Java Persistence API (JPA) and Object Relational Mapping

Sadegh Aliakbary

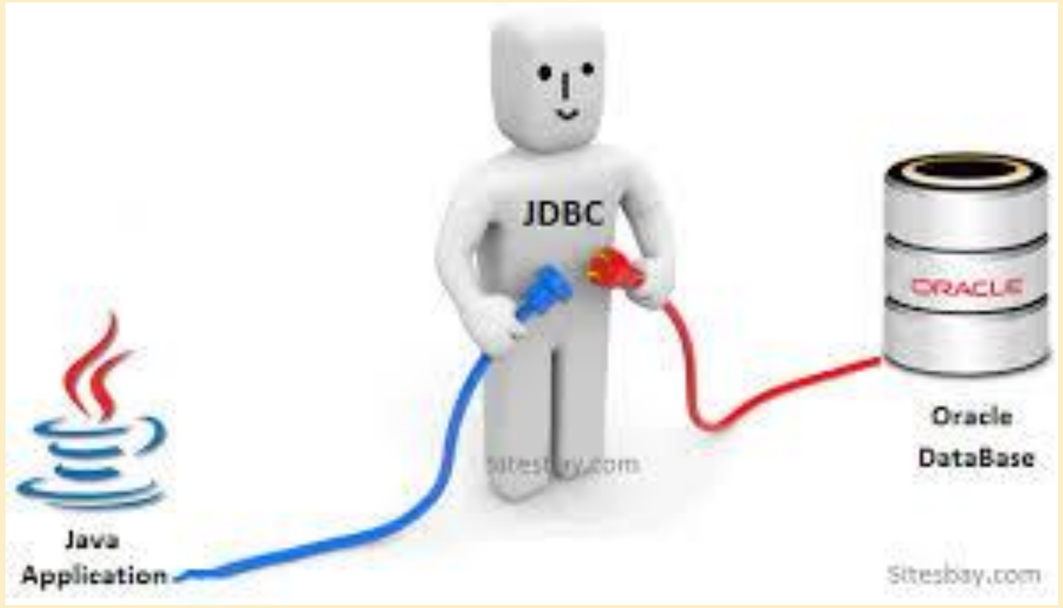
Table of Contents

Getting Ready

Start the Exercise Project

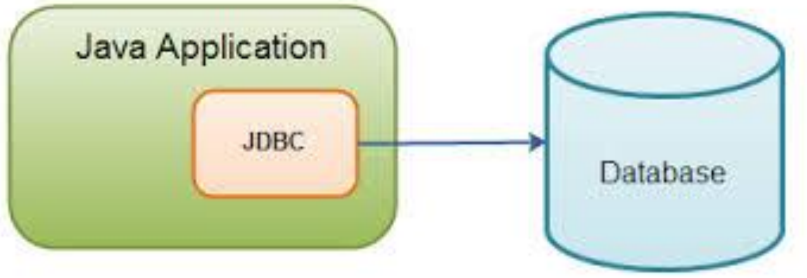
- Review the Layers (Service, Manager, DAO)
- Changes needed in Tomcat
  - Context.xml
- Deploying the Project
- Running Tomcat
  - It will generate some DB tables, that we will use in this session…

Java Database Connectivity (JDBC)



JDBC

- JDBC API provides a **standard database-independent interface** to interact with RDBMSs.

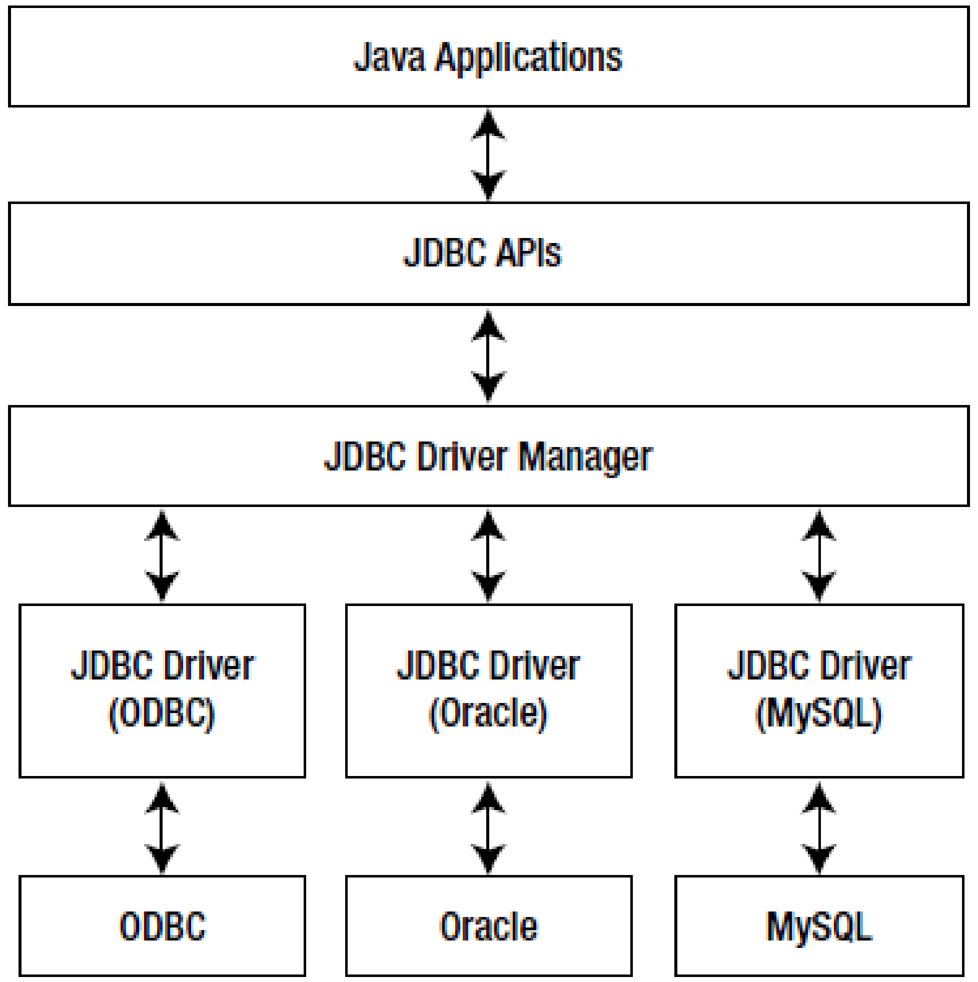


- JDBC API is used to
  - connect a java application to a database
  - query the data, and/or update data.
- Using JDBC API relieves you of the effort to learn specific protocols for different databases
  - But the SQL syntax of different DBs may still differ

JDBC Driver

- The collection of implementation classes
  that is supplied by a vendor
  to interact with a specific database
- The ‘JDBC driver’ is used to connect to the RDBMS.
- Is usually delivered as a JAR file
- E.g., Oracle JDBC driver, MySQL JDBC driver

The Architecture of JDBC



JDBC Example

```
//Class.forName("com.mysql.jdbc.Driver");    Now: Optional
String SQL = "SELECT * FROM classes";
try (
    Connection conn = DriverManager.getConnection(
        "jdbc:mysql://localhost/cse3330a",
        "root", "123");
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(SQL)
    ) {

  while (rs.next()) {
    System.out.println(rs.getString("prof"));
  }
}
```

JDBC

To connect to a database:

1. Obtain the JDBC driver class files (JAR)
   - add them to the CLASSPATH.
2. Construct a connection URL.
3. Use the static ‘getConnection()’ method of ‘DriverManager’ to establish a connection.
4. Create statements on the connection
   - and execute SQL queries

‘Statement’ interface

- ‘**execute()**’ method
  - is used to execute a SQL statement which does not return a value,
  - such as ‘CREATE TABLE’.
- ‘**executeUpdate()**’ method
  - is used for SQL that updates a database,
  - as in ‘INSERT’, ‘UPDATE’ and ‘DELETE’ SQL statements.
  - It returns the number of rows affected.
- ‘**executeQuery()**’
  - is used for SQL that produces a **resultset**,
  - as in ‘SELECT’ SQL statements.

Commit & Rollback

- The ‘**auto-commit**’ property for the ‘**Connection**’ object
  - is set to ‘true’ by default.
- If a ‘**Connection**’ is not in auto-commit mode:
- you must call the ‘**commit()**’ or ‘**rollback()**’ method of the ‘**Connection**’ object to commit or rollback the transaction

AutoCommit

```
try {
 Connection conn =   get the connection…
 conn.setAutoCommit(false);
 Statement stmt = conn.createStatement();
 stmt.executeUpdate(sql1);
 stmt.executeUpdate(sql2);

 conn.commit();
 conn.close();
}
catch (SQLException e){
 conn.rollback();
 e.printStackTrack();
 conn.close();
}
```

ResultSet Example

```
Connection conn =  …//get a Connection object …
Statement  stmt = conn.getStatement();
String sql = “select person_id, first_name,”+
                        “last_name,dob,income from person”;
ResultSet rs = stmt.executeQuery(sql);
while (rs.next())  {
  int personID=rs.getInt(1);
  String firstName=rs.getString(2);
  String lastName = rs.getString(3);
  java.sql.Date dob = rs.getDate(4);
  double income = rs.getDouble(5);
  //do something with the retrieved values from the cols.
}
```

Overview of JDBC Concepts

- JDBC Driver
- Connection
- Statement
- ResultSet
- Commit & rollback

JDBC Pros and Cons

- Pros
  - Clean and simple SQL processing
  - Very good for small applications
  - Simple syntax ➔ easy to learn
- Cons
  - Complex if it is used in large projects
  - Large programming overhead
  - No encapsulation
  - Query is DBMS specific

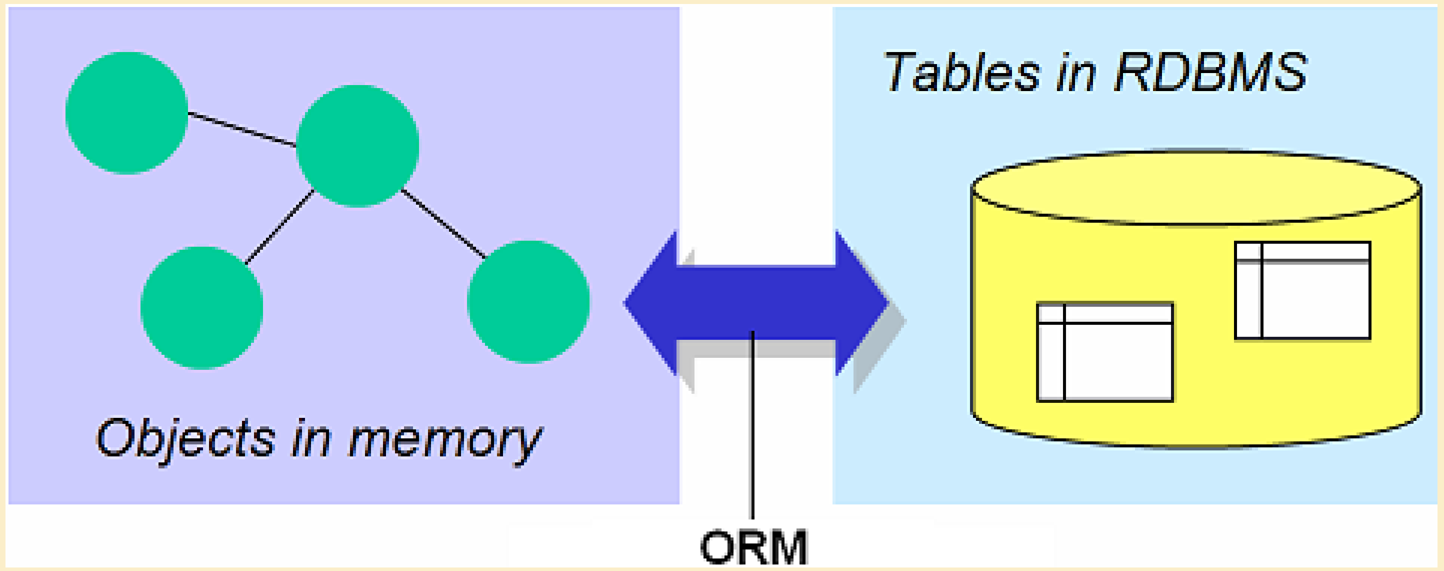
JDBC Exercise

- Implement **save** and **load** methods in ContactDAO
  - Using **<u>JDBC</u>**
- Add “age” field to ContactEntity
  - Revise the ContactDAO class
  - How much effort is needed?!

Other Concepts in JDBC

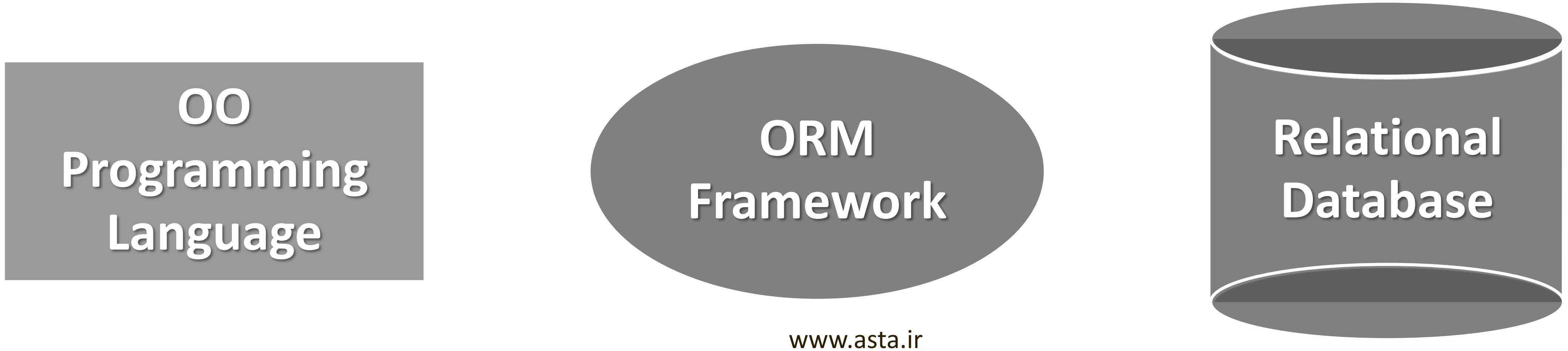
- PreparedStatement
  - And CallableStatement
- RowSet
- SQL Injection

Object Relational Mapping

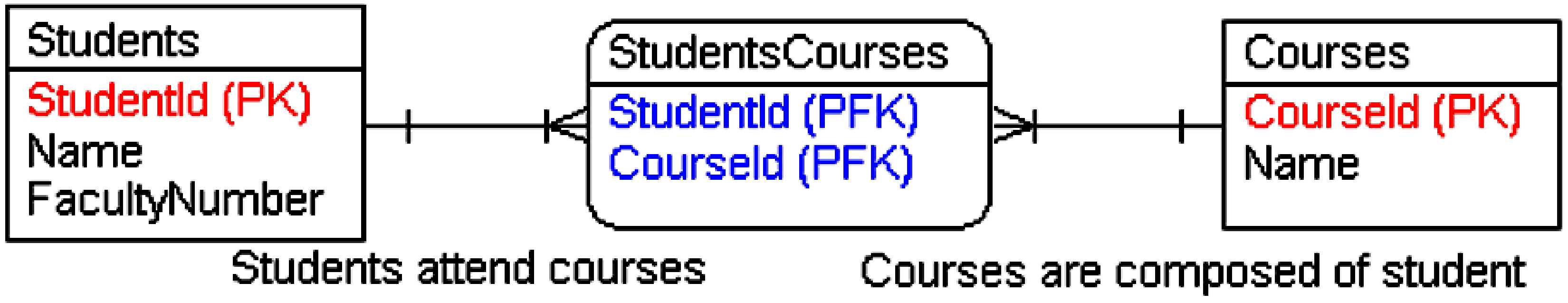


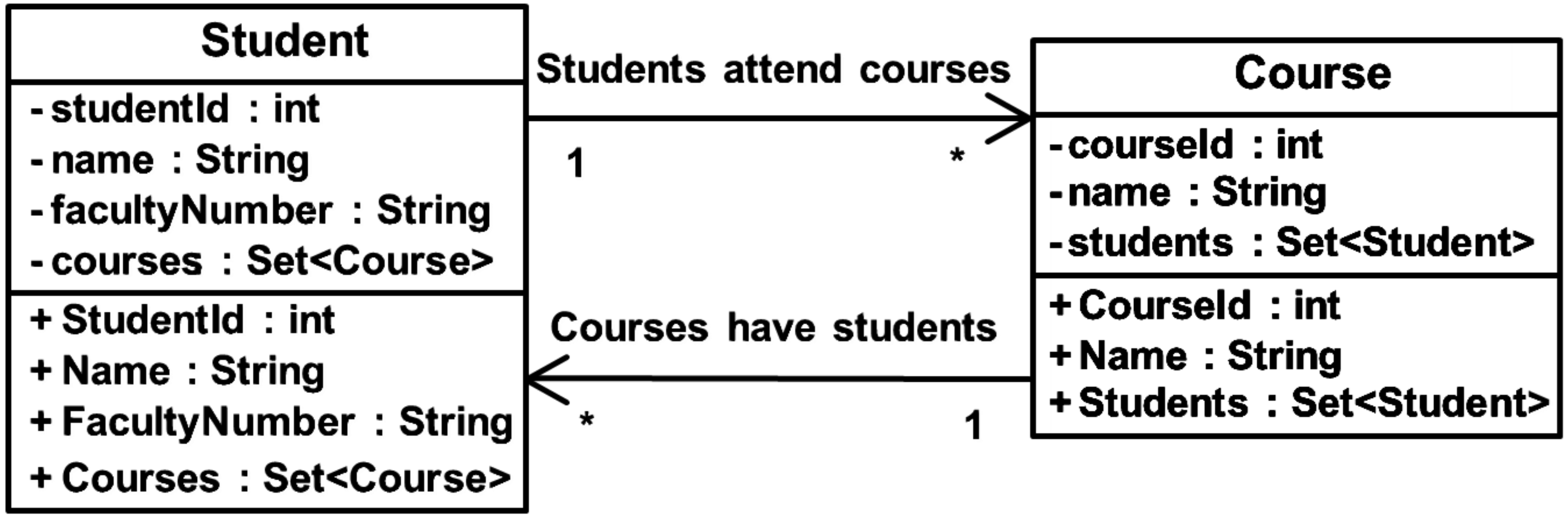
What is ORM?

- In **relational databases**:
  business entities are represented as tables + relationships
- In **object-oriented languages**:
  business entities are represented as classes
- Object relational mapping frameworks (ORMs) are used for:
  mapping business entities to database tables



ORM – Example


Students
StudentId (PK)
Name
FacultyNumber
StudentsCourses
StudentId (PFK)
CourseId (PFK)
Courses
CourseId (PK)
Name
Students attend courses
Courses are composed of student


Student
- studentId : int
- name : String
- facultyNumber : String
- courses : Set<Course>
+ StudentId : int
+ Name : String
+ FacultyNumber : String
+ Courses : Set<Course>
Students attend courses
1
*
Courses have students
*
1
Course
- courseId : int
- name : String
- students : Set<Student>
+ CourseId : int
+ Name : String
+ Students : Set<Student>

Simple Scenarios (Pseudocode)

```
Person person = new Person(“Ali Alavi”, 22, “0078934563”);
orm.save(person);
```

```
Person[] people = orm.loadAll(Person.class);
```

```
query = “SELECT from Student where instructor.dep.name=‘ce’  ”;
Entity[] found = orm.find(query);
```

How does an ORM framework work?

- E.g., how does it find the corresponding table for Student class?
  - It needs some meta-information
    - Class-table correspondence
    - Field-column correspondence
  - What is the format of such meta-data?
    - XML or Annotations

How does an ORM framework work? (cont’d)

- Now the ORM framework knows that:
  - ir.hr.Person class corresponds to PERSON table
  - and Person.name field corresponds to FULL_NAME column
- How does it operate?
  - Suppose you want to write the ORM.save(Object o) method
  - How to implement ***save*** method?
  - Java Feature: Reflection
    - Reflection dynamically finds and manipulates classes and fields

ORM Technologies

- ORM (Object-Relational Mapping) technologies
  - Map database tables to objects and enables CRUD operations, queries, concurrency, transactions, etc.
  - Simplifies the development of DB applications
- ORM in the Java world:
  - Java Persistence API (**JPA**) – the standard for ORM in Java
  - **Hibernate** – the most popular ORM library for Java (open source)
  - EclipseLink – ORM for Java by Eclipse foundation (open source)
  - …

ORM in Java: Products and History

- Hibernate ORM – http://hibernate.org/orm/
  - The first popular ORM framework in the Java world (2001)
  - Alternative to J2EE persistence technology called "EJB“
  - Belongs to JBoss (RedHat)

- EclipseLink – https://eclipse.org/eclipselink/
  - ORM for Java by Eclipse foundation
  - Maps classes to database, XML and Web services

ORM in Java: Products and History (2)

- Java Persistence API (JPA)
  - The official standard for ORM in Java and Java EE (JSR 338)
  - Implemented by most Java ORMs like Hibernate ORM, EclipseLink, OpenJPA, Apache JDO, Oracle TopLink, DataNucleus, …
  - Hibernate also supports JPA
  - EclipseLink is the reference implementation

History

- Java v1.1 (1997) included JDBC
- J2EE v1.2 (1999) introduced EJB Entity beans
- Entity beans introduced, so others did it better
  - Hibernate (2001)
- JPA v1.0 (JSR 220) was released in 2006
- JPA v2.0 (JSR 317) was released in 2009

Introduction to JPA

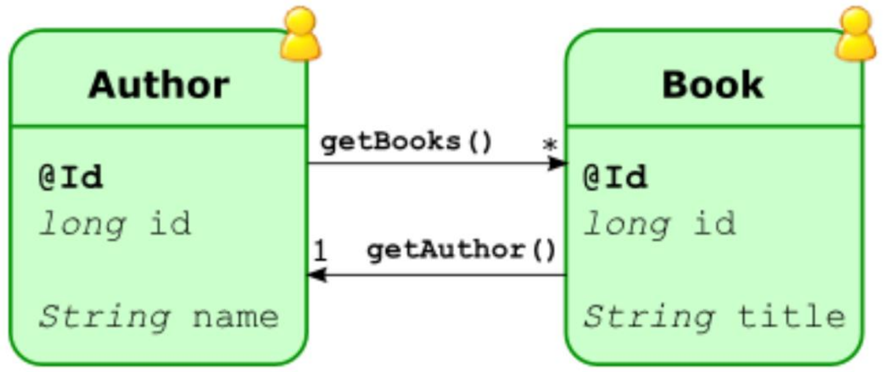
JPA

Java **P**ersistence **A**PI

About JPA

- What is Java Persistence API (JPA)?
  - Database persistence technology for Java (official standard)
    - Object-relational mapping (ORM) technology
    - Operates with POJO entities with annotations or XML mappings
    - Implemented by many ORM engines: Hibernate, EclipseLink, …
  - JPA maps Java classes to database tables
    - Maps relationships between tables as associations between classes
  - Provides CRUD functionality and queries
    - Create, read, update, delete + queries

Entities in JPA



- A JPA entity is just a POJO class
- The java classes corresponding to persistent data
  - E.g., User, Author, Book, etc.
  - Name more?!
- Non-final fields/properties, no-arguments constructor
- No required interfaces
- No requirement for business or callback interfaces
- Direct field or property-based access
- Getter/setter can contain logic (e.g. validation)

The Minimal JPA Entity: Class Definition

- Must be indicated as an Entity
  - `@Entity` annotation on the class:

```
@Entity
public class Employee {
 …
}
```

The Minimal JPA Entity: Primary Key

- Must have a persistent identifier (primary key):

```
@Entity
public class Employee {
  @Id int id;

  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
}
```

Primary Key (Id) Definitions

- Simple id – single field/property

```
@Id int id;
```

- Compound id – multiple fields

```
@Id String firstName;
@Id String lastName;
```

Primary Key Identifier Generation

- Identifiers can be generated in the database
  - `@GeneratedValue` on the ID field

```
@Id @GeneratedValue int id;
```

- Several pre-defined generation strategies:
  - `IDENTITY. uses a database identity column`
  - `SEQUENCE. uses a database sequence`
  - `AUTO. Either identity or sequence or …`
    - (depending on the underlying DB)

Simple Column Mappings

- Mapping a class field to a database column:

```
@Entity
public class Message {
  private String message;
  public void setMessage(String msg) { message = msg; }
  public String getMessage() { return message; }
}
```

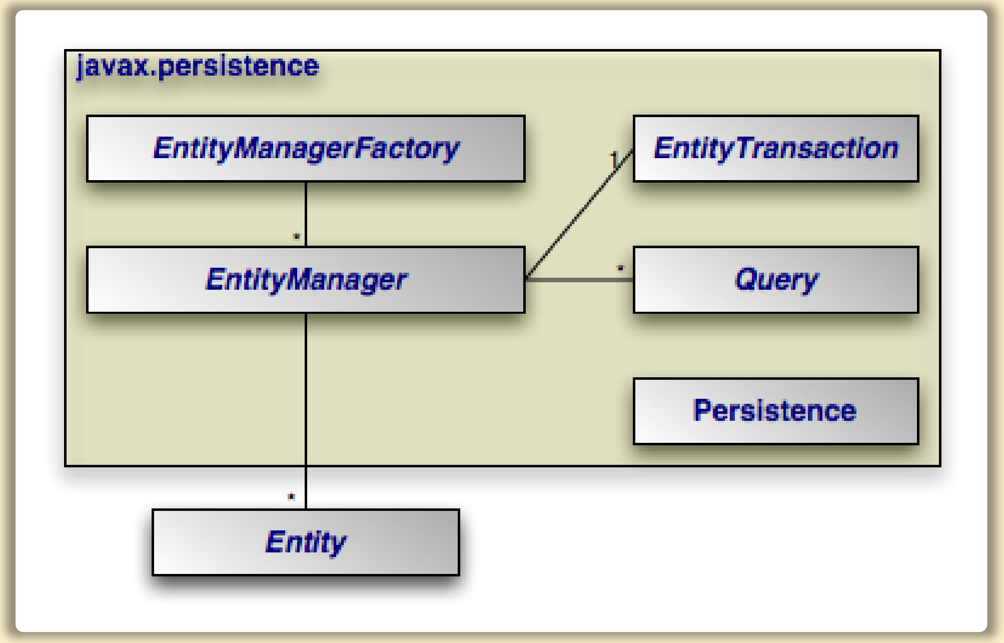
- A column name can be explicitly given:

```
@Column(name="SAL")
private double salary;
```

Example

```java
@Entity
@Table(name = "ADDRESS")
public class Address implements Serializable {
    @Id
    @Column(name = "ID")
    private Integer id;
    @Column(name = "CITY")
    private String city;
    @Column(name = "STATE")
    private String state;
    @Column(name = "STREET")
    private String street;
    @Column(name = "ZIP")
    private String zip;
    ...
}
```

Persistence Contexts and EntityManager



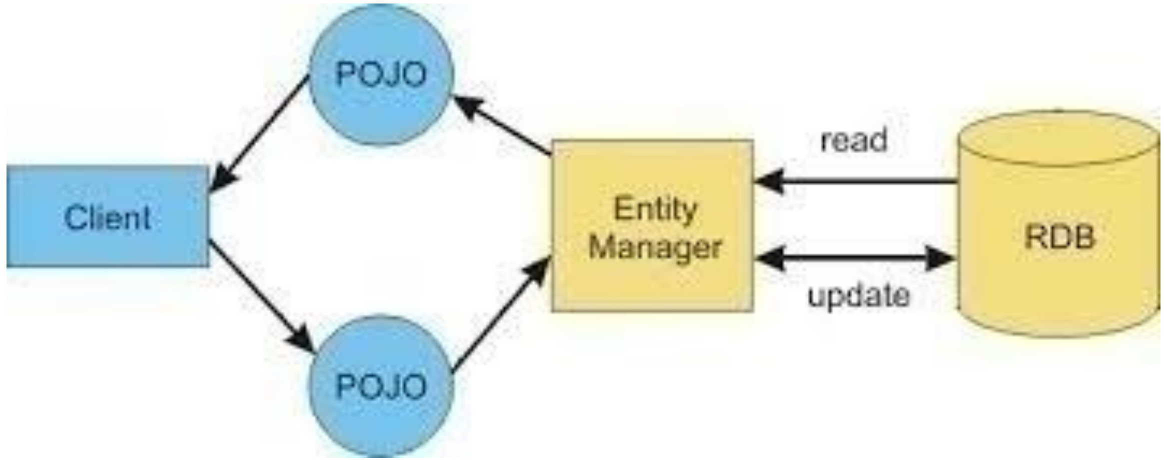
Manipulating Database Entities

Persistence Context (PC)

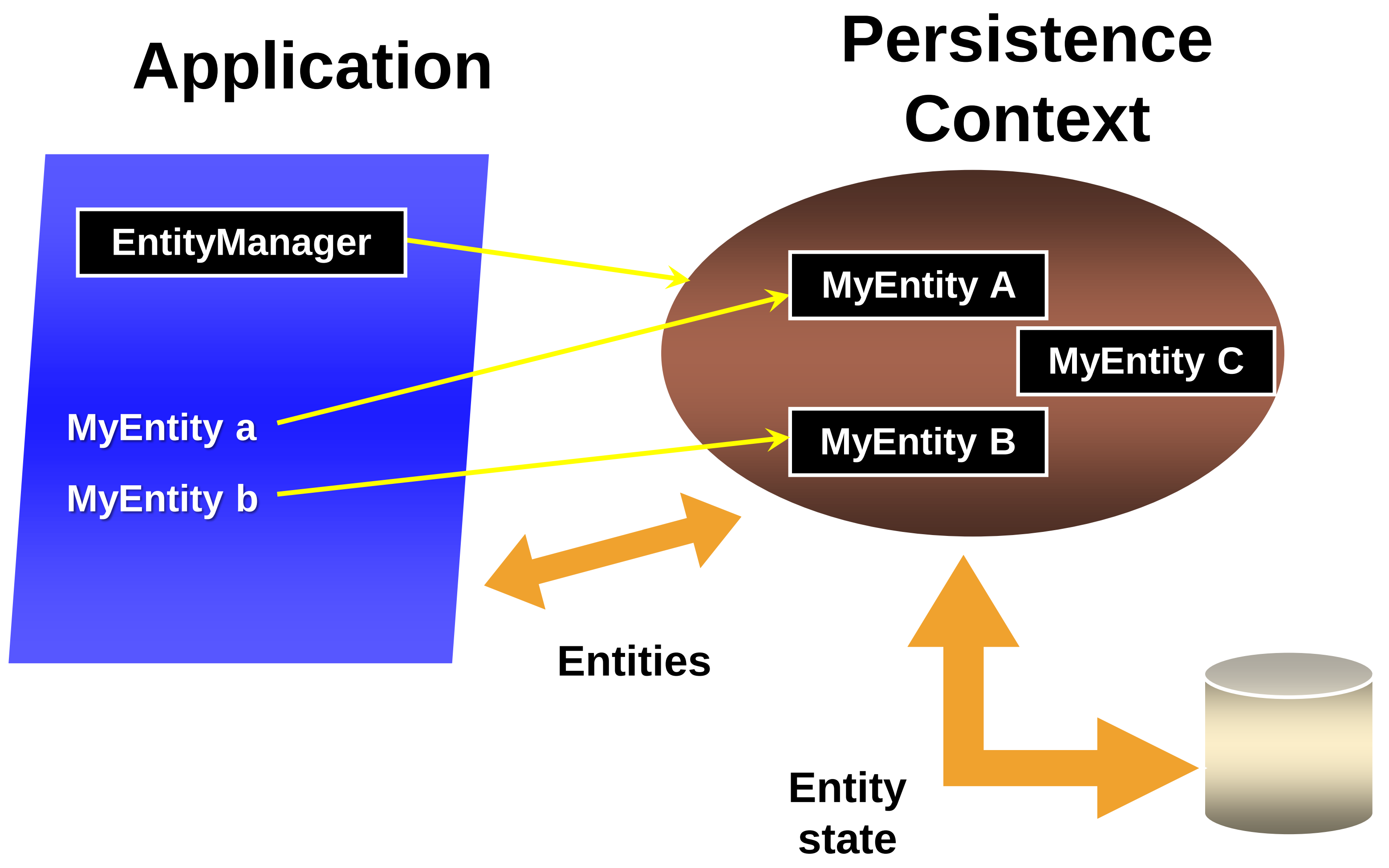
- The persistence context (PC)
  - Holds a set of “**managed**” entity instances
  - Keyed by persistent identity (primary key)
  - Only one entity with a given persistent ID may exist in the PC
- Managed by **`EntityManager`**
  - The PC change as a result of operations on **`EntityManager`** API

The EntityManager

- Client-visible object for operating on entities
  - API for all the basic persistence operations (CRUD)
  - Manages connection and transaction
- Can think of it as a **proxy** to a persistence context



Persistence Context (PC) and Entities



Overview of PC

- Entities are managed by an **EntityManager** instance **using persistence context**
- Each **EntityManager** instance is associated with **a *persistence context***
- Within the **persistence context**, the entity instances and their **lifecycle are managed**
- A **persistence context** is like a ***cache*** which contains a set of **persistent entities**
- So once the **transaction is finished**, all persistent objects are detached from the **EntityManager's persistence context** and are no longer managed

Operations on Entities

- **`EntityManager`** API
  - **`persist()`** – persists given entity object into the DB
    - (SQL INSERT)
  - **merge()** – updates given entity in the DB
    - (SQL UPDATE)
  - **`remove()`** – deletes given entity into the DB
    - (SQL DELETE by PK)

EntityManager.flush() method

- **flush()**
- Forces changes in the PC to be sent to the database
- It is automatically called on transaction commit (but not vice versa)
  - It does not call transaction commit. The transaction may be rolled-back
- Note:

Entitymanager.flush() vs EntityManager.getTransaction().commit

Other EntityManager Operations

- **`find()`** – execute a simple query by PK
  - (SQL SELECT by PK)
- **`createQuery()`** – creates a query instance using dynamic JPQL
- **`createNamedQuery()`**
  - Creates an instance for a predefined JPQL query
- **`createNativeQuery()`**
  - Creates an instance for an SQL query

EntityManager.persist()

- Insert a new entity instance into the database
- Save the persistent state of the entity and any owned relationship references
- Entity instance becomes **managed**

```
public Customer createCustomer(int id, String name) {
    Customer cust = new Customer(id, name);
    entityManager.persist(cust);
    return cust;
}
```

Updating an Entity

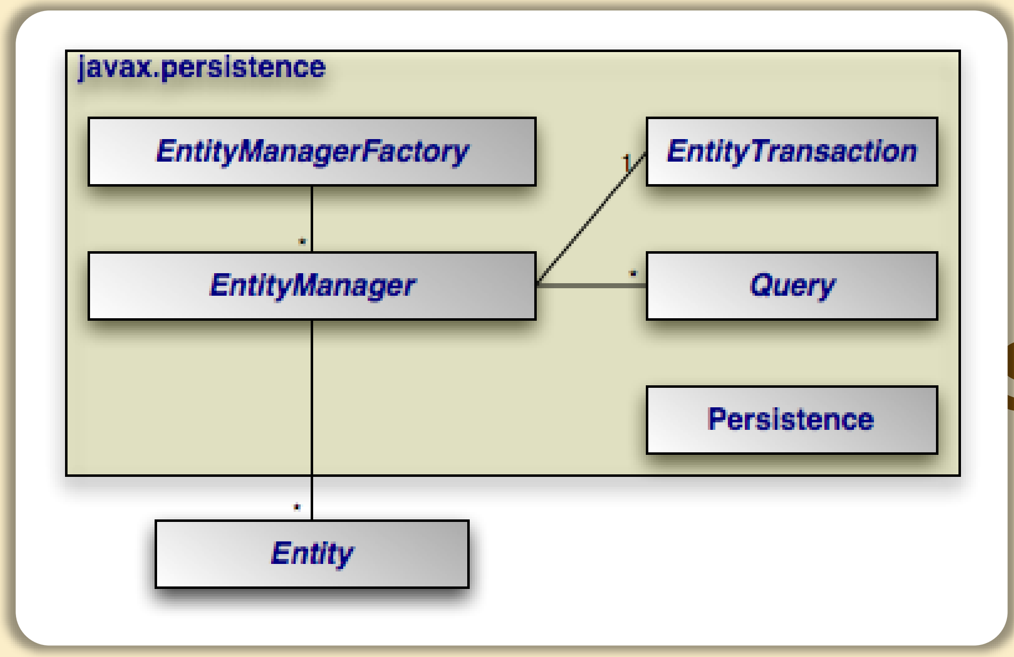
- 1- By just changing a **managed*** entity
  - And committing the transaction
- 2- By invoking **merge()** over a **detached*** entity
  - And committing the transaction

- * **Detached/Managed** entities: described later

find() and remove()

- **`find()`**
  - Obtain a managed entity instance (SQL SELECT by PK)
  - Return null if not found
- **`remove()`**
  - Delete a managed entity by PK

```
public void removeCustomer(Long custId) {
  Customer cust = entityManager.find(Customer.class, custId);
  entityManager.remove(cust);
}
```



Creating Standalone JPA Applications

EntityManagerFactory Class

- **`javax.persistence.EntityManagerFactory`**
- Obtained by the **`Persistance`**
- Creates **`EntityManager`** for a named persistence unit or configuration
- In Java SE environment the persistence unit configuration is defined in the **`META-INF/persistence.xml`** file

Sample Config: META-INF/persistence.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0">
  <persistence-unit name="hellojpa">
    <class>model.Message</class>
    <properties>
      <property name="ConnectionURL"
        value="jdbc:derby:messages-db;create=true"/>
      <property name="ConnectionDriverName"
        value="org.apache.derby.jdbc.EmbeddedDriver"/>
      <property name="ConnectionUserName" value=""/>
      <property name="ConnectionPassword" value=""/>
    </properties>
  </persistence-unit>
</persistence>
```

Automatic Table Generation

- JPA may be configured to automatically create the database tables
  - During application deployment
  - Typically used during the development phase of a **release**,
  - not against a **production database**
- ***javax.persistence.schema-generation.database.action*** property
  - Configures the behavior of JPA for table generation

javax.persistence.schema-generation.database.action

```
<persistence ...>
 <persistence-unit name="WISE">
   ...
   <properties>
   ...
   <property name="javax.persistence.schema-generation.database.action"
             value="drop-and-create"/>
   </properties>
 </persistence-unit>
</persistence>
```

- Since JPA 2.1 (2013)
- Possible values:
  - none, create, drop-and-create, …

JPA Bootstrap – Example

```
public class PersistenceExample {
  public static void main(String[] args) {
    EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("hellojpa");
    EntityManager em = emf.createEntityManager();
    em.getTransaction().begin();
    // Perform finds, execute queries, update entities, etc.
    em.getTransaction().commit();
    em.close();
    emf.close();
  }
}
```

JPA Configuration in Spring

JPA Configuration with Spring

- Many things should be configured for JPA: For Example?!
  - Which JPA implementation is used?
    - Hibernate? EclipseLink?
  - Database Configuration
    - Which DBMS? Oracle? MySQL?
    - DB URL? User/pass?
  - Which packages should be scanned for entities?
  - How to manager Transactions?
  - How to **inject an EntityManager** into DAO classes?
- **Spring** makes configurations easy

Spring Configuration Example

```
<bean id="myEmf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="packagesToScan" value="ir.asta.training.contacts.entities" />
    <property name="jpaVendorAdapter">
      <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
    </property>
    <property name="jpaProperties">
      <props>
        <prop key="hibernate.hbm2ddl.auto">create-drop</prop>
        <prop key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>
      </props>
    </property>
  </bean>
```

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.cj.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost:3306/spring_jpa" />
    <property name="username" value=“root" />
    <property name="password" value=“123" />
  </bean>
```

```
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="myEmf" />
  </bean>
  <tx:annotation-driven />
```

Now, With Spring We Can:

```
@Named("contactDao")
public class ContactDao {

    @PersistenceContext
    private EntityManager entityManager;

    public void save(ContactEntity entity) {
    entityManager.persist(entity);
    }

    public void delete(ContactEntity entity) {
     entityManager.remove(entity);
    }
}
```

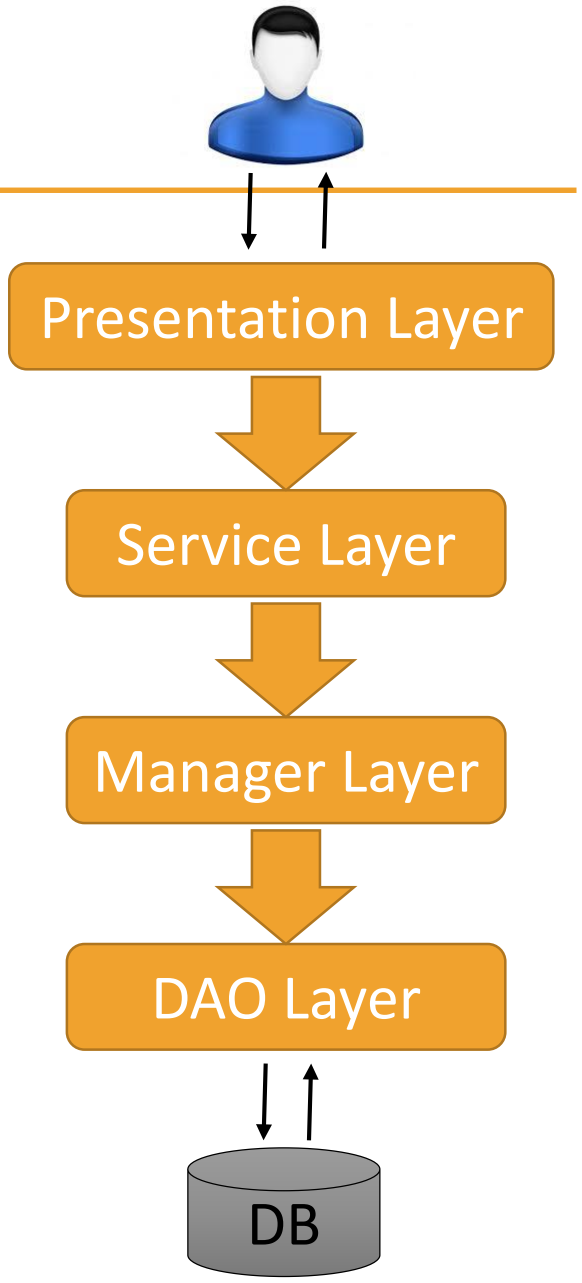
Exercise on JPA and Spring

- Review the Layers
- Review the Spring configurations

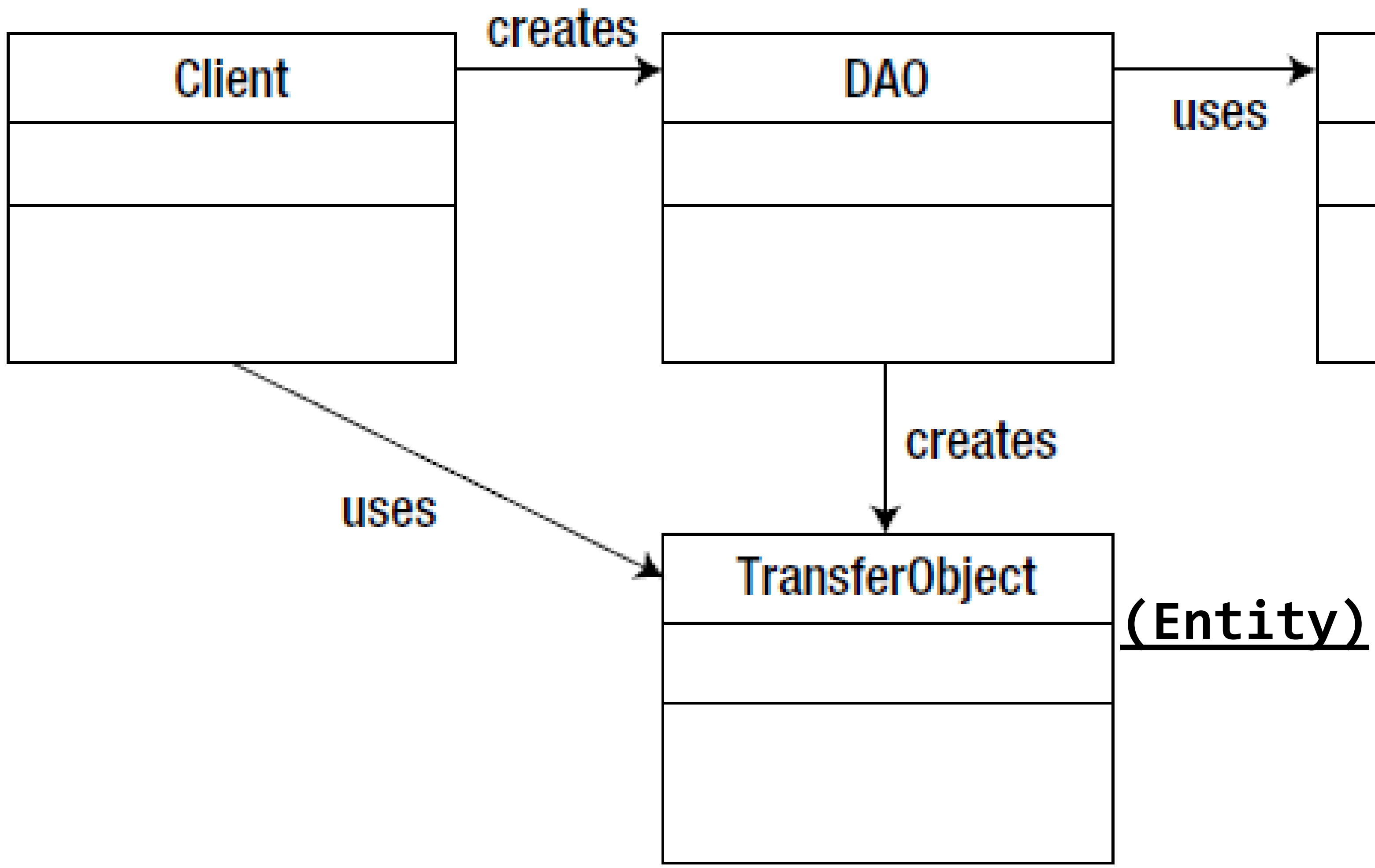
DAO and Generic DAO Patterns

Layering

- Enterprise applications usually are divided into logical layers
- Three-layer
  - UI, Business Logic, Data
- Four-Layer
  - UI, Service, Business Logic, Data
- Data Layer: Collection of classes related to data access (DAO layer)



Data Access Object Pattern (DAO)



DAO

- Abstracts the details of the underlying persistence mechanism
- Hides the implementation details of the data source from its clients
- **Loose coupling** between core business logic and persistence mechanism

1. Write Entity Classes

```
@Entity  @Table(name="WISE_CONTACT")
public class ContactEntity {
     Long id;
     String name;
     @Id
     @Column(name = "CONTACT_ID")
     @GeneratedValue(strategy=GenerationType.AUTO)
     public Long getId() {
        return id;
     }
     public void setId(Long id) {   this.id = id;   }

     @Basic(fetch=FetchType.EAGER)
     @Column(name="NAME_")
     public String getName() {
        return name;
     }
     public void setName(String name) { this.name = name; }
}
```

2. DAO implementations

```
@Named("contactDao")
public class ContactDao {

    @PersistenceContext
    private EntityManager entityManager;

    public void save(ContactEntity entity) {
    entityManager.persist(entity);
    }

    public void delete(ContactEntity entity) {
     entityManager.remove(entity);
    }
}
```

Generic DAO

- The problem with many DAO implementations
- Similar methods for
  - Load
  - Save, update, delete
  - Search
- The solution?
  - A Generic DAO class
  - DAO implementation classes inherit from it
    - and implement the DAO interface

Generic DAO Example

```
class AbstractJpaDAO<T extends BaseEntity<U>, U extends Serializable>{
    @PersistenceContext
    private EntityManager entityManager;
    public void save(T entity) {
        etityManager.persist(entity);
    }
    public T load(U id) {
        return (T) etityManager.find(getEntityClass(), id);
    }
    public abstract Class<T> getEntityClass() ;
    //and find, search, update, many other methods.
}
```

```
public class ContactDao extends AbstractJpaDAO<ContactEntit, Long>{
 public Class<ContactEntity> getEntityClass() {
  return ContactEntity.class;
 }
}
```

Exercise on JPA Basics

- Implement **find** method in ContactDAO
  - Using JPA

Transaction Management
(by Spring Transaction)

Programmatic Transactions

```
public void save(ContactEntity entity) {
  EntityTransaction entityTransaction =
       entityManager.getTransaction();

  entityTransaction.begin();

  entityManager.persist(entity);

  entityTransaction.commit();
}
```

Using Spring @Transactional

- With Spring @Transactional, the transaction management is made simple

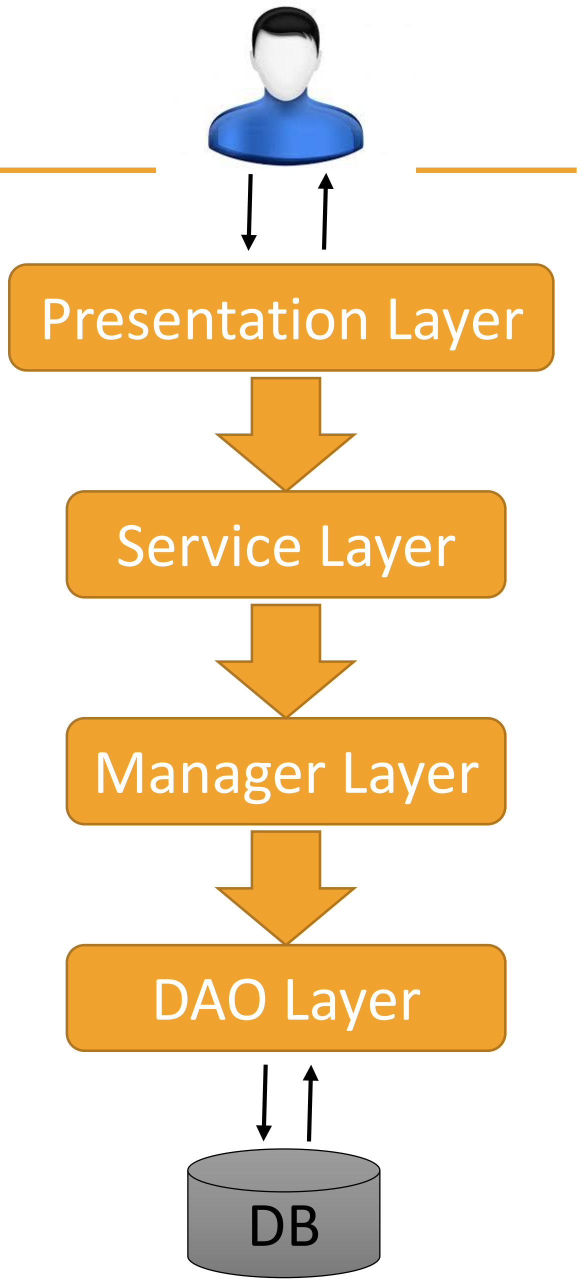
```
@Transactional
public void businessLogic() {
    ... use entity manager inside a transaction ...
}
```

- This **declarative transactions** are much more convenient and readable
- To use declarative transactions, we should configure it:

```
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="myEmf" />
 </bean>
 <tx:annotation-driven />
```

Layering and Transactions

- Methods of which layer become transactional?
- Manager
- Why?



Exercise on JPA

- Implement **save** method in ContactDAO
  - Using JPA
  - Note to Transactions
- Add “**email**” field to ContactEntity
  - Revise the ContactDAO class
  - How much effort is needed?!
    - Compare it with the JDBC approach
- Simulate an “account-transfer” method in Manager
  - Which invokes two dao methods in a transactional method
  - Test the transactionality! Does it roll-back?

JPA Queries

Using **JPQL**

JPA Queries

- JPA supports powerful querying API
  - Dynamic or statically defined (**named queries**)
- Criteria using **JPQL** (Java Persistence API Query Language)
  - SQL-like language
- Native SQL support (when required)
- Named parameters bound at execution time (**no SQL injection**)
- **Pagination** and ability to restrict size of result
- Single/multiple-entity results, data projections
- Bulk update and delete operations on an entity

JPA Query API

- Query instances are obtained from factory methods on **`EntityManager`**, e.g.

```
Query query = entityManager.createQuery(
   "SELECT e from Employee e");
```

- JPA query API:
  - **`getResultList()`** – execute query returning multiple results
  - **`getSingleResult()`** – execute query returning single result
  - **`executeUpdate()`** – execute bulk update or delete

JPA Query API (2)

- **`setMaxResults()`** – set the maximum number of results to retrieve
- **`setFirstResult()`** `– set the first result row number (starting row in paginations)`
- **`setParameter()`** – bind a value to a named or positional parameter
- **`setFlushMode()`** – apply a flush mode to the query when it gets run

Example: Pagination with JPQL

```
Query query = entityManager.createQuery("From Foo");
int pageNumber = 1;
int pageSize = 10;
query.setFirstResult((pageNumber-1) * pageSize);
query.setMaxResults(pageSize);
List fooList = query.getResultList();
```

Dynamic Queries – Example

- Use **`createQuery()`** factory method at runtime
  - Pass in the JPQL query string
- Get results by **`getResultList()`** / **`getSingleResult()`**

```
public List findAll(String entityName){
  return entityManager.createQuery(
    "select e from " + entityName + " e")
    .setMaxResults(100)
    .getResultList();
}
```

JPQL Examples

```
Query query = em.createQuery("Select e FROM Employee e WHERE e.id = :id");
query.setParameter("id", id);
Employee result2 = (Employee)query.getSingleResult();
```

```
TypedQuery<Object[]> query = em.createQuery(
  "SELECT c.name, c.capital.name FROM Country AS c", Object[].class);
List<Object[]> results = query.getResultList();
for (Object[] result : results) {
  System.out.println("Country: " + result[0] + ", Capital: " + result[1]);
}
```

Named Queries

- Named queries are once defined and invoked later many times

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

```
public List findSalesByCustomer(Customer cust) {
  return (List<Customer>)entityManager.
    createNamedQuery("Sale.findByCustId")
    .setParameter("custId", cust.getId())
    .getResultList();
}
```

Named Query Examples

```
@Entity
@NamedQuery(name="Country.findAll", query="SELECT c FROM Country c")
public class Country {
   ...
}
```

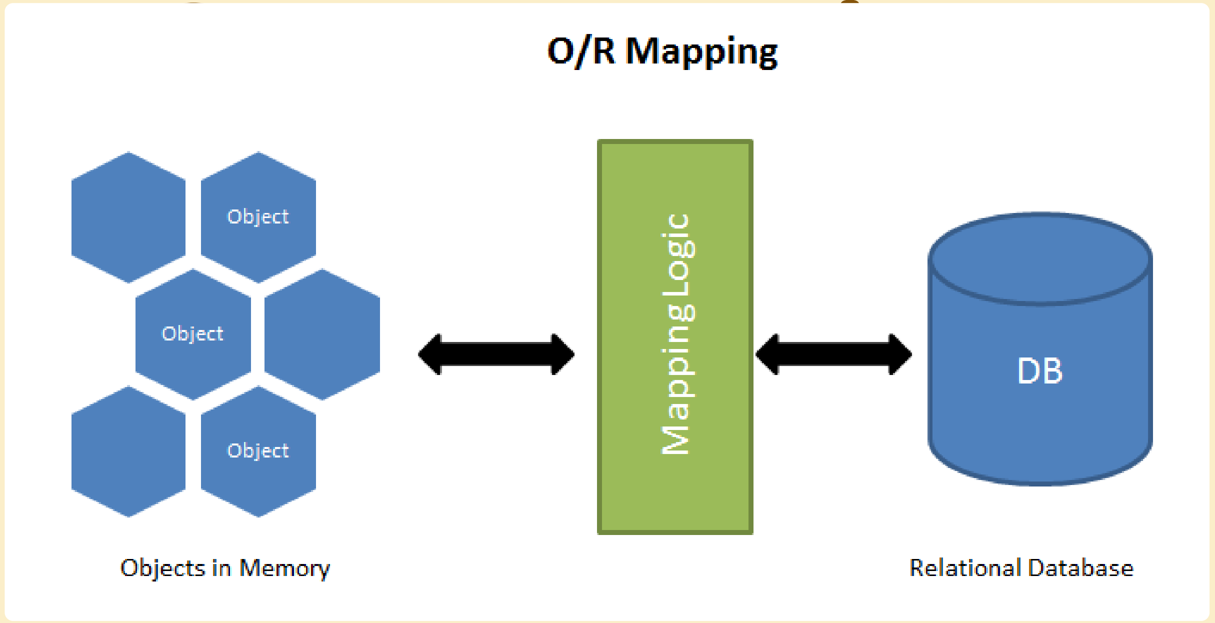
```
@Entity
@NamedQueries({
 @NamedQuery(name="Country.findAll", query="SELECT c FROM Country c"),
     @NamedQuery(name="Country.findByName",
                 query="SELECT c FROM Country c WHERE c.name = :name"),
})
public class Country {
   ...
}
```

Exercise on JPA

- Implement **findAll** method in ContactDAO
  - Using JPQL
- Implement **findByNamePrefix** method in ContactDAO
  - Using JPQL

  - Like expression
  - `SELECT e FROM Employee e WHERE p.name LIKE 'Mich%‘`
  - `Parameterized: ... WHERE p.name LIKE :name||'%'`

Mapping Tables to Classes by Annotations



Object / Relational Mapping

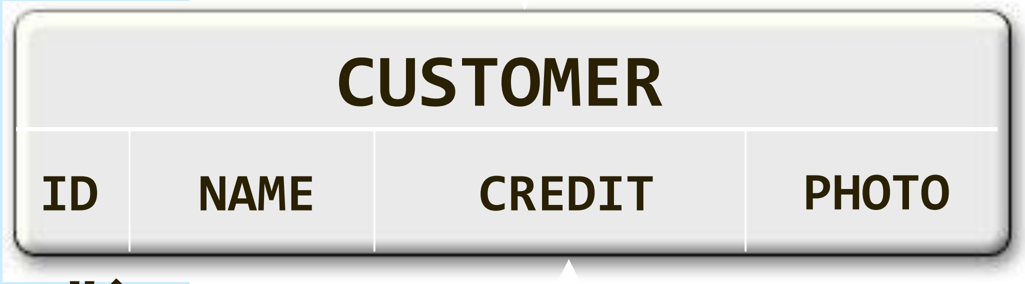
- Map persistent object state to relational database
  - Map relationships between entities
- Metadata may be described as **annotations** or XML (or both)
- Annotations
  - Describe the logical (object) model, e.g. **`@OneToMany`**
  - Describe the physical model (DB tables and columns), e.g. **`@Table`**

Simple Mappings

- Direct mappings of fields to columns
  - **`@Basic`** – optional, indicates simple mapped attribute
  - Can specify **`fetch=EAGER`** / **`fetch=LAZY`**
  - **`LAZY`**: container defers loading until the field or property is accessed (load on demand)
  - **`EAGER`**: the field or relationship loaded when the referencing entity is loaded (pre-load)
- Maps any of the common simple Java types
  - Primitives (**`int`**, **`long`**, **`String`**), wrappers, serializable, etc.
- Used in conjunction with **`@Column`**
  - Can override any of the defaults

Simple Mappings (Annotations)

```
@Entity
public class Customer {
    @Id
    int id;
    @Basic @Column(name="name")
    String name;
    @Basic @Column(name="CREDIT")
    int creditRating;
    @Lob
    Image photo;
}
```



CUSTOMER			
ID	NAME	CREDIT	PHOTO

Simple Mappings (XML)

- It is also possible to describe mappings by XML files
  - But we focus on annotation-based mappings

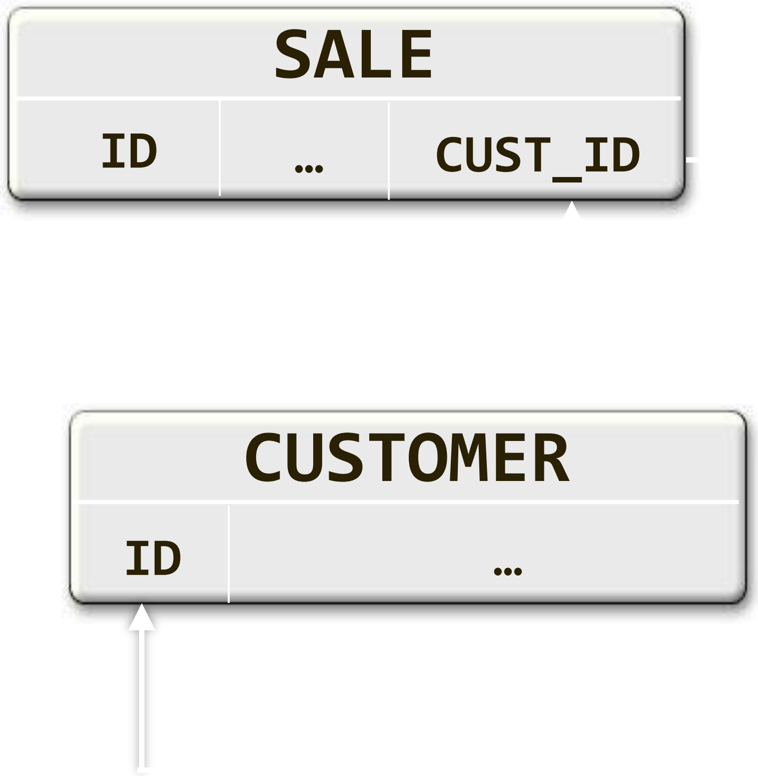
```xml
<entity class="model.Customer">
  <attributes>
    <id name="id" />
    <basic name="name" />
    <basic name="creditRating">
      <column name="CREDIT" />
    </basic>
    <basic name="photo"><lob /></basic>
  </attributes>
</entity>
```

Relationship Mappings

- Cardinality:
  - **`@ManyToOne`**, **`@OneToOne`** – single entity
  - **`@OneToMany`**, **`@ManyToMany`** – collection
- Direction (navigability):
  - **Unidirectional**: we can go from entity A to entity B only.
  - **Bi-directional**: we can go from entity A to entity B & vice-versa.
- Owning and inverse sides
  - Owning side specifies the physical mapping
    - **`@JoinColumn`** to specify foreign key DB column
    - **`@JoinTable`** decouples physical relationship mappings from entity tables

Many-To-One Mapping (Annotations)

```
@Entity
public class Sale {
    @Id
    int id;
    ...
    @ManyToOne
    @JoinColumn(name="CUST_ID")
    Customer cust;
}
```



One-To-Many Mapping (Attributes)

```
@Entity
public class Customer {
   @Id
   int id;
   @OneToMany(mappedBy="cust")
   Set<Sale> sales;
}

@Entity
public class Sale {
   @Id
   int id;
   @ManyToOne
   Customer cust;
}
```

CUSTOMER	
ID	…

SALE		
ID	…	CUST_ID

Many-To-Many Example

```
@Entity
public class Employee {
  @Id
  @Column(name="ID")
  private long id;
  ...
  @ManyToMany
  @JoinTable(
     name="EMP_PROJ",
     joinColumns=@JoinColumn(name="EMP_ID", referencedColumnName="ID"),
     inverseJoinColumns=@JoinColumn(name="PROJ_ID", referencedColumnName="ID"))
  private List<Project> projects;
  .....
}
```

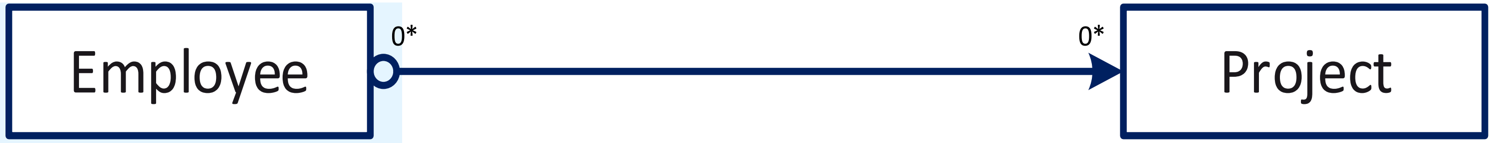
```
@Entity
public class Project {
  @Id
  @Column(name="ID")
  private long id;
  ...
  @ManyToMany(mappedBy="projects")
  private List<Employee> employees;
  ...
}
```

Bi-directional and Unidirectional Relationships

- The relationship may be unidirectional
- If one end does not need to use the relationship
- Example: Unidirectional ManyToMany relationship:

```
@Entity
public class Employee{
    @Id
    private Integer id;

    @JoinTable(name = "PROJECT_EMPLOYEE",
    joinColumns={@JoinColumn(name="EMPLOYEES_ID",
    referencedColumnName="ID")},inverseJoinColumns
    ={@JoinColumn(name="PROJECTS_ID",
    referencedColumnName = "ID")})
    @ManyToMany
    private List<Project> projectList;
    ...
}
```



```
@Entity
public class Project {
    @Id
    private Integer id;
    ...
}
```

Lazy Collection Fetching

- A collection is fetched when the application invokes an operation upon that collection
- This is the default for collections
- Fetch Types:
  - **EAGER**: Defines that data must be eagerly fetched
  - **LAZY**: Defines that data can be lazily fetched (Upon request)
- Remind:
  - Properties (basic fields) can also be lazy loaded (But default is eager)

Lazy Fetching Example

```
@Entity
public class ProductEntity {
    ...
    @ManyToMany(fetch=FetchType.EAGER)
    private List<CategoryEntity> categories;
}
```

```
@ManyToMany(fetch=FetchType.LAZY)
```

Cascade Strategies

Cascading Actions

- Cascading of entity operations to related entities
  - may be defined per relationship
  - Configurable globally in the mapping files
- By **default** no operations are cascaded.
  - Developers must do operations for relations of an entity

Persisting Relationships

```java
@Entity
@Table(name = "PHONE")
public class Phone implements Serializable {
    ...
   @ManyToOne(cascade = CascadeType.ALL)
   @JoinColumn(name = "EMPLOYEE_ID", referencedColumnName = "ID")
   private Employee employee;
}
```

```java
Employee emp = new Employee(); // setters skipped
Phone phone = new Phone(); // setters skipped
phone.setEmployee(emp);
em.persist(phone);
// Consequently, em will also persist the Employee
```

Cascade Types:

- *ALL*
  - Cascade all operations
- *PERSIST*
  - Cascade persist operation
- *MERGE*
  - Cascade merge operation
- *REMOVE*
- *REFRESH*
- *DETACH*

Exercise

- Assume the relationships
  - Order [1-n] Item
  - Item [n-1] Product
  - Product [n-n] Category
- Define Entities
  - Add annotations
  - Set appropriate lazy/eager
- Defines Services:
  - Add product (with a POST request to …/product/save)
    - With no category
  - Add Order
  - Add Item (…/add-item/23/12/3 ➔ Create an item: count=3, product_id=12, order-id=23)

Exercise (cont’d)

- Defines Services:
  - Add-product-item
    - …/category/categoryname/productname
    - (Adds a product in a new category)
  - Does it work with no cascade?
    - No, results in an exception
  - Set appropriate Cascade type

Bean Validation

Introduction to Bean Validation

- Validating input received from the user
  - to maintain data integrity is an important part of application logic.
- The Java API for JavaBean Validation ("**Bean Validation**")
  - provides a facility for validating **objects**
- The Bean Validation model is supported by **constraints** in the form of **annotations**
- Constraints can be **built in** or **user defined**.
  - User-defined constraints are called **custom constraints**
- Integrated with JSF, **JPA**, CDI, JAX-RS, etc.

Built-In Bean Validation Constraints

Constraint	Description	Example
@AssertFalse	The value of the field or property must be false.	@AssertFalse boolean isUnsupported;
@AssertTrue	The value of the field or property must be true.	@AssertTrue boolean isActive;
@DecimalMax	The value of the field or property must be a decimal value lower than or equal to the number in the value element.	@DecimalMax("30.00") BigDecimal discount;
@DecimalMin	The value of the field or property must be a decimal value greater than or equal to the number in the value element.	@DecimalMin("5.00") BigDecimal discount;

Built-In Bean Validation Constraints

Constraint	Description	Example
@Digits	The value of the field or property must be a number within a specified range. The integer element specifies the maximum integral digits for the number, and the fraction element specifies the maximum fractional digits for the number.	@Digits(integer=6 ,fraction=2) BigDecimal price;
@Future	The value of the field or property must be a date in the future.	@Future Date eventDate;
@Max	The value of the field or property must be an integer value lower than or equal to the number in the value element.	@Max(10) int quantity;
@Min	The value of the field or property must be an integer value greater than or equal to the number in the value element.	@Min(5) int quantity;

Built-In Bean Validation Constraints

Constraint	Description	Example
@NotNull	The value of the field or property must not be null.	@NotNull String username;
@Null	The value of the field or property must be null.	@Null String unusedString;
@Past	The value of the field or property must be a date in the past.	@Past Date birthday;
@Pattern	The value of the field or property must match the regular expression defined in the regexp element.	@Pattern(regexp="\\(\\d{3}\\)\\d{3}-\\d{4}") String phoneNumber;

Built-In Bean Validation Constraints

Constraint	Description	Example
@Size	The size of the field or property is evaluated and must match the specified boundaries. If the field or property is a String, the size of the string is evaluated. If the field or property is a Collection, the size of the Collection is evaluated. If the field or property is a Map, the size of the Map is evaluated. If the field or property is an array, the size of the array is evaluated. Use one of the optional max or min elements to specify the boundaries.	@Size(min=2, max=240) String briefMessage;

Field Level Validation

```
public class Name {
    @NotNull
    @Size(min=1, max=16)
    private String firstname;
    @NotNull
    @Size(min=1, max=16)
    private String lastname;
}
```

Validating Constructors and Methods

```
public class Employee {
    ...
    public Employee(@NotNull String name) { ... }

    public void setSalary(
            @NotNull
            @Digits(integer = 6, fraction = 2) BigDecimal salary,
            @NotNull
            String currencyType) {

    }
    ...
}
```

Summary

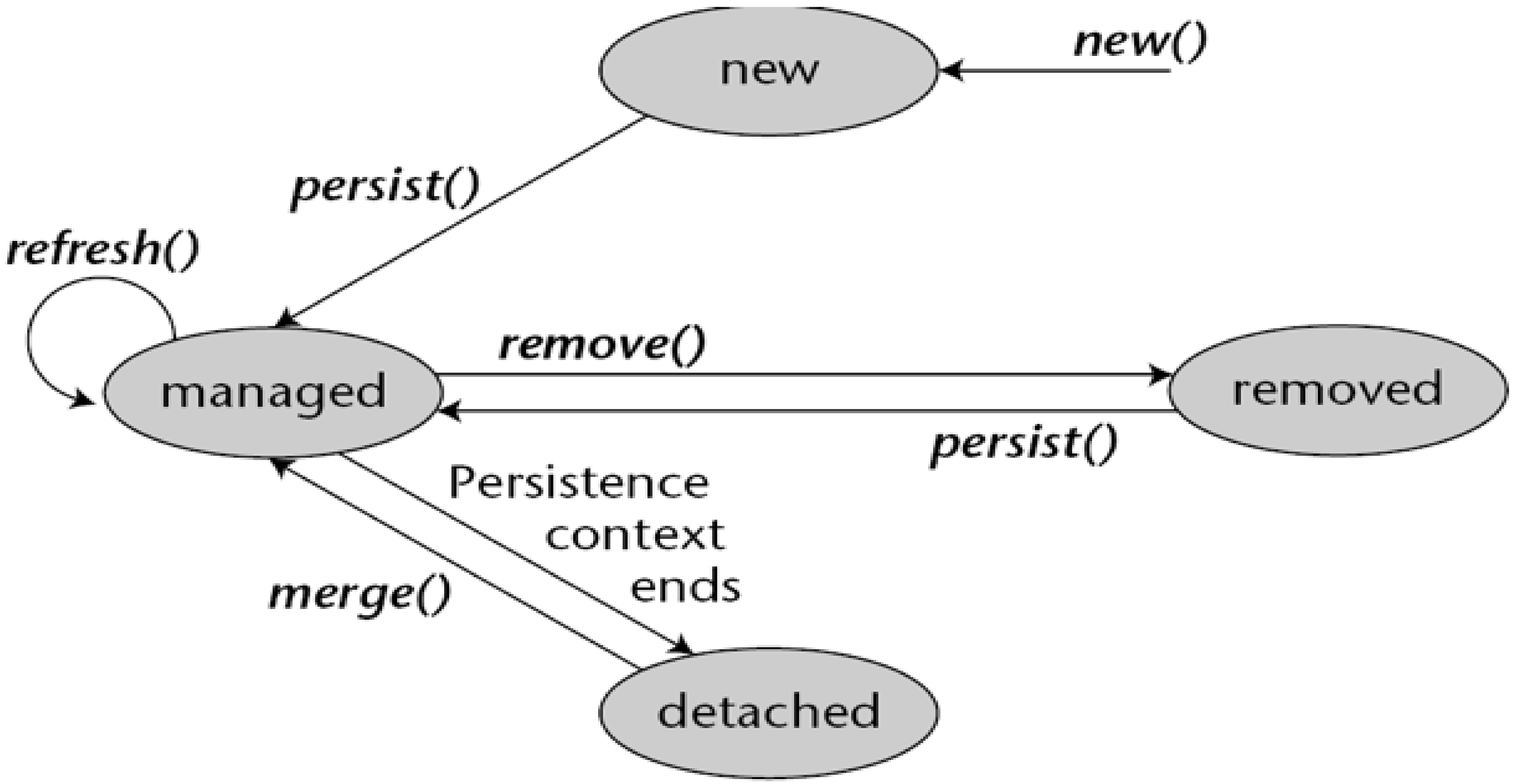
- Declarative constraint management across application layers
- Constraint
  - Restriction on a bean, field, property, method parameter, return value
  - Not null, between 10 and 45, valid email, etc
  - @Max, @Min, @Size, @NotNull, @Pattern, @Future, @Past
  - Custom constraints
  - Evaluated automatically
- Integrated with JSF, JPA, CDI, JAX-RS

Exercise

- Set the following validation rules for contact entity
  - **Age** is a Required field
  - **Age** should be in the range [0,200]
  - **Name** contains no digits

Managed vs Detached

Life cycle of Entity



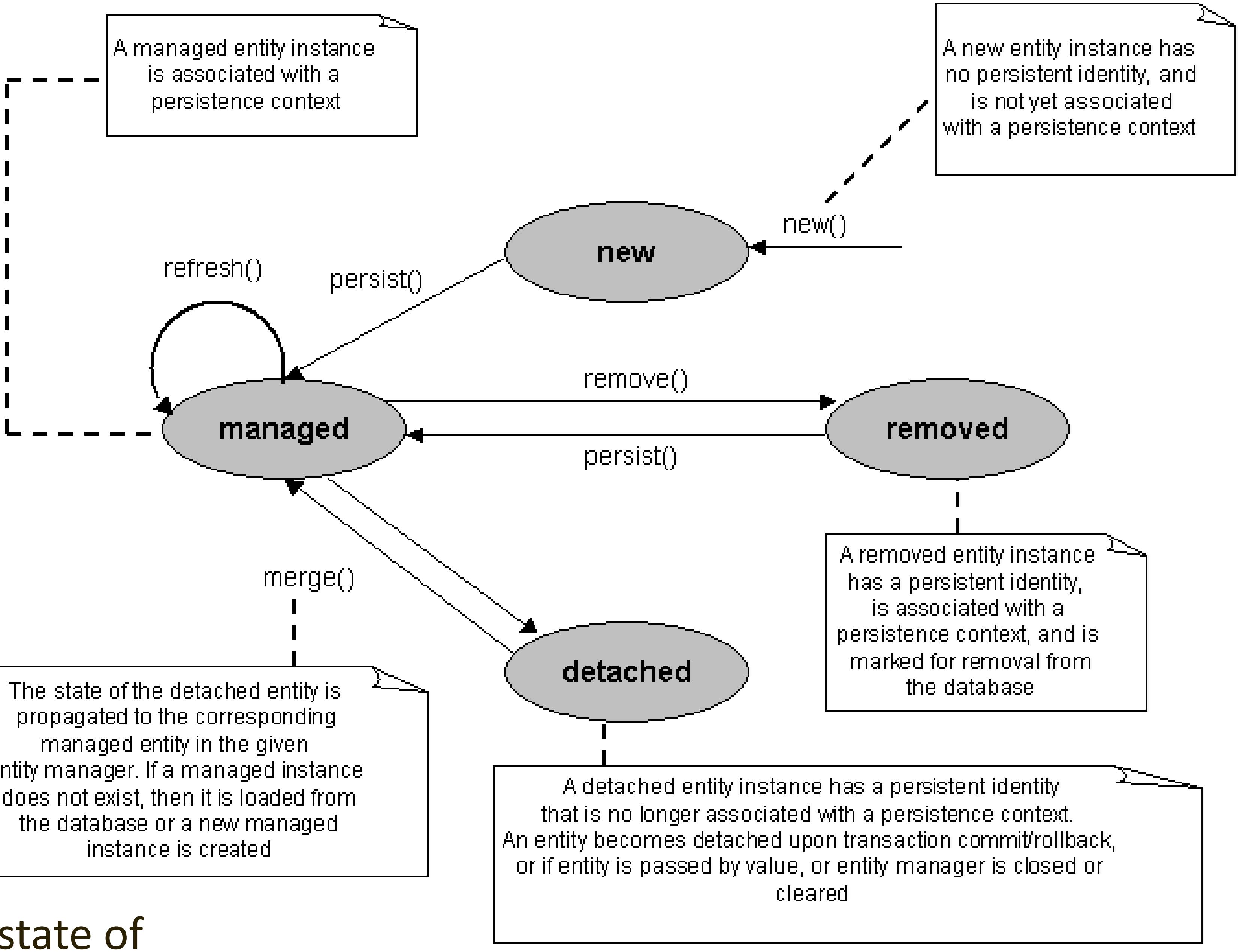
Managed Objects

- A **managed** object is the one read in the current persistence context
- The **PC** will track changes to every managed object
- If the **same object** is read again in the same persistence context,
  or traversed through another managed object's relationship,
  the same identical (==) object will be returned.
- Calling **persist**() on a new object will also make it become **managed**
- Calling **merge**() on a **detached** object will return a managed copy
- A **removed** object will no longer be managed
  - after a **flush**() or **commit**().

Detached Objects

- A **detached** object is one that is **not managed** in the current PC
- This could be
  - an object read through a different persistence context
  - or an object that was cloned or serialized
- A **new object** is also considered **detached** until persist() is called on it
- An object that was **removed** and **flushed** or **committed**, will become detached
- A **managed** object should only ever **reference** other managed objects
- and a **detached** object should only reference other detached objects
- Incorrectly relating managed and detached objects :
  - one of the most common problems users run into in JPA

Entities Lifecycl


A managed entity instance is associated with a persistence context
A new entity instance has no persistent identity, and is not yet associated with a persistence context
new()
new
refresh()
persist()
managed
remove()
persist()
removed
A removed entity instance has a persistent identity, is associated with a persistence context, and is marked for removal from the database
merge()
detached
The state of the detached entity is propagated to the corresponding managed entity in the given entity manager. If a managed instance does not exist, then it is loaded from the database or a new managed instance is created
A detached entity instance has a persistent identity that is no longer associated with a persistence context. An entity becomes detached upon transaction commit/rollback, or if entity is passed by value, or entity manager is closed or cleared

`merge()` – synchronize the state of detached entity with the PC

merge()

- Merges the state of detached entity into a managed copy of the detached entity
  - Returned entity has a different Java identity than the detached entity

```
public Customer storeUpdatedCustomer(Customer cust) {
  return entityManager.merge(cust);
}
```

Life cycle of Entity

- **New**: The bean exists in the memory but is not mapped to DB yet. It is not yet mapped to persistence context (via entity manager)
- **Managed**: The bean is mapped to the DB. The changes effect the data in DB.
- **Detached**: The bean is not mapped to the persistence context anymore.
- **Removed**: The bean is still mapped to the persistence context and the DB, but it is programmed to be deleted.

Life cycle of Entity

- **remove()**: To delete the data
- **set(), get()**: If the bean is in managed state, these methods (of the entity bean) are used to access or modify the data.
- **persist()**: To create the data. The bean goes to managed state.
- **merge()**: To take a bean from detached state to a managed state.

When will the JPA Entities become Detached?

- When the transaction (in transaction-scoped persistence context) commits, entities managed by the persistence context become detached.
- If an application-managed persistence context is closed, all managed entities become detached.
- Using clear method
- using detach method
- rollback

Criteria APIs

Criteria API

- The Java Persistence Criteria API
- Defines dynamic queries
  - through the construction of object-based query definition objects
  - rather than use of the string-based approach of JPQL
- Criteria API allows dynamic queries to be built programmatically
  - offering better integration with the Java language
  - than a string-based approach
- The Criteria API was added in JPA 2.0
- The Criteria API delete and update support was added in JPA 2.1

Criteria API

- **JPA Criteria API** provides an alternative way for defining JPA queries
  - **type-safe** way to express a query
- Mainly useful for building dynamic queries
  - whose exact structure is only known at runtime
- Allows developers to **find**, **modify**, and **delete** persistent entities
  - Similar to **JPQL**
- The **Criteria API** and **JPQL** are closely related
  - designed to allow similar operations in their queries.

The First Example

- Consider this simple Criteria query:

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<User> cq = cb.createQuery(User.class);
Root<User> user = cq.from(User.class);
cq.select(user);
TypedQuery<User> q = em.createQuery(cq);
List<User> allUsers = q.getResultList();
```

- returns all instances of the User entity in the data source
- The equivalent JPQL query is: `SELECT u FROM User u`

JPQL vs Criteria API, an Example:

- With JPQL:

```
List<User> users =
entityManager.createQuery("select e from User e").getResultList();
```

- With criteria API:

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<User> cq = cb.createQuery(User.class);
Root<User> user = cq.from(User.class);
cq.select(user);
TypedQuery<User> q = em.createQuery(cq);
List<User> allUsers = q.getResultList();
```

- **JPQL** may look simpler, but **criteria API** has its own advantages

CriteriaBuilder

- A **CriteriaBuilder** is obtained from an EntityManager
  - `CriteriaBuilder cb = entityManager.getCriteriaBuilder();`
- CriteriaBuilder is used to construct **CriteriaQuery** objects and their expressions
- CriteriaBuilder API:
  - **createQuery**() - Creates a CriteriaQuery
    - **createQuery(Class)**: using generics to avoid casting the result class
  - createTupleQuery() - Creates a CriteriaQuery that returns map like Tuple objects
    - instead of object arrays for multiselect queries
  - createCriteriaDelete(Class) and createCriteriaUpdate(Class)
    - Creates a CriteriaDelete or CriteriaUpdate to delete/update a batch of objects (JPA 2.1)

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<User> cq = cb.createQuery(User.class);
```

CriteriaQuery API

- **(CriteriaQuery** defines a database **select** query**)**
- **from(Class)** : Defines an element in the query's **from clause**
  - At least one from element is required for the query to be valid
  - Returns a **Root**<EntityClass> instance
- **select(Selection):** Defines the query's **select clause**
  - If not set, the first root will be selected by default
- **where(Expression), where(Predicate...) :** Defines the where clause
  - By default all instances of the class are selected.

CriteriaQuery Example

```
CriteriaBuilder cb = ...;
CriteriaQuery<User> cq = cb.createQuery(User.class);
Root<User> user = cq.from(User.class);
cq.select(user);
criteriaQuery.where(cb.greaterThan(user.get("age"), 18));
```

```
CriteriaQuery<CategoryEntity> cq = cb.createQuery(CategoryEntity.class);
Root<CategoryEntity> cat = cq.from(CategoryEntity.class);
cq.select(cat);
Predicate pred = cb.like(cat.get("name"), "Laptop%");
cq.where(pred);
```

```
Predicate like = cb.like(cat.get("name"), "Lap%");
Predicate idgt = cb.greaterThan(cat.get("id"), 5);
Predicate and = cb.and(like, idgt);
cq.where(and);
```

Other CriteriaQuery API

- **distinct(boolean)**: filter duplicate results (defaults to false)
- **orderBy**(Order...), **orderBy**(List<Order>): the order clause
  - By default the results are not ordered
- **groupBy**(Expression...), groupBy(List<Expression>)
  - Defines the query's group by clause
  - By default the results are not grouped.
- **having**(Expression), **having**(Predicate...): the **having** clause
  - Having allows grouped results to be filtered.

Multiple Selects in One CriteriaQuery

- Calls to the **from()** method are **additive**
- Each call adds another **root** to the **query**
- resulting in a **Cartesian product**
  - if no further constraints are applied in the WHERE clause

```
CriteriaQuery<Department> c = cb.createQuery(Department.class);
Root<Department> dept = c.from(Department.class);
Root<Employee> emp = c.from(Employee.class);
c.select(dept)
  .distinct(true)
  .where(cb.equal(dept, emp.get("department")));
```

```
SELECT DISTINCT d FROM Department d, Employee e WHERE d = e.department
```

Path Expressions

- The **path expression** is the key to the power and flexibility of the **JPQL** language
- And it is likewise a central piece of the **Criteria API**
- **Roots** are actually just a special type of path expression
- The **get()** method is equivalent to the **dot** operator used in JPQL path expressions

```
CriteriaQuery<Employee> c = cb.createQuery(Employee.class);
Root<Employee> emp = c.from(Employee.class);
c.select(emp)
 .where(cb.equal(emp.get("address").get("city"), "Yazd"));
```

```
SELECT e FROM Employee e WHERE e.address.city = 'Yazd'
```

Basic Structure

JPQL Clause	Criteria API Method
SELECT	select()
FROM	from()
WHERE	where()
ORDER BY	orderBy()
GROUP BY	groupBy()
HAVING	having()

Building Expressions

JPQL Operator	CriteriaBuilder Method
AND	and()
OR	or()
NOT	not()
=	equal()
<>	notEqual()
>	gt()
>=	ge()
<	lt()
<=	le()

Building Expressions

JPQL Operator	CriteriaBuilder Method
EXISTS	exist()
NOT EXISTS	not(exist())
IS EMPTY	isEmpty()
IS NOT EMPTY	isNotEmpty()
LIKE	lik()
NOT LIKE	notLike()
BETWEEN	between()
IN	in()
NOT IN	not(in())
IS NULL	isNull()
IS NOT NULL	isNotNull()

Building Expressions

JPQL Operator	CriteriaBuilder Method
AVG	avg()
MIN	min()
MAX	max()
COUNT	count()
COUNT DISTINCT	countDistinct()
LOWER	lower()
UPPER	upper()
CONCAT	concat()
CURRENT_TIME	currentTime()

In Expressions Example

```java
CriteriaQuery<Employee> c = cb.createQuery(Employee.class);
Root<Employee> emp = c.from(Employee.class);
c.select(emp)
  .where(emp.get("address")
    .get("state").in("Yazd","Qom"));
```

```sql
SELECT e FROM Employee e
WHERE e.address.state IN ('Yazd', 'Qom')
```

Join

```
SELECT p FROM Teacher t JOIN t.phones p
```

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Phone> query = cb.createQuery(Phone.class);
Root<Teacher> teacher = query.from(Teacher.class);
Join<Teacher, Phone> phones = teacher.join("phones");
query.select(phones).where(cb.equal(teacher.get("firstName"), "Sadegh"));
```

- teacher.join("phones") creates a join
  - on the collection field named **phones**
- Variable **phones** is similar to the p variable
  - that represents join in JPQL

Multiselect

```
CriteriaQuery criteriaQuery = criteriaBuilder.createQuery();
Root employee = criteriaQuery.from(Employee.class);
criteriaQuery.multiselect(employee.get("firstName"), employee.get("lastName"));
Query query = entityManager.createQuery(criteriaQuery);
List<Object[]> result = query.getResultList();
```

Parameters in Criteria API

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery cq = cb.createQuery();
Root<Employee> employee = cq.from(Employee.class);
criteriaQuery.where(
        cb.equal(employee.get("firstName"),
                cb.parameter(String.class, "first")),
        cb.equal(employee.get("lastName"),
                cb.parameter(String.class, "last"))
    );
Query query = entityManager.createQuery(cq);
query.setParameter("first", "Bob");
query.setParameter("last", "Smith");
List<Employee> = query.getResultList();
```

The ORDER BY Clause

- The **orderBy()** method
- Accepts one or more Order objects
  - created by the **asc()** and **desc()** methods of **CriteriaBuilder**

```
CriteriaQuery<Employee> c = cb.createQuery(Employee.class);
Root<Employee> emp = c.from(Employee.class);
c.orderBy(cb.desc(emp.get("name")), cb.asc(emp.get("age")));
```

```
SELECT e ROM Employee e
ORDER BY d.name DESC, d.age ASC
```

The GROUP BY and HAVING Clauses

```
CriteriaQuery<Object[]> c = cb.createQuery(Object[].class);
Root<Employee> emp = c.from(Employee.class);
Join<Employee,Project> project = emp.join("projects");
c.multiselect(emp, cb.count(project))
 .groupBy(emp)
 .having(cb.ge(cb.count(project),2));
```

```
SELECT e, COUNT(p)
FROM Employee e JOIN e.projects p
GROUP BY e HAVING COUNT(p) >= 2
```

Bulk Update and Delete

```
CriteriaUpdate<Employee> q = cb.createCriteriaUpdate(Employee.class);
Root<Employee> emp = q.from(Employee.class);
q.set(emp.get("salary"), cb.sum(emp.get("salary"), 5000))
  .where(cb.lt(emp.get("salary"), 1000000));
```

```
UPDATE Employee e SET e.salary = e.salary + 5000  WHERE e.salary<1000000
```

```
CriteriaDelete<Employee> q = cb.createCriteriaDelete(Employee.class);
Root<Employee> emp = q.from(Employee.class);
q.where(cb.isNull(emp.get("dept")));
```

```
DELETE FROM Employee e WHERE e.department IS NULL
```

JPQL vs Criteria API

- JPQL queries are defined as **strings**, similarly to SQL
  - JPA criteria queries are defined by **instantiation** of Java objects
- The criteria API: **errors** can be detected earlier
  - during **compilation** rather than at **runtime**
- For many developers string based JPQL queries are **easier** to use and understand
  - JPQL queries are similar to **SQL** queries
- For **simple static queries**: string based JPQL queries may be preferred
- For **dynamic queries** that are built at runtime: the criteria API may be preferred
  - E.g., building a dynamic query based on fields that a user fills at runtime
  - in a form that contains many optional fields
  - cleaner when using the JPA criteria API because it eliminates many **string concatenations**
- JPQL and JPA criteria based queries are equivalent in **power**

Exercise

- First, fill some items and products into the DB
- Defines Services:
  - Select product-name of items of an order with quantity>param
  - .../min-items/order-id/min-quantity

Appropriate inheritance strategies

1- Mapped Superclass Strategy

- It maps each concrete class to its own table
- A mapped superclass is not an entity, and there is no table for it

```
@MappedSuperclass
public abstract class Publication {
 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 @Column(name = "id")
 protected Long id;
 ...
}
```

```
@Entity(name = “Book”)
public class Book extends Publication {
  @Column
  private int pages;
  …
}
```

2- Table per Class Strategy

- Similar to the mapped superclass strategy
- But the superclass is now also an entity
- Each of the concrete classes gets still mapped to its own table
- This mapping allows you to use polymorphic queries
    - and to define relationships to the superclass

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Publication {
 @Id
 @GeneratedValue(strategy = GenerationType.AUTO)
 @Column(name = “id”)
 protected Long id;
 ...
```

3- Single Table Strategy

- It maps all entities of the inheritance structure to the same table

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = “Publication_Type”)
public abstract class Publication {...}
```

4- Joined Strategy

- It maps each class of the inheritance hierarchy to its own table
  - Even the parent class
- The tables of the subclasses are much smaller
  - They hold only the columns specific for the mapped entity class
  - and a **primary key** with the same value as the record in the table of the superclass.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Publication {
```

Choosing a Strategy

- 1- Mapped Superclass Strategy
- 2- Table per Class Strategy
- 3- Single Table Strategy
- 4- Joined Strategy

The End