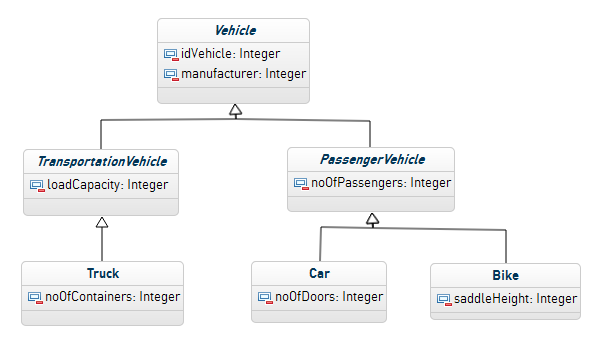
============JPA INHERITENCE====

<http://www.thejavageek.com/2014/05/14/jpa-single-table-inheritance-example/>

1. JPA Single Inheritance
2. JPA Join Inheritance

# JPA Single Table Inheritance Example:

We have object model as depicted

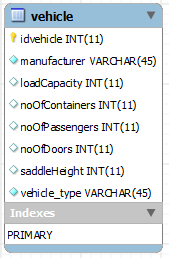
[](http://www.thejavageek.com/wp-content/uploads/2014/05/VehicleClassDiagram.png)

Class diagram for vehicles

This is what class diagram in this JPA single table inheritance example says:

* **Vehicle** is an abstract class with fields **idVehicle** and **manufacturer**.
* **TransportationVehicle** and **PassengerVehicle** are also abstract classes with fields **loadCapacity** and **noOfPassengers** respectively.
* **Truck**, **Car** and **Bike** are concrete classes with **noOfContainers**, **noOfDoors** and **saddleHeight** as their respective properties.

For this name suggests, jpa single table inheritance example uses only one table to store all the fields of all classes that come in hierarchy. We have to create a database table as below.

[](http://www.thejavageek.com/wp-content/uploads/2014/05/VehicleTable.png)

Database table for vehicle inheritance

It is obvious that

* **idVehicle** and **manufacturer** columns correspond to **Vehicle** class and these fields are inherited by all subclasses.These columns will have values for all types of vehicles, i.e. **Car**, **Truck** and **Bike**
* **loadCapacity**is field of class **TransportationVehicle** and will be inherited by **Truck.**So all trucks will have a value in row for this column.
* **noOfPassengers**is field of class **PassengerVehicle**and will be inherited by **Car** and **Bike.**So all cars and bikes will have a value for this column.
* **noOfContainers**, **noOfDoors** and **saddleHeight**will have values that are only specific to **Truck**, **Car**and **Bike** respectively as these fields are unique to them.
* There is an extra column called **vehicle\_type**in table. This is used to identify the type of object in entity model. It will store values those are specific to classes. It is called as Discriminator column.

## Discriminator Column:

It is used to identify the type of object in database. This is what javadoc says about [@DiscriminatorColumn](http://docs.oracle.com/javaee/5/api/javax/persistence/DiscriminatorColumn.html)

Is used to define the discriminator column for the [SINGLE\_TABLE](http://docs.oracle.com/javaee/5/api/javax/persistence/InheritanceType.html#SINGLE_TABLE) and [JOINED](http://docs.oracle.com/javaee/5/api/javax/persistence/InheritanceType.html#JOINED) inheritance mapping strategies.

The strategy and the discriminator column are only specified in the root of an entity class hierarchy or subhierarchy in which a different inheritance strategy is applied

For our Vehicle model, it is done using.

|  |
| --- |
| *@DiscriminatorColumn*(name = "VEHICLE\_TYPE")  public abstract class Vehicle {  //Vehicle class code  } |

Let us directly move to jpa single table inheritance example and design java classes corresponding to object model and database shown in diagram above.

|  |
| --- |
| package com.thejavageek.jpa.entities;    import javax.persistence.DiscriminatorColumn;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.Inheritance;  import javax.persistence.InheritanceType;  import javax.persistence.Table;  import javax.persistence.TableGenerator;    *@Entity*  *@Table*(name = "VEHICLE")  *@Inheritance*(strategy = InheritanceType.SINGLE\_TABLE)  *@DiscriminatorColumn*(name = "VEHICLE\_TYPE")  public abstract class Vehicle {    *@TableGenerator*(name = "VEHICLE\_GEN", table = "ID\_GEN", pkColumnName = "GEN\_NAME", valueColumnName = "GEN\_VAL", allocationSize = 1)  *@Id*  *@GeneratedValue*(strategy = GenerationType.TABLE, generator = "VEHICLE\_GEN")  private int idVehicle;  private String manufacturer;    public int getIdVehicle() {  return idVehicle;  }    public void setIdVehicle(int idVehicle) {  this.idVehicle = idVehicle;  }    public String getManufacturer() {  return manufacturer;  }    public void setManufacturer(String manufacturer) {  this.manufacturer = manufacturer;  }    } |

**Vehicle** class is abstract but it is our root entity and inheritance starts from here, hence it needs to be annotated with [@Inheritance](http://docs.oracle.com/javaee/5/api/javax/persistence/Inheritance.html).  *@Inheritance*(strategy = InheritanceType.SINGLE\_TABLE)  specifies we are using single table strategy.

* *@DiscriminatorColumn*(name = "VEHICLE\_TYPE") specifies the name of discriminator column which stores the values to identify type of entity in database.
* @GeneratedValue and @TableGenarator are used to [create automatic id using table generator.](http://www.thejavageek.com/2014/01/14/automatic-id-creation-using-jpa-table-generator/)
* You will notice [@MappedSuperClass](http://docs.oracle.com/javaee/5/api/javax/persistence/MappedSuperclass.html) annotation here:

|  |
| --- |
| package com.thejavageek.jpa.entities;  import javax.persistence.MappedSuperclass;  @MappedSuperclass  public abstract class TransportationVehicle extends Vehicle {  private int loadCapacity;  public int getLoadCapacity() {  return loadCapacity;  }  public void setLoadCapacity(int loadCapacity) {  this.loadCapacity = loadCapacity;  }  } |
| package com.thejavageek.jpa.entities;  import javax.persistence.MappedSuperclass;  @MappedSuperclass  public abstract class PassengerVehicle extends Vehicle {  private int noOfpassengers;  public int getNoOfpassengers() {  return noOfpassengers;  }  public void setNoOfpassengers(int noOfpassengers) {  this.noOfpassengers = noOfpassengers;  }  } |

## MappedSuperClass annotation:

* Classes annotated with **@MappedSuperClass** are like abstract classes in java
* They are non persistent.
* They cannot be queried over and are not mapped to any database table.
* They are only used to contribute state and behavior to entities those are inherited from them.
* It is good practice to mark abstract classes as **@MappedSuperClass** in jpa inheritance.
* In our case, annotating **TransportationVehicle** and **PassengerVehicle** as **@MappedSuperClass**enables **Truck**, **Car** and **Bike** classes to hold their state.

|  |
| --- |
| package com.thejavageek.jpa.entities;  import javax.persistence.DiscriminatorValue;  import javax.persistence.Entity;  @Entity  @DiscriminatorValue(value = "Truck")  public class Truck extends TransportationVehicle{  private int noOfContainers;  public int getNoOfContainers() {  return noOfContainers;  }  public void setNoOfContainers(int noOfContainers) {  this.noOfContainers = noOfContainers;  }  } |
| import javax.persistence.DiscriminatorValue;  import javax.persistence.Entity;  @Entity  @DiscriminatorValue(value = "Bike")  public class Bike extends PassengerVehicle {  private int saddleHeight;  public int getSaddleHeight() {  return saddleHeight;  }  public void setSaddleHeight(int saddleHeight) {  this.saddleHeight = saddleHeight;  }  } |
| package com.thejavageek.jpa.entities;  import javax.persistence.DiscriminatorValue;  import javax.persistence.Entity;  @Entity  @DiscriminatorValue(value = "Car")  public class Car extends PassengerVehicle {  private int noOfDoors;  public int getNoOfDoors() {  return noOfDoors;  }  public void setNoOfDoors(int noOfDoors) {  this.noOfDoors = noOfDoors;  }  } |

Notice the **@DisciriminatorValue** annotation in this jpa single table inheritance example.

## DiscriminatorValue annotation:

This is what javadoc says about [@DiscriminatorValue.](http://docs.oracle.com/javaee/5/api/javax/persistence/DiscriminatorValue.html)

Is used to specify the value of the discriminator column for entities of the given type. The DiscriminatorValueannotation can only be specified on a concrete entity class. If the DiscriminatorValue annotation is not specified and a discriminator column is used, a provider-specific function will be used to generate a value representing the entity type. If the [DiscriminatorType](http://docs.oracle.com/javaee/5/api/javax/persistence/DiscriminatorType.html) is [STRING](http://docs.oracle.com/javaee/5/api/javax/persistence/DiscriminatorType.html#STRING), the discriminator value default is the entity name.

In our jpa single table inheritance example, we have specified discriminator value such that when a bike instance is persisted, database row corresponding to that instance will have value “Bike” in column **vehicle\_type.**Similarly “Car” is persisted for car instance and “Truck” for a truck instance. Values from discriminator column i.e **vehicle\_type**enables jpa identify the type of object.

Now let us test this jpa single table inheritance example:

## persistence.xml:

|  |
| --- |
| <persistence version="2.1"  xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence\_2\_1.xsd">  <persistence-unit name="AdvancedMapping">  <class>com.thejavageek.jpa.entities.Vehicle</class>  <class>com.thejavageek.jpa.entities.Truck</class>  <class>com.thejavageek.jpa.entities.Bike</class>  <class>com.thejavageek.jpa.entities.Car</class>  <class>com.thejavageek.jpa.entities.PassengerVehicle</class>  <class>com.thejavageek.jpa.entities.TransportationVehicle</class>  <properties>  <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />  <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/yourDBName" />  <property name="javax.persistence.jdbc.user" value="yourDBUserName" />  <property name="javax.persistence.jdbc.password" value="userDBPassword" />  </properties>  </persistence-unit>  </persistence> |
| Code to insert entities |
| /\* Create EntityManagerFactory \*/  EntityManagerFactory emf = Persistence  .createEntityManagerFactory("AdvancedMapping");  /\* Create EntityManager \*/  EntityManager em = emf.createEntityManager();  EntityTransaction transaction = em.getTransaction();  transaction.begin();  Bike cbr1000rr = new Bike();  cbr1000rr.setManufacturer("honda");  cbr1000rr.setNoOfpassengers(1);  cbr1000rr.setSaddleHeight(30);  em.persist(cbr1000rr);  Car avantador = new Car();  avantador.setManufacturer("lamborghini");  avantador.setNoOfDoors(2);  avantador.setNoOfpassengers(2);  em.persist(avantador);  Truck truck = new Truck();  truck.setLoadCapacity(100);  truck.setManufacturer("mercedes");  truck.setNoOfContainers(2);  em.persist(truck);  transaction.commit(); |

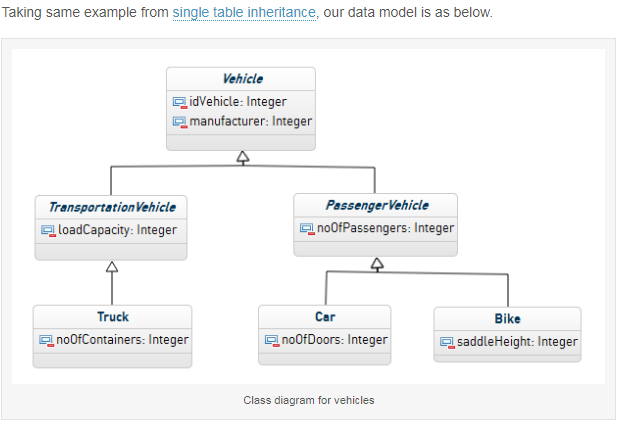
## Run this code and you will see data inserted into database

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| idVehicle | manufacturer | loadCapacity | noOfContainers | noOfPassengers | noOfDoors | saddleHeight | vehicle\_type |
| 6 | honda |  |  | 1 |  | 30 | Bike |
| 7 | lamborghini |  |  | 2 | 2 |  | Car |
| 8 | mercedes | 100 | 2 |  |  |  | Truck |

The advantage of single table strategy is it does not require complex joins for retrieval and insertion of entities, but on the other hand it wastes database space as many columns need to be nullable and there isn’t any data for them.

I hope this article has helped understand single table strategy using jpa single table inheritance example. Please feel free to provide feedback and suggest improvements.

JPA Joined Inheritance Example:



Class diagram for JPA joined inheritance example says:

* **Vehicle** is an abstract class with fields **idVehicle** and **manufacturer**.
* **TransportationVehicle** and **PassengerVehicle** are also abstract classes with fields **loadCapacity** and **noOfPassengers** respectively.
* **Truck**, **Car** and **Bike** are concrete classes with **noOfContainers**, **noOfDoors** and **saddleHeight** as their respective properties.

Now let us create java classes for above diagram.

## Vehicle.java

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | |  | package com.thejavageek.jpa.entities;    import javax.persistence.DiscriminatorColumn;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.Inheritance;  import javax.persistence.InheritanceType;  import javax.persistence.Table;  import javax.persistence.TableGenerator;    *@Entity*  *@Table*(name = "VEHICLE")  *@Inheritance*(strategy = InheritanceType.JOINED)  Is used to define the discriminator column for the [SINGLE\_TABLE](https://docs.oracle.com/javaee/5/api/javax/persistence/InheritanceType.html#SINGLE_TABLE) and [JOINED](https://docs.oracle.com/javaee/5/api/javax/persistence/InheritanceType.html#JOINED) inheritance mapping strategies.  *@DiscriminatorColumn*(name = "VEHICLE\_TYPE")  public abstract class Vehicle {    *@TableGenerator*(name = "VEHICLE\_GEN", table = "ID\_GEN", pkColumnName = "GEN\_NAME", valueColumnName = "GEN\_VAL", allocationSize = 1)  *@Id*  *@GeneratedValue*(strategy = GenerationType.TABLE, generator = "VEHICLE\_GEN")  private int idVehicle;  private String manufacturer;    public int getIdVehicle() {  return idVehicle;  }    public void setIdVehicle(int idVehicle) {  this.idVehicle = idVehicle;  }    public String getManufacturer() {  return manufacturer;  }    public void setManufacturer(String manufacturer) {  this.manufacturer = manufacturer;  }    } | |

* @Inheritance(strategy = InheritanceType.JOINED) specifies we are creating joined inheritance.
* @DiscriminatorColumn(name = "VEHICLE\_TYPE") specifies database column that stores information about type of entities. You can refer [javadoc](http://docs.oracle.com/cd/E17802_01/products/products/persistence/javadoc-1_0-fr/javax/persistence/DiscriminatorColumn.html).
* @TableGenerator and @GeneratedValue are used for [automatic id creation in jpa](http://www.thejavageek.com/2014/01/14/automatic-id-creation-using-jpa-table-generator/).

## TransportationVehicle.java

|  |
| --- |
| package com.thejavageek.jpa.entities;  import javax.persistence.MappedSuperclass;  @MappedSuperclass  public abstract class TransportationVehicle extends Vehicle {  private int loadCapacity;  public int getLoadCapacity() {  return loadCapacity;  }  public void setLoadCapacity(int loadCapacity) {  this.loadCapacity = loadCapacity;  }  } |

|  |  |
| --- | --- |
| Truck.java | PassengerVehicle.java |
| package com.thejavageek.jpa.entities;  import javax.persistence.Entity;  @Entity  public class Truck extends TransportationVehicle {  private int noOfContainers;  public int getNoOfContainers() {  return noOfContainers;  }  public void setNoOfContainers(int noOfContainers) {  this.noOfContainers = noOfContainers;  }  } | package com.thejavageek.jpa.entities;  import javax.persistence.MappedSuperclass;  @MappedSuperclass  public abstract class PassengerVehicle extends Vehicle {  private int noOfpassengers;  public int getNoOfpassengers() {  return noOfpassengers;  }  public void setNoOfpassengers(int noOfpassengers) {  this.noOfpassengers = noOfpassengers;  }  } |

|  |  |
| --- | --- |
| Car.java | Bike.java |
| package com.thejavageek.jpa.entities;  import javax.persistence.Entity;  @Entity  public class Car extends PassengerVehicle {  private int noOfDoors;  public int getNoOfDoors() {  return noOfDoors;  }  public void setNoOfDoors(int noOfDoors) {  this.noOfDoors = noOfDoors;  }  } | package com.thejavageek.jpa.entities;  import javax.persistence.Entity;  @Entity  public class Bike extends PassengerVehicle {  private int saddleHeight;  public int getSaddleHeight() {  return saddleHeight;  }  public void setSaddleHeight(int saddleHeight) {  this.saddleHeight = saddleHeight;  }  } |

Things to notice here:

**Vehicle** is an abstract class but every vehicle needs to be identified by jpa hence it should have an id and hence it becomes an entity that has representation in database. So database will have a table mapped to **Vehicle** class.

**TransportationVehicle** and **PassengerVehicle**are abstract but we have mapped them as [@MappedSuperClass.](http://docs.oracle.com/cd/E17802_01/products/products/persistence/javadoc-1_0-fr/javax/persistence/MappedSuperclass.html) They will have not separate table mapped in database. But they have fields **loadCapacity** and **noOfPassengers**respectively. So we need to create them in database tables those are mapped to subclasses.

So, there will be a table named **Truck** with column **loadCapacity**(represents  field inherited from mapped superclass **TransportationVehicle)** and **noOfContainers**.

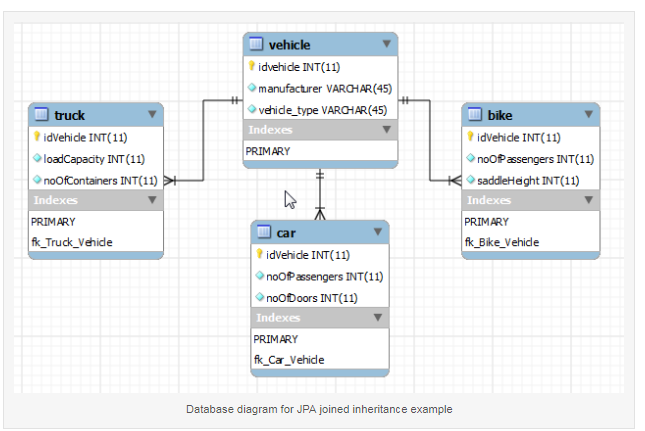
A table named **Car** with column **noOfPassengers**(represents field inherited from mapped superclass **PassengerVehicle**) and **noOfDoors**.

A table named **Bike** with column **noOfPassengers**(represents field inherited from mapped superclass **Passengervehicle**) and **saddleHeight**.

Now there are four tables, **Vehicle**, **Truck**, **Car** and **Bike**. As the name JPA joined inheritance example obviously suggests, they need to be joined so that inheritance relationship is maintained.

Hence, subclass entities will have primary keys that will refer to primary keys of parent i.e. Vehicle class using foreign key relationship.

Now we will create database for this jpa joined inheritance example.



**Vehicle** table stores information about **Vehicle** class.

**Truck** table stores information about **Truck** class with inherited field from **TransportationVehicle**

**Car** and **Bike** tables store information about themselves and inherited field from **PassengerVehicle**

Primary keys of **Truck**, **Car** and **Bike** class are actually foreign keys that refer to primary keys of **Vehicle**. so subclass specific information can be accessed from superclass using foreign keys.

|  |  |
| --- | --- |
| persistence.xml: |  |
| <?xml version="1.0" encoding="UTF-8"?>  <persistence version="2.1"  xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence\_2\_1.xsd">  <persistence-unit name="AdvancedMapping">  <class>com.thejavageek.jpa.entities.Vehicle</class>  <class>com.thejavageek.jpa.entities.Truck</class>  <class>com.thejavageek.jpa.entities.Bike</class>  <class>com.thejavageek.jpa.entities.Car</class>  <class>com.thejavageek.jpa.entities.PassengerVehicle</class>  <class>com.thejavageek.jpa.entities.TransportationVehicle</class>  <properties>  <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />  <property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/yourDBName" />  <property name="javax.persistence.jdbc.user" value="yourDBUserName" />  <property name="javax.persistence.jdbc.password" value="userDBPassword" />  </properties>  </persistence-unit>  </persistence> | /\* Create EntityManagerFactory \*/  EntityManagerFactory emf = Persistence  .createEntityManagerFactory("AdvancedMapping");  /\* Create EntityManager \*/  EntityManager em = emf.createEntityManager();  EntityTransaction transaction = em.getTransaction();  transaction.begin();  Bike cbr1000rr = new Bike();  cbr1000rr.setManufacturer("honda");  cbr1000rr.setNoOfpassengers(1);  cbr1000rr.setSaddleHeight(30);  em.persist(cbr1000rr);  Car aventador = new Car();  aventador.setManufacturer("lamborghini");  aventador.setNoOfDoors(2);  aventador.setNoOfpassengers(2);  em.persist(aventador);  Truck truck = new Truck();  truck.setLoadCapacity(1000);  truck.setManufacturer("volvo");  truck.setNoOfContainers(2);  em.persist(truck);  transaction.commit(); |

This will insert data into tables as below.

## Vehicle Table:

|  |  |  |
| --- | --- | --- |
| idvehicle | manufacturer | vehicle\_type |
| 9 | honda | Bike |
| 10 | lamborghini | Car |
| 11 | volvo | Truck |

## Bike Table:

|  |  |  |
| --- | --- | --- |
| idvehicle | noOfPassengers | saddleHeight |
| 9 | 1 | 30 |

## Car Table:

|  |  |  |
| --- | --- | --- |
| idvehicle | noOfPassengers | noOfDoors |
| 10 | 2 | 2 |

## Truck Table:

|  |  |  |
| --- | --- | --- |
| idvehicle | loadCapacity | noOfContainers |
| 11 | 1000 | 2 |

The advantage of joined inheritance strategy is that it does not waste database space as in single table strategy. On the other hand, because of multiple joins involved for every insertion and retrieval, performance becomes and issue when inheritance hierarchies become wide and deep.

I hope this article helped understand jpa joined inheritance example.

What is Hibernate O/R Mapping

**Hello Friends In Last chapters you learned about how to write our first hibernate application, there are two approach we have used for that—**

1. **Using Annotation for Model Class Object Mapping**
2. **Another way Using Hibernate O/R Mapping file(.hbm.xml) for Model Class Object Mapping.**

**In the last example we created userDetails.hbm.xml to map UserDetails Object to the UserDetails table in the database. Now let’s understand the each component of the mapping file.**

**To recall here is the content of userDetails.hbm.xml:**

**<?xml version=”1.0″?>**

**<!DOCTYPE hibernate-mapping PUBLIC**

**“-//Hibernate/Hibernate Mapping DTD 3.0//EN”**

**“http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd”>**

**<hibernate-mapping>**

**<class name=”com.sdnext.hibernate.tutorial.dto.UserDetails” table=”UserDetails”>**

**<id name=”userId” type=”long” column=”ID” >**

**<generator class=”assigned”/>**

**</id>**

**<property name=”userName”>**

**<column name=”UserName” />**

**</property>**

**</class>**

**</hibernate-mapping>**

1.<hibernate-mapping> element:

The first or root element of hibernate mapping document is <hibernate-mapping> element. Between the <hibernate-mapping> tag class element(s) are present.

2. <class> element:

The <Class> element maps the class object with corresponding entity in the database. It also tells what table in the database has to access and what column in that table it should use. Within one <hibernate-mapping> element, several <class> mappings are possible.

3.<id> element:

The <id> element in unique identifier to identify an object. In fact <id> element map with the primary key of the table. In our code :

<id name=”userId” type=”long” column=”ID” >

primary key maps to the ID field of the table UserDetails. Following is the attributes of <id> element

name – Property name of the persistence model class

type – The java data type used for that property

column – Name of the column for that property of the persistence object

unsaved-value – This is the value used to determine if a class has been made persistent. If the value of the id attribute is null, then it means that this object has not been persisted.

4.<generator> element:

Used to create primary key for new record, there are some commonly used generators type given below…

Increment- used to generate primary keys of type long, short or int that are unique only.

Sequence – used to generate primary keys for DB2, Oracle, SAP Database.

Assigned – is used when application code generates the primary key.

Native – selects identity, sequence or hilo depending upon the capabilities of the underlying db.

Identity – supports identity columns in DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL. The returned identifier is of type long, short or int.

Uuid – Unique use ID of 128 bits generated from using algorithm and return type is String

hilo – generated by the hi/lo Algorithm

seqhilo – generated by the hi/lo Algorithm according to sequence of database

select – select from database triggered value

foreign – associated with the other model objects

5.<property> element: define standard Java attributes and their mapping into database schema.

That is all about the mapping file for O/R mapping. I hope you was understand its all elements and its working in the hibernate.

## Mapping:

* Mapping file is the heart of hibernate application.
* Every ORM tool needs this mapping, mapping is the mechanism of placing an object properties into column’s of a table.
* Mapping can be given to an ORM tool either in the form of an XML or in the form of the annotations.
* The mapping file contains mapping from a pojo class name to a table name and pojo class variable names to table column names.
* While writing an hibernate application, we can construct one or more mapping files, mean a hibernate application can contain any number of  mapping files.

generally an object contains 3 properties like

* Identity (Object Name)
* State (Object values)
* Behavior (Object Methods)

Actually mapping can be done using 2 ways,

* XML
* Annotations.
* **Now we will look how to relate  XML Mapping to Annotation**  
  Mapping a class **UserDetails**to Table **USER\_DETAIL in XML —**
* <class name=”com.sdnext.hibernate.tutorial.dto.UserDetails” table=”USER\_DETAIL”>
* Now Mapping a class **UserDetails**to Table **USER\_DETAIL in Annotation —**
* @Entity  
  @Table (name=”USER\_DETAIL”)  
  public class UserDetails{}
* here**@Entity** declares the class as an entity (i.e. a persistent POJO class)  
  **@Table** is set at the class level; it allows you to define the table, catalog, and schema names for your entity mapping. If no **@Table** is defined the default values are used: the unqualified class name of the entity.
* Mapping **primary key** **USER\_ID** of table to **property userId** of class **UserDetails in XML**
* **<id name=”userId” type=”long” column=”USER\_ID” >**
* Mapping **primary key USER\_ID** of table to **property userId** of class **UserDetails in Annotation**
* @Entity  
  @Table (name=”USER\_DETAILS”)  
  public class UserDetails  
  {  
  @Id  
  @Column(name=”USER\_ID”)  
  private long userId;  
  **}**
* here **@Id** declares the identifier property of this entity. The class **UserDetails**is mapped to the **USER\_TABLE** table, using the column **USER\_ID** as its primary key column.  
  The column(s) used for a property mapping can be defined using the **@Column** annotation. Use it to override default values .
* **Id**Generator Class**Mapping**in **XML**

<id name=”userId” type=”long” column=”USER\_ID” >  
<generator class=”auto”/>  
</id>

**Id**Generator Class **Mapping**in **Annotation**

@Id  
@Column(name=”USER\_ID”)  
@GeneratedValue(strategy=GenerationType.AUTO)  
private long  userId;

Different GenerationType Enum Properties….

* AUTO – either identity column, sequence or table depending on the underlying DB
* TABLE – table holding the id
* IDENTITY – identity column
* SEQUENCE – sequence
* identity copy – the identity is copied from another entity

**@GeneratedValue**Provides for the specification of generation strategies for the values of primary keys.  
**Enum GenerationType** Defines the types of primary key generation strategies.

Mapping **Column**to the **property**of class in **XML**

<property name=”userName” column=”USER\_NAME”>

Mapping **Column**to the **property**of class in **Annotation**

**@Column(name=”USER\_NAME”)  
  private String userName;**

**@Column**– provides the name of the column in a table if it is different from the attribute name. (By default, the two names are assumed to be the same.)  
**More JPA  Annotations for Class Field:**  
**@Basic –**The use of the **Basic**annotation is optional for persistent fields and properties of these types. If the **Basic**annotation is not specified for such a field or property, the default values of the **Basic**annotation will apply. Example-

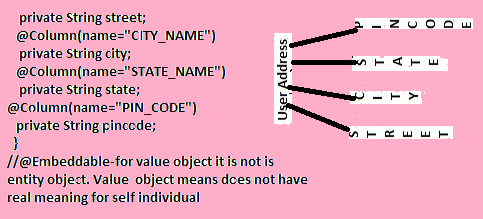
**@Basic**

**private String userName;**

**@Transient** – using when if you want skip any field of entity class to save in the database. Example-  
**@Transient**  
**private String middleName;**

**@Embedded-**using when if property or field of persistence class is Embeddable  persistence class.  
Example- **class Address{**

**@Column(name=”STREET”)**



**@Embedded**  
**private Address address;**

**@ElementColllection-**Defines a collection of instances of a basic type or embeddable class. Must be specified if the collection is to be mapped by means of a collection table. Example-  
**@ElementCollection**  
**private Collection<Address> lisOfAddresses = new ArrayList<Address>();**

**@Id-**Specifies the primary key of an entity. Example-  
**@Id**  
**private long userId;**

**@EmbeddedId-**composite primary key of an embeddable class.**Example-**  
**@Embeddable**  
**Class Address{**  
**@EmbeddedId**  
**private int addressId;**  
**—-**  
**—-**  
**}**

**@Version-**Specifies the version field or property of an entity class that serves as its optimistic lock value. The version is used to ensure integrity when performing the merge operation and for optimistic concurrency control. Example-  
**@Version**  
**private int addressId;**

**@Temporal-**This annotation must be specified for persistent fields or properties of type **java.util.Date** and **java.util.Calendar**. Example-  
**@Column(name=”JOIN\_DATE”)  
@Temporal(TemporalType.DATE)  
private Date joinDate;**

Hibernate Projection:

<http://www.onlinetutorialspoint.com/hibernate/hibernate-projection-example.html>

## Projections & Aggregations

The Criteria API provides the **org.hibernate.criterion.Projections** class, which can be used to get average, maximum, or minimum of the property values. The Projections class is similar to the Restrictions class, in that it provides several static factory methods for obtaining **Projection** instances.

Following are the few examples covering different scenarios and can be used as per requirement −

Criteria cr = session.createCriteria(Employee.class);

// To get total row count.

cr.setProjection(Projections.rowCount());

// To get average of a property.

cr.setProjection(Projections.avg("salary"));

// To get distinct count of a property.

cr.setProjection(Projections.countDistinct("firstName"));

// To get maximum of a property.

cr.setProjection(Projections.max("salary"));

// To get minimum of a property.

cr.setProjection(Projections.min("salary"));

// To get sum of a property.

cr.setProjection(Projections.sum("salary"));