

Object-Relational Mapping (ORM)

Ehab Ababneh

A decorative graphic consisting of several horizontal lines of varying lengths and colors (teal and white) extending from the right side of the slide.

Outline

- Introduction
- Hibernate
 - A first Example
 - Hibernate's Architecture
 - Hibernate's Main Classes
 - Mapping entities' relationships (i.e. entities to objects)
 - Mapping OO relationships (i.e. classes to entities)
 - Mapping associations
 - Mapping Containers
 - Mapping inheritance

Introduction

- Objects are the basic building blocks for software systems.
- And, generally speaking, a software system need to persist its state.
- Relational data model is the most prevalent (Oracle, MySQL, SQLServer, etc.).
- Bridge the gap between different paradigms.

Introduction

- Not only that, but it also provides data query and retrieval facilities and can significantly reduce development time otherwise spent with manual data handling in SQL and JDBC
- We will be looking at Hibernate as an example.

Hibernate

A decorative graphic consisting of a solid teal horizontal bar that spans the width of the slide. Below this bar, on the right side, there are several horizontal lines of varying lengths and colors, including teal and white, creating a layered, modern look.

Hibernate

- Hibernate is free.
- Hibernate is an open source project.
- Hibernate currently is in release 3.6.8. But a fifth CR of 4.0.0 is out as well.
- Founder and current project leader is Gavin King.

Part I - A First example

Configuring Hibernate, Storing and retrieving objects

A First Example

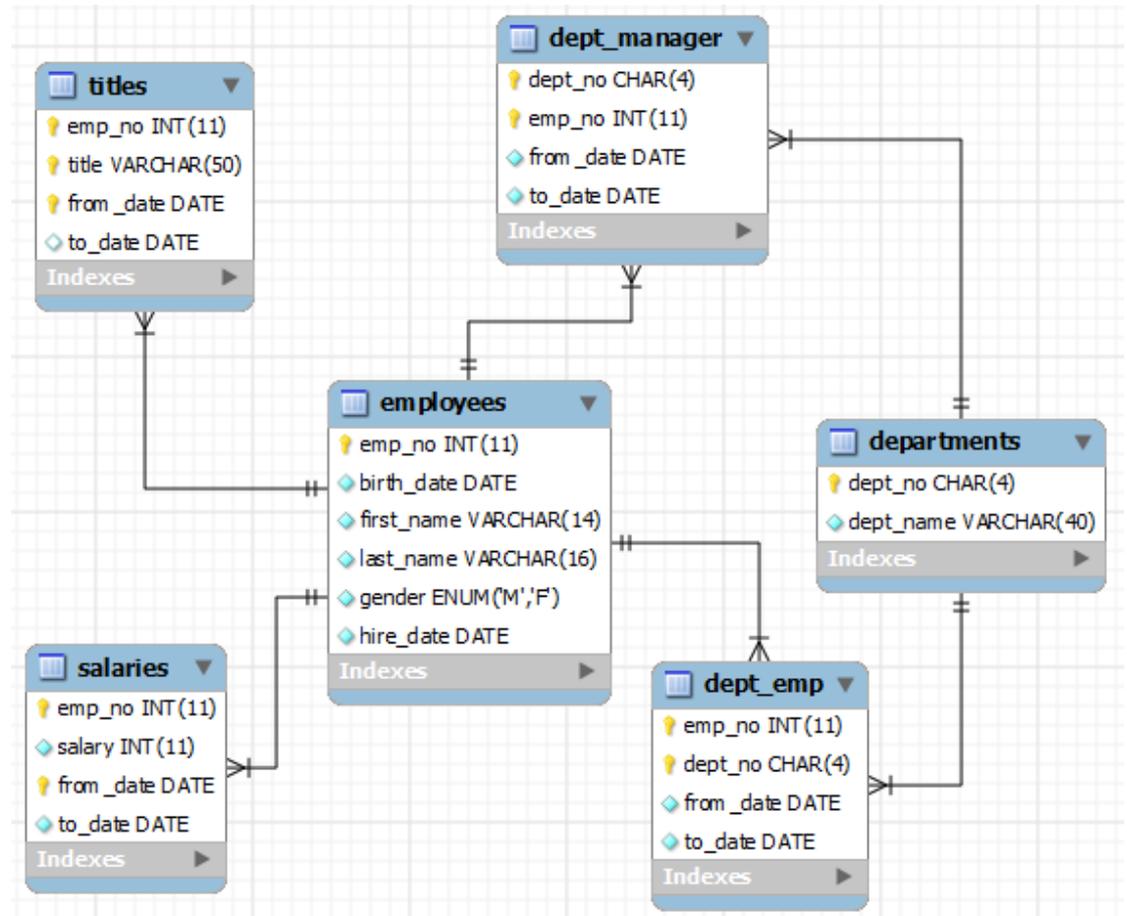
- Persist an object
- Retrieve an object

The Setup

- The database: employees database from Launchpad.
 - The database: http://launchpadlibrarian.net/24493586/employees_db-full-1.0.6.tar.bz2
 - Installation Instructions: <http://dev.mysql.com/doc/employee/en/employee.html#employees-installation>
- Hibernate runtime, which can be downloaded from SourceForge.
- JPA from "lib\jpa" within the hibernate distribution.
- Jar files in the "lib\required" folder in hibernate's distribution.
- Self4j
- MySqlConnectionJ.

The Setup

- Schema



A First Example

- What do we need in order to persist and retrieve an object?

A First Example

- A way to tell Hibernate about the database
- Our persistent class
- A way to tell Hibernate how to map the persistent class to the database table(s)
- The main program to glue thing together

A First Example

- Hibernate Configuration file

- File name is "hibernate.cfg.xml" and the file itself is in the class path.
- Lines 8-10 specify that we are using a mysql JDBC driver. So, it must be in the class path.
- Lines 11-13 specify the connection URL to the database. The database name is "employees" and mysql is running on localhost on default port.
- Lines 14-15 are database user account information.
- Line 18 tells hibernate to use mysql's flavor of SQL or "dialect". Other dialects are listed later on.
- Line 21 specifies a Class-to-Table mapping file (shown later).

```
1 <?xml version='1.0' encoding='utf-8'?>
2 <!DOCTYPE hibernate-configuration PUBLIC
3 "-//Hibernate/Hibernate Configuration DTD//EN"
4 "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
5
6 <hibernate-configuration>
7   <session-factory>
8     <property name="hibernate.connection.driver_class">
9       com.mysql.jdbc.Driver
10    </property>
11    <property name="hibernate.connection.url">
12      jdbc:mysql://localhost/employees
13    </property>
14    <property name="hibernate.connection.username">root</property>
15    <property name="hibernate.connection.password"></property>
16    <property name="hibernate.connection.pool_size">10</property>
17    <property name="show_sql">>true</property>
18    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
19    <property name="hibernate.hbm2ddl.auto">update</property>
20    <!-- Mapping files -->
21    <mapping resource="employee.hbm.xml"/>
22
23   </session-factory>
24 </hibernate-configuration>
```

A First Example

- Persistence Class
 - This is the java class that is to be mapped to a database table.
 - The persistent class follows the JavaBean standard (no-arg constructor in addition to a setter and getter for each mapped attribute).

```
1 package hibernatetutorial.entities;
2
3 import java.util.Date;
4
5 public class Employee {
6
7     String firstName;
8     String lastName;
9     Date birthDate;
10    Date hireDate;
11    char gender;

    // setters and getters
    .....
60 }
```

A First Example

- Mapping File
 - Line 5 links the class to be mapped and the database table.
 - Lines 6-8 specify the identifier property of the persistent class objects which is "employeeNumber" and map that property to the database column "emp_no" in the employees database table. The lines also specify that it is the table's primary key and the method by which it is generated (in this case it is "assigned") other methods for primary key generation are listed later.
 - Honor the camelCase style. Notice that in the mapping file a property "firstName" is specified and class Employee has the methods setFirstName(String) and getFirstName().

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping
DTD 3.0//EN"
3     "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
4 <hibernate-mapping>
5   <class name="hibernatetutorial.entities.Employee"
table="employees">
6     <id column="emp_no" name="employeeNumber">
7       <generator class="assigned"/>
8     </id>
9     <property name="firstName">
10      <column name="first_name"/>
11    </property>
12    <property name="lastName">
13      <column name="last_name"/>
14    </property>
15    <property name="hireDate">
16      <column name="hire_date"/>
17    </property>
18    <property name="birthDate">
19      <column name="birth_date"/>
20    </property>
21    <property name="gender">
22      <column name="gender"/>
23    </property>
24  </class>
25 </hibernate-mapping>
```

A First Example

- Main Program
 - Lines 23-24 instantiate a SessionFactory object. This is where the hibernate.cfg.xml is read. This basically tells Hibernate how to find the database.
 - Line 27 calls on the SessionFactory to get a session object. The session object is the interface between our application and Hibernate. So, if we wanted to ask Hibernate to do something for us we do it through this session object.
 - Line 29 starts a transaction.
 - Lines 38-44 instantiate an object of the persistent class and give values for its attributes.
 - Line 47 saves the object in the session object. That means that the object is a live in the session's persistent store.
 - Line 48 commits the transaction and the object is now saved to the database.

```
23     sessionFactory =
24         new Configuration().configure().buildSessionFactory();
25
26     // get a session
27     Session session = sessionFactory.openSession();
28
29     Transaction tx = session.beginTransaction();
30
31     //Create new instance of Employee and set values in it.
32     System.out.println("Inserting Record ...");
33
34     Calendar thirtyYears = Calendar.getInstance();
35     thirtyYears.add(Calendar.YEAR, -30);
36     thirtyYears.getTime();
37
38     Employee employee = new Employee();
39     employee.setEmployeeNumber(199999);
40     employee.setBirthDate(thirtyYears.getTime());
41     employee.setFirstName("Matthew");
42     employee.setLastName("Gheen");
43     employee.setHireDate(new Date());
44     employee.setGender('M');
45
46     // save the newly created object -> db table row.
47     session.save(employee);
48     tx.commit();
49     System.out.println("Done!");
```

Few Notes

- Hibernate Configuration can be specified in an XML file -just like the way we did it, using a hibernate.properties text file or programmatically.
- Hibernate can connect to a database using JDBC just like in our example, or it can use other connection sources like JNDI.
- If you can take a second look at the persistent class and you can see that no extra code has been added to it to handle anything related to persistence. In other words it is just a Plain Old Java Object (POJO) as Martin Fowler would describe it.
- The setter and getter need not be public.
- You can tell Hibernate to bypass setters and getters of any mapped field in the persistent class. And in that case Hibernate would change/get the value persistent class's data member directly. For example you can change the method of field access for the first name property like this:

```
<property name="firstName" access="field" ><column name="first_name"/></property>
```
- Hibernate may need to commit the object created in case the id is generated by the database (i.e. identity). In this case, an explicit rollback is a must in case the object is deemed to be no longer needed.

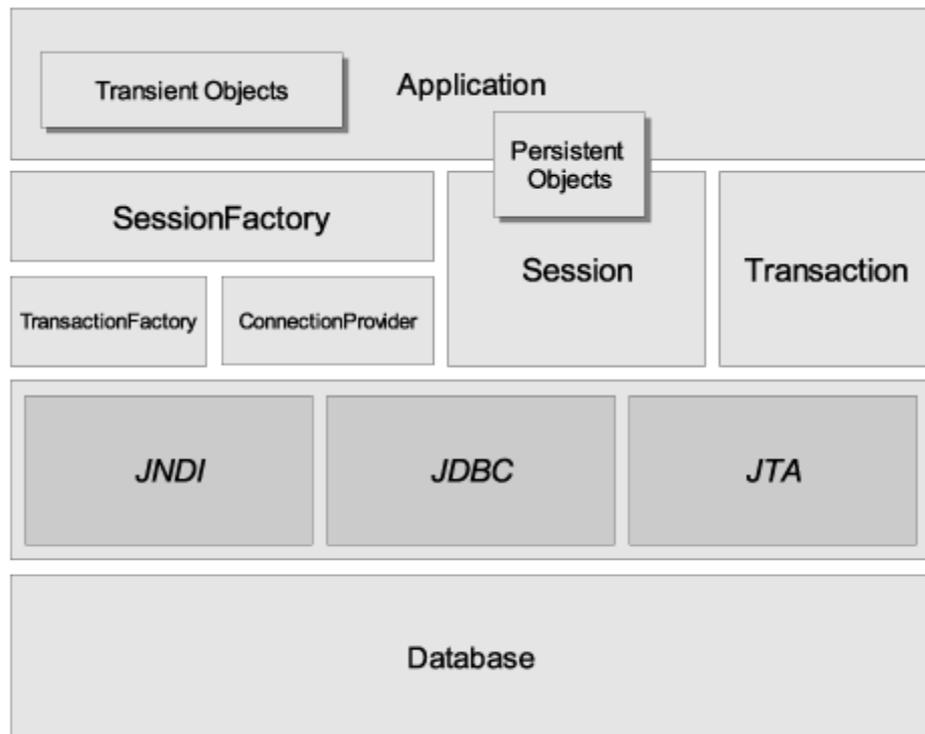
A First Example

- Main Program for querying the database.
 - Lines 27-31 are nothing new
 - Lines 35-38 create a Criteria object and add Restriction objects to build the “WHERE” clause in the SQL query.
 - Hibernate return a List interface with the query results.
 - Lines 41-45 iterate through the list just like any other Java List.

```
27     sessionFactory =
28         new Configuration().configure().buildSessionFactory();
29
30     // get a session
31     Session session = sessionFactory.openSession();
32
33     //session.beginTransaction();
34
35     List employees = session.createCriteria(Employee.class)
36         .add( Restrictions.eq("lastName", "Markovitch"))
37         .add( Restrictions.eq("firstName", "Margareta") )
38         .list();
39
40
41     for (Employee emp : (List<Employee>) employees )
42     {
43         System.out.println(emp.getLastName() + ", "
44             + emp.getFirstName() + " had the titles:");
45     }
```

Architecture Overview

- Layered architecture
- Each database connection in Hibernate is created by creating an instance of Session interface. Session represents a single connection with database. Session objects are created from SessionFactory object.



Hibernate's Main Classes

- **SessionFactory (org.hibernate.SessionFactory)**
 - A thread-safe, immutable cache of compiled mappings for a single database. A factory for org.hibernate.Session instances. A client of org.hibernate.connection.ConnectionProvider. Optionally maintains a second level cache of data that is reusable between transactions at a process or cluster level.
- **Session (org.hibernate.Session)**
 - A single-threaded, short-lived object representing a conversation between the application and the persistent store. Wraps a JDBC java.sql.Connection. Factory for org.hibernate.Transaction. Maintains a first level cache of persistent the application's persistent objects and collections; this cache is used when navigating the object graph or looking up objects by identifier.
- **Persistent objects and collections**
 - Short-lived, single threaded objects containing persistent state and business function. These can be ordinary JavaBeans/POJOs. They are associated with exactly one org.hibernate.Session. Once the org.hibernate.Session is closed, they will be detached and free to use in any application layer (for example, directly as data transfer objects to and from presentation).

Hibernate's Main Classes

- **Transient and detached objects and collections**
 - Instances of persistent classes that are not currently associated with a `org.hibernate.Session`. They may have been instantiated by the application and not yet persisted, or they may have been instantiated by a closed `org.hibernate.Session`.
- **Transaction (`org.hibernate.Transaction`)**
 - (Optional) A single-threaded, short-lived object used by the application to specify atomic units of work. It abstracts the application from the underlying JDBC, JTA or CORBA transaction. A `org.hibernate.Session` might span several `org.hibernate.Transactions` in some cases. However, transaction demarcation, either using the underlying API or `org.hibernate.Transaction`, is never optional.

Hibernate's Main Classes

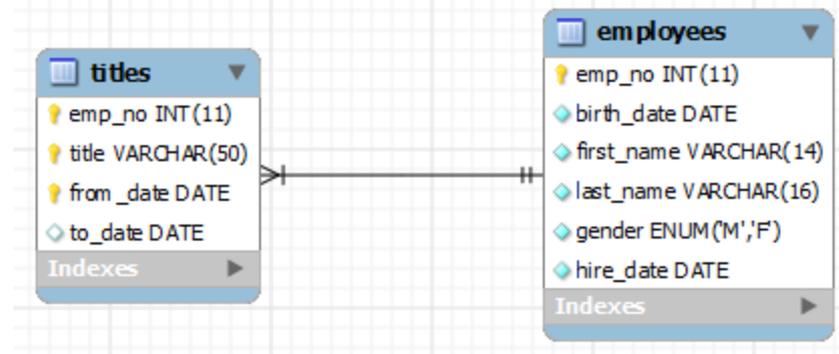
- **ConnectionProvider**
(**org.hibernate.connection.ConnectionProvider**)
 - (Optional) A factory for, and pool of, JDBC connections. It abstracts the application from underlying `javax.sql.DataSource` or `java.sql.DriverManager`. It is not exposed to application, but it can be extended and/or implemented by the developer.
- **TransactionFactory** (**org.hibernate.TransactionFactory**)
 - (Optional) A factory for `org.hibernate.Transaction` instances. It is not exposed to the application, but it can be extended and/or implemented by the developer.

Part II - Mapping Entities

One-to-many, many-to-many

Mapping One-to-Many

- In our example database, an employee may have/had several titles (history data).
- “Titles” has a composite key. The foreign key referencing “employees.id” is part of that composite key.



Mapping One-to-Many

- Checklist for implementing this relationship:
 - The new “Title” persistent class
 - Adding a container of some sort in the Employee class to hold its related Title instances.
 - Mapping file for the “Title” class.
 - Modifying Employee’s mapping to tell Hibernate about the new relationship.
 - Modifications to the main program.

Mapping One-to-Many

- The new “Title” persistent class
 - Another simple POJO i.e. setters, getters, and no-arg constructor.
 - Lines 27-59 The composite id is represented with an inner serializable class.

```
1 package hibernatetutorial.entities;
2
3 import java.io.Serializable;
4 import java.util.Date;
5
6 public class Title {
7
8     Date toDate;
9     Id id;
10
11     // .....
12     // setters and getters for toDate and id.
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27     public static class Id implements Serializable {
28
29         long employeeld;
30         String name;
31         Date fromDate;
32
33         // .....
34         // setters and getters for members in Id class.
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59     }
60 }
```

Mapping One-to-Many

- Adding a container in Employee class to hold its titles.
 - The lines basically add a Set container for holding the instances of Title Class that belong to this instance of Employee. Hibernate will handle populating this container with the appropriate Title instances.

```
private Set titles;  
  
public Set getTitles() {  
    return titles;  
}  
  
public void setTitles(Set titles) {  
    this.titles = titles;  
}
```

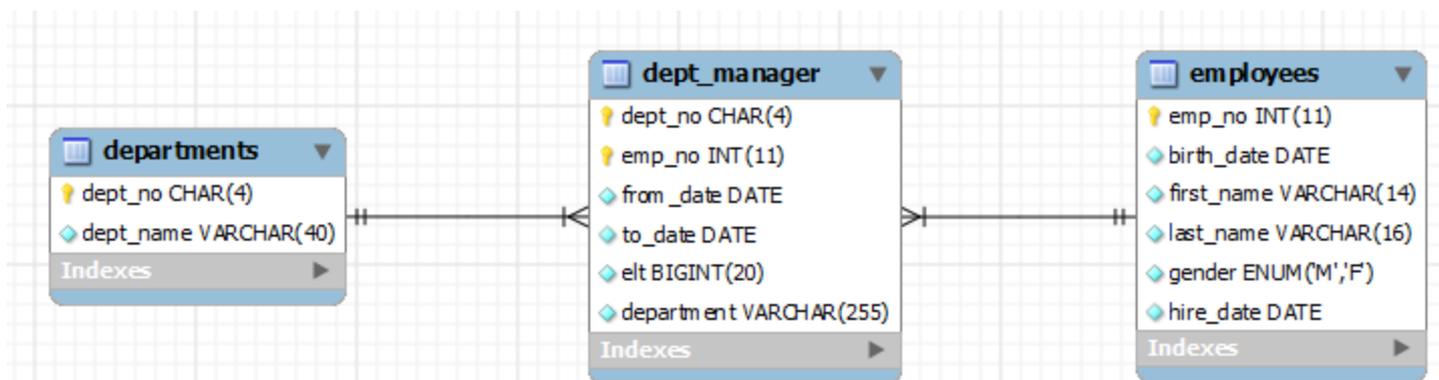
Mapping One-to-Many

- Adding a container in Employee class to hold its titles.
 - Lines 26-29 is the only change from the original employee mapping. These line tell Hibernate to put the instances of the Title class in a Set container within the Employee class.
 - Line 26 tells Hibernate that the continuer name is “titles” and is of type Set. It also tells Hibernate the order by which to fetch rows from the titles table (order-by=”column asc| desc”). And that it should populate the instance once the Employee instance is created. (lazy=”false”). The attributes order-by and lazy are optimal.
 - Line 27 tells Hibernate how to match the appropriate rows from titles table with the appropriate row from the employees table.
 - Line 28 tells Hibernate what class to instantiate an object from when finding a row in titles table related to the instance of this instance of the Employee class.

```
.....  
22 <property name="gender">  
23   <column name="gender"/>  
24 </property>  
25  
26 <set name="titles" order-by="from_date asc" lazy="false">  
27   <key column="emp_no"/>  
28   <one-to-many class="hibernatetutorial.entities.Title"/>  
29 </set>  
30  
31 </class>  
32  
33 </hibernate-mapping>
```

Mapping Many-to-Many

- Many-to-many relationships are usually represented with a separate table with foreign keys to both participating entities.
- The relationship may have additional attributes.



Mapping Many-to-Many

- Mapping many-to-many relationships when the relationship has not additional attributes is relatively easy.
- In this example Hibernate knows that from the <many-to-many> how to match the right departments with the right managers.
- As before, the Department persistent class has nothing unusual. It contains id and name attribute and their setters/getters.
- A Set container “departmentsManaged” is been added to the Employee persistent class to hold the appropriate Department instances.

```
<set name="departmentsManaged" table="dept_manager">  
  <key column="emp_no"/>  
  <many-to-many column="dept_no" class="hibernatetutorial.entities.Department"/>  
</set>
```

Mapping Many-to-Many

- Mapping many-to-many relationships when the relationship has additional attributes requires splitting the relationship into two many-to-one relationships and creating another persistent class that represents the relationship.

```
1 package hibernatetutorial.entities;
2
3 import java.util.Date;
4
5 public class DepartmentManager {
6     Date fromDate;
7     Date toDate;
8     Department department;
9
10    // + setters and getters
11
12 }
31
32
33
34
35
36
37
38
39
40
41
42
43
```

```
<set name="departmentsManaged" table="dept_manager">
  <key column="emp_no"/>
  <composite-element
    class="hibernatetutorial.entities.DepartmentManager">
    <property name="fromDate" type="date"
      column="from_date" not-null="true"/>
    <property name="toDate" type="date"
      column="to_date" not-null="true"/>
    <many-to-one name="department"
      class="hibernatetutorial.entities.Department"
      column="dept_no" not-null="true"/>
    </composite-element>
  </set>
```

Part III - Mapping Objects

Associations, Inheritance

Mapping Associations

- Associations can be represented using the previously mentioned techniques (one-to-one, one-to-many, many-to-one, and many-to-many).
- Mapping these relationships is relatively easy, and can be easier once you get to know how to map different types of containers.
- We have seen an example for mapping a Set of “Titles”.
- We will take a look at mapping another type of container.

Mapping Associations

- Mapping A “Map” Collection:
 - Tag <map> tells Hibernate to map the collection to an implementation of the Map interface.
 - The tag <key> tells Hibernate how to pick the columns from the database table.
 - The tags <map-key> and <one-to-many> tell Hibernate what fields are to be used as a key-value pair for the map.

```
<map name="titles" >  
  <key column="emp_no"/>  
  <map-key column="title" type="string"/>  
  <one-to-many class="hibernatetutorial.entities.Title"/>  
</map>
```

Mapping Associations

- Hibernate supports a wide range of Java collections. The most widely used are Maps, Lists, and Sets.
- Here is a link with many examples and details about mapping collections:

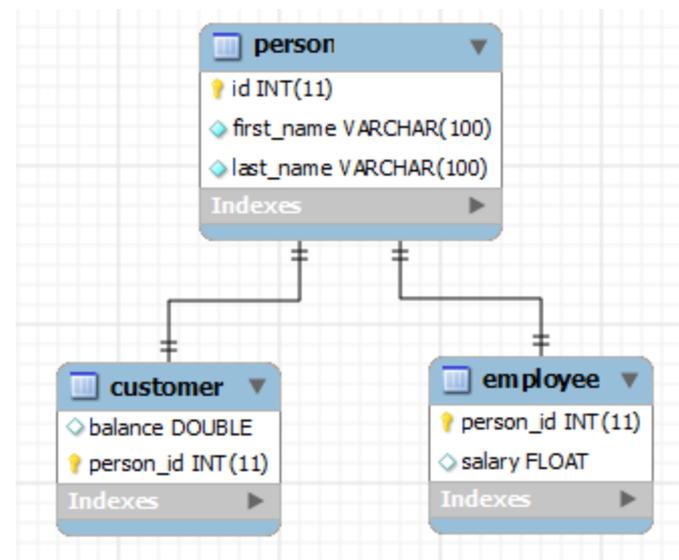
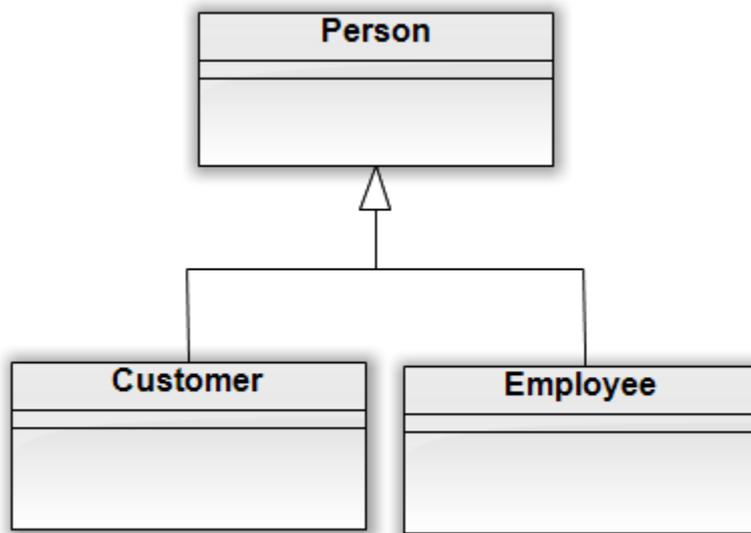
<http://docs.jboss.org/hibernate/core/3.3/reference/en/html/collections.html>

Mapping Inheritance

- Three main strategies for mapping inheritance:
 - One table per class
 - One table per class hierarchy
 - One table per concrete class.

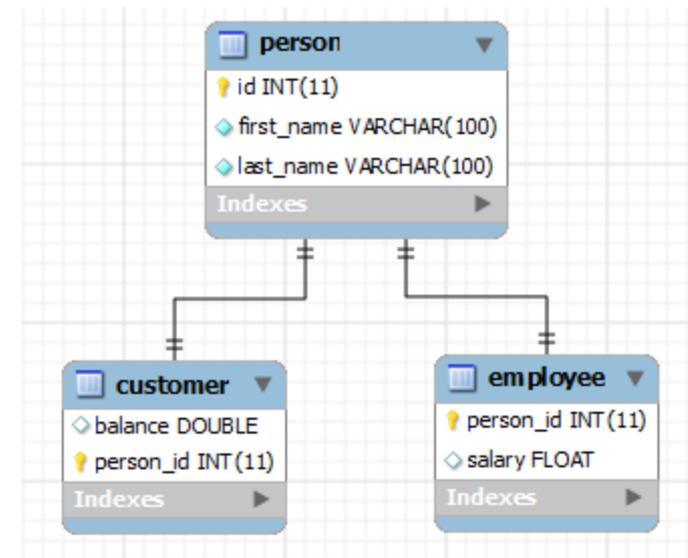
Mapping One-to-Many

- One table per class strategy:



Mapping Inheritance

- One table per class strategy:
 - In this strategy a table is created for each class in the hierarchy. The primary key from the super class is used as a primary key in the sub class as well. The primary in the sub class is also marked as a foreign key that references the primary key in the super class.
 - Notice that there person_id in the customer is used as a primary key in the Customer table as well as a foreign key that references id in the Person table. Likewise in the Employee table.



Mapping Inheritance

- One table per class strategy:
 - The three POJOs

Person persistent class:

```
1 public class Person {
2     long id;
3     String firstName;
4     String lastName;
5
6     // + setters and getters
7 }
```

The Customer persistent class:

```
1 public class Customer extends Person {
2     float balance;
3
4     // + setters and getters
5 }
```

The Employee persistent class:

```
1 public class Employee extends Person {
2
3     float salary;
4
5     // + setters and getters
6 }
```

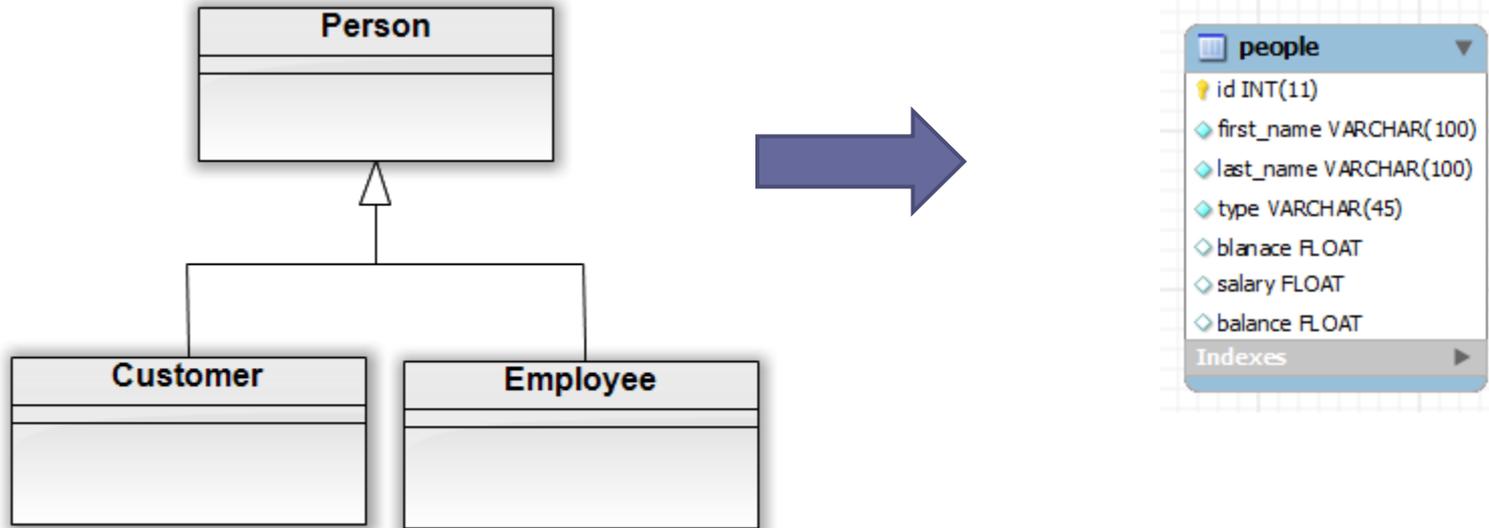
Mapping Inheritance

- One table per class strategy:
 - The mapping file:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate
Mapping DTD 3.0//EN"
3     "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
4 <hibernate-mapping>
5
6 <class name="Person" table="Person">
7   <id name="id" type="long">
8     <generator class="assigned"/>
9   </id>
10  <property name="firstName" column="first_name"/>
11  <property name="lastName" column="last_name"/>
12
13  <joined-subclass name="Employee" table="employee">
14    <key column="person_id"/>
15    <property name="salary" column="salary"/>
16  </joined-subclass>
17  <joined-subclass name="Customer" table="Customer">
18    <key column="person_id"/>
19    <property name="balance" column="balance"/>
20  </joined-subclass>
21 </class>
22
23 </hibernate-mapping>
```

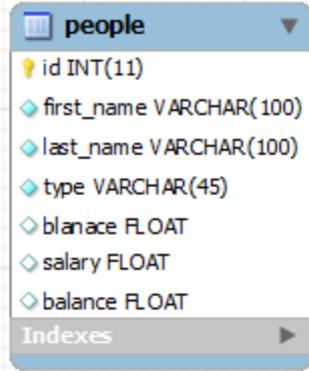
Mapping One-to-Many

- One table per class hierarchy strategy:



Mapping Inheritance

- One table per class strategy:
 - In this strategy all classes in the inheritance are flattened into one table. The created table includes all properties from all classes in the hierarchy in addition to a column that tells which concrete class the current row is an instance of.
 - The column “type” in this table is the discriminator between Employee instances and Customer instances.



A screenshot of a database table named 'people'. The table has the following columns: id (INT(11)), first_name (VARCHAR(100)), last_name (VARCHAR(100)), type (VARCHAR(45)), blance (FLOAT), salary (FLOAT), and balance (FLOAT). There is also an 'Indexes' section at the bottom.

Column Name	Data Type
id	INT(11)
first_name	VARCHAR(100)
last_name	VARCHAR(100)
type	VARCHAR(45)
blance	FLOAT
salary	FLOAT
balance	FLOAT

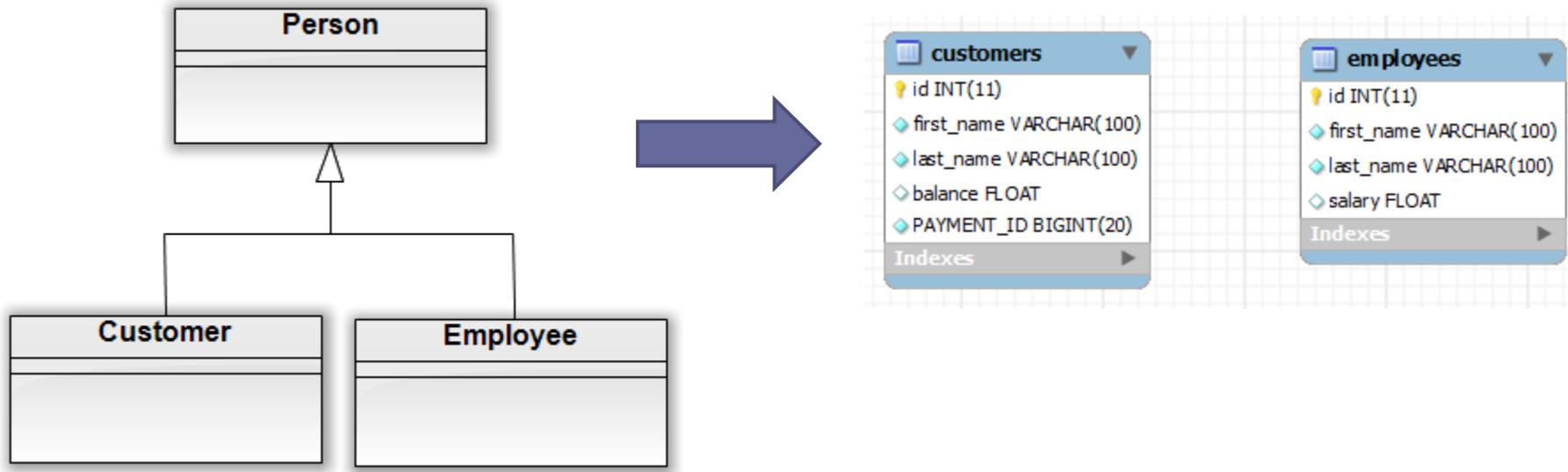
Mapping Inheritance

- One table per concrete class strategy:
 - The mapping file:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate
Mapping DTD 3.0//EN"
3     "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
4 <hibernate-mapping>
5
6 <class name="Person" table="People">
7     <id name="id" type="long" column="id">
8         <generator class="assigned"/>
9     </id>
10    <discriminator column="type" type="string"/>
11    <property name="firstName" column="first_name"/>
12    <property name="lastName" column="last_name"/>
13
14    <subclass name="Customer" discriminator-value="CUSTOMER">
15        <property name="balance" column="balance"/>
16    </subclass>
17    <subclass name="Employee" discriminator-value="EMPLOYEE">
18        <property name="salary" column="salary"/>
19    </subclass>
20 </class>
21 </hibernate-mapping>
```

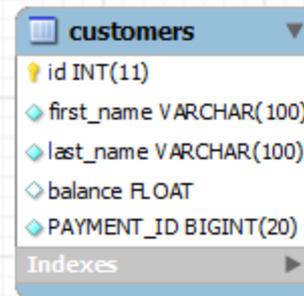
Mapping One-to-Many

- One table per concrete class strategy:



Mapping Inheritance

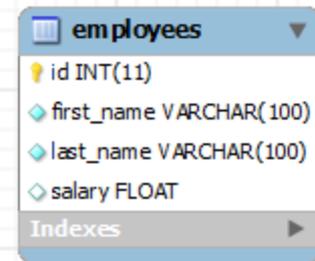
- One table per class strategy:
 - In this strategy a table is created per concrete class and each table contains the attributes of its matching concrete class in addition to the attributes of the base class.
 - Notice that the employees table contains the attributes from the Employees class in addition to the attributes from the Person class.



customers

- id INT(11)
- first_name VARCHAR(100)
- last_name VARCHAR(100)
- balance FLOAT
- PAYMENT_ID BIGINT(20)

Indexes



employees

- id INT(11)
- first_name VARCHAR(100)
- last_name VARCHAR(100)
- salary FLOAT

Indexes

Mapping Inheritance

- One table per class hierarchy strategy:
 - The mapping file:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate
Mapping DTD 3.0//EN"
3     "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
4 <hibernate-mapping>
5
6 <class name="Person">
7   <id name="id" type="long" column="id">
8     <generator class="assigned"/>
9   </id>
10  <property name="firstName" column="first_name"/>
11  <property name="lastName" column="last_name"/>
12
13  <union-subclass name="Customer" table="Customers">
14    <property name="balance" column="balance"/>
15  </union-subclass>
16  <union-subclass name="Employee" table="Employees">
17    <property name="salary" column="salary"/>
18  </union-subclass>
19 </class>
20 </hibernate-mapping>
```