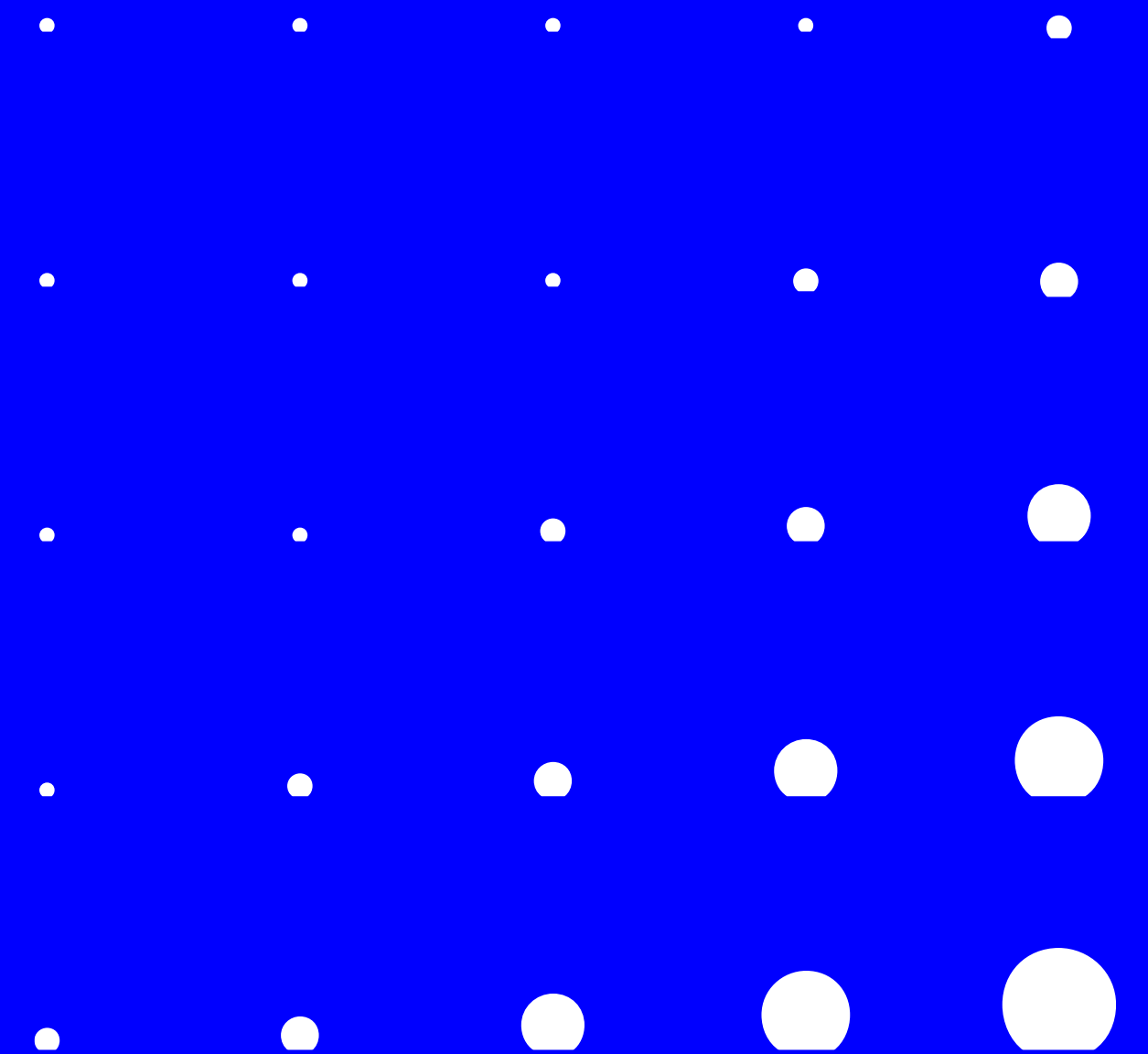


## Die Java Persistence API JPA



# Was sind die Probleme bislang?

- Man erstellt zunächst aus der OOA/OOD ein UML Klassenmodell.
- Dazu erstellt man parallel ein E/R-Modell für die Datenhaltung.
- Dieses bildet man dann in eine Datenbank-Struktur ab, z.B. über phpMyAdmin.
- Jede Änderung,
  - am UML Klassenmodell wie das Hinzufügen einer neuen Eigenschaft hat eine Änderung im Code,
  - sowie am E/R-Modell wie das Hinzufügen eines neuen Attributs zur Folge
  - und wiederum eine Änderung in der Datenbankstruktur, wie das Hinzufügen einer neuen Spalte in einer Datenbanktabelle.
- Geht das nicht auch in einem einzigen Schritt???

# Was ist die Idee eines objektrelationalen Mappers (ORM)?

- Die Idee eines objektrelationalen Mappers (ORM) besteht darin, dass man nur noch die UML Klassenmodellierung durchführt.
- Die separate E/R-Modellierung soll verschwinden, da die UML mindestens dieselbe Mächtigkeit der E/R-Modellierung beinhaltet.
- Nach der Erstellung der UML-Klassendiagramme codet man diese Klassen.
- Diese Klassen werden innerhalb des Codes zusätzlich noch mit einigen Annotationen versehen.
  - Jede Änderung bleibt also in der UML und im Code.
  - Diese Annotationen sorgen dann dafür, dass alles in DB-Tabellen abgelegt wird.
    - Die gesamte Tabellenstruktur wird dabei dynamisch und voll automatisiert erzeugt und verwaltet.

# Was ist die Idee eines objektrelationalen Mappers (ORM)?

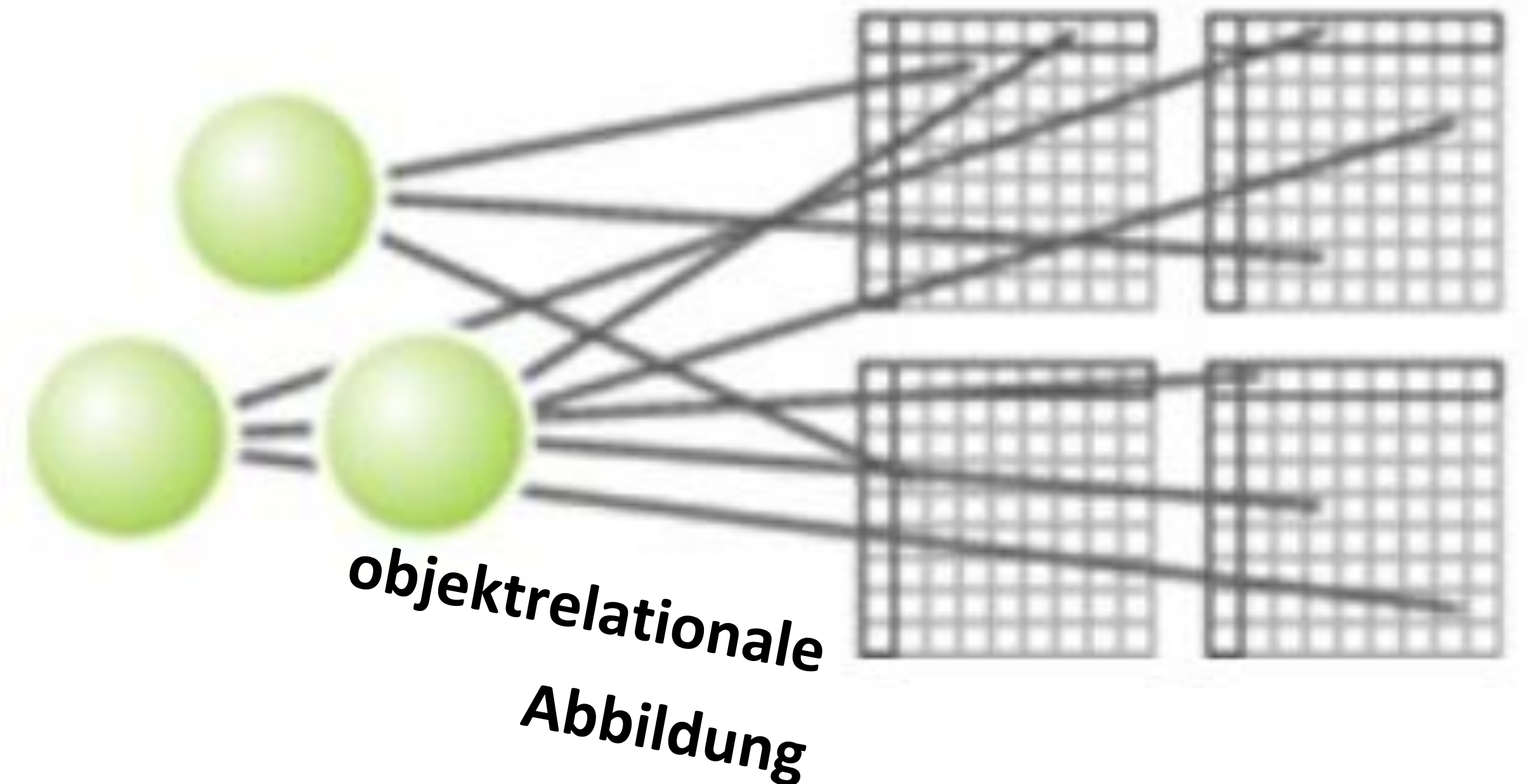
- Die Objektrelationale Abbildung ist eine Technik der Softwareentwicklung, mit der ein in einer objektorientierten Programmiersprache geschriebenes Anwendungsprogramm seine Objekte in einer relationalen Datenbank ablegen kann.
- Dem Programm erscheint die Datenbank dann als objektorientierte Datenbank, was die Programmierung erleichtert.

# Was ist die Idee eines objektrelationalen Mappers (ORM)?

- Objektorientierte Programmiersprachen (OOP) 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 der relationalen Algebra.
  - Dieser konzeptionelle Widerspruch wurde in den 1990er Jahren als object-relational impedance mismatch bekannt.

# Grundlegende ORM-Techniken

- Im einfachsten Fall werden Klassen auf Tabellen abgebildet, jedes Objekt entspricht einer Tabellenzeile und für jedes Attribut wird eine Tabellenspalte reserviert.
- Die Identität eines Objekts entspricht dem Primärschlüssel der Tabelle.
- Hat ein Objekt eine Referenz auf ein anderes Objekt, so kann diese mit einer Fremdschlüssel-Primärschlüssel-Beziehung in der Datenbank dargestellt werden.





# Was ist die Java Persistence API (JPA)?

- Die Java Persistence API (JPA) ist eine Schnittstelle für Java-Anwendungen, die die Zuordnung und die Übertragung von Objekten zu Datenbankeinträgen vereinfacht.
- Sie vereinfacht die Lösung des Problems der objektrelationalen Abbildung, das darin besteht, Laufzeit-Objekte einer Java-Anwendung über eine einzelne Sitzung hinaus zu speichern (Persistenz), wobei relationale Datenbanken eingesetzt werden können, die ursprünglich nicht für objektorientierte Datenstrukturen vorgesehen sind.
- Die Java Persistence API wurde als Projekt der JSR 220 Expert Group entwickelt und im Mai 2006 erstmals veröffentlicht.

## Neben der API, welche im persistence-Package definiert ist, besteht die Java Persistence aus folgenden Komponenten...

- Eine Persistence Entity ist ein Plain Old Java Object (POJO), das üblicherweise auf eine einzelne Tabelle in der relationalen Datenbank abgebildet wird.
  - Instanzen dieser Klasse entsprechen hierbei den Zeilen der Tabelle.
  - Persistence Entities können je nach Designvorgabe als einfache Datenhaltungs-Klassen realisiert werden, die mit einem Struct in C vergleichbar sind, oder als Business-Objekte inklusive Business-Logik.
- Die Beziehungen zwischen den einzelnen Tabellen werden über objektrelationale Metadaten ausgedrückt.
- Diese sind entweder als Java-Annotationen angelegt und/oder in einer separaten XML-Datei abgelegt.



## Neben der API, welche im persistence-Package definiert ist, besteht die Java Persistence aus folgenden Komponenten...

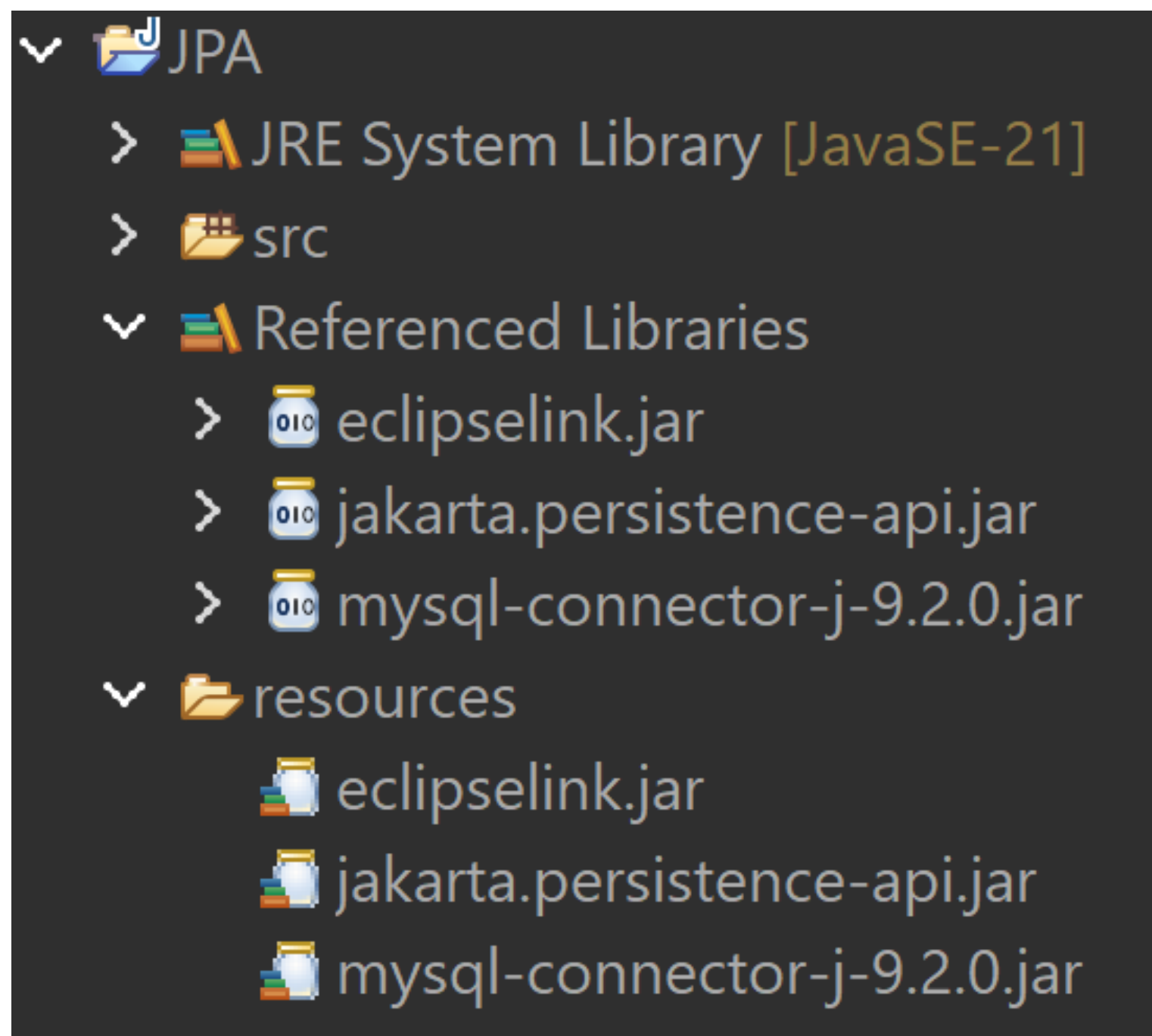
- Die Java Persistence Query Language (JPQL) wird genutzt, um Abfragen bezüglich der in der Datenbank gespeicherten Entitäten durchzuführen.
  - Diese Abfragen ähneln syntaktisch SQL-Abfragen, beziehen sich aber auf Entitäten statt auf Datenbanktabellen.
  - Die in JPQL formulierten Abfragen werden zur Laufzeit in SQL-Statements übersetzt, die vom Ziel-DBS verstanden werden.
  - Durch diese Abstraktion kann das Datenbanksystem transparent ausgetauscht werden, während die Java-Klassen vollständig erhalten bleiben.
- Im Unterschied zu JPQL erlaubt die JPA auch die Verwendung von direkten SQL-Abfragen, wobei diese als Native Query bezeichnet werden.
  - Beim Einsatz von Native Queries muss jedoch der Anwender selbst darauf achten, dass die Abfrage vom Zielsystem verstanden wird.

# Was ist EclipseLink?

- EclipseLink ist ein Open-Source-Persistenz- und ORM-Framework der Eclipse Foundation.
- EclipseLink ermöglicht die Interaktion mit verschiedenen
  - Datensystemen,
  - Datenbanken,
  - Web-Diensten und
  - Object XML Mappings (OXM).
- EclipseLink ist eine der führenden Implementierungen für Jakarta Persistence.

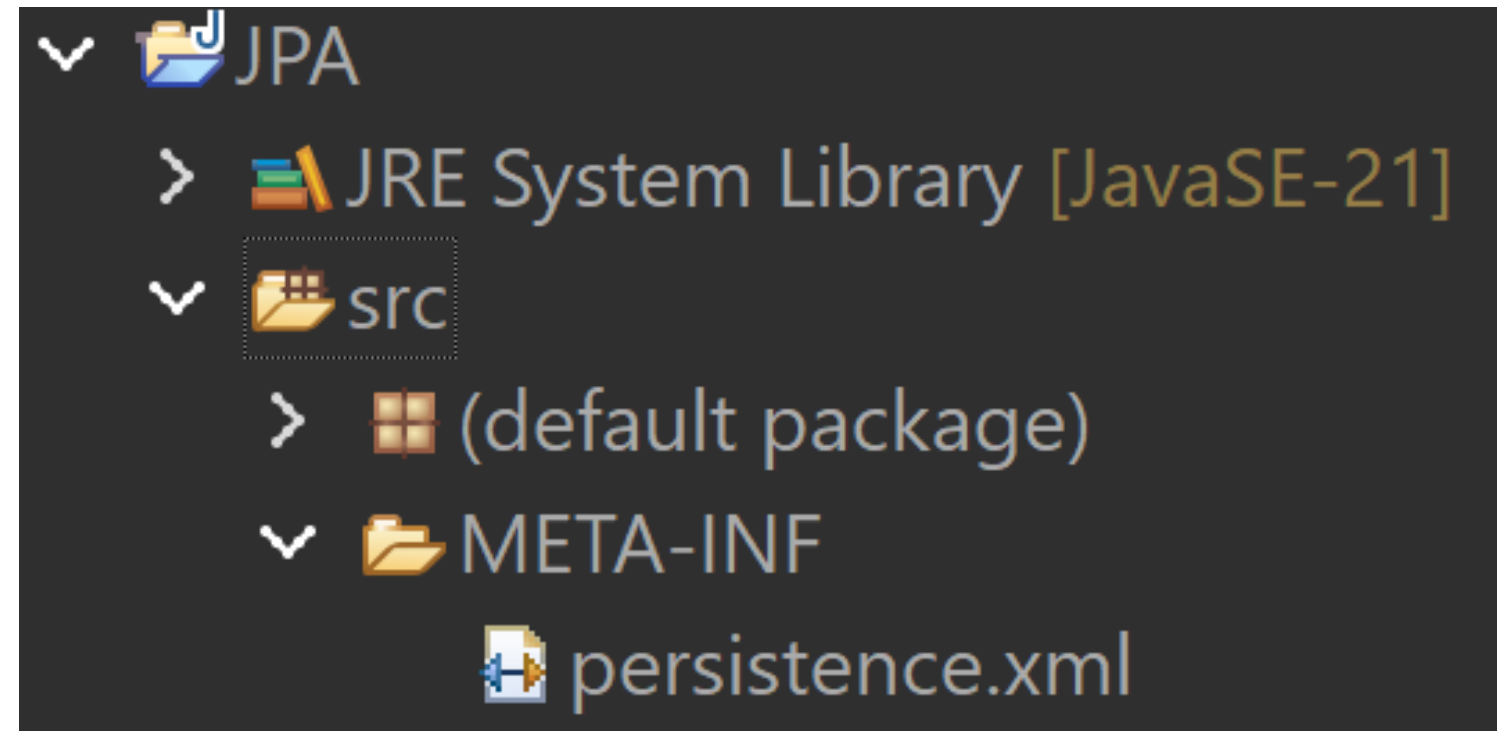
# Einbinden von EclipseLink und JPA in ein Eclipse-Projekt

- Zuerst die folgenden JARs in einen resources-Ordner kopieren und diese dann in den Build-Path integrieren.
  - JPA verwendet seinerseits JDBC für den Zugriff, so dass wir auch wieder den Connector benötigen:



# Konfigurieren des Zugriffs via XML-Datei

- Erstellen Sie nun einen META-INF Ordner unterhalb von src und dort eine Datei mit dem Namen persistence.xml:





# Konfigurieren des Zugriffs via XML-Datei

- Erstellen Sie nun einen META-INF Ordner unterhalb von src und dort eine Datei mit dem Namen persistence.xml:

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="3.0">
  <persistence-unit name="DM_JPA" transaction-type="RESOURCE_LOCAL">
    <class>Kunde</class>
    <class>Privatkunde</class>
    <class>Geschaeftskunde</class>
    <class>VIPKunde</class>
    <class>Rechnung</class>
    <properties>
      <!-- JDBC-Verbindung -->
      <property name="jakarta.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
      <property name="jakarta.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/DM_JPA?createDatabaseIfNotExist=true"/>
      <property name="jakarta.persistence.jdbc.user" value="root"/>
      <property name="jakarta.persistence.jdbc.password" value=""/>
      <!-- JPA-Implementierung per EclipseLink -->
      <property name="jakarta.persistence.provider" value="org.eclipse.persistence.jpa.PersistenceProvider"/>
      <property name="eclipselink.ddl-generation" value="create-tables"/>
      <property name="eclipselink.logging.level" value="FINE"/>
    </properties>
  </persistence-unit>
</persistence>
```

- Der Name der persistence-unit muss nachher genau mit dem Namen im Java-Quellcode übereinstimmen.

# Konfigurieren des Zugriffs via XML-Datei

- Erstellen Sie nun einen META-INF Ordner unterhalb von src und dort eine Datei mit dem Namen persistence.xml:

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="3.0">
  <persistence-unit name="DM_JPA" transaction-type="RESOURCE_LOCAL">
    <class>Kunde</class>
    <class>Privatkunde</class>
    <class>Geschaeftskunde</class>
    <class>VIPKunde</class>
    <class>Rechnung</class>
    <properties>
      <!-- JDBC-Verbindung -->
      <property name="jakarta.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
      <property name="jakarta.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/DM_JPA?createDatabaseIfNotExist=true"/>
      <property name="jakarta.persistence.jdbc.user" value="root"/>
      <property name="jakarta.persistence.jdbc.password" value=""/>
      <!-- JPA-Implementierung per EclipseLink -->
      <property name="jakarta.persistence.provider" value="org.eclipse.persistence.jpa.PersistenceProvider"/>
      <property name="eclipselink.ddl-generation" value="create-tables"/>
      <property name="eclipselink.logging.level" value="FINE"/>
    </properties>
  </persistence-unit>
</persistence>
```

- Die Klassen, die in die Datenbank persistiert werden sollen, muss man einzeln auflisten.



# Konfigurieren des Zugriffs via XML-Datei

- Erstellen Sie nun einen META-INF Ordner unterhalb von src und dort eine Datei mit dem Namen persistence.xml:

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="3.0">
  <persistence-unit name="DM_JPA" transaction-type="RESOURCE_LOCAL">
    <class>Kunde</class>
    <class>Privatkunde</class>
    <class>Geschaeftskunde</class>
    <class>VIPKunde</class>
    <class>Rechnung</class>
    <properties>
      <!-- JDBC-Verbindung -->
      <property name="jakarta.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
      <property name="jakarta.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/DM_JPA?createDatabaseIfNotExist=true"/>
      <property name="jakarta.persistence.jdbc.user" value="root"/>
      <property name="jakarta.persistence.jdbc.password" value=""/>
      <!-- JPA-Implementierung per EclipseLink -->
      <property name="jakarta.persistence.provider" value="org.eclipse.persistence.jpa.PersistenceProvider"/>
      <property name="eclipselink.ddl-generation" value="create-tables"/>
      <property name="eclipselink.logging.level" value="FINE"/>
    </properties>
  </persistence-unit>
</persistence>
```

- Der Treiber kommt aus der JAR mit dem JDBC-Connector.

# Konfigurieren des Zugriffs via XML-Datei

- Erstellen Sie nun einen META-INF Ordner unterhalb von src und dort eine Datei mit dem Namen persistence.xml:

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="3.0">
  <persistence-unit name="DM_JPA" transaction-type="RESOURCE_LOCAL">
    <class>Kunde</class>
    <class>Privatkunde</class>
    <class>Geschaeftskunde</class>
    <class>VIPKunde</class>
    <class>Rechnung</class>
    <properties>
      <!-- JDBC-Verbindung -->
      <property name="jakarta.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
      <property name="jakarta.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/DM_JPA?createDatabaseIfNotExist=true"/>
      <property name="jakarta.persistence.jdbc.user" value="root"/>
      <property name="jakarta.persistence.jdbc.password" value=""/>
      <!-- JPA-Implementierung per EclipseLink -->
      <property name="jakarta.persistence.provider" value="org.eclipse.persistence.jpa.PersistenceProvider"/>
      <property name="eclipselink.ddl-generation" value="create-tables"/>
      <property name="eclipselink.logging.level" value="FINE"/>
    </properties>
  </persistence-unit>
</persistence>
```

- Der Zugriff erfolgt lokal auf den TCP-Port 3306, den Standardport des MySQL/MariaDB-Servers.
- Die Datenbank DM\_JPA wird komplett neu angelegt, wenn sie noch nicht existiert. Dazu muss der angemeldete Benutzer natürlich berechtigt sein.



# Konfigurieren des Zugriffs via XML-Datei

- Erstellen Sie nun einen META-INF Ordner unterhalb von src und dort eine Datei mit dem Namen persistence.xml:

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="3.0">
  <persistence-unit name="DM_JPA" transaction-type="RESOURCE_LOCAL">
    <class>Kunde</class>
    <class>Privatkunde</class>
    <class>Geschaeftskunde</class>
    <class>VIPKunde</class>
    <class>Rechnung</class>
    <properties>
      <!-- JDBC-Verbindung -->
      <property name="jakarta.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
      <property name="jakarta.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/DM_JPA?createDatabaseIfNotExist=true"/>
      <property name="jakarta.persistence.jdbc.user" value="root"/>
      <property name="jakarta.persistence.jdbc.password" value=""/>
      <!-- JPA-Implementierung per EclipseLink -->
      <property name="jakarta.persistence.provider" value="org.eclipse.persistence.jpa.PersistenceProvider"/>
      <property name="eclipselink.ddl-generation" value="create-tables"/>
      <property name="eclipselink.logging.level" value="FINE"/>
    </properties>
  </persistence-unit>
</persistence>
```

- Mit diesem Benutzernamen und mit diesem Kennwort authentifiziert sich die Java-Anwendung gegen den Datenbankserver.
- Der Benutzer muss mit dem Kennwort entsprechend auf dem Datenbankserver vorhanden sein.

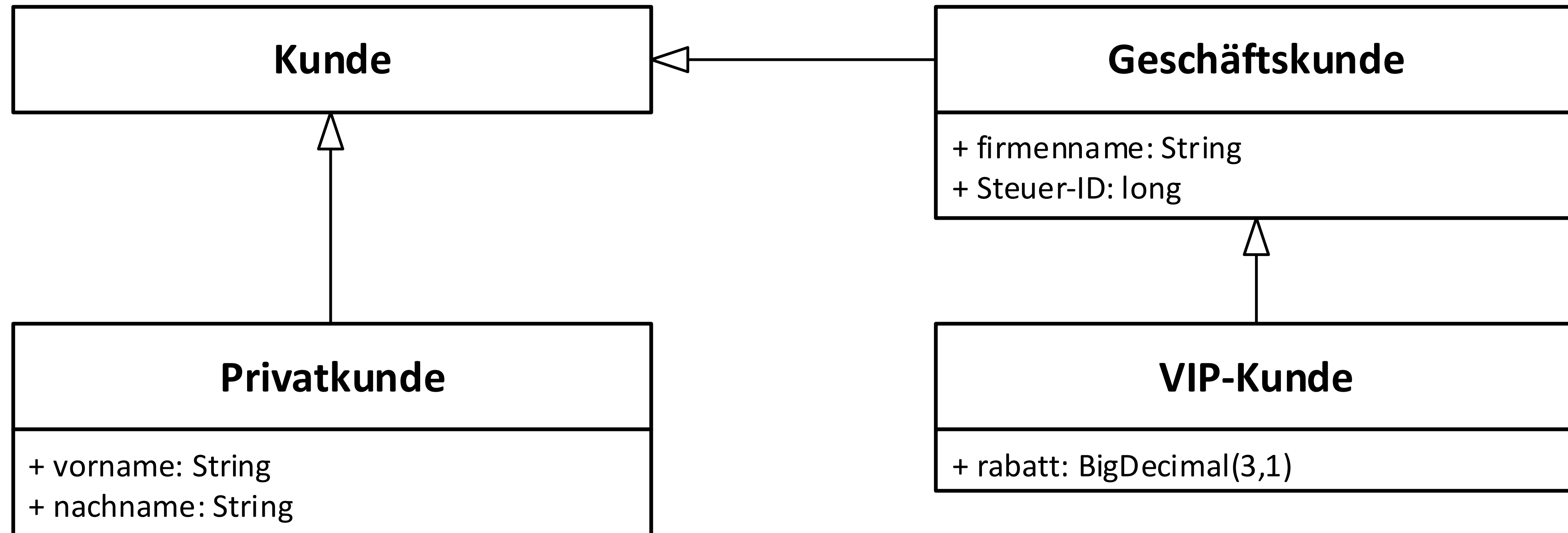
# Konfigurieren des Zugriffs via XML-Datei

- Erstellen Sie nun einen META-INF Ordner unterhalb von src und dort eine Datei mit dem Namen persistence.xml:

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="3.0">
  <persistence-unit name="DM_JPA" transaction-type="RESOURCE_LOCAL">
    <class>Kunde</class>
    <class>Privatkunde</class>
    <class>Geschaeftskunde</class>
    <class>VIPKunde</class>
    <class>Rechnung</class>
    <properties>
      <!-- JDBC-Verbindung -->
      <property name="jakarta.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
      <property name="jakarta.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/DM_JPA?createDatabaseIfNotExist=true"/>
      <property name="jakarta.persistence.jdbc.user" value="root"/>
      <property name="jakarta.persistence.jdbc.password" value=""/>
      <!-- JPA-Implementierung per EclipseLink -->
      <property name="jakarta.persistence.provider" value="org.eclipse.persistence.jpa.PersistenceProvider"/>
      <property name="eclipselink.ddl-generation" value="create-tables"/>
      <property name="eclipselink.logging.level" value="FINE"/>
    </properties>
  </persistence-unit>
</persistence>
```

- Sind Datenbank-Tabellen noch nicht vorhanden, so werden sie automatisch angelegt.

# Erstellen der Fachlogik in Java auf Basis der Aufgabenstellung



# Erstellen der Fachlogik in Java auf Basis der Aufgabenstellung: Der Kunde...

```
import jakarta.persistence.*;

@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Kunde {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    public Kunde() {} // nötig für autom. Objekterzeugung

    public Kunde(Long id) {
        setId(id);
    }

    // Getter & Setter
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
}
```



# Erstellen der Fachlogik in Java auf Basis der Aufgabenstellung: Der Privatkunde als spezieller Kunde...

```
import jakarta.persistence.Entity;

@Entity
public class Privatkunde extends Kunde {
    private String vorname;
    private String nachname;

    public Privatkunde() {} // nötig für autom. Objekterzeugung

    public Privatkunde(String vorname, String nachname) {
        setVorname(vorname);
        setNachname(nachname);
    }

    // Getter & Setter
    public String getVorname() { return vorname; }
    public void setVorname(String vorname) { this.vorname = vorname; }

    public String getNachname() { return nachname; }
    public void setNachname(String nachname) { this.nachname = nachname; }
}
```

# Erstellen der Fachlogik in Java auf Basis der Aufgabenstellung: Der Geschäftskunde als spezieller Kunde...

```
import jakarta.persistence.Entity;

@Entity
public class Geschaeftskunde extends Kunde {
    private String firmenname;
    private long steuerId;

    public Geschaeftskunde() {} // nötig für autom. Objekterzeugung

    public Geschaeftskunde(String firmenname, long steuerId) {
        setFirmenname(firmenname);
        setSteuerId(steuerId);
    }

    // Getter & Setter
    public String getFirmenname() { return firmenname; }
    public void setFirmenname(String firmenname) { this.firmenname = firmenname; }

    public long getSteuerId() { return steuerId; }
    public void setSteuerId(long steuerId) { this.steuerId = steuerId; }
}
```

## Erstellen der Fachlogik in Java auf Basis der Aufgabenstellung: Der VIP-Kunde als spezieller Geschäftskunde...

```
import java.math.BigDecimal;
import jakarta.persistence.Entity;

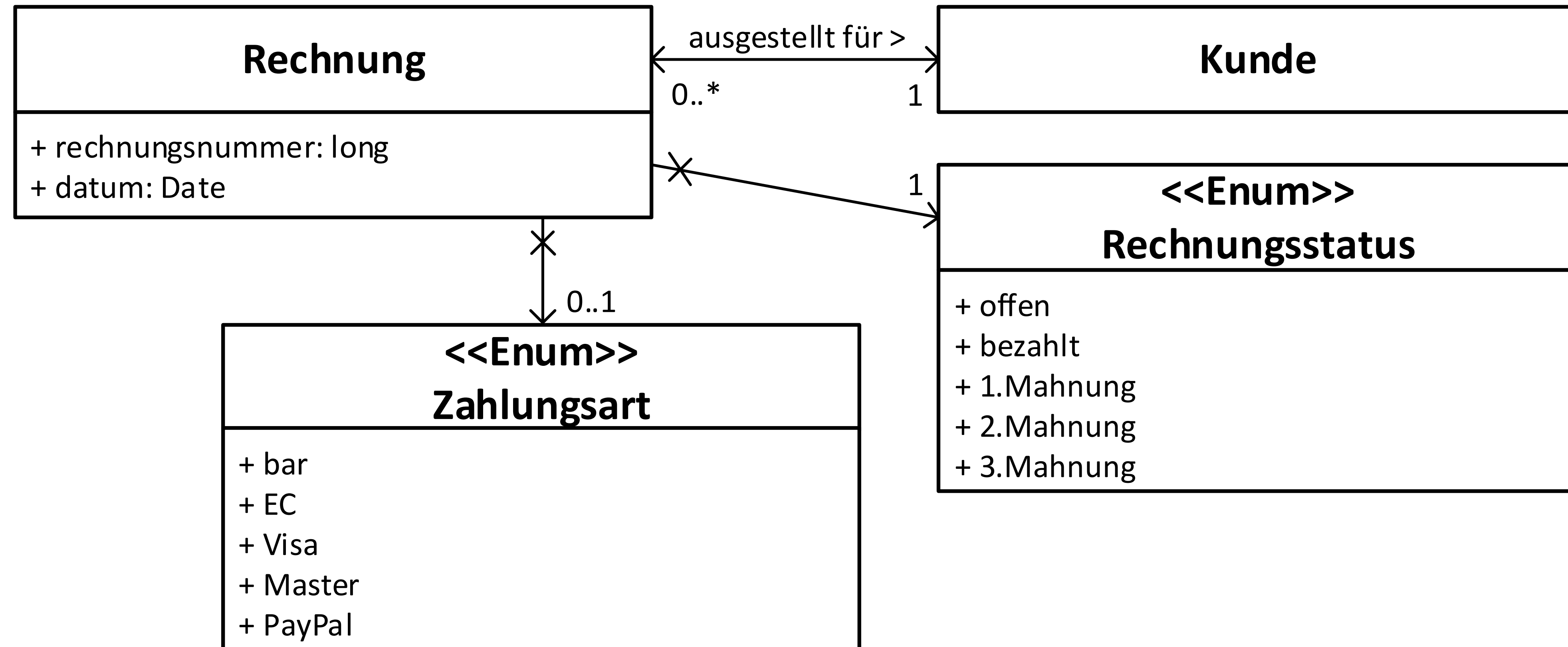
@Entity
public class VIPKunde extends Geschäftskunde {
    private BigDecimal rabatt;

    public VIPKunde() {} // nötig für autom. Objekterzeugung

    public VIPKunde(String firmenname, long steuerId, BigDecimal rabatt) {
        super(firmenname, steuerId);
        setRabatt(rabatt);
    }

    // Getter & Setter
    public BigDecimal getRabatt() { return rabatt; }
    public void setRabatt(BigDecimal rabatt) { this.rabatt = rabatt; }
}
```

# Erstellen der Fachlogik in Java auf Basis der Aufgabenstellung



# Erstellen der Fachlogik in Java auf Basis der Aufgabenstellung: Die Rechnung...

```
import java.util.Date;
import jakarta.persistence.*;

@Entity
public class Rechnung {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long rechnungsnummer;

    @Temporal(TemporalType.DATE)
    private Date datum;

    @ManyToMany
    private Kunde kunde;

    @Enumerated(EnumType.STRING)
    private Rechnungsstatus status;

    @Enumerated(EnumType.STRING)
    private Zahlungsart zahlungsart;

    public Rechnung() {} // nötig für autom. Objekterzeugung

    public Rechnung(Date datum, Kunde kunde, Rechnungsstatus status, Zahlungsart zahlungsart) {
        setDatum(datum);
        setKunde(kunde);
        setStatus(status);
        setZahlungsart(zahlungsart);
    }
}
```



# Erstellen der Fachlogik in Java auf Basis der Aufgabenstellung: Die Rechnung, der Rechnungsstatus und die Zahlungsart...

```
// Getter & Setter
public Long getRechnungsnummer() { return rechnungsnummer; }
public void setRechnungsnummer(Long rechnungsnummer) { this.rechnungsnummer = rechnungsnummer; }

public Date getDatum() { return datum; }
public void setDatum(Date datum) { this.datum = datum; }

public Kunde getKunde() { return kunde; }
public void setKunde(Kunde kunde) { this.kunde = kunde; }

public Rechnungsstatus getStatus() { return status; }
public void setStatus(Rechnungsstatus status) { this.status = status; }

public Zahlungsart getZahlungsart() { return zahlungsart; }
public void setZahlungsart(Zahlungsart zahlungsart) { this.zahlungsart = zahlungsart; }
}
```

```
public enum Zahlungsart {
    BAR, EC, VISA, MASTER, PAYPAL;
}
```

```
public enum Rechnungsstatus {
    OFFEN, BEZAHLT, ERSTE_MAHNUNG, ZWEITE_MAHNUNG, DRITTE_MAHNUNG;
}
```



# 4 Kunden und 4 Rechnungen in Java persistieren...

```
import java.math.BigDecimal;
import java.util.Date;
import jakarta.persistence.*;

public class Schreiben_01 {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("DM_JPA");
        EntityManager em = emf.createEntityManager();

        // Transaktion starten und Kunden speichern
        em.getTransaction().begin();
        Privatkunde pk1 = new Privatkunde("Max", "Mustermann");
        Privatkunde pk2 = new Privatkunde("Anna", "Schmidt");
        Geschaeftskunde gk1 = new Geschaeftskunde("Muster GmbH", 123456789);
        VIPKunde vk1 = new VIPKunde("VIP Solutions", 987654321, BigDecimal.valueOf(5.0));
        em.persist(pk1);
        em.persist(pk2);
        em.persist(gk1);
        em.persist(vk1);
        em.getTransaction().commit();
        System.out.println("☑ Kunden wurden erfolgreich gespeichert!\n");

        // 2 Rechnungen speichern
        em.getTransaction().begin();
        Rechnung r1 = new Rechnung(new Date(), pk1, Rechnungsstatus.OFFEN, Zahlungsart.PAYPAL);
        Rechnung r2 = new Rechnung(new Date(), pk2, Rechnungsstatus.BEZAHLT, Zahlungsart.VISA);
        Rechnung r3 = new Rechnung(new Date(), gk1, Rechnungsstatus.ERSTE_MAHNUNG, Zahlungsart.EC);
        Rechnung r4 = new Rechnung(new Date(), vk1, Rechnungsstatus.OFFEN, Zahlungsart.MASTER);
        em.persist(r1);
        em.persist(r2);
        em.persist(r3);
        em.persist(r4);
        em.getTransaction().commit();
        System.out.println("☑ Rechnungen wurden erfolgreich gespeichert!\n");
    }
}
```

# 4 Kunden und 4 Rechnungen in Java persistieren...

[EL Fine]: server: 2025-02-13 13:59:59.776--Thread(Thread[#1,main,5,main])--Configured server platform:  
org.eclipse.persistence.platform.server.NoServerPlatform

[EL Config]: metadata: 2025-02-13 13:59:59.839--ServerSession(928466577)--Thread(Thread[#1,main,5,main])--The access type for the persistent class [class Kunde] is set to [FIELD].

[EL Config]: metadata: 2025-02-13 13:59:59.846--ServerSession(928466577)--Thread(Thread[#1,main,5,main])--The access type for the persistent class [class Privatkunde] is set to [FIELD].

[EL Config]: metadata: 2025-02-13 13:59:59.848--ServerSession(928466577)--Thread(Thread[#1,main,5,main])--The access type for the persistent class [class Geschaeftskunde] is set to [FIELD].

[EL Config]: metadata: 2025-02-13 13:59:59.848--ServerSession(928466577)--Thread(Thread[#1,main,5

...

[EL Fine]: sql: 2025-02-13 14:00:00.43--ClientSession(1452442375)--Connection(1928301845)--Thread(Thread[#1,main,5,main])--INSERT INTO RECHNUNG (DATUM, STATUS, ZAHLUNGSART, KUNDE\_ID) VALUES (?, ?, ?, ?)

bind => [2025-02-13, BEZAHLT, VISA, 2]

[EL Fine]: sql: 2025-02-13 14:00:00.438--ClientSession(1452442375)--Connection(1928301845)--Thread(Thread[#1,main,5,main])--INSERT INTO RECHNUNG (DATUM, STATUS, ZAHLUNGSART, KUNDE\_ID) VALUES (?, ?, ?, ?)

bind => [2025-02-13, ERSTE\_MAHNUNG, EC, 3]

[EL Fine]: sql: 2025-02-13 14:00:00.439--ClientSession(1452442375)--Connection(1928301845)--Thread(Thread[#1,main,5,main])--INSERT INTO RECHNUNG (DATUM, STATUS, ZAHLUNGSART, KUNDE\_ID) VALUES (?, ?, ?, ?)

bind => [2025-02-13, OFFEN, MASTER, 4]

[EL Fine]: sql: 2025-02-13 14:00:00.44--ClientSession(1452442375)--Connection(1928301845)--Thread(Thread[#1,main,5,main])--INSERT INTO RECHNUNG (DATUM, STATUS, ZAHLUNGSART, KUNDE\_ID) VALUES (?, ?, ?, ?)

bind => [2025-02-13, OFFEN, PAYPAL, 1]

☒ Rechnungen wurden erfolgreich gespeichert!

# Was sehen wir in der Datenbank?

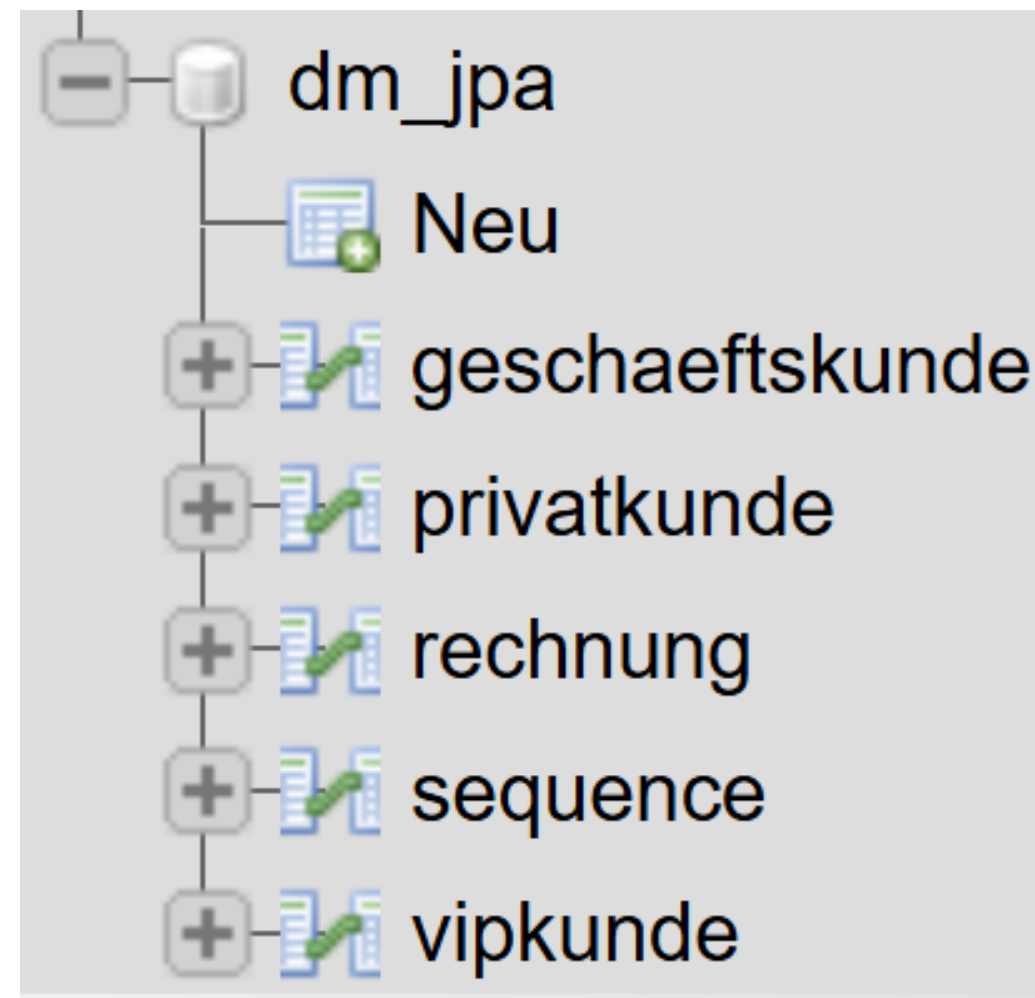


Tabelle	Aktion							Datensätze	Typ	Kollation	Größe
geschaeftskunde	★	Anzeigen	Struktur	Suche	Einfügen	Leeren	Löschen	1	InnoDB	utf8mb4_general_ci	16,0 KiB
privatkunde	★	Anzeigen	Struktur	Suche	Einfügen	Leeren	Löschen	2	InnoDB	utf8mb4_general_ci	16,0 KiB
rechnung	★	Anzeigen	Struktur	Suche	Einfügen	Leeren	Löschen	4	InnoDB	utf8mb4_general_ci	16,0 KiB
sequence	★	Anzeigen	Struktur	Suche	Einfügen	Leeren	Löschen	1	InnoDB	utf8mb4_general_ci	16,0 KiB
vipkunde	★	Anzeigen	Struktur	Suche	Einfügen	Leeren	Löschen	1	InnoDB	utf8mb4_general_ci	16,0 KiB
5 Tabellen	Gesamt							9	InnoDB	utf8mb4_general_ci	80,0 KiB



# Was sehen wir in der Datenbank?

✔ Zeige Datensätze 0 - 0 (1 insgesamt, Die Abfrage dauerte 0,0001 Sekunden.)

`SELECT * FROM `geschaeftskunde``

☐ Messen [ [Inline bearbeiten](#) ] [ [Bearbeiten](#) ] [ [SQL erklären](#) ] [ [PHP-Code erzeugen](#) ] [ [Aktualisieren](#) ]

☐ Alles anzeigen | Anzahl der Datensätze: 

25 ▾

 Zeilen filtern: 

Diese Tabelle durchsuchen

Zusätzliche Optionen

↔ T ↔ ▾

☐  Bearbeiten  Kopieren  Löschen

ID	FIRMENNAME	STEUERID
3	Muster GmbH	123456789

# Was sehen wir in der Datenbank?

✔ Zeige Datensätze 0 - 1 (2 insgesamt, Die Abfrage dauerte 0,0003 Sekunden.)

```
SELECT * FROM `privatkunde`
```

☐ Messen [ [Inline bearbeiten](#) ] [ [Bearbeiten](#) ] [ [SQL erklären](#) ] [ [PHP-Code erzeugen](#) ] [ [Aktualisieren](#) ]

☐ Alles anzeigen | Anzahl der Datensätze: 

25 ▾

 Zeilen filtern: 

Diese Tabelle durchsuchen

Zusätzliche Optionen

<div>↔ T ↔ ▾</div>			ID	NACHNAME	VORNAME
<input type="checkbox"/>	Bearbeiten	Kopieren	Löschen	1 Mustermann	Max
<input type="checkbox"/>	Bearbeiten	Kopieren	Löschen	2 Schmidt	Anna

# Was sehen wir in der Datenbank?

✔ Zeige Datensätze 0 - 0 (1 insgesamt, Die Abfrage dauerte 0,0002 Sekunden.)

```
SELECT * FROM `vipkunde`
```

☐ Messen [ [Inline bearbeiten](#) ] [ [Bearbeiten](#) ] [ [SQL erklären](#) ] [ [PHP-Code erzeugen](#) ] [ [Aktualisieren](#) ]

☐ Alles anzeigen

Anzahl der Datensätze: 

25

 ▾

Zeilen filtern: 


Diese Tabelle durchsuchen

Zusätzliche Optionen

↔ T ↔ ▾	ID	FIRMENNAME	RABATT	STEUERID
<input type="checkbox"/> Bearbeiten  Kopieren  Löschen	4	VIP Solutions	5	987654321



# Was sehen wir in der Datenbank?





 Zeige Datensätze 0 - 3 (4 insgesamt, Die Abfrage dauerte 0,0003 Sekunden.)

`SELECT * FROM `rechnung``

☐ Messen [ [Inline bearbeiten](#) ] [ [Bearbeiten](#) ] [ [SQL erklären](#) ] [ [PHP-Code erzeugen](#) ] [ [Aktualisieren](#) ]

☐ Alles anzeigen | Anzahl der Datensätze:  Zeilen filtern:  Nach Schlüssel sortieren:

Zusätzliche Optionen

				RECHNUNGSNUMMER	DATUM	STATUS	ZAHLUNGSART	KUNDE_ID
<input type="checkbox"/>	 Bearbeiten	 Kopieren	 Löschen	1	2025-02-13	BEZAHLT	VISA	2
<input type="checkbox"/>	 Bearbeiten	 Kopieren	 Löschen	2	2025-02-13	ERSTE_MAHNUNG	EC	3
<input type="checkbox"/>	 Bearbeiten	 Kopieren	 Löschen	3	2025-02-13	OFFEN	MASTER	4
<input type="checkbox"/>	 Bearbeiten	 Kopieren	 Löschen	4	2025-02-13	OFFEN	PAYPAL	1

# Geschriebene Daten wieder in Java einlesen...

```
import java.util.List;
import jakarta.persistence.*;

public class Lesen_01 {
    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("DM_JPA");
        EntityManager em = emf.createEntityManager();

        // Alle Privatkunden aus der Datenbank abrufen
        List<Privatkunde> privatkunden = em.createQuery("SELECT p FROM Privatkunde p", Privatkunde.class).getResultList();
        System.out.println("✂ Privatkunden in der Datenbank:");
        for (Privatkunde p : privatkunden) {
            System.out.println(" - ID: " + p.getId() + ", Name: " + p.getVorname() + " " + p.getNachname());
        }
        System.out.println();

        // Alle Geschäftskunden abrufen
        List<Geschaeftskunde> geschaeftskunden = em.createQuery("SELECT g FROM Geschaeftskunde g", Geschaeftskunde.class).getResultList();
        System.out.println("✂ Geschäftskunden in der Datenbank:");
        for (Geschaeftskunde g : geschaeftskunden) {
            System.out.println(" - ID: " + g.getId() + ", Firmenname: " + g.getFirmenname() + ", Steuer-ID: " + g.getSteuerId());
        }
        System.out.println();

        // Alle VIP-Kunden abrufen
        List<VIPKunde> vipkunden = em.createQuery("SELECT v FROM VIPKunde v", VIPKunde.class).getResultList();
        System.out.println("✂ VIP-Kunden in der Datenbank:");
        for (VIPKunde v : vipkunden) {
            System.out.println(" - ID: " + v.getId() + ", Firmenname: " + v.getFirmenname() + ", Rabatt: " + v.getRabatt() + "%");
        }
        System.out.println();
    }
}
```

# Geschriebene Daten wieder in Java einlesen...

```
// Alle Rechnungen abrufen
List<Rechnung> rechnungen = em.createQuery("SELECT r FROM Rechnung r", Rechnung.class).getResultList();
System.out.println("⚡ Rechnungen in der Datenbank:");
for (Rechnung r : rechnungen) {
    System.out.println(" - Rechnungsnummer: " + r.getRechnungsnummer() +
        ", Datum: " + r.getDatum() +
        ", Kunde-ID: " + r.getKunde().getId() +
        ", Kunden-Art: " + r.getKunde().getClass().getSimpleName() +
        ", Status: " + r.getStatus() +
        ", Zahlungsart: " + r.getZahlungsart());
}
System.out.println();

// 7. EntityManager schließen
em.close();
emf.close();
}
}
```

# 4 Kunden und 4 Rechnungen in Java persistieren...

...

📌 Privatkunden in der Datenbank:

- ID: 1, Name: Max Mustermann

- ID: 2, Name: Anna Schmidt

...

📌 Geschäftskunden in der Datenbank:

- ID: 3, Firmenname: Muster GmbH, Steuer-ID: 123456789

- ID: 4, Firmenname: VIP Solutions, Steuer-ID: 987654321

...

📌 VIP-Kunden in der Datenbank:

- ID: 4, Firmenname: VIP Solutions, Rabatt: 5%

...

📌 Rechnungen in der Datenbank:

- Rechnungsnummer: 1, Datum: Thu Feb 13 00:00:00 CET 2025, Kunde-ID: 2, Kunden-Art: Privatkunde, Status: BEZAHLT, Zahlungsart: VISA

- Rechnungsnummer: 2, Datum: Thu Feb 13 00:00:00 CET 2025, Kunde-ID: 3, Kunden-Art: Geschäftskunde, Status: ERSTE\_MAHNUNG, Zahlungsart: EC

- Rechnungsnummer: 3, Datum: Thu Feb 13 00:00:00 CET 2025, Kunde-ID: 4, Kunden-Art: VIPKunde, Status: OFFEN, Zahlungsart: MASTER

- Rechnungsnummer: 4, Datum: Thu Feb 13 00:00:00 CET 2025, Kunde-ID: 1, Kunden-Art: Privatkunde, Status: OFFEN, Zahlungsart: PAYPAL