Any enterprise application performs database operations by storing and retrieving vast amounts of data. Despite all the available technologies for storage management, application developers normally struggle to perform database operations efficiently.

Generally, Java developers use lots of code or proprietary framework to interact with the database. Therefore, it is advisable to use Object Relational Mapping (ORM) to reduct the burden of interacting with the database.

ORM forms a bridge between object models (Java program) and relational models (database program) like JDBC.

At the same time ORM solutions like Hibernate aim to abstract from the specific product used to store the data. This allows using the same Java code with different database products without the need to write code that handles the subtle differences between the supported products.

**What is ORM?**

ORM stands for **O**bject-**R**elational **M**apping (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:

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.'

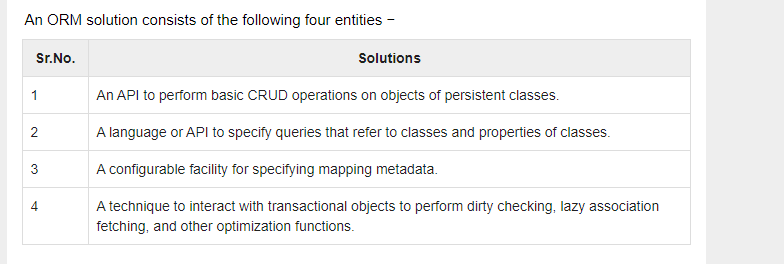
4. No need to deal with the database implementation.

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**:



**Dirty Checking** is one of the features of **hibernate**. In **dirty checking**, **hibernate** automatically detects whether an object is modified (or) not and need to be updated. As long as the object is in persistent state i.e., bound to a particular Session(org. **hibernate**

Say you have a parent and that parent has a collection of children. Hibernate now can "lazy-load" the children, which means that it does not actually load all the children when loading the parent. Instead, it loads them when requested to do so. You can either request this explicitly or, and this is far more common, hibernate will load them automatically when you try to access a child.

Hibernate is also a JPA provider, that means Hibernate implements the Java Persistence API (JPA). **JPA is a vendor independent specification for mapping Java objects to the tables of relational databases.**

Hibernate is one of the most popular Object/Relational Mapping (ORM) framework in the Java world. It allows developers to map the object structures of normal Java classes to the relational structure of a database. With the help of an ORM framework the work to store data from object instances in memory to a persistent data store and load them back into the same object structure becomes significantly easier

**Hibernate consists of three different components**:

• Entities: The classes that are mapped by Hibernate to the tables of a relational database system are simple Java classes (Plain Old Java Objects).

• Object-relational metadata: The information how to map the entities to the relational database is either provided by annotations (since Java 1.5) or by legacy XML-based configuration files. The information in these files is used at runtime to perform the mapping to the data store and back to the Java objects.

• Hibernate Query Language (HQL): When using Hibernate, queries send to the database do not have to be formulated in native SQL but can be specified using Hibernate’s query language. As these queries are translated at runtime into the currently used dialect of the chose product, queries formulated in HQL are independent from the SQL dialect of a specific vendor.

In this tutorial we are going through different aspects of the framework and will develop a simple Java SE application that stores and retrieves data in/from a relational database. We will use the following libraries/environments:

• maven >= 3.0 as build environment

• Hibernate(4.3.8.Final)

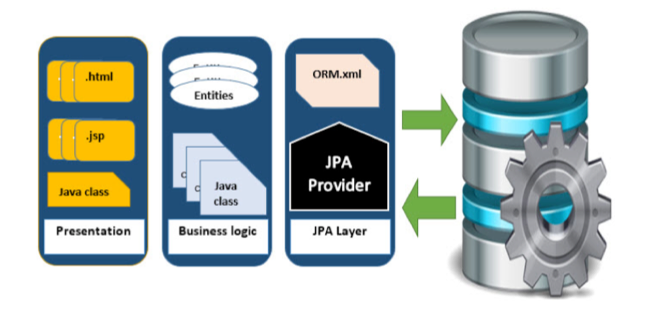
• H2 as relational database (1.3.176)

## What is JPA?

Java Persistence API is a collection of classes and methods to persistently store the vast amounts of data into a database which is provided by the Oracle Corporation.

## Where to use JPA?

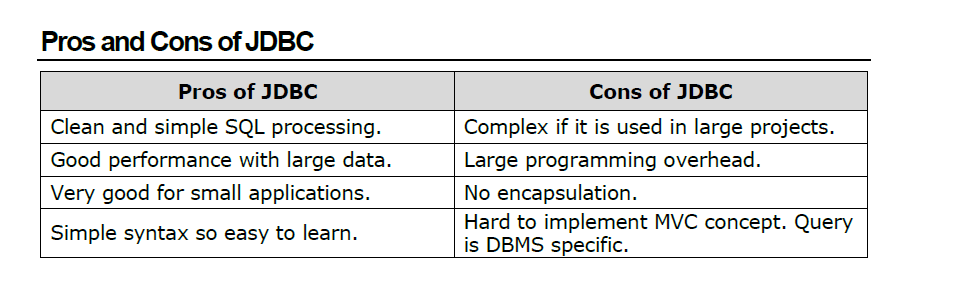
To reduce the burden of writing codes for relational object management, a programmer follows the ‘JPA Provider’ framework, which allows easy interaction with database instance. Here the required framework is taken over by JPA.



**What is JDBC?**

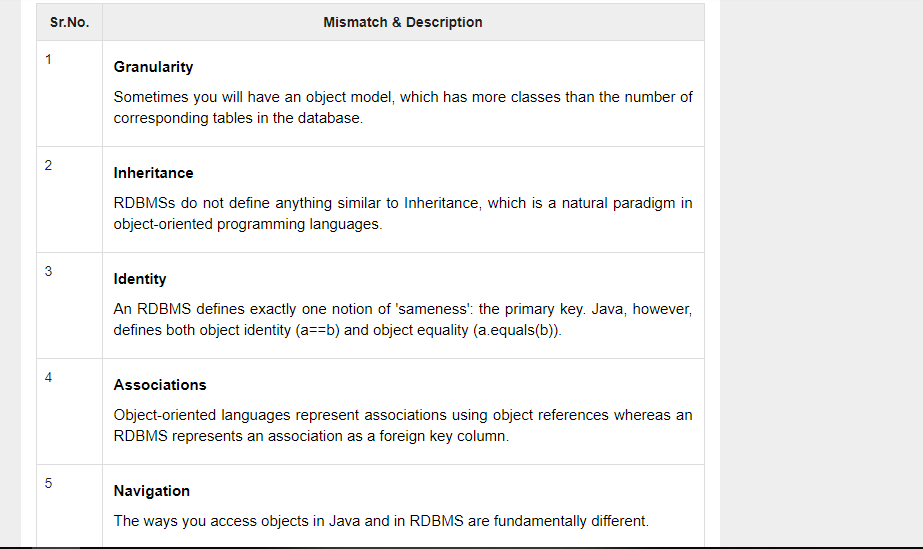
JDBC stands for **Java Database Connectivity**. It provides a set of Java API for accessing the relational databases from Java program. These Java APIs enables Java programs to execute SQL statements and interact with any SQL compliant database.

JDBC provides a flexible architecture to write a database independent application that can run on different platforms and interact with different DBMS without any modification.



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

When we work with an object-oriented system, there is a mismatch between the object model and the relational database table. RDBMSs represent data in a tabular format whereas object-oriented languages, such as Java or C# represent it as an interconnected graph of objects.



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.

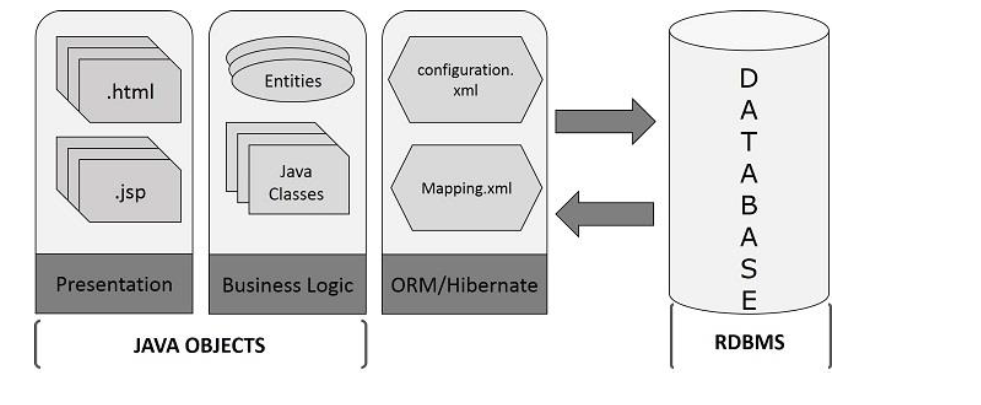
* Enterprise JavaBeans Entity Beans
* Java Data Objects
* Castor
* TopLink
* Spring DAO
* Hibernate
* And many more

2. HIBERNATE – OVERVIEW

Hibernate is an **O**bject-**R**elational **M**apping(ORM) solution for JAVA. It is an open source persistent framework created by Gavin King in 2001. It is a powerful, high performance Object-Relational Persistence and Query service for any Java Application.

Hibernate maps Java classes to database tables and from Java data types to SQL data types and relieves the developer from 95% of common data persistence related programming tasks.

Hibernate sits between traditional Java objects and database server to handle all the works in persisting those objects based on the appropriate O/R mechanisms and patterns.



**Hibernate Advantages**

Here we have listed down the advantages of using Hibernate:

 Hibernate takes care of mapping Java classes to database tables using XML files and without writing any line of code. If there is a change in the database or in any table, then all that you need to change are the XML file properties.

 Provides simple APIs (classes and methods) for storing and retrieving Java objects directly to and from the database.

 Hibernate supports **Inheritance**, **Association relations**, and **Collections**.

 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.

Hibernate Supports only unchecked exceptions, so no need to write try, catch, or throws blocks. Generally we have a Hibernate translator which converts Checked exceptions to Unchecked.

 Minimizes database access with smart fetching strategies.

 Hibernate has its own query language. That is **Hiberate Query Language** (HQL) which contains database independent controlers.

 Manipulates Complex associations of objects of your database.

 Hibernate supports caching mechanism: It reduces the number of round trips (transactions) between an application and the database. It increases the application performance.

 Hibernate supports annotations, apart from XML..

**Supported Databases**

Hibernate supports almost all the major RDBMS database servers. Following is a list of few of the database engines that Hibernate supports:

 HSQL Database Engine

 DB2/NT

 MySQL

 PostgreSQL

 FrontBase

 Oracle

 Microsoft SQL Server Database

 Sybase SQL Server

 Informix Dynamic Server

Hibernate supports a variety of other technologies as well including:

 XDoclet Spring

 J2EE

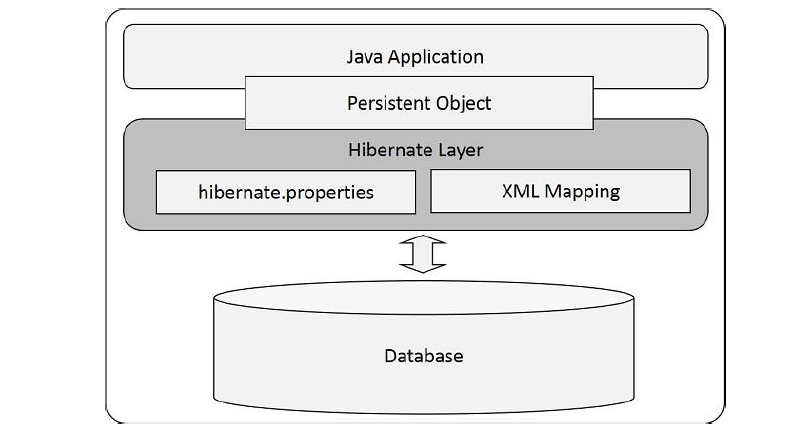
 Eclipse plug-ins

 Maven

HIBERNATE – ARCHITECTURE

Hibernate has a layered architecture which helps the user to operate without having to know the underlying APIs. Hibernate makes use of the database and configuration data to provide persistence services (and persistent objects) to the application.

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

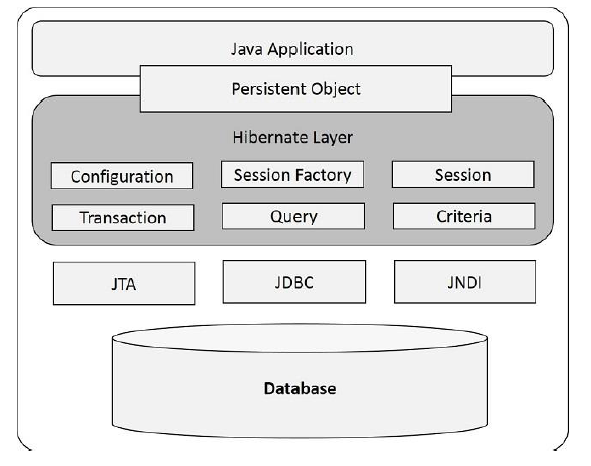


Following is a detailed view of the Hibernate Application Architecture with a few important core classes in the Hibernate layer.

**Hibernate uses various existing Java APIs, like JDBC, Java Transaction API (JTA), and Java Naming and Directory Interface (JNDI). JDBC** provides a rudimentary(basic) level of abstraction of functionality common to relational databases, allowing almost any database with a JDBC driver to be supported by Hibernate. JNDI and JTA support Hibernate to be integrated with J2EE application servers.

The following section gives a brief description about the class objects which are involved in the Hibernate Layer of the given architecture diagram. This sectiong gives you a theritical idea of how the hibernate class objects are used to build an application.

JNDI and JTA support Hibernate to be integrated with J2EE application servers.



**Configuration Object**

**Configuration** is a serializable class. It is the first Hibernate object that you need to create in any Hibernate application. It is usually created only once during application initialization. It allows the application to specify properties and mapping documents to be used. The Configuration object provides two keys components:

 **Database Connection:** A database connection is most important for Enterprise and Database applications. It is handled through one or more configuration files supported by Hibernate. Those are **hibernate.properties** file and **hibernate.cfg.xml** file.

 **Mapping Setup:** This component creates the connection between the Java classes and database tables. It creates mapping between each entity java class and each table in the database.

**SessionFactory Object**

**SessionFactory** is a Factory Interface used to create **Session** instances. After adding the properties and Mapping files to the Configuration object, it is used to create a SessionFactory object which in turn configures Hibernate (Front-end javaclasses and Back-end tables) for the application.

SessionFactory is a thread-safe object and used by all the threads of an application. It is a heavyweight object, usually created during application start-up and kept for later use. You would need one SessionFactory object per database using a separate configuration file.

So, if you are using multiple databases, then you would have to create multiple SessionFactory objects.

**Session Object**

**Session** is an Interface that wraps the JDBC connection.

That means, it creates a physical connection between the application and a database.

The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database.

Persistent objects are saved and retrieved through a Session object.

The Lifecycle of a Session is bounded by the beginning and end of a logical trasaction. It contains three states:

 **transient:** never persistent, currently not associated with any Session.

 **persistent:** currently associated with unique Session.

 **detached:** previously persistent, currently not associated with any Session.

The session objects should not be kept open for a long time because they are not usually thread-safe. They should be created and destroyed them as needed.

**Transaction Object**

**Transaction** is an Interface and it represents a unit of work with the database and most of the RDBMS supports transaction functionality. Transactions in Hibernate are handled by an underlying transaction manager.

This is an optional object and Hibernate applications may choose not to use this interface, instead managing transactions in their own application code.

**Query Object**

**Query** is an interface and it is used in **SQL** or Hibernate Query Language (**HQL**) string to retrieve data from the database and create objects. A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

**Criteria Object**

**Criteria** is an interface and it is used for retrieving entity data by composing Criterion (Interface) objects. Criterion Objects work like a condition **(WHERE and IF)** in the SQL query, all the criterion objects (conditions) are added to the Criteria Object and that object will be executed and used for retrieving entity data in objects.

## Salient features of the Hibernate framework

#### Object/Relational Mapping

Hibernate, as an ORM framework, allows the mapping of the Java domain object with database tables and vice versa. As a result, business logic is able to access and manipulate database entities via Java objects. It helps to speed up the overall development process by taking care of aspects such as transaction management, automatic primary key generation, managing database connections and related implementations, and so on.

#### JPA provider

Hibernate does support the [**Java Persistence API**](https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html)**(JPA)** specification. JPA is a set of specifications for accessing, persisting, and managing data between Java objects and relational database entities.

#### Idiomatic persistence

Any class that follows [object-oriented principles](https://howtodoinjava.com/java/oops/object-oriented-programming/) such as inheritance, polymorphism, and so on, can be used as a persistent class.

#### High performance and scalability

Hibernate supports techniques such as different fetching strategies, lazy initialization, optimistic locking, and so on, to achieve high performance, and it scales well in any environment.

#### Easy to maintain

Hibernate is easier to maintain as it requires no special database tables or fields. It generates SQL at system initialization time. It is much quicker and easier to maintain compared to JDBC.

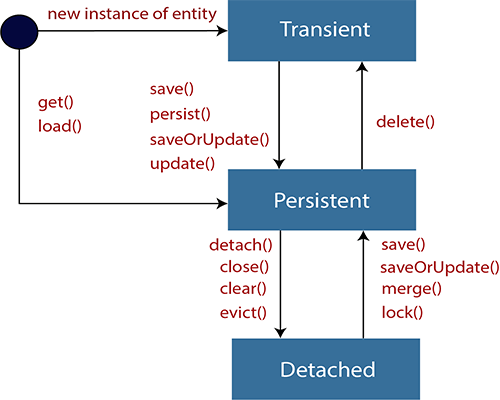
Many times in our application, we face a situation where we have to compare two objects to check their equality for satisfying some business rules. In core java, we have already much knowledge about checking equality of objects, but in hibernate we need to take care of few extra things as well. Let’s learn what are those extra concepts.

# **Hibernate Lifecycle**

In Hibernate, either we create an object of an entity and save it into the database, or we fetch the data of an entity from the database. Here, each entity is associated with the lifecycle. The entity object passes through the different stages of the lifecycle.

The Hibernate lifecycle contains the following states: -

* Transient state
* Persistent state
* Detached state



### **Transient state**

* The transient state is the initial state of an object.
* Once we create an instance of POJO class, then the object entered in the transient state.
* Here, an object is not associated with the Session. So, the transient state is not related to any database.
* Hence, modifications in the data don't affect any changes in the database.
* The transient objects exist in the heap memory. They are independent of Hibernate.

Employee e=**new** Employee(); //Here, object enters in the transient state.

e.setId(101);

e.setFirstName("Gaurav");

e.setLastName("Chawla");

### **Persistent state**

* As soon as the object associated with the Session, it entered in the persistent state.
* Hence, we can say that an object is in the persistence state when we save or persist it.
* Here, each object represents the row of the database table.
* So, modifications in the data make changes in the database.

We can use any of the following methods for the persistent state.

1. session.save(e);
2. session.persist(e);
3. session.update(e);
4. session.saveOrUpdate(e);
5. session.lock(e);
6. session.merge(e);

### **Detached State**

* Once we either close the session or clear its cache, then the object entered into the detached state.
* As an object is no more associated with the Session, modifications in the data don't affect any changes in the database.
* However, the detached object still has a representation in the database.
* If we want to persist the changes made to a detached object, it is required to reattach the application to a valid Hibernate session.
* To associate the detached object with the new hibernate session, use any of these methods - load(), merge(), refresh(), update() or save() on a new session with the reference of the detached object.

We can use any of the following methods for the detached state.

1. session.close();
2. session.clear();
3. session.detach(e);
4. session.evict(e);

# Hibernate Entities Equality and Identity

Many times in our application, we face a situation where we have to compare two objects to check their equality for satisfying some business rules. In core java, we have already much knowledge about checking equality of objects, but in hibernate we need to take care of few extra things as well. Let’s learn what are those extra concepts.

We already learned about various [**states of hibernate entities in their life-cycle**](https://howtodoinjava.com/hibernate/hibernate-entity-persistence-lifecycle-states/). There we discussed that hibernate mostly work with persistent objects only. As we know that when we have a persistent object in hibernate, that object represents both :

* An instance of a class in a particular Java virtual machine (JVM)
* A row (or rows) in a database table (or tables)

We know enough around first concept. I will focus on second point.

### Objects fetched from same session

Requesting a persistent object again **from the same Hibernate session returns the same Java instance of a class**, which means that you can compare the objects using the standard Java ‘==’ equality syntax.

Let’s see a quick example:

|  |
| --- |
| public static void main(String[] args)  {      Session sessionOne = HibernateUtil.getSessionFactory().openSession();      sessionOne.beginTransaction();        // Create new Employee object      EmployeeEntity emp = new EmployeeEntity();      emp.setFirstName("Lokesh");      emp.setLastName("Gupta");      //Save employee      sessionOne.save(emp);        sessionOne.getTransaction().commit();        //Get employee id      Integer genEmpId = emp.getEmployeeId();        //New session where we will fetch the employee two times and compare the objects      Session sessionTwo = HibernateUtil.getSessionFactory().openSession();      sessionTwo.beginTransaction();        EmployeeEntity employeeObj1 = (EmployeeEntity) sessionTwo.get(EmployeeEntity.class, genEmpId);      EmployeeEntity employeeObj2 = (EmployeeEntity) sessionTwo.get(EmployeeEntity.class, genEmpId);        //Checking equality      System.out.println(employeeObj1 == employeeObj2);        HibernateUtil.shutdown();  }    Output:    true |

You see above that we got two instances on EmployeeEntity and both are actually same java object instance.

### Objects fetched from different sessions

If you request a persistent object from **more than one Hibernate session, Hibernate will provide distinct instances from each session**, and the == operator will return false if you compare these object instances.

Let’s compare instances “emp” and “employeeObj1” in above example and you will get the result as false; because both are fetched in separate sessions.

|  |
| --- |
| System.out.println(emp == employeeObj1);  System.out.println(emp.equals(employeeObj1));    Output:    false  false |

So if you are comparing objects in two different sessions, you will need to implement the equals() method on your Java persistence objects, which you should do as a regular occurrence anyway. (Just don’t forget to override hashCode() along with it.)

Now let’s add equals() method as suggested and then see the behavior change while checking the equality of both instances on EmployeeEntity.

|  |
| --- |
| @Entity  @Table(name = "Employee")  public class EmployeeEntity implements Serializable  {     private static final long serialVersionUID = -1798070786993154676L;     @Id     @Column(name = "ID", unique = true, nullable = false)     @GeneratedValue(strategy = GenerationType.SEQUENCE)     private Integer           employeeId;     @Column(name = "FIRST\_NAME", unique = false, nullable = false, length = 100)     private String            firstName;     @Column(name = "LAST\_NAME", unique = false, nullable = false, length = 100)     private String            lastName;       @Override     public boolean equals(Object o) {         if (this == o) return true;         if (!(o instanceof EmployeeEntity)) return false;           EmployeeEntity otherEmployee = (EmployeeEntity) o;           if (getEmployeeId() != null ?             !getEmployeeId().equals(otherEmployee.getEmployeeId()) : otherEmployee.getEmployeeId() != null)             return false;         if (getFirstName() != null ?             !getFirstName().equals(otherEmployee.getFirstName()) : otherEmployee.getFirstName() != null)             return false;         if (getLastName() != null ?             !getLastName().equals(otherEmployee.getLastName()) : otherEmployee.getLastName() != null)             return false;           return true;     }       @Override     public int hashCode() {     int result = getEmployeeId() != null ? getEmployeeId().hashCode() : 0;         result = 31 \* result + (getFirstName() != null ? getFirstName().hashCode() : 0);         result = 31 \* result + (getLastName() != null?getLastName().hashCode() : 0);         return result;     }       //Setters and Getters  } |

Now lets again check the equality using equals() method. [‘==’ will return false, we know that].

|  |
| --- |
| System.out.println(emp.equals(employeeObj1));    Output:    true |

Now both objects are equal logically as well as programmatically.

### Bullet Points

1. Requesting a persistent object again from the same Hibernate session returns the “same java instance” of a class.
2. Requesting a persistent object from the different Hibernate session returns “different java instance” of a class.
3. As a best practice, always implement [equals() and hashCode()](https://howtodoinjava.com/java/basics/java-hashcode-equals-methods/) methods in your hibernate entities; and always compare them using equals() method only.

# Understanding Hibernate First Level Cache with Example

Caching is facility provided by ORM frameworks which help users to get fast running web application, while help framework itself to reduce number of queries made to database in a single transaction. Hibernate also provide this caching functionality, in two layers.

* **Fist level cache**: This is enabled by default and works in session scope. Read more about [**hibernate first level cache**](https://howtodoinjava.com/hibernate/understanding-hibernate-first-level-cache-with-example/).
* **Second level cache**: This is apart from first level cache which is available to be used globally in session factory scope.

1. **First-level cache** : It represents the default cache used by Hibernate Session object while interacting with the database. It is also called as session cache and caches objects within the current session. All requests from the Session object to the database must pass through the first-level cache or session cache. One must note that the first-level cache is available with the session object until the Session object is live.

### **Important facts**

1. First level cache is associated with “session” object and other session objects in application can not see it.
2. The scope of cache objects is of session. Once session is closed, cached objects are gone forever.
3. First level cache is enabled by default and you can not disable it.
4. When we query an entity first time, it is retrieved from database and stored in first level cache associated with hibernate session.
5. If we query same object again with same session object, it will be loaded from cache and no sql query will be executed.
6. The loaded entity can be removed from session using evict() method. The next loading of this entity will again make a database call if it has been removed using evict() method.
7. The whole session cache can be removed using clear() method. It will remove all the entities stored in cache.

### **How second level cache works**

Lets write all the facts point by point:

1. Whenever hibernate session try to load an entity, the very first place it look for cached copy of entity in first level cache (associated with particular hibernate session).
2. If cached copy of entity is present in first level cache, it is returned as result of load method.
3. If there is no cached entity in first level cache, then second level cache is looked up for cached entity.
4. If second level cache has cached entity, it is returned as result of load method. But, before returning the entity, it is stored in first level cache also so that next invocation to load method for entity will return the entity from first level cache itself, and there will not be need to go to second level cache again.
5. If entity is not found in first level cache and second level cache also, then database query is executed and entity is stored in both cache levels, before returning as response of load() method.
6. Second level cache validate itself for modified entities, if modification has been done through hibernate session APIs.
7. If some user or process make changes directly in database, the there is no way that second level cache update itself until “timeToLiveSeconds” duration has passed for that cache region. In this case, it is good idea to invalidate whole cache and let hibernate build its cache once again. You can use below code snippet to invalidate whole hibernate second level cache.

save()- As the method name suggests, hibernate save() can be used to save entity to database. We can invoke this method outside a transaction. If we use this without transaction and we have cascading between entities, then only the primary entity gets saved unless we flush the session.

persist()-Hibernate persist is similar to save (with transaction) and it adds the entity object to the persistent context, so any further changes are tracked. If the object properties are changed before the transaction is committed or session is flushed, it will also be saved into database. Also, we can use persist() method only within the boundary of a transaction, so it’s safe and takes care of any cascaded objects. Finally, persist doesn't return anything so we need to use the persisted object to get the generated identifier value.

**Clear () :**  When this method get called inside transaction boundry then all objects which are currently associate with particular session will be  disconnected / clean or no longer associate with that Session instance.

Therefore, after calling this method nothing will be performed on

**evict():** Removes the object from the session. This method is used to dissociate/disconnect the specified object from the session.

**Close() :**Close session by calling **session.close()** method, means End the session and releasing the JDBC Connection and clean up.

# HQL – Hibernate Query Language Examples

#### 1.1. HQL Update Statement

UPDATE alters the details of existing objects in the database. In-memory entities, managed or not, will not be updated to reflect changes resulting from issuing UPDATE statements.

|  |
| --- |
| HQL UPDATE statement syntax |
| UPDATE [VERSIONED]     [FROM] path [[AS] alias] [, ...]     SET property = value [, ...]     [WHERE logicalExpression] |

* path – fully qualified name of the entity or entities
* alias – used to abbreviate references to specific entities or their properties, and must be used when property names in the query would otherwise be ambiguous.
* VERSIONED – means that the update will update time stamps, if any, that are part of the entity being updated.
* property – names of properties of entities listed in the FROM path.
* logicalExpression – a where clause.

An example of the update statement. In this example, we are updating Employee data with **hql update query multiple columns**.

|  |
| --- |
| hql update statement example |
| Query query=session.createQuery("update Employee set age=:age where name=:name");  query.setInteger("age", 32);  query.setString("name", "Lokesh Gupta");  int modifications=query.executeUpdate(); |