

What is ORM?

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

An ORM system has the following advantages over plain JDBC –

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

An ORM solution consists of the following four entities –

Sr.No	Solutions
1	An API to perform basic CRUD operations on objects of persistent classes.
2	A language or API to specify queries that refer to classes and properties of classes.
3	A configurable facility for specifying mapping metadata.
4	A technique to interact with transactional objects to perform dirty checking, lazy association fetching, and other optimization functions.

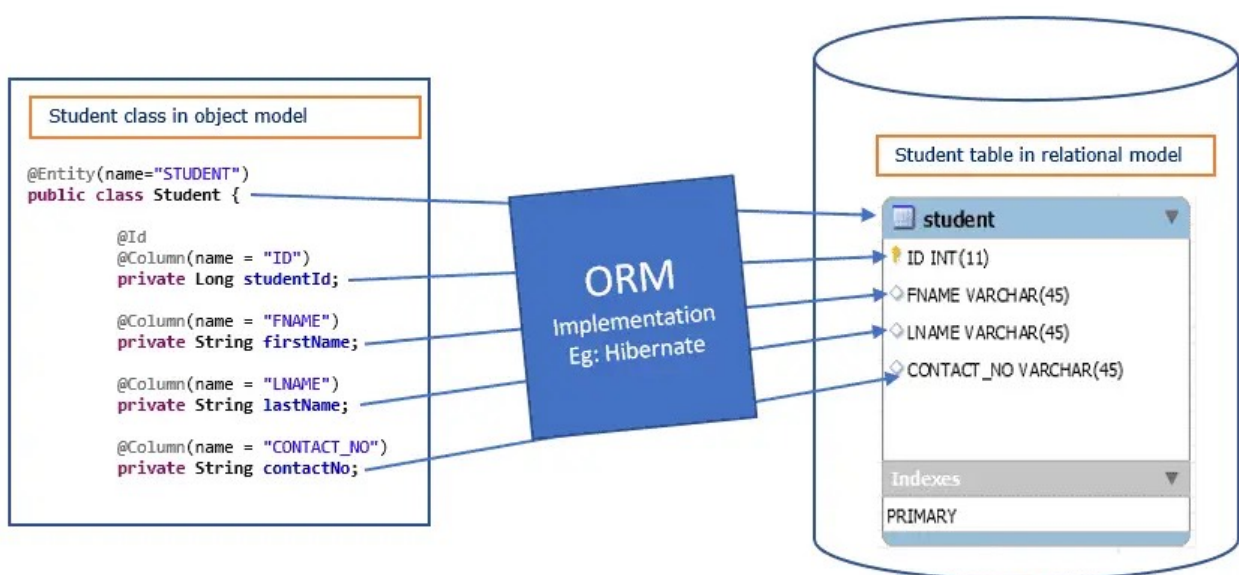
Java ORM Frameworks

There are several persistent frameworks and ORM options in Java. A persistent framework is an ORM service that stores and retrieves objects into a relational database.

- Enterprise JavaBeans Entity Beans
- Java Data Objects
- Castor
- TopLink
- Spring DAO
- Hibernate

1. What is Object Relational Mapping (ORM)?

ORM or **Object Relational Mapping** is a system that implements the responsibility of mapping the Object to Relational Model. That means it is responsible to store **Object Model** data into **Relational Model** and further read the data from Relational Model into Object Model.



ORM implements responsibility of mapping the Object to Relational Model.

2. Why **ORM** (Object Relational Mapping)?

When work with object oriented programming to persist data in RDBMS, there is mismatches between object model and relational model if work with traditional techniques like JDBC. **ORM** fills gap in the following mismatches between Object model and relational model.

3. Impedance Mismatch between Object Model and Relational Model

The **Object Oriented** (Domain) model use classes whereas the relational database use tables. This creates a gap (The Impedance Mismatch). Due to the difference between the two different models, getting the data and associations from objects into relational table structure and vice versa requires a lot of tedious programming.

Hibernate Advantages

- Hibernate takes care of mapping Java classes to database tables using XML files and without writing any line of code.
- Provides simple APIs for storing and retrieving Java objects directly to and from the database.
- If there is change in the database or in any table, then you need to change the XML file properties only.
- Abstracts away the unfamiliar SQL types and provides a way to work around familiar Java Objects.
- Hibernate does not require an application server to operate.
- Manipulates Complex associations of objects of your database.
- Minimizes database access with smart fetching strategies.
- Provides simple querying of data.

Supported Databases

Hibernate supports almost all the major RDBMS. Following is a list of few of the database engines supported by Hibernate –

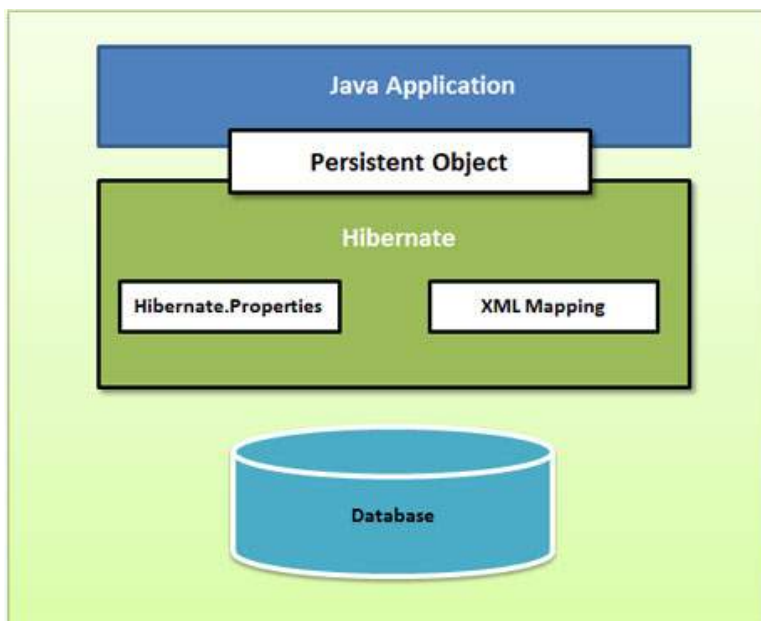
- HSQL Database Engine
- DB2/NT
- MySQL
- PostgreSQL
- FrontBase
- Oracle
- Microsoft SQL Server Database
- Sybase SQL Server
- Informix Dynamic Server

Supported Technologies

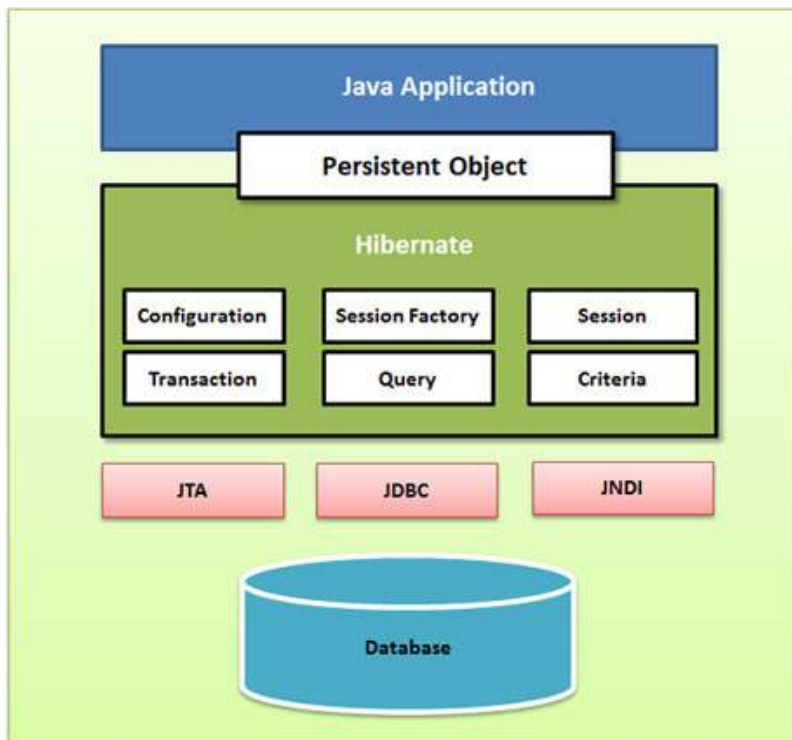
Hibernate supports a variety of other technologies, including –

- XDoclet Spring
- J2EE
- Eclipse plug-ins
- Maven

Following is a very high level view of the Hibernate Application Architecture.



Following is a detailed view of the Hibernate Application Architecture with its important core classes.



Hibernate Example 1

```
package com.example.hibernate;
```

```
import javax.persistence.Entity;  
import javax.persistence.Id;
```

```
@Entity  
public class Student {
```

```
    @Id  
    private int id;  
    private String name;  
    private String city;  
    public Student() {  
        super();  
        // TODO Auto-generated constructor stub  
    }  
    private Student(int id, String name, String city) {  
        super();  
        this.id = id;
```

```

        this.name = name;
        this.city = city;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    @Override
    public String toString() {
        // TODO Auto-generated method stub
        return this.id+" : "+this.name+" : "+this.city ;
    }
}

```

hibernate.cfg.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-
configuration-3.0.dtd">
    <hibernate-configuration>
        <session-factory>

```

```

        <property
name="connection.driver_class">com.mysql.cj.jdbc.Driver</
property>
        <property name="connection.url">jdbc:mysql://
localhost:3306/test</property>
        <property name="connection.username">root</
property>
        <property
name="connection.password">Remember001</property>
        <property
name="dialect">org.hibernate.dialect.MySQL5Dialect</
property>
        <property name="hbm2ddl.auto">create</property>
        <property name="show_sql">true</property>
        <mapping class="com.example.hibernate.Student"/>
        <mapping class="com.example.hibernate.Address"/>
        </session-factory>
    </hibernate-configuration>

```

```

package com.example.hibernate;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Date;

```

```

import
javax.transaction.HeuristicMixedExceptio
n;
import
javax.transaction.HeuristicRollbackExcep
tion;
import
javax.transaction.RollbackException;
import
javax.transaction.SystemException;

```

```

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;;

public class App {

    public static void main(String[] args) throws
SecurityException, RollbackException,
HeuristicMixedException, HeuristicRollbackException,
SystemException, IOException {
        System.out.println("Project Started..");
        Configuration cfg = new Configuration();
        cfg.configure("hibernate.cfg.xml");

        SessionFactory factory =
cfg.buildSessionFactory();
        //System.out.println(factory);
        //System.out.println(factory.isClosed());
    }
}

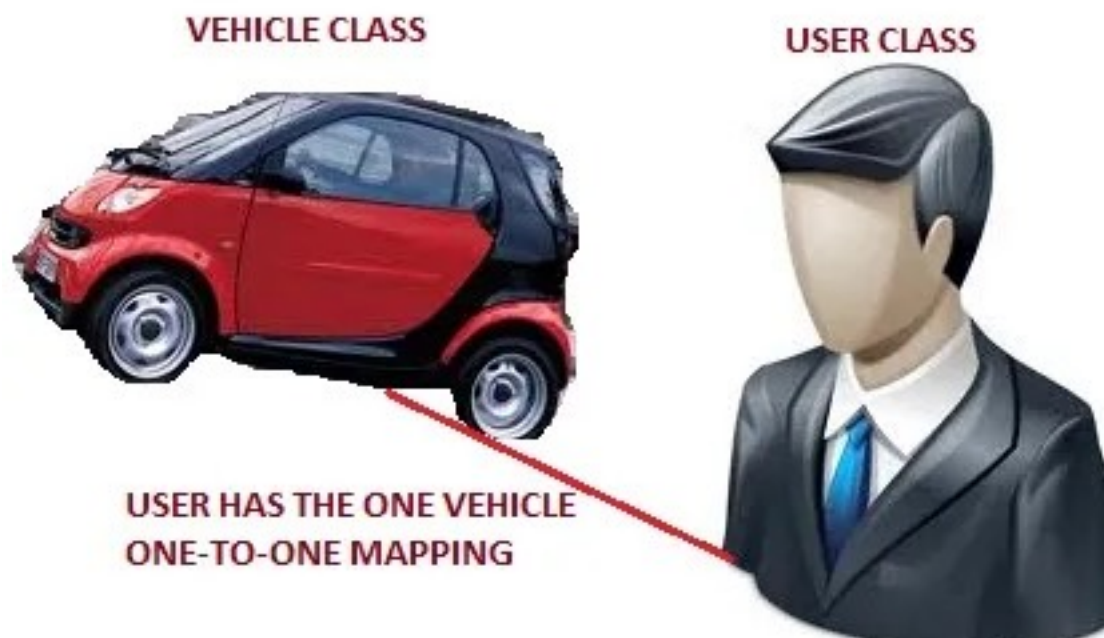
```

#	Method name	Return	Description	Issued SQL statement
1	beginTransaction()	Transaction	Creates a Transaction object or returns an existing one, for working under context of a transaction.	
2	getTransaction()	Transaction	Returns the current transaction.	
3	get(Class class, Serializable id)	Object	Loads a persistent instance of the given class with the given id, into the session.	SELECT

4	<code>load(Class class, Serializable id)</code>	Object	Does same thing as <code>get()</code> method, but throws an <code>ObjectNotFoundException</code> error if no row with the given id exists.	SELECT
5	<code>persist(Object)</code>	void	saves a mapped object as a row in database	INSERT
6	<code>save(Object)</code>	Serializable	Does same thing as <code>persist()</code> method, plus returning a generated identifier.	INSERT
7	<code>update(Object)</code>	void	Updates a detached instance of the given object and the underlying row in database.	UPDATE
8	<code>saveOrUpdate(Object)</code>	void	Saves the given object if it does not exist, otherwise updates it.	INSERT or UPDATE
9	<code>delete(Object)</code>	void	Removes a persistent object and the underlying row in database.	DELETE
10	<code>close()</code>	void	Ends the current session.	
11	<code>flush()</code>	void	Flushes the current session. This method should be called before committing the transaction and closing the session.	
12	<code>disconnect()</code>	void	Disconnects the session from current JDBC connection.	

One to One Mapping in Hibernate Example

In this tutorial of One to One Mapping in Hibernate Example we will learning what happens when an entity class has the field of the entity type object. Means that one entity is inside the one entity known as One-2-One Mapping.

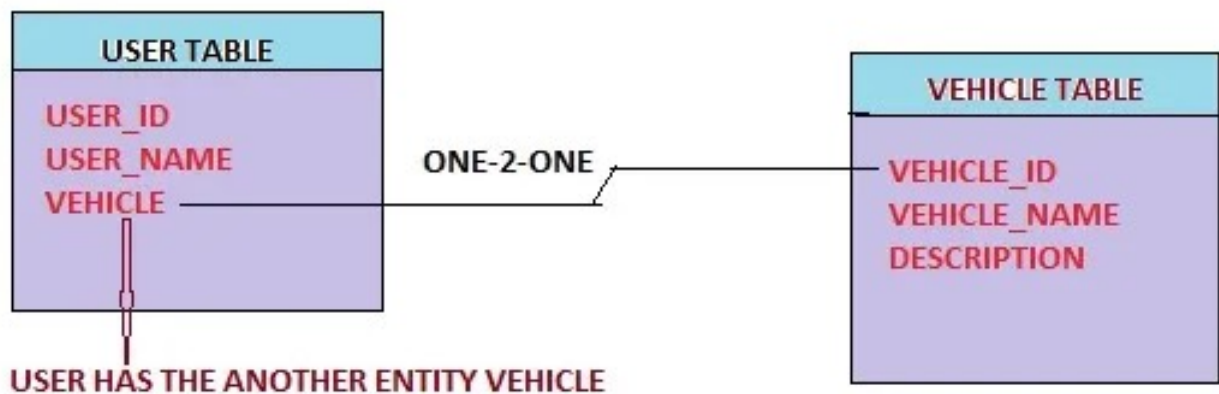


Hibernate Mapping One-to-One

In this example you will learn how to map one-to-one relationship using Hibernate. Consider the following relationship between UserDetails and Vehicle entity.



According to the relationship each user should have a unique vehicle.



For that we will use the following annotation.

@OneToOne:

Target:

Fields (including property get methods) Defines a single-valued association to another entity that has one-to-one multiplicity. It is not normally necessary to specify the associated target entity explicitly since it can usually be inferred from the type of the object being referenced. If the relationship is bidirectional, the non-owning side must use the mappedBy element of the **@OneToOne** annotation to specify the relationship field or property of the owning side. The **@OneToOne** annotation may be used within an embeddable class to specify a relationship from the embeddable class to an entity class.

Now we look the following Example related to the One to One mapping.

1. First Create Vehicle Class

Vehicle.java

```

package com.hibernate.dto;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="VEHICLE")
public class Vehicle
  
```

```

{
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name="VEHICLE_ID")
private int vehicleId;

    @Column(name="VEHICLE_NAME")
private String vehicleName;

public int getVehicleId() {
return vehicleId;
}
public void setVehicleId(int vehicleId) {
this.vehicleId = vehicleId;
}
public String getVehicleName() {
return vehicleName;
}
public void setVehicleName(String vehicleName) {
this.vehicleName = vehicleName;
}
}
}

```

2. Create the User Class

UserDetails.java

```

package com.hibernate.dto;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import javax.persistence.Table;

```

```

@Entity
@Table (name="USER_DETAIL")
public class UserDetails
{
    @Id
    @Column(name="USER_ID")
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int    userId;

    @Column(name="USER_NAME")
    private String userName;

    @OneToOne
        @JoinColumn(name="VEHICLE_ID")
    private Vehicle vehicle;

    public Vehicle getVehicle() {
        return vehicle;
    }
    public void setVehicle(Vehicle vehicle) {
        this.vehicle = vehicle;
    }
    public int getUserId() {
        return userId;
    }
    public void setUserId(int userId) {
        this.userId = userId;
    }
    public String getUserName() {
        return userName;
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
}

```

3. Create the hibernate configuration file.

hibernate.cfg.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE hibernate-configuration PUBLIC  
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"  
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
```

```
<hibernate-configuration>  
<session-factory>  
<!-- Database connection settings -->  
<property name="connection.driver_class">com.mysql.jdbc.Driver</property>  
<property name="connection.url">jdbc:mysql://localhost:3306/hibernateDB</  
property>  
<property name="connection.username">root</property>  
<property name="connection.password">root</property>
```

```
<!-- JDBC connection pool (use the built-in) -->  
<property name="connection.pool_size">1</property>
```

```
<!-- SQL dialect -->  
<property name="dialect">org.hibernate.dialect.MySQLDialect</property>
```

```
<!-- Enable Hibernate's automatic session context management -->  
<property name="current_session_context_class">thread</property>
```

```
<!-- Disable the second-level cache -->  
<property name="cache.provider_class"  
>org.hibernate.cache.NoCacheProvider</property>
```

```
<!-- Echo all executed SQL to stdout -->  
<property name="show_sql">true</property>
```

```
<!-- Drop and re-create the database schema on startup -->  
<property name="hbm2ddl.auto">create</property>
```

```
<mapping class="com.hibernate.dto.UserDetails"/>  
<mapping class="com.hibernate.dto.Vehicle"/>
```

```
</session-factory>
</hibernate-configuration>
```

4. Create Test Demo class for run this code.

HibernateTestDemo.java

```
package com.hibernate;
```

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;
```

```
import com.hibernate.dto.UserDetails;
import com.hibernate.dto.Vehicle;
```

```
public class HibernateTestDemo {
/**
```

```
* @param args
*/
```

```
public static void main(String[] args)
{
```

```
UserDetails user = new UserDetails(); //create the user entity
Vehicle vehicle = new Vehicle(); //create the vehicle entity
```

```
vehicle.setVehicleName("BMW Car"); //set vehicle name
```

```
user.setUserName("Dinesh Rajput"); //set the user name
user.setVehicle(vehicle); //set the vehicle entity to the field of the user entity
i.e. vehicle entity inside the user entity
```

```
SessionFactory sessionFactory = new
AnnotationConfiguration().configure().buildSessionFactory(); //create session
factory object
Session session = sessionFactory.openSession(); //create the session object
session.beginTransaction(); //create the transaction from the session object
```

```
session.save(vehicle); // save the vehicle entity to the database
session.save(user); // save the user entity to the database
```

```
session.getTransaction().commit(); //close the transaction
session.close(); //close the session
}
}
```

OUTPUT:

```
log4j:WARN No appenders could be found for logger
(org.hibernate.cfg.annotations.Version).
log4j:WARN Please initialize the log4j system properly.
Hibernate: insert into VEHICLE (VEHICLE_NAME) values (?)
Hibernate: insert into USER (USER_NAME, VEHICLE_ID) values (?, ?)
```

```
<terminated> HibernateTestDemo (2) [Java Application] C:\Program Files (x86)\Java\jdk1.6.0_23\bin\javaw.exe (29-Apr-20
log4j:WARN No appenders could be found for logger (org.hibernate.cfg.annotations.Version).
log4j:WARN Please initialize the log4j system properly.
Hibernate: insert into VEHICLE (VEHICLE_NAME) values (?)
Hibernate: insert into USER (USER_NAME, VEHICLE_ID) values (?, ?)
```

Now we look the created tables for that.

USER_ID	USER_NAME	VEHICLE_ID	FK
<input type="checkbox"/> 1	Dinesh Rajput	1	
*	(NULL)	(NULL)	

USER TABLE

PK

VEHICLE_ID	VEHICLE_NAME
<input type="checkbox"/> 1	BMW Car
*	(NULL)

VEHICLE TABLE

PK

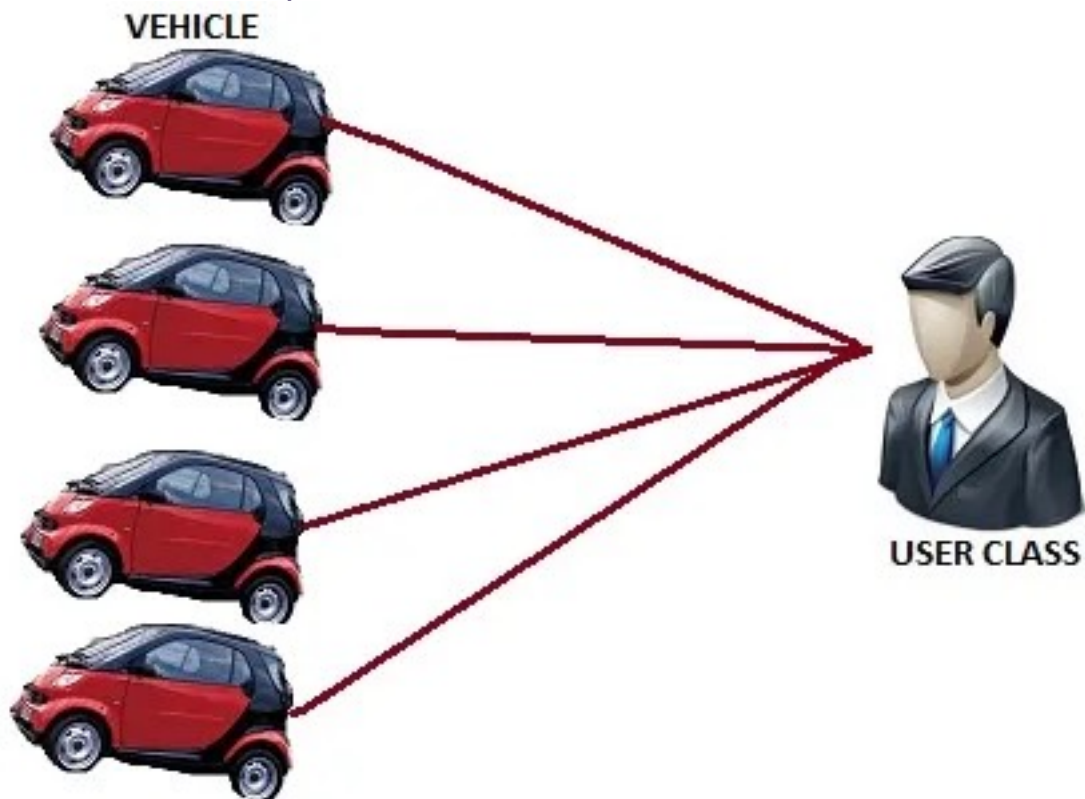
ONE-TO-ONE

One to Many Mapping in Hibernate Example

In this tutorial of one to many mapping in hibernate example we will discuss about the One To Many Mapping. A one-to-many relationship occurs when one entity is related to many occurrences in another entity.

In this chapter you will learn how to map one-to-many relationship using Hibernate. Consider the following relationship between UserDetails Class and Vehicle entity.

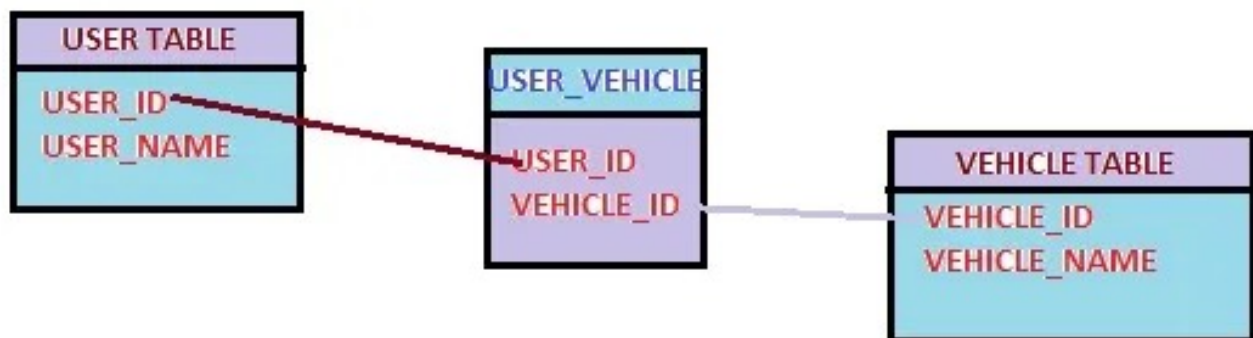
One user have the multiple vehicles.



Class diagram for that given below.



In this example UserDetails class has the collection of the another entity class Vehicle. So the given below diagram so table structure for that.



For that we will use the following annotation.

@OneToMany:

Target:

Fields (including property get methods) Defines a many-valued association with one-to-many multiplicity. If the collection is defined using generics to specify the element type, the associated target entity type need not be specified; otherwise the target entity class must be specified. If the relationship is bidirectional, the mappedBy element must be used to specify the relationship field or property of the entity that is the owner of the relationship. The **OneToMany** annotation may be used within an **embeddable** class contained within an entity class to specify a relationship to a collection of entities. If the relationship is bidirectional, the mappedBy element must be used to specify the relationship field or property of the entity that is the owner of the relationship.

Now we look the following Example related to the One to Many mapping.

1. First Create Vehicle Class

Vehicle.java

```
package com.hibernate.dto;
```

```

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="VEHICLE")
public class Vehicle
{
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="VEHICLE_ID")
    private int vehicleId;

    @Column(name="VEHICLE_NAME")
    private String vehicleName;

    public int getVehicleId() {
        return vehicleId;
    }
    public void setVehicleId(int vehicleId) {
        this.vehicleId = vehicleId;
    }
    public String getVehicleName() {
        return vehicleName;
    }
    public void setVehicleName(String vehicleName) {
        this.vehicleName = vehicleName;
    }
}

```

2. Create the User Class

UserDetails.java

```

package com.hibernate.dto;

import java.util.ArrayList;
import java.util.Collection;

```

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.OneToMany;
import javax.persistence.Table;
```

```
@Entity
```

```
@Table (name="USER")
```

```
public class UserDetails
```

```
{
```

```
@Id
```

```
@Column(name="USER_ID")
```

```
@GeneratedValue(strategy=GenerationType.AUTO)
```

```
private int    userId;
```

```
@Column(name="USER_NAME")
```

```
private String userName;
```

```
@OneToMany
```

```
    @JoinTable( name="USER_VEHICLE",
```

```
                joinColumns=@JoinColumn(name="USER_ID"),
```

```
                inverseJoinColumns=@JoinColumn(name="VEHICLE_ID"))
```

```
//its optional using for name configuration of the join table
```

```
private Collection<Vehicle> vehicle = new ArrayList<Vehicle>();
```

```
public int getUserId() {
```

```
    return userId;
```

```
}
```

```
public Collection<Vehicle> getVehicle() {
```

```
    return vehicle;
```

```
}
```

```
public void setVehicle(Collection<Vehicle> vehicle) {
```

```
    this.vehicle = vehicle;
```

```
}
```

```
public void setUserId(int userId) {
```

```
    this.userId = userId;
```

```
}
```

```

public String getUserName() {
    return userName;
}
public void setUserName(String userName) {
    this.userName = userName;
}
}

```

3. Create the hibernate configuration file.

hibernate.cfg.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

```

```

<hibernate-configuration>
<session-factory>
<!-- Database connection settings -->
<property name="connection.driver_class">com.mysql.jdbc.Driver</property>
<property name="connection.url">jdbc:mysql://localhost:3306/hibernateDB</
property>
<property name="connection.username">root</property>
<property name="connection.password">root</property>

```

```

<!-- JDBC connection pool (use the built-in) -->
<property name="connection.pool_size">1</property>

```

```

<!-- SQL dialect -->
<property name="dialect">org.hibernate.dialect.MySQLDialect</property>

```

```

<!-- Enable Hibernate's automatic session context management -->
<property name="current_session_context_class">thread</property>

```

```

<!-- Disable the second-level cache -->
<property name="cache.provider_class">
org.hibernate.cache.NoCacheProvider</property>

```

```

<!-- Echo all executed SQL to stdout -->
<property name="show_sql">>true</property>

```

```
<!-- Drop and re-create the database schema on startup -->
<property name="hbm2ddl.auto">create</property>

<mapping class="com.hibernate.dto.UserDetails"/>
<mapping class="com.hibernate.dto.Vehicle"/>

</session-factory>
</hibernate-configuration>
```

4. Create Test Demo class for run this code.
HibernateTestDemo.java

```
package com.hibernate;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;

import com.hibernate.dto.UserDetails;
import com.hibernate.dto.Vehicle;

public class HibernateTestDemo {
    /**
     * @param args
     */
    public static void main(String[] args)
    {
        UserDetails user = new UserDetails(); //create the user entity object

        Vehicle vehicle = new Vehicle(); //create the first vehicle entity object
        Vehicle vehicle2 = new Vehicle(); //create the second vehicle entity

        vehicle.setVehicleName("BMW Car"); //set the value to the vehicle entity
        vehicle2.setVehicleName("AUDI Car");

        user.setUserName("Dinesh Rajput"); //Set the value to the user entity
        user.getVehicle().add(vehicle); //add vehicle to the list of the vehicle
        user.getVehicle().add(vehicle2);

        SessionFactory sessionFactory = new
        AnnotationConfiguration().configure().buildSessionFactory(); //create session
```

factory object

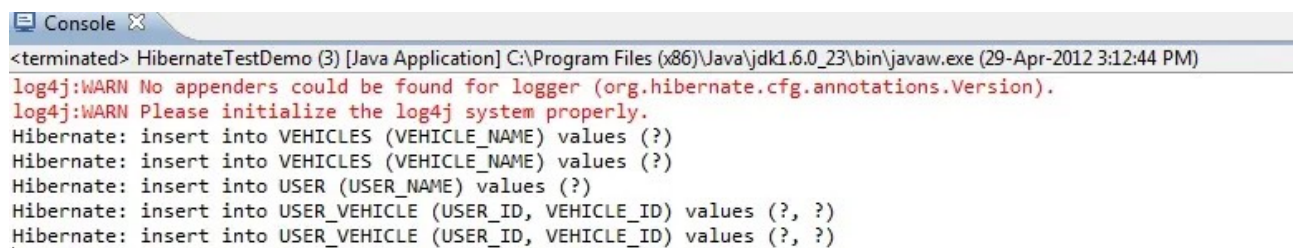
```
Session session = sessionFactory.openSession(); //create the session object  
session.beginTransaction(); //start the transaction of the session object
```

```
session.save(vehicle); //saving the vehicle to the database  
session.save(vehicle2);  
session.save(user); //save the user to the database
```

```
session.getTransaction().commit(); //close the transaction  
session.close(); //close the session  
}  
}
```

OUTPUT:

```
log4j:WARN No appenders could be found for logger  
(org.hibernate.cfg.annotations.Version).  
log4j:WARN Please initialize the log4j system properly.  
Hibernate: insert into VEHICLES (VEHICLE_NAME) values (?)  
Hibernate: insert into VEHICLES (VEHICLE_NAME) values (?)  
Hibernate: insert into USER (USER_NAME) values (?)  
Hibernate: insert into USER_VEHICLE (USER_ID, VEHICLE_ID) values  
(?, ?)  
Hibernate: insert into USER_VEHICLE (USER_ID, VEHICLE_ID) values  
(?, ?)
```



```
Console X  
<terminated> HibernateTestDemo (3) [Java Application] C:\Program Files (x86)\Java\jdk1.6.0_23\bin\javaw.exe (29-Apr-2012 3:12:44 PM)  
log4j:WARN No appenders could be found for logger (org.hibernate.cfg.annotations.Version).  
log4j:WARN Please initialize the log4j system properly.  
Hibernate: insert into VEHICLES (VEHICLE_NAME) values (?)  
Hibernate: insert into VEHICLES (VEHICLE_NAME) values (?)  
Hibernate: insert into USER (USER_NAME) values (?)  
Hibernate: insert into USER_VEHICLE (USER_ID, VEHICLE_ID) values (?, ?)  
Hibernate: insert into USER_VEHICLE (USER_ID, VEHICLE_ID) values (?, ?)
```

Now we look the table structure about this example.

	USER_ID	USER_NAME
<input type="checkbox"/>	1	Dinesh Rajput
*	(NULL)	(NULL)

PK

USER TABLE

User has USER_ID is 1 which has two model of cars BMW and AUDI in the VEHICLE table

Both tables are mapped by this table has USER_ID and VEHICLE_ID columns as Foreign keys

	USER_ID	VEHICLE_ID
<input type="checkbox"/>	1	1
<input type="checkbox"/>	1	2
*	(NULL)	(NULL)

USER_VEHICLE

PK of the USER TABLE

PK of the VEHICLE TABLE

PK

VEHICLE TABLE

	VEHICLE_ID	VEHICLE_NAME
<input type="checkbox"/>	1	BMW Car
<input type="checkbox"/>	2	AUDI Car
*	(NULL)	(NULL)

Now how can implement this mapping through mapping file(.hbm.xml) instead of the annotations.

For user class..

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.hibernate.dto.UserDetails" table="USER">
```

```
<id name="userId" type="long" column="ID" >
```

```
<generator class="assigned"/>
```

```
</id>
```

```
<property name="userName">
```

```
<column name="UserName" />
```

```
</property>
```

```
<list name="vehicle" table="STUDENT_VEHICLE" cascade="all">
```

```
<key column="USER_ID" />
```

```
<many-to-many column="VEHICLE_ID" unique="true"
```

```
class="
```

```
com.hibernate.dto.Vehicle" />
```

```
</list>
```



```
</class>
```

```
</hibernate-mapping>
```

Mapping File For Vehicle Class...
vehicle.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.hibernate.dto.Vehicle" table="VEHICLE">
```

```
<id name="userId" type="long" column="ID" >
```

```
<generator class="assigned"/>
```

```
</id>
```

```
<property name="vehicleName" column="VEHICLE_NAME"> </property>
```

```
</class>
```

```
</hibernate-mapping>
```

So it is brief description about the One To Many Mapping with using annotation and also using .hbm.xml files for entity class.

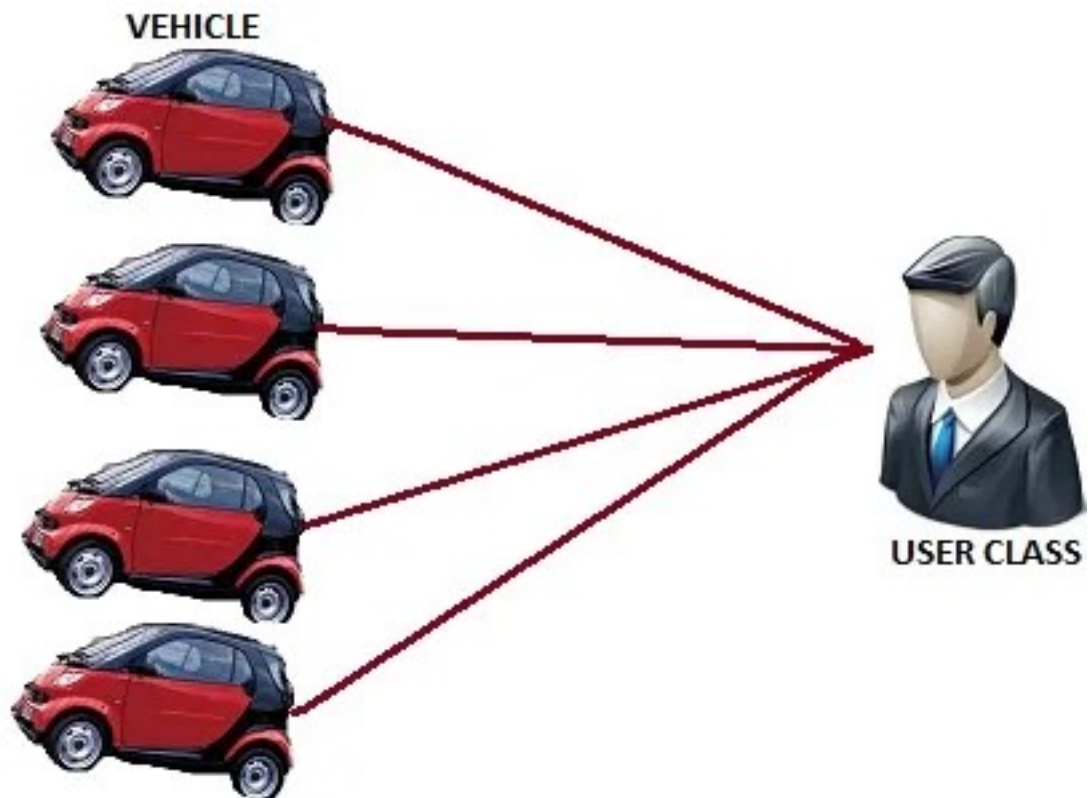
One to Many Mapping in Hibernate Example

In [previous tutorial](#) we look that what is One To One Mapping and also discussed some examples about that.

In this tutorial of one to many mapping in hibernate example we will discuss about the One To Many Mapping. A one-to-many relationship occurs when one entity is related to many occurrences in another entity.

In this chapter you will learn how to map one-to-many relationship using Hibernate. Consider the following relationship between UserDetails Class and Vehicle entity.

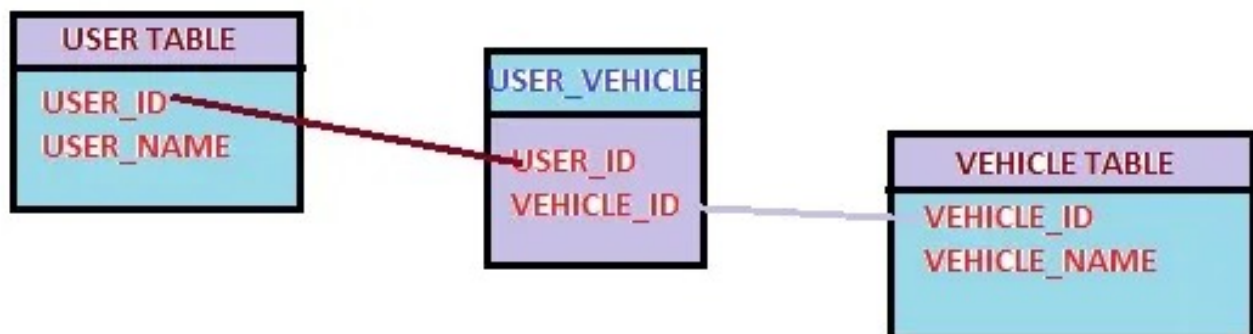
One user have the multiple vehicles.



Class diagram for that given below.



In this example UserDetails class has the collection of the another entity class Vehicle. So the given below diagram so table structure for that.



For that we will use the following annotation.

@OneToMany:

Target:

Fields (including property get methods) Defines a many-valued association with one-to-many multiplicity. If the collection is defined using generics to specify the element type, the associated target entity type need not be specified; otherwise the target entity class must be specified. If the relationship is bidirectional, the mappedBy element must be used to specify the relationship field or property of the entity that is the owner of the relationship. The **OneToMany** annotation may be used within an **embeddable** class contained within an entity class to specify a relationship to a collection of entities. If the relationship is bidirectional, the mappedBy element must be used to specify the relationship field or property of the entity that is the owner of the relationship.

Now we look the following Example related to the One to Many mapping.

1. First Create Vehicle Class

Vehicle.java

```
package com.hibernate.dto;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="VEHICLE")
public class Vehicle
{
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="VEHICLE_ID")
    private int vehicleId;

    @Column(name="VEHICLE_NAME")
    private String vehicleName;
```

```

public int getVehicleId() {
    return vehicleId;
}
public void setVehicleId(int vehicleId) {
    this.vehicleId = vehicleId;
}
public String getVehicleName() {
    return vehicleName;
}
public void setVehicleName(String vehicleName) {
    this.vehicleName = vehicleName;
}
}

```

2. Create the User Class

UserDetails.java

```

package com.hibernate.dto;

import java.util.ArrayList;
import java.util.Collection;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table (name="USER")
public class UserDetails
{
    @Id
    @Column(name="USER_ID")
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int    userId;

```

```
@Column(name="USER_NAME")
```

```
private String userName;
```

```
@OneToMany
```

```
    @JoinTable( name="USER_VEHICLE",
```

```
        joinColumns=@JoinColumn(name="USER_ID"),
```

```
        inverseJoinColumns=@JoinColumn(name="VEHICLE_ID")) //its
```

```
optional using for name configuration of the join table
```

```
private Collection<Vehicle> vehicle = new ArrayList<Vehicle>();
```

```
public int getUserId() {
```

```
    return userId;
```

```
}
```

```
public Collection<Vehicle> getVehicle() {
```

```
    return vehicle;
```

```
}
```

```
public void setVehicle(Collection<Vehicle> vehicle) {
```

```
    this.vehicle = vehicle;
```

```
}
```

```
public void setUserId(int userId) {
```

```
    this.userId = userId;
```

```
}
```

```
public String getUserName() {
```

```
    return userName;
```

```
}
```

```
public void setUserName(String userName) {
```

```
    this.userName = userName;
```

```
}
```

```
}
```

3. Create the hibernate configuration file.

hibernate.cfg.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE hibernate-configuration PUBLIC
```

```
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
```

```
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
```

```
<hibernate-configuration>
```

```
<session-factory>
```

```

<!-- Database connection settings -->
<property name="connection.driver_class">com.mysql.jdbc.Driver</property>
<property name="connection.url">jdbc:mysql://localhost:3306/hibernateDB</
property>
<property name="connection.username">root</property>
<property name="connection.password">root</property>

<!-- JDBC connection pool (use the built-in) -->
<property name="connection.pool_size">1</property>

<!-- SQL dialect -->
<property name="dialect">org.hibernate.dialect.MySQLDialect</property>

<!-- Enable Hibernate's automatic session context management -->
<property name="current_session_context_class">thread</property>

<!-- Disable the second-level cache -->
<property name="cache.provider_class"
>org.hibernate.cache.NoCacheProvider</property>

<!-- Echo all executed SQL to stdout -->
<property name="show_sql">true</property>

<!-- Drop and re-create the database schema on startup -->
<property name="hbm2ddl.auto">create</property>

<mapping class="com.hibernate.dto.UserDetails"/>
<mapping class="com.hibernate.dto.Vehicle"/>

</session-factory>
</hibernate-configuration>

```

4. Create Test Demo class for run this code.
 HibernateTestDemo.java

```

package com.hibernate;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;

```

```

import com.hibernate.dto.UserDetails;
import com.hibernate.dto.Vehicle;

public class HibernateTestDemo {
    /**
     * @param args
     */
    public static void main(String[] args)
    {
        UserDetails user = new UserDetails(); //create the user entity object

        Vehicle vehicle = new Vehicle(); //create the first vehicle entity object
        Vehicle vehicle2 = new Vehicle(); //create the second vehicle entity

        vehicle.setVehicleName("BMW Car"); //set the value to the vehicle entity
        vehicle2.setVehicleName("AUDI Car");

        user.setUserName("Dinesh Rajput"); //Set the value to the user entity
        user.getVehicle().add(vehicle); //add vehicle to the list of the vehicle
        user.getVehicle().add(vehicle2);

        SessionFactory sessionFactory = new
        AnnotationConfiguration().configure().buildSessionFactory(); //create session
        factory object
        Session session = sessionFactory.openSession(); //create the session object
        session.beginTransaction(); //start the transaction of the session object

        session.save(vehicle); //saving the vehicle to the database
        session.save(vehicle2);
        session.save(user); //save the user to the database

        session.getTransaction().commit(); //close the transaction
        session.close(); //close the session
    }
}

```

OUTPUT:

```

log4j:WARN No appenders could be found for logger
(org.hibernate.cfg.annotations.Version).
log4j:WARN Please initialize the log4j system properly.
Hibernate: insert into VEHICLES (VEHICLE_NAME) values (?)

```


Hibernate: insert into VEHICLES (VEHICLE_NAME) values (?)

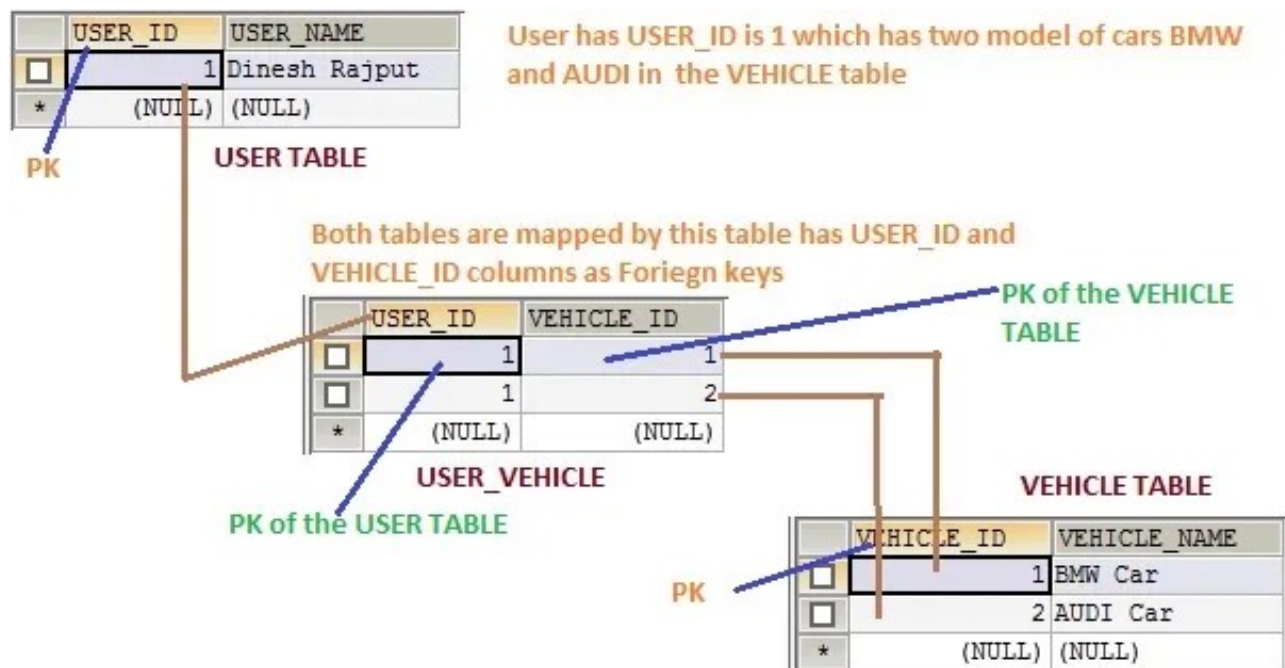
Hibernate: insert into USER (USER_NAME) values (?)

Hibernate: insert into USER_VEHICLE (USER_ID, VEHICLE_ID) values (?, ?)

Hibernate: insert into USER_VEHICLE (USER_ID, VEHICLE_ID) values (?, ?)

```
Console
<terminated> HibernateTestDemo (3) [Java Application] C:\Program Files (x86)\Java\jdk1.6.0_23\bin\javaw.exe (29-Apr-2012 3:12:44 PM)
log4j:WARN No appenders could be found for logger (org.hibernate.cfg.annotations.Version).
log4j:WARN Please initialize the log4j system properly.
Hibernate: insert into VEHICLES (VEHICLE_NAME) values (?)
Hibernate: insert into VEHICLES (VEHICLE_NAME) values (?)
Hibernate: insert into USER (USER_NAME) values (?)
Hibernate: insert into USER_VEHICLE (USER_ID, VEHICLE_ID) values (?, ?)
Hibernate: insert into USER_VEHICLE (USER_ID, VEHICLE_ID) values (?, ?)
```

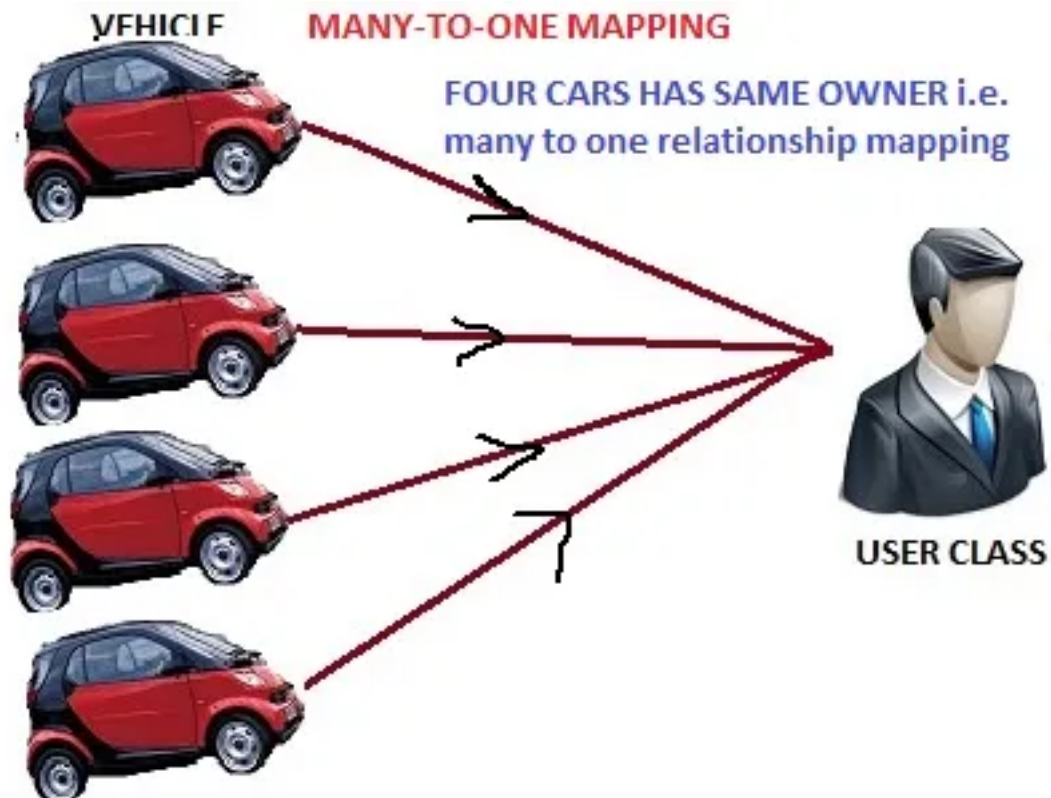
Now we look the table structure about this example.



Now how can implement this mapping through mapping file(.hbm.xml) instead of the annotations.

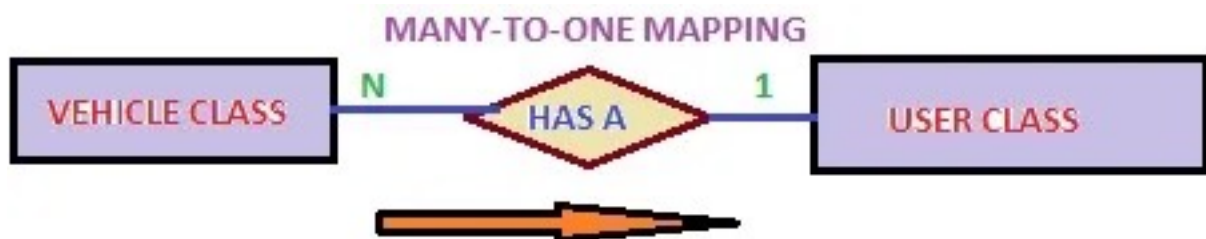
Many-to-One Relationships

A many-to-one relationship is where one entity contains values that refer to another entity (a column or set of columns) that has unique values. In relational databases, these many-to-one relationships are often enforced by foreign key/primary key relationships, and the relationships typically are between fact and dimension tables and between levels in a hierarchy.



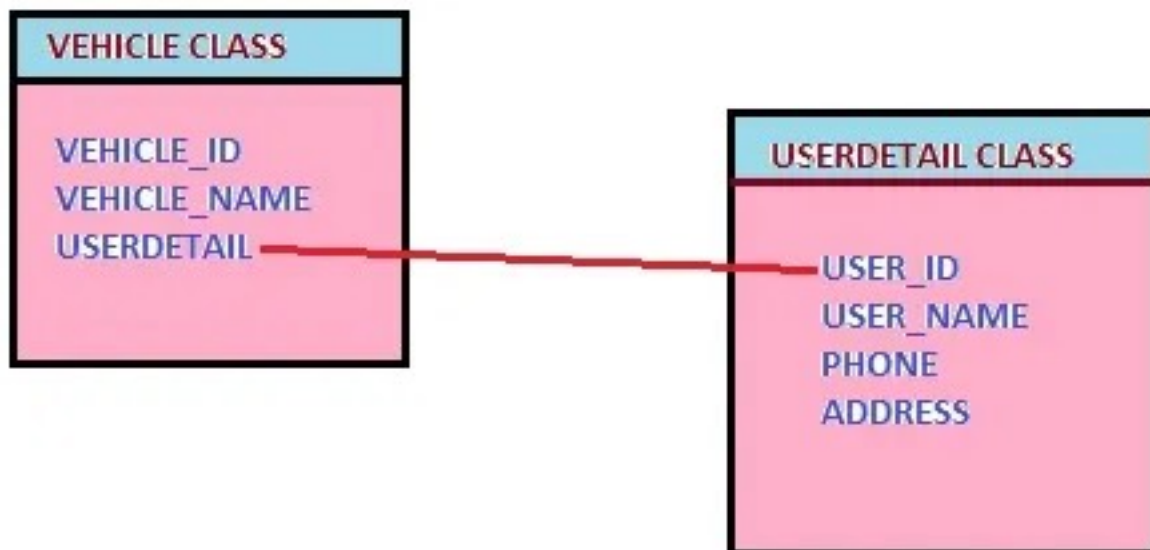
In this example, multiple vehicles (BMW Car, AUDI Car, Maruti Car and Mahindra etc.) are linked to the same User (whose primary key is 1).

Class diagram for that is given below.



According to the relationship, many vehicles can have the same owner.

To create this relationship you need to have a USER and VEHICLE table. The relational model is shown below.



For that, we will use the following annotation.

@ManyToOne :

Target:

Fields (including property get methods) Defines a single-valued association to another entity class that has many-to-one multiplicity. It is not normally necessary to specify the target entity explicitly since it can usually be inferred from the type of the object being referenced. If the relationship is bidirectional, the non-owning OneToMany entity side must use the mappedBy element to specify the relationship field or property of the entity that is the owner of the relationship. The ManyToOne annotation may be used within an embeddable class to specify a relationship from the embeddable class to an entity class. If the relationship is bidirectional, the non-owning OneToMany entity side must use the mappedBy element of the OneToMany annotation to specify the relationship field or property of the embeddable field or property on the owning side of the relationship. The dot (".") notation syntax must be used in the mappedBy element to indicate the relationship attribute within the embedded attribute. The value of each identifier used with the dot notation is the name of the respective embedded field or property.

Now we look the following Example related to the One to Many mapping.

UserDetails.java

```
package com.hibernate.dto;
```

```
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
```

```
@Entity
@Table (name="USER")
public class UserDetails
{
    @Id
    @Column(name="USER_ID")
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int    userId;
```

```
    @Column(name="USER_NAME")
    private String userName;
    public int getUserId() {
        return userId;
    }
    public void setUserId(int userId) {
        this.userId = userId;
    }
    public String getUserName() {
        return userName;
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
}
```

Vehicle.java

```
package com.hibernate.dto;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
```

```

import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity
@Table(name="VEHICLE")
public class Vehicle
{
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="VEHICLE_ID")
    private int vehicleId;

    @Column(name="VEHICLE_NAME")
    private String vehicleName;

    @ManyToOne
    @JoinColumn(name="USER_ID")
    private UserDetails user;

    public UserDetails getUser() {
        return user;
    }
    public void setUser(UserDetails user) {
        this.user = user;
    }
    public int getVehicleId() {
        return vehicleId;
    }
    public void setVehicleId(int vehicleId) {
        this.vehicleId = vehicleId;
    }
    public String getVehicleName() {
        return vehicleName;
    }
    public void setVehicleName(String vehicleName) {
        this.vehicleName = vehicleName;
    }
}

```

hibernate.cfg.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE hibernate-configuration PUBLIC
```

```
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
```

```
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
```

```
<hibernate-configuration>
```

```
<session-factory>
```

```
<!-- Database connection settings -->
```

```
<property name="connection.driver_class">com.mysql.jdbc.Driver</property>
```

```
<property name="connection.url">jdbc:mysql://localhost:3306/hibernateDB</property>
```

```
<property name="connection.username">root</property>
```

```
<property name="connection.password">root</property>
```

```
<!-- JDBC connection pool (use the built-in) -->
```

```
<property name="connection.pool_size">1</property>
```

```
<!-- SQL dialect -->
```

```
<property name="dialect">org.hibernate.dialect.MySQLDialect</property>
```

```
<!-- Enable Hibernate's automatic session context management -->
```

```
<property name="current_session_context_class">thread</property>
```

```
<!-- Disable the second-level cache -->
```

```
<property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>
```

```
<!-- Echo all executed SQL to stdout -->
```

```
<property name="show_sql">true</property>
```

```
<!-- Drop and re-create the database schema on startup -->
```

```
<property name="hbm2ddl.auto">create</property>
```

```
<mapping class="com.hibernate.dto.UserDetails"/>
```

```
<mapping class="com.hibernate.dto.Vehicle"/>
```

```
</session-factory>
```

```
</hibernate-configuration>
```

HibernateTestDemo.java

```
package com.hibernate;
```

```
import org.hibernate.Session;  
import org.hibernate.SessionFactory;  
import org.hibernate.cfg.AnnotationConfiguration;
```

```
import com.hibernate.dto.UserDetails;  
import com.hibernate.dto.Vehicle;
```

```
public class HibernateTestDemo {  
    /**  
     * @param args  
     */  
    public static void main(String[] args)  
    {  
        UserDetails user = new UserDetails(); //create an user entity  
  
        Vehicle vehicle = new Vehicle(); //create a vehicle entity  
        Vehicle vehicle2 = new Vehicle(); //create second vehicle entity  
  
        vehicle.setVehicleName("BMW Car"); //set BMW car  
        vehicle.setUser(user); //set user for that car  
  
        vehicle2.setVehicleName("AUDI Car"); //set second car Audi  
        vehicle2.setUser(user); //set user for that car  
  
        user.setUserName("Dinesh Rajput"); //set user property  
  
        SessionFactory sessionFactory = new  
        AnnotationConfiguration().configure().buildSessionFactory(); //create the  
        session factory object  
        Session session = sessionFactory.openSession(); //create the session object  
        session.beginTransaction(); //create the transaction object  
        session.save(vehicle);  
        session.save(vehicle2);  
        session.save(user);  
        session.getTransaction().commit();  
        session.close();  
    }  
}
```

}

OUTPUT:

log4j:WARN No appenders could be found for logger (org.hibernate.cfg.annotations.Version).

log4j:WARN Please initialize the log4j system properly.

Hibernate: insert into VEHICLE (USER_ID, VEHICLE_NAME) values (?, ?)

Hibernate: insert into VEHICLE (USER_ID, VEHICLE_NAME) values (?, ?)

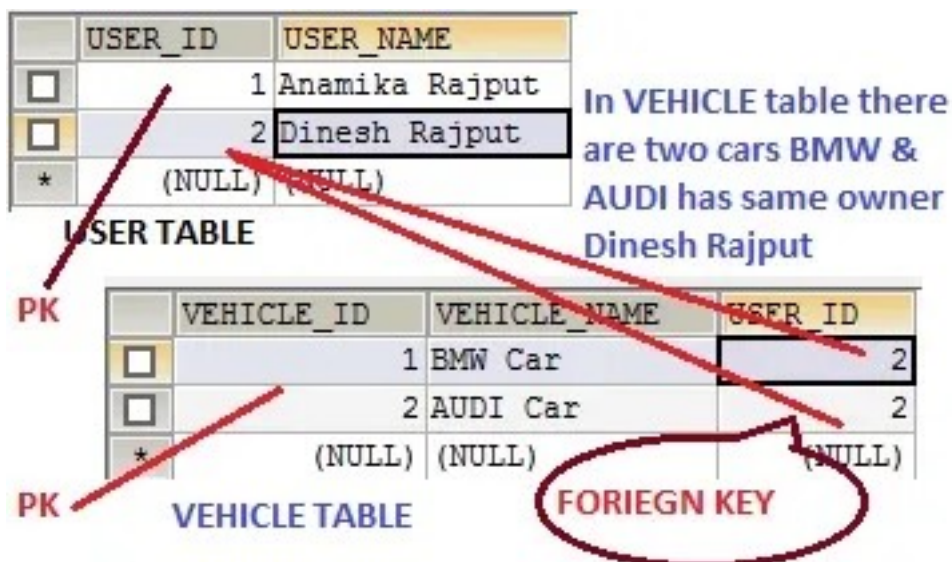
Hibernate: insert into USER (USER_NAME) values (?)

Hibernate: update VEHICLE set USER_ID=?, VEHICLE_NAME=? where VEHICLE_ID=?

Hibernate: update VEHICLE set USER_ID=?, VEHICLE_NAME=? where VEHICLE_ID=?

```
<terminated> HibernateTestDemo (4) [Java Application] C:\Program Files (x86)\Java\jdk1.6.0_23\bin\javaw.exe (30-Apr-2012 10:45:01 PM)
log4j:WARN No appenders could be found for logger (org.hibernate.cfg.annotations.Version).
log4j:WARN Please initialize the log4j system properly.
Hibernate: insert into VEHICLE (USER_ID, VEHICLE_NAME) values (?, ?)
Hibernate: insert into VEHICLE (USER_ID, VEHICLE_NAME) values (?, ?)
Hibernate: insert into USER (USER_NAME) values (?)
Hibernate: update VEHICLE set USER_ID=?, VEHICLE_NAME=? where VEHICLE_ID=?
Hibernate: update VEHICLE set USER_ID=?, VEHICLE_NAME=? where VEHICLE_ID=?
```

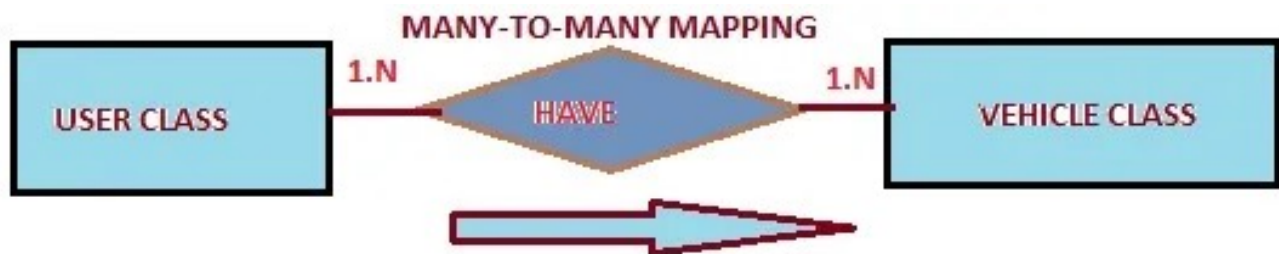
Now we look at the table structure about this example.



Many to Many Mapping in Hibernate

Many to many mapping in hibernate is required when each record in an entity may have many linked records in another entity and vice-versa.

In this tutorial you will learn how to map many-to-many relationship using Hibernate. Consider the following relationship between Vehicle and UserDetails entity.



According to the relationship a user can have in any number of vehicles and the vehicle can have any number of users.

UserDetail.java

```
package com.hibernate.dto;

import java.util.ArrayList;
import java.util.Collection;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.Table;

@Entity
@Table (name="USER")
public class UserDetails
```



```

{
    @Id
    @Column(name="USER_ID")
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int     userId;

    @Column(name="USER_NAME")
    private String  userName;

    @ManyToMany
        private Collection<Vehicle>  vehicle = new
ArrayList<Vehicle>();

    public int getUserId() {
        return userId;
    }
    public Collection<Vehicle> getVehicle() {
        return vehicle;
    }
    public void setVehicle(Collection<Vehicle> vehicle) {
        this.vehicle = vehicle;
    }
    public void setUserId(int userId) {
        this.userId = userId;
    }
    public String getUsername() {
        return userName;
    }
    public void setUsername(String userName) {
        this.userName = userName;
    }
}

```

Vehicle.java

```

package com.hibernate.dto;

import java.util.ArrayList;
import java.util.Collection;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

```

```

import javax.persistence.Table;

@Entity
@Table(name="VEHICLE")
public class Vehicle
{
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="VEHICLE_ID")
    private int vehicleId;

    @Column(name="VEHICLE_NAME")
    private String vehicleName;

    @ManyToMany(mappedBy="vehicle")
    private Collection<UserDetails> user = new
ArrayList<UserDetails>();

    public Collection<UserDetails> getUser() {
        return user;
    }
    public void setUser(Collection<UserDetails> user) {
        this.user = user;
    }
    public int getVehicleId() {
        return vehicleId;
    }
    public void setVehicleId(int vehicleId) {
        this.vehicleId = vehicleId;
    }
    public String getVehicleName() {
        return vehicleName;
    }
    public void setVehicleName(String vehicleName) {
        this.vehicleName = vehicleName;
    }
}

```

hibernate.cfg.xml:

```

<hibernate-configuration>
    <session-factory>
        <!-- Database connection settings -->

```

```

<property
name="connection.driver_class">com.mysql.cj.jdbc.Driver</
property>
    <property name="connection.url">jdbc:mysql://
localhost:3306/hibernateDB</property>
    <property name="connection.username">root</
property>
    <property name="connection.password">root</
property>

    <!-- JDBC connection pool (use the built-in) -->
    <property name="connection.pool_size">1</
property>

    <!-- SQL dialect -->
    <property
name="dialect">org.hibernate.dialect.MySQLDialect</
property>

    <!-- Enable Hibernate's automatic session context
management -->
    <property
name="current_session_context_class">thread</property>

    <!-- Disable the second-level cache -->
    <property
name="cache.provider_class">org.hibernate.cache.NoCachePr
ovider</property>

    <!-- Echo all executed SQL to stdout -->
    <property name="show_sql">true</property>

    <!-- Drop and re-create the database schema on
startup -->
    <property name="hbm2ddl.auto">create</property>

    <mapping class="com.hibernate.UserDetails"/>
    <mapping class="com.hibernate.Vehicle"/>

</session-factory>
</hibernate-configuration>
HibernateTestDemo.java

```

```

package com.hibernate;

```

```

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;

import com.hibernate.UserDetails;
import com.hibernate.Vehicle;

public class HibernateTestDemo {

    /**
     * @param args
     */
    public static void main(String[] args)
    {
        UserDetails user = new UserDetails();
        UserDetails user2 = new UserDetails();

        Vehicle vehicle = new Vehicle();
        Vehicle vehicle2 = new Vehicle();

        vehicle.setVehicleName("Car");
        vehicle.getUser().add(user);
        vehicle.getUser().add(user2);

        vehicle2.setVehicleName("Jeep");
        vehicle2.getUser().add(user2);
        vehicle2.getUser().add(user);

        user.setUserName("First User");
        user2.setUserName("Second User");
        user.getVehicle().add(vehicle);
        user.getVehicle().add(vehicle2);
        user2.getVehicle().add(vehicle);
        user2.getVehicle().add(vehicle2);

        SessionFactory sessionFactory = new
AnnotationConfiguration().configure().buildSessionFactory
();

        Session session = sessionFactory.openSession();
        session.beginTransaction();
        session.save(vehicle);
        session.save(vehicle2);
        session.save(user);
    }
}

```

```
        session.save(user2);  
        session.getTransaction().commit();  
        session.close();  
    }  
}
```

Hibernate Query Language and the Query Object

Hibernate created a new language named Hibernate Query Language (HQL), the syntax is quite similar to database SQL language. The main difference between is HQL uses class name instead of table name, and property names instead of column name.

Hibernate uses the following ways to retrieve objects from the database:

- Hibernate Query Language (HQL)
- Query By Criteria (QBC) and Query BY Example (QBE) using Criteria API
- Native SQL queries

The most preferred way is using the Hibernate Query Language (HQL), which is an easy-to-learn and powerful query language, designed as a minimal object-oriented extension to SQL. HQL has syntax and keywords/clauses very similar to SQL. It also supports many other SQL-like features, such as aggregate functions (for example: `sum()`, `max()`) and clauses such as `group by` and `order by` clause.

Why WE USE HQL?

Although it is possible to use native SQL queries directly with a Hibernate-based persistence layer, it is more efficient to use HQL instead. The reasons of choosing HQL over the other two methods are given below.

- HQL allows representing SQL queries in object-oriented terms—by using objects and properties of objects.
- Instead of returning plain data, HQL queries return the query result(s) in the form of object(s)/tuples of object(s) that are ready to be accessed, operated upon, and manipulated programmatically. This approach does away with the routine task of creating and populating objects from scratch with the “resultset” retrieved from database queried.
- HQL fully supports polymorphic queries. That is, along with the object to be returned as a query result, all child objects (objects of subclasses) of the given object shall be returned.