Hibernate Framework

Majrul Ansari

Java Technology Trainer

My Introduction

- Working on Java Technology since 9999 ;-)
- More than 8 years of Corporate Training experience, otherwise a Consultant/Freelancer
- Brainbench **Certified** Hibernate Professional
- Springing & Hibernating for more than 5 years now
- Apart from Spring & Hibernate, EJB, Struts, JSF, REST, WebServices and JME, Android are some of the other technologies that I am comfortable with

Agenda

- Introduction to Hibernate and its features
- Role of JPA
- Writing entity classes and mapping metadata
- Handling different forms of associations and relationships between entities
- Exploring fetching strategies like lazy, eager, batch and others
- Understanding HQL/Query API, Criteria API
- Concurrency/Locking support in Hibernate
- Caching support

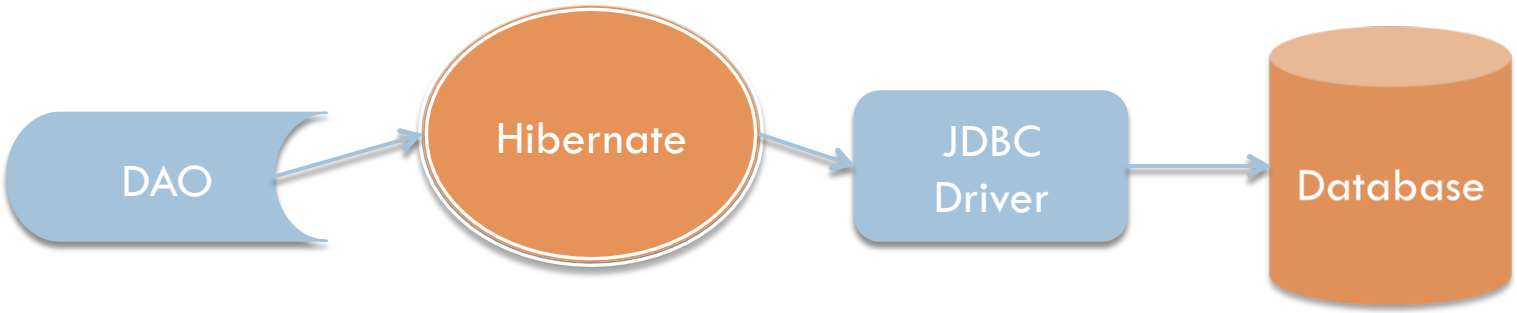
To begin with

- Introduction to Hibernate
- Understanding Object states in Hibernate
- Hibernate and it's different versions
- Hibernate and JPA support

Hibernate is an ORM

- Hibernate is an ORM (Object Relational Mapping) tool/framework with a powerful API for managing persistence of Objects
- Each row is represented as an instance of a class mapped to the corresponding table
- Hibernate provides an API which completely hides the underlying JDBC calls from the developer

Introduction

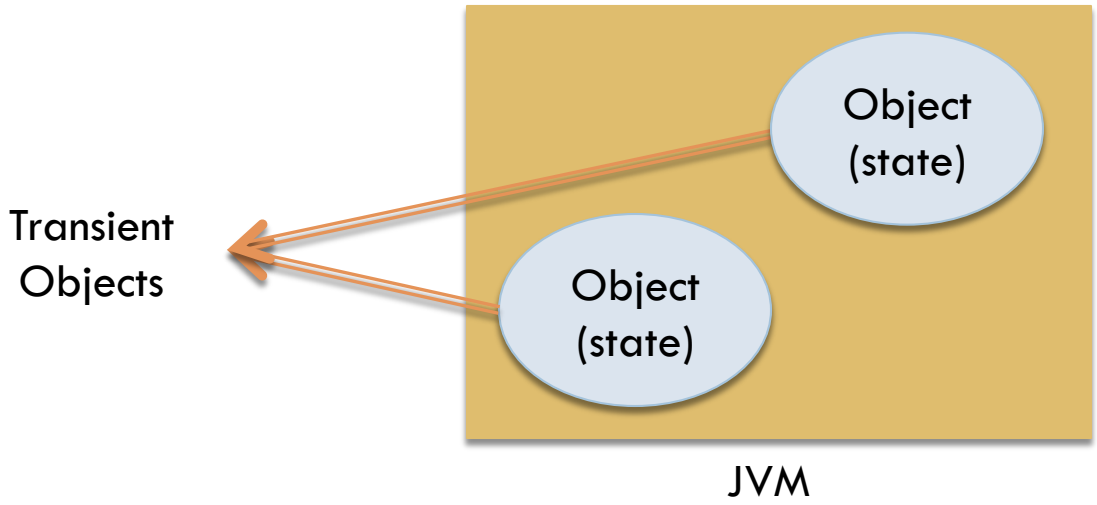


Different states on an Object

- An Object can be in any of these states:
  - Transient Object
  - Persistent Object
  - Detached Object
- When an object is created, it's transient in nature
- When an object is associated with the persistence layer, it's a persistent object
- When an object is no more associated with the persistence layer, it's a detached object

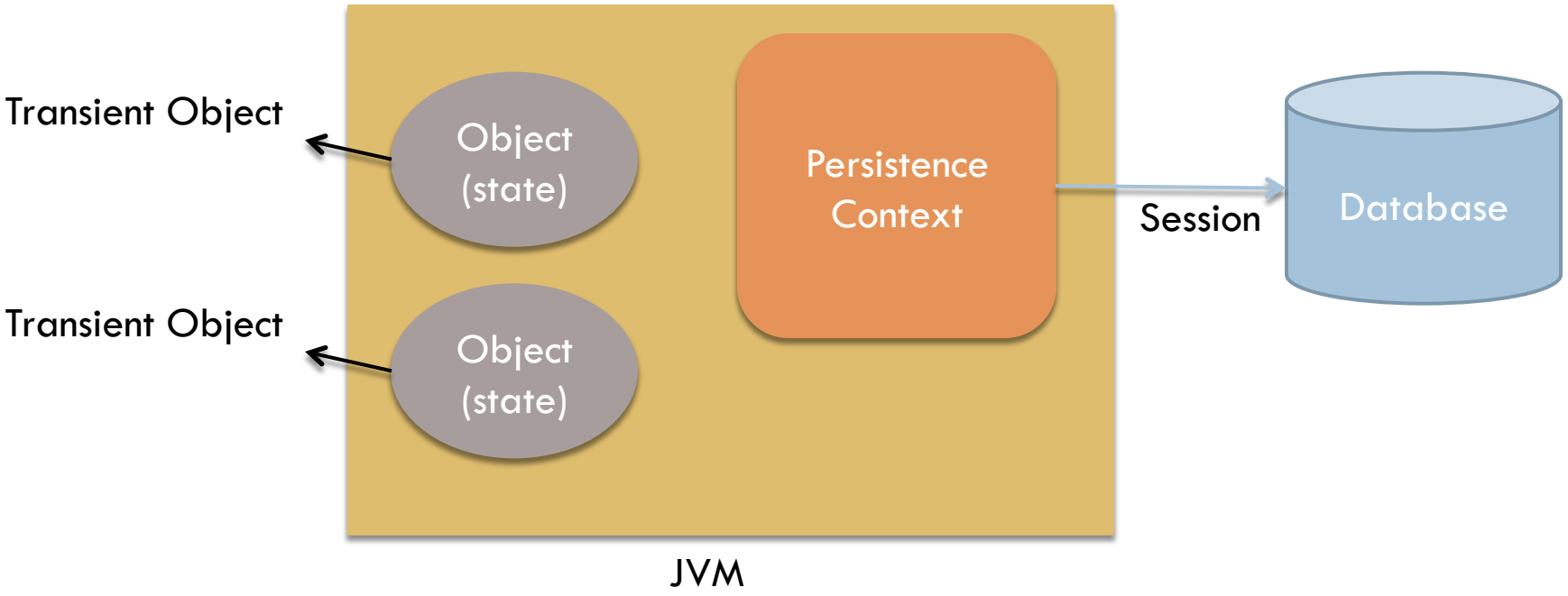
Transient Object

- Is an instance of a class created within the JVM process scope and valid as long as reference to the same exists
- Modifying the state of transient object does not affects the database



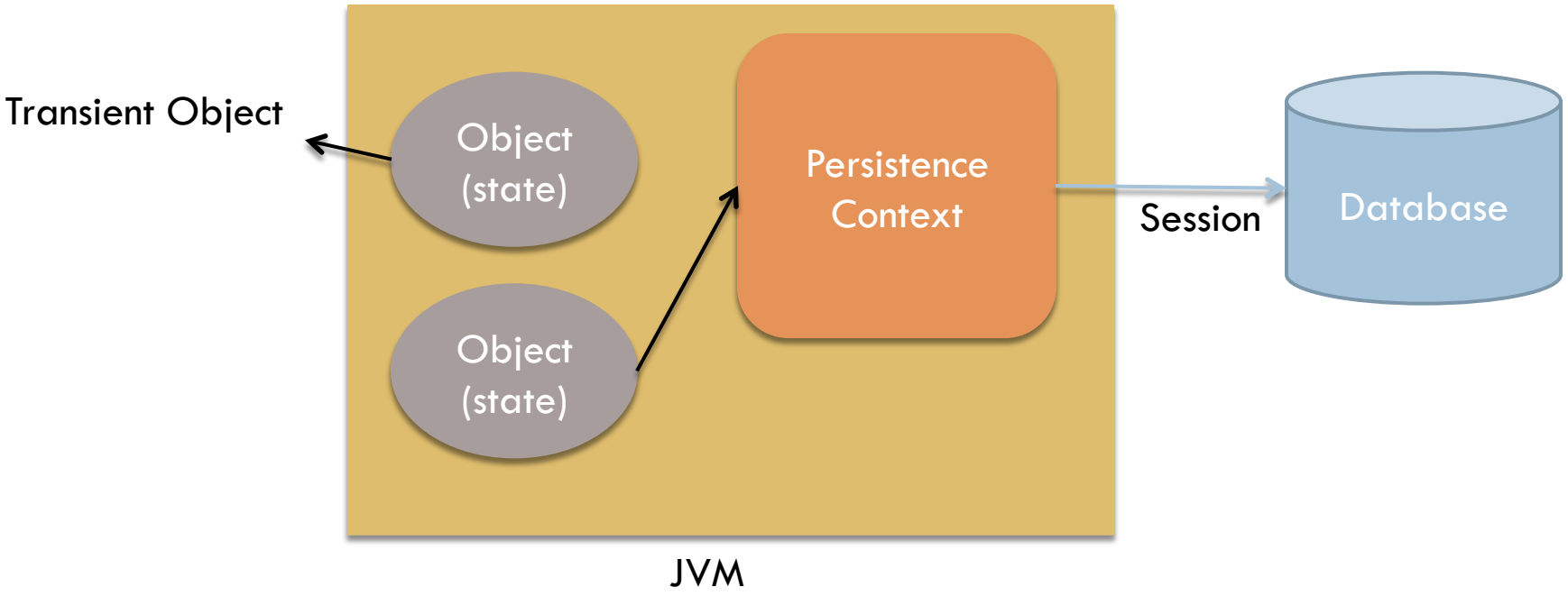
Persistent Object

- As soon as a transient object is associated with a persistence context, it's a persistent object
- Modifying the state of a persistent object will be synchronized with the underlying database



Persistent Object

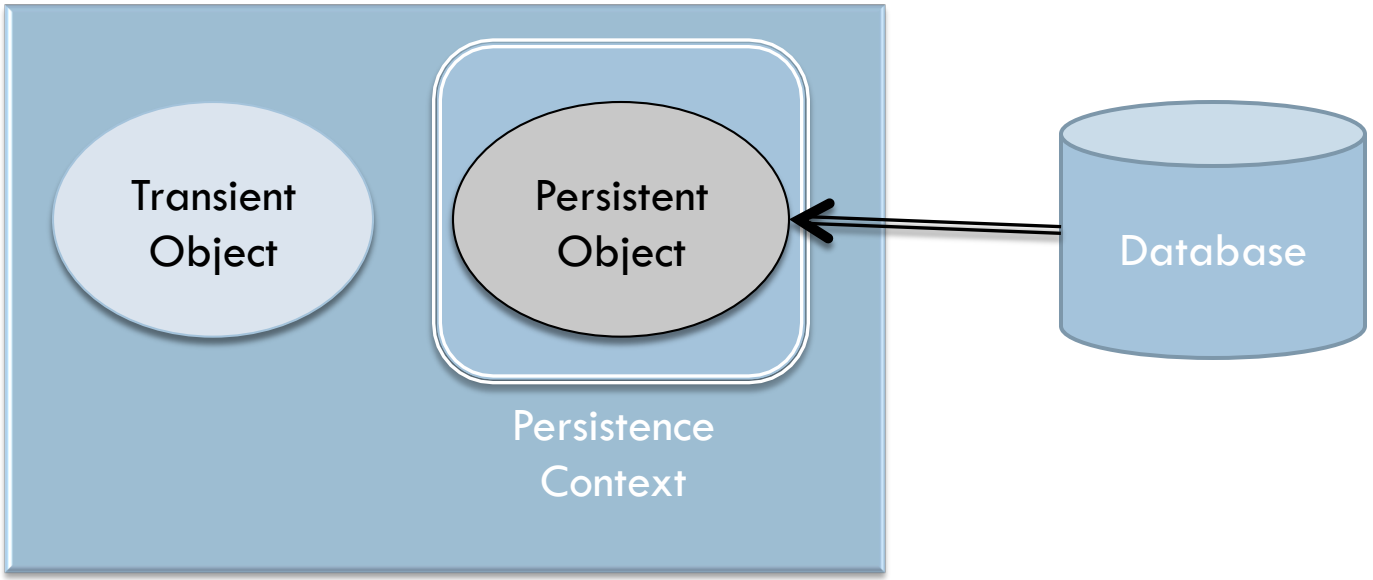
- As soon as a transient object is associated with a persistence context, it's a persistent object
- Modifying the state of a persistent object will be synchronized with the underlying database



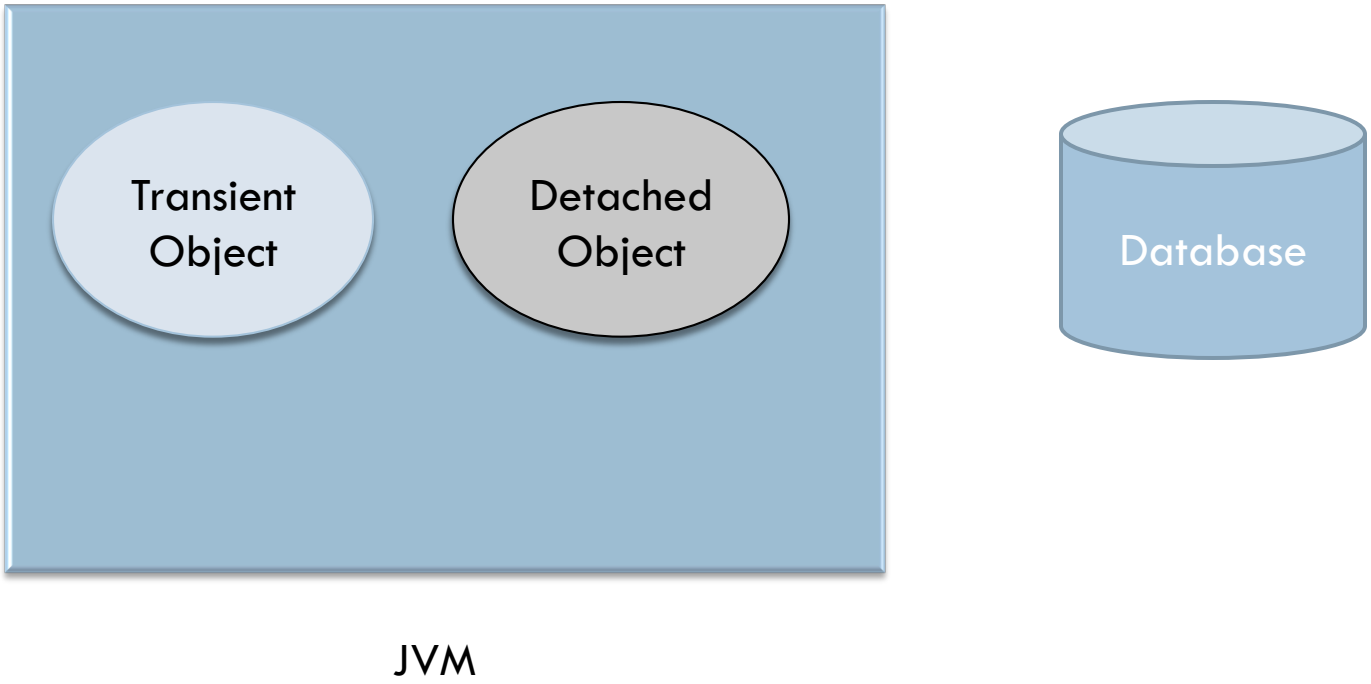
Detached Object

- An object which was loaded in some persistence context but the context has been closed on behalf of some transactional process being committed/rolled back
- Modifying state of detached instance will not be updated in the database till not reattached with some persistence context

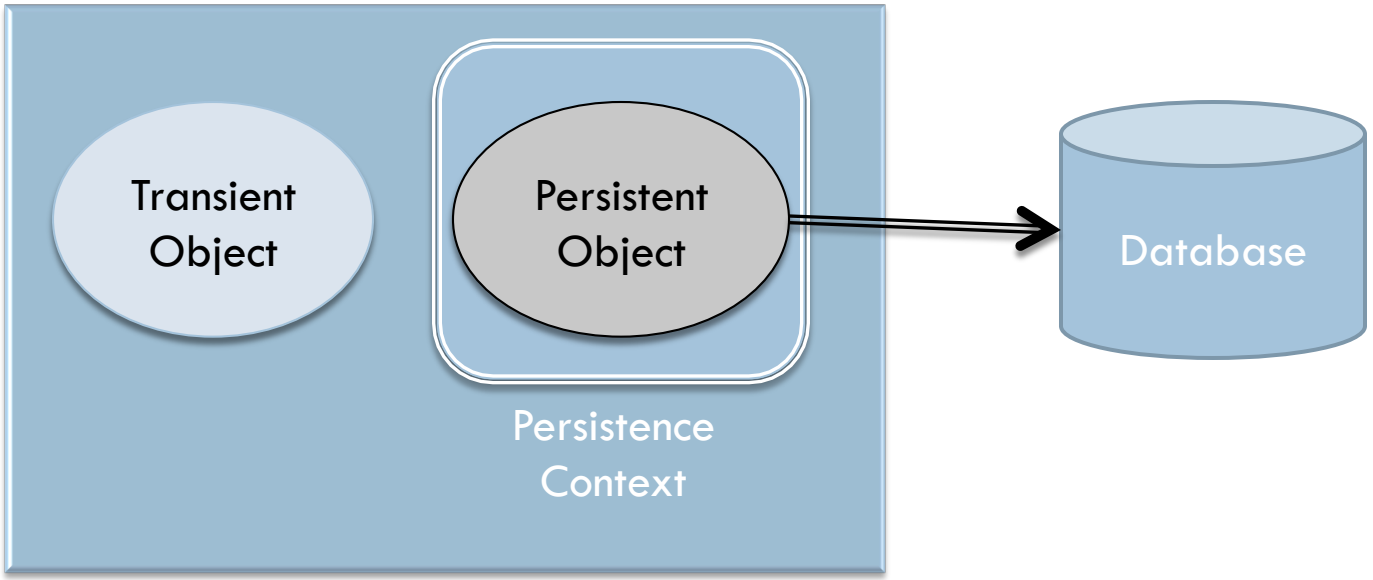
Detached Object



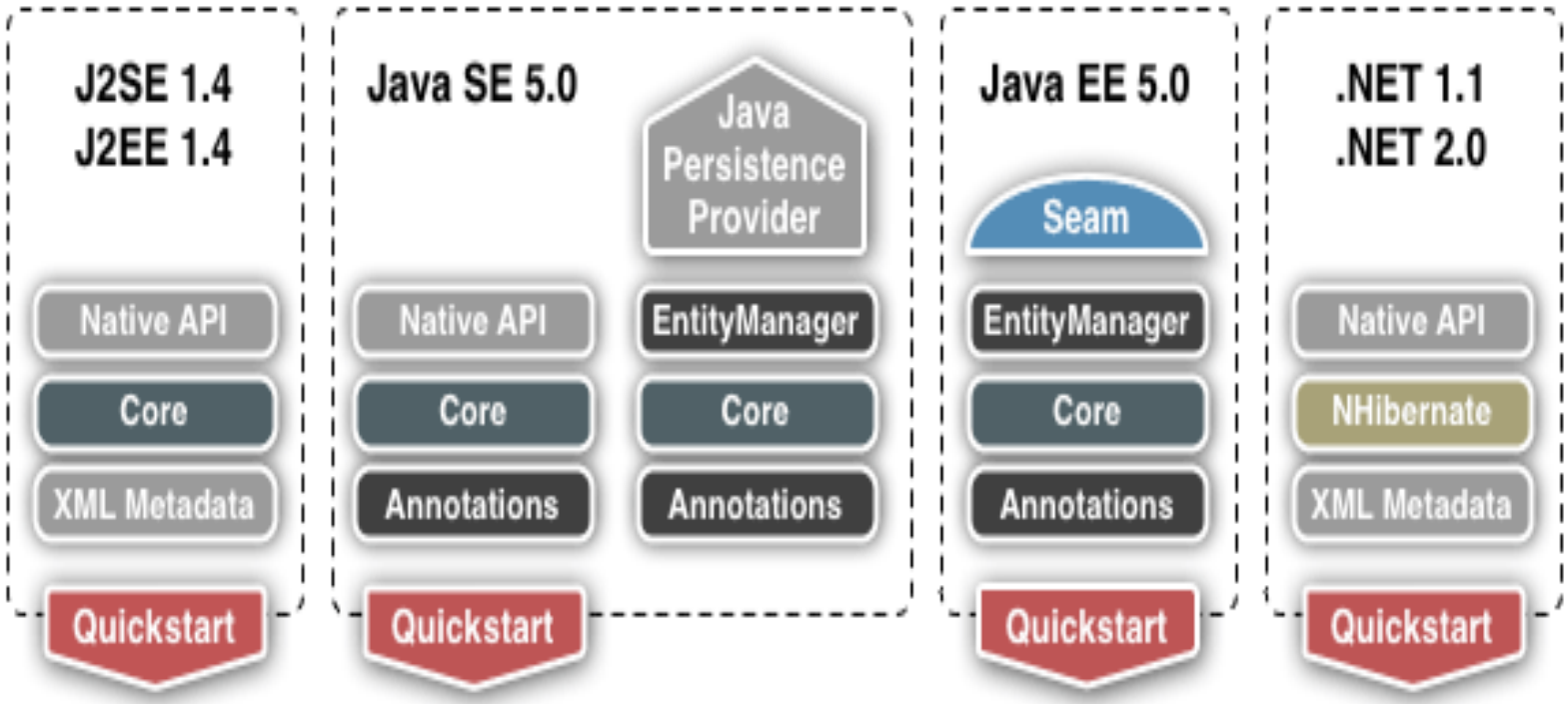
Detached Object



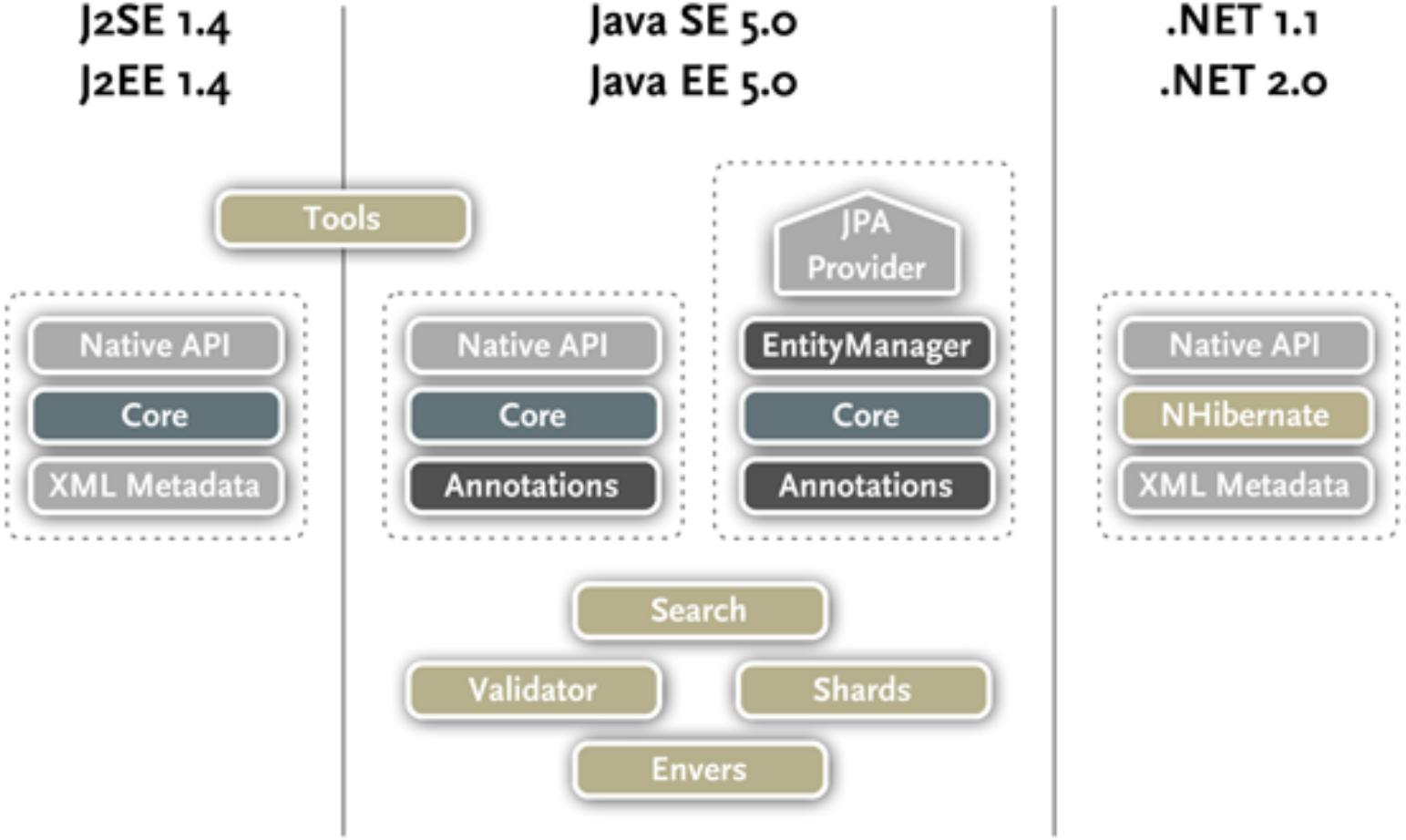
Detached Object



Hibernate and it's different versions



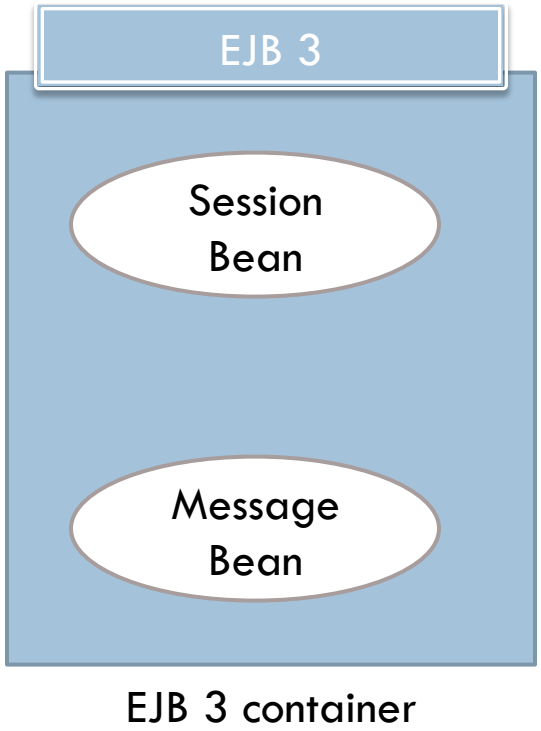
Hibernate in it's latest avatar



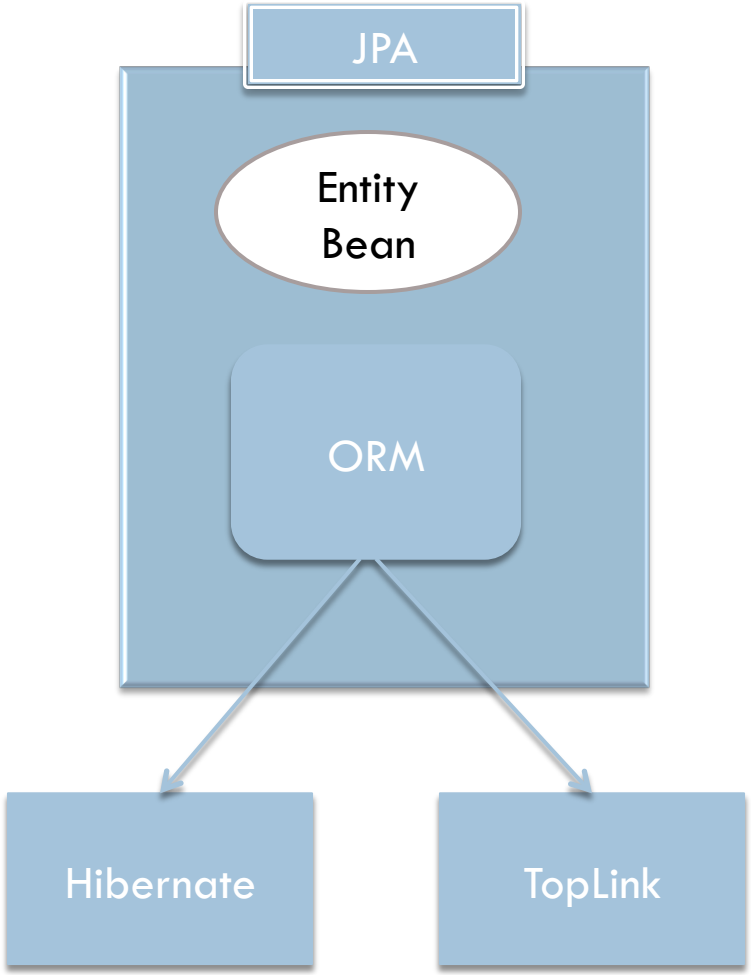
Cont'd…

- Hibernate 3.2
  - Most commonly used production version of Hibernate
  - Full support for JPA 1.0
- Hibernate 3.5 & 3.6
  - Full support for JPA 2.0
- Hibernate 4.x
  - Is the latest revision with lot's of structural changes

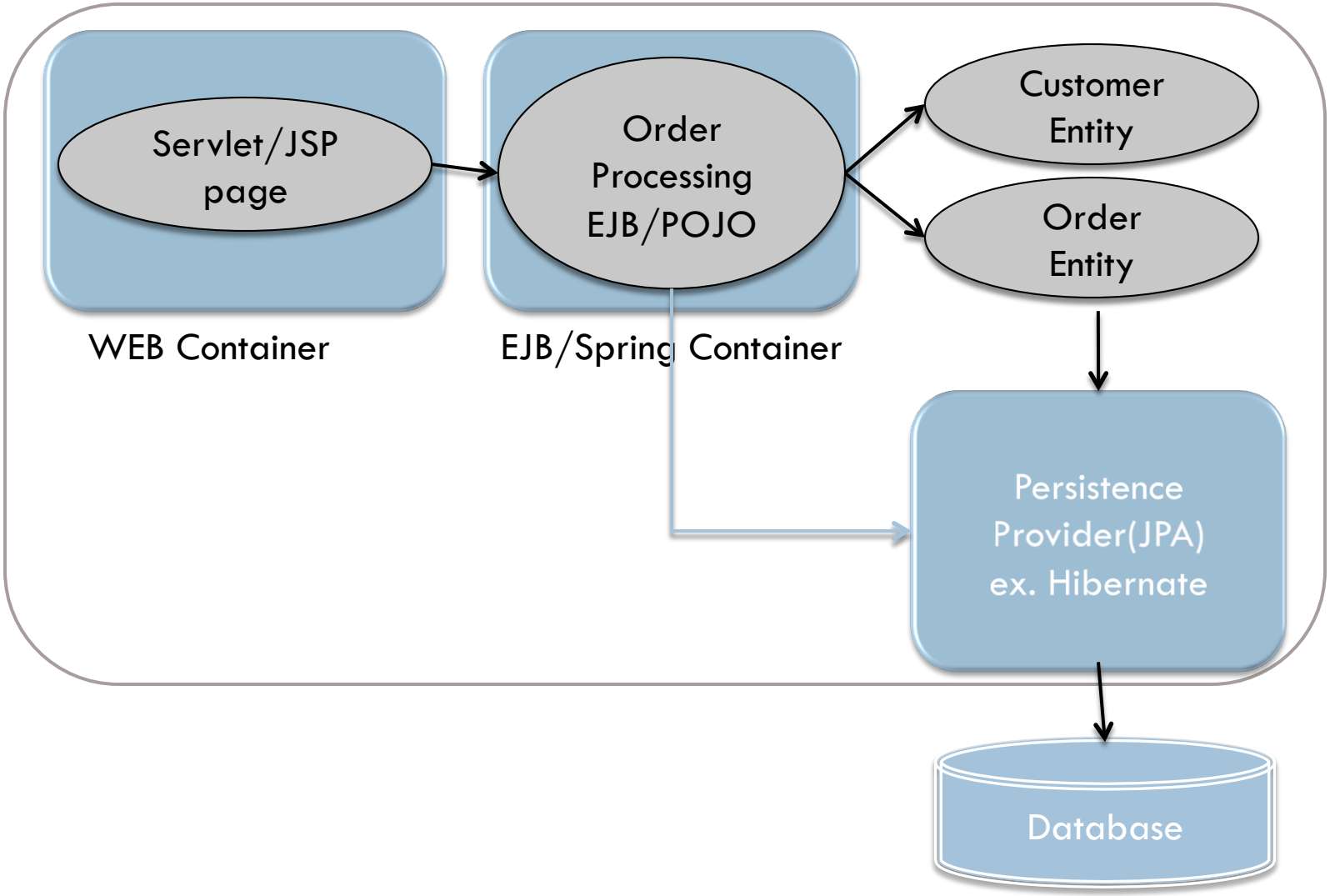
Hibernate and JPA



EJB 3 container



Hibernate and JPA



Hibernate design goal

- Hibernate's goal is to relieve the developer from 95 percent of common data persistence related programming tasks, compared to manual coding with SQL and the JDBC API
- Then what is left!
  - Handling stored procedures
  - Handling UDT
  - etc…

About Stored procedures

- **Hibernate may not be the best solution for data-centric applications that only use stored-procedures to implement the business logic in the database, it is most useful with object-oriented domain models and business logic in the Java-based middle-tier**
- However, Hibernate can certainly help you to remove or encapsulate vendor-specific SQL code and will help with the common task of result set translation from a tabular representation to a graph of objects

Manual JDBC vs. Hibernate

- What if you have 100s of tables. Lot's of repetitive JDBC code across DAO's
- How to minimize database hits while writing JDBC code on our own a.k.a Caching
- How to capitalize on performance optimization and features specific to database & it's drivers
- Handling exceptions, try, catch & finally is not really peace of mind

Lab 01 - Agenda

- Our first example on Hibernate
- Mapping entities using XML as well as Annotations approach
- Hibernate API for persistent activities
- Using JPA instead of Hibernate API

Entity class

- An entity in Hibernate is a simple POJO class with
  - Usual getters/setters and a default constructor at least
- An entity represents persistent state, so no business logic should be written in this class
- An entity can hold non persistent state which can also be referred to as transient state

POJO class


```
public class CD {

        private int id;
        private String title;
        private String artist;
        private Date purchaseDate;
        private double cost;
        …
```

Mapping an entity

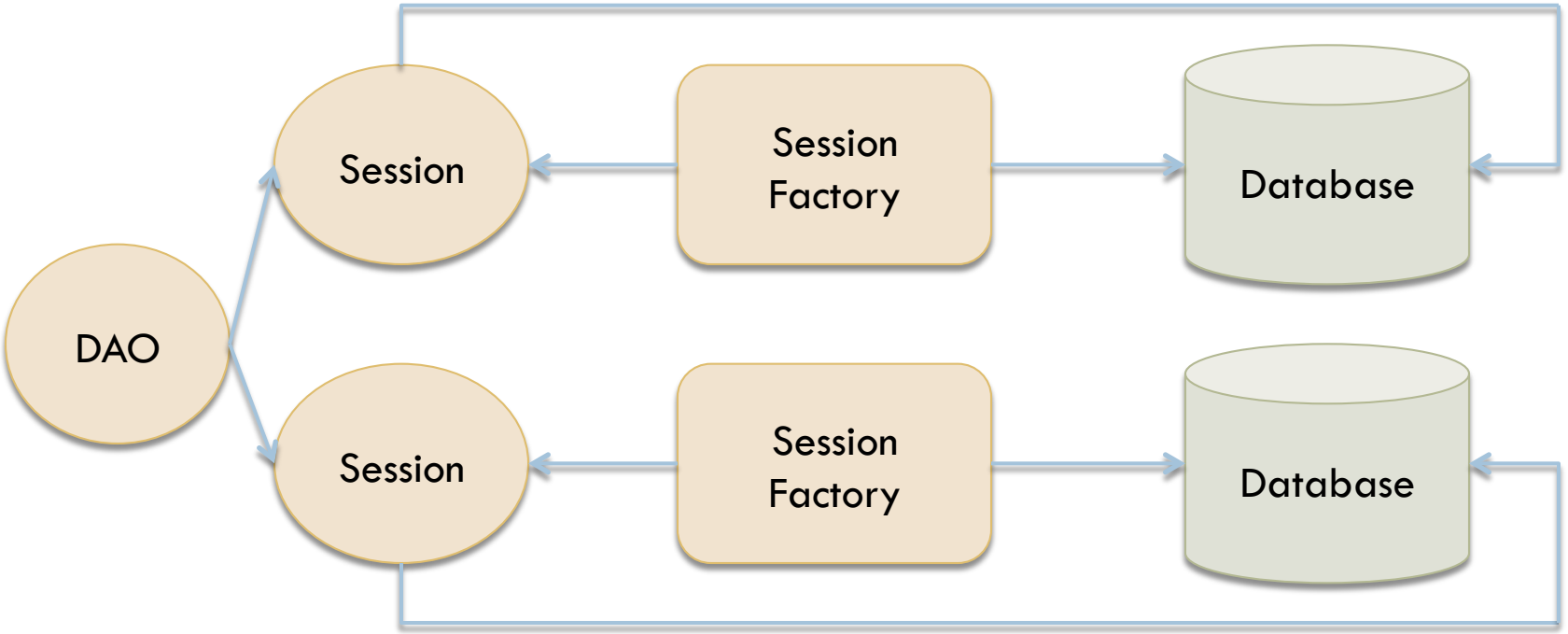
- Traditionally, mapping an entity to a corresponding table in the database was achieved using xml files
- In JDK 5, annotations were introduced as an alternative to xml style configuration. In annotations, the same metadata is provided within the entity class itself
- First let us have a look at the xml way of configuration and then we will compare it with annotations

CD.hbm.xml


```
<class name=”ex01.xml.CD" table="CD">
      <id name="id" type="int">
       <column name="id" not-null="true" />
         <generator class="increment"/>
       </id>

       <property name="title" />
       <property name="artist" />
       <property name="purchaseDate" type="date">
              <column name="purchase_date" />
       </property>
       <property name="cost" />
</class>
```

Building a SessionFactory

- A SessionFactory is a representation of a Database instance in Hibernate. A SessionFactory object provides Sessions to interact with the target database for us



Cont'd…

- Information about the database and mapping of persistent entities can be provided through hibernate.cfg.xml file. A SessionFactory object can be constructed without any xml file as well. But to start with we will have an xml. Also when creating a SessionFactory object, Hibernate searches for hibernate.properties file if found in the classpath. So we can keep some settings in the properties file and rest in the xml
- So after the configuration file is ready, we just need to create a SessionFactory object and then obtain a Session to perform persistence activities

hibernate.cfg.xml


```
<session-factory>
        <property name="hibernate.dialect">
                org.hibernate.dialect.MySQLDialect
        </property>
        <property name="hibernate.connection.driver_class">
                com.mysql.jdbc.Driver
        </property>

        <mapping resource=”ex01/xml/CD.hbm.xml" />
        …
</session-factory>
```

A Simple hibernate.cfg.xml file

CDTest.java


```
@Test
public void testCase1() {
        SessionFactory sessionFactory =
                new Configuration()
                        .configure("ex01/xml/hibernate.cfg.xml")
                        .buildSessionFactory();

        Session session = sessionFactory.getCurrentSession();
        Transaction tx = session.beginTransaction();

        CD cd = new CD("Some Title", "Some Artist",
                                                new Date(), 9.99);
        session.save(cd);
        tx.commit();
}
```

Hibernate 3.x okay, 4.x deprecated

Annotations instead of XML


```
@Entity
@Table(name = "CD")
@GenericGenerator(name="incr", strategy="increment")
public class CD {

        @Id @GeneratedValue(generator="incr")
        private int id;

        private String title;
        private String artist;

        @Temporal(TemporalType.DATE)
        private Date purchaseDate;

        private double cost;
```

Annotations approach

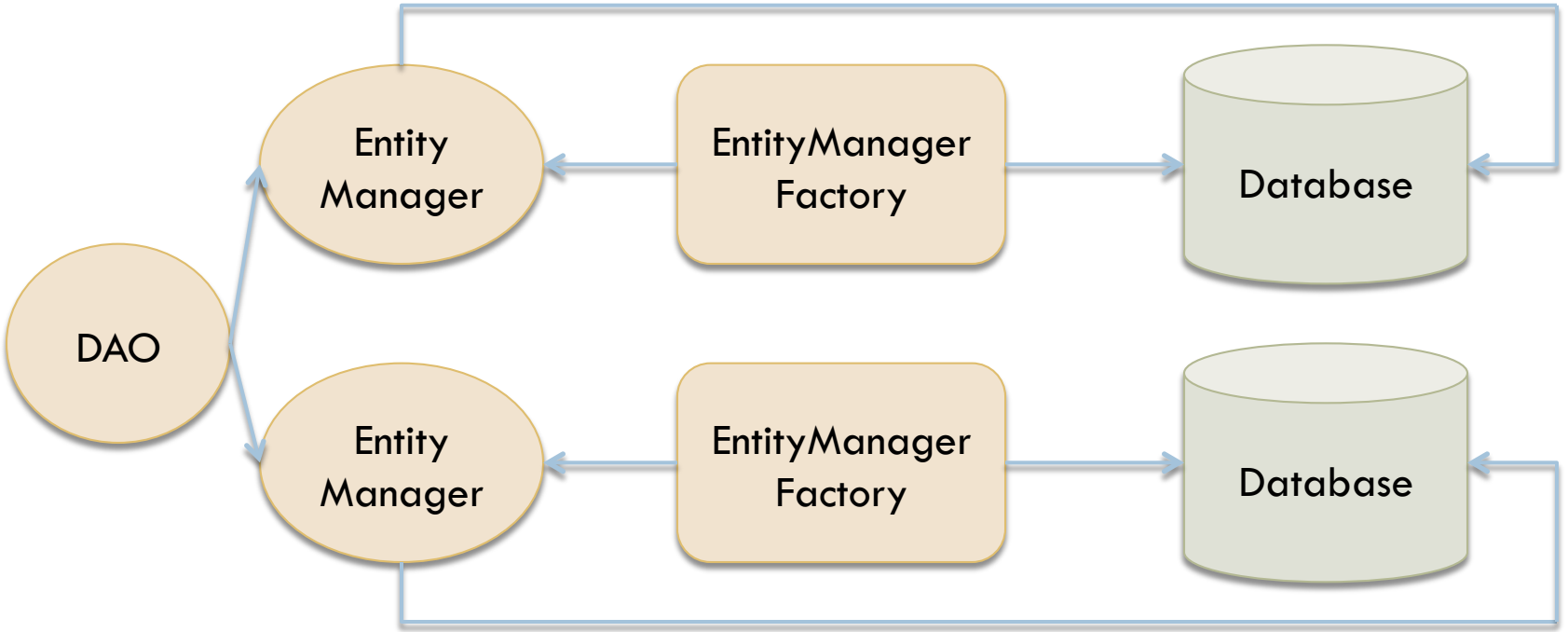
- Hibernate is 100% JPA complaint ORM. So instead of introducing it's own annotations like XML, it rather supports all existing JPA annotations
- Which means one can still use Hibernate API for JPA annotated entities
- In the hibernate.cfg.xml file, we can provide the names of annotated classes instead of hbm files as an alternative

hibernate.cfg.xml


```
<session-factory>
        <property name="hibernate.dialect">
                org.hibernate.dialect.MySQLDialect
        </property>
        <propertyname="hibernate.connection.driver_class">
                com.mysql.jdbc.Driver
        </property>
        <property name="hibernate.connection.url">
                jdbc:mysql://localhost:3306/test
        </property>
        <mapping class=”ex02.annotations.CD" />
</session-factory>
```

Using JPA instead of Hibernate API

- In JPA, the names of the interfaces are different from Hibernate
- SessionFactory becomes EntityManagerFactory
- Session becomes EntityManager



Cont'd…

- In JPA, all ORM specific configuration is done by default in persistence.xml file. This file should be present in the META-INF folder of our project. This file will is read whenever we create the EntityManagerFactory object
- Even in JPA, we can map entities using xml, but that's not the general practice. Annotations are a preferred way of providing entity metadata in JPA

META-INF/persistence.xml file


```
<persistence-unit name="JPA">
        <provider>
                org.hibernate.ejb.HibernatePersistence
        </provider>

        <class>ex02.annotations.CD</class>

        <properties>
         <property
                name="hibernate.dialect"
                value="org.hibernate.dialect.MySQLDialect" />
         <property
                name="hibernate.connection.driver_class"
                value="com.mysql.jdbc.Driver" />
        …
        </properties>
```

Benefit of JPA over Hibernate API

- By using the Hibernate API, we are tightly bound to a particular vendor. When using JPA, we can transparently introduce an ORM in our project without directly depending upon it
- Changing from one ORM to another won't effect our DAO classes badly since we are using a standard API
- This does not means we don't use Hibernate anymore, rather
  - If we wish to any other ORM other than Hibernate, it's better we use JPA instead of learning the ORM's proprietary classes

Test Class


```
@Test
public void testCase1() {
        EntityManagerFactory entityManagerFactory =
                Persistence.createEntityManagerFactory(
                        ”persistent-unit-name");
        EntityManager entityManager =
                entityManagerFactory.createEntityManager();
        EntityTransaction tx = entityManager.getTransaction();
        tx.begin();

        CD cd = new CD("Some Title", "Some Artist",
                                                new Date(), 9.99);
        entityManager.persist(cd);
        tx.commit();
        entityManager.close();
}
```

JPA support in Application Server

- Since JPA is part of the JEE 5 specification, all Application Servers need to support JPA
  - Which means all AS will come with an inbuilt ORM which will be JPA compliant
- Spring also provides full support for JPA
- Which means when using EJB 3/Spring, one can use DI (Dependency Injection) to directly access JPA *EntityManagerFactory* & *EntityManager* instances

Some of the API methods


```
CD cd = (CD) session.get(CD.class, 1);

CD cd = (CD) entityManager.find(CD.class, 1);

session.update(cd);

session.saveOrUpdate(cd);

session.merge(cd);

entityManager.merge(cd);

session.delete(cd);

entityManager.remove(cd);
```

- get/find method is used for fetching a record based on the pk column
- update method is used for updating a detached object in the database
- saveOrUpdate can be used for insert/update depending on whether the object is transient/ detached
- merge also can be used for the same purpose like saveOrUpdate
- delete/remove deletes a record from the database

More about saveOrUpdate and merge

- saveOrUpdate() does the following:
  - if the object is already persistent in this session, do nothing
  - if another object associated with the session has the same identifier, throw an exception
  - if the object has no identifier property, save() it
  - if the object's identifier has the value assigned to a newly instantiated object, save() it
  - if the object is versioned by a <version> or <timestamp>, and the version property value is the same value assigned to a newly instantiated object, save() it otherwise update() the object
- and merge() is very different:
  - if there is a persistent instance with the same identifier currently associated with the session, copy the state of the given object onto the persistent instance
  - if there is no persistent instance currently associated with the session, try to load it from the database, or create a new persistent instance
  - the persistent instance is returned
  - the given instance does not become associated with the session, it remains detached

Lab 02 - Agenda

- Understanding different types of identities in Hibernate like
  - Surrogate keys
  - Business keys
  - Composite keys
- Recommendation for overriding hashCode & equals

Identifier terminology in Hibernate

- Surrogate key
  - An additional column in the table which will be the primary key in the database and mostly be auto generated
  - Immutable
- Natural key / Business key
  - A unique non-null column in the database
  - Immutable, but can be otherwise
- Composite keys
  - More than one column forming the primary key

About hashCode and equals

- You have to override the equals() and hashCode() methods if you:
  - intend to put instances of persistent classes in a Set (the recommended way to represent many-valued associations)
- Hibernate guarantees equivalence of persistent identity (database row) and Java identity only inside a particular session scope. When you mix instances retrieved in different sessions, you must implement equals() and hashCode() if you wish to have meaningful semantics for Sets

About hashCode and equals

- It is recommended that we implement equals() and hashCode() using *Business key equality*. Business key equality means that the equals() method compares only the properties that form the business key not the primary key. It is a key that would identify our instance in the real world (a *natural* candidate key)
- Those fields which form the business identity of an object can be marked as natural-id/@NaturalId making the configuration more self-documenting

Example

Annotations approach

```
public class Book {

        private int id;
        private long isbn;
        private String title;
        private String author;
        private String publication;
```

```
@Entity
public class Book {

        @Id @GeneratedValue
        private int id;
        @NaturalId
        private String isbn;
```

```
<id name="id" type="int" length="5">
    <generator class="increment" />
</id>
<natural-id>
    <property name="isbn" />
</natural-id>
```

XML approach

Composite Primary key

- As this is a very common requirement, many entities will have a composite primary key made up of more than one column
- The general recommendation is to create a separate class representing the composite structure and the main entity holds a reference to the object of this class
- We will see the xml as well as the annotation version for this example

Example


```
public class Person {

        private Person.Id id;
        private String name;
        private int age;

        public static class Id implements Serializable {
                private String country;
                private int medicareNumber;
```

The primary key class need not be an inner class

Person.hbm.xml


```
<class name="Person" table="persons">
    <composite-id name="id" class="Person$Id">
      <key-property name="country">
        <column name="country" not-null="true"/>
      </key-property>
      <key-property name="medicareNumber">
        <column name="medicare_number" not-null="true"/>
      </key-property>
    </composite-id>
```

So by now we have seen all three identifier tags, i.e. id, natural-id & composite-id

Person.java


```
@Entity
public class Person {

	@EmbeddedId private Person.Id id;

	@Embeddable
	public static class Id implements Serializable {
		private String country;
		private int medicareNumber;
```

Agenda for Lab 03

- This is the most important lab and the backbone of this training program
- Hibernate is all about mapping and this lab goes into details of mapping entities
- Till now we have seen how to map simple entities to the database, now we will see how to map relational entities
- We will stress on both **Inheritance** and **Association** with the help of the examples found in this lab section

Cont'd…

- Overall we will discuss
  - Different ways of representing inheritance in the database
  - Different ways of dealing with unidirectional and bidirectional relationships
  - Once again I will show you both, xml as well as annotations approach wherever possible

Inheritance mapping

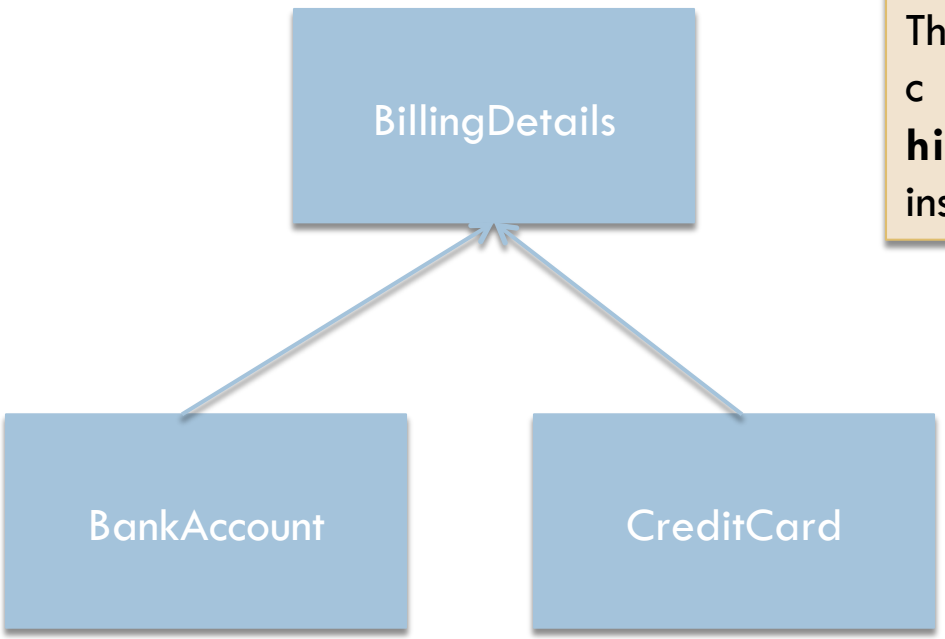
- Hibernate as well as JPA supports three basic inheritance mapping strategies
  - Single table per class hierarchy
  - Table per subclass
  - Table per concrete class
- Furthermore by mixing these strategies, we can achieve some more ways of managing inheritance

Cont'd…

- We will see in all total 5 ways of managing inheritance relationship
  - Single table per class hierarchy using a discriminator column
  - Separate table per subclass
  - Separate table per subclass using a discriminator column
  - Separate table per concrete class
  - Separate table per concrete class using SQL union
- Because of so many options available, please explore the labs step by step and carefully

Inheritance example overview

- Our example on inheritance is made up of three small entities. I will ignore the attributes of these entities to save some space

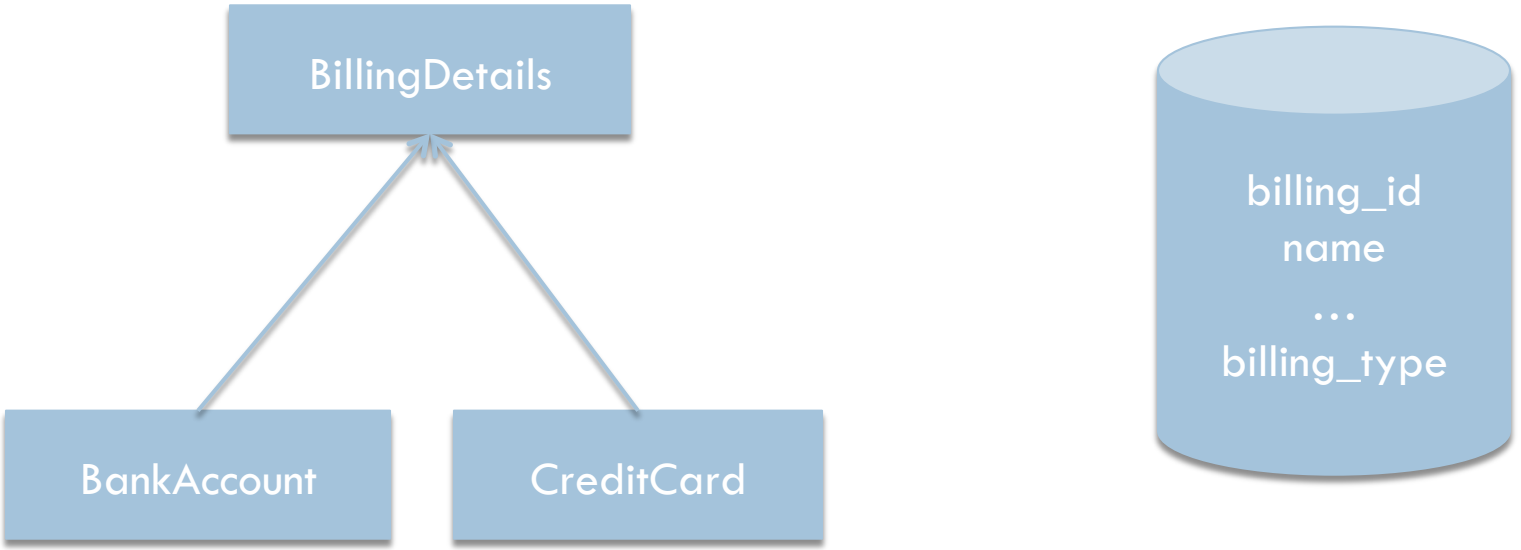


The source code for these 3 classes can be found in **hibernate.inheritance** package inside **Lab 03** folder

1. Single table per class hierarchy

- Means even though we have 3 Java classes, in the database we will have a single table to store all the information
- In such a case, we need an extra column in the table so that it can be used for identifying what record was inserted in the database, BankAccount or a CreditCard
- This extra column is called as discriminator column in the Hibernate/JPA mapping

Overview



Lab 03 / hibernate / inheritance / tableperhierarchy / BillingDetails.hbm.xml

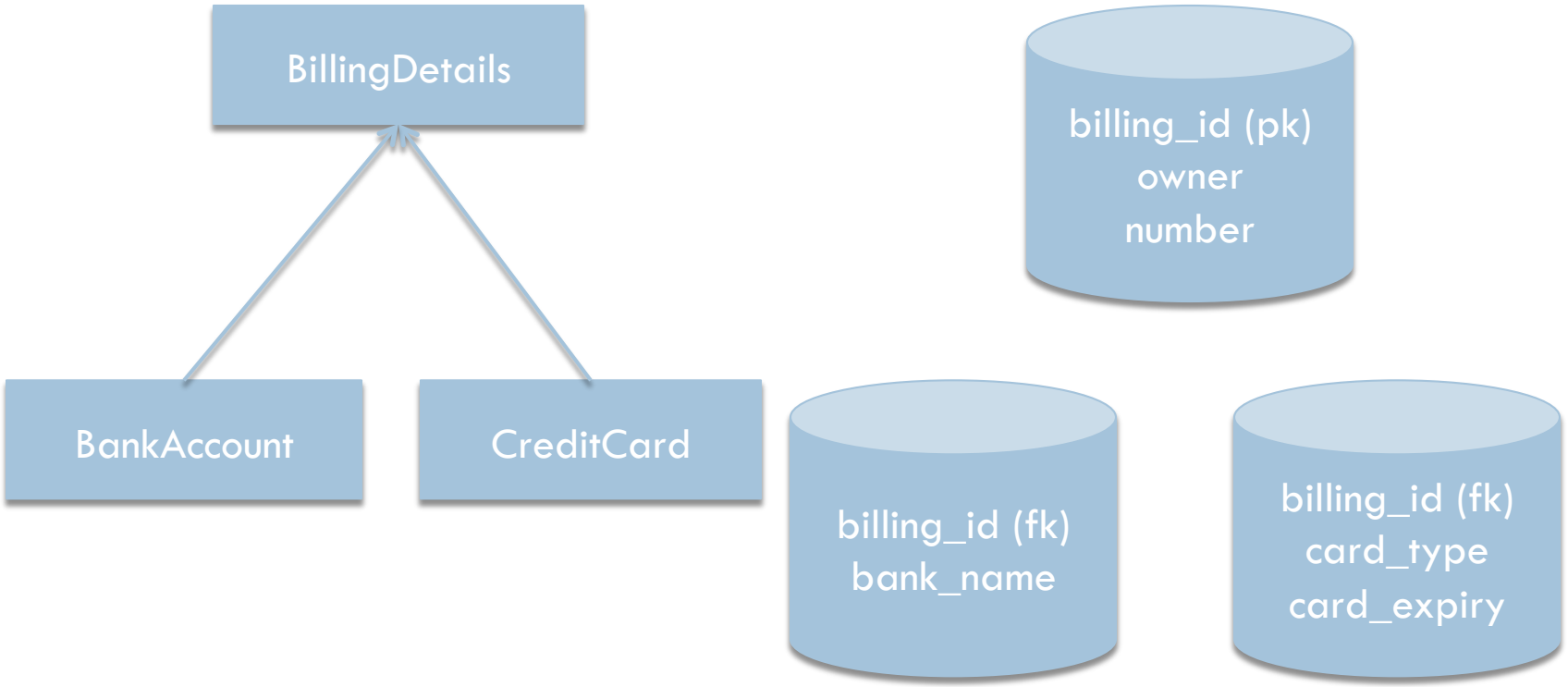
The configuration


```
<class name="BillingDetails" table="billing_details_1">
 <id name="id" column="billing_id" type="int">
  <generator class="increment" />
 </id>
 <discriminator column="billing_type" type="string" />
 <property name="owner" />
 <property name="number" column="no" />
 <subclass name="BankAccount" discriminator-value="BA">
  <property name="bankName" column="bank_name" />
 </subclass>
 <subclass name="CreditCard" discriminator-value="CC">
  <property name="type" column="card_type" />
  <property name="expiryMonth" column="expiry_month" />
  <property name="expiryYear" column="expiry_year" />
 </subclass>
</class>
```

2. Separate table per subclass

- In this strategy, we will have separate tables in the database, one for each subclass. The common fields of the parent class will be mapped to a common table

Overview



Lab 03 / hibernate / inheritance / tablepersubclass / BillingDetails.hbm.xml

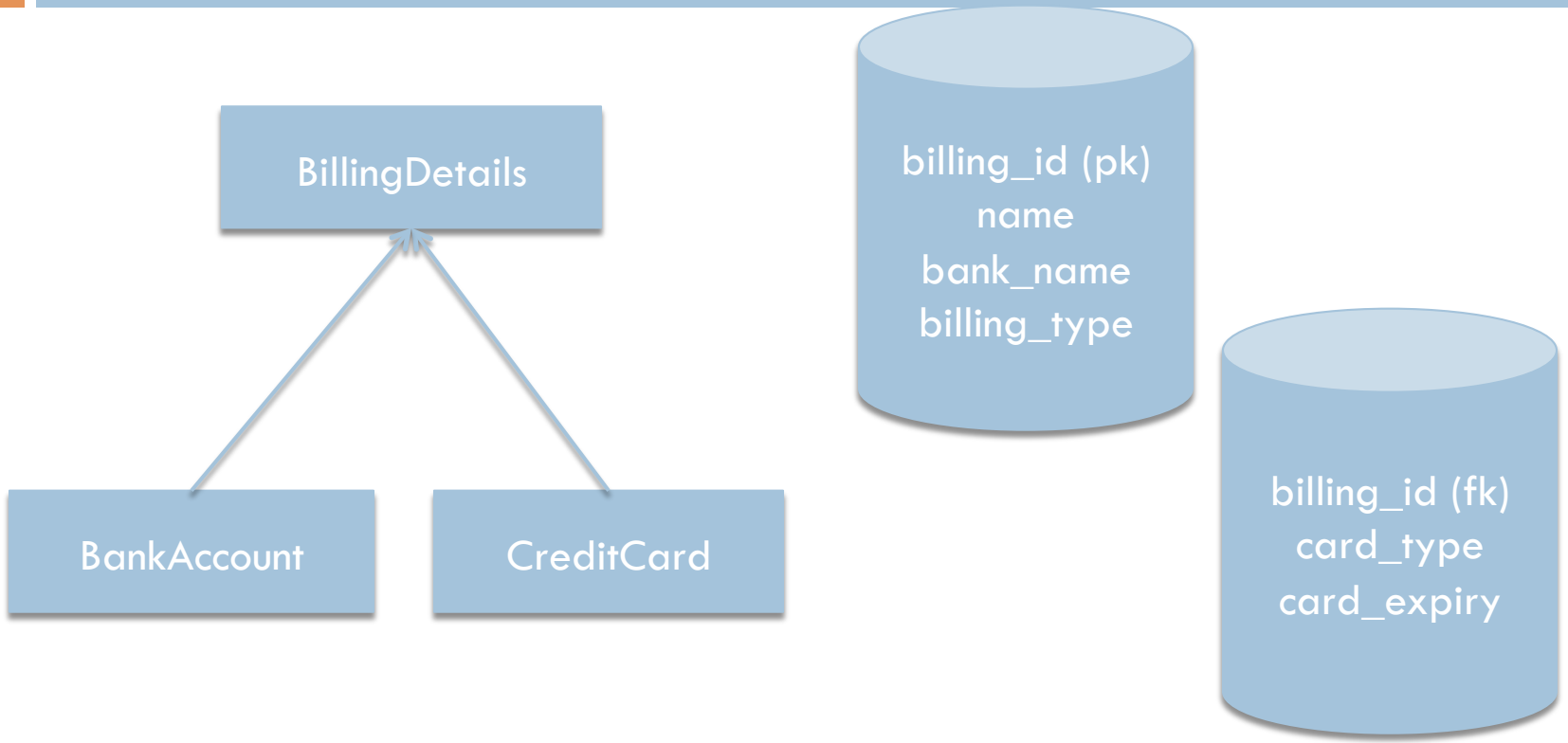
Configuration


```
<class name="BillingDetails" table="billing_details_4">
 <id name="id" column="billing_id" type="int">
  <generator class="increment" />
 </id>
 <property name="owner" />
 <property name="number" column="no" />

 <joined-subclass name="CreditCard" table="creditcard_details_4">
  <key column="billing_id" />
  <property name="type" column="card_type" />
  <property name="expiryMonth" column="expiry_month" />
  <property name="expiryYear" column="expiry_year" />
 </joined-subclass>

 <joined-subclass name="BankAccount" table="bankaccount_details_4">
  <key column="billing_id" />
  <property name="bankName" column="bank_name" />
 </joined-subclass>
</class>
```

3. Table per subclass w/discriminator

- With the help of a discriminator column in the parent table, it is possible that we can mix the previous two strategies
- In this example, we will have one common parent table, a separate table for CreditCard, but for BankAccount let's merge the information in the parent table itself

Overview



Lab 03 / hibernate / inheritance / tablepersubclass / discriminator / BillingDetails.hbm.xml

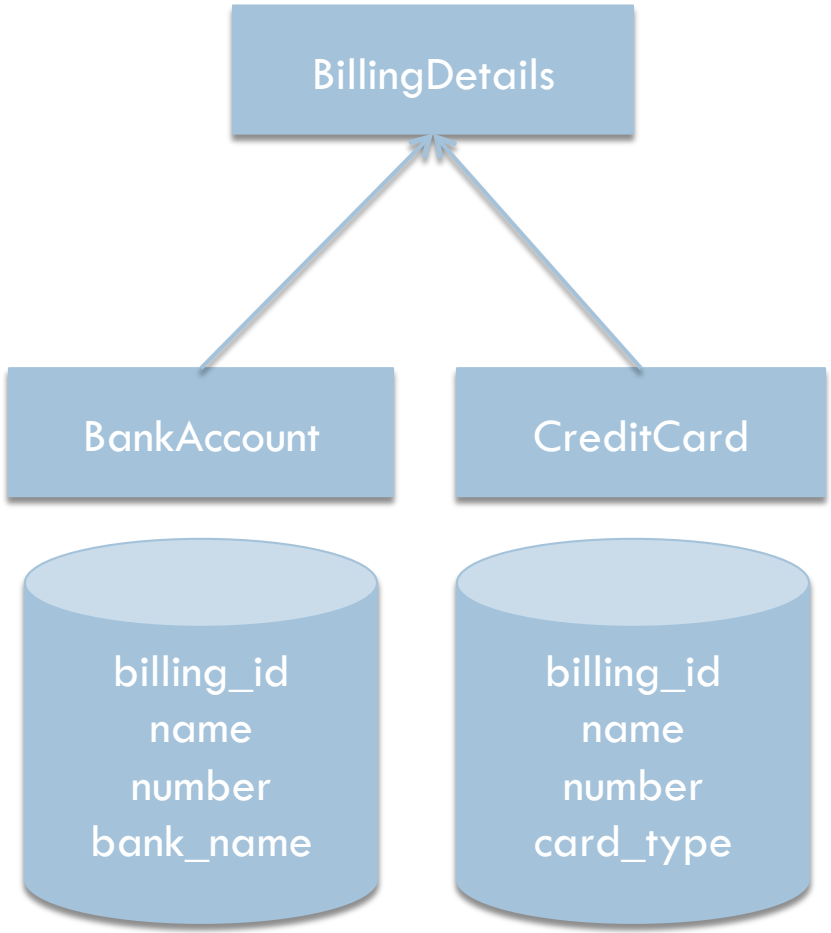
Configuration


```
<class name="BillingDetails" table="billing_details_5">
 <id name="id" column="billing_id" type="int">
  <generator class="native" />
 </id>

 <discriminator column="billing_type" />
 <property name="owner" />
 <property name="number" column="no" />

 <subclass name="CreditCard" discriminator-value="CC">
  <join table="credit_card_details_5">
   <key column="billing_id" />
    <property name="type" column="card_type" />
    <property name="expiryMonth" column="expiry_month" />
    <property name="expiryYear" column="expiry_year" />
  </join>
 </subclass>

 <subclass name="BankAccount" discriminator-value="BA">
  <property name="bankName" column="bank_name" />
</subclass>
</class>
```

4. Separate table per concrete class

- In this strategy, we will have separate tables for both the subclasses in the database, there won't be any parent table, the common fields will be repeated in both the tables

Overview



Lab 03 / hibernate / inheritance / tableperconcreteclass / BillingDetails.hbm.xml

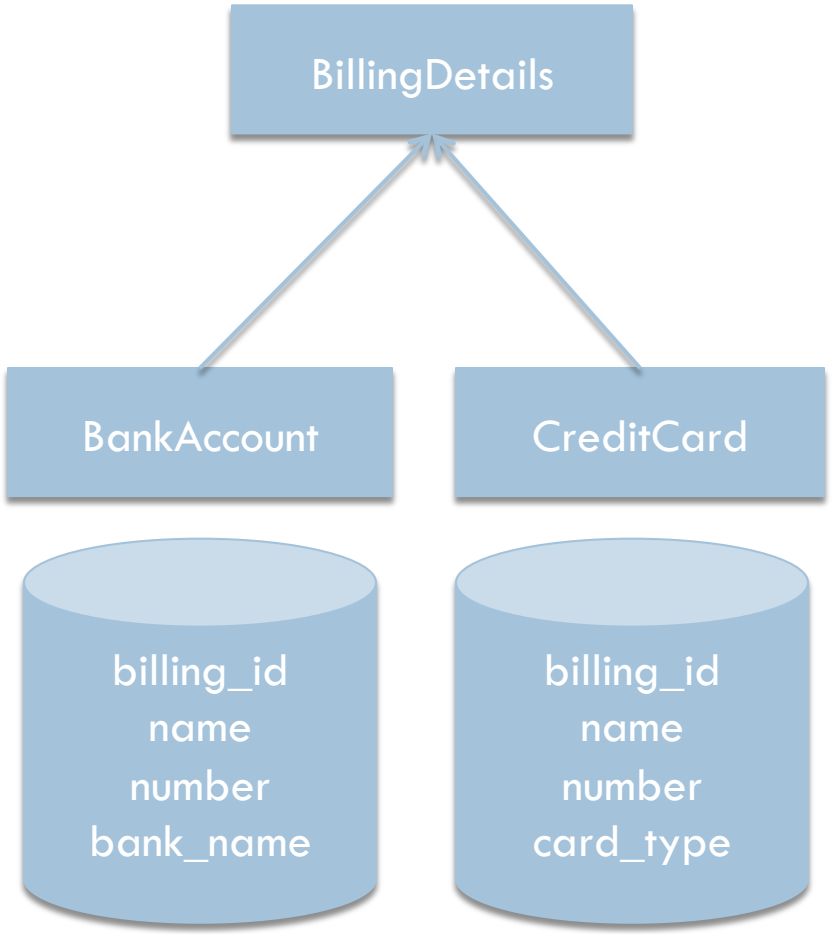
Configuration


```
<class name="BankAccount" table="bankaccount_details_2">
        <id name=“id”>
           <generator class=“increment” />
        </id>
        <property name="owner" />
        <property name="number" column="acno" />
        <property name="bankName" column="bank_name" />
</class>

<class name="CreditCard" table="creditcard_details_2">
        <id name=“id”>
           <generator class=“increment” />
        </id>
        <property name="owner" />
        <property name="number" column="card_no" />
        <property name="type" column="card_type" />
</class>
```

5. Sep. table per concrete class using union

- In this strategy also, we will have separate tables for both the subclasses in the database, there won't be any parent table, the common fields will be repeated in both the tables. The only difference is that hibernate will use SQL union to perform respective operations in the database

Overview



Lab 03 / hibernate / inheritance / tableperconcreteclass / union / BillingDetails.hbm.xml

Configuration


```
<class name="BillingDetails" abstract="true">
 <id name="id" column="billing_id" type="int">
  <generator class="increment" />
 </id>

 <property name="owner" />
 <property name="number" column="no" />

 <union-subclass name="CreditCard" table="creditcard_details_3">
  <property name="type" column="card_type" />
  <property name="expiryMonth" column="expiry_month" />
  <property name="expiryYear" column="expiry_year" />
 </union-subclass>

 <union-subclass name="BankAccount" table="bankaccount_details_3">
  <property name="bankName" column="bank_name" />
 </union-subclass>
</class>
```

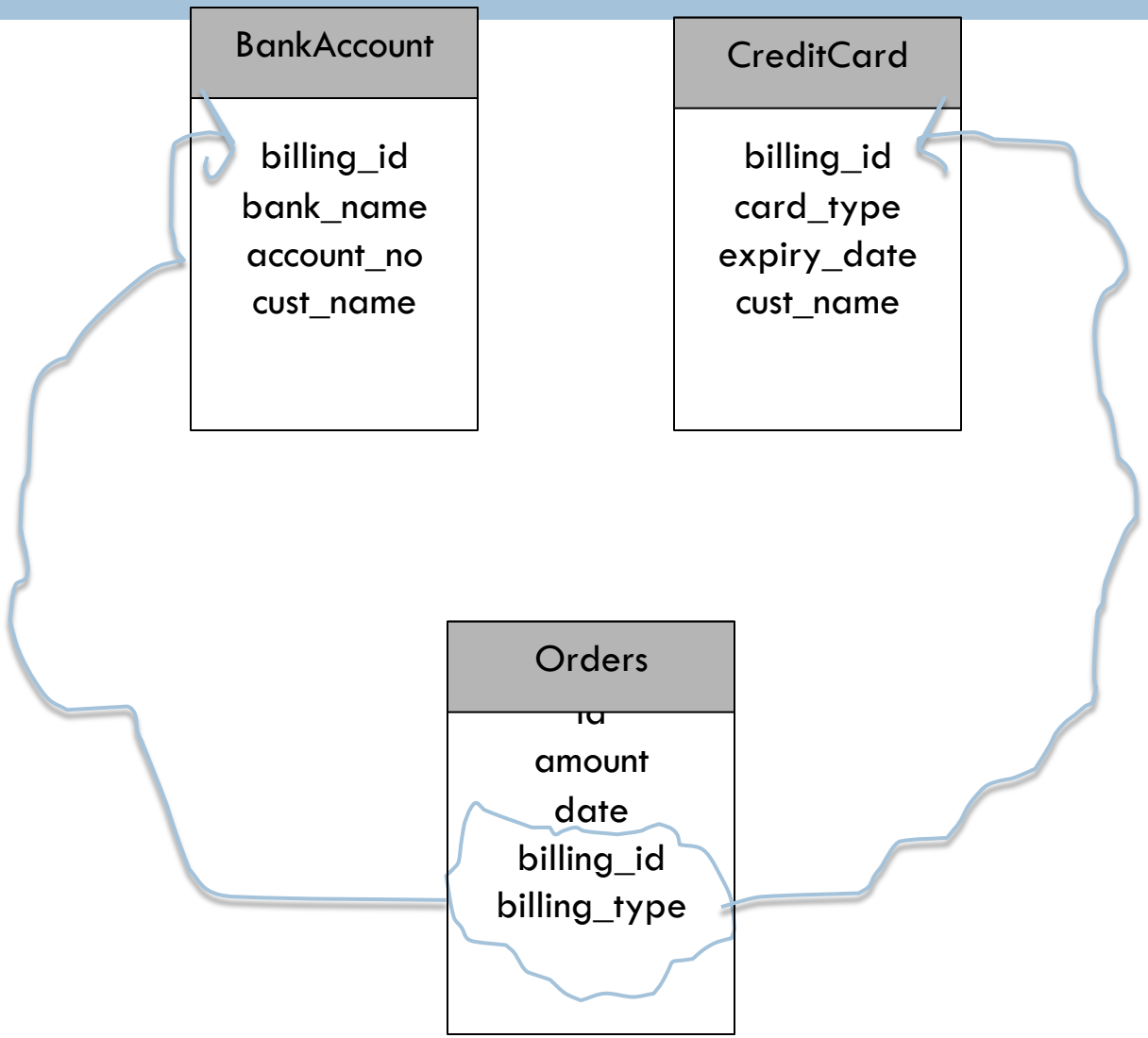
Associations

- Next we will see how we can map associated entities in Hibernate like one-to-many, many-to-many and go on
- Since we saw examples on inheritance, let's first complete how we can achieve polymorphic association in Hibernate
- We will use <any> and <many-to-any> tags in Hibernate for polymorphic association

any and many-to-any association

- **any** represents a polymorphism one-to-one association
  - Example: Order-> BillingDetails (BankAccount/ CreditCard)
- **many-to-any** represents a polymorphic many-to-many association
  - Customer -> Subscription (Magazine/OnlineService)

any assocation



Example


```
public class Order {

        private int id;
        private Date orderDate;
        private double amount;

        //polymorphic association
        private BillingDetails billingDetails;
```

Configuration

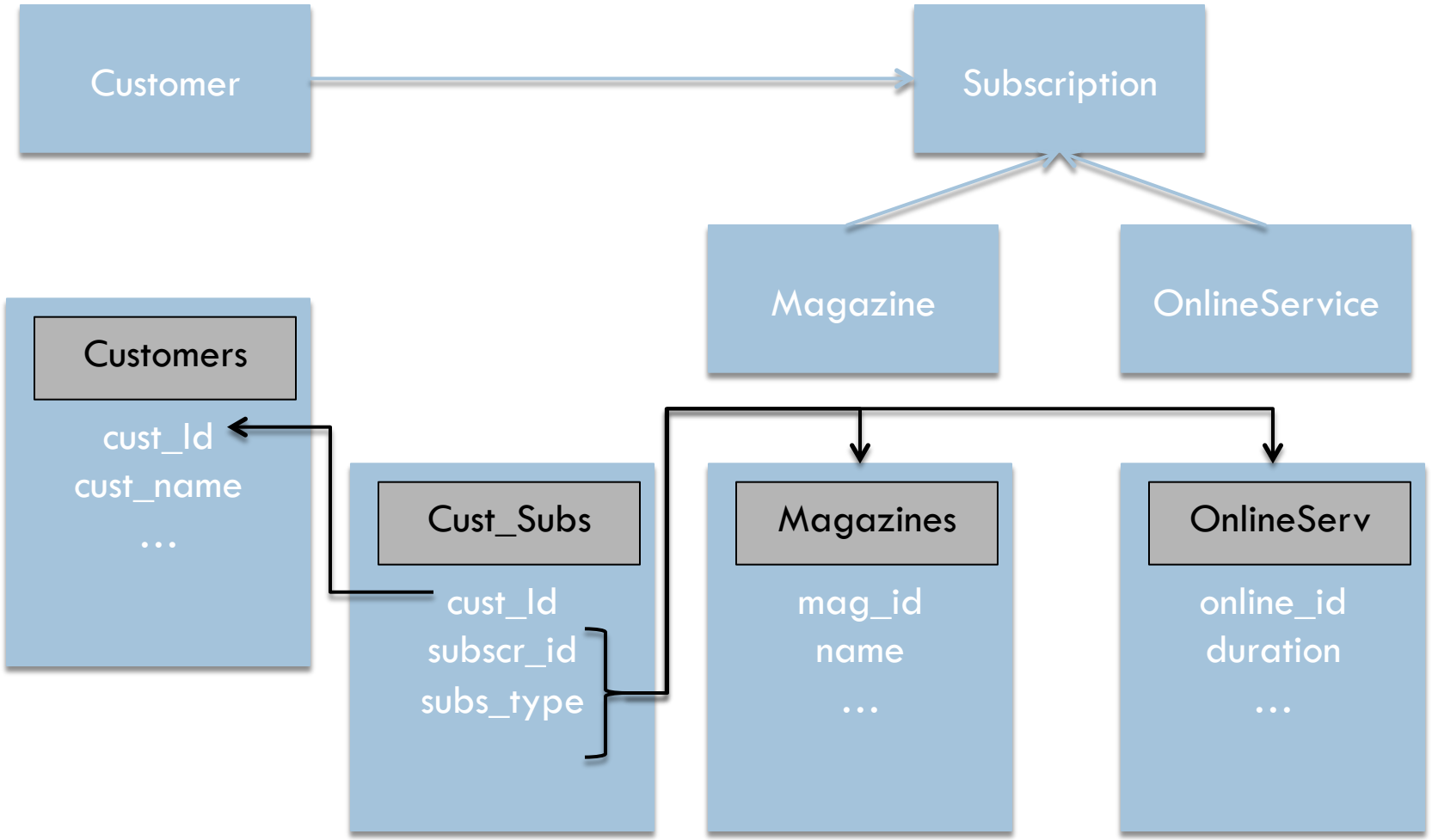

```
<class name="Order" table="orders_123">
 <id name="id" column="order_id">
  <generator class="increment" />
 </id>

 <any name="billingDetails"
  meta-type="string" id-type="int" cascade="save-update">
   <meta-value value="BA" class="BankAccount" />
   <meta-value value="CC" class="CreditCard" />
   <column name="billing_type" />
   <column name="billing_id" />
 </any>

 <property name="orderDate" />
 <property name="amount" />
</class>
```

Lab 03 / hibernate / anyone / Order.hbm.xml

Cont'd…


```
<class name="BankAccount" table="bankaccount_details_2">
        <id name="id">
           <generator class="increment" />
        </id>
        <property name="owner" />
        <property name="number" column="acno" />
        <property name="bankName" column="bank_name" />
</class>

<class name="CreditCard" table="creditcard_details_2">
        <id name="id">
           <generator class="increment" />
        </id>
        <property name="owner" />
        <property name="number" column="card_no" />
        <property name="type" column="card_type" />
</class>
```

many to any association



Configuration

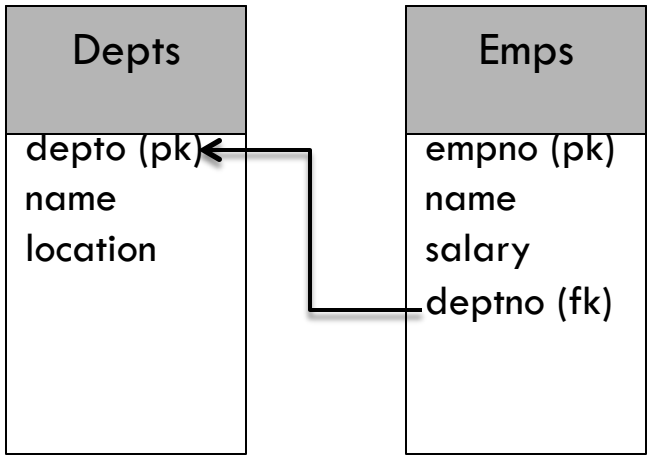

```
<class name="Customer" table="customers_007">
 <id name="id" column="cust_id">
  <generator class="increment" />
 </id>

 <set name="subscriptions" table="customer_subscriptions”
         cascade="save-update">
  <key column="cust_id" />
  <many-to-any id-type="int" meta-type="string">
   <meta-value value="M" class="Magazine" />
   <meta-value value="O" class="OnlineService" />
   <column name="subscription_type" />
   <column name="subscription_id" />
   </many-to-any>
 </set>

 <property name="name" />

</class>
```

Lab 03 / hibernate / manytoany / Customer.hbm.xml

one to many association

- The most common form of association is the *one to many* association. For ex: Customer->Order, Order->LineItem, Department->Employee, etc…
- All forms of association, *one-to-one*, *one-to-many*, *many-to-many* can be represented in an unidirectional as well as bidirectional fashion when writing the mapping classes
- Generally projects prefer bidirectional association

one to many association

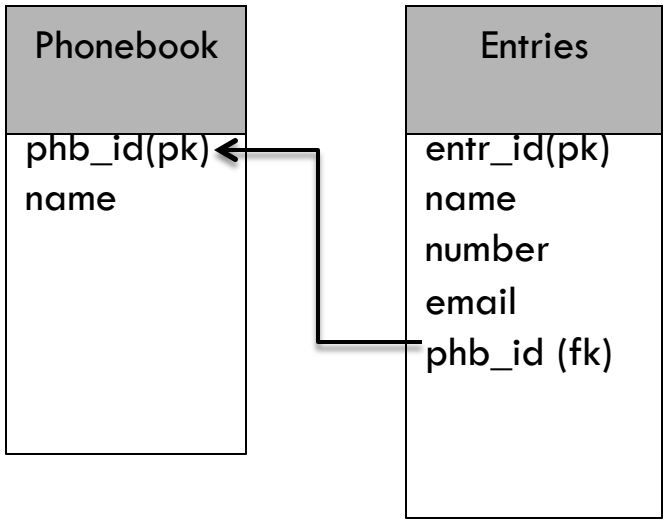


In this example, Department and Employee class represent a bi-directional *one-to-many* association

```
public class Department {

 private int deptno;
 private String name;
 private String location;
 private Set<Employee> employees;
```

```
public class Employee {

 private int empno;
 private String name;
 private double salary;
 private Department dept;
```

one to many association



In this example, PhoneBook and Entry class represent a uni-directional *one-to-many* association

```
public class PhoneBook {

 private int id;
 private String name;
 private Set<Entry> entries;
```

```
public class Entry {

 private int id;
 private String name;
 private String number;
 private String email;
```

Phonebook-Entry association

- So what's wrong with the Phonebook-Entry association?
- Let's take one use case at a time and find out what code Hibernate generates to understand the issues one by one

The configuration


```
<class name="PhoneBook" table="phonebook">
    <id name="id">
       <generator class="increment" />
    </id>

        <set name="entries" cascade="save-update" inverse="false">
                  <key column="phonebook_id" />
                  <one-to-many class="Entry" />
        </set>

        <property name="name" type="string" />
</class>

<class name="Entry" table="entries">
        <id name="id" length="5">
                  <generator class="increment" />
        </id>
        <property name="name" />
        <property name="number" column="phoneNumber" />
        <property name="email" />
</class>
```

inverse="false" means it's a unidirectional association

Name of the foreign key column in the child i.e entries table

Lab 03 / hibernate / onetomanyuni/ PhoneBook.hbm.xml

Adding PhoneBook along with some Entries


```
PhoneBook phBook = new PhoneBook();
phBook.setName("My PhoneBook");

Set<Entry> entries = new HashSet<Entry>();
entries.add(new Entry("Entry1", 12345, "entry1@domain.com"));
entries.add(new Entry("Entry2", 12345, "entry1@domain.com"));
phBook.setEntries(entries);

session.save(phBook);
```

Generated SQL by Hibernate

```
insert into phonebook (name, id) values (?, ?)
insert into entries (name, phoneNumber, email, id) values (?, ?, ?, ?)
insert into entries (name, phoneNumber, email, id) values (?, ?, ?, ?)
update entries set phonebook_id=? where id=?
update entries set phonebook_id=? where id=?
```

Adding a new Entry to an existing PhoneBook


```
Entry newEntry = new Entry("New Entry", 123456, ”entry@www.com");

PhoneBook phBook = (PhoneBook) session.get(PhoneBook.class, 1);
phBook.getEntries().add(newEntry);
```

Generated SQL by Hibernate

select phonebook0_.id as id0_0_, phonebook0_.name as name0_0_ from phonebook phonebook0_ where phonebook0_.id=?

select entries0_.phonebook_id as phonebook5_0_1_, entries0_.id as id1_, entries0_.id as id1_0_, entries0_.name as name1_0_, entries0_.phoneNumber as phoneNum3_1_0_, from entries entries0_ where entries0_.phonebook_id=?

insert into entries (name, phoneNumber, email, id) values (?, ?, ?, ?)
update entries set phonebook_id=? where id=?

Problem with detached objects

- The major problem is when handling detached objects. Let's take an example to make it clear
- Consider that we have a PhoneBook in the database along with some entries so the following code is what will work:

```
PhoneBook phBook = (PhoneBook) session.get(PhoneBook.class, 1);
phBook.getEntries().iterator();
```

- Because of lazy loading, if we don't load the entries for the Phonebook before detaching it, we will get *LazyInitializationException*

Cont'd…


```
Entry newEntry = new Entry(”Very New Entry", 123456,
                                                ”entry@www.com");
phBook.getEntries().add(newEntry);
```

- Since the Phonebook object is detached, the above entry which we have added is transient, i.e. it hasn't yet been persisted, we need to reattach to save the changes
- So next step is to open a Session, invoke saveOrUpdate/merge method to save the Phonebook graph in the database
- Let's see, what happens if we try to update a PhoneBook which already contained 5 entries and we added a 6th one

Generated SQL

- select phonebook0_.id as id0_0_, phonebook0_.name as name0_0_ from phonebook phonebook0_ where phonebook0_.id=?
- select entries0_.phonebook_id as phonebook5_0_1_, entries0_.id as id1_, entries0_.id as id1_0_, entries0_.name as name1_0_, entries0_.phoneNumber as phoneNum3_1_0_, entries0_.email as email1_0_ from entries entries0_ where entries0_.phonebook_id=?
- insert hibernate.onetomanyuni.Entry */ insert into entries (name, phoneNumber, email, id) values (?, ?, ?, ?)
- update hibernate.onetomanyuni.PhoneBook */ update phonebook set name=? where id=?
- update hibernate.onetomanyuni.Entry */ update entries set name=?, phoneNumber=?, email=? where id=?
- update hibernate.onetomanyuni.Entry */ update entries set name=?, phoneNumber=?, email=? where id=?
- update hibernate.onetomanyuni.Entry */ update entries set name=?, phoneNumber=?, email=? where id=?
- update hibernate.onetomanyuni.Entry */ update entries set name=?, phoneNumber=?, email=? where id=?
- hibernate.onetomanyuni.PhoneBook.entries */ update entries set phonebook_id=? where id=?

Some recommendations

- We understood from our discussion the problems in navigating from parent to child when using an ORM. So the best option is to keep our association bi-directional and always navigate from child to parent for all common operations
- Yes, we can use the parent end of the association for fetching purpose, for ex: phBook.getEntries(), will return all the entries associated with this phonebook. A bi-directional association will help in writing HQL/Criteria join queries comfortably
- We can use cascade="none" so that developers will not be able to use the association for any other reason except fetch

Bidirectional mapping


```
<class name="Department" table="depts">
 <id name="deptno">
  <generator class="assigned" />
 </id>

 <set name="employees" cascade="delete" inverse="true">
  <key column="deptno" on-delete="cascade" />
  <one-to-many class="Employee" />
 </set>

 <property name="name" type="string" />
 <property name="location" type="string" />
</class>

<class name="Employee" table="emps">
 <id name="empno">
  <generator class="assigned" />
 </id>
 <many-to-one name="dept" column="deptno" not-null="true" />
 <property name="name" />
 <property name="salary" />
</class>
```

Lab 03 / hibernate / onetomanybi / Employee.hbm.xml

Lab Session

- Time to spend some time on your own to recollect all that we discussed
- *Duration: 40 mins*

More examples

- The examples which I have provided you are as follows:
  - *many-to-many* association
    - Lab 03 / hibernate / manytomany package
  - *one-to-one* association
    - Lab 03 / hibernate / onetoone package
  - *component and dynamic-component* example
    - Lab 03 / hibernate / component package
  - *join* tag example
    - Lab 03 / hibernate / join package
- Similarly, you will also find JPA equivalent examples in the same Lab directory

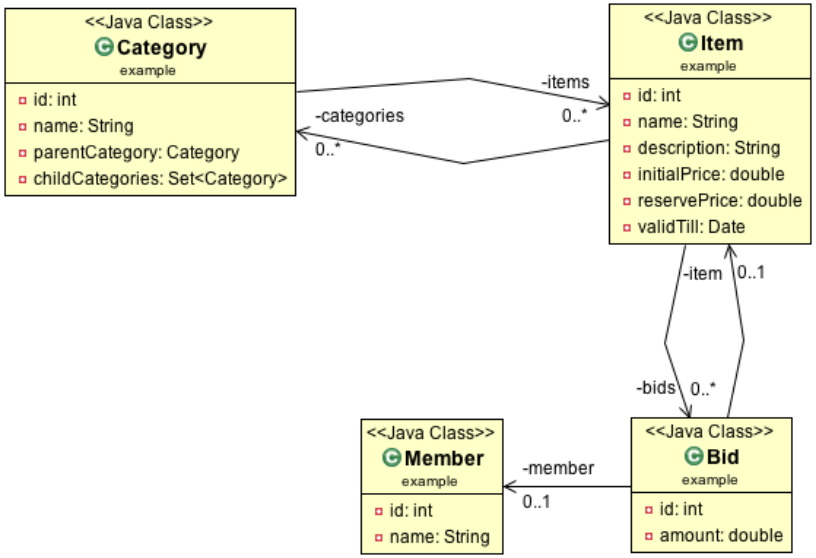
Lab 04 - Agenda

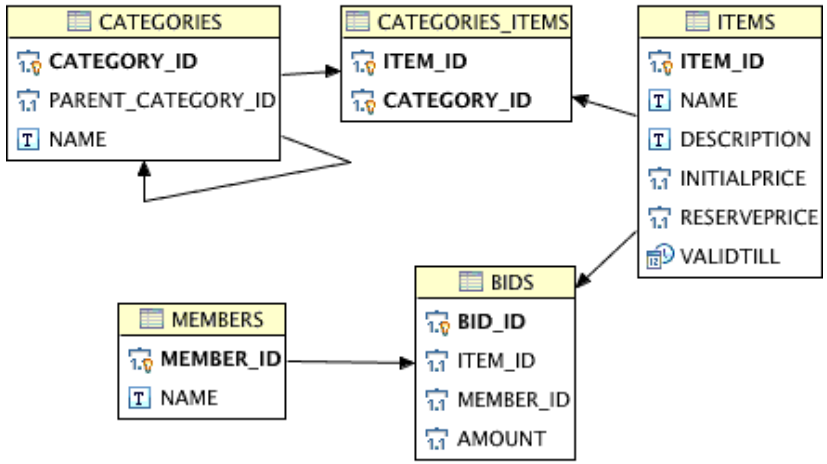
- Understanding different fetching strategies
- Understanding support for executing select queries in Hibernate
- Exploring HQL (Hibernate Query Language) and Criteria API
- Support for different fetching strategies

Fetching strategies

- Hibernate supports different ways of fetching object graph
  - Lazy fetching
  - Eager fetching
  - Outer Join fetching
  - Batch fetching
  - Sub-select fetching
  - Immediate fetching

Example Overview





Lazy fetching

- By default all forms of associations are lazily loaded by default except one-to-one. For ex:

```
<class name="Item" table="items”>
 <set name="categories" table="categories_items" cascade="all">
  <key column="item_id" />
  <many-to-many column="category_id" class="Category" />
 </set>
 <set name="bids” cascade="all">
  <key column="item_id" />
  <one-to-many class="Bid" />
</set>
…
```

- So any query executed on *Item* class won't result in the corresponding *bids* or *categories* information getting loaded

Eager fetching

- We need to specify *lazy="false"* to enable eager loading. For ex:

```
<class name="Item" table="items">
 <set name="categories" lazy="false" table="categories_items">
  <key column="item_id" />
  <many-to-many column="category_id" class="Category" />
 </set>
 <set name="bids" lazy="false">
  <key column="item_id" />
  <one-to-many class="Bid" />
</set>
…
```

- So any query executed on *Item* class will result in the corresponding *bids* and *categories* information getting loaded

Outer join fetching

- By performing an outer join, Hibernate can fetch the Item along with it's bids and categories information in a single query
- For ex:

```
<class name="Item" table="items”>
 <set name="bids” fetch=“join”>
  <key column="item_id" />
  <one-to-many class="Bid" />
</set>
…
```

- So any query executed on *Item* class will result in the corresponding *bids* information getting loaded without a separate SQL being generated

Introducing HQL

- HQL allows developers to write queries transparent to the differences which arises when using different databases
- HQL leverages the same syntax of SQL so learning a new QL doesn't requires lot of time
- HQL queries directly return collection of objects, so there is no need to worry about resultset translation
- For ex:
  - Select item from Item as item where item.initialPrice > 10000

Cont'd…

- The following statement will return all the items from the database in form of objects of Item class

```
String queryString = "from Item";
List list = session.createQuery(queryString).list();
```

- Now let's see some more fetching strategies supported by Hibernate. So what happens if we keep *lazy="false"* for one of the association of Item class?

N+1 problem


```
<class name="Item" table="items”>
 <set name="bids” lazy=“false”>
  <key column="item_id" />
  <one-to-many class="Bid" />
</set>
…
```

select * from items item0_
select * from bids where item_id = ?
select * from bids where item_id = ?
select * from bids where item_id = ?
…

So for each item loaded, hibernate has to fetch the corresponding bids for that item from the database. This is also called as (n+1) problem

Solution

- There are two options to solve this problem, one is by using *batch-size="n"* value for the association and the other is by using *fetch="subselect"*
- Let's see what happens when we use these options

Using batch-size attribute


```
<class name="Item" table="items”>
 <set name="bids” lazy=“false” batch-size=“10”>
  <key column="item_id" />
  <one-to-many class="Bid" />
</set>
…
```

select * from items item0_
select * from bids where item_id in (?,?,?,?,?,?,?,?,?,?)
select * from bids where item_id in (?,?,?,?,?)
…

So if we fire a query on items table and the query returns 15 records, hibernate will instead of generating 15 separate selects to get the bids for each item, will get it in just 2 hits

Using subselect


```
<class name="Item" table="items”>
 <set name="bids” lazy=“false” fetch=“subselect”>
  <key column="item_id" />
  <one-to-many class="Bid" />
</set>
…
```

select * from items item0_
select * from bids where item_id in (select item_id from items)

In this case, for the no. of items loaded based on the where condition if any, hibernate will perform a subselect to get the bids for all the items currently loaded in the memory

Pagination support

- Both the Query as well as Criteria API provide the following methods for supporting pagination:

```
String queryString3 = "from Item";
Query query = session.createQuery(queryString3);
query.setFirstResult(0);
query.setMaxResults(10);
List list = query.list();
Iterator itr = list.iterator();
displayItem(itr,queryString);
```

- So instead of loading all the items, we are asking the database to return us only 10 items. Hibernate with the help of the Dialect class, will generate the appropriate SQL to fetch the records

Joins

- select distinct i from Item i join i.bids b
  - The above HQL will perform an inner join on the bids association of an Item
- select distinct i from Item i join fetch i.bids b
  - A "fetch" join allows associations or collections of values to be initialized along with their parent objects using a single select
- HQL complies to ANSI-SQL standard, so expect all the common features available to developers

Named Queries

- Named Queries allow queries to be externalized into the configuration metadata, so as to facilitate parsing of queries on startup rather than when the query is executed

```
<query name="example.items.getAll”>
 <![CDATA[
   select distinct i from example.Item i join fetch i.bids b where i.id = ?
  ]]>
</query>
```

- To execute a named query:

```
Query q = session.getNamedQuery("example.items.getAll");
q.setInteger(0, 1);
List items = q.list();
displayItem(items.iterator(),"");
```

Criteria API

- Criteria API is used for building queries dynamically as compared to HQL where queries are framed as String objects

```
Criteria criteria = session.createCriteria(Item.class);
criteria.add(Restrictions.like( "name","S%"));
List list = criteria.list();
Iterator itr = list.iterator();
displayItem(itr);
```

- The above criteria will generate the necessary SQL required for fetch only those items whose name matches the like expression

Cont'd…


```
Criteria criteria = session.createCriteria(Item.class);
criteria.add(Restrictions.gt("initialPrice", new Double(100.0)));
criteria.createCriteria("categories");
criteria.add(Restrictions.like("name","E%"));
List list = criteria.list();
Iterator itr = list.iterator();
displayItem(itr);
```

- In the above criteria, we are joining the *categories* association with further restrictions

Projections

- Projections allow us to fetch scalar values rather than entity objects

```
Criteria criteria = session.createCriteria(Item.class);
criteria.setProjection(Projections.projectionList()
                .add(Projections.property("name"))
                .add(Projections.property("initialPrice")));
List results = criteria.list();
Iterator itr = results.iterator();
while(itr.hasNext()) {
                Object[] result = (Object[]) itr.next();
                System.out.println(Arrays.toString(result));
}
```

Lab Session

- Please spend some time to revisit the different features we discussed
- *Duration: 60 mins*

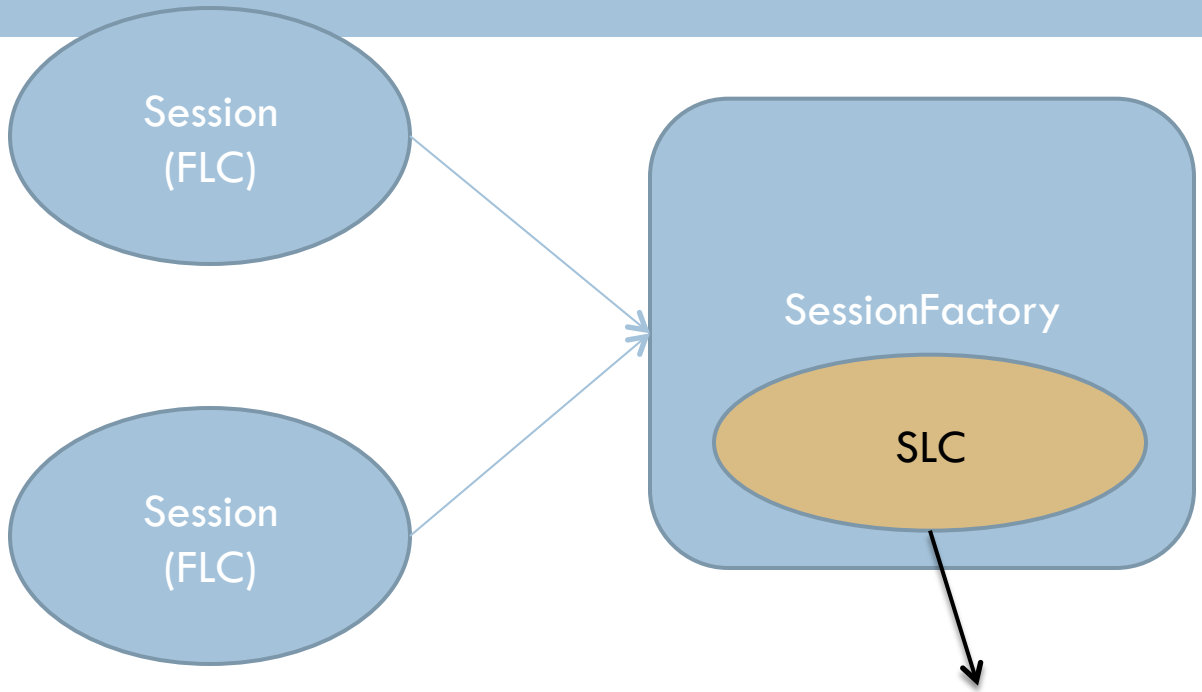
Lab 05 - Agenda

- Understanding support for Caching in Hibernate
- Using EhCache for caching data

Caching support in Hibernate

- There are two levels of cache in Hibernate
  - Transactional Cache
    - Also called as First Level Cache(FLC) implemented by Hibernate Session
  - Global Cache
    - Also called as Second Level Cache(SLC) implemented by Cache providers plugged with Hibernate SessionFactory

Cont'd…



Settings for enabling SLC

```
hibernate.cache.use_second_level_cache = true
hibernate.cache.region.factory_class =
        org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory
hibernate.cache.use_query_cache = true
```

Cont'd…

- Whenever you pass an object to save(), update() or saveOrUpdate(), and whenever you retrieve an object using load(), get(), list(), iterate() or scroll(), that object is added to the internal cache of the Session(FLC)
- If SLC caching has been enabled, FLC contents would be written to the SLC
- Following cache modes are supported
  - read only
  - read write
  - nonstrict read write
  - transactional (for JTA)

Enabling caching of entities


```
<class name="Cat" table="cats">
  <cache usage="read-only" />
  <id name="id" column="cat_id" type="int">
     <generator class="increment" />
  </id>
…
```

- In the above configuration we have enabled a *read-only* cache for Cat instances

Query Caching

- When executing queries using HQL/Criteria, the result won't be cached even though we might have enabled caching for that entity till we don't explicitly enable query caching

```
List cats = session.createQuery("from Cat")
                   .setCacheable(true).list();
```

- In the above query, we are manually enabling caching for the same

Statistics API

- Using the *Statistics* and *Cache* API, we can collect all the details we need for profiling and identifying cache contents

```
Map cacheEntries = sessionFactory.getStatistics()
                    .getSecondLevelCacheStatistics("example.Cat")
                    .getEntries();
```

- The above piece of code will give us how many instances of Cat are present in the memory
- JPA style, Cache interface has been introduced to provide support for standard method naming conventions

Example


```
SecondLevelCacheStatistics stats = sessionFactory.getStatistics().
                                        getSecondLevelCacheStatistics("example.Cat");

System.out.println("No of objects in the cache : "+stats.getElementCountInMemory());
System.out.println("Approx memory occupied : "+stats.getSizeInMemory());

Cache slc = sessionFactory.getCache();
slc.evictEntityRegion("example.Cat");

stats = sessionFactory.getStatistics().getSecondLevelCacheStatistics("example.Cat");
System.out.println("No of objects in the cache : "+stats.getElementCountInMemory());
System.out.println("Approx memory occupied : "+stats.getSizeInMemory());
```

Lab Session

- With the help of a small lab, we will see how well Hibernate manages the cache, and do experiment with the different options available once again
- *Duration: 30 mins*

Lab 6 - Agenda

- Discussing how to handle concurrency issues in Hibernate
- Hibernate support for locking
- Versioning and timestamping support

Locking Support in Hibernate

- We are looking out for ways by which we can prevent concurrent updates at the same time
- Locking of the row is a way by which we can easily achieve the same
- Hibernate supports both the forms of locking:
  - Optimistic Locking (relies on version/timestamp column)
  - Pessimistic Locking (relies on database to manage row level locks)

Optimistic Locking

update products set name = ?, price = ?,
version = 2 where id = 99 and version = 1

ID	NAME	PRICE	VERSION
99	LAPTOP	99999	1

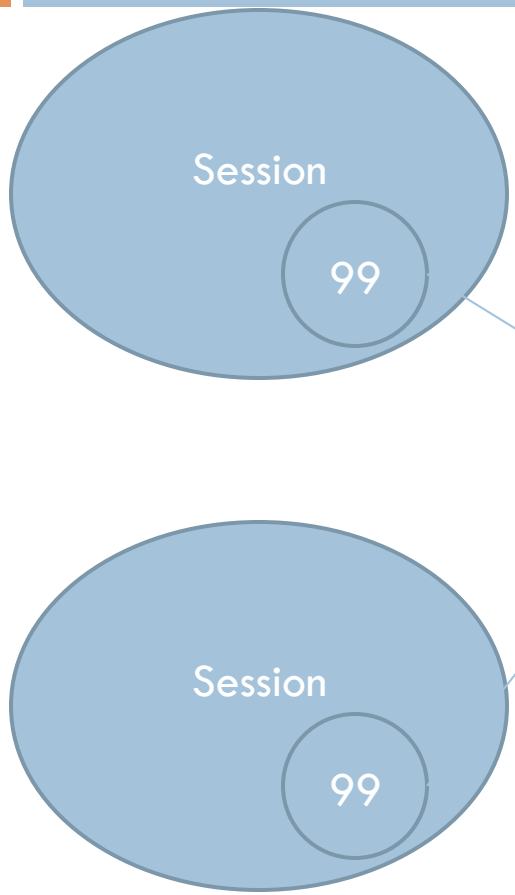
Optimistic Locking

update products set name = ?, price = ?,
version = 2 where id = 99 and version = 1

ID	NAME	PRICE	VERSION
99	LAPTOP	99999	2

Error!

Pessimistic Locking



select * from products where id = 99 for update/
update no wait

ID	NAME	PRICE
99	LAPTOP	99999

Different Lock Modes

- OPTIMISTIC
  - Optimisticly assume that transaction will not experience contention for entities. The entity version will be verified near the transaction end
- OPTIMISTIC_FORCE_INCREMENT
  - Optimisticly assume that transaction will not experience contention for entities. The entity version will be verified and incremented near the transaction end.
- PESSIMISTIC_FORCE_INCREMENT
  - Immediately increment the version of the entity
- PESSIMISTIC_READ
  - Requesting the database to provide a shared lock, which means other transactions will be able to read the values from only the current transaction can modify the same
- PESSIMISTIC_WRITE
  - Requesting the database to provide an exclusive read/write lock to the current transaction. Other transactions will have to wait to even read the data till the current transaction is not completed.

Conclusion

contactme@majrul.com

Thanks a lot! See you next time then!