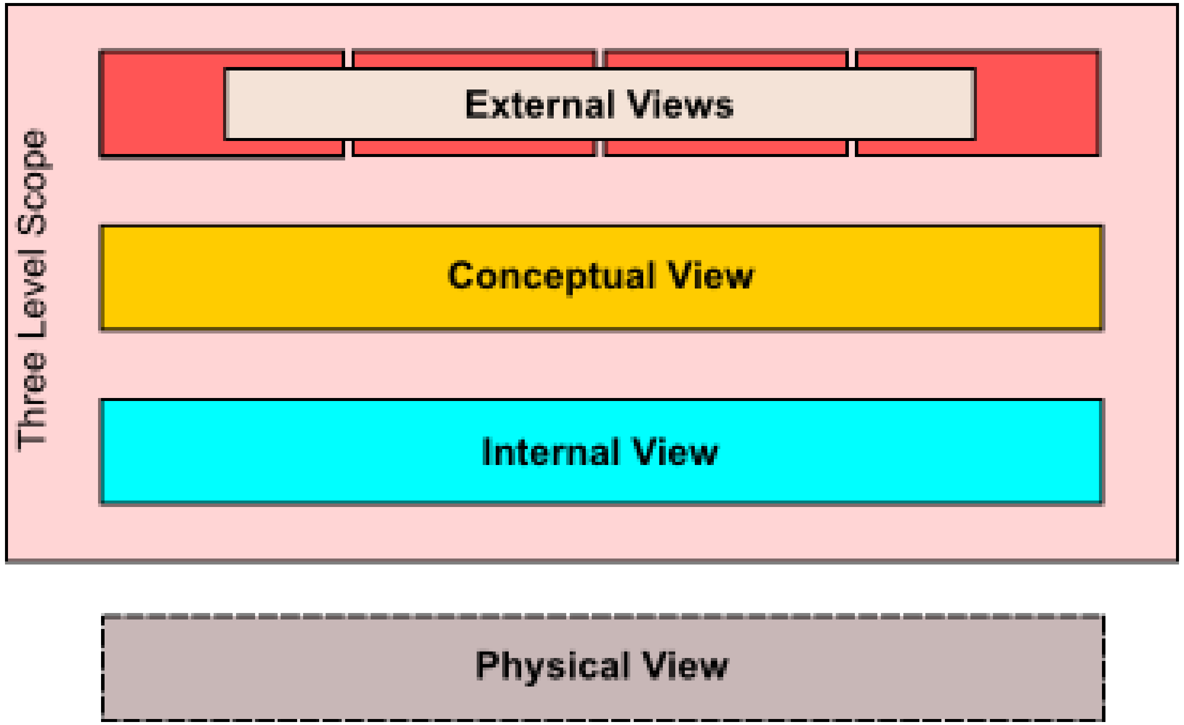
Programming in Java

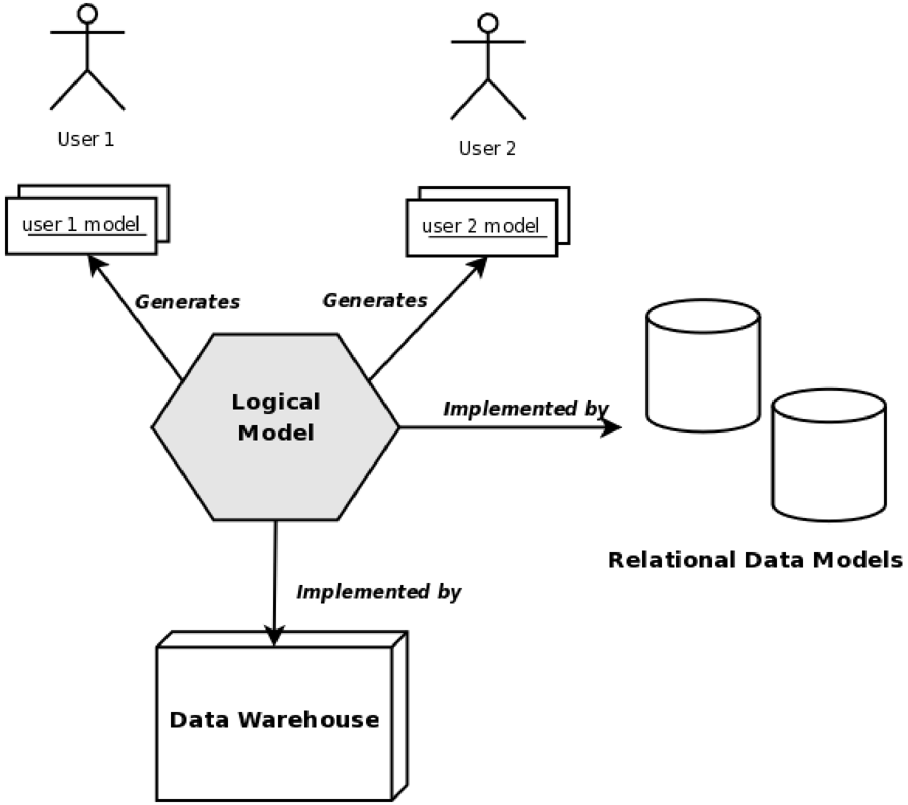
4. Java Persistence API

Ways of Thinking about Data

- Data exists in three layers
- The internal model of the data a group of users have
  - Their view of the data based on what they need in their context
  - These are often the informal business objects
- The conceptual model
  - Often called the logical model
  - A rigorously defined description of the data
  - Often represented in some schema
- The implementation model
  - How the data is organized for specific uses
  - Relational model, dimensional model, etc

ANSII-SPARC 3 Level Architecture





ANSII-SPARC 3 Level Architecture

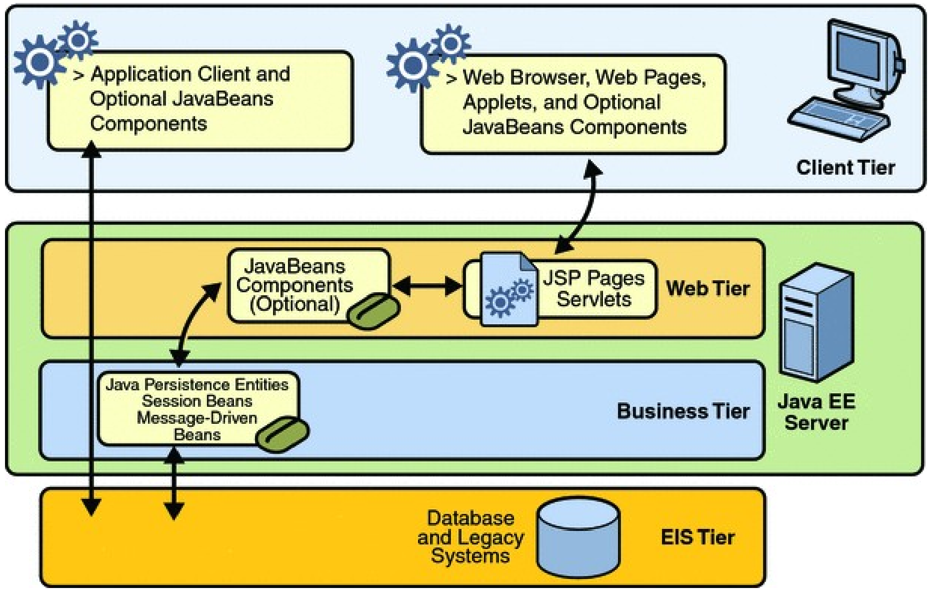
- External views
  - Not consistent across groups of users
  - We can’t model data to one external view and have it usable by other external views
- Conceptual view
  - Data is defined using predicates
  - “For the purpose of this project, a customer is defined to be”
  - Common approach in science and law where we need to agree on precisely what terms mean
- Internal View
  - How we organize the conceptual definitions for use
  - Relational models organize optimize our data layout for transactional processing
  - Dimensional models optimize the data for exploratory queries in data warehouses

Coupling and Suppleness

- Coupling the application to the internal layer is a common mistake
  - This produces coupling between the client and relational model
  - If the relational model is changed, the client may break
- There are other forms of implementation models
  - Document data bases like Mongodb
  - We still don’t want to do any coupling
- We usually introduce a layer of indirection
  - A persistence backing service
  - Data is passed in a logical schema to and from the client
  - The backing service then maps to the correct underlying implementation
  - Data is received from the client and returned in a schema that is implementation independent

The ORM Problem

- Originally relational data was the only game in town and applications had to connect to existing corporate data centers which were all relations data bases
  - Mapping the conceptual objects in the external view to relational data base is called the Object Relational Mapping Problem
  - A lot of Java Enterprise edition historically wrestled with this issue

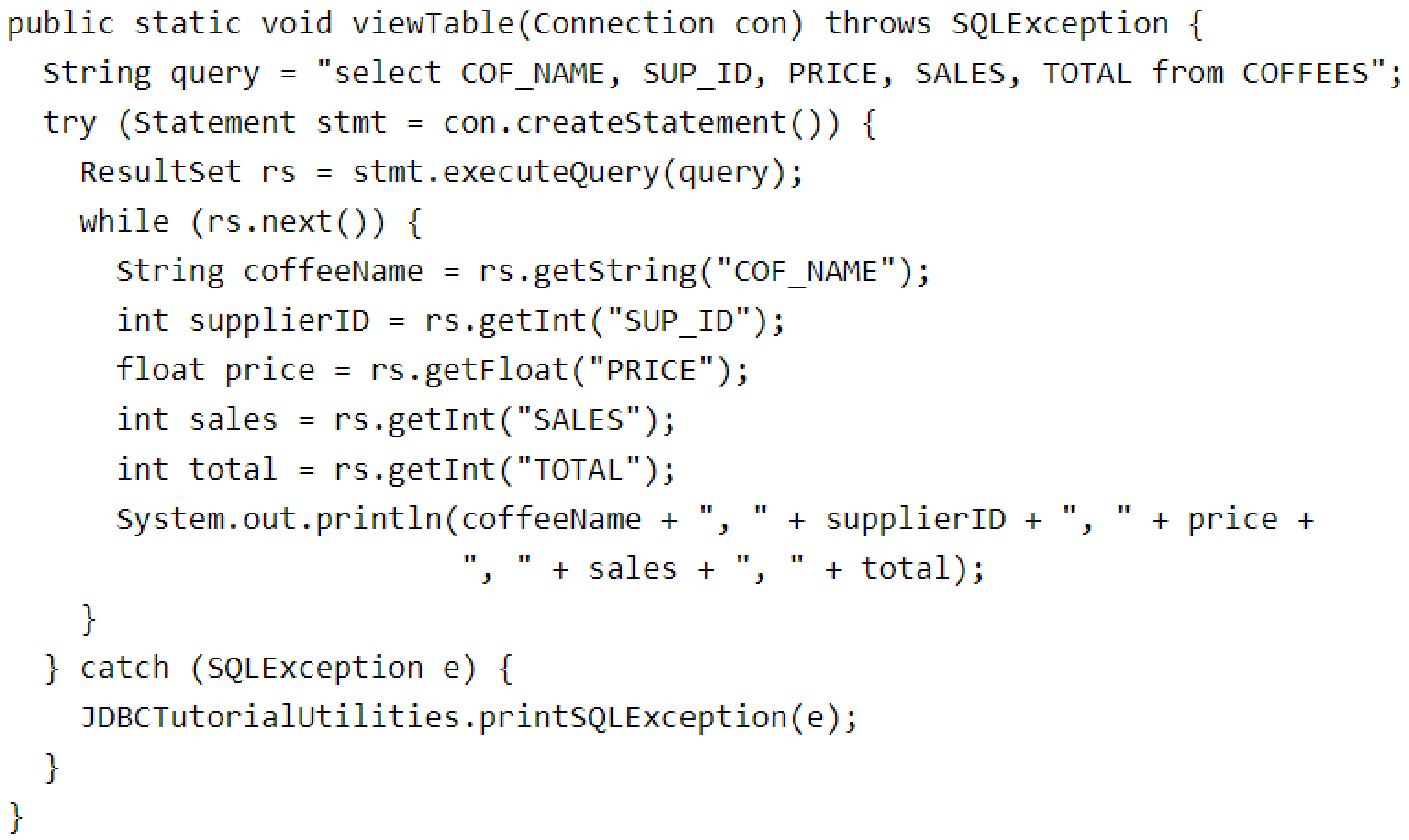


> Application Client and Optional JavaBeans Components
> Web Browser, Web Pages, Applets, and Optional JavaBeans Components
Client Tier
JavaBeans Components (Optional)
JSP Pages Servlets
Web Tier
Java Persistence Entities Session Beans Message-Driven Beans
Business Tier
Java EE Server
Database and Legacy Systems
EIS Tier

The ORM Problem

- The problem was that the java data objects had to be mapped to the underlying relational database
  - This was originally handled by JDBC code
  - Tried to create a layer of abstraction between the actual database and the Java code
  - But the Java code had to execute SQL statements and interpret the result
- The resulting code was often brittle and tightly coupled to the database
  - Changes to the underlying database could break a lot of Java code
- The problem is that users thought in terms of “account objects”
  - At the internal level,an account is a record in a table
  - ORM is intended to map the account object to the right table, and vice versa

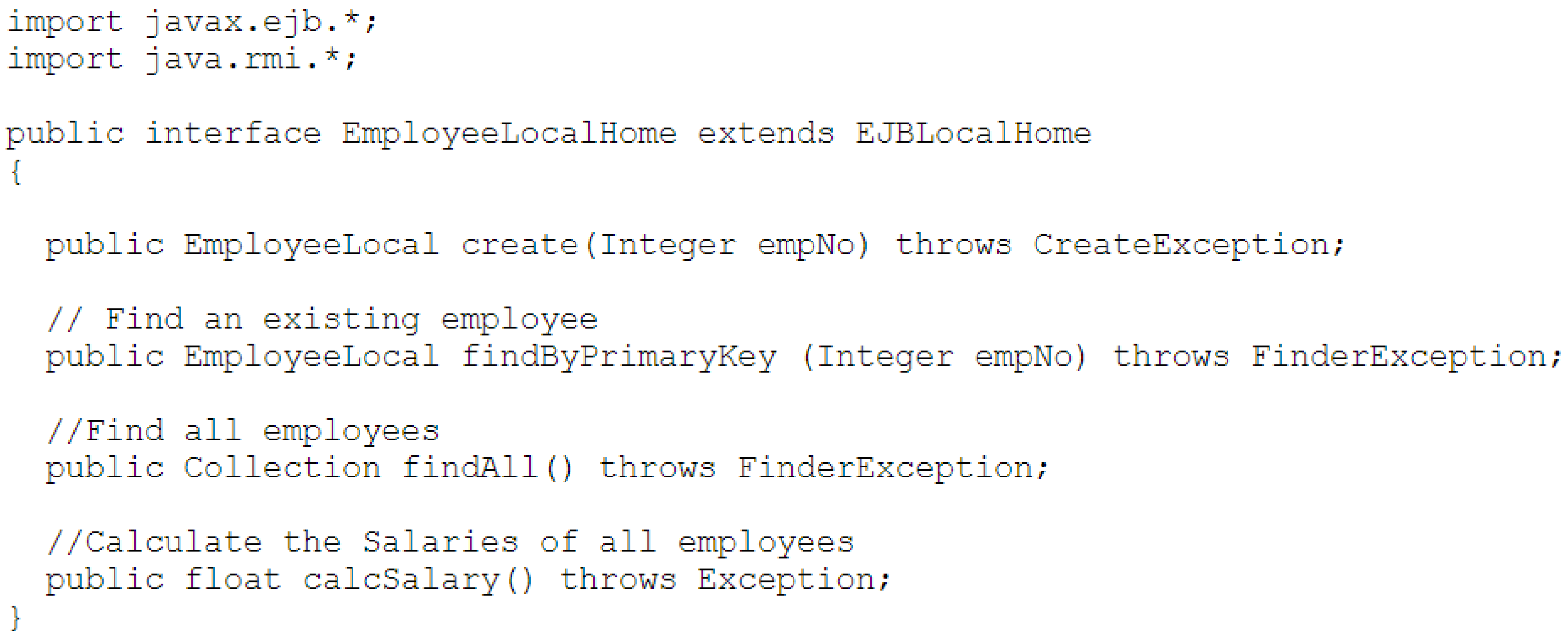
JDBC Example from Oracle Docs



```
public static void viewTable(Connection con) throws SQLException {
  String query = "select COF_NAME, SUP_ID, PRICE, SALES, TOTAL from COFFEES";
  try (Statement stmt = con.createStatement()) {
    ResultSet rs = stmt.executeQuery(query);
    while (rs.next()) {
      String coffeeName = rs.getString("COF_NAME");
      int supplierID = rs.getInt("SUP_ID");
      float price = rs.getFloat("PRICE");
      int sales = rs.getInt("SALES");
      int total = rs.getInt("TOTAL");
      System.out.println(coffeeName + ", " + supplierID + ", " + price +
                         ", " + sales + ", " + total);
    }
  } catch (SQLException e) {
    JDBCTutorialUtilities.printSQLException(e);
  }
}
```

J2EE Entity Beans

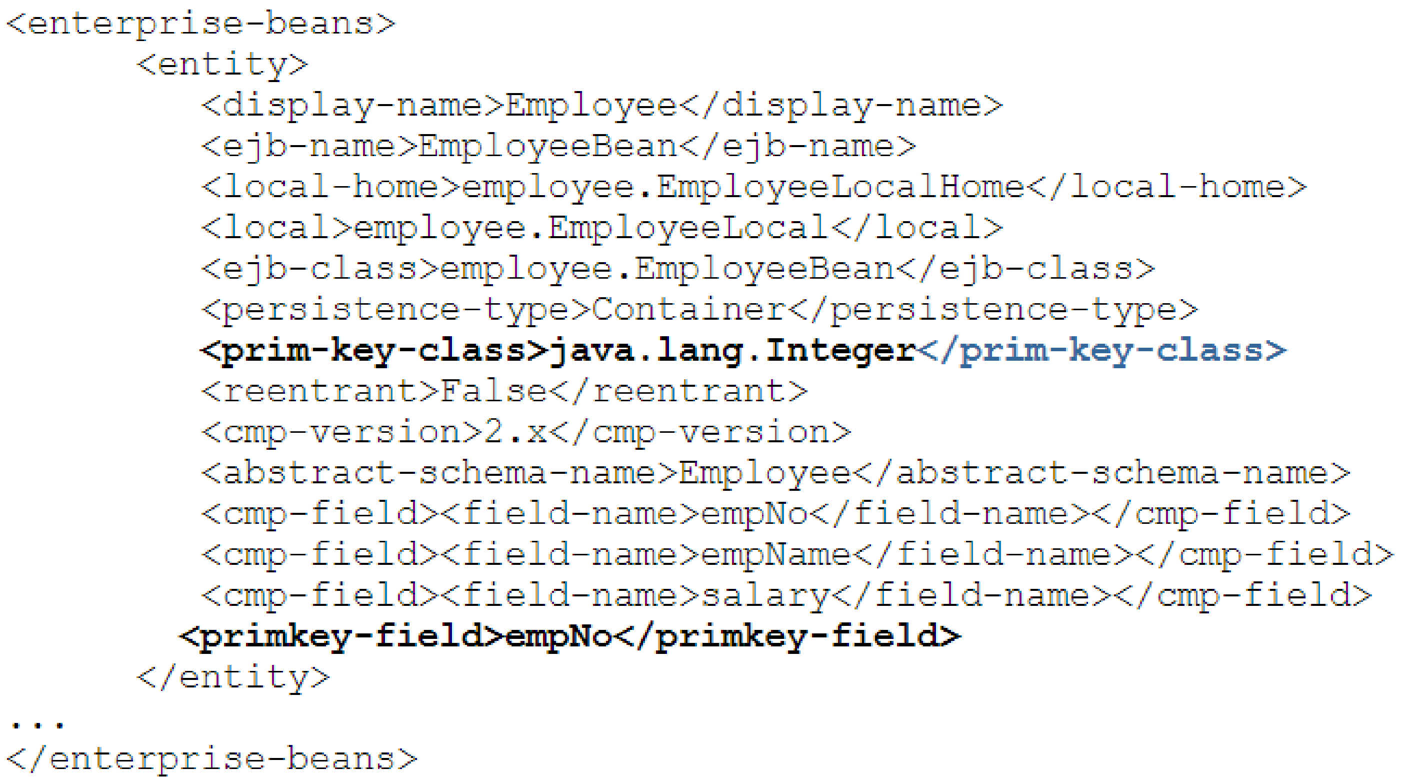
- The alternative to directly accessing the database from a POJO was implemented in J2EE as “Entity Beans”
  - POJO – “Plain old Java Object”



```
import javax.ejb.*;
import java.rmi.*;

public interface EmployeeLocalHome extends EJBLocalHome
{

  public EmployeeLocal create(Integer empNo) throws CreateException;

  // Find an existing employee
  public EmployeeLocal findByPrimaryKey (Integer empNo) throws FinderException;

  //Find all employees
  public Collection findAll() throws FinderException;

  //Calculate the Salaries of all employees
  public float calcSalary() throws Exception;
}
```

J2EE Entity Beans

- The underlying database representation was not required in the Java code
  - Instead, it was moved into XML configuration files
  - These became very difficult to work with



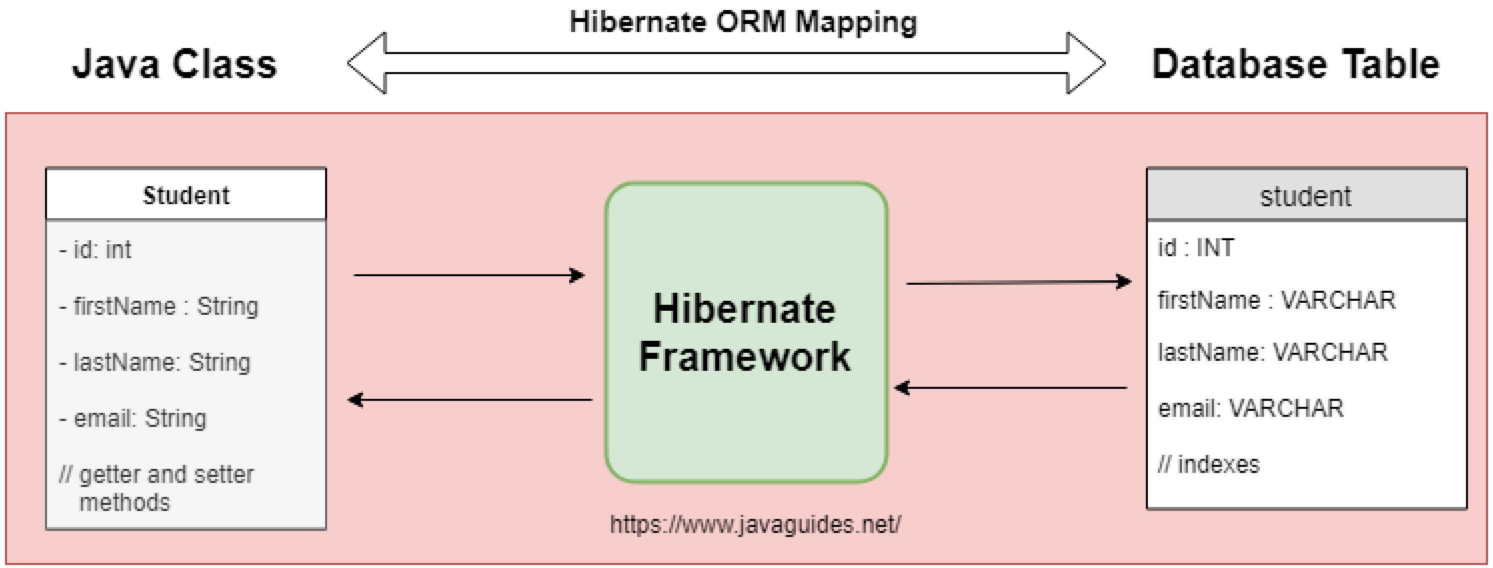
```
<enterprise-beans>
      <entity>
         <display-name>Employee</display-name>
         <ejb-name>EmployeeBean</ejb-name>
         <local-home>employee.EmployeeLocalHome</local-home>
         <local>employee.EmployeeLocal</local>
         <ejb-class>employee.EmployeeBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>java.lang.Integer</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-version>2.x</cmp-version>
         <abstract-schema-name>Employee</abstract-schema-name>
         <cmp-field><field-name>empNo</field-name></cmp-field>
         <cmp-field><field-name>empName</field-name></cmp-field>
         <cmp-field><field-name>salary</field-name></cmp-field>
        <primkey-field>empNo</primkey-field>
      </entity>
...
</enterprise-beans>
```

The JPA Standard

- By the time EJB 3.0 came around the JPA specification had been released
  - Like the rest of the EE specifications, it defined an interface
  - The interface standardizes how Java interacts with persistent data
  - Utilizes the concept of an “entity”
  - Abstracts out the general concept of a query to be independent of the underlying database
- The intent was to decouple the way the code referred to persistent objects from how they were actually implemented
  - The class that implements the interface does the work of mapping the data
  - If a different type of data persistence is used
  - Then the client code is kept the same, talking to the JPA interface
  - But the class that implements the interface is changed.

The JPA Standard

- Like other specifications, JPA defines an interface
  - This is implemented in various ORM products
  - Hibernate is a popular implementation



Hibernate ORM Mapping

Java Class

Database Table

Student

- id: int

- firstName : String

- lastName: String

- email: String

// getter and setter methods

Hibernate Framework

student

id : INT

firstName : VARCHAR

lastName: VARCHAR

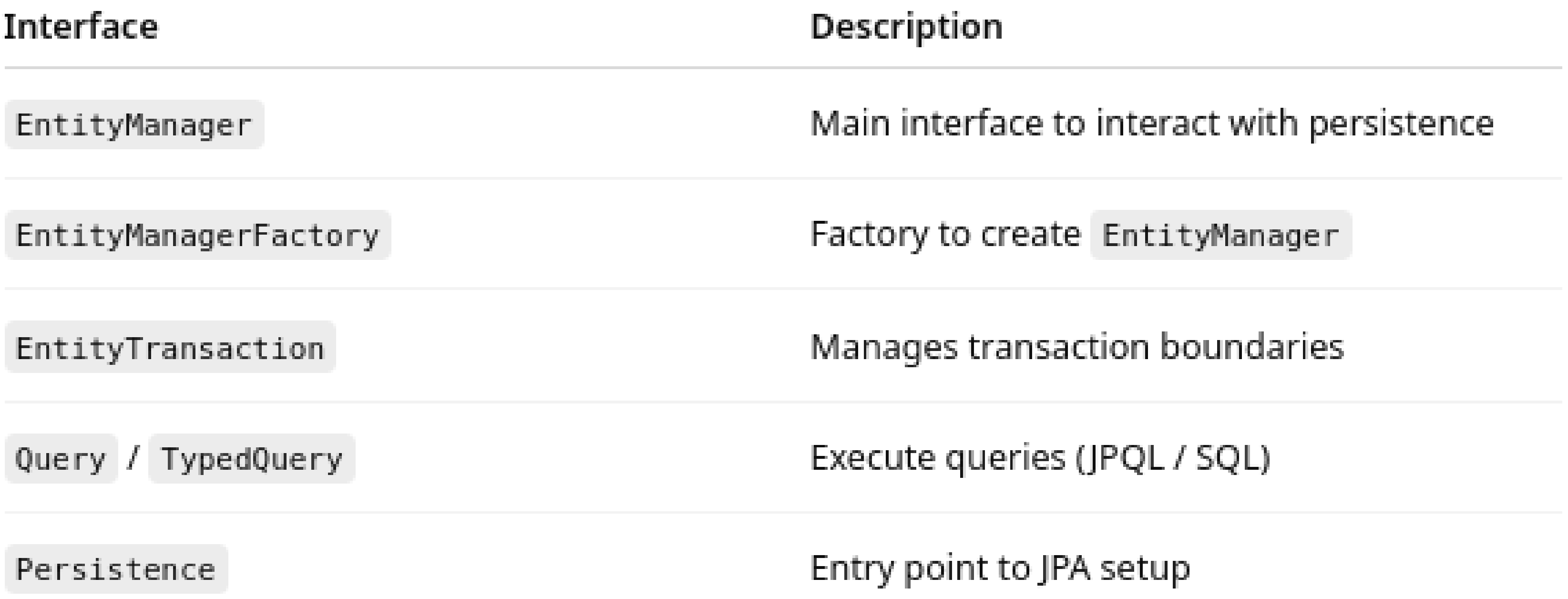
email: VARCHAR

// indexes

https://www.javaguides.net/

JPA Architecture and Interfaces

- JPA is made up of several interfaces
  - These define how code interacts with the the persistence layer
  - The persistence layer is where the code that actually interacts with the database lives
- The main interfaces are summarized in the table below



Interface	Description
`EntityManager`	Main interface to interact with persistence
`EntityManagerFactory`	Factory to create `EntityManager`
`EntityTransaction`	Manages transaction boundaries
`Query` / `TypedQuery`	Execute queries (JPQL / SQL)
`Persistence`	Entry point to JPA setup

EntityManager

- EntityManager is the main interface provided by the Java Persistence API (JPA) to
  - Allows code to interact with a database using Java objects instead of SQL.
  - Acts as translation layer between a Java application and a database.
- Provides basic CRUD functionality
  - Create new records (entities)
  - Retrieve records
  - Update records
  - Delete records
- Run queries using JPQL (Java Persistence Query Language)
  - JPQL works with entity classes and their fields, not table names or columns.
  - Decouples the logic of the query with the physical layout of the database
  - JPQL is database-agnostic.

EntityManagerFactory

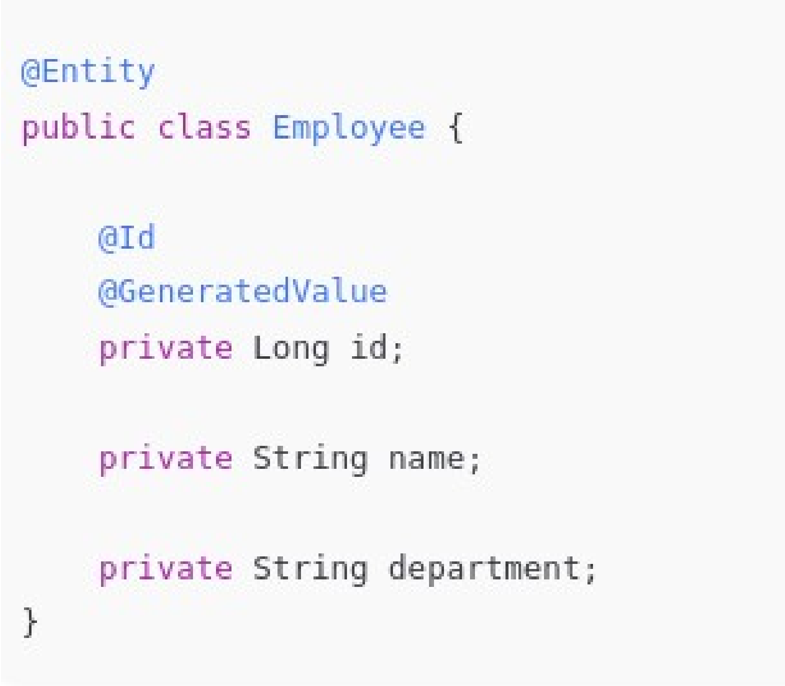
- Represents a thread-safe, heavyweight object that is responsible for:
  - Creating and managing EntityManager instances
  - Holding database configuration and metadata
  - Caching entity mappings
  - Managing the underlying connection pool (via the JPA provider)
- One EntityManagerFactory is created per application
  - It must be closed explicitly when the app shuts down
  - Expensive to create so only created once

EntityManagerFactory Responsibilities

- Reads the persistence.xml File
  - This contains the mapping from classes to database tables
    - *Database connection info (JDBC URL, driver, username, etc.)*
    - *Entity class declarations*
    - *JPA provider settings (e.g., Hibernate-specific properties)*
- Loads the JPA ORM Provider
  - JPA delegates to the provider (e.g., Hibernate, EclipseLink).
  - The provider implements the low-level persistence logic.
- Parses and validates @Entity classes
  - For all entity classes found in the code are:
    - *Scanned for annotations like @Entity, @Id, @OneToMany, etc.*
    - *Validated (e.g., checking if primary keys are defined)*
    - *Mapped to corresponding database tables*
- Establishes and maintains database connections

@Entity

- @Entity is a marker annotation
  - Class annotated with @Entity are treated as persistent entities
  - Means this is a Java class whose instances are stored as rows in a database table.
- JPA registers the class as an entity during application startup.
- Expects a corresponding table in the database
  - May be configured to create one if it doesn’t exist
  - Manages the class's instances using an object-relational mapping (ORM).
- Other annotations
  - @Id - Marks id as the primary key
  - @GeneratedValue - Tells JPA to auto-generate the ID value



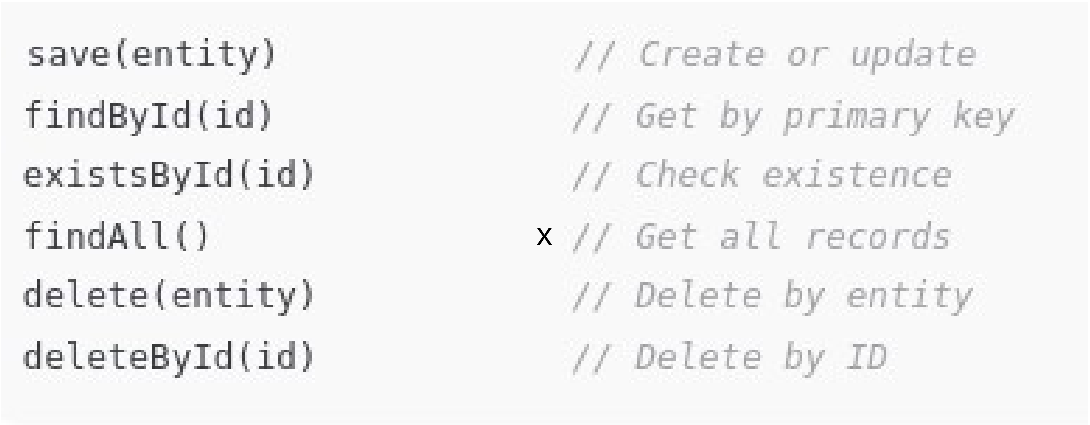
```java
@Entity
public class Employee {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    private String department;
}
```

@Entity

- In order to facilitate the ORM mapping, entity classes have to meet certain requirements
  - No-arg constructor - Must have a public or protected no-argument constructor
  - Unique identifier - Must have a field or property annotated with @Id
  - Not final - The class must not be final
  - Not abstract - Must be concrete if used directly
  - Serializable (optional) - Often recommended, especially in distributed apps

Spring Data JPA

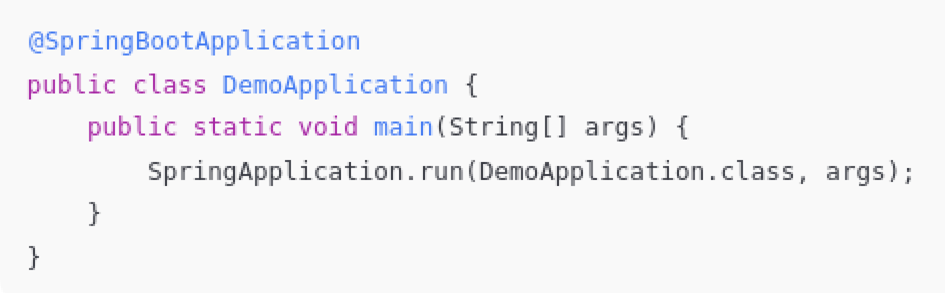
- Spring Data JPA is a module of the Spring Data project
  - Integrates JPA into the Spring Framework
  - Eliminates most boilerplate JPA code (e.g., EntityManager usage)
  - Provides repository interfaces for CRUD and custom queries
  - Supports integration with Spring Boot for easy setup
- Provides a basic CRUDRepository Interface
  - Can be extended to add specialized methods



```
save(entity)               // Create or update
findById(id)               // Get by primary key
existsById(id)             // Check existence
findAll()                x // Get all records
delete(entity)             // Delete by entity
deleteById(id)             // Delete by ID
```

Spring Boot Application Class

- Spring Boot automatically:
  - Detects @Entity classes
  - Scans and wires repository interfaces
  - Configures the JPA provider (e.g., Hibernate)
  - Loads the database connection from application.properties
- We will be using Spring Data in the lab
  - This simplifies the code you will have to work with



```java
@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

Lab 4-1

Spring Data JPA

Java™