Object Relational Mapping (ORM) using Java Persistence API (JPA)

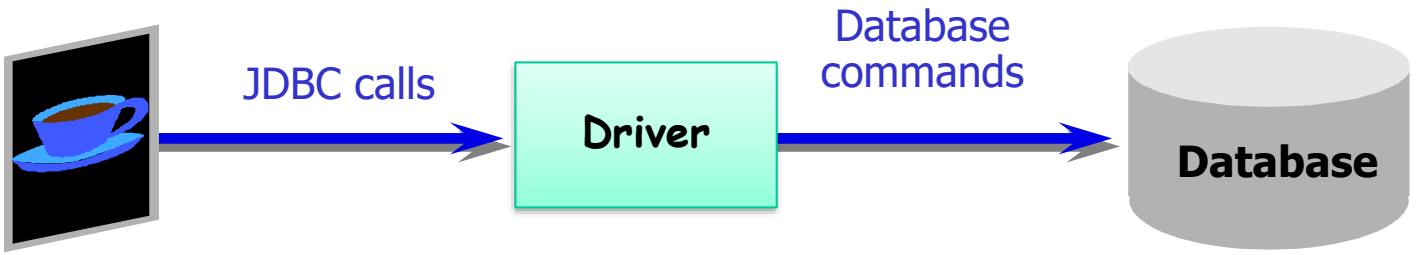
Dr. Abdelkarim Erradi

CSE @ QU

Outline

1. Review of JDBC
2. Object Relational Mapping (ORM)
3. Basic JPA Annotations
4. Annotations for Entity Relationships
5. JPA Programming
6. Java Persistence Query Language (JPQL)
7. Entity Inheritance

JDBC Review

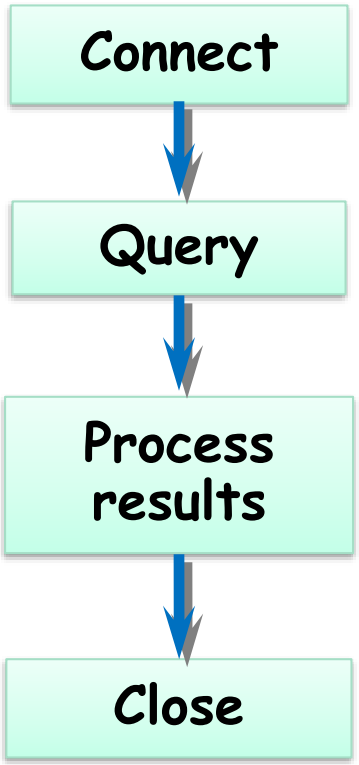


JDBC Overview

- JDBC (Java Database Connectivity) is an API for accessing databases from Java applications
  - API = a collection of classes and interfaces
  - JDBC allows establishing database **connection**, executing SQL **statements**, manipulating **query results**
- Need a JDBC driver for your database engine
- API represents key **runtime** entities: **Connection**, **Statement**, **ResultSet**
  - `Connection` is used to connect to the database
  - `Statement` used to submit a query to the database
  - `ResultSet` provides access to a table of data returned by executing a Statement

5 Steps for Using JDBC

1. Connect to the database
2. Create a Statement object
3. Execute a query using the Statement
4. Process the results
5. Close the connection



```
@Resource(mappedName="jdbc/demo")
private DataSource dataSource;

public List<Contact> getContactsUsingJDBC() {
    List<Contact> contacts = new ArrayList<>();
    try (Connection dbConnection = dataSource.getConnection();
        Statement statement = dbConnection.createStatement()) {

        ResultSet rs = statement.executeQuery("select * from contact");
        while(rs.next()) {
            int id = rs.getInt("id");
            String title = rs.getString("title");
            String name = rs.getString("name");
            String dob  = rs.getString("dob");
            String gender = rs.getString("gender");
            String relationship = rs.getString("relationship");
            String email = rs.getString("email");
            String phone = rs.getString("phone");
            Contact contact = new Contact(id, title, name, dob, gender,
                    relationship, email, phone);
            contacts.add(contact);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return contacts;
}
```

SQL Statements

- Structured Query Language (SQL)
  - Language used to define, alter and access the elements described above
- **C**reating data:

```
INSERT into PERSON (first_name, last_name)
  VALUES ('Ahmed', 'Sayed')
```

- **R**eading data:

```
SELECT first_name FROM person WHERE last_name = 'Sayed'
```

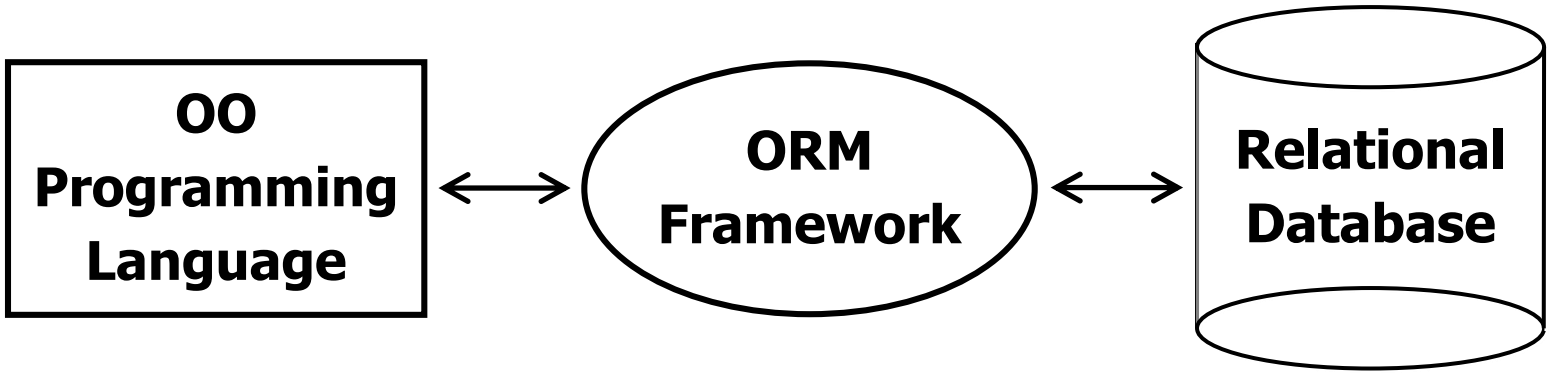
- **U**pdating data:

```
UPDATE person SET first_name = 'Ali' where
  last_name = 'Sayed'
```

- **D**eleting data:

```
DELETE from person where last_name = 'Sayed'
```

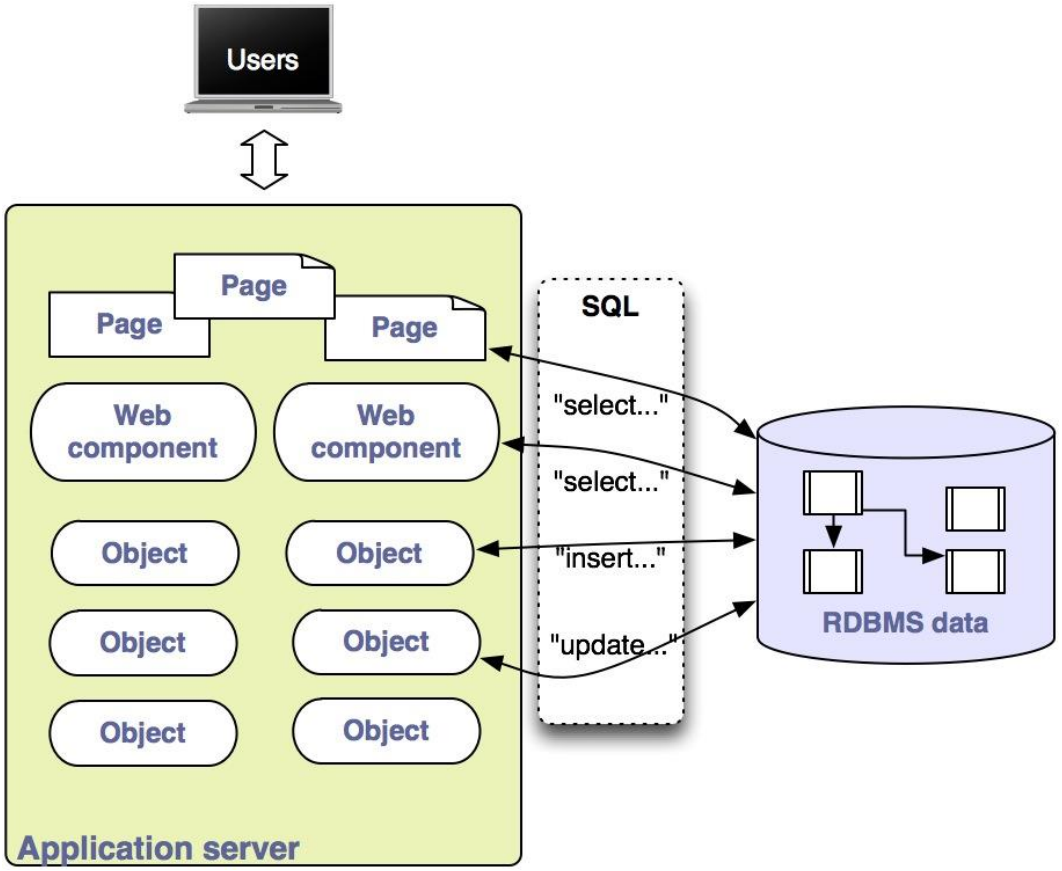
Object-Relational Mapping (ORM)



The Problem – Impedance Mismatch

❒ OOP concepts:

- Classes having Attributes and Methods
- Classes have relationships:
  * **Composition** (e.g., Student has many sections)
  * **Inherence** (e.g., HoD extends Instructor)



❒ RDBMS concepts:

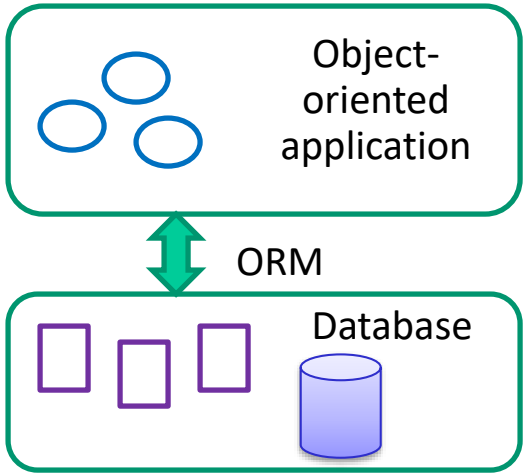
- Tables, columns, primary key, foreign keys

Object-Relational Impedance Mismatch

ORM = Solution for Impedance Mismatch

- OO Programming style is widely used
- But… need easy way to persist object data in the database

Impedance Mismatch (conceptual mismatch between OOP and database worlds)



=> Solution is Object-Relational Mapping (ORM) –

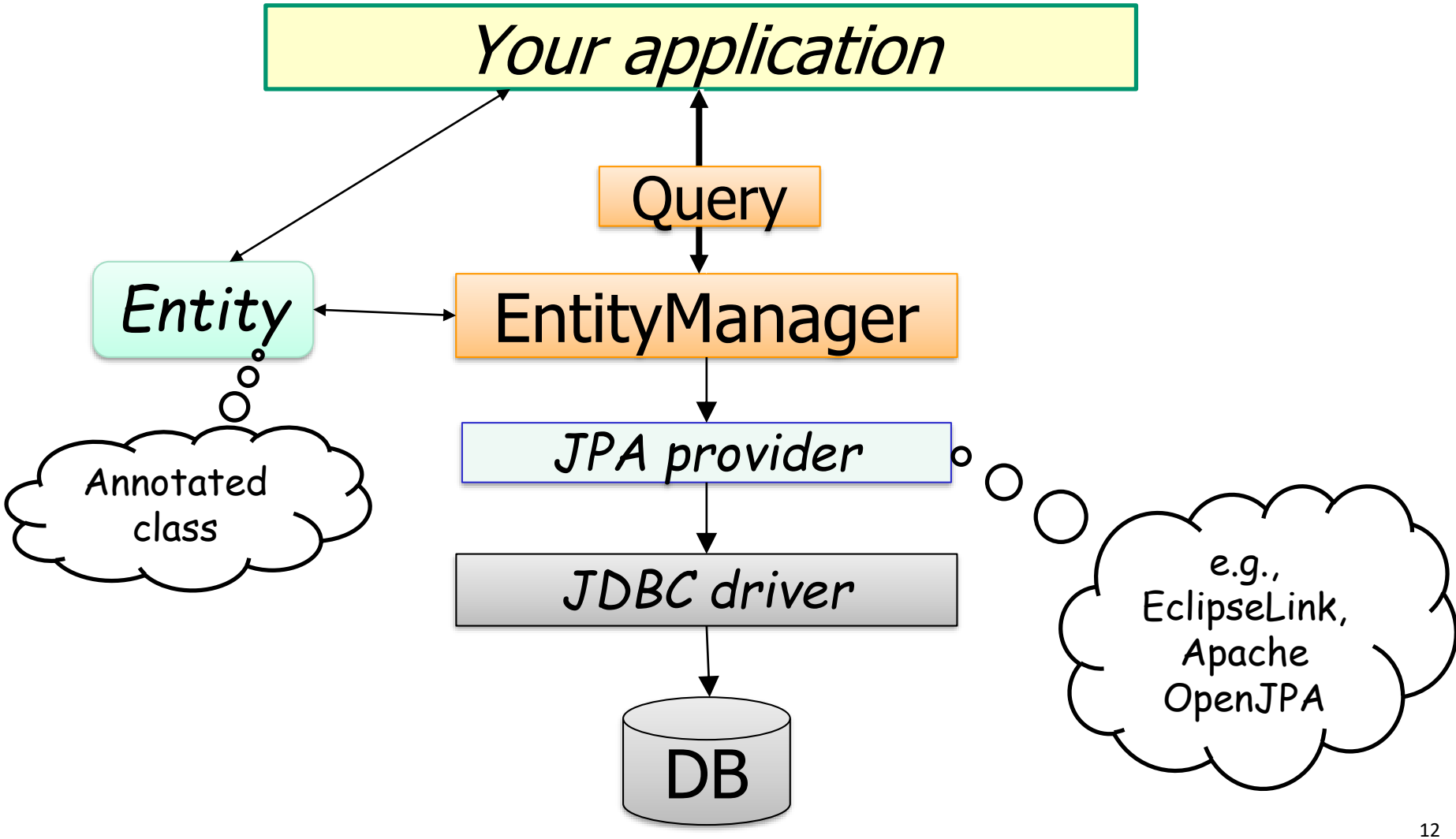
a software layer that **shuttles data back and forth between table rows and objects**

ORM Design Goals

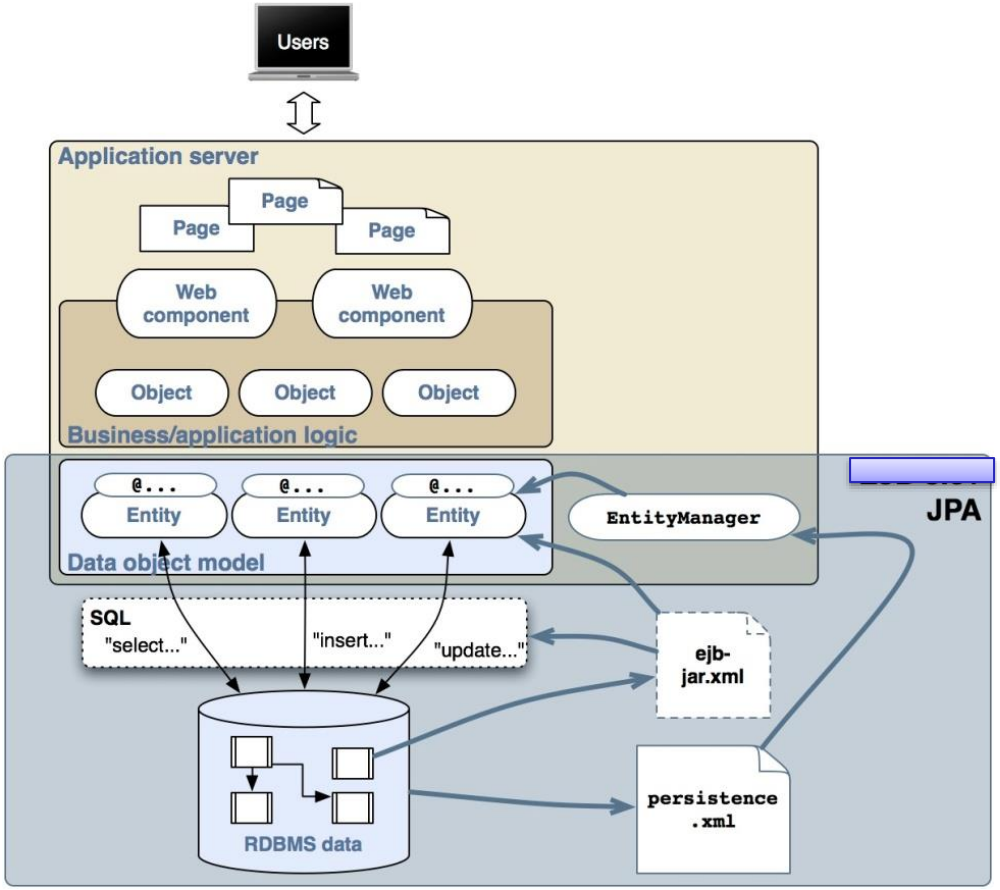
- Make it easier for OO developers to **save** and **retrieve** *objects* to/from DB
  - Query and retrieve data from tables and create the corresponding objects
  - **Synchronize** objects with the database by auto-generating the required insert/ update/delete SQL statements
  - Save and recreate ***associations*** between objects

Java Persistence API (JPA) Architecture

JPA is a Java standard for ORM



JPA Elements



- **O-R mappings** using annotations defines how Java classes can be mapped to database tables
  - Entity X is mapped to table A, property X.Y is mapped to column A.B, etc.
- **Programming API** for storage/retrieval of entities (i.e., read/write to DB using `EntityManager`
  JPQL - Java Persistence Query Language
- **`Persistence.xml`**
  - Defines configuration details to connect to the database

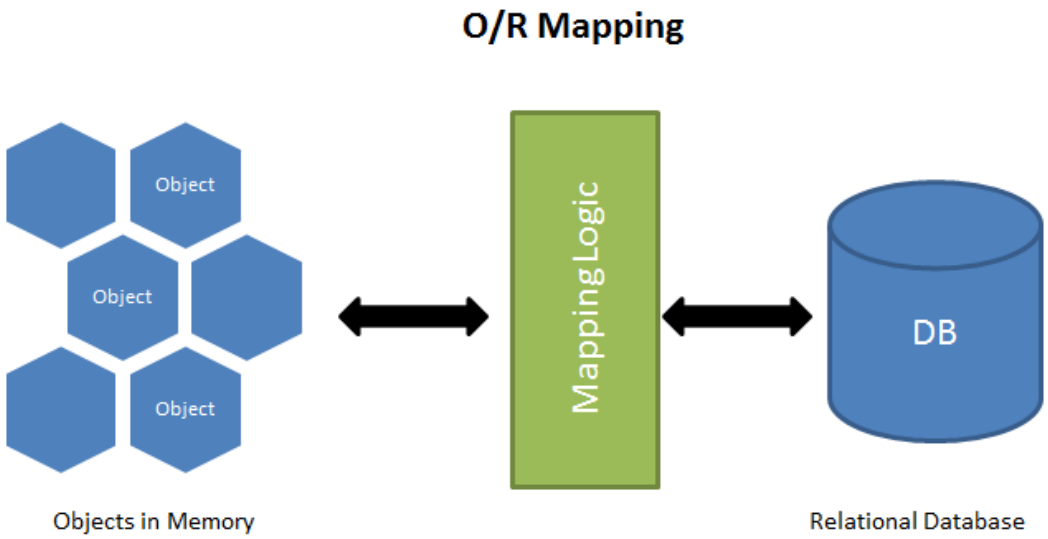
persistence.xml

- A **`persistence.xml`** file defines one or more persistence units

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="mvcappPU" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>java:app/jdbc/ContactDB</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties>
            <!-- Possible values: "none", "create", "drop-and-create", "drop" -->
            <property name="javax.persistence.schema-generation.database.action" value="drop-and-create"/>
            <property name="javax.persistence.schema-generation.scripts.action" value="drop-and-create"/>
            <!-- You can find the generated script under C:\glassfish4\glassfish\domains\domain1\config-->
            <property name="javax.persistence.schema-generation.scripts.create-target" value="ContactDB_CreateScript.sql"/>
            <property name="javax.persistence.schema-generation.scripts.drop-target" value="ContactDB_DropScript.sql"/>
    </properties>
  </persistence-unit>
</persistence>
```

- The **`Database Connection Pool`** and the **`Database Resource`** could be created using **`glassfish-resources.xml`** (see posted example)

Basic JPA Annotations

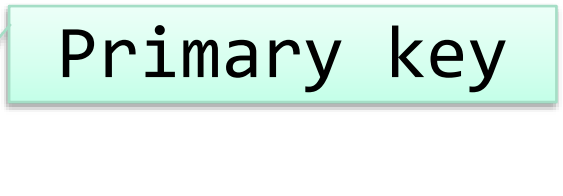


Minimal Entity Annotation

- Entity represents a table in a relational database, and each entity instance corresponds to a row in that table

- A class must annotated with @Entity

```
@Entity
public class Employee {
      @Id int id;
      public int getId() { return id; }
      ….
}
```



Each entity object has a unique id that Uniquely identifies the entity in memory and in the DB

Identifier Generation

- Identifiers can be generated in the database by specifying **`@GeneratedValue`** on the identifier
- The most common generation strategies is `IDENTITY`

  - The value gets auto incremented by 1 by the DB

```
@Id @GeneratedValue (strategy=GenerationType.IDENTITY)
int id;
```

Customizing Entity Annotation

- In most cases, the defaults are sufficient

-> Configuration by Exception

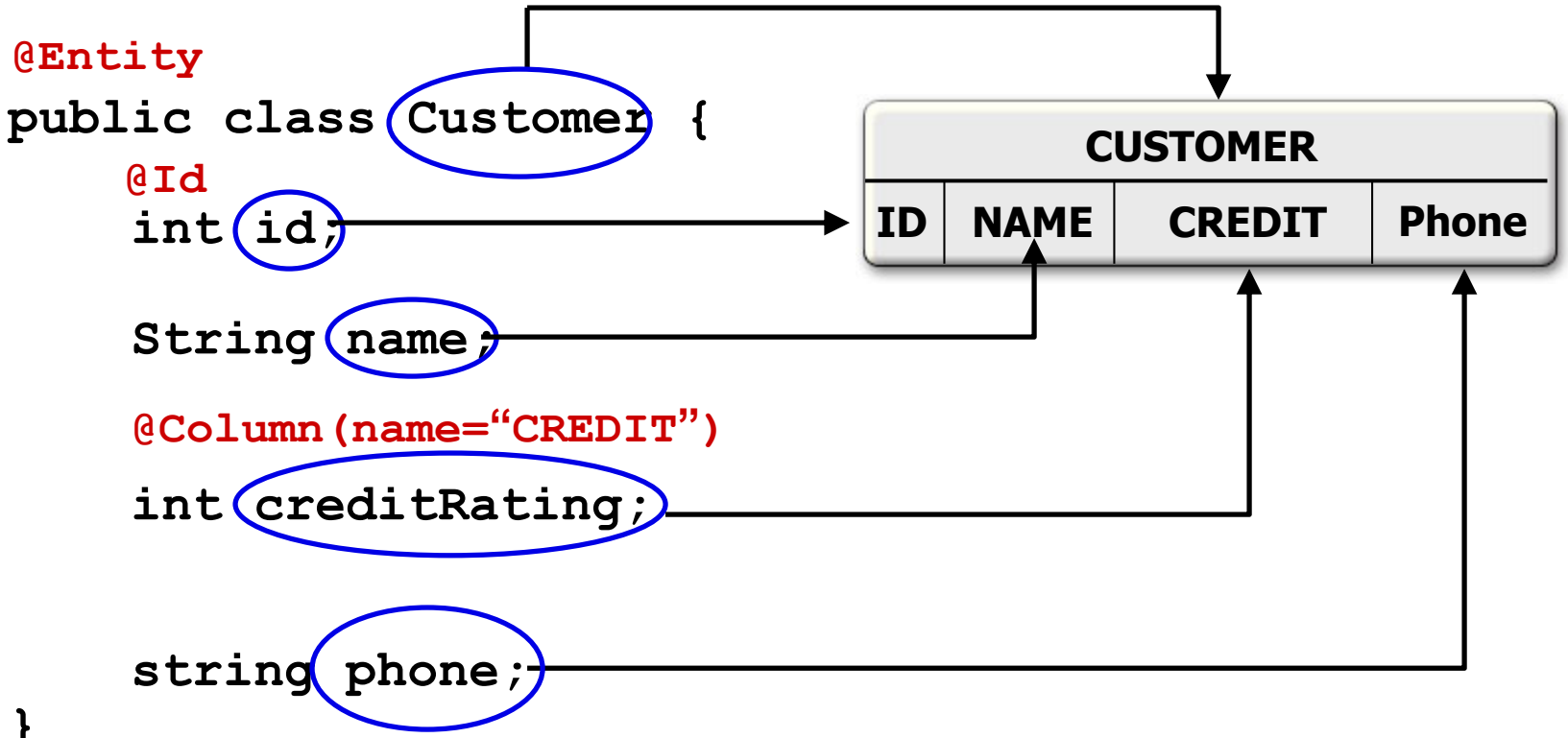
- By default the table name corresponds to the unqualified name of the class
  - Customization:

```
@Entity (name="FULLTIME_EMPLOYEE")
public class Employee{ …… }
```

- The defaults of columns can be customized using the `@Column` annotation

```
@Id @Column(name = “EMPLOYEE_ID”, nullable = false)
private String id;

@Column(name = “FULL_NAME” nullable = true, length = 100)
private String name;
```

Simple Mappings using Annotations



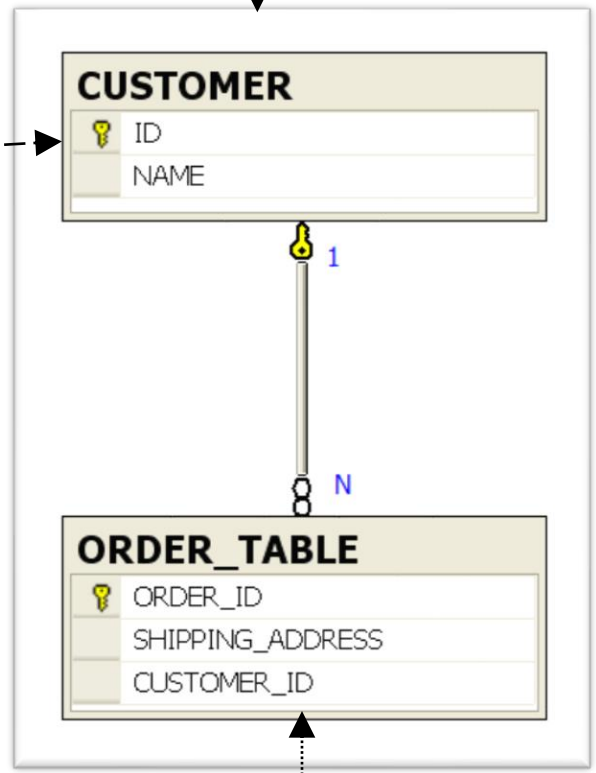
Annotations for Entity Relationships

Back

Bidirectional OneToMany Mapping

```
@Entity
public class Customer {

   @Id
    int id;
   ...
   @OneToMany(mappedBy="customer")

  List<Order> orders;
}

@Entity(name = "ORDER_Table")
public class Order {
   @Id @Column(name = "ORDER_ID")
   int id;
   ...
  @ManyToOne
  @JoinColumn(name="CUSTOMER_ID")

  Customer customer;
}
```

Field that has the other end of the relationship



Foreign Key

Rules for bidirectional relationships

- The **Many side** (i.e., the entity having the foreign key) of a bidirectional relationship defines the mapping to the database using **`@JoinColumn`** to specify foreign key column
- The **One** side of a bidirectional relationship must refer to the Many side attribute having **`@ManyToOne`** annotation using of the **`mappedBy`** attribute
  - The **mappedBy** attribute designates the attribute in the Many side

Unidirectional One-to-Many

- **`@OneToMany`** annotation can be unidirectional (does not contain a mappedBy element).
  - **`@JoinColumn`** refers to a foreign key column in the target table

```
@Entity
public class Customer {
   …
@OneToMany
@JoinColumn(name="Customer_ID")
List<Order> orders;
   …
}
```

Foreign key column

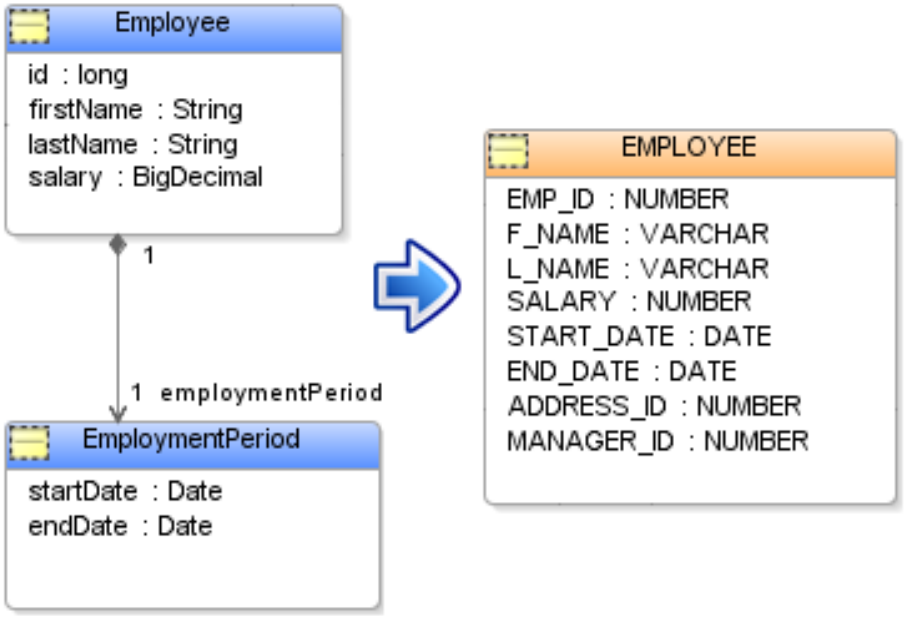
Relation Attributes

- Relationships data may be loaded or “fetched” as EAGER or LAZY
  - **LAZY** - hint to the Container to defer loading until the field or property is accessed
    - LAZY is the default for OneToMany and ManyToMany
  - **EAGER** - requires that the field or relationship be loaded when the referencing entity is loaded
    - EAGER is the default for ManyToOne and OneToOne
- Cascading of entity operations to related entities

  `ALL, PERSIST, MERGE, REMOVE, REFRESH`

```
@OneToMany(
        cascade = {CascadeType.PERSIST, CascadeType.MERGE},
        fetch = FetchType.EAGER)
```

Mapping 1-to-1 Whole-Part to one Table



1-to-1 Whole-Part

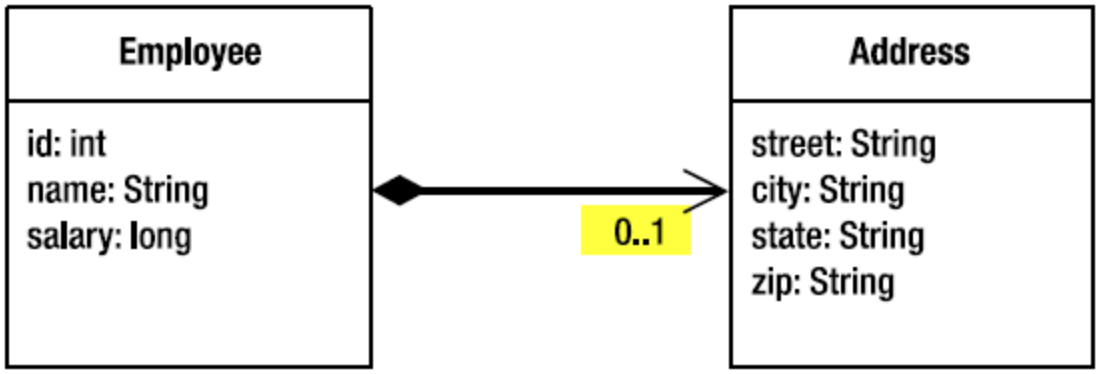
Once Table

- Attributes of embeddable object is mapped to the same table that represents the owning entity
- An embeddable object is a dependent part and cannot be directly persisted or queried

```
@Embeddable
public class EmploymentPeriod {
   @Column(name="START_DATE")
   private java.sql.Date startDate;

   @Column(name="END_DATE")
   private java.sql.Date endDate;
   ...
}
```

```
@Entity
public class Employee {
   @Id
   private long id;
   ...
   @Embedded
   private EmploymentPeriod period;
   ...
}
```

@Embeddable Another Example



Employee *and* Address *relationship*

```
@Embeddable
public class Address {
    private String street;
    private String city;
    private String state;
    @Column(name="ZIP_CODE")
    private String zip;
    // ...
}
```

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    @Embedded private Address address;
    // ...
}
```

@IdClass can be used as a Compound Primary Key

- An entity might have a compound primary key that is made of multiple attributes
- The entity with a compound key needs to be annotated with with *@IdClass*
  - **IdClass** is a class without any annotations to wrap up the compound key attributes

Compound Primary Keys using IdClass

```
public class ReviewPK implements Serializable {
      private int proposalID;
      private int reviewerID;
      // getters & setters
    …
}
```

IdClass is a class without any annotations to wrap the compound primary key

```
@Entity
@IdClass(qu.jpa.ReviewPK.class)
public class Review {
    @Id private int proposalID;
    @Id private int reviewerID;
... }
```

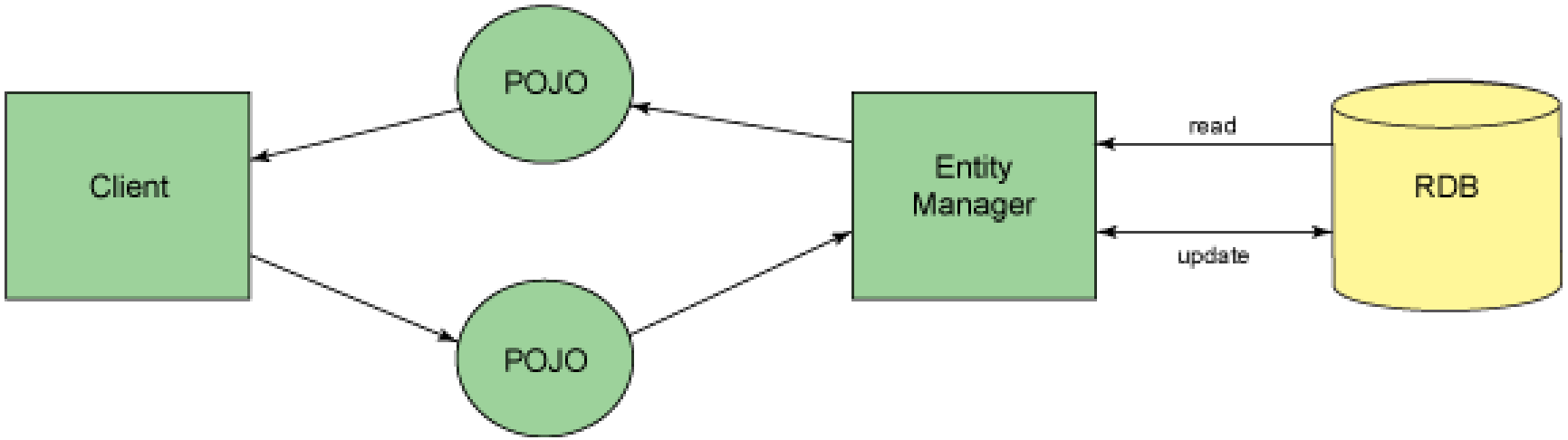
@JoinColumn(name="??", insertable=false, updatable=false)

- **insertable=false, updatable=false** makes the relationship read only. i.e., when inserting or updating Absence the attributes from Student and Section are not used

```
@Entity  public class Absence {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int absenceId;
    private int crn;
    private int studentId;
    @Temporal(TemporalType.DATE)
    private Date absenceDate;
    @Transient //For non-mapped attributes
    String formattedDate;
    @ManyToOne
    @JoinColumn(name="studentId", insertable=false, updatable=false)
    private Student student;
    @ManyToOne
    @JoinColumn(name="crn", insertable=false, updatable=false)
    private Section section;
```

@Temporal(TemporalType.*DATE*) *is used for date attributes*

JPA Programming

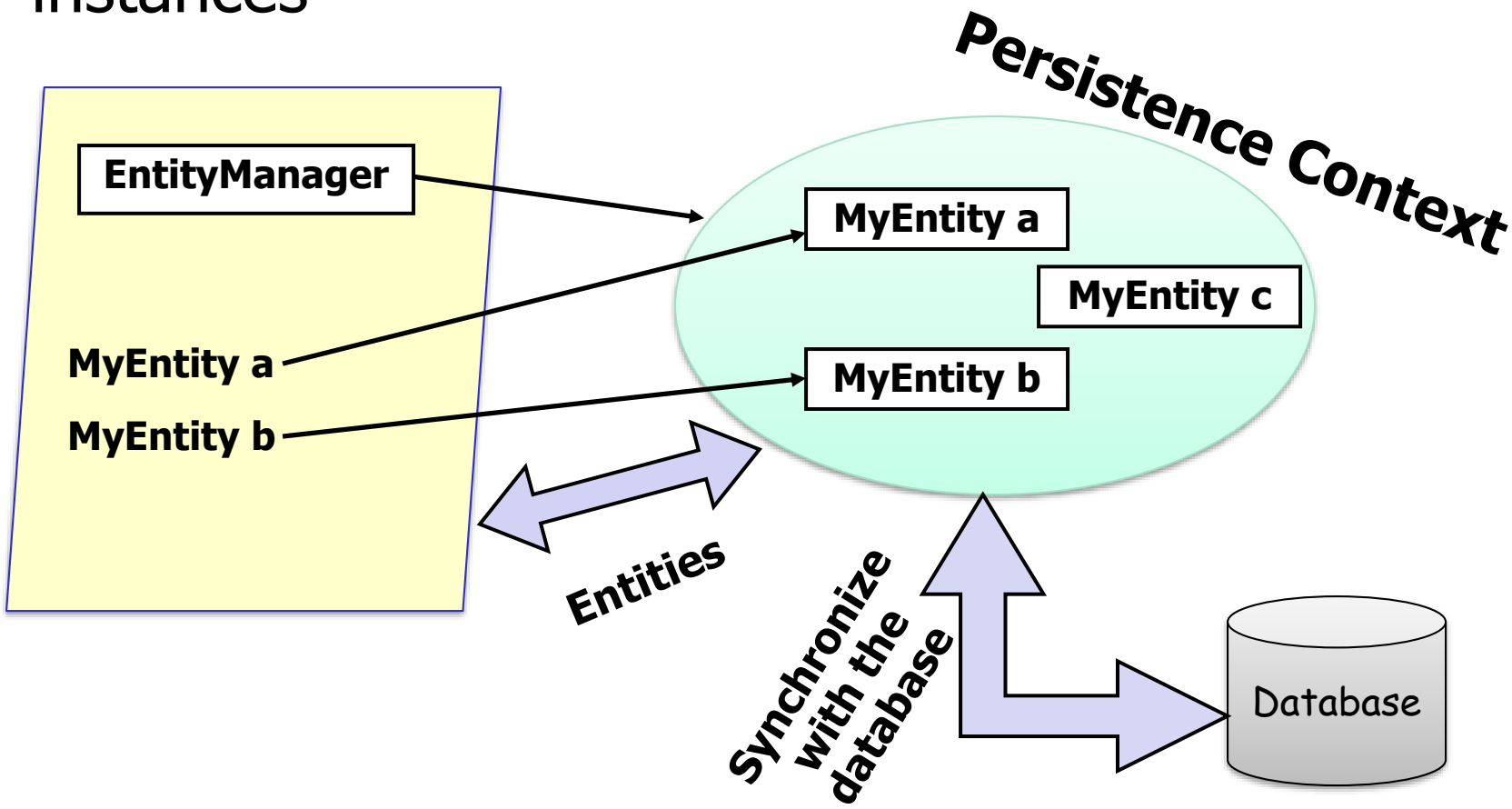


Entity Manager

- Entities are managed by the **`Entity Manager`**
- The Entity Manager is the most important class in JPA.
- It contains the lifecycle APIs for entities
  - `find()`, `persist()`, `merge()`, `remove()`
- Source for Queries
  - `createNamedQuery`, `createQuery()`, `createNativeQuery()`

Persistence Context

- Each EntityManager instance is associated with a **persistence context (PC)**
  - **PC = in memory store** for "managed" entity instances



Entity Manager Operations

- EntityManager API
  - `persist()` - **Insert** the an entity instance into the PC
  - `remove()` - **Delete** the entity instance from the PC
  - `refresh()` - **Reload** the entity instance from the DB
  - `merge()` - **Update** an entity instance in the PC
  - `find()` - **Find** an entity instance by primary key
  - `contains()` - Determine if entity is managed by PC
  - `flush()` - **Synchronize** the PC with the database
  - `createQuery()` - Create query instance using **dynamic JPQL**
  - `createNamedQuery()` - Create instance for a predefined JPQL query
  - `createNativeQuery()` - Create instance for an **SQL query**

find()

- Gets an entity instance by Primary Key. Returns null if not found

```
@PersistenceContext
private EntityManager em;
…
public Customer getCustomer(long customerId) {
    return em.find(Customer.class, customerId);
}
```

persist()

- Insert a new entity instance

```
@PersistenceContext
private EntityManager em;
…
public Customer addCustomer(int fName, String lName)
{
    Customer customer = new Customer(fName, lName);
    em.persist(customer);
    return customer;
}
```

- It is common to return the new entity as it contains the auto-generated **id**

remove()

- Delete an entity instance

```
public void removeCustomer(int customerId) {
   Customer customer = entityManager.getReference(Customer.class,
customerId);
   entityManager.remove(customer);
}                          OR
public void removeCustomer(int customerId) {
   Query query = entityManager.createQuery("delete from Customer
        where customerId = :customerId");
    query.setParameter("customerId", customerId);
    query.executeUpdate();
}
```

merge()

- Updates an entity instance

```
@PersistenceContext
private EntityManager em;
…
public void update(Customer customer) {
      em.merge(customer);
}
```

Important Note

- Just because you called persist(), doesn't mean it's in the database
- JPA synchronizes to the database:
  - when a repository method completes execution
  - when em**.flush()** is called explicitly

Java Persistence Query Language (JPQL)

Back

JPA Queries

- JPA supports **named** queries, **dynamic** queries and **native** queries
- Query instances are obtained the EntityManager using:

  `createNamedQuery`(), **`createQuery`**() & **createNativeQuery**
- Query class API:

  `getResultList()` – execute query returning multiple results

  `getSingleResult()` – execute query returning single result

  `executeUpdate()` – execute bulk update or delete

  `setMaxResults()` – set the maximum number of results to retrieve

  `setParameter()` – bind a value to a named or positional parameter

Named Queries

```
@NamedQueries({
    @NamedQuery(name="Sale.findByCustId",
    query="select s from Sale s
           where s.customer.id = :custId
           order by s.salesDate")})

public List<Sale> getSalesByCustomer(int custId) {
 return entityManager.createNamedQuery("Sale.findByCustId")
         .setParameter("custId", custId)
         .getResultList();
}
```

- **Statically** defined queries
- Use **createNamedQuery**() method and pass in the query name already defined in the annotation
- Query names are “globally” scoped
- Get compiled and errors reported at compile time

Dynamic Queries

```
public Customer getCustomer(String customerName) {
  Query query = em.createQuery("select c from
      Customer c where c.name = :customerName ");
  query.setParameter("customerName", customerName);
  return (Customer) query.getSingleResult();
}
```

Native Queries

- Use **`createNativeQuery()`** method and pass in the SQL query string at runtime
- Use when you need to use native SQL of the target database

```
Query query = em.createNativeQuery("select *
from users where username = :username",
qu.jpa.User.class);

query.setParameter("username", "shrek");

User user = query.getSingleResult();
```

JPQL

- JPQL has all the power of SQL:
  - Bulk update and delete operations
  - Joins
  - Group By / Having
  - Subqueries
  - Etc.

Basic Query Syntax

- **A Basic Select Query -** return All players

```
SELECT p FROM Player p
```

- **Eliminating Duplicate Values**

  SELECT DISTINCT p FROM Player p

-
```
SELECT avg(e.salary) FROM
Employee e
WHERE e.salary > 80000
```

  - Return Avg Salary of Employees with salary > 80000

Example JPQL Queries

- **A Basic Select Query -** return All players

```
SELECT p FROM Player p
```

- Queries can have named parameters that are prefixed with a **colon** (**:**)
- To pass the parameter to the query use method:

```
query.setParameter(String name, Object value)
```

```
public List<Customer> findByName (String name) {
  return em.CreateQuery (
   "SELECT c FROM Customer c " +
   "WHERE c.name LIKE :custName")
   .setParameter("custName", name)
    .getResultList();
}
```

Bulk Update & Delete Example

```
public void assignManager(Department dept, Employee manager) {
     em.createQuery("UPDATE Employee e " +
                    "SET e.manager = ?1 " +
                    "WHERE e.department = ?2")
       .setParameter(1, manager)
       .setParameter(2, dept)
       .executeUpdate();
}
```

```
public void removeEmptyProjects() {
     em.createQuery("DELETE FROM Project p " +
                    "WHERE p.employees IS EMPTY")
       .executeUpdate();
}
```

Why Use JPQL?

- Once the mapping is defined, the programmer, can design the queries by just looking at the java classes
- JPQL isolates you from the mapping logic
  - For example, your object may be mapped into separate tables but you do not bother with the join when writing your query
- JPQL Named Queries are compiled and any errors (such as a missing column) are reported by the compiler
- JPQL makes it less likely that you will be using vendor-specific, non-portable SQL

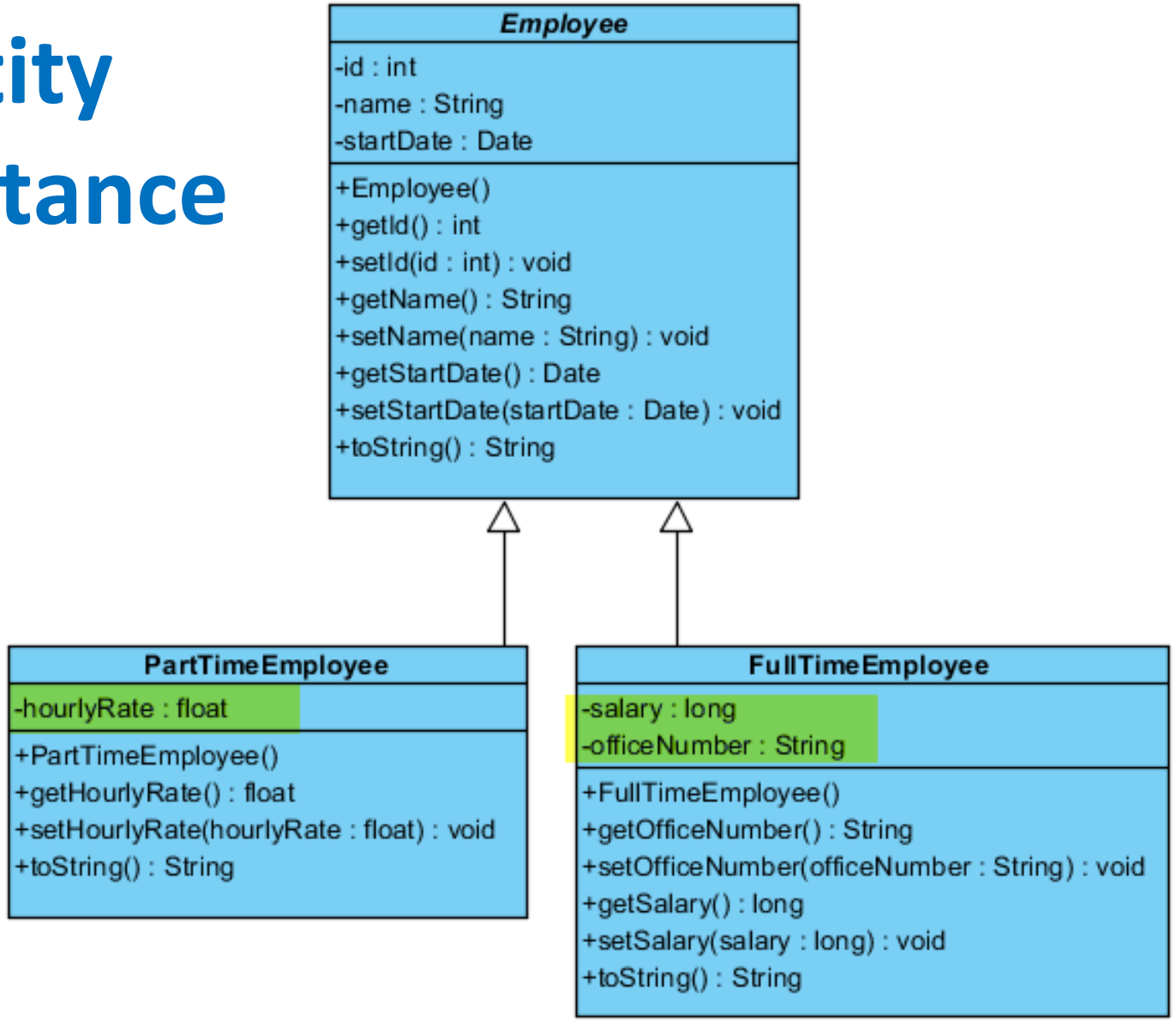
Why Use JPQL?

- Once the mapping is defined, the programmer, can design the queries by just looking at the java classes
- JPQL isolates you from the mapping logic
  - For example, your object may be mapped into separate tables but you do not bother with the join when writing your query
- JPQL Named Queries are compiled and any errors (such as a missing column) are reported by the compiler
- JPQL makes it less likely that you will be using vendor-specific, non-portable SQL

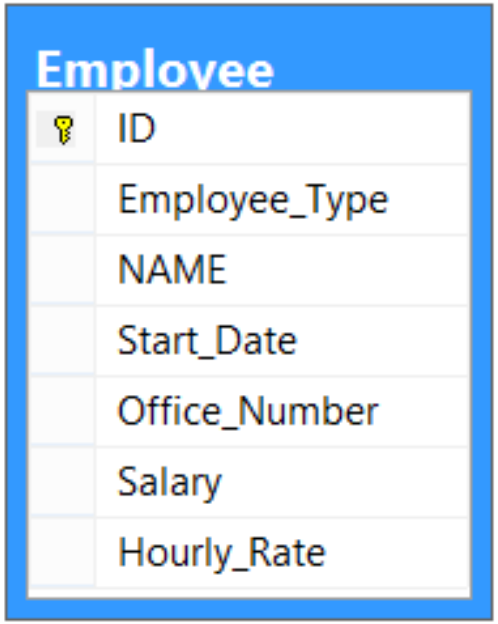
Entity Inheritance

Back

Entity Inheritance

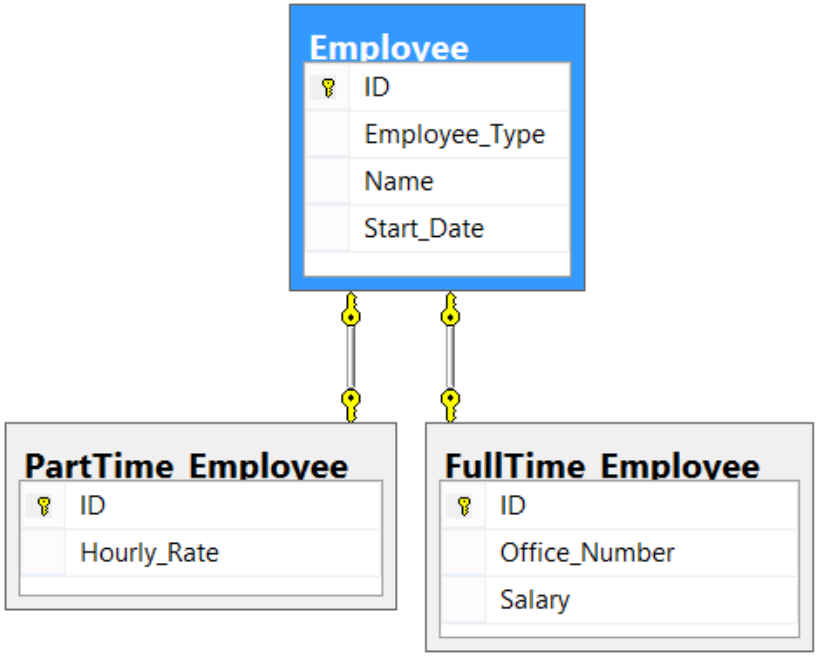


- 3 Strategies to Map this to a Database:
  - `SINGLE_TABLE, JOINED, TABLE_PER_CLASS`



Single Table strategy

(this is the default)



Joined Table strategy

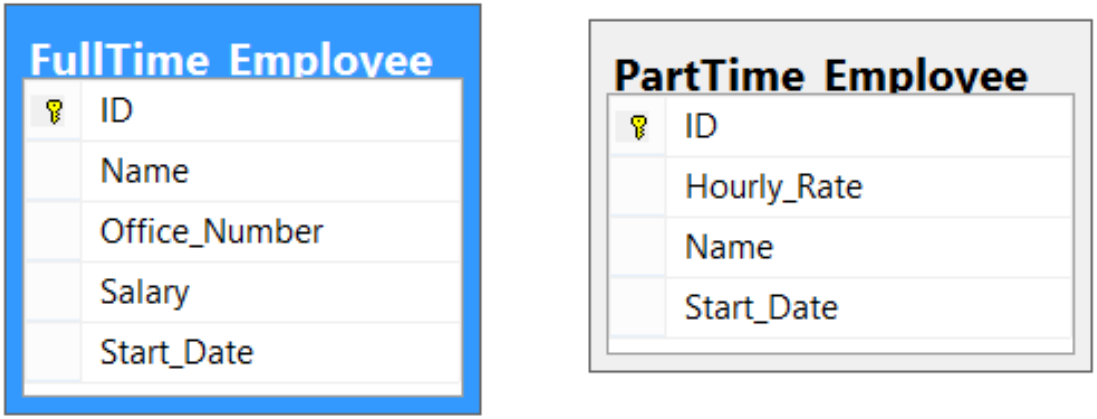
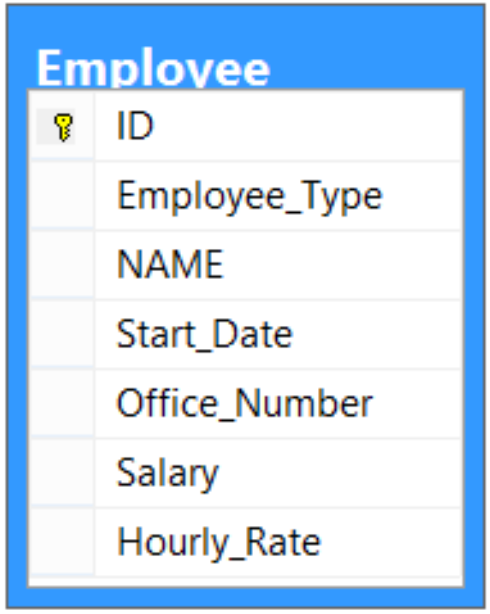


Table per Concrete Class strategy

Single Table Strategy




```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="Employee_Type",
                     discriminatorType=DiscriminatorType.CHAR)
public abstract class Employee {
```

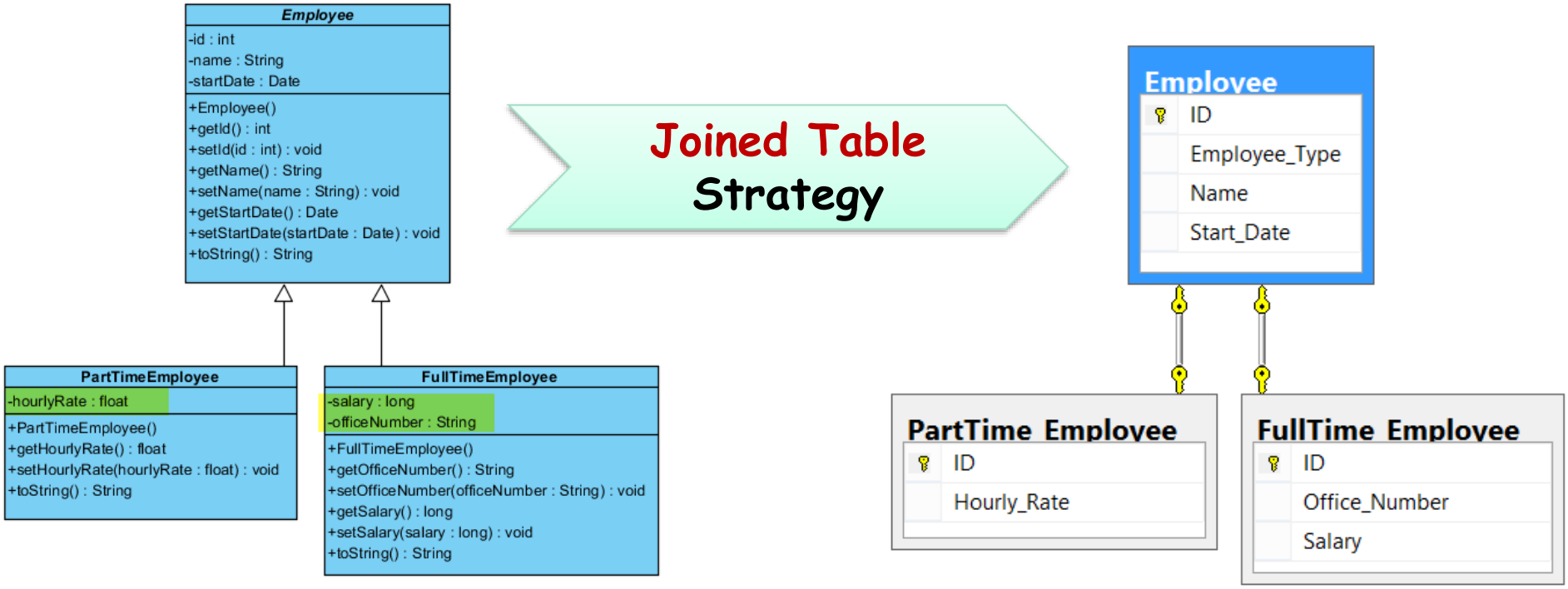
```
@Entity
@DiscriminatorValue("F")
public class FullTimeEmployee extends Employee {
```

```
@Entity
@DiscriminatorValue("P")
public class PartTimeEmployee extends Employee {
```

- **Single Table - Advantages**

  - Simple and fastest since:
    - loading entities requires querying only one table (no join required)
    - Persisting or updating a persistent instance requires only a single INSERT or UPDATE statement
- **Single Table - Disadvantages**
  - The larger the inheritance model gets, the "**wider**" the mapped table gets – DB not normalized
    - For every field in the entire inheritance hierarchy, a column must exist in the mapped table.
    - A wide or **deep inheritance hierarchy will result in tables with many mostly-empty columns**
  - A change to any class in the hierarchy requires the single table to be altered

=> only suitable for small inheritance hierarchies



One table for each class in the hierarchy

- A parent class is represented by a single common table
- Each child class is represented by a separate table that contains fields specific to the child class as well as the columns that represent its primary key
- Foreign key relationship exists between parent common table and subclass tables

Joined Table Strategy

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="Employee_Type",
        discriminatorType=DiscriminatorType.CHAR)
public abstract class Employee {


      @Entity
      @Table(name="FullTime_Employee")
      @DiscriminatorValue("F")
      public class FullTimeEmployee extends Employee {


@Entity
@Table(name="PartTime_Employee")
@DiscriminatorValue("P")
public class PartTimeEmployee extends Employee {
```

Joined Strategy - Advantages

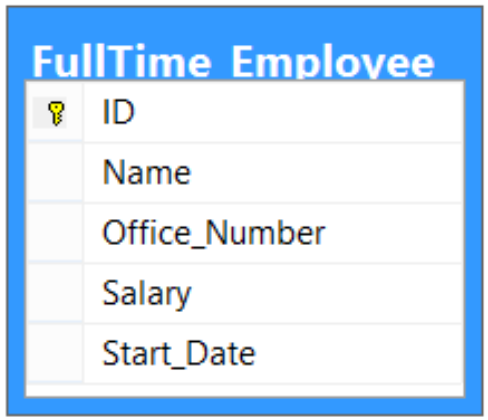
- Using joined subclass tables results in the most *normalized* database schema
  - without any duplicate columns or unwanted nullable columns
  - database schema similar to classes model
- Easier to maintain:
  - Adding a subclass only required adding the corresponding database table (rather than having to change the structure of existing tables).

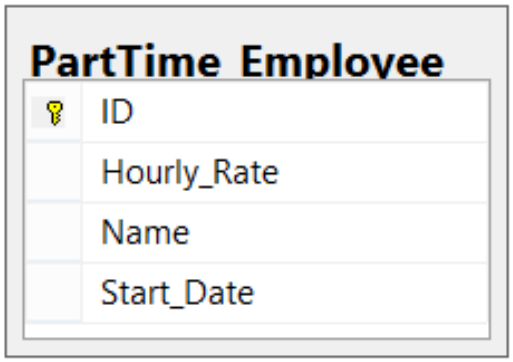
Joined Strategy - Disadvantages

- Retrieving any subclass requires one or more database joins, and storing subclasses requires multiple INSERT or UPDATE statements.
  - Poor performance in deep hierarchies

=> is best suited to large inheritance hierarchies (deep or wide)

Table Per Concrete Class Strategy






```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Employee {

@Entity
@Table(name="FullTime_Employee")
public class FullTimeEmployee extends Employee {


@Entity
@Table(name="PartTime_Employee")
public class PartTimeEmployee extends Employee {
```

- Table Per Concrete Class - Advantages

  - No need for joins
  - Does not require columns to be made nullable
    - results in a DB schema that is relatively simple to understand
- Table Per Concrete Class - Disadvantages
  - A UNION of subclass tables is performed when querying on the superclass => may impact performance
  - The duplication of column corresponding to superclass fields causes the DB design to not be normalized.
    - This makes it hard to perform aggregate SQL queries on the duplicated columns.

=> best suited to wide, but not deep, inheritance hierarchies in which the superclass queries are rarely needed

Summary

JPA emerged from best practices of existing ORM products. It offers:

- ✓ **Standardized object-relational mapping** specified using annotations

- ✓ Simple, lightweight and powerful persistent API
- ✓ Feature-rich query language
- ✓ Support for entity **relationships** and **inheritance**
- ✓ Works for both Java SE and Java EE

Resources

- Java Persistence Tutorials

http://www.vogella.com/tutorials/JavaPersistenceAPI/article.html

http://docs.oracle.com/javaee/7/tutorial/doc/persistence-intro.htm

- JPA Annotation Reference

http://en.wikibooks.org/wiki/Java_Persistence

- JPA Examples

http://wiki.eclipse.org/EclipseLink/Examples/JPA