

BBM490

ENTERPRISE WEB ARCHITECTURE

Mert ÇALIŞKAN

Hacettepe University Spring Semester '15



Week 7

JPA

We'll cover Java Persistence API to highlight the features that it brings to the persistency world.



Multiple Tables with JDBC

- We have seen JDBC and did insert/update on User table. We iterated over the result set and took care of the mapping of the User data coming from the DB.
- But what about if we have multiple tables and having a bloated Result Set?





Multiple Tables with JDBC

- First we need to prepare an SQL with a join like,

```
String SQL = "select "+  
    "u.userid,"+  
    "u.firstname,"+  
    "u.lastname,"+  
    "u.password,"+  
    "u.type,"+  
    "a.id,"+  
    "a.addressline1,"+  
    "a.addressline2,"+  
    "a.county,"+  
    "a.city,"+  
    "a.country,"+  
    "a.zip"+  
    " from user u, address a"+  
    " where a.userid = u.userid";  
ResultSet rs = stmt.executeQuery(SQL);
```




Multiple Tables with JDBC

- Then we need to iterate over the result set and extract the data for users and their addresses like,

```
while (rs.next()) {  
    String userid = rs.getString("userid");  
    String firstname = rs.getString("firstname");  
    String lastname = rs.getString("lastname");  
    String password = rs.getString("password");  
    int type = rs.getInt("type");
```

```
    int addressId = rs.getInt("id");  
    String addressline1 = rs.getString("addressline1");  
    String addressline2 = rs.getString("addressline2");  
    String county = rs.getString("county");  
    String city = rs.getString("city");  
    String country = rs.getString("country");  
    int zip = rs.getInt("zip");
```

```
    .....
```



Multiple Tables with JDBC

- And store the user and address data in an structure like,

```
Map<String, User> users = new HashMap<String, User>();
```

```
Set<Address> addresses = new HashSet<Address>();
```

```
User user = new User(userid, firstname, lastname, password, type);  
users.put(userid, user);
```

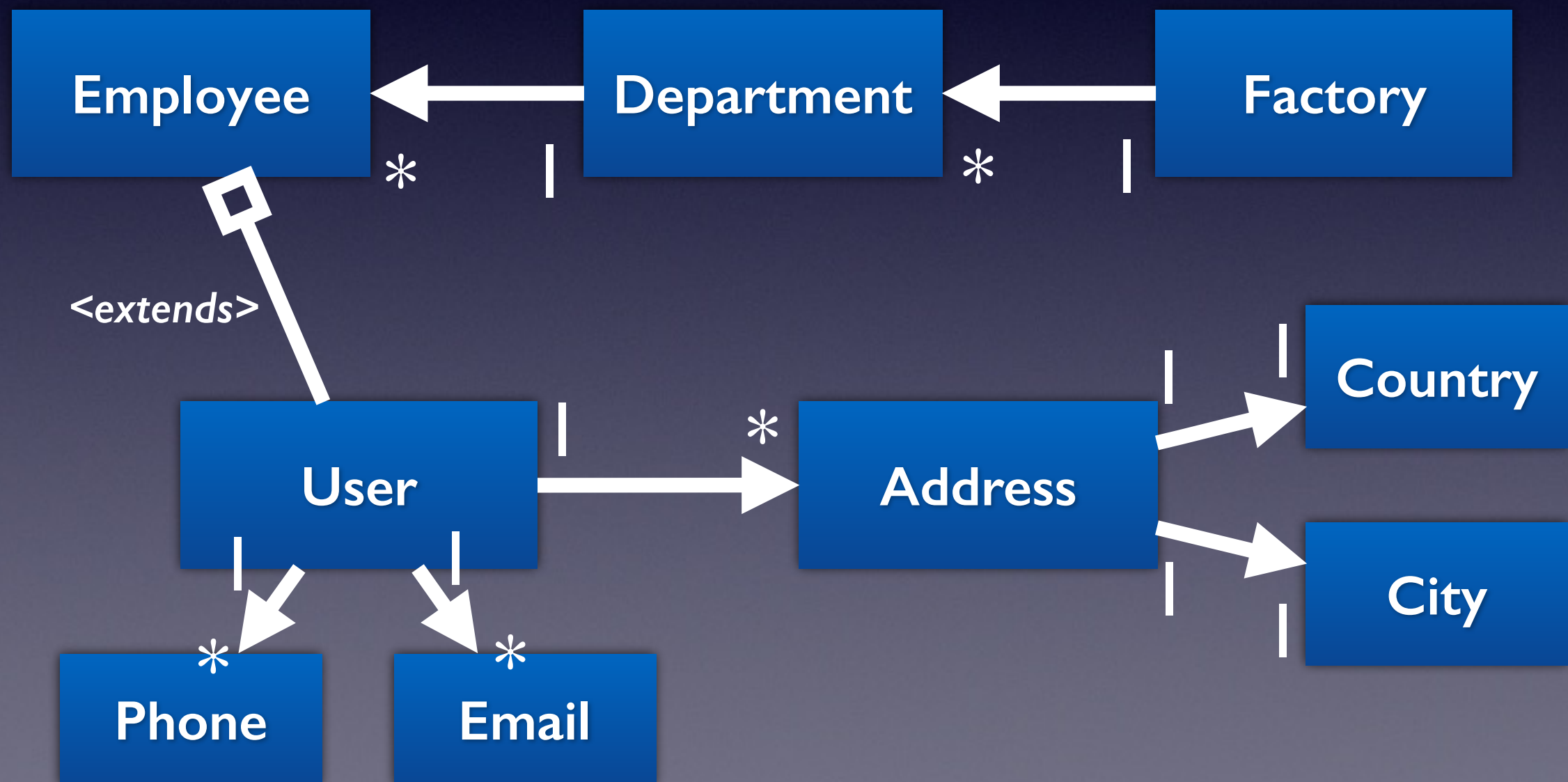
```
Address address = new Address(addressId, addressline1,  
    addressline2, county, city, country, zip, users.get(userid));
```

```
addresses.add(address);
```



Multiple Tables with JDBC

- How about if we have an object graph like this? will it be easy to construct its data from DB via ResultSets? How to do that?





ORM to the rescue

- While doing application development, we are thinking more in object oriented. We define classes and their relations in between.
- And when we want to persist the data into a database for instance we are more thinking relational centric. We create tables and define relations in between by mapping Primary keys along with their Foreign keys.
- So we need a mapping to relate our thoughts from object oriented world to the relational database world.



What's ORM?

- Stands for Object Relational Mapping.
- It is an approach for handling the persistent data on the application side and it tries to isolate the database layer from the application.
- Object world: consists of objects associated with each other (composition) or with hierarchy (inheritance)
- Relational world: consists of tables and columns inside those tables and primary along with foreign key for handling relations.

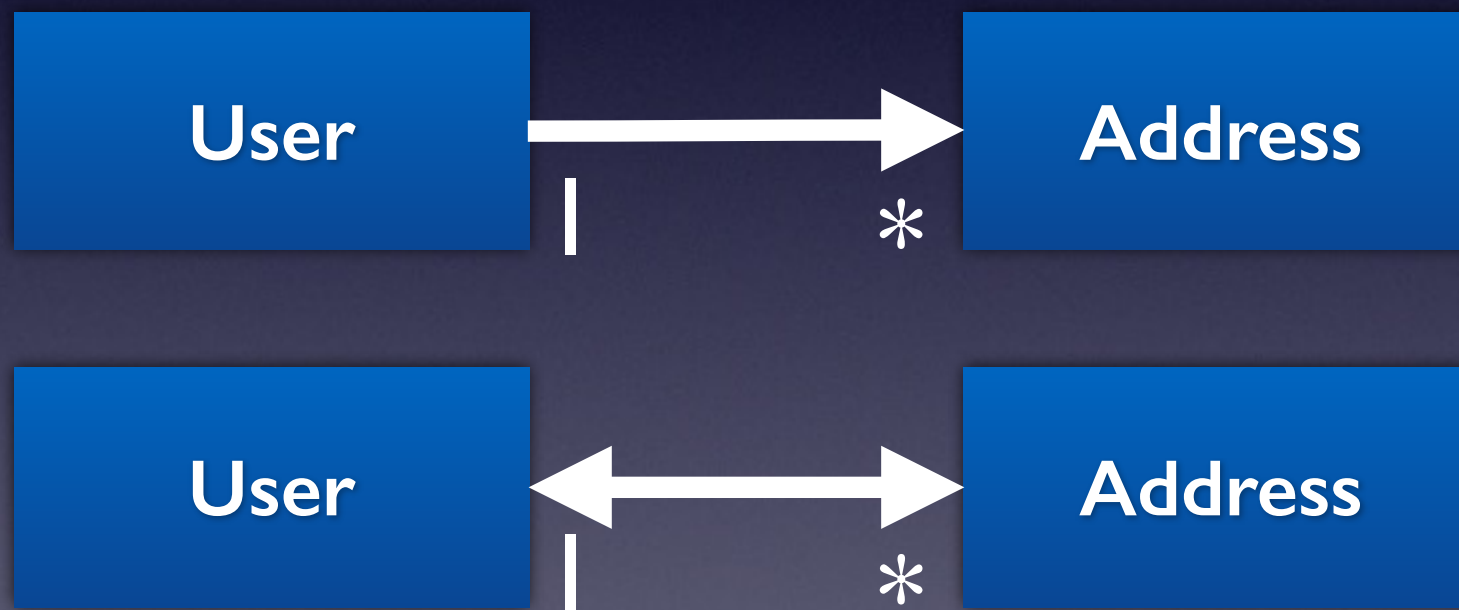


Question here is:
Object World maps to the
Relational World?



Paradigm Mismatch

- In object world, objects have associations with other objects and this associations also contains directional information like, unidirectional or bidirectional.



- In relational world we define an association with PK and FK but they don't provide any information about direction.



Paradigm Mismatch

- The desire of granularity in OO world.

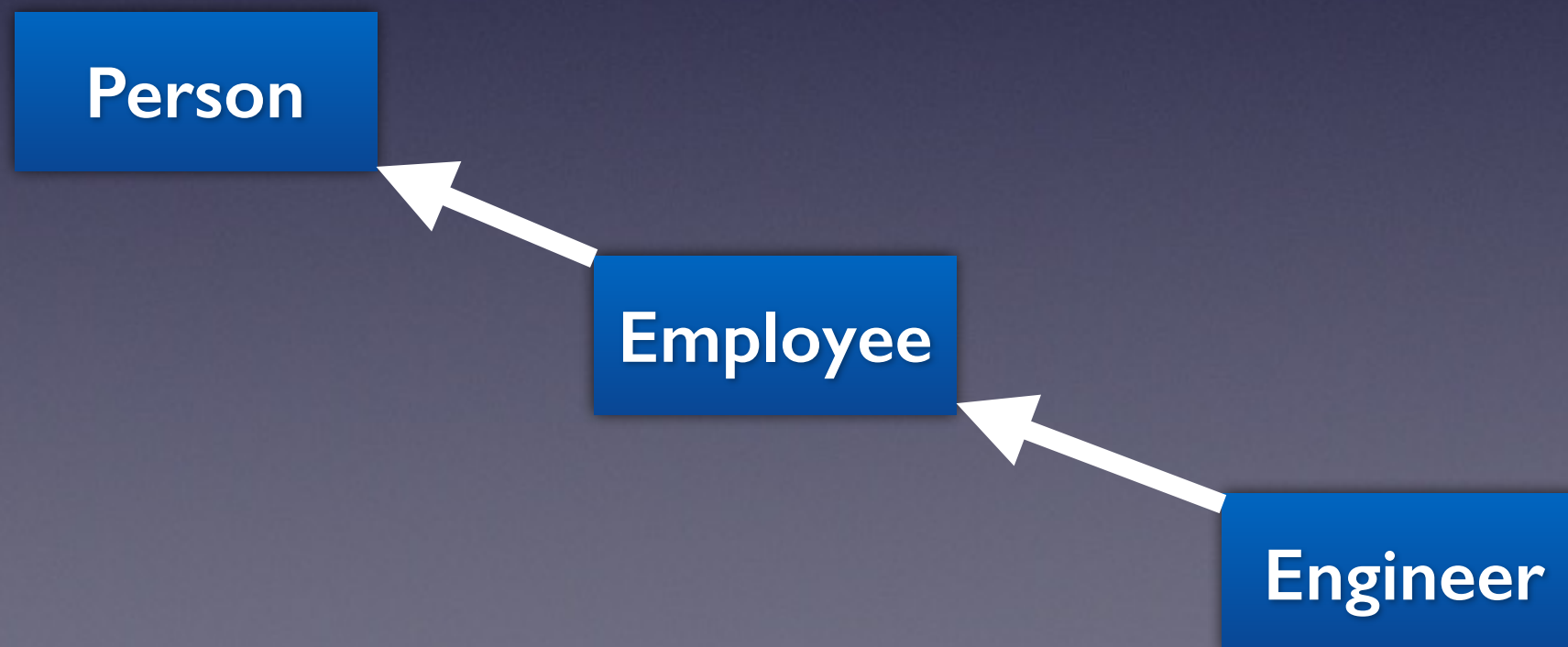


- Here a User object contains a list of addresses for a user instance. We have a well defined object graph by defining classes and if we map these objects to database tables we'll come up with multiple tables and try to join them while fetching data. DB people tend to combine the related data into one table for performance concerns.



Paradigm Mismatch

- Inheritance and polymorphism in OO world.
- We can define abstract types and define relationships with different objects at runtime.
- Inheritance is not supported in relational world as it's done in the object world.





Paradigm Mismatch

- Polymorphism that is used in object world is not supported in the relational world.
- Traversal of object graph in object world is node by node. (`factory.department[0].employee[2]`)
- In relational world we tend to reduce the number of SQLs with joins (`select * from factory, department, employee where ...`)
- These 2 approaches also contradicts with each other.



What ORM offers

- Metadata mapping between object model and relational model
- CRUD API for operations that need to be performed over objects.
- An object query language
- Different fetch strategies, and object network traversal facilities to improve memory usage and performance of data fetch times



What's JPA?

- JPA stands for Java Persistence API
- JPA itself is a specification, it provides an interface to standardise ORM features and functionalities.
- There are various JPA implementations like Hibernate, EclipseLink, DataNucleus, OpenJPA and etc.
- Since it brings abstraction with features like, automatic metadata discovery, simple configuration and standard data access approach it becomes easy for switching between JPA implementations.



object model relational model

- So we are doing the mapping
 - between classes and tables,
 - between attributes of a class and columns in a table
 - between object associations and foreign keys
 - between Java types and SQL types.



What's an Entity?

- Entity is an object that maps to a table in a relational database.
- Entity is mapped to a database with metadata definition, which either could be via
 - XML
 - Annotations
- There is usually one to one mapping between entity class and its corresponding table.
- But an entity can be mapped to multiple tables as well.



Anatomy of an Entity

- `@Entity` annotation defines that User class is a persistent type, which has a corresponding table.
- `@Table` annotation specifies the name of the table.

```
@Entity
@Table(name="users")
public class User {
    @Id
    @GeneratedValue
    private Long id;
}
```



Anatomy of an Entity

- If `@Table` is not used, the name of the table will be same with the class name.
- `@Id` annotation marks the primary key attribute. The name of the column will be same as the name of the property. `@Column` could be use to define a different name for the primary column name.
- Primary keys can be composed with one or more column and the data could be provided by the application, which are meaningful for business. These are called natural primary keys.



Anatomy of an Entity

- But having single primary key column and populating its data with unrelated content is more popular (like sequencing numbers or globally unique identifiers). Such primary keys called surrogate (vekil in TR) or synthetic primary keys. Your application will not be responsible for producing its value.
- `@GeneratedValue` annotation tells JPA that application won't deal with assigning value, and JPA vendor should handle it instead.



Mapping attributes to columns

- By default any attribute inside an object treated as persistent.
- If you want to be an attribute not to be persisted it should be marked with `@Transient`.
- JPA annotations could either be defined on fields (field level access) or on getter methods (getter level access). With first approach JPA won't need get/set methods, all of the work will be done with Reflection API.
- JPA defines the access strategy by looking at `@Id`



Entity Manager

- JPA provides an API to perform CRUD operations.
We can use the API for,
 - selecting an entity with its type (User.class) and its primary key (10 for instance)
 - inserting a new entity into the table (User table)
 - updating its changed attributes (like lastname)
 - delete an entity from table.
- Those operations are provided by JPA EntityManager.

persistence.xml



- It's the main entry point for JPA configuration.
- The file resides under META-INF/persistence.xml
- There can be several JPA configurations in an application, each configuration is called persistence unit.
- persistence unit knows how to connect the DB. One option is with DriverManager (like we did in Spring configuration).

persistence.xml



- ORM tools provide abstraction on top of DB and cause of this they can work with different DB and generate SQLs according to the DBs. They use dialects to achieve this. persistence unit contains the dialect definition. Hibernate automatically finds out the dialect through the JDBC driver that is used.
- You can also tell Hibernate to update the schema by synchronising it with the metadata by setting *hibernate.hbm2ddl.auto* property.

persistence.xml



```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
    xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://
java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="bbm490-jpa" transaction-type="RESOURCE_LOCAL">
    <properties>
      <property name="hibernate.connection.driver_class"
        value="oracle.jdbc.driver.OracleDriver"/>
      <property name="hibernate.connection.url"
        value="jdbc:oracle:thin:@localhost:1521/orcl"/>
      <property name="hibernate.connection.username" value="BUWORKSHOP"/>
      <property name="hibernate.connection.password" value="BUWORKSHOP"/>
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.Oracle10gDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="create"/>
    </properties>
  </persistence-unit>
</persistence>
```

Associating between Objects



- Association between objects can be defined with Multiplicity (number of instances for each side of association) and Directionality (which direction is that association is navigable) perspectives.
 - One-to-one (1:1)
 - Many-to-one (M:1)
 - One-to-many (1:M)
 - Many-to-many (N:M)

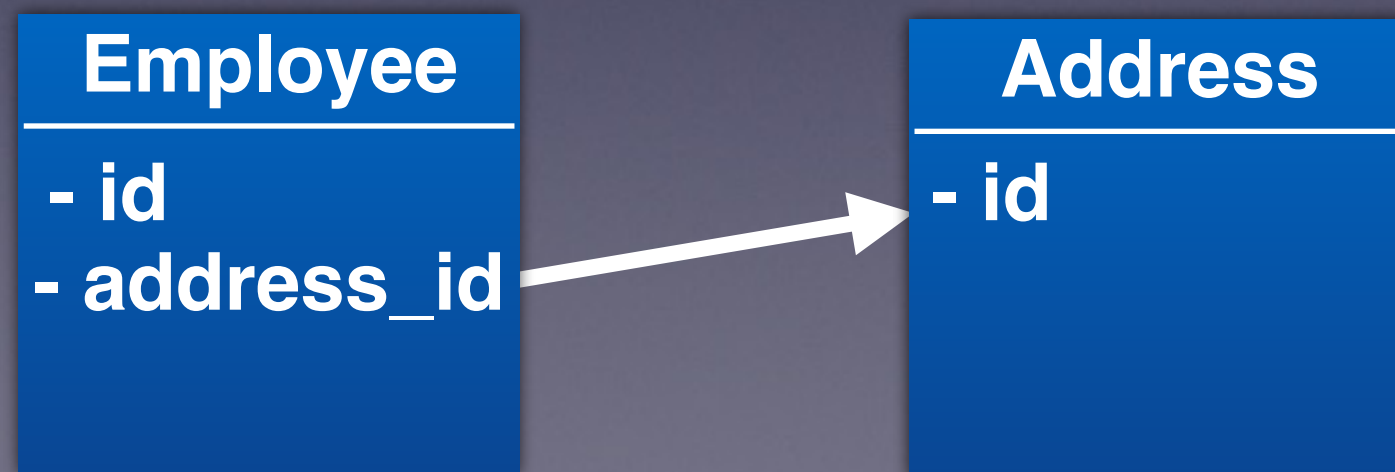


1:1

- one to one means 2 objects are associated with each other. An example could be like a employee has an address.

```
@Entity
public class Employee {
    @OneToOne
    private Address address;
}
```

- Tables generated will be as follows.



Let's create a Maven Project



- We will do a one-to-one mapping with a Maven project.
- New dependencies on horizon:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>4.3.9.Final</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>4.3.9.Final</version>
</dependency>
```

src:jpa-one-to-one



@JoinColumn

- If you don't provide any @JoinColumn as in the example given, the Employee table will contain a column named "address_id" by default. You can alter the name of the column with:

```
public class Employee {  
    @OneToOne  
    @JoinColumn(name="my_sweet_address_id")  
    private Address address;  
}
```

- @JoinColumn specifies the FK-PK relation between Employee and Address tables. It defines the column in the Employee table.



Under the hood...

- Employee and Address entities are defined as persistent using @Entity annotation.
- Their id defined with @Id attribute, one on the property and one on the getter method.
- @GeneratedValue is also defined, possible strategies for it are: identity, autoincrement, sequence, uuid. Oracle is using sequence. MySQL is using autoincrement.
- There is a one to one relation defined within a Employee and his Address.

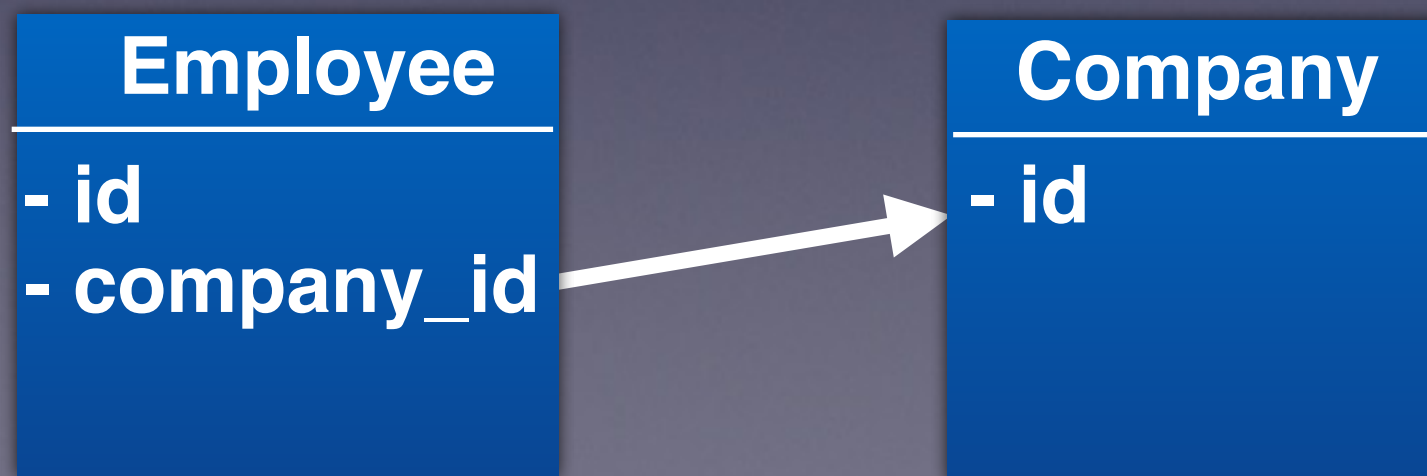


M:1

- many to one means there are many objects referring to a same instance. An example could be like there can be several employees working for a company.

```
@Entity
public class Employee {
    @ManyToOne
    private Company company;
}
```

- Tables generated will be as follows.



src:jpa-many-to-one

M:1



- `@JoinColumn` specifies the FK-PK relation between tables. *optional* states that employee instance cannot exist without an associated Company.

```
@Entity
public class Employee {
    @ManyToOne(optional=false)
    @JoinColumn(name="company_id")
    private Company company;
}
```



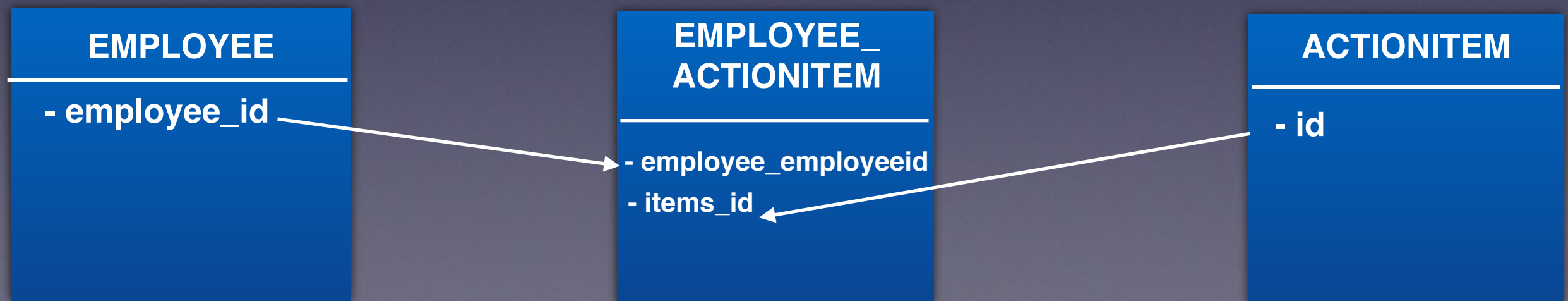
I:M

- one-to-many means an instance can have relation with many objects of the same type. An example could be: employee has a list of actions items.

```
@Entity
public class Employee {

    @OneToMany
    private Set<ActionItem> items = new HashSet<ActionItem>();
}
```

- Tables generated will be as follows.



src:jpa-one-to-many



I:M

- `@JoinColumn` specifies the FK-PK relation between tables. A type from Java Collection API (here `java.util.Set`) is used to define variable type.

```
@Entity
public class Employee {

    @OneToMany
    @JoinColumn(name="my_sweet_item_id")
    private Set<ActionItem> items = new HashSet<ActionItem>();
}
```

- This eliminates the need of extra table.

1:M w/ mappedBy



- Let's have a domain with Employee has a list of ActionItems as given below.



- So if we declare mappedBy on Employee like,

```
public class Employee {
    @OneToMany(mappedBy = "employee")
    private Set<ActionItem> items = new HashSet<ActionItem>();
}
```

- We're stating that association will be managed from the employee property defined in the ActionItem.



1:M w/ mappedBy




- If we create two Action Items, persist them and then add them into Employee and then saving employee like given below:

```
ActionItem item1 = new ActionItem("Write the Code");  
ActionItem item2 = new ActionItem("Run the Test");  
entityManager.persist(item1);  
entityManager.persist(item2);
```

```
Employee employee = new Employee("John", "Goodman", "dog", 1);  
employee.getItems().add(item1);  
employee.getItems().add(item2);
```

```
entityManager.persist(employee);
```

- will result in DB records like given below:

	 ID	 NAME	 EMPLOYEE_EMPLOYEEID
1	1	Write the Code	(null)
2	2	Run the Test	(null)

1:M w/ mappedBy



- So we first need manage the association from the mappedBy side.
 - As we stated before:
With mappedBy attribute JPA identifies attribute to which it will look at to manage associations. The side on which mappedBy is used can be seen as a mirror, or read-only.

```
employee.addItem(item1);  
...  
public void addItem(ActionItem item) {  
    item.setEmployee(this);  
    items.add(item);  
}
```

M:N



- many to many means several objects of one type can refer to several objects of another type. An example could be like a Department could contain more than one employee and an employee could be in more than one department.

```
@Entity
public class Department {
    @ManyToMany
    private Set<Employee> employees = new HashSet<Employee>();
}
```

M:N



- @JoinTable can be used here instead of @JoinColumn because there can be several instances of department and employee on each side, which are associated with an instance on the other side. Therefore we cannot keep association data in a column that referring to the other side's table.
- Instead we need an intermediate table, called as “association table”, department_employee, between those two tables, and it contains references to primary key columns in each table.

```
@Entity
public class Department {
    @ManyToMany
    @JoinTable(name = "department_employee",
        joinColumns = @JoinColumn(name = "department_id"),
        inverseJoinColumns = @JoinColumn(name = "employee_id"))
    private Set<Employee> employees = new HashSet<Employee>();
}
```




Handling Dates

- JPA usually maps Java types to SQL types with the most appropriate one. But some configuration needed sometimes like with dates.
- Java has `java.util.Date` but SQL offers *date*, *time* and *timestamp*. *date* keeps day/month/year. *time* keeps hour/minute/second. *timestamp* keeps all together with nanosecond precision.
- `@Temporal` can be used with type `Date`, `Time`, `TimeStamp` to configure the usage.

```
@Temporal(TemporalType.DATE)  
private Date birthDate;
```



Handling enums

- By default JPA persists enums with their ordinal values, a column with numeric SQL type in the database. But this is not a good practice since the list of enums can be changed like a new one can be inserted in the middle of the list and all the integer values will get shifted.
- To store the enums with their string value, it can be configured like,

```
@Enumerated(EnumType.STRING)  
private Gender gender;
```



cascade attribute

- cascade attribute defined in @OneToMany annotation to tell the JPA on how it will act on Address entity when a CRUD operation is done on Employee entity. So when an employee is persisted we don't need to deal with his addresses. This is called transitive persistence.

ALL	Cascade all operations
PERSIST	Cascade the persist operation
MERGE	Cascade the merge operation
REMOVE	Cascade the remove operation
REFRESH	Cascade the refresh operation
DETACH	Cascade the detach operation (jpa2.0)

src:jpa-one-to-many-cascaded



Base Entity

- We can group common attributes in a base class and extend all of our entities from it.
(DRY Principle)
- We need to introduce some new annotations first, in order to understand the idea of base class.
 - `@MappedSuperclass`
 - `@PrePersist`
 - `@PreUpdate`



@MappedSuperclass

- A mapped superclass is not a persistent class, but allow common mappings to be defined for its subclasses.

```
@MappedSuperclass
public abstract class User {

    @Id
    @GeneratedValue
    private int id;
}
```

```
@Entity
public class Employee extends User {

    private String firstname;
    private String lastname;
    private String password;
    private int type;
}
```

src:jpa-mappedsuperclass



Pre Processors

- @PrePersist: Executed before the entity manager persist operation is actually executed or cascaded. This call is synchronous with the persist operation.
- @PreUpdate: Executed before the database UPDATE operation.



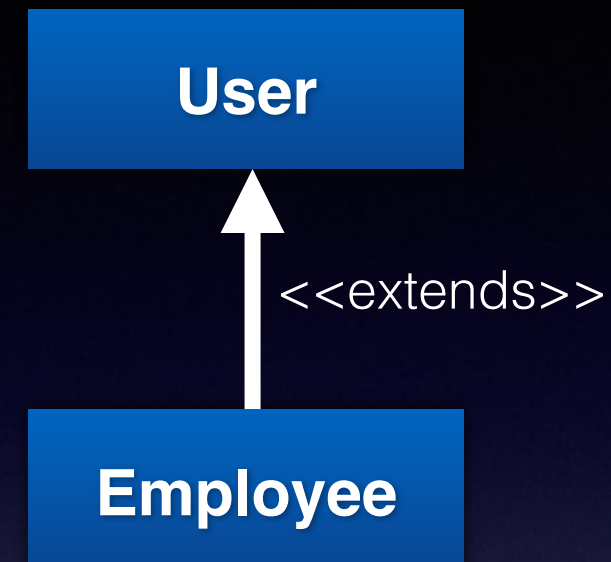
More on Processors...

- **@PrePersist:** Executed before the entity manager persist operation is actually executed or cascaded. This call is synchronous with the persist operation.
- **@PreRemove:** Executed before the entity manager remove operation is actually executed or cascaded. This call is synchronous with the remove operation.
- **@PostPersist:** Executed after the entity manager persist operation is actually executed or cascaded. This call is invoked after the database INSERT is executed.
- **@PostRemove:** Executed after the entity manager remove operation is actually executed or cascaded. This call is synchronous with the remove operation.
- **@PreUpdate:** Executed before the database UPDATE operation.
- **@PostUpdate:** Executed after the database UPDATE operation.
- **@PostLoad:** Executed after an entity has been loaded into the current persistence context or an entity has been refreshed.



Inheritance Strategies

- Let's think of a model as given here:



- If we don't use any `@MappedSuperClass` annotation, the default behaviour for this inheritance will be `SINGLE_TABLE` strategy, which means that only one table will be generated for the class hierarchy.
- The `@Inheritance` annotation can be specified on the entity class that is the root of the entity class hierarchy.
- If no inheritance type is specified for an entity class hierarchy, the `SINGLE_TABLE` mapping strategy is used.



Inheritance Strategies

- InheritanceType is an enum that resides under JPA. It has 3 values,
- SINGLE_TABLE: A single table per class hierarchy.
- TABLE_PER_CLASS: A table per concrete entity class.
- JOINED: A strategy in which fields that are specific to a subclass are mapped to a separate table than the fields that are common to the parent class, and a join is performed to instantiate the subclass.



Handling Transactions

- While handling the data, we created instances of `EntityManager` and `EntityTransaction`. To persist employee and addresses to the database for instance it's necessary to have transactions defined.
- We will soon use Spring to handle transactions more gently. It's not viable to open/close transactions every time we need to persist some data.
- We need to go a more non-obtrusive declarative approach where some transactional mechanism is applied on our code.



Detailed Scenarios

@Embeddable & @Embedded



- @Embeddable defines a class whose instances are stored as an intrinsic part of an owning entity.
- This can be stated with an example like: Think of a class named Period that has a startDate and an endDate.
- If we say that Employee has a WorkingPeriod, what would be the result in class diagram-wise and table-wise?

@Embeddable & @Embedded



- The class definitions will be as below.

```
@Embeddable
public class Period {
    @Temporal(TemporalType.DATE)
    private Date startDate;
    @Temporal(TemporalType.DATE)
    private Date endDate;
}
```

```
@Entity
public class Employee {
    @Embedded
    private Period workingPeriod;
}
```

- In table-wise:

The screenshot shows a database management tool interface with a tab for 'EMPLOYEE'. The 'Columns' tab is selected, displaying the table structure. The columns are: EMPLOYEEID, FIRSTNAME, LASTNAME, PASSWORD, TYPE, ENDDATE, and STARTDATE. Below the column list, a single row of data is shown for EMPLOYEEID 1, with FIRSTNAME 'John', LASTNAME 'Goodman', PASSWORD 'dog', TYPE 1, ENDDATE '12-JAN-15', and STARTDATE '10-JAN-15'.

	EMPLOYEEID	FIRSTNAME	LASTNAME	PASSWORD	TYPE	ENDDATE	STARTDATE
1	1	John	Goodman	dog	1	12-JAN-15	10-JAN-15

src:jpa-embeddable

Collection of Elements



- We did `@OneToMany` and `@ManyToMany` but the other side of the association was always an entity.
- How about persisting a list of Strings?
- For this: JPA offers `@ElementCollection`, defines a collection of instances of a basic type (remember `@Basic?`) or `@Embeddable` class.
- How a Set of Strings (`Set<String>`) will be stored?
Any ideas?



Collection of Elements

- The definition will be as below:

```
@ElementCollection  
private Set<String> nicknames = new HashSet<String>();
```

- The table structure will be as below:

The screenshot shows a SQL Enterprise Manager window with two tabs: 'BUWORKSHOP' and 'EMPLOYEE_NICKNAMES'. The 'EMPLOYEE_NICKNAMES' tab is active, displaying the 'Columns' view. The table has two columns: 'EMPLOYEE_EMPLOYEEID' and 'NICKNAMES'. The 'Data' tab is also visible, showing three rows of data. The first row has '1' for 'EMPLOYEE_EMPLOYEEID' and 'The Dude' for 'NICKNAMES'. The second row has '2' for 'EMPLOYEE_EMPLOYEEID' and 'The Jackal' for 'NICKNAMES'. The third row has '3' for 'EMPLOYEE_EMPLOYEEID' and 'The Ripper' for 'NICKNAMES'.

	EMPLOYEE_EMPLOYEEID	NICKNAMES
1	1	The Dude
2	1	The Jackal
3	1	The Ripper

src:jpa-elementcollection



Collection of Elements

- How about storing a list of @Embeddable?
- Let's think of an example where a Lecture can have Set of Period, where each period defines the StartTime and EndTime of a Lecture.

```
@Entity
public class Lecture {
    @ElementCollection
    private Set<Period> timings = new HashSet<Period>();
}
```

```
@Embeddable
public class Period {
    @Temporal(TemporalType.TIMESTAMP)
    private Date startTime;
    @Temporal(TemporalType.TIMESTAMP)
    private Date endTime;
}
```


Ordering of Elements



- What if we wanted an order between the elements of timings of a Lecture?
- @OrderColumn provides this feature as:

```
@ElementCollection  
@OrderColumn  
private List<Period> timings = new LinkedList<Period>();
```

Persisting Maps



- Suppose that Employee contains a list of Phone, where phones stored with a type in a map as:

```
private Map<PhoneType, Phone> phones = new HashMap<PhoneType, Phone>();
```

- To store the list, we will define a one-to-many relationship by depicting the key column for map.



To
CRUD
or not to
CRUD

For CRUD



- To find the employee instances along with their addresses:

```
Employee found = entityManager.find(Employee.class, _id_);
```

- To delete an employee instance with their addresses:

```
User found = entityManager.find(Employee.class, 5);  
entityManager.remove(found);  
found = entityManager.find(Employee.class, 5);  
System.out.println(found);
```

- To show the SQLs generated, put property to persistence.xml:

```
<property name="hibernate.show_sql" value="true" />
```




JPQL

- Stands for Java Persistence Query Language
- JPA also offers an object query language whose structure is similar to SQL, but instead of using table, and column names, we are able to use entity names and properties in Java classes.
- Printing the lastname of the retrieved employees with a query would be like:

```
Query query = entityManager.createQuery("select e from Employee e  
where e.firstname like :firstname");  
query.setParameter("firstname", "John");  
List<User> resultList = query.getResultList();  
for (Employee e : resultList) {  
    System.out.println(e.getLastname());  
}
```



Spring JPA Support



Spring JPA Support

- With Spring, it gets easier to use JPA.
Spring provides:
 - Easier and more powerful persistence unit configuration
 - Automatic EntityManager management
 - Integrated transaction management
 - Easier Testing



New Dependencies

```
<dependency>  
  <groupId>org.springframework</groupId>  
  <artifactId>spring-context</artifactId>  
  <version>4.1.6.RELEASE</version>  
</dependency>
```

org.springframework.context.support.ClassPathXmlApplicationContext

```
<dependency>  
  <groupId>org.springframework</groupId>  
  <artifactId>spring-orm</artifactId>  
  <version>4.1.6.RELEASE</version>  
</dependency>
```

org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean



LocalContainerEntityManagerFactoryBean

- We should first create an EntityManager as we did in the sample application. We will use a Factory Bean to create the Entity Manager.
- dataSource: Contains the resource definition to connect to the underlying database.
- packagesToScan: Scans for JPA annotations.
- hibernate.dialect: Hibernate uses the dialect to generate correct SQL statements to execute on the DB.



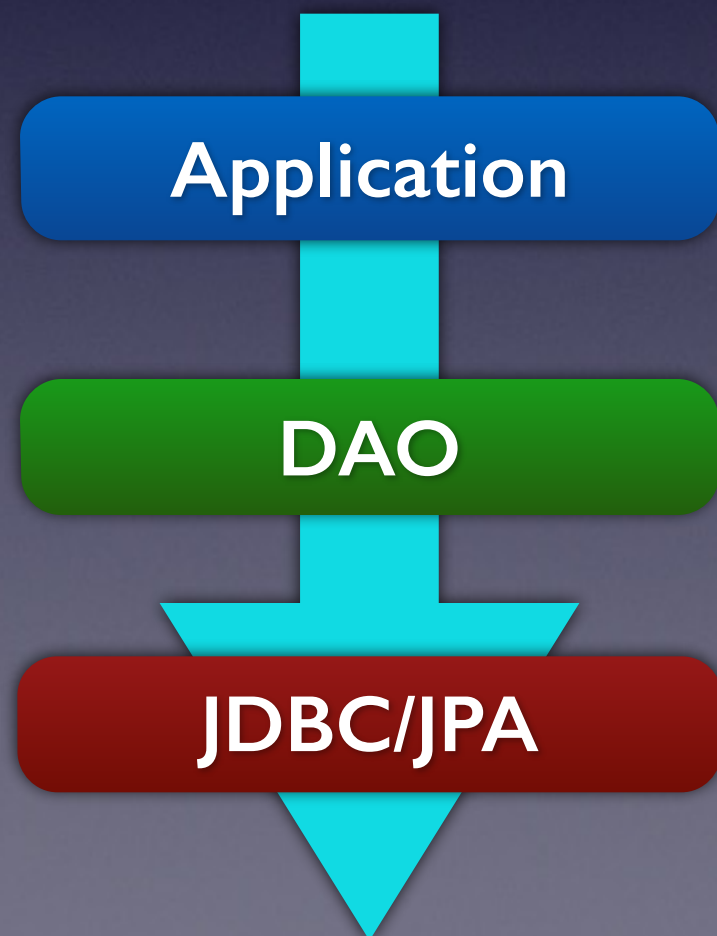
LocalContainerEntityManagerFactoryBean

- persistenceProviderClass:
HibernatePersistenceProvider defined. It's an implementation of javax.persistence.PersistenceProvider interface. Its role is to create actual EntityManagerFactory instance specific to the JPA vendor, which is Hibernate in our case.



Remember the DAO Pattern?

- DAO stands for Data Access Object
- The pattern is a widely accepted approach to make an abstraction on top of the persistency layer/code.



- You can easily replace the DAO layer and its attached JDBC/JPA code since the persistency layer is detached from the application code.



@PersistenceContext

- We'll inject EntityManager into our DAO layer easily with @PersistenceContext annotation, which is coming from JPA.

```
@Repository
public class UserDaoImpl implements UserDao {

    @PersistenceContext
    private EntityManager entityManager;

    @Transactional
    public void create(User user) {
        entityManager.persist(user);
    }

    public User find(int id) {
        return entityManager.find(User.class, id);
    }
}
```

src:jpa-one-to-many-with-spring