CMPS 356

Web API Using Java EE

Dr. Abdelkarim Erradi
Dept. of Computer Science & Engineering
QU

Outline

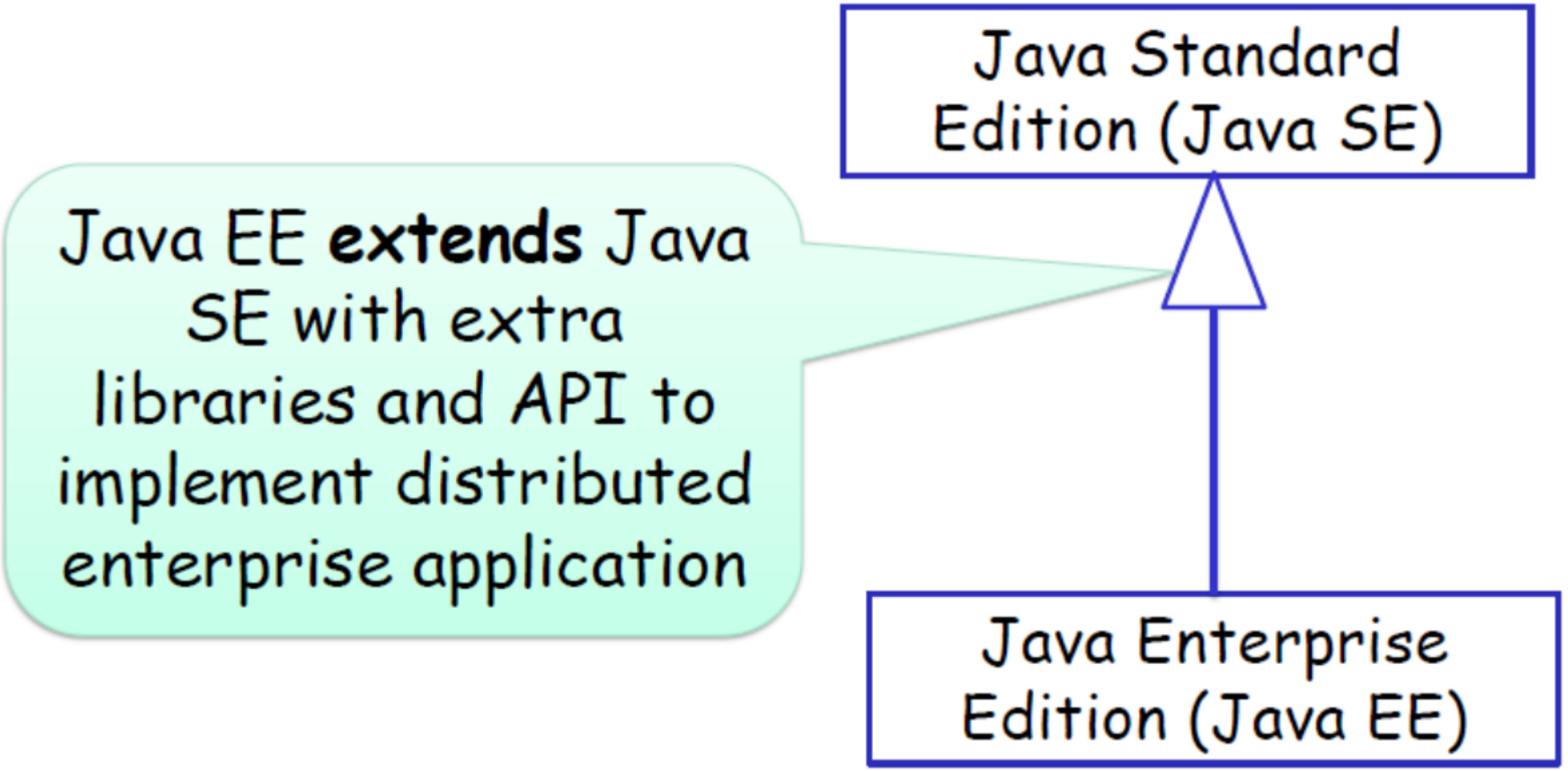
1. Introduction to Java EE
2. REST Services Programming using JAX-RS
3. Review of JDBC
4. Java Persistence API (JPA)

Introduction to Java EE

What is Java EE?

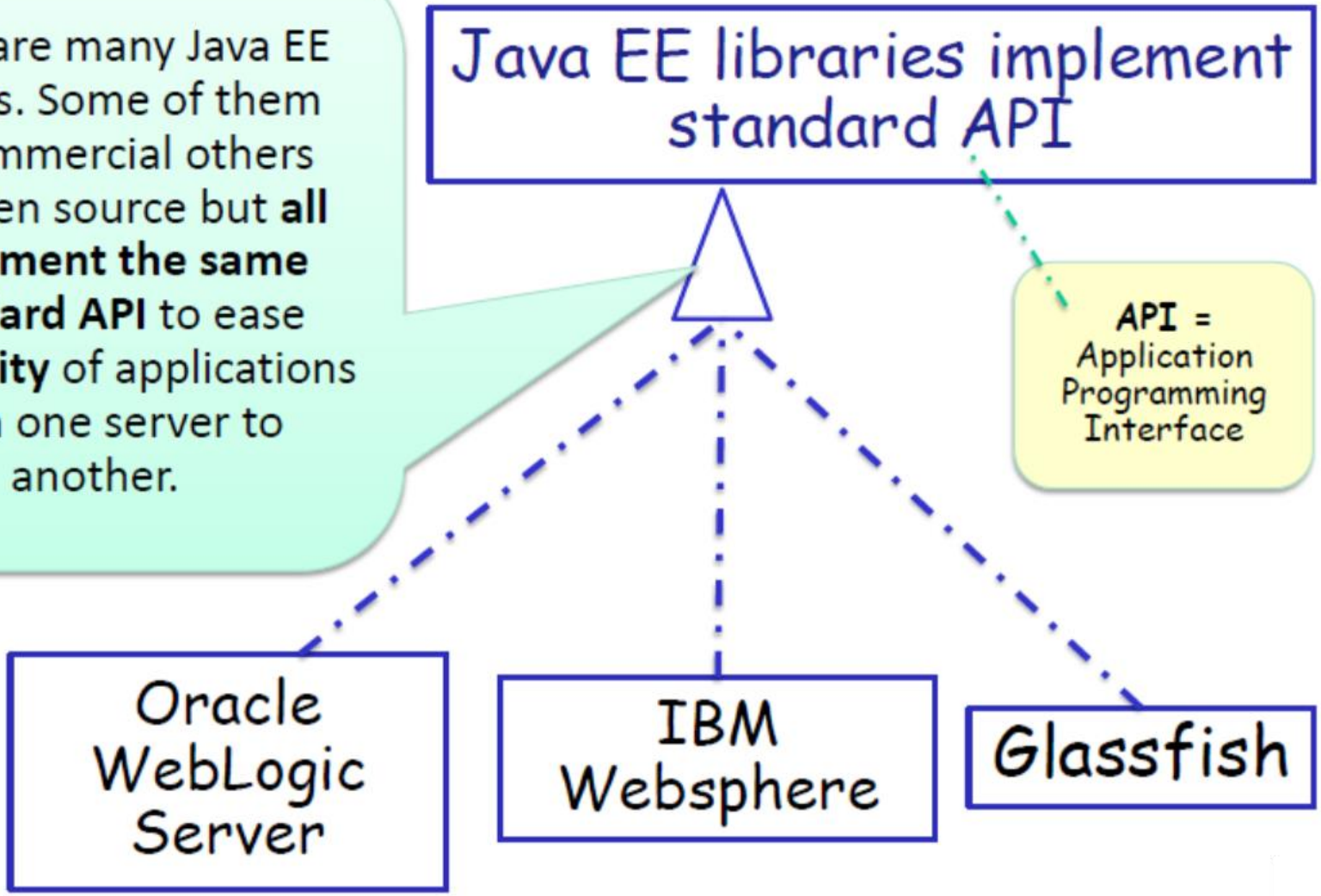
- Java EE is a widely used platform for building enterprise, web-deployed applications
- It is a set of standard Java APIs for enterprise applications
  - API Standardization via Java Community Process (JCP)
- The Java EE platform differs from Java SE in that it adds libraries which provide functionality to build and deploy distributed, multi-tier Java applications, based largely on modular components running on an application server
  - **Component/Container** design pattern

Java Editions



Java EE Servers

There are many Java EE servers. Some of them are commercial others are open source but **all implement the same standard API** to ease **portability** of applications from one server to another.



Java EE Servers

- IBM WebSphere Application Server
- OracleWebLogic Application Server
- Sun GlassFish
- JBoss Application Server
- Apache Geronimo
- SAP NetWeaver Application Server
- So many others both open source and commercial

REST Services Programming using JAX-RS

REST in Java: JAX-RS

- JAX-RS is Java API for creating REST services
- JAX-RS uses annotations to simplify the development of REST services and clients
- Many implementations are available:
  - **Jersey**: reference implementation from Oracle
  - Apache CXF
  - RESTEasy, Jboss's implementation.

@Path

- Specifies the URI Path of the resource corresponding to a class

```
@Path("/contacts")
@Stateless
public class ContactService {
```

- Variables can be embedded in the URI, and then retrieved with the **@PathParam** annotation

```
@GET
@Path("/{id}")
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Response getContact(@PathParam("id") int contactId) {
    Contact contact = contactRepository.getContact(contactId);
    if (contact != null) {
```

Sub-resources

- **@Path** may be used on classes and such classes are referred to as root resource classes
- **@Path** may also be used on methods of root resource classes

```
@Path("/contacts")
@Stateless
public class ContactService {
    @GET
    @Path("/countries")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getCountries() {
        List<String> countries = contactRepository.getCountries();
        String json = (new Gson()).toJson(countries);
        System.out.println(json);
        return Response.ok(json).build();
    }
```

HTTP Methods

- **@GET**, **@PUT**, **@POST**, **@HEAD** annotations correspond to the HTTP methods
  - Represent the actions to be performed on resources

```
@POST
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Response addContact(Contact contact) throws URISyntaxException {
    contact = contactRepository.addContact(contact);
    String location = String.format("/contacts/%s", contact.getContactId());
    String msg = String.format("contact #%s created successfully", contact.getCo
    return Response.created(new URI(location)).entity(msg).build();
}

@PUT
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Response updateContact(Contact contact) {
    contactRepository.updateContact(contact);
    String msg = String.format("Contact #%s updated sucessfully", contact.getCo
    return Response.ok(msg).build();
}
```

@Produces

- Specifies the MIME media types of **representations** a resource can produce typically XML or JSON (directly interpretable by JavaScript and Ajax)
  - Multiple representations of the same resource are allowed

```
@GET
@Path("/{id}")
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Response getContact(@PathParam("id") int contactId) {
    Contact contact = contactRepository.getContact(contactId);
    if (contact != null) {
        return Response.ok(contact).build();
    } else {
        String msg = String.format("Contact # %d not found", contactId);
        return Response.status(Response.Status.NOT_FOUND).entity(msg).build();
    }
}
```

@Consumes

- It is used to specify the MIME media types of representations a resource can consume

```
@POST
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Response addContact(Contact contact) throws URISyntaxException {
    contact = contactRepository.addContact(contact);
    String location = String.format("/contacts/%s", contact.getContactId());
    String msg = String.format("contact #%s created successfully", contact.getCo
    return Response.created(new URI(location)).entity(msg).build();
}
```

Building Responses

- **Response** class can be used to build the HTTP response to return to the caller
  - You may specify the response headers such as the **status code** and the **location** of a newly created resource

```
@POST
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Response addContact(Contact contact) throws URISyntaxException {
    contact = contactRepository.addContact(contact);
    String location = String.format("/contacts/%s", contact.getContactId());
    String msg = String.format("contact #%s created successfully", contact.getC
    return Response.created(new URI(location)).entity(msg).build();
}
```

Model the URIs

- In REST Services, endpoints are *resources* and identified with a URI
- For examples in an order management application we might have 3 top-level resources

```
/orders
/orders/{id}
/products
/products/{id}
/customers
/customers/{id}
```

Resources

- Java EE Tutorials

http://docs.oracle.com/javaee/7/tutorial/

https://www.youtube.com/playlist?list=PL74xrT3oGQfCCLFJ2HCTR_iN5hV4penDz

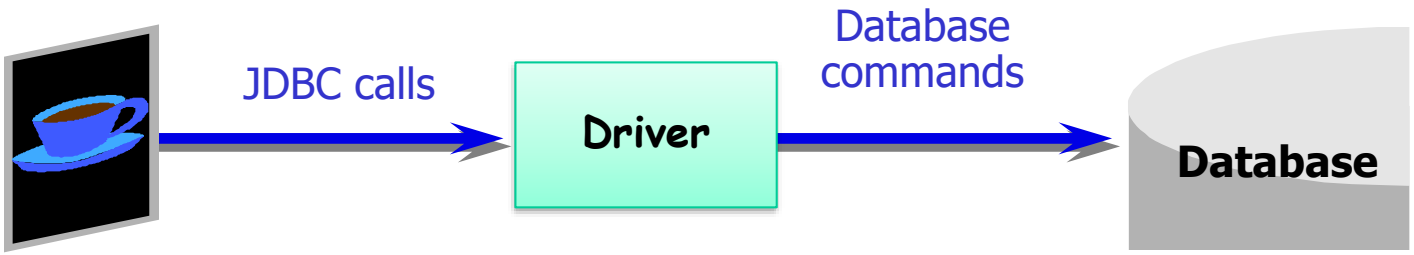
- Java EE hands-on lab

https://github.com/javaee-samples/javaee7-hol

- Code examples for '***Java EE 7 Essentials***' book

https://github.com/javaee-samples/javaee7-samples

JDBC Review

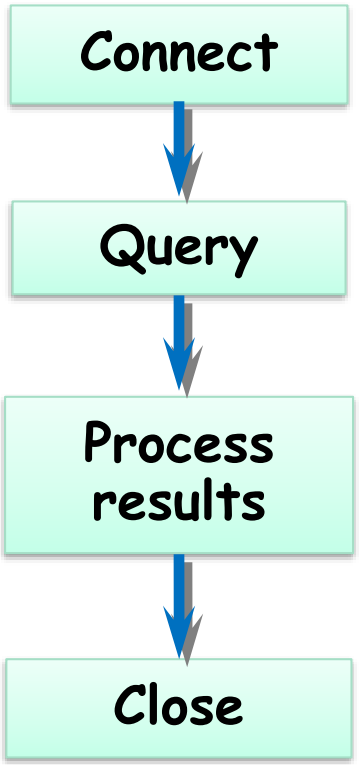


JDBC Overview

- 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'
```

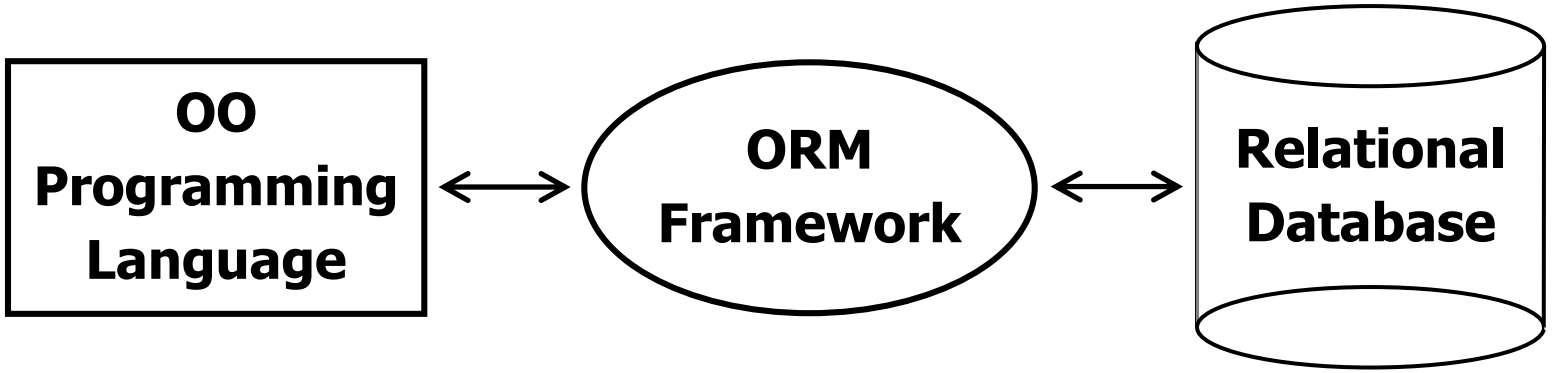
- **U**pdating data:

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

- **D**eleting data:

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

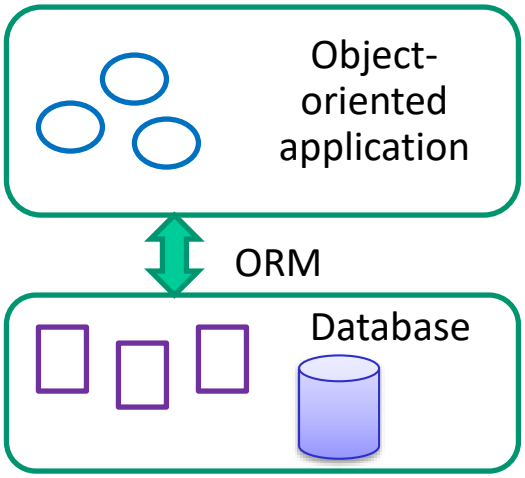
Java Persistence API (JPA)



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)



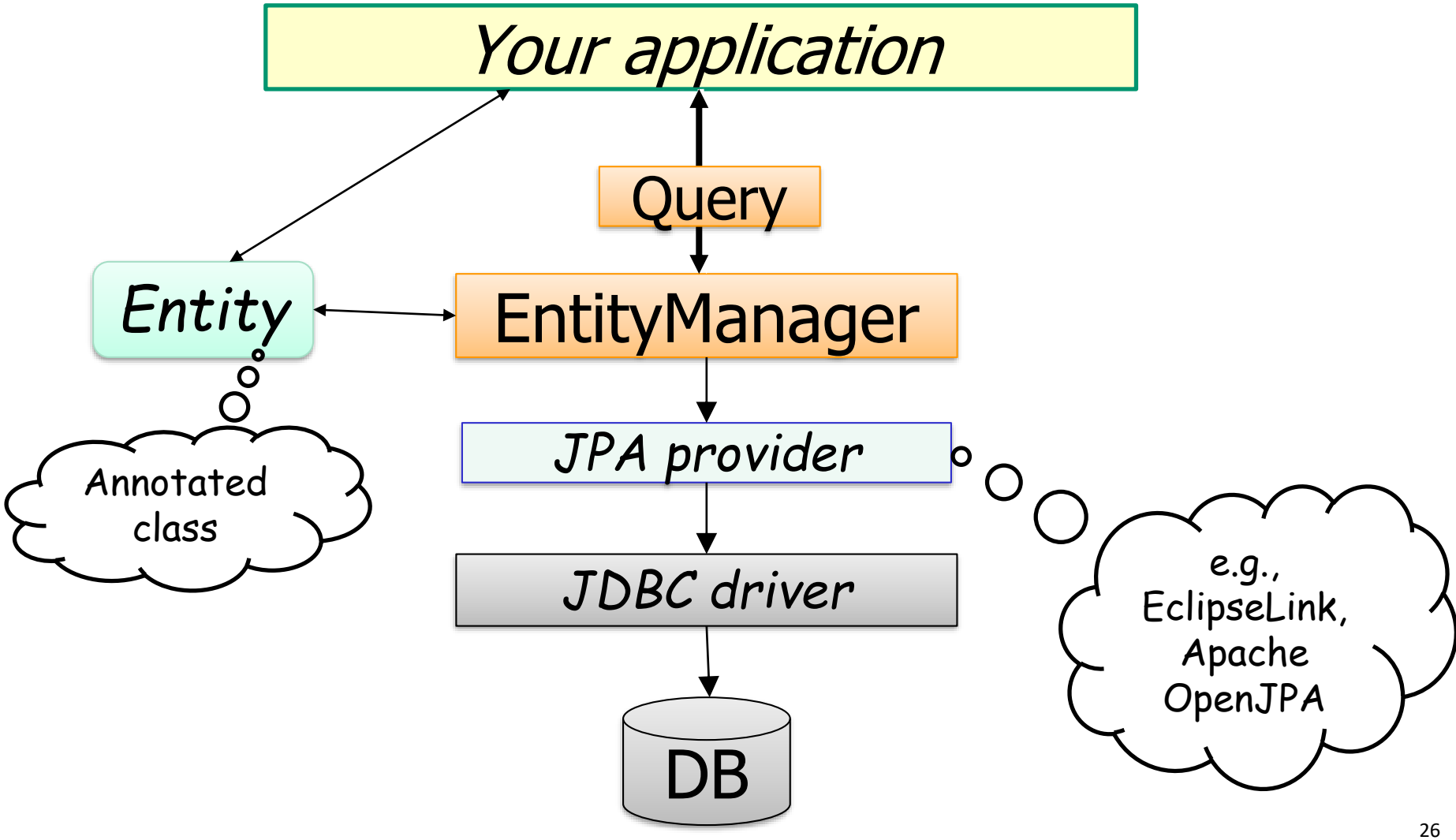
=> Solution is Object-Relational Mapping (ORM) – a software layer that **shuttles data back and forth between table rows and objects**

ORM Design Goals

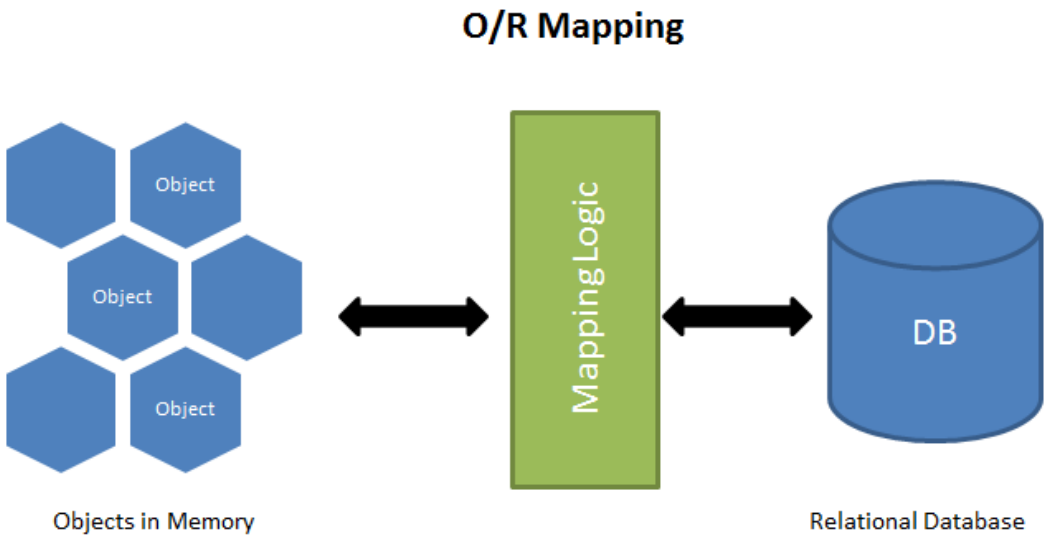
- Make it easier for OO developers to **save** and **retrieve** *objects* to/from DB
  - Query and retrieve data from tables and create the corresponding objects
  - **Synchronize** objects with the database by auto-generating the required insert/ update/delete SQL statements
  - Save and recreate ***associations***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

Java Persistence API (JPA) Architecture

JPA is a Java standard for ORM



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; }
        ….
}
```

Primary key

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:

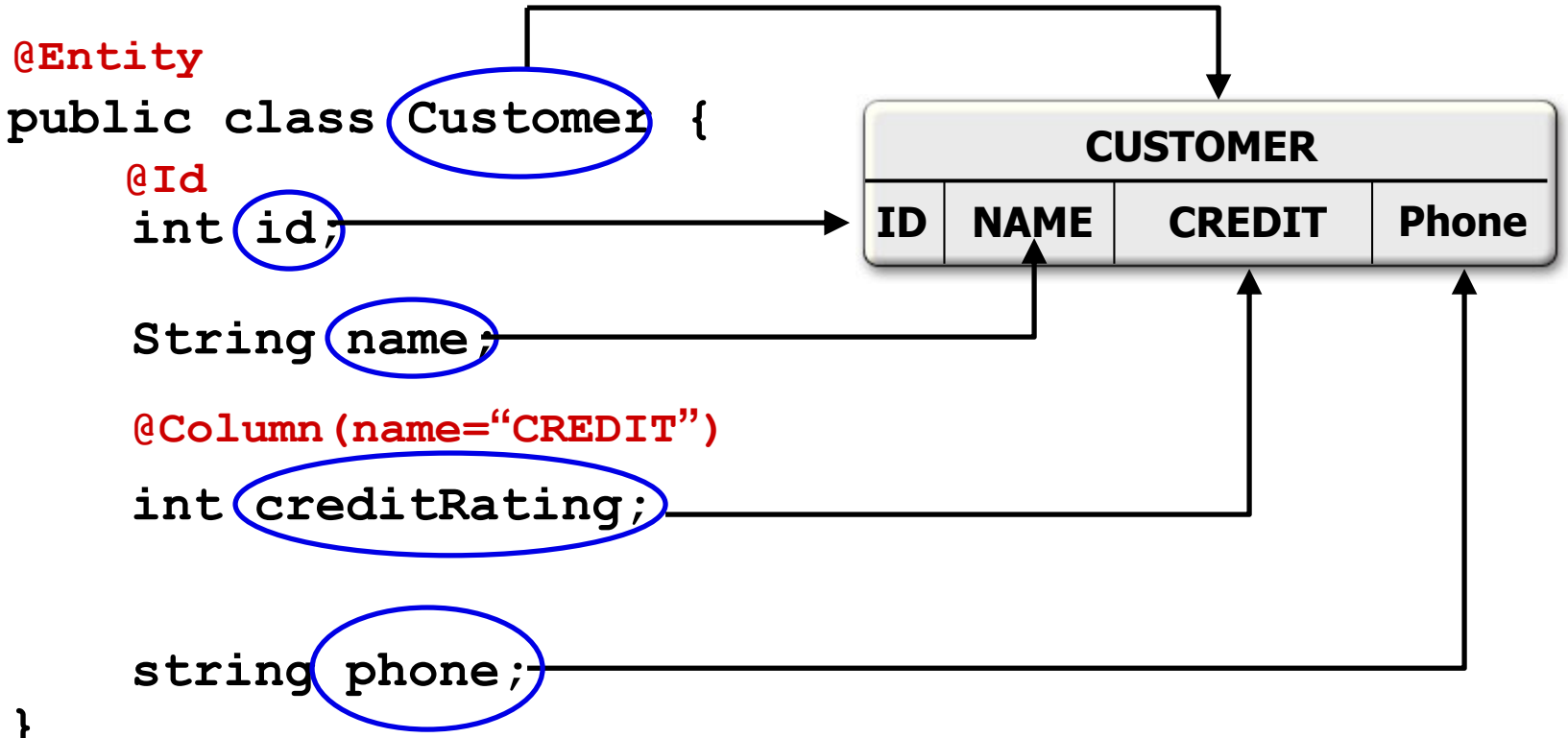
```
@Entity (name="FULLTIME_EMPLOYEE")
public class Employee{ …… }
```

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

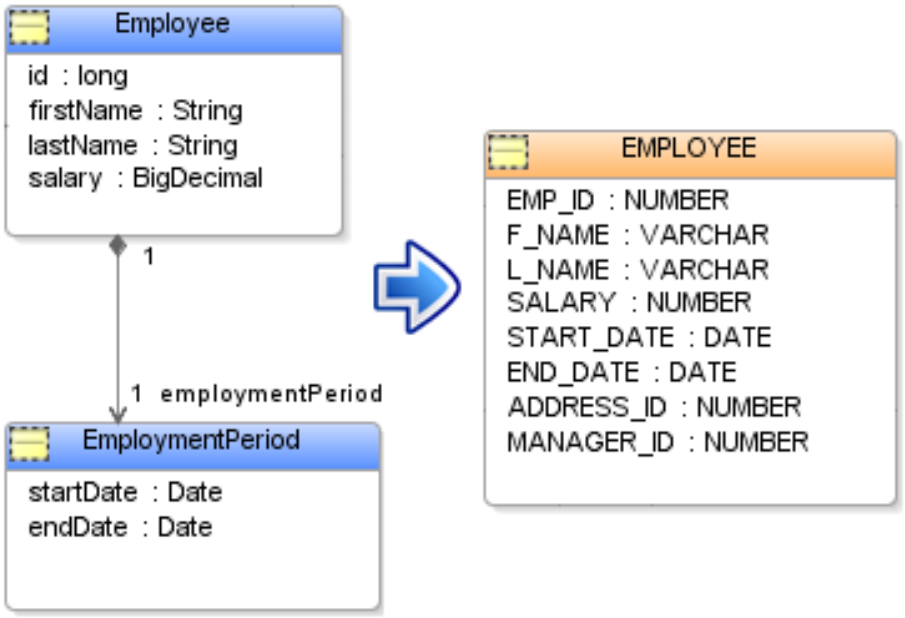
```
@Id @Column(name = “EMPLOYEE_ID”, nullable = false)
private String id;

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

Simple Mappings using Annotations



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



1-to-1 Whole-Part

Once Table

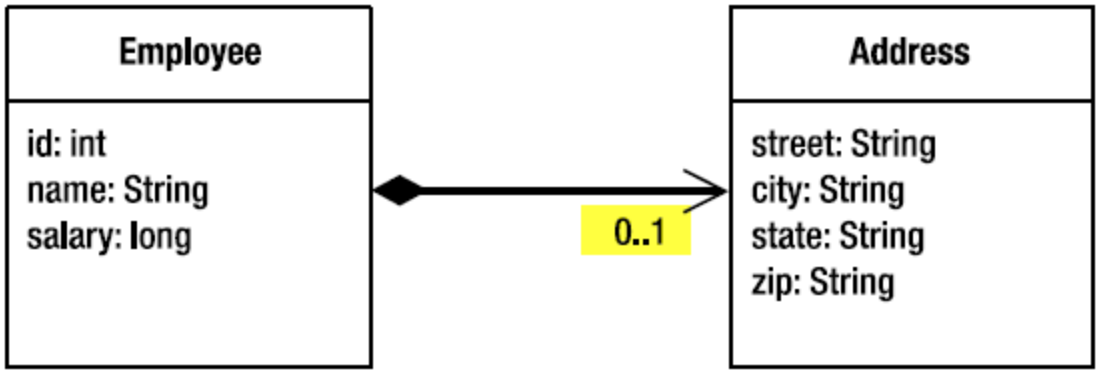
- Attributes of embeddable object is mapped to the same table that represents the owning entity
- An embeddable object is a dependent part and cannot be directly persisted or queried

```
@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

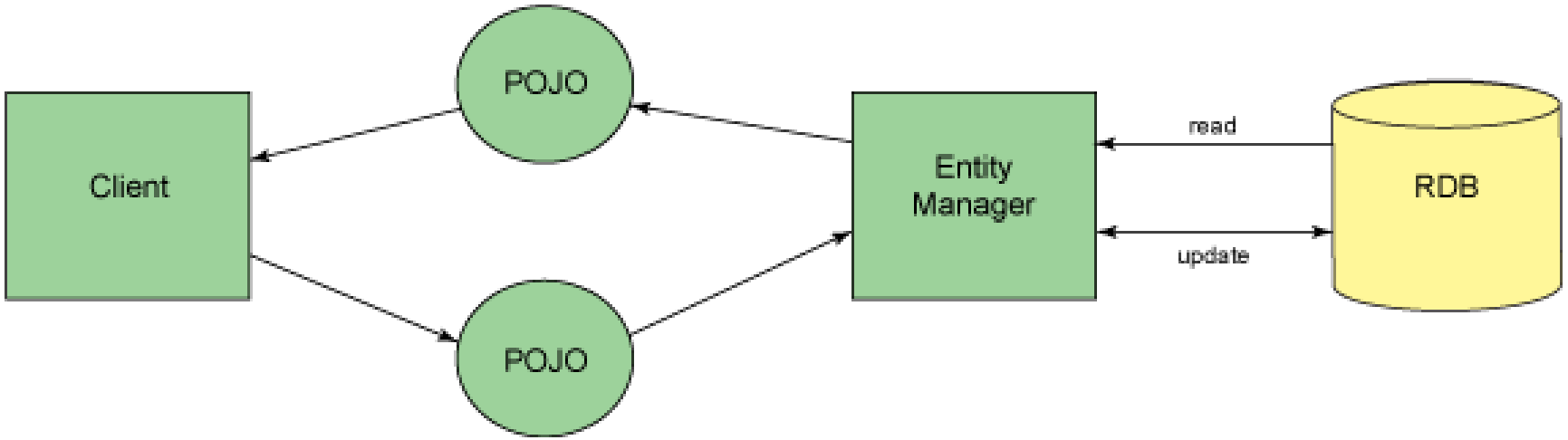


Employee *and* Address *relationship*

```
@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;
    // ...
}
```

JPA Programming

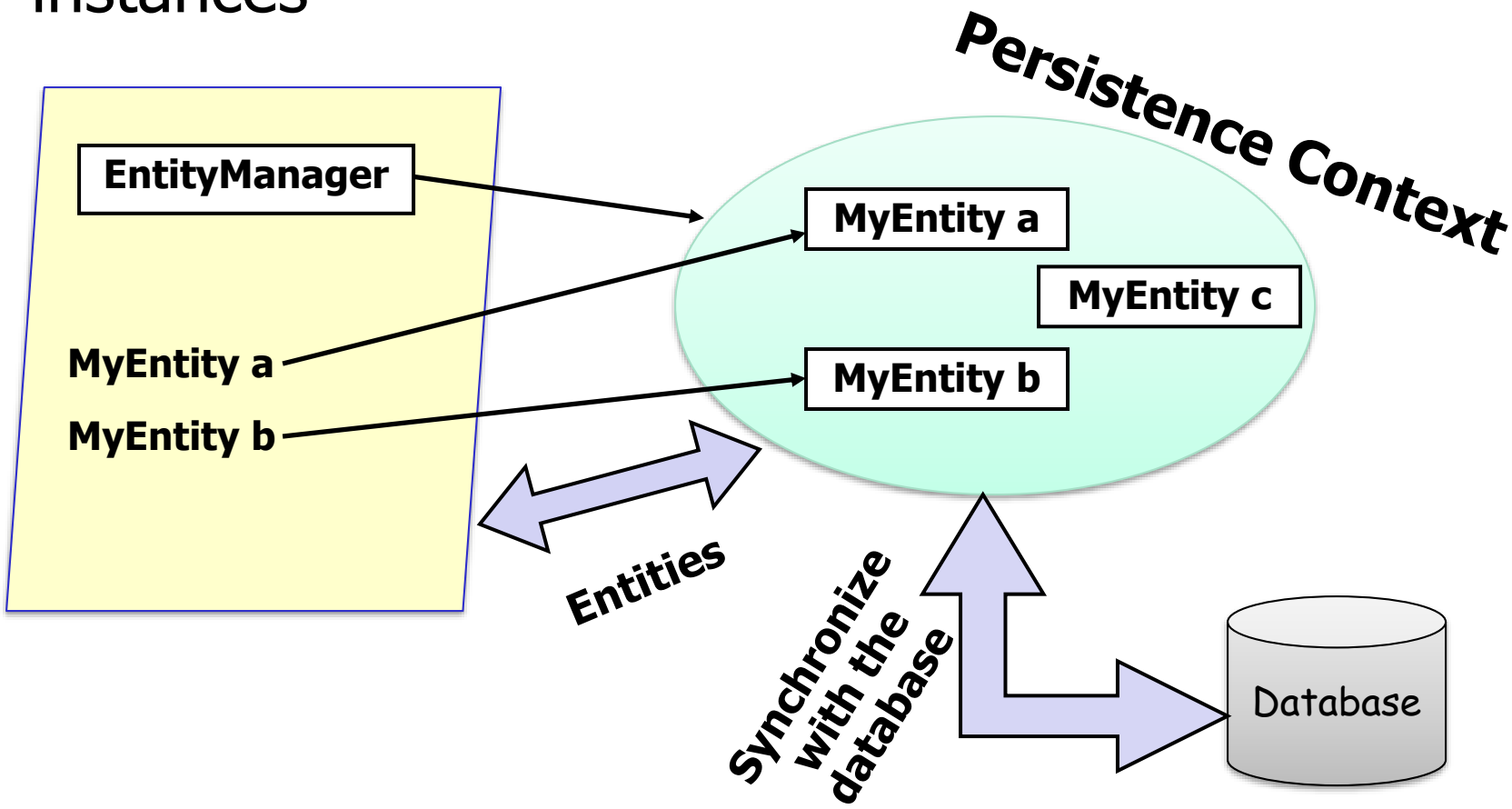


Entity Manager

- 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
  - `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

SQL Queries

- Create a query object using **`createNativeQuery()`** method and pass in the SQL query string
- 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 parameter

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

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

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

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