## Hibernate (Framework)

HB Introduction

HB Architecture

Understanding

First HB application

## Hibernate in IDE

HB in Eclipse

## Hibernate Example

HB with annotation

HB Web application

HB Generator classes

HB Dialects

## Inheritance Mapping

Inheritance Mapping

Table Per Hierarchy

TPH using Annotation

Table Per Concrete

TPC using Annotation

Table Per Subclass

**TPS using Annotation**

Hibernate Tutorial

### Hibernate Framework

Hibernate framework simplifies the development of java application to interact with the database. Hibernate is an open source, lightweight, ORM (Object Relational Mapping) tool.

An ORM tool simplifies the data creation, data manipulation and data access. It is a programming technique that maps the object to the data stored in the database.



The ORM tool internally uses the JDBC API to interact with the database.

### Advantages of Hibernate Framework

There are many advantages of Hibernate Framework. They are as follows:

**1) Opensource and Lightweight:** Hibernate framework is opensource under the LGPL license and lightweight.

**2) Fast performance:** The performance of hibernate framework is fast because cache is internally used in hibernate framework. There are two types of cache in hibernate framework first level cache and second level cache. First level cache is enabled bydefault.

**3) Database Independent query:** HQL (Hibernate Query Language) is the object-oriented version of SQL. It generates the database independent queries. So you don't need to write database specific queries. Before Hibernate, If database is changed for the project, we need to change the SQL query as well that leads to the maintenance problem.

**4) Automatic table creation:** Hibernate framework provides the facility to create the tables of the database automatically. So there is no need to create tables in the database manually.

**5) Simplifies complex join:** To fetch data form multiple tables is easy in hibernate framework.

**6) Provides query statistics and database status:** Hibernate supports Query cache and provide statistics about query and database status.

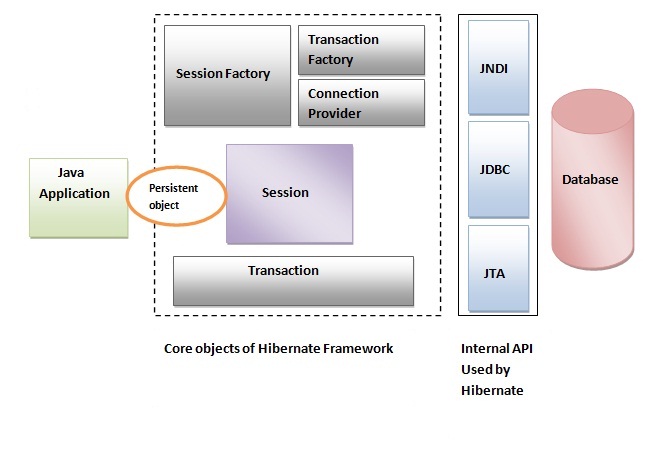
# Hibernate Architecture

The Hibernate architecture includes many objects persistent object, session factory, transaction factory, connection factory, session, transaction etc.

There are 4 layers in hibernate architecture java application layer, hibernate framework layer, backhand api layer and database layer.Let's see the diagram of hibernate architecture:



This is the high level architecture of Hibernate with mapping file and configuration file.



Hibernate framework uses many objects session factory, session, transaction etc. alongwith existing Java API such as JDBC (Java Database Connectivity), JTA (Java Transaction API) and JNDI (Java Naming Directory Interface).

## Elements of Hibernate Architecture

|  |
| --- |
| For creating the first hibernate application, we must know the elements of Hibernate architecture. They are as follows: |

#### SessionFactory

The SessionFactory is a factory of session and client of ConnectionProvider. It holds second level cache (optional) of data. The org.hibernate.SessionFactory interface provides factory method to get the object of Session.

#### Session

The session object provides an interface between the application and data stored in the database. It is a short-lived object and wraps the JDBC connection. It is factory of Transaction, Query and Criteria. It holds a first-level cache (mandatory) of data. The org.hibernate.Session interface provides methods to insert, update and delete the object. It also provides factory methods for Transaction, Query and Criteria.

#### Transaction

The transaction object specifies the atomic unit of work. It is optional. The org.hibernate.Transaction interface provides methods for transaction management.

#### ConnectionProvider

It is a factory of JDBC connections. It abstracts the application from DriverManager or DataSource. It is optional.

#### TransactionFactory

It is a factory of Transaction. It is optional.

# First Hibernate Example without IDE

Here, we are going to create the first hibernate application without IDE. For creating the first hibernate application, we need to follow following steps:

1. Create the Persistent class
2. Create the mapping file for Persistent class
3. Create the Configuration file
4. Create the class that retrieves or stores the persistent object
5. Load the jar file
6. Run the first hibernate application without IDE

# Example to create the Hibernate Application in Eclipse IDE

Here, we are going to create a simple example of hibernate application using eclipse IDE. For creating the first hibernate application in Eclipse IDE, we need to follow following steps:

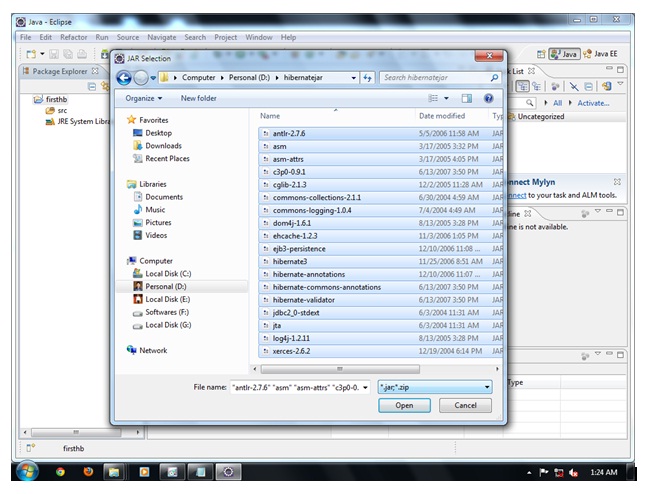
1. Create the java project
2. Add jar files for hibernate
3. Create the Persistent class
4. Create the mapping file for Persistent class
5. Create the Configuration file
6. Create the class that retrieves or stores the persistent object
7. Run the application

### 1) Create the java project

|  |
| --- |
| Create the java project by **File Menu** - **New** - **project** - **java project**. Now specify the project name e.g. firsthb then **next**- **finish**. |

### 2) Add jar files for hibernate

|  |
| --- |
| To add the jar files **Right click on your project** - **Build path** - **Add external archives**. Now select all the jar files as shown in the image given below then click open. |

download the required jar file 

In this example, we are connecting the application with oracle database. So you must add the ojdbc14.jar file.

download the ojdbc14.jar file

### 3) Create the Persistent class

Here, we are creating the same persistent class which we have created in the previous topic. To create the persistent class, Right click on src - New - Class - specify the class with package name (e.g. com.javatpoint.mypackage) - finish.

#### Employee.java

1. **package** com.javatpoint.mypackage;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String firstName,lastName;
7. **public** **int** getId() {
8. **return** id;
9. }
10. **public** **void** setId(**int** id) {
11. **this**.id = id;
12. }
13. **public** String getFirstName() {
14. **return** firstName;
15. }
16. **public** **void** setFirstName(String firstName) {
17. **this**.firstName = firstName;
18. }
19. **public** String getLastName() {
20. **return** lastName;
21. }
22. **public** **void** setLastName(String lastName) {
23. **this**.lastName = lastName;
24. }

27. }

### 4) Create the mapping file for Persistent class

Here, we are creating the same mapping file as created in the previous topic. To create the mapping file, Right click on src - new - file - specify the file name (e.g. employee.hbm.xml) - ok. It must be outside the package.

#### employee.hbm.xml

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3. "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
6. <hibernate-mapping>
7. <**class** name="com.javatpoint.mypackage.Employee" table="emp1000">
8. <id name="id">
9. <generator **class**="assigned"></generator>
10. </id>
12. <property name="firstName"></property>
13. <property name="lastName"></property>
15. </**class**>
17. </hibernate-mapping>

### 5) Create the Configuration file

The configuration file contains all the informations for the database such as connection\_url, driver\_class, username, password etc. The hbm2ddl.auto property is used to create the table in the database automatically. We will have in-depth learning about Dialect class in next topics. To create the configuration file, right click on src - new - file. Now specify the configuration file name e.g. hibernate.cfg.xml.

#### hibernate.cfg.xml

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3. "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4. "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
6. <hibernate-configuration>
8. <session-factory>
9. <property name="hbm2ddl.auto">update</property>
10. <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
11. <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
12. <property name="connection.username">system</property>
13. <property name="connection.password">oracle</property>
14. <property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>
15. <mapping resource="employee.hbm.xml"/>
16. <mapping resource="sal.hbm.xml"/>
17. <mapping resource="product.hbm.xml"/>
18. </session-factory>
20. </hibernate-configuration>

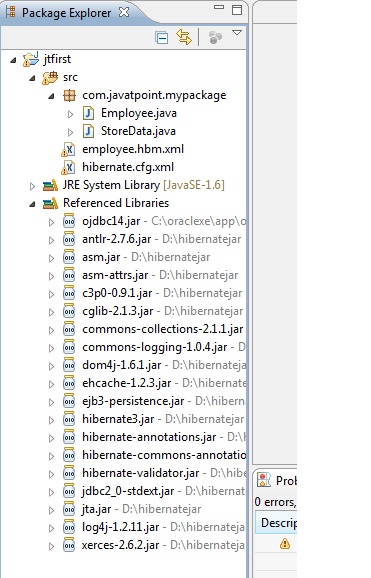
### 6) Create the class that retrieves or stores the persistent object

In this class, we are simply storing the employee object to the database.

1. **package** com.javatpoint.mypackage;
3. **import** org.hibernate.Session;
4. **import** org.hibernate.SessionFactory;
5. **import** org.hibernate.Transaction;
6. **import** org.hibernate.cfg.Configuration;
8. **public** **class** StoreData {
9. **public** **static** **void** main(String[] args) {
11. //creating configuration object
12. Configuration cfg=**new** Configuration();
13. cfg.configure("hibernate.cfg.xml");//populates the data of the configuration file
15. //creating seession factory object
16. SessionFactory factory=cfg.buildSessionFactory();
18. //creating session object
19. Session session=factory.openSession();
21. //creating transaction object
22. Transaction t=session.beginTransaction();
24. Employee e1=**new** Employee();
25. e1.setId(115);
26. e1.setFirstName("sonoo");
27. e1.setLastName("jaiswal");
29. session.persist(e1);//persisting the object
31. t.commit();//transaction is committed
32. session.close();
34. System.out.println("successfully saved");
36. }
37. }

### 7) Run the application

|  |
| --- |
| Before running the application, determine that directory structure is like this. |



|  |
| --- |
| To run the hibernate application, right click on the StoreData class - Run As - Java Application. |

# Hibernate with Annotation

The hibernate application can be created with annotation. There are many annotations that can be used to create hibernate application such as @Entity, @Id, @Table etc.

Hibernate Annotations are based on the JPA 2 specification and supports all the features.

All the JPA annotations are defined in the javax.persistence.\* package. Hibernate **EntityManager** implements the interfaces and life cycle defined by the JPA specification.

The core advantage of using hibernate annotation is that you don't need to create mapping (hbm) file. Here, hibernate annotations are used to provide the meta data.

The hibernate application can be created with annotation. There are many annotations that can be used to create hibernate application such as @Entity, @Id, @Table etc.

Hibernate Annotations are based on the JPA 2 specification and supports all the features.

All the JPA annotations are defined in the javax.persistence.\* package. Hibernate **EntityManager** implements the interfaces and life cycle defined by the JPA specification.

The core advantage of using hibernate annotation is that you don't need to create mapping (hbm) file. Here, hibernate annotations are used to provide the meta data.

### 1) Add the jar file for oracle and annotation

For oracle you need to add **ojdbc14.jar** file. For using annotation, you need to add:

* **hibernate-commons-annotations.jar**
* **ejb3-persistence.jar**
* **hibernate-annotations.jar**

### 2) Create the Persistent class

Here, we are creating the same persistent class which we have created in the previous topic. But here, we are using annotation.

**@Entity** annotation marks this class as an entity.

**@Table** annotation specifies the table name where data of this entity is to be persisted. If you don't use @Table annotation, hibernate will use the class name as the table name bydefault.

**@Id** annotation marks the identifier for this entity.

**@Column** annotation specifies the details of the column for this property or field. If @Column annotation is not specified, property name will be used as the column name bydefault.

#### Employee.java

1. **package** com.javatpoint;
3. **import** javax.persistence.Entity;
4. **import** javax.persistence.Id;
5. **import** javax.persistence.Table;
7. @Entity
8. @Table(name= "emp500")
9. **public** **class** Employee {
10. @Id
11. **private** **int** id;
12. **private** String firstName,lastName;
14. **public** **int** getId() {
15. **return** id;
16. }
17. **public** **void** setId(**int** id) {
18. **this**.id = id;
19. }
20. **public** String getFirstName() {
21. **return** firstName;
22. }
23. **public** **void** setFirstName(String firstName) {
24. **this**.firstName = firstName;
25. }
26. **public** String getLastName() {
27. **return** lastName;
28. }
29. **public** **void** setLastName(String lastName) {
30. **this**.lastName = lastName;
31. }

34. }

### 3) Add mapping of Persistent class in configuration file

open the hibernate.cgf.xml file, and add an entry of mapping resource like this:

1. <mapping **class**="com.javatpoint.Employee"/>

Now the configuration file will look like this:

#### hibernate.cfg.xml

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3. "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4. "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
6. <hibernate-configuration>
8. <session-factory>
9. <property name="hbm2ddl.auto">create</property>
10. <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
11. <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
12. <property name="connection.username">system</property>
13. <property name="connection.password">oracle</property>
14. <property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>
16. <mapping **class**="com.javatpoint.Employee"/>
17. </session-factory>
19. </hibernate-configuration>

### 4) Create the class that retrieves or stores the persistent object

|  |
| --- |
| In this class, we are simply storing the employee object to the database. Here, we are using the **AnnotationConfiguration** class to get the information of mapping from the persistent class. |

1. **package** com.javatpoint.mypackage;
3. **package** com.javatpoint;
5. **import** org.hibernate.\*;
6. **import** org.hibernate.cfg.\*;
8. **public** **class** Test {
9. **public** **static** **void** main(String[] args) {
10. Session session=**new** AnnotationConfiguration()
11. .configure().buildSessionFactory().openSession();
13. Transaction t=session.beginTransaction();
15. Employee e1=**new** Employee();
16. e1.setId(1001);
17. e1.setFirstName("sonoo");
18. e1.setLastName("jaiswal");
20. Employee e2=**new** Employee();
21. e2.setId(1002);
22. e2.setFirstName("vimal");
23. e2.setLastName("jaiswal");
25. session.persist(e1);
26. session.persist(e2);
28. t.commit();
29. session.close();
30. System.out.println("successfully saved");
31. }
32. }

# Web Application with Hibernate

Here, we are going to create a web application with hibernate. For creating the web application, we are using JSP for presentation logic, Bean class for representing data and DAO class for database codes.

As we create the simple application in hibernate, we don't need to perform any extra operations in hibernate for creating web application. In such case, we are getting the value from the user using the JSP file.

### Example to create web application using hibernate

In this example, we are going to insert the record of the user in the database. It is simply a registration form.

#### index.jsp

This page gets input from the user and sends it to the register.jsp file using post method.

1. <form action="register.jsp" method="post">
2. Name:<input type="text" name="name"/><br><br/>
3. Password:<input type="password" name="password"/><br><br/>
4. Email ID:<input type="text" name="email"/><br><br/>
5. <input type="submit" value="register"/>"
7. </form>

#### register.jsp

This file gets all request parameters and stores this information into an object of User class. Further, it calls the register method of UserDao class passing the User class object.

1. <%@page **import**="com.javatpoint.mypack.UserDao"%>
2. <jsp:useBean id="obj" **class**="com.javatpoint.mypack.User">
3. </jsp:useBean>
4. <jsp:setProperty property="\*" name="obj"/>
6. <%
7. **int** i=UserDao.register(obj);
8. **if**(i>0)
9. out.print("You are successfully registered");
11. %>

#### User.java

It is the simple bean class representing the Persistent class in hibernate.

1. **package** com.javatpoint.mypack;
3. **public** **class** User {
4. **private** **int** id;
5. **private** String name,password,email;
7. //getters and setters
9. }

#### user.hbm.xml

It maps the User class with the table of the database.

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3. "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
6. <hibernate-mapping>
7. <**class** name="com.javatpoint.mypack.User" table="u400">
8. <id name="id">
9. <generator **class**="increment"></generator>
10. </id>
11. <property name="name"></property>
12. <property name="password"></property>
13. <property name="email"></property>
14. </**class**>
16. </hibernate-mapping>

#### UserDao.java

A Dao class, containing method to store the instance of User class.

1. **package** com.javatpoint.mypack;

4. **import** org.hibernate.Session;
5. **import** org.hibernate.Transaction;
6. **import** org.hibernate.cfg.Configuration;
8. **public** **class** UserDao {
10. **public** **static** **int** register(User u){
11. **int** i=0;
12. Session session=**new** Configuration().
13. configure().buildSessionFactory().openSession();
15. Transaction t=session.beginTransaction();
16. t.begin();
18. i=(Integer)session.save(u);
20. t.commit();
21. session.close();
23. **return** i;
24. }
25. }

#### hibernate.cfg.xml

It is a configuration file, containing informations about the database and mapping file.

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3. "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4. "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
6. <hibernate-configuration>
8. <session-factory>
9. <property name="hbm2ddl.auto">create</property>
10. <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
11. <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
12. <property name="connection.username">system</property>
13. <property name="connection.password">oracle</property>
14. <property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>
16. <mapping resource="user.hbm.xml"/>
17. </session-factory>
19. </hibernate-configuration>

# Generator classes in Hibernate

The <generator> subelement of id used to generate the unique identifier for the objects of persistent class. There are many generator classes defined in the Hibernate Framework.

All the generator classes implements the **org.hibernate.id.IdentifierGenerator interface**. The application programmer may create one's own generator classes by implementing the IdentifierGenerator interface. Hibernate framework provides many built-in generator classes:

1. assigned
2. increment
3. sequence
4. hilo
5. native
6. identity
7. seqhilo
8. uuid
9. guid
10. select
11. foreign
12. sequence-identity

### 1) assigned

It is the default generator strategy if there is no <generator> element . In this case, application assigns the id. For example:

1. ....
2. <hibernate-mapping>
3. <**class** ...>
4. <id ...>
5. <generator **class**="assigned"></generator>
6. </id>
8. .....
10. </**class**>
11. </hibernate-mapping>

### 2) increment

It generates the unique id only if no other process is inserting data into this table. It generates **short**, **int** or **long** type identifier. The first generated identifier is 1 normally and incremented as 1. Syntax:

1. ....
2. <hibernate-mapping>
3. <**class** ...>
4. <id ...>
5. <generator **class**="increment"></generator>
6. </id>
8. .....
10. </**class**>
11. </hibernate-mapping>

### 3) sequence

It uses the sequence of the database. if there is no sequence defined, it creates a sequence automatically e.g. in case of Oracle database, it creates a sequence named HIBERNATE\_SEQUENCE. In case of Oracle, DB2, SAP DB, Postgre SQL or McKoi, it uses sequence but it uses generator in interbase. Syntax:

1. .....
2. <id ...>
3. <generator **class**="sequence"></generator>
4. </id>
5. .....

For defining your own sequence, use the param subelement of generator.

1. .....
2. <id ...>
3. <generator **class**="sequence">
4. <param name="sequence">your\_sequence\_name</param>
5. </generator>
6. </id>
7. .....

### 4) hilo

It uses high and low algorithm to generate the id of type short, int and long. Syntax:

1. .....
2. <id ...>
3. <generator **class**="hilo"></generator>
4. </id>
5. .....

### 5) native

It uses identity, sequence or hilo depending on the database vendor. Syntax:

1. .....
2. <id ...>
3. <generator **class**="native"></generator>
4. </id>
5. .....

### 6) identity

It is used in Sybase, My SQL, MS SQL Server, DB2 and HypersonicSQL to support the id column. The returned id is of type short, int or long.

### 7) seqhilo

It uses high and low algorithm on the specified sequence name. The returned id is of type short, int or long.

### 8) uuid

It uses 128-bit UUID algorithm to generate the id. The returned id is of type String, unique within a network (because IP is used). The UUID is represented in hexadecimal digits, 32 in length.

### 9) guid

|  |
| --- |
| It uses GUID generated by database of type string. It works on MS SQL Server and MySQL. |

### 10) select

|  |
| --- |
| It uses the primary key returned by the database trigger. |

### 11) foreign

|  |
| --- |
| It uses the id of another associated object, mostly used with <one-to-one> association. |

### 12) sequence-identity

|  |
| --- |
| It uses a special sequence generation strategy. It is supported in Oracle 10g drivers only. |

# SQL Dialects in Hibernate

For connecting any hibernate application with the database, you must specify the SQL dialects. There are many Dialects classes defined for RDBMS in the org.hibernate.dialect package. They are as follows:

|  |  |
| --- | --- |
| **RDBMS** | **Dialect** |
| Oracle (any version) | org.hibernate.dialect.OracleDialect |
| Oracle9i | org.hibernate.dialect.Oracle9iDialect |
| Oracle10g | org.hibernate.dialect.Oracle10gDialect |
| MySQL | org.hibernate.dialect.MySQLDialect |
| MySQL with InnoDB | org.hibernate.dialect.MySQLInnoDBDialect |
| MySQL with MyISAM | org.hibernate.dialect.MySQLMyISAMDialect |
| DB2 | org.hibernate.dialect.DB2Dialect |
| DB2 AS/400 | org.hibernate.dialect.DB2400Dialect |
| DB2 OS390 | org.hibernate.dialect.DB2390Dialect |
| Microsoft SQL Server | org.hibernate.dialect.SQLServerDialect |
| Sybase | org.hibernate.dialect.SybaseDialect |
| Sybase Anywhere | org.hibernate.dialect.SybaseAnywhereDialect |
| PostgreSQL | org.hibernate.dialect.PostgreSQLDialect |
| SAP DB | org.hibernate.dialect.SAPDBDialect |
| Informix | org.hibernate.dialect.InformixDialect |
| HypersonicSQL | org.hibernate.dialect.HSQLDialect |
| Ingres | org.hibernate.dialect.IngresDialect |
| Progress | org.hibernate.dialect.ProgressDialect |
| Mckoi SQL | org.hibernate.dialect.MckoiDialect |
| Interbase | org.hibernate.dialect.InterbaseDialect |
| Pointbase | org.hibernate.dialect.PointbaseDialect |
| FrontBase | org.hibernate.dialect.FrontbaseDialect |
| Firebird | org.hibernate.dialect.FirebirdDialect |

# Hibernate Inheritance Mapping Tutorial

We can map the inheritance hierarchy classes with the table of the database. There are three inheritance mapping strategies defined in the hibernate:

1. Table Per Hierarchy
2. Table Per Concrete class
3. Table Per Subclass

#### Table Per Hierarchy

In table per hierarchy mapping, single table is required to map the whole hierarchy, an extra column (known as discriminator column) is added to identify the class. But nullable values are stored in the table .

Table Per Hierarchy using xml file

Table Per Hierarchy using Annotation

#### Table Per Concrete class

In case of table per concrete class, tables are created as per class. But duplicate column is added in subclass tables.

Table Per Concrete class using xml file  
Table Per Concrete class using Annotation

#### Table Per Subclass

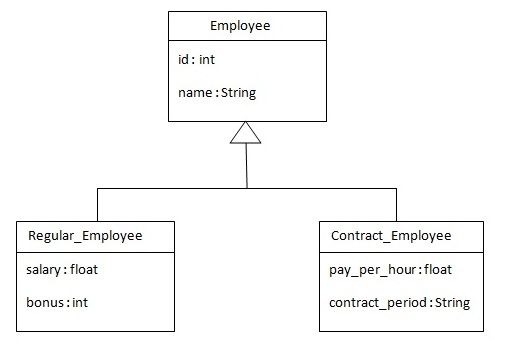
In this strategy, tables are created as per class but related by foreign key. So there are no duplicate columns.

Table Per Subclass using xml file  
Table Per Subclass using Annotation

# Hibernate Table Per Hierarchy using xml file

By this inheritance strategy, we can map the whole hierarchy by single table only. Here, an extra column (also known as **discriminator column**) is created in the table to identify the class.

Let's understand the problem first. I want to map the whole hierarchy given below into one table of the database.

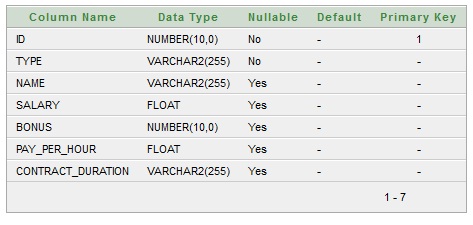


There are three classes in this hierarchy. Employee is the super class for Regular\_Employee and Contract\_Employee classes. Let's see the mapping file for this hierarchy.

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3. "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
6. <hibernate-mapping>
7. <**class** name="com.javatpoint.mypackage.Employee" table="emp121" discriminator-value="emp">
8. <id name="id">
9. <generator **class**="increment"></generator>
10. </id>
12. <discriminator column="type" type="string"></discriminator>
13. <property name="name"></property>
15. <subclass name="com.javatpoint.mypackage.Regular\_Employee" discriminator-value="reg\_emp">
16. <property name="salary"></property>
17. <property name="bonus"></property>
18. </subclass>
20. <subclass name="com.javatpoint.mypackage.Contract\_Employee" discriminator-value="con\_emp">
21. <property name="pay\_per\_hour"></property>
22. <property name="contract\_duration"></property>
23. </subclass>
25. </**class**>
27. </hibernate-mapping>

|  |
| --- |
| In case of table per class hierarchy an discriminator column is added by the hibernate framework that specifies the type of the record. It is mainly used to distinguish the record. To specify this, **discriminator** subelement of class must be specified. |
| The **subclass** subelement of class, specifies the subclass. In this case, Regular\_Employee and Contract\_Employee are the subclasses of Employee class. |

The table structure for this hierarchy is as shown below:



### Example of Table per class hierarchy

In this example we are creating the three classes and provide mapping of these classes in the employee.hbm.xml file.

### 1) Create the Persistent classes

You need to create the persistent classes representing the inheritance. Let's create the three classes for the above hierarchy:

*File: Employee.java*

1. **package** com.javatpoint.mypackage;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
7. //getters and setters
8. }

*File: Regular\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **public** **class** Regular\_Employee **extends** Employee{
4. **private** **float** salary;
5. **private** **int** bonus;
7. //getters and setters
8. }

*File: Contract\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **public** **class** Contract\_Employee **extends** Employee{
4. **private** **float** pay\_per\_hour;
5. **private** String contract\_duration;
7. //getters and setters
8. }

### 2) Create the mapping file for Persistent class

The mapping has been discussed above for the hierarchy.

*File: employee.hbm.xml*

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3. "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
6. <hibernate-mapping>
7. <**class** name="com.javatpoint.mypackage.Employee" table="emp121" discriminator-value="emp">
8. <id name="id">
9. <generator **class**="increment"></generator>
10. </id>
12. <discriminator column="type" type="string"></discriminator>
13. <property name="name"></property>
15. <subclass name="com.javatpoint.mypackage.Regular\_Employee" discriminator-value="reg\_emp">
16. <property name="salary"></property>
17. <property name="bonus"></property>
18. </subclass>
20. <subclass name="com.javatpoint.mypackage.Contract\_Employee" discriminator-value="con\_emp">
21. <property name="pay\_per\_hour"></property>
22. <property name="contract\_duration"></property>
23. </subclass>
25. </**class**>
27. </hibernate-mapping>

### 3) Add mapping of hbm file in configuration file

Open the hibernate.cgf.xml file, and add an entry of mapping resource like this:

1. <mapping resource="employee.hbm.xml"/>

Now the configuration file will look like this:

*File: hibernate.cfg.xml*

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3. "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4. "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
6. <hibernate-configuration>
8. <session-factory>
9. <property name="hbm2ddl.auto">update</property>
10. <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
11. <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
12. <property name="connection.username">system</property>
13. <property name="connection.password">oracle</property>
14. <property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>
15. <mapping resource="employee.hbm.xml"/>
16. </session-factory>
18. </hibernate-configuration>

The hbm2ddl.auto property is defined for creating automatic table in the database.

### 4) Create the class that stores the persistent object

In this class, we are simply storing the employee objects in the database.

*File: StoreData.java*

1. **package** com.javatpoint.mypackage;
2. **import** org.hibernate.\*;
3. **import** org.hibernate.cfg.\*;
5. **public** **class** StoreData {
6. **public** **static** **void** main(String[] args) {
7. Session session=**new** Configuration().configure("hibernate.cfg.xml")
8. .buildSessionFactory().openSession();
10. Transaction t=session.beginTransaction();
12. Employee e1=**new** Employee();
13. e1.setName("sonoo");
15. Regular\_Employee e2=**new** Regular\_Employee();
16. e2.setName("Vivek Kumar");
17. e2.setSalary(50000);
18. e2.setBonus(5);
20. Contract\_Employee e3=**new** Contract\_Employee();
21. e3.setName("Arjun Kumar");
22. e3.setPay\_per\_hour(1000);
23. e3.setContract\_duration("15 hours");
25. session.persist(e1);
26. session.persist(e2);
27. session.persist(e3);
29. t.commit();
30. session.close();
31. System.out.println("success");
32. }
33. }

### Output:

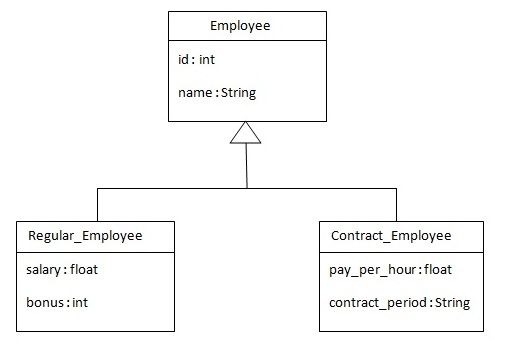


# Hibernate Table Per Hierarchy using Annotation

In the previous page, we have mapped the inheritance hierarchy with one table only using xml file. Here, we are going to perform this task using annotation. You need to use @Inheritance(strategy=InheritanceType.SINGLE\_TABLE), @DiscriminatorColumn and @DiscriminatorValue annotations for mapping table per hierarchy strategy.

In case of table per hierarchy, only one table is required to map the inheritance hierarchy. Here, an extra column (also known as **discriminator column**) is created in the table to identify the class.

Let's see the inheritance hierarchy:



There are three classes in this hierarchy. Employee is the super class for Regular\_Employee and Contract\_Employee classes.

|  |
| --- |
| The table structure for this hierarchy is as shown below: |

### Example of Hibernate Table Per Hierarchy using Annotation

You need to follow following steps to create simple example:

* Create the persistent classes
* Create the configuration file
* Create the class to store the fetch the data

### 1) Create the Persistent classes

You need to create the persistent classes representing the inheritance. Let's create the three classes for the above hierarchy:

*File: Employee.java*

1. **package** com.javatpoint.mypackage;
2. **import** javax.persistence.\*;
4. @Entity
5. @Table(name = "employee101")
6. @Inheritance(strategy=InheritanceType.SINGLE\_TABLE)
7. @DiscriminatorColumn(name="type",discriminatorType=DiscriminatorType.STRING)
8. @DiscriminatorValue(value="employee")
10. **public** **class** Employee {
11. @Id
12. @GeneratedValue(strategy=GenerationType.AUTO)
14. @Column(name = "id")
15. **private** **int** id;
17. @Column(name = "name")
18. **private** String name;
20. //setters and getters
21. }

*File: Regular\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **import** javax.persistence.\*;
5. @Entity
6. @DiscriminatorValue("regularemployee")
7. **public** **class** Regular\_Employee **extends** Employee{
9. @Column(name="salary")
10. **private** **float** salary;
12. @Column(name="bonus")
13. **private** **int** bonus;
15. //setters and getters
16. }

*File: Contract\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **import** javax.persistence.Column;
4. **import** javax.persistence.DiscriminatorValue;
5. **import** javax.persistence.Entity;
7. @Entity
8. @DiscriminatorValue("contractemployee")
9. **public** **class** Contract\_Employee **extends** Employee{
11. @Column(name="pay\_per\_hour")
12. **private** **float** pay\_per\_hour;
14. @Column(name="contract\_duration")
15. **private** String contract\_duration;
17. //setters and getters
18. }

### 2) Add the persistent classes in configuration file

|  |
| --- |
| Open the hibernate.cgf.xml file, and add entries of entity classes like this: |

1. <mapping **class**="com.javatpoint.mypackage.Employee"/>
2. <mapping **class**="com.javatpoint.mypackage.Contract\_Employee"/>
3. <mapping **class**="com.javatpoint.mypackage.Regular\_Employee"/>
4. </pre></div>
5. <table >
6. <tr><td>Now the configuration file will look like **this**:
7. </td></tr>
8. </table>
9. <span id="filename">File: hibernate.cfg.xml</span>
10. <div **class**="codeblock"><pre name="code" **class**="java" >
11. <?xml version='1.0' encoding='UTF-8'?>
12. <!DOCTYPE hibernate-configuration PUBLIC
13. "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
14. "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
16. <!-- Generated by MyEclipse Hibernate Tools.                   -->
17. <hibernate-configuration>
19. <session-factory>
20. <property name="hbm2ddl.auto">update</property>
21. <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
22. <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
23. <property name="connection.username">system</property>
24. <property name="connection.password">oracle</property>
25. <property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>
27. <mapping **class**="com.javatpoint.mypackage.Employee"/>
28. <mapping **class**="com.javatpoint.mypackage.Contract\_Employee"/>
29. <mapping **class**="com.javatpoint.mypackage.Regular\_Employee"/>
30. </session-factory>
32. </hibernate-configuration>

The hbm2ddl.auto property is defined for creating automatic table in the database.

### 3) Create the class that stores the persistent object

In this class, we are simply storing the employee objects in the database.

*File: StoreTest.java*

1. **package** com.javatpoint.mypackage;
3. **import** org.hibernate.\*;
4. **import** org.hibernate.cfg.\*;
6. **public** **class** StoreData {
7. **public** **static** **void** main(String[] args) {
8. AnnotationConfiguration cfg=**new** AnnotationConfiguration();
9. Session session=cfg.configure("hibernate.cfg.xml").buildSessionFactory().openSession();
11. Transaction t=session.beginTransaction();
13. Employee e1=**new** Employee();
14. e1.setName("sonoo");
16. Regular\_Employee e2=**new** Regular\_Employee();
17. e2.setName("Vivek Kumar");
18. e2.setSalary(50000);
19. e2.setBonus(5);
21. Contract\_Employee e3=**new** Contract\_Employee();
22. e3.setName("Arjun Kumar");
23. e3.setPay\_per\_hour(1000);
24. e3.setContract\_duration("15 hours");
26. session.persist(e1);
27. session.persist(e2);
28. session.persist(e3);
30. t.commit();
31. session.close();
32. System.out.println("success");
33. }
34. }

#### Output:

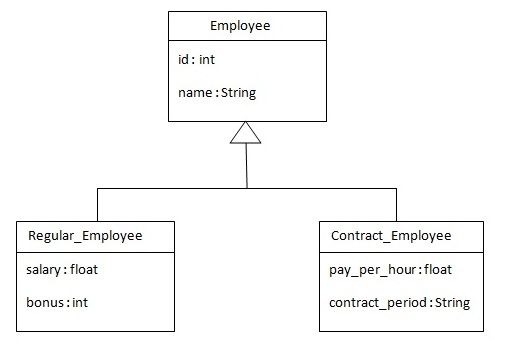


# Table Per Concrete class using xml file

In case of Table Per Concrete class, there will be three tables in the database having no relations to each other. There are two ways to map the table with table per concrete class strategy.

* By union-subclass element
* By Self creating the table for each class

Let's understand what hierarchy we are going to map.



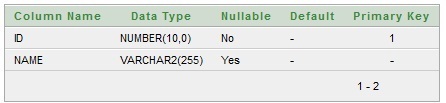
Let's see how can we map this hierarchy by union-subclass element:

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3. "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
6. <hibernate-mapping>
7. <**class** name="com.javatpoint.mypackage.Employee" table="emp122">
8. <id name="id">
9. <generator **class**="increment"></generator>
10. </id>
12. <property name="name"></property>
14. <union-subclass name="com.javatpoint.mypackage.Regular\_Employee" table="regemp122">
15. <property name="salary"></property>
16. <property name="bonus"></property>
17. </union-subclass>
19. <union-subclass name="com.javatpoint.mypackage.Contract\_Employee" table="contemp122">
20. <property name="pay\_per\_hour"></property>
21. <property name="contract\_duration"></property>
22. </union-subclass>
24. </**class**>
26. </hibernate-mapping>

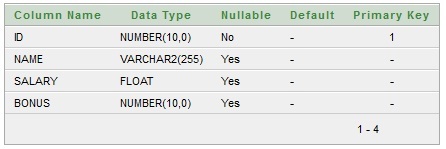
|  |
| --- |
| In case of table per concrete class, there will be three tables in the database, each representing a particular class. |
| The **union-subclass** subelement of class, specifies the subclass. It adds the columns of parent table into this table. In other words, it is working as a union. |

|  |
| --- |
| The table structure for each table will be as follows: |

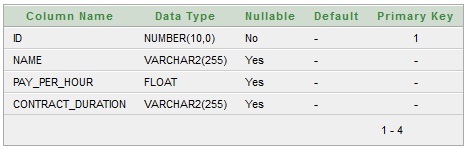
#### Table structure for Employee class

#### Table structure for Regular\_Employee class

#### Table structure for Contract\_Employee class



### Example of Table per concrete class

In this example we are creating the three classes and provide mapping of these classes in the employee.hbm.xml file.

### 1) Create the Persistent classes

You need to create the persistent classes representing the inheritance. Let's create the three classes for the above hierarchy:

*File: Employee.java*

1. **package** com.javatpoint.mypackage;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
7. //getters and setters
8. }

*File: Regular\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **public** **class** Regular\_Employee **extends** Employee{
4. **private** **float** salary;
5. **private** **int** bonus;
7. //getters and setters
8. }

*File: Contract\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **public** **class** Contract\_Employee **extends** Employee{
4. **private** **float** pay\_per\_hour;
5. **private** String contract\_duration;
7. //getters and setters
8. }

### 2) Create the mapping file for Persistent class

The mapping has been discussed above for the hierarchy.

*File: employee.hbm.xml*

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
3. "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
4. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

7. <hibernate-mapping>
8. <**class** name="com.javatpoint.mypackage.Employee" table="emp122">
9. <id name="id">
10. <generator **class**="increment"></generator>
11. </id>
13. <property name="name"></property>
15. <union-subclass name="com.javatpoint.mypackage.Regular\_Employee" table="regemp122">
16. <property name="salary"></property>
17. <property name="bonus"></property>
18. </union-subclass>
20. <union-subclass name="com.javatpoint.mypackage.Contract\_Employee" table="contemp122">
21. <property name="pay\_per\_hour"></property>
22. <property name="contract\_duration"></property>
23. </union-subclass>
25. </**class**>
27. </hibernate-mapping>

### 3) Add mapping of hbm file in configuration file

Open the hibernate.cgf.xml file, and add an entry of mapping resource like this:

1. <mapping resource="employee.hbm.xml"/>

Now the configuration file will look like this:

*File: hibernate.cfg.xml*

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3. "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4. "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
6. <hibernate-configuration>
8. <session-factory>
9. <property name="hbm2ddl.auto">update</property>
10. <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
11. <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
12. <property name="connection.username">system</property>
13. <property name="connection.password">oracle</property>
14. <property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>
15. <mapping resource="employee.hbm.xml"/>
16. </session-factory>
18. </hibernate-configuration>

The hbm2ddl.auto property is defined for creating automatic table in the database.

### 4) Create the class that stores the persistent object

In this class, we are simply storing the employee objects in the database.

*File: StoreData.java*

1. **package** com.javatpoint.mypackage;
3. **import** org.hibernate.\*;
4. **import** org.hibernate.cfg.\*;
6. **public** **class** StoreData {
7. **public** **static** **void** main(String[] args) {
8. Session session=**new** Configuration().configure("hibernate.cfg.xml")
9. .buildSessionFactory().openSession();
11. Transaction t=session.beginTransaction();
13. Employee e1=**new** Employee();
14. e1.setName("sonoo");
16. Regular\_Employee e2=**new** Regular\_Employee();
17. e2.setName("Vivek Kumar");
18. e2.setSalary(50000);
19. e2.setBonus(5);
21. Contract\_Employee e3=**new** Contract\_Employee();
22. e3.setName("Arjun Kumar");
23. e3.setPay\_per\_hour(1000);
24. e3.setContract\_duration("15 hours");
26. session.persist(e1);
27. session.persist(e2);
28. session.persist(e3);
30. t.commit();
31. session.close();
32. System.out.println("success");
33. }
34. }

# Table Per Concrete class using Annotation

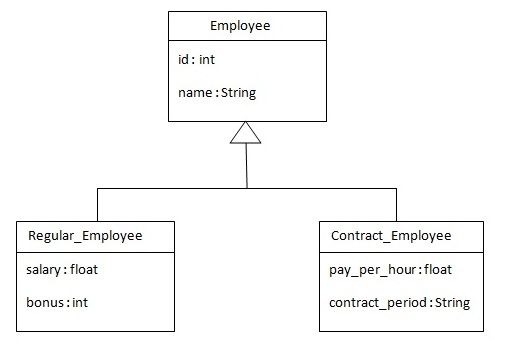
In case of Table Per Concrete class, tables are created per class. So there are no nullable values in the table. Disadvantage of this approach is that duplicate columns are created in the subclass tables.

Here, we need to use @Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS) annotation in the parent class and @AttributeOverrides annotation in the subclasses.

**@Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)** specifies that we are using table per concrete class strategy. It should be specified in the parent class only.

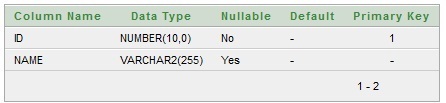
**@AttributeOverrides** defines that parent class attributes will be overriden in this class. In table structure, parent class table columns will be added in the subclass table.

The class hierarchy is given below:

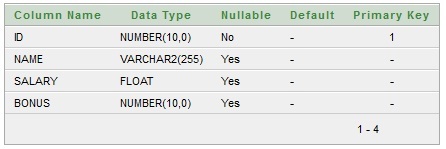


|  |
| --- |
| The table structure for each table will be as follows: |

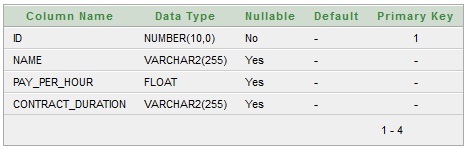
#### Table structure for Employee class

#### Table structure for Regular\_Employee class

#### Table structure for Contract\_Employee class



### Example of Table per concrete class

In this example we are creating the three classes and provide mapping of these classes in the employee.hbm.xml file.

### 1) Create the Persistent classes

You need to create the persistent classes representing the inheritance. Let's create the three classes for the above hierarchy:

*File: Employee.java*

1. **package** com.javatpoint.mypackage;
2. **import** javax.persistence.\*;
4. @Entity
5. @Table(name = "employee102")
6. @Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)
8. **public** **class** Employee {
9. @Id
10. @GeneratedValue(strategy=GenerationType.AUTO)
12. @Column(name = "id")
13. **private** **int** id;
15. @Column(name = "name")
16. **private** String name;
18. //setters and getters
19. }

*File: Regular\_Employee.java*

1. **package** com.javatpoint.mypackage;
2. **import** javax.persistence.\*;
4. @Entity
5. @Table(name="regularemployee102")
6. @AttributeOverrides({
7. @AttributeOverride(name="id", column=@Column(name="id")),
8. @AttributeOverride(name="name", column=@Column(name="name"))
9. })
10. **public** **class** Regular\_Employee **extends** Employee{
12. @Column(name="salary")
13. **private** **float** salary;
15. @Column(name="bonus")
16. **private** **int** bonus;
18. //setters and getters
19. }

*File: Contract\_Employee.java*

1. **package** com.javatpoint.mypackage;
2. **import** javax.persistence.\*;
3. @Entity
4. @Table(name="contractemployee102")
5. @AttributeOverrides({
6. @AttributeOverride(name="id", column=@Column(name="id")),
7. @AttributeOverride(name="name", column=@Column(name="name"))
8. })
9. **public** **class** Contract\_Employee **extends** Employee{
11. @Column(name="pay\_per\_hour")
12. **private** **float** pay\_per\_hour;
14. @Column(name="contract\_duration")
15. **private** String contract\_duration;
17. **public** **float** getPay\_per\_hour() {
18. **return** pay\_per\_hour;
19. }
20. **public** **void** setPay\_per\_hour(**float** payPerHour) {
21. pay\_per\_hour = payPerHour;
22. }
23. **public** String getContract\_duration() {
24. **return** contract\_duration;
25. }
26. **public** **void** setContract\_duration(String contractDuration) {
27. contract\_duration = contractDuration;
28. }
29. }

### 2) Add mapping of hbm file in configuration file

*File: hibernate.cfg.xml*

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3. "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4. "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
6. <!-- Generated by MyEclipse Hibernate Tools.                   -->
7. <hibernate-configuration>
8. <session-factory>
9. <property name="hbm2ddl.auto">update</property>
10. <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
11. <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
12. <property name="connection.username">system</property>
13. <property name="connection.password">oracle</property>
14. <property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>
16. <mapping **class**="com.javatpoint.mypackage.Employee"/>
17. <mapping **class**="com.javatpoint.mypackage.Contract\_Employee"/>
18. <mapping **class**="com.javatpoint.mypackage.Regular\_Employee"/>
19. </session-factory>
20. </hibernate-configuration>

The hbm2ddl.auto property is defined for creating automatic table in the database.

### 3) Create the class that stores the persistent object

In this class, we are simply storing the employee objects in the database.

*File: StoreData.java*

1. **package** com.javatpoint.mypackage;
3. **import** org.hibernate.\*;
4. **import** org.hibernate.cfg.\*;
6. **public** **class** StoreData {
7. **public** **static** **void** main(String[] args) {
8. AnnotationConfiguration cfg=**new** AnnotationConfiguration();
9. Session session=cfg.configure("hibernate.cfg.xml").buildSessionFactory().openSession();
11. Transaction t=session.beginTransaction();
13. Employee e1=**new** Employee();
14. e1.setName("sonoo");
16. Regular\_Employee e2=**new** Regular\_Employee();
17. e2.setName("Vivek Kumar");
18. e2.setSalary(50000);
19. e2.setBonus(5);
21. Contract\_Employee e3=**new** Contract\_Employee();
22. e3.setName("Arjun Kumar");
23. e3.setPay\_per\_hour(1000);
24. e3.setContract\_duration("15 hours");
26. session.persist(e1);
27. session.persist(e2);
28. session.persist(e3);
30. t.commit();
31. session.close();
32. System.out.println("success");
33. }
34. }

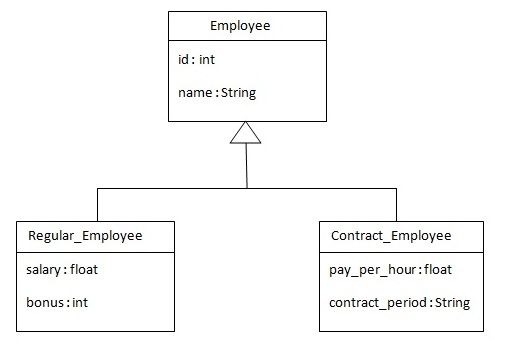
# Table Per Subclass Example using xml file

In case of Table Per Subclass, subclass mapped tables are related to parent class mapped table by primary key and foreign key relationship.

The **<joined-subclass>** element of class is used to map the child class with parent using the primary key and foreign key relation.

In this example, we are going to use hb2ddl.auto property to generate the table automatically. So we don't need to be worried about creating tables in the database.

Let's see the hierarchy of classes that we are going to map.



Let's see how can we map this hierarchy by joined-subclass element:

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
4. "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
5. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

8. <hibernate-mapping>
9. <**class** name="com.javatpoint.mypackage.Employee" table="emp123">
10. <id name="id">
11. <generator **class**="increment"></generator>
12. </id>
14. <property name="name"></property>
16. <joined-subclass name="com.javatpoint.mypackage.Regular\_Employee" table="regemp123">
17. <key column="eid"></key>
18. <property name="salary"></property>
19. <property name="bonus"></property>
20. </joined-subclass>
22. <joined-subclass name="com.javatpoint.mypackage.Contract\_Employee" table="contemp123">
23. <key column="eid"></key>
24. <property name="pay\_per\_hour"></property>
25. <property name="contract\_duration"></property>
26. </joined-subclass>
28. </**class**>
29. </hibernate-mapping>

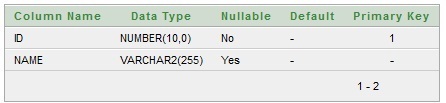
In case of table per subclass class, there will be three tables in the database, each representing a particular class.

The **joined-subclass** subelement of class, specifies the subclass.

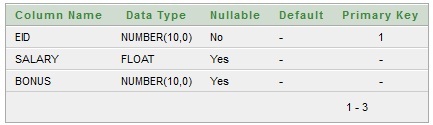
The **key** sub-element of joined-subclass is used to generate the foreign key in the subclass mapped table. This foreign key will be associated with the primary key of parent class mapped table.

The table structure for each table will be as follows:

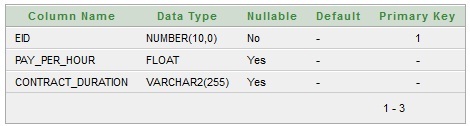
#### Table structure for Employee class

#### Table structure for Regular\_Employee class

#### Table structure for Contract\_Employee class



### Example of Table per subclass class

In this example we are creating the three classes and provide mapping of these classes in the employee.hbm.xml file.

### 1) Create the Persistent classes

You need to create the persistent classes representing the inheritance. Let's create the three classes for the above hierarchy:

*File: Employee.java*

1. **package** com.javatpoint.mypackage;
3. **public** **class** Employee {
4. **private** **int** id;
5. **private** String name;
7. //getters and setters
8. }

*File: Regular\_Employee.java*

1. **package** com.javatpoint.mypackage;
2. **public** **class** Regular\_Employee **extends** Employee{
3. **private** **float** salary;
4. **private** **int** bonus;
6. //getters and setters
7. }

*File: Contract\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **public** **class** Contract\_Employee **extends** Employee{
4. **private** **float** pay\_per\_hour;
5. **private** String contract\_duration;
7. //getters and setters
8. }

### 2) Create the mapping file for Persistent class

The mapping has been discussed above for the hierarchy.

*File: employee.hbm.xml*

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-mapping PUBLIC
4. "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
5. "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

8. <hibernate-mapping>
9. <**class** name="com.javatpoint.mypackage.Employee" table="emp123">
10. <id name="id">
11. <generator **class**="increment"></generator>
12. </id>
14. <property name="name"></property>
16. <joined-subclass name="com.javatpoint.mypackage.Regular\_Employee" table="regemp123">
17. <key column="eid"></key>
18. <property name="salary"></property>
19. <property name="bonus"></property>
20. </joined-subclass>
22. <joined-subclass name="com.javatpoint.mypackage.Contract\_Employee" table="contemp123">
23. <key column="eid"></key>
24. <property name="pay\_per\_hour"></property>
25. <property name="contract\_duration"></property>
26. </joined-subclass>
28. </**class**>
29. </hibernate-mapping>

### 3) create configuration file

Open the hibernate.cgf.xml file, and add an entry of mapping resource like this:

1. <mapping resource="employee.hbm.xml"/>

Now the configuration file will look like this:

*File: hibernate.cfg.xml*

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3. "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4. "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
6. <hibernate-configuration>
8. <session-factory>
9. <property name="hbm2ddl.auto">update</property>
10. <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
11. <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
12. <property name="connection.username">system</property>
13. <property name="connection.password">oracle</property>
14. <property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>
15. <mapping resource="employee.hbm.xml"/>
16. </session-factory>
18. </hibernate-configuration>

The hbm2ddl.auto property is defined for creating automatic table in the database.

### 4) Create the class that stores the persistent object

In this class, we are simply storing the employee objects in the database.

*File: StoreData.java*

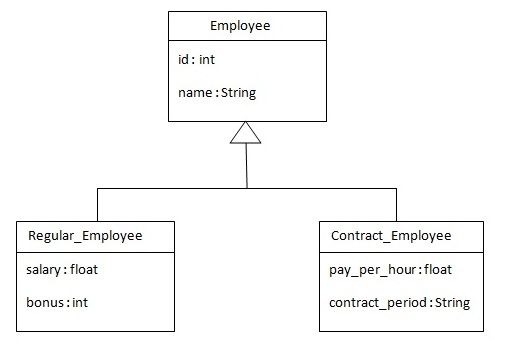
1. **package** com.javatpoint.mypackage;
3. **import** org.hibernate.\*;
4. **import** org.hibernate.cfg.\*;
6. **public** **class** StoreData {
7. **public** **static** **void** main(String[] args) {
8. Session session=**new** Configuration().configure("hibernate.cfg.xml")
9. .buildSessionFactory().openSession();
11. Transaction t=session.beginTransaction();
13. Employee e1=**new** Employee();
14. e1.setName("sonoo");
16. Regular\_Employee e2=**new** Regular\_Employee();
17. e2.setName("Vivek Kumar");
18. e2.setSalary(50000);
19. e2.setBonus(5);
21. Contract\_Employee e3=**new** Contract\_Employee();
22. e3.setName("Arjun Kumar");
23. e3.setPay\_per\_hour(1000);
24. e3.setContract\_duration("15 hours");
26. session.persist(e1);
27. session.persist(e2);
28. session.persist(e3);
30. t.commit();
31. session.close();
32. System.out.println("success");
33. }
34. }

# Table Per Subclass using Annotation

As we have specified earlier, in case of table per subclass strategy, tables are created as per persistent classes but they are reated using primary and foreign key. So there will not be duplicate columns in the relation.

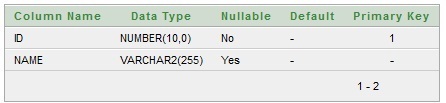
We need to specify **@Inheritance(strategy=InheritanceType.JOINED)** in the parent class and **@PrimaryKeyJoinColumn** annotation in the subclasses.

Let's see the hierarchy of classes that we are going to map.

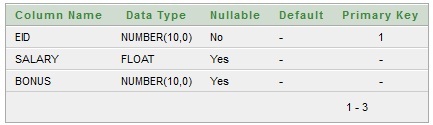
 

The table structure for each table will be as follows:

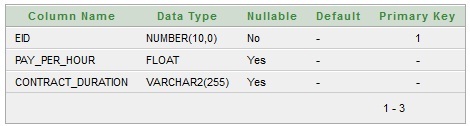
#### Table structure for Employee class

#### Table structure for Regular\_Employee class

#### Table structure for Contract\_Employee class



### Example of Table per subclass class using Annotation

In this example we are creating the three classes and provide mapping of these classes in the employee.hbm.xml file.

### 1) Create the Persistent classes

You need to create the persistent classes representing the inheritance. Let's create the three classes for the above hierarchy:

*File: Employee.java*

1. **package** com.javatpoint.mypackage;
2. **import** javax.persistence.\*;
4. @Entity
5. @Table(name = "employee103")
6. @Inheritance(strategy=InheritanceType.JOINED)
8. **public** **class** Employee {
9. @Id
10. @GeneratedValue(strategy=GenerationType.AUTO)
12. @Column(name = "id")
13. **private** **int** id;
15. @Column(name = "name")
16. **private** String name;
18. //setters and getters
19. }

*File: Regular\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **import** javax.persistence.\*;
5. @Entity
6. @Table(name="regularemployee103")
7. @PrimaryKeyJoinColumn(name="ID")
8. **public** **class** Regular\_Employee **extends** Employee{
10. @Column(name="salary")
11. **private** **float** salary;
13. @Column(name="bonus")
14. **private** **int** bonus;
16. //setters and getters
17. }

*File: Contract\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **import** javax.persistence.\*;
5. @Entity
6. @Table(name="contractemployee103")
7. @PrimaryKeyJoinColumn(name="ID")
8. **public** **class** Contract\_Employee **extends** Employee{
10. @Column(name="pay\_per\_hour")
11. **private** **float** pay\_per\_hour;
13. @Column(name="contract\_duration")
14. **private** String contract\_duration;
16. //setters and getters
17. }

### 2) create configuration file

Open the hibernate.cgf.xml file, and add an entry of mapping resource like this:

1. <mapping **class**="com.javatpoint.mypackage.Employee"/>
2. <mapping **class**="com.javatpoint.mypackage.Contract\_Employee"/>
3. <mapping **class**="com.javatpoint.mypackage.Regular\_Employee"/>

Now the configuration file will look like this:

*File: hibernate.cfg.xml*

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3. "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4. "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
6. <!-- Generated by MyEclipse Hibernate Tools.                   -->
7. <hibernate-configuration>
9. <session-factory>
10. <property name="hbm2ddl.auto">update</property>
11. <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
12. <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
13. <property name="connection.username">system</property>
14. <property name="connection.password">oracle</property>
15. <property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>
17. <mapping **class**="com.javatpoint.mypackage.Employee"/>
18. <mapping **class**="com.javatpoint.mypackage.Contract\_Employee"/>
19. <mapping **class**="com.javatpoint.mypackage.Regular\_Employee"/>
20. </session-factory>
22. </hibernate-configuration>

The hbm2ddl.auto property is defined for creating automatic table in the database.

### 3) Create the class that stores the persistent object

In this class, we are simply storing the employee objects in the database.

*File: StoreData.java*

1. **package** com.javatpoint.mypackage;
2. **import** org.hibernate.\*;
3. **import** org.hibernate.cfg.\*;
5. **public** **class** StoreData {
6. **public** **static** **void** main(String[] args) {
7. AnnotationConfiguration cfg=**new** AnnotationConfiguration();
8. Session session=cfg.configure("hibernate.cfg.xml").buildSessionFactory().openSession();
10. Transaction t=session.beginTransaction();
12. Employee e1=**new** Employee();
13. e1.setName("sonoo");
15. Regular\_Employee e2=**new** Regular\_Employee();
16. e2.setName("Vivek Kumar");
17. e2.setSalary(50000);
18. e2.setBonus(5);
20. Contract\_Employee e3=**new** Contract\_Employee();
21. e3.setName("Arjun Kumar");
22. e3.setPay\_per\_hour(1000);
23. e3.setContract\_duration("15 hours");
25. session.persist(e1);
26. session.persist(e2);
27. session.persist(e3);
29. t.commit();
30. session.close();
31. System.out.println("success");
32. }
33. }