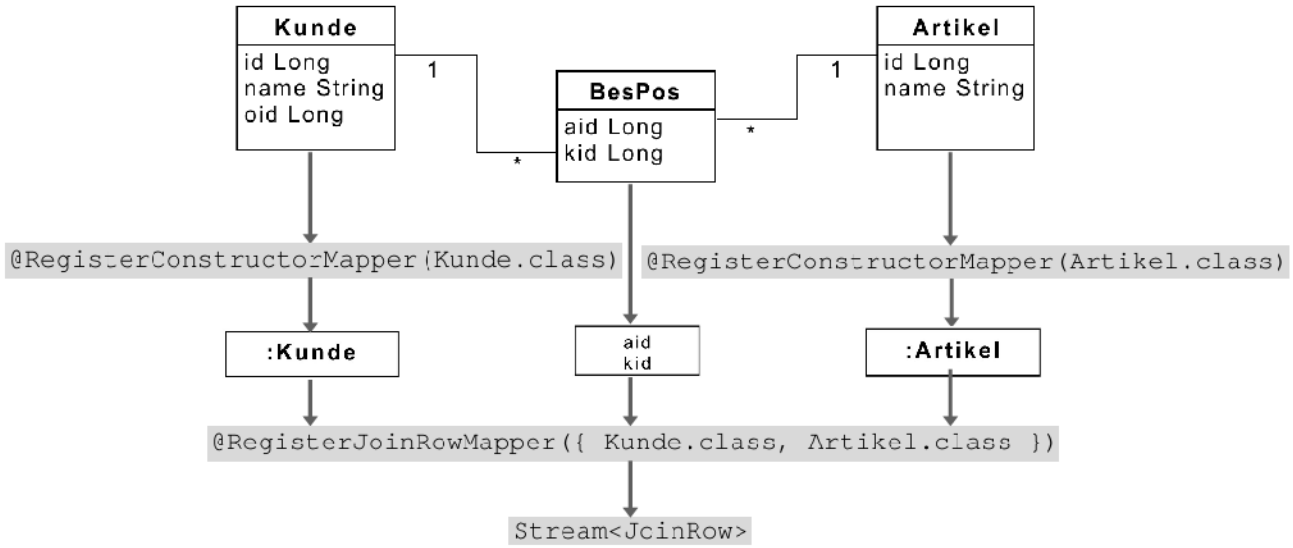
```
8  @Data
9  @NoArgsConstructor
10 public class BesPos {
11
12    @ColumnName("bp_aid")
13    private Long aid;
14
15    @ColumnName("bp_kid")
16    private Long kid;
17
18    public BesPos(final Long aid, final Long kid) {
19       this.aid = aid;
20       this.kid = kid;
21    }
22 }
```

Die Klasse `BesPosDAO` muss jetzt bei der Methode `getKundeArtikel()` besonders behandelt werden.

```
32    @SqlQuery("""
33          SELECT k.id AS k_id, k.name AS k_name, k.oid AS o_id,
34                 o.name AS o_name, o.plz AS o_plz,
35                 a.id AS a_id, a.name AS a_name
36             FROM bespos bp INNER JOIN kunde k   ON bp.kid=k.id
37                            INNER JOIN artikel a ON bp.aid=a.id
38                            LEFT JOIN  ort o     ON k.oid=o.id
39             ORDER BY k.id
40          """)
41    @RegisterConstructorMapper(Kunde.class)
42    @RegisterConstructorMapper(Artikel.class)
43    @RegisterJoinRowMapper({ Kunde.class, Artikel.class })
44    Stream<JoinRow> getKundeArtikel();
```

Notiz:

Die Objekte der Klasse `Kunde` und `Artikel` werden über den `ConstructorMapper` erzeugt. Anschließend werden diese mit dem `JoinRowMapper` miteinander verbunden und stehen dann als `Stream` zur Verfügung.



n-m Abfrage ausführen

```
19      final Jdbi jdbi = Jdbi.create("jdbc:mariadb://localhost:3306/bsp", "bsp", "");
20
21      System.out.println("Connect bsp ...");
22
23      jdbi.installPlugin(new SqlObjectPlugin());
24      jdbi.installPlugin(new GuavaPlugin());
25      final Handle handle = jdbi.open();
26
27      final Multimap<Kunde, Artikel> joined = HashMultimap.create();
28      handle.attach(BesPosDAO.class)
29            .getKundeArtikel()
30            .forEach(jr -> joined.put(jr.get(Kunde.class), jr.get(Artikel.class)));
31
32      System.out.println("##########################");
33      joined.keySet()
34            .stream()
35            .sorted()
36            .forEach(k -> {
37               System.out.println(k);
38               joined.get(k)
39                     .stream()
40                     .sorted()
41                     .forEach(a -> System.out.println("   " + a));
42            });
```

Ausgabe

```
Connect bsp ...
##########################
Kunde(id=1, name=k_1, ort=Ort(id=1, name=o_1, plz=p_1))
   Artikel(id=1, name=a_1)
   Artikel(id=2, name=a_2)
   Artikel(id=3, name=a_3)
   Artikel(id=5, name=a_5)
Kunde(id=2, name=k_2, ort=Ort(id=3, name=o_3, plz=p_3))
   Artikel(id=1, name=a_1)
   Artikel(id=4, name=a_4)
Kunde(id=3, name=k_3, ort=Ort(id=5, name=o_5, plz=p_5))
   Artikel(id=3, name=a_3)
Kunde(id=4, name=k_4, ort=null)
   Artikel(id=3, name=a_3)
```

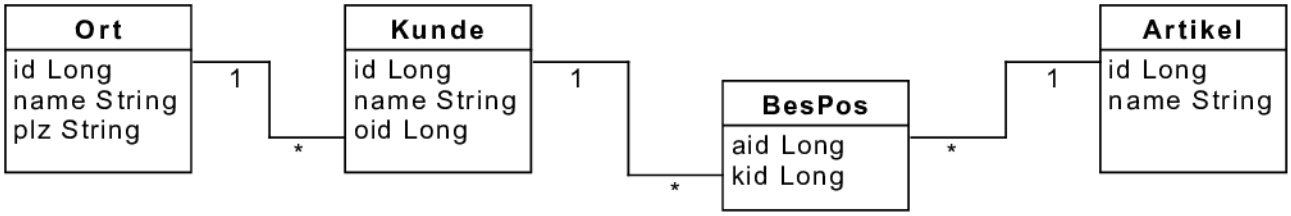

Aufgabe JDBI-02-2

Analysieren Sie das Beispiel und probieren Sie es aus.

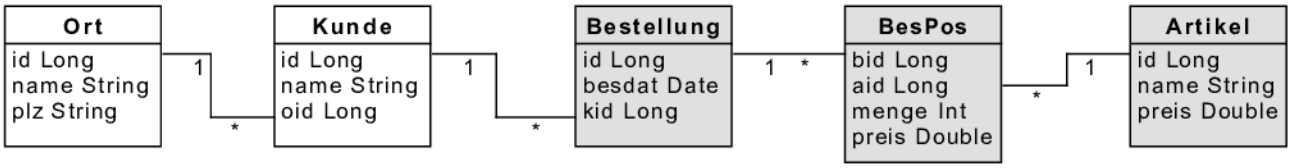
Notiz:

Aufgaben JDBI-02-3

Das bisherige Modell sieht wie folgt aus:



1. Erweitern bzw. passen Sie die Klassen soweit an, dass eine Bestellung zusätzlich eingefügt wird und im OOP-Bereich abgefragt werden kann. Fügen Sie dazu passende Datensätze hinzu. Achten Sie auch darauf, dass nicht alle Verbindungen vorhanden sein müssen, d. h. `null`-Werte in der Beziehung vorhanden sein können.



2. Erweitern bzw. passen Sie die Klassen soweit an, dass der Kunde einer Branche angehört und diese von einem Mitarbeiter betreut wird und im OOP-Bereich abgefragt werden kann. Fügen Sie dazu passende Datensätze hinzu. Achten Sie auch darauf, dass nicht alle Verbindungen vorhanden sein müssen, d. h. `null`-Werte in der Beziehung vorhanden sein können.