**Hibernate Tutorial**

# Goal

* Learn Hibernate

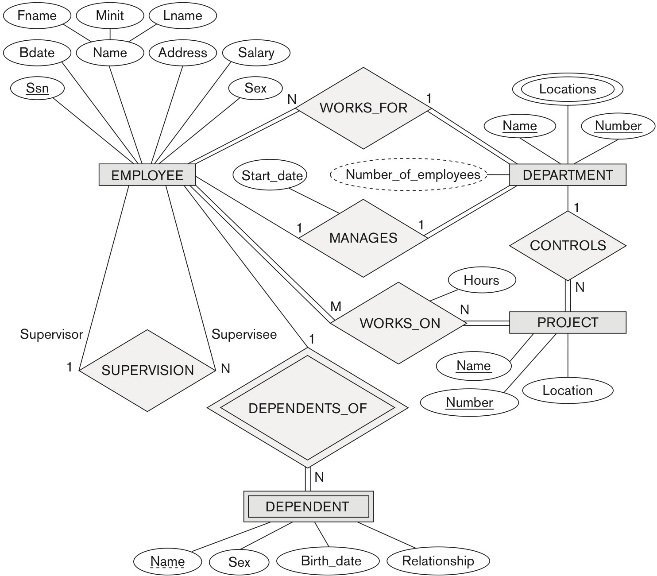
# Motivation

OOP is a natural way to understand the problem domain. In OOP, everything is an object. As a result, ER diagram or UML class diagram are created for conceptual modeling. A conceptual or object model contains entity types (or classes) and relationships.

For example, in the company domain, people need to keep track of a company’s employees, departments, and projects. So, here are some requirements from this domain:

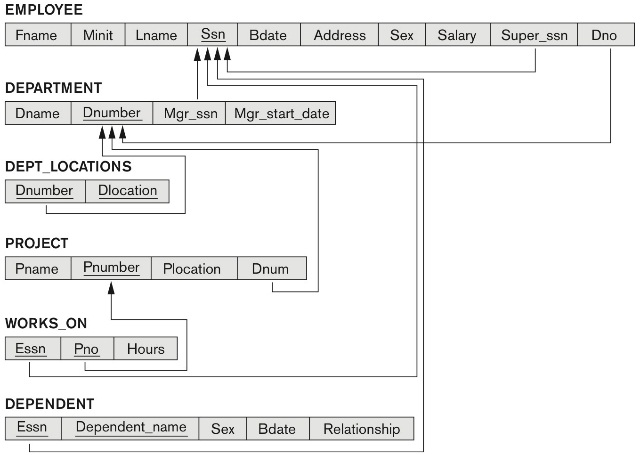
* The company is organized into departments. Each department has a unique name, a unique number, and a particular employee who manages the department. We keep track of the start date when that employee began managing the department.
* A department may have several locations. A department controls a number of projects, each of which has a unique name, a unique number, and a single location.
* We store each employee’s name, Social Security number, address, salary, sex (gender), and birth date. An employee is assigned to one department, but may work on several projects, which are not necessarily controlled by the same department. We keep track of the current number of hours per week that an employee works on each project. We also keep track of the direct supervisor of each employee (who is another employee).
* We want to keep track of the dependents of each employee for insurance purposes. We keep each dependent’s first name, sex, birth date, and relationship to the employee.

After reading the requirements, experienced developers can come up with an object model expressed in ER diagram.



However, the database we are using nowadays are relational database management systems (RDBMS). They cannot store “entities” or “objects,” as a result, developers have to manually convert the ER diagram to tables (Think about your homework and midterm 1 questions).

For example, tables are derived based on the above ER diagram.



## The Object-Relational Impedance Mismatch

When we work with an object-oriented system, there is a mismatch between the object model and the relational database. RDBMSs represent data in a tabular format whereas object-oriented languages, such as Java or C# represent it as an interconnected graph of objects. Check out an example in <https://www.tutorialspoint.com/hibernate/orm_overview.htm>

“Object-Relational Impedance Mismatch” (sometimes called the “paradigm mismatch”) is just a fancy way of saying that object models and relational models do not work very well together. RDBMSs represent data in a tabular format, whereas object-oriented languages, such as Java, represent it as an interconnected graph of objects. Loading and storing graphs of objects using a tabular relational database exposes us to 5 mismatch problems… ​<http://hibernate.org/orm/what-is-an-orm/>

# ORM

ORM stands for Object-Relational Mapping (ORM) is a programming technique for converting data between relational databases and object-oriented programming languages such as Java, C#, etc.

An ORM solution has the following advantages over plain JDBC:

1. Let business code access objects rather than DB tables.
2. Hides details of SQL queries from OO logic.
3. Based on JDBC “under the hood.”
4. No need to deal with the database implementation.
5. Entities based on business concepts rather than database structure.
6. Transaction management and automatic key generation.
7. Fast development of application.

# Hibernate

Hibernate ORM (Hibernate in short) is an object-relational mapping tool for the Java programming language. It provides a framework for mapping an object-oriented domain model to a relational database. Hibernate handles object-relational impedance mismatch problems by replacing direct, persistent database accesses with high-level object handling methods.

Hibernate’s primary feature is:

* mapping from Java classes to database tables, and
* mapping from Java data types to SQL data types.
* provides data query and retrieval facilities.
* generates SQL calls and relieves the developer from the manual handling and object conversion of the result set.

# Set Up Hibernate in Eclipse

You can easily add Hibernate capability for your Java project.

## Download and unzip Hibernate 5.4.9.Final

<https://sourceforge.net/projects/hibernate/files/hibernate-orm/5.4.9.Final/hibernate-release-5.4.9.Final.zip/download>

You will find the following items after unzipping.

A screenshot of a cell phone

Description automatically generated

You will need the JARs in lib folder.

* documentation
* lib
  + required :JARs that must have in your project
  + optional
* project

Under required, you can see those 18 Jars, we call them Hibernate Jars:

A screenshot of a cell phone

Description automatically generated

## Create a new Java project

Add all the following Jars in the project.

* JDBC driver Jar
* Hibernate Jars
* JUnit Jars

A screenshot of a cell phone

Description automatically generated

# Create a Hibernate Core Configuration XML File

A screenshot of a cell phone

Description automatically generated

Hibernate requires knowing in advance — where to find the mapping information that defines how your Java classes relate to the database tables. Hibernate also requires a set of configuration settings related to database and other related parameters. All such information is usually supplied as a standard Java properties file called hibernate.properties, or as an XML file named hibernate.cfg.xml. I will consider XML formatted file hibernate.cfg.xml to specify required Hibernate properties in my examples. Most of the properties take their default values and it is not required to specify them in the property file unless it is really required. This file is kept in the root directory of your application’s classpath.

Hibernate properties

1. hibernate.dialect
   1. This property makes Hibernate generate the appropriate SQL for the chosen database.
2. hibernate.connection.driver\_class
   1. The JDBC driver class.
3. hibernate.connection.url
   1. The JDBC URL to the database instance.
4. hibernate.connection.username
   1. The database username.
5. hibernate.connection.password
   1. The database password.
6. hibernate.connection.pool\_size
   1. Limits the number of connections waiting in the Hibernate database connection pool.
7. hibernate.connection.autocommit
   1. Allows autocommit mode to be used for the JDBC connection.

hibernate.cfg.xml

|  |
| --- |
| <!--  ~ Hibernate, Relational Persistence for Idiomatic Java  ~  ~ License: GNU Lesser General Public License (LGPL), version 2.1 or later.  ~ See the lgpl.txt file in the root directory or <http://www.gnu.org/licenses/lgpl-2.1.html>.  -->  <!DOCTYPE hibernate-configuration PUBLIC  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  <hibernate-configuration>  <session-factory>  <!-- Set up basic connection properties -->  <property name=*"hibernate.connection.driver\_class"*>com.mysql.cj.jdbc.Driver</property>  <property name=*"hibernate.connection.url"*>jdbc:mysql://172.16.197.140/company1?serverTimezone=UTC</property>  <property name=*"hibernate.connection.username"*>bingyang</property>  <property name=*"hibernate.connection.password"*>ABc123456$!</property>  <!-- Configure SQL Dialect -->  <property name=*"hibernate.dialect"*>org.hibernate.dialect.MySQL57Dialect</property>    <!-- The following properties are optional -->  <!-- Print SQL statements to Console -->  <property name=*"show\_sql"*>true</property>  <property name=*"hibernate.format\_sql"*>true</property>  <!-- Create DB Tables automatically from Java Beans -->  <property name=*"hibernate.hbm2ddl.auto"*>create</property>  <!-- optional section ends -->    <!-- Hibernate needs to know the Java Beans -->  <mapping class=*"edu.tcu.cs.company.domain.Employee"*/>  <mapping class=*"edu.tcu.cs.company.domain.Department"*/>  <mapping class=*"edu.tcu.cs.company.domain.Project"*/>  </session-factory>  </hibernate-configuration> |

# Hibernate Annotations for POJOs

One big part of Hibernate is the annotations it provided. I will explain them using examples.

## @Entity, @Table, @Column, @Id, @GeneratedValue and @Version

Every persistent POJO class is declared using the @Entity annotation (at the class level):

|  |
| --- |
| @Entity //@Entity declares the class as an entity  @Table(name="t\_user")//a DB table named t\_user will be created based on this entity  public class User  {  private int id;  private String username;  private String password;  private Date born;  private Date registerDate;  /\*  In DB table, the corresponding column name is register\_date  if you don’t add this (which is fine), the column name is registerDate  \*/  @Column(name="register\_date")  public Date getRegisterDate()  {  return registerDate;  }  public void setRegisterDate(Date registerDate)  {  this.registerDate = registerDate;  }  @Id　　 //@Id declares the PK of the table  @GeneratedValue //This sets the PK to be automated generated  public int getId()  {  return id;  }  public void setId(int id)  {  this.id = id;  }  　　............  } |

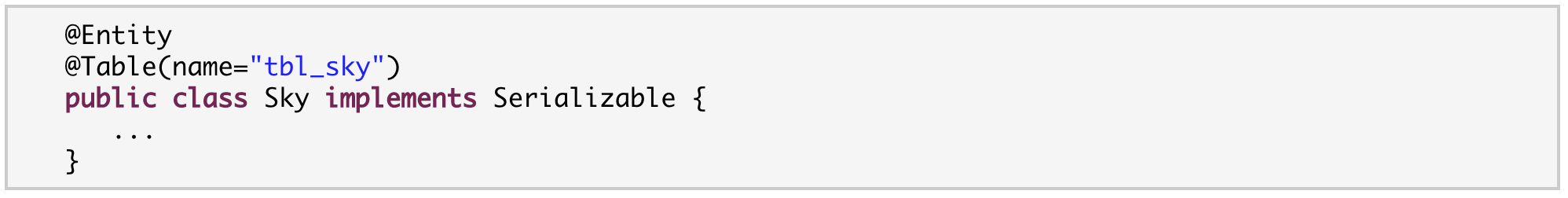
Note: You can either annotate the getter method, or directly annotate the private field. Annotating the getter is recommended.

@Entity marks a class as a persistent class whose objects or instances will be stored in database tables, so this class MUST be a POJO i.e. has a public no-argument constructor, getters and setters for each private field.

@Table allows you to define the table, catalog, and schema names for your entity mapping.

@Table(name = "", catalog = "", schema = "")

If no @Table is defined, the default values are used: the unqualified class name of the entity.



@Column annotation is used to specify the details of the column to which a field or property will be mapped. You can use column annotation with the following most commonly used attributes

* name: attribute permits the name of the column to be explicitly specified.
* length: attribute permits the size of the column used to map a value particularly for a String value.
* nullable: attribute permits the column to be marked NOT NULL when the schema is generated.
* unique: attribute permits the column to be marked as containing only unique values.

You can also add optimistic locking capability to an entity using the @Version annotation:



The version property will be mapped to the OPTLOCK column, and Hibernate will use it to detect conflicting updates (preventing lost updates you might otherwise see with the last-commit-wins strategy).

## Mapping entity associations/relationships

### @OneToOne

You can associate entities through a one-to-one relationship using @OneToOne. There are three cases for OneToOne associations:

1. either the associated entities share the same primary keys values,
2. a foreign key is held by one of the entities (note that this FK column in the database should be constrained unique to simulate OneToOne multiplicity), or
3. an association table is used to store the link between the two entities (a unique constraint has to be defined on each FK to ensure the one to one multiplicity)

#### Shared PK

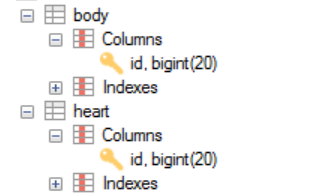
First, we could map a real one-to-one association using shared primary keys: This is a common model to have a dependent object share the primary key of its owner. In the case of a OneToOne the dependent’s primary key is the same as the owner, and in the case of a ManyToOne the dependent’s primary key is composed of the owner’s primary key and another locally unique field (Do you remember in Company DB, every employee can have zero or more dependents?).

For example, one body only has one heart, and one heart only belongs to one body. Here is how we model them using the shared PK.

|  |
| --- |
| @Entity  public class Body {  private Long id;  private Heart heart;  @Id  @GeneratedValue  public Long getId() {  return id;  }  public void setId(Long id) {  this.id = id;  }  @OneToOne  @PrimaryKeyJoinColumn  public Heart getHeart() {  return heart;  }  public void setHeart(Heart heart) {  this.heart = heart;  }  public Body() {  super();  }  } |

|  |
| --- |
| @Entity  public class Heart {  private Long id;  @Id  @GeneratedValue  public Long getId() {  return id;  }  public void setId(Long id) {  this.id = id;  }  public Heart() {  super();  }  } |

The above code creates two tables sharing the same PK.



The OneToOne association is called unidirectional (means you can access Heart from Body object but you cannot access Body from Heart object).

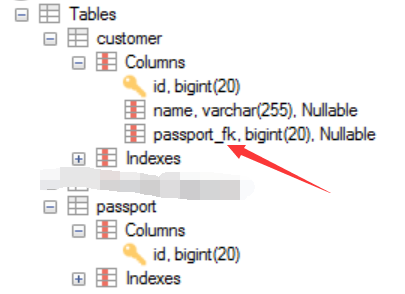
#### FK in one entity

Next, let’s move to the second way to implement OneToOne relationship. In the following example, the associated entities are linked through an explicit foreign key column (Different from the first approach, each entity has its own PK). One customer has only one passport, and one passport only belongs to one customer.

|  |
| --- |
| @Entity  public class Customer {  private Long id;  private String name;  private Passport passport;  @Id  @GeneratedValue  public Long getId() {  return id;  }  public void setId(Long id) {  this.id = id;  }  public Customer() {  super();  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  @OneToOne  @JoinColumn(name = "passport\_fk")  public Passport getPassport() {  return passport;  }  public void setPassport(Passport passport) {  this.passport = passport;  }  } |

|  |
| --- |
| @Entity  public class Passport {  private Long id;  private Customer owner;  @Id  @GeneratedValue  public Long getId() {  return id;  }  public void setId(Long id) {  this.id = id;  }  public Passport() {  super();  }  @OneToOne(mappedBy=“passport”)  public Customer getOwner() {  return owner;  }  public void setOwner(Customer owner) {  this.owner = owner;  }  } |

The above code creates tables like this:



A Customer is linked to a Passport, with a foreign key column named passport\_fk in the corresponding Customer table. The join column is declared with the @JoinColumn annotation which looks like the @Column annotation.

The OneToOne association is called bidirectional (means you can access Passport from Customer object and you can access Customer from Passport object). In a bidirectional relationship, one of the sides has to be the owner: the owner is responsible for the association column(s) update. In this example, we pick Customer to be the owner and responsible for maintaining this relationship. To declare a side as not responsible for the relationship, the attribute mappedBy is used. mappedBy refers to the property name of the association on the owner side (in our case, this is Customer’s passport attribute). As you can see, you don’t have to (must not) declare the join column since it has already been declared on the owner’s side.

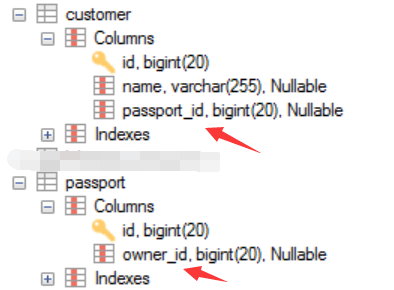
If no @JoinColumn is declared on the owner side, the defaults apply. A join column(s) will be created in the owner table and its name will be the concatenation of the name of the relationship in the owner side, **\_** (underscore), and the name of the primary key column(s) in the owned side. In this example passport\_id because the property name is passport and the column id of Passport is id.

If we take out the mappedBy, what will happen? See the following example.

|  |
| --- |
| @Entity  public class Customer {  private Long id;  private String name;  private Passport passport;  @Id  @GeneratedValue  public Long getId() {  return id;  }  public void setId(Long id) {  this.id = id;  }  public Customer() {  super();  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  @OneToOne  public Passport getPassport() {  return passport;  }  public void setPassport(Passport passport) {  this.passport = passport;  }  } |

|  |
| --- |
| @Entity  public class Passport {  private Long id;  private Customer owner;  @Id  @GeneratedValue  public Long getId() {  return id;  }  public void setId(Long id) {  this.id = id;  }  public Passport() {  super();  }  @OneToOne  public Customer getOwner() {  return owner;  }  public void setOwner(Customer owner) {  this.owner = owner;  }  } |

The above code creates tables like this:



Since there is no mappedBy, both sides are responsible for maintaining the relationship. This is not how we model OneToOne relation, so, make sure you add mappedBy to declare the owner of the relationship.

#### Middle table

The third way to represent a OneToOne relationship (using an association table) is quite exotic. We create a middle table to hold the one-to-one relations. This is a little waste of space, but it works.



A customer is linked to a passport through an association table named CustomerPassports; this association table has a foreign key column named passport\_fk pointing to the Passport table (materialized by the inverseJoinColumn, and a foreign key column named customer\_fk pointing to the Customer table materialized by the joinColumns attribute. You must declare the join table name and the join columns explicitly in such a mapping.

### @OneToMany and @ManyToOne

This is the most common relationships.

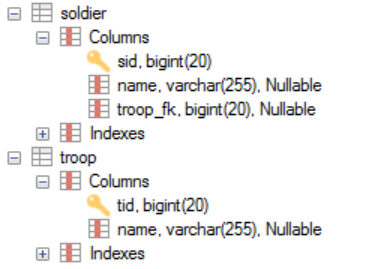
For example, a Troop has many Soldier objects, many Soldier objects belongs to one Troop object.

|  |
| --- |
| @Entity  public class Troop {  private Long tid;  private String name;  private List<Soldier> soldiers;  @Id  @GeneratedValue  public Long getTid() {  return tid;  }  public void setTid(Long tid) {  this.tid = tid;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  @OneToMany(mappedBy = "troop")  public List<Soldier> getSoldiers() {  return soldiers;  }  public void setSoldiers(List<Soldier> soldiers) {  this.soldiers = soldiers;  }  public Troop() {  super();  }  } |

|  |
| --- |
| @Entity  public class Soldier {  private Long sid;  private String name;  private Troop troop;  @Id  @GeneratedValue  public Long getSid() {  return sid;  }  public void setSid(Long sid) {  this.sid = sid;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  @ManyToOne  @JoinColumn(name = "troop\_fk")  public Troop getTroop() {  return troop;  }  public void setTroop(Troop troop) {  this.troop = troop;  }  public Soldier() {  super();  }  } |

Based on mappedBy, Soldier is the owner of the relationship.

Hibernate creates two tables based on the code above:



I am now explaining this annotation: @OneToMany(mappedBy= “troop”) in Troop class. As we mentioned before, mappedBy signals Hibernate that the foreign key for the relationship is on the other side. Since, mappedBy is defined in Troop, the other side is Soldier. You will find a private attribute called troop in Soldier class. Soldier class is responsible for maintaining this relationship.

Note: if you don’t add mappedBy (which is fine), Hibernate will create a third table (Troop\_Soldier) to store this one-to-many relationship. As we all know, this also works, but you don’t have to use a third table to store a one-to-many relationship.

Now, I am explaining this annotation in Soldier class.

@ManyToOne

@JoinColumn(name= “troop\_fk”)

Since Soldier is responsible for maintaining the relationship, its corresponding table has a column called troop\_fk that references the Troop table. If you don’t add this @JoinColumn(name= “troop\_fk”), Hibernate will create a name for this FK automatically (troop\_id).

**The above example is bidirectional one-to-many (which is very common and is enough for working on the project).** If you are interested to know more, keep reading. Now, let’s talk about unidirectional OneToMany and unidirectional many-to-one.

#### Unidirectional OneToMany

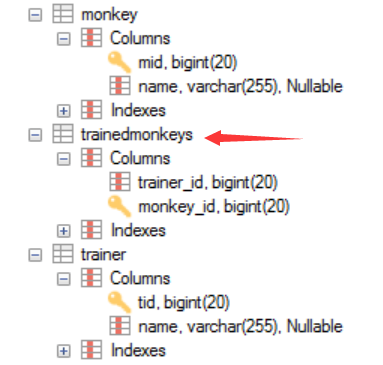
A unidirectional OneToMany with join table is described through an @JoinTable.

For example, one trainer trains many monkeys, so from a Trainer object, we can access the monkeys he/she trains; however, since we define this as unidirectional OneToMany relationship, from a Monkey object, we couldn’t find out who is the trainer. How do we implement this?

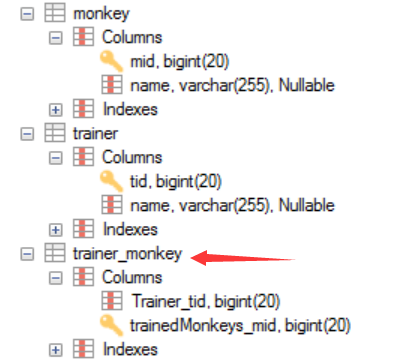
|  |
| --- |
| @Entity  public class Trainer {  private Long tid;  private String name;  private List<Monkey> trainedMonkeys;  @Id  @GeneratedValue  public Long getTid() {  return tid;  }  public void setTid(Long tid) {  this.tid = tid;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  @OneToMany  @JoinTable(  name="TrainedMonkeys",  joinColumns=@JoinColumn(name="trainer\_id"),  inverseJoinColumns=@JoinColumn(name="monkey\_id")  )  public List<Monkey> getTrainedMonkeys() {  return trainedMonkeys;  }  public void setTrainedMonkeys(List<Monkey> trainedMonkeys) {  this.trainedMonkeys = trainedMonkeys;  }  public Trainer() {  super();  }  } |

|  |
| --- |
| @Entity  public class Monkey {  private Long mid;  private String name;  @Id  @GeneratedValue  public Long getMid() {  return mid;  }  public void setMid(Long mid) {  this.mid = mid;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public Monkey() {  super();  }  } |

As you can see, we didn’t add any annotation in the Monkey class. Trainer class describes a unidirectional relationship with Monkey using the join table “TrainedMonkeys” with a foreign keytrainer\_id to Trainer (joinColumns) and a foreign key monkey\_id to Monkey (inversejoinColumns). Here is the table created by Hibernate.



If you take out the @JoinTable statement in Trainer class, just keep @OneToMany, you will get the following tables. It also works, but the middle table’s name is created automatically.

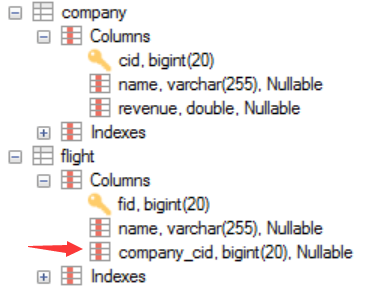


#### Unidirectional ManyToOne

|  |
| --- |
| @Entity  public class Company {  private Long cid;  private String name;  private Double revenue;  @Id  @GeneratedValue  public Long getCid() {  return cid;  }  public void setCid(Long cid) {  this.cid = cid;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public Double getRevenue() {  return revenue;  }  public void setRevenue(Double revenue) {  this.revenue = revenue;  }  public Company() {  super();  }  } |

|  |
| --- |
| @Entity  public class Flight {  private Long fid;  private String name;  private Company company;    @Id  @GeneratedValue  public Long getFid() {  return fid;  }  public void setFid(Long fid) {  this.fid = fid;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  @ManyToOne  public Company getCompany() {  return company;  }  public void setCompany(Company company) {  this.company = company;  }  public Flight() {  super();  }  } |

Hibernate creates the following tables based on the above code:



In this case, there is no need to specify mappedBy.

### @ManyToMany

You can associate entities through a many-to-many relationship using two approaches:

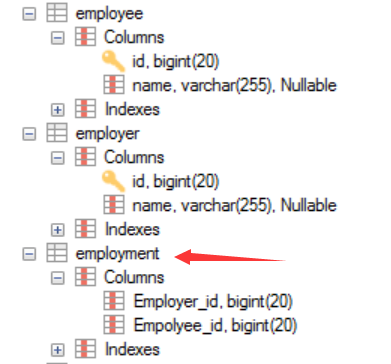
1. use @ManyToMany,
2. use two @OneToMany

The first approach: a ManyToMany association is defined logically using the @ManyToMany annotation. You also have to describe the association table and the join conditions using the @JoinTable annotation. If the association is bidirectional, one side has to be the owner and one side has to be the inverse end (ie. it will be ignored when updating the relationship values in the association table):

|  |
| --- |
| @Entity  public class Employer {  private Long id;  private String name;  private List<Employee> employees;  @Id  @GeneratedValue  public Long getId() {  return id;  }  public void setId(Long id) {  this.id = id;  }  public Employer() {  super();  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  @ManyToMany  @JoinTable(  name="Employment",  joinColumns=@JoinColumn(name="Employer\_id"),  inverseJoinColumns=@JoinColumn(name="Empolyee\_id")  )  public List<Employee> getEmployees() {  return employees;  }  public void setEmployees(List<Employee> employees) {  this.employees = employees;  }  } |

|  |
| --- |
| @Entity  public class Employee {  private Long id;  private String name;  private List<Employer> employers;  @Id  @GeneratedValue  public Long getId() {  return id;  }  public void setId(Long id) {  this.id = id;  }  public Employee() {  super();  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  @ManyToMany(mappedBy=“employees”)  public List<Employer> getEmployers() {  return employers;  }  public void setEmployers(List<Employer> employers) {  this.employers = employers;  }  } |

The above code will generate tables:

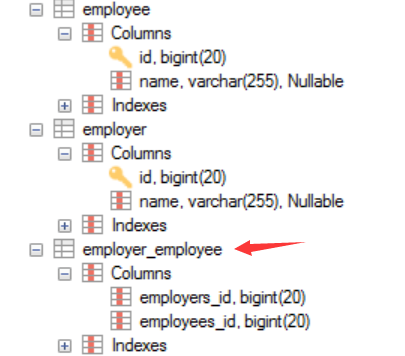


Without describing any physical mapping in a bidirectional ManyToMany, the following rules applied. The table name is the concatenation of the owner table name, \_ and the other side table name. The foreign key name(s) referencing the owner table is the concatenation of the other side property name, **\_**, and the owner primary key column(s). The foreign key name(s) referencing the other side is the concatenation of the owner property name, **\_**, and the other side primary key column(s). These are the same rules used for a unidirectional OneToMany relationship.

|  |
| --- |
| @Entity  public class Employer {  private Long id;  private String name;  private List<Employee> employees;  @Id  @GeneratedValue  public Long getId() {  return id;  }  public void setId(Long id) {  this.id = id;  }  public Employer() {  super();  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  @ManyToMany  public List<Employee> getEmployees() {  return employees;  }  public void setEmployees(List<Employee> employees) {  this.employees = employees;  }  } |

|  |
| --- |
| @Entity  public class Employee {  private Long id;  private String name;  private List<Employer> employers;  @Id  @GeneratedValue  public Long getId() {  return id;  }  public void setId(Long id) {  this.id = id;  }  public Employee() {  super();  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  @ManyToMany(mappedBy=“employees”)  public List<Employer> getEmployers() {  return employers;  }  public void setEmployers(List<Employer> employers) {  this.employers = employers;  }  } |

The above code will generate the following tables:



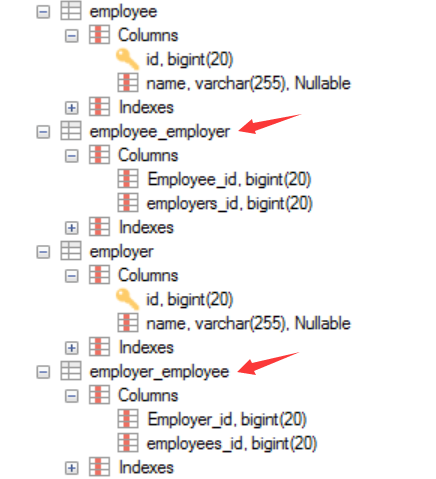
An employer\_employee table is used as the join table. The employers\_id column is a foreign key to the Employer table. The employees\_id column is a foreign key to the Employee table.

Can we remove the mappedBy? See the following.

|  |
| --- |
| @Entity  public class Employer {  private Long id;  private String name;  private List<Employee> employees;  @Id  @GeneratedValue  public Long getId() {  return id;  }  public void setId(Long id) {  this.id = id;  }  public Employer() {  super();  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  @ManyToMany  public List<Employee> getEmployees() {  return employees;  }  public void setEmployees(List<Employee> employees) {  this.employees = employees;  }  } |

|  |
| --- |
| @Entity  public class Employee {  private Long id;  private String name;  private List<Employer> employers;  @Id  @GeneratedValue  public Long getId() {  return id;  }  public void setId(Long id) {  this.id = id;  }  public Employee() {  super();  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  @ManyToMany  public List<Employer> getEmployers() {  return employers;  }  public void setEmployers(List<Employer> employers) {  this.employers = employers;  }  } |

The above code will generate incorrect tables:



**The above ManyToMany is very common and can solve most problems (including the project).**

Second approach to implement many-to-many relationship. Convert a ManyToMany to two OneToMany relationships.

For example, there are three classes: Admin, Role and AdminRole

|  |
| --- |
| @Entity  @Table(name="t\_admin")  public class Admin  {  private int id;  private String name;  private Set<AdminRole> ars;  public Admin()  {  ars = new HashSet<AdminRole>();  }  public void add(AdminRole ar)  {  ars.add(ar);  }  @Id  @GeneratedValue  public int getId()  {  return id;  }  public void setId(int id)  {  this.id = id;  }  public String getName()  {  return name;  }  public void setName(String name)  {  this.name = name;  }  @OneToMany(mappedBy="admin")  public Set<AdminRole> getArs()  {  return ars;  }  public void setArs(Set<AdminRole> ars)  {  this.ars = ars;  }  } |

|  |
| --- |
| @Entity  @Table(name="t\_role")  public class Role  {  private int id;  private String name;  private Set<AdminRole> ars;  public Role()  {  ars = new HashSet<AdminRole>();  }  public void add(AdminRole ar)  {  ars.add(ar);  }  @Id  @GeneratedValue  public int getId()  {  return id;  }  public void setId(int id)  {  this.id = id;  }  public String getName()  {  return name;  }  public void setName(String name)  {  this.name = name;  }  @OneToMany(mappedBy="role")  public Set<AdminRole> getArs()  {  return ars;  }  public void setArs(Set<AdminRole> ars)  {  this.ars = ars;  }  } |

|  |
| --- |
| @Entity  @Table(name="t\_admin\_role")  public class AdminRole  {  private int id;  private String name;  private Admin admin;  private Role role;  @Id  @GeneratedValue  public int getId()  {  return id;  }  public void setId(int id)  {  this.id = id;  }  public String getName()  {  return name;  }  public void setName(String name)  {  this.name = name;  }  @ManyToOne  @JoinColumn(name="aid")  public Admin getAdmin()  {  return admin;  }  public void setAdmin(Admin admin)  {  this.admin = admin;  }  @ManyToOne  @JoinColumn(name="rid")  public Role getRole()  {  return role;  }  public void setRole(Role role)  {  this.role = role;  }  } |

## Mapping Inheritance

Hibernate provides several strategies to handle superclass and subclasses.

1. MappedSuperclass – annotate the parent class
2. Single Table – the entities from different classes with a common ancestor are placed in a single table
3. Joined Table – each class has its table and querying a subclass entity requires joining the tables
4. Table-Per-Class – all the properties of a class, are in its table, so no join is required

Each strategy results in a different database structure.

### MappedSperclass

Person class

|  |
| --- |
| @MappedSuperclass  public class Person {  private Long id;  private String name;  @Id  public Long getId() {  return id;  }  public void setId(Long id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public Person() {  super();  }  } |

Notice that this class no longer has an @Entity annotation, as it won’t be persisted in the database by itself.

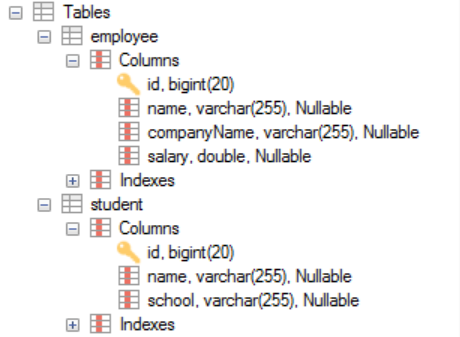
Next, let’s add an Employee subclass:

|  |
| --- |
| @Entity  public class Employee extends Person {  private String companyName;  private Double salary;  public String getCompanyName() {  return companyName;  }  public void setCompanyName(String companyName) {  this.companyName = companyName;  }  public Double getSalary() {  return salary;  }  public void setSalary(Double salary) {  this.salary = salary;  }  public Employee() {  super();  }  } |

Student class

|  |
| --- |
| @Entity  public class Student extends Person {  private String school;  public String getSchool() {  return school;  }  public void setSchool(String school) {  this.school = school;  }  public Student() {  super();  }  } |

Hibernate generates the tables like this:



### Single table

The Single Table strategy creates ONLY ONE table for each class hierarchy. This is also the default strategy chosen by Hibernate if we don’t specify one explicitly.

If you want different strategy, we can define it by adding the @Inheritance annotation to the superclass. I will show you the default one first.

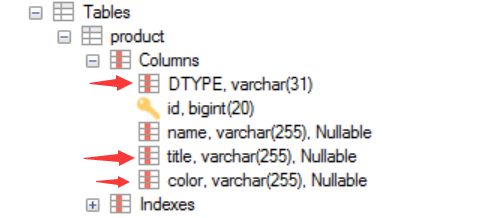
Product class

|  |
| --- |
| @Entity  public class Product {  private Long id;  private String name;  @Id  public Long getId() {  return id;  }  public void setId(Long id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public Product() {  super();  }  } |

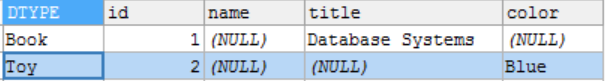
Then I define two subclasses: Book and Toy, they are both kind of Product

|  |
| --- |
| @Entity  public class Book extends Product {  private String title;  public String getTitle() {  return title;  }  public void setTitle(String title) {  this.title = title;  }  public Book() {  super();  }  }  @Entity  public class Toy extends Product{  private String color;  public String getColor() {  return color;  }  public void setColor(String color) {  this.color = color;  }  public Toy() {  super();  }  } |

Hibernate generate the tables:



Since the records for all entities will be in the same table, Hibernate needs a way to differentiate between them. By default, this is done through a discriminator column called DTYPE which has the name of the entity as a value. In this case, if a book instance is inserted, its DTYPE is “Book,” if a toy instance is inserted, its DTYPE is “Toy.”



### Joined table

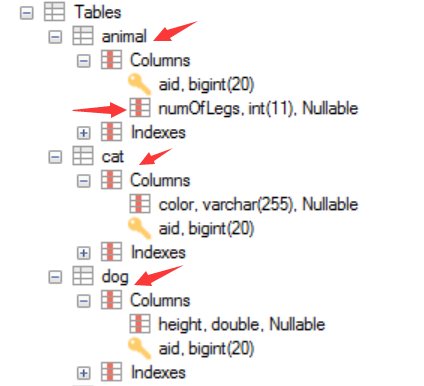
Using this strategy, EACH class in the hierarchy is mapped to its OWN table. The only column which repeatedly appears in all the tables is the identifier, which will be used for joining them when needed. Let’s create a superclass that uses this strategy:

|  |
| --- |
| @Entity  @Inheritance(strategy = InheritanceType.JOINED)  public class Animal {  private Long aid;  private Integer numOfLegs;  @Id  @GeneratedValue  public Long getAid() {  return aid;  }  public void setAid(Long aid) {  this.aid = aid;  }  public Integer getNumOfLegs() {  return numOfLegs;  }  public void setNumOfLegs(Integer numOfLegs) {  this.numOfLegs = numOfLegs;  }  public Animal() {  super();  }  } |

Then, we can simply define two subclasses: Dog and Cat

|  |
| --- |
| @Entity  public class Dog extends Animal {  private Double height;  public Double getHeight() {  return height;  }  public void setHeight(Double height) {  this.height = height;  }  public Dog() {  super();  }  }  @Entity  public class Cat extends Animal {  private String color;  public String getColor() {  return color;  }  public void setColor(String color) {  this.color = color;  }  public Cat() {  super();  }  } |

Hibernate generates the tables:



The primary keys of Dog and Cat also have a foreign key constraint to the primary key of its parent entity, Animal. numOfLegs is only in table animal.

The disadvantage of this inheritance mapping method is that retrieving entities requires joins between tables, which can result in lower performance for large numbers of records.

### Table Per Class

The Table Per Class strategy maps each entity to its table which contains all the properties of the entity, including the ones inherited. The resulting schema is similar to the one using @MappedSuperclass, but unlike it, table per class will indeed define entities for parent classes, allowing associations and polymorphic queries as a result. To use this strategy, we only need to add the @Inheritance annotation to the base class:

|  |
| --- |
| @Entity  @Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)  public class Vehicle {  @Id  private long vehicleId;    private String manufacturer;    // standard constructor, getters, setters  } |

Then, we can create the sub-classes in the standard way.

This is not very different from merely mapping each entity without inheritance. The distinction is apparent when querying the base class, which will return all the sub-class records as well by using a UNION statement in the background. The use of UNION can also lead to inferior performance when choosing this strategy. Another issue is that we can no longer use identity key generation.

# Native Hibernate APIs

A screenshot of a social media post

Description automatically generated

A screenshot of a social media post

Description automatically generated

Remember, Session is NOT thread-safe, so every time, you cannot reuse a session. If you want to use a session, invoke sessionFactory to open a new one for you.

## Session APIs

<https://docs.jboss.org/hibernate/orm/3.5/api/org/hibernate/Session.html>

### save

A screenshot of a cell phone

Description automatically generated

### persist

A screenshot of a social media post

Description automatically generated

### get

A screenshot of a cell phone

Description automatically generated

### createSQLQuery

A screenshot of a cell phone

Description automatically generated

e.g. session.createSQLQuery(“select \* from Employee”);

### update

A screenshot of a cell phone

Description automatically generated

### saveOrUpdate

A screenshot of a social media post

Description automatically generated

### delete

A screenshot of a cell phone

Description automatically generated

# Object States in Hibernate – Transient, Persistent and Detached

Object states in Hibernate plays a vital role in the execution of code in an application. Hibernate has provided three different states for an object of a POJO class. These three states are also called as life cycle states of an object.

## 1. Transient Object State:

An object which is not associated with hibernate session and does not represent a row in the database is considered as transient. It will be garbage collected if no other object refers to it.

An object that is created for the first time using the new() operator is in transient state. When the object is in transient sate then it will not contain any identifier (primary key value). You need to use save, persist or saveOrUpdate methods to persist the transient object.

## 2. Persistent Object State:

An object that is associated with the hibernate session is called as Persistent object. When the object is in persistent state, then it represents one row of the database and consists of an identifier value. You can make a transient instance persistent by associating it with a Session.

## 3. Detached Object State:

Object which is just removed from hibernate session is called as detached object. The state of the detached object is called as detached state.

When the object is in detached sate then it contains identity but you can’t do persistence operation with that identity.

Any changes made to the detached objects are not saved to the database. The detached object can be reattached to the new session and save to the database using update, saveOrUpdate and merge methods.

## Summary of Object States Using a State Transition Diagram

