HIBERNATE

MR. Ashok

1. Introduction
2. ORM (Object Relational Mapping)
3. Hibernate Architecture
4. POJO, Mapping and Configuration files in Hibernate
5. Installation, Simple Hibernate Application Requirements
6. CRUD operations
7. AUTO DDL
8. Enable Logging and Commenting
9. Singleton Design pattern
10. State of Objects in Hibernate
11. Generators in Hibernate
12. Composite-Primary key
13. Component Mapping
14. Connection pooling in Hibernate
15. Transaction management
16. Batch processing in Hibernate
17. Hibernate Annotations
18. CRUD Operations using Hibernate Annotations
19. HQL (Hibernate Query Language)
20. Pagination Using HQL
21. Criteria API
22. Native SQL Queries
23. Working with procedures in Hibernate
24. Named Queries
25. Filters in Hibernate
26. Hibernate Caching Mechanism
    - First level cache
    - Second level cache
    - Query Cache
27. Relationships in Hibernate.
    - One to One
    - One to Many
    - Many to One
    - Many to Many
28. Fetching Strategies in Hibernate
29. Cascade Types
30. Versioning and Timestamping
31. Hibernate Validator framework

Introduction

All applications requires some persistent mechanism to store application generated data. In java we can store data using variables and Objects. But these variables and Objects is will be stored into Secondary memory. This secondary memory will be available till the application is executing. Once application execution completed this secondary memory will be vanished, then data which is stored in secondary memory also will get vanished. So we will lose the data if store in variables and Objects. To overcome this problem we need to go for Persistence stores.

What is Persistence and what is Persistence store?

The process of storing and maintaining the data for long time is called Persistence. To store and maintain data for long time we need Persistence stores. Below are the Persistence stores available

1. File
2. RDBMS

Files are used to store small amount of data.
Databases are used to store huge amount of data and these are having lot of advantages when compared with the files.
After storing the data into Persistence stores, we can perform some operations on Persistent data, these operations are called as Persistence operations. Below are the persistence operations

CURD or CRUD or SCUD

C- Create /Insert

U – Update

R – Retrieve

D – Delete

To perform these persistence operations on persistence data we need Persistence Technologies. JDBC is one of the Persistence Technology. It is released by Sun micro system as part of JDK 1.1v.

Drawbacks of JDBC

- In JDBC, we have to write lot of boiler plate of code to perform persistence operations
- Here as a programmer every time we need to open and close the connection (Opening and closing connection for multiple times will decrease Performance of the application)
- JDBC supports only Database dependent queries
- In JDBC we need to write SQL queris in various places, after the program has created if the table structure is modified then the JDBC program doesn't work, again we need to modify and compile and re-deploy required, which is tedious work.
- Every line of JDBC code will throw Checked Exception(java.lang.SQLException), as a programmer we have to handle all these exceptions
- JDBC used to generate database related error codes if an exception occurs, but java programmers are unknown about this error codes.
- There is AUTO DDL support in JDBC and programmer is responsible to generate Primary Keys for the tables.

In order to overcome above problems, Hibernate came into picture..!!

What is Hibernate

Hibernate is an open-source light weight ORM framework.

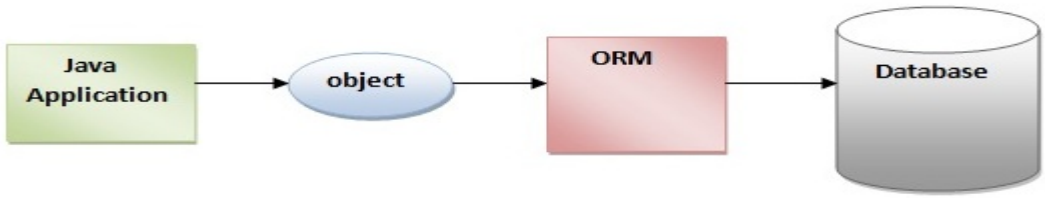
Hibernate was started in 2001 by Gavin King with colleagues from Cirrus Technologies as an alternative to using EJB2-style entity beans. The original goal was to offer better persistence capabilities than those offered by EJB2, by simplifying the complexities and supplementing certain missing features.

In early 2003, the Hibernate development team began Hibernate2 releases, which offered many significant improvements over the first release.

JBoss, Inc. (now part of Red Hat) later hired the lead Hibernate developers in order to further its development.

Actually Hibernate is much more than ORM Tool (Object - Relational Mapping) because today it is providing lots of features in the persistence data layer.

ORM means Objects to Relational Mapping i.e. Java Persistence Objects are mapped to Relational databases by using ORM tools.



Advantages of Hibernate

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

ORM

Hibernate ORM easily solves the data mismatch found between the object oriented classes of an application and relational database. ORM connects these two with ease through the use of the XML mapping file. It enables to gain complete control over the application as well the database design. This feature makes Hibernate flexible and powerful.

Transparent Persistence

Hibernate's transparent persistence ensures automatic connection between the application's objects with the database tables. This feature prevents developers from writing lines of connection code. Transparent persistence enables hibernate to reduce the development time and maintenance cost.

Database independent

Hibernate is database independent. It can be used to connect with any database like Oracle, MySQL, Sybase and DB2 to name a few. This cross database portability of Hibernate is easily achieved by changing a parameter 'database dialect' in the configuration file. Database independency is considered as one of the major advantages of Hibernate.

HQL

Hibernate supports a powerful query language called HQL (Hibernate Query Language). This query language is more powerful than SQL and is completely object oriented. HQL's advanced features like pagination and dynamic profiling are not present in SQL.
HQL can be used to implement some of the prominent object oriented concepts like inheritance, polymorphism and association.

Dual-layer Caching

Hibernate supports both first level and second level caching mechanisms. The first level caching is associated with Session object which is used by default. The second level caching is associated with Session Factory object.
Through caching concept, Hibernate retains the objects in cache so as to reduce repeated hits to the database. This feature makes Hibernate highly scalable and optimizes the application's performance.

Version Property

Hibernate supports optimistic locking through its version property feature. This functionality supports multiple transactions without affecting one another.
For example, when two or more users try to alter a database entity at the same time, the version field avoids the conflict and gives preference to the user who commits the changes first. The other user will be prompted with an error message and will be asked to restart the process.

Open Source

Hibernate is available as an open source software with zero cost product license. This light weight software can be downloaded from its source website hibernate.org.

Scalability

Hibernate is highly scalable. It adapts itself in any environment. It may be an intranet application with few hundreds of users or large critical application with thousands of users. Hibernate supports both the applications equally.

Lazy-Loading

The lazy-loading concept fetches only the necessary object that is required for the execution of an application.

For example, if there is one parent class and n number of child classes, during an execution, there is no need to load all the child classes. Instead, only the class that is required for the query or join need to be loaded. This concept of lazy-loading prevents the unnecessary loading of objects. It enhances the performance of the application.

Why We Have to Use ORM Tools?

Object-relational mapping, in the purest sense, is a programming technique that supports the conversion of incompatible types in object-oriented programming languages, specifically between a data store and programming objects. You can use an ORM framework to persist model objects to a relational database and retrieve them, and the ORM framework will take care of converting the data between the two otherwise incompatible states. Most ORM tools rely heavily on metadata about both the database and objects, so that the objects need to know nothing about the database and the database doesn't need to know anything about how the data is structured in the application. ORM

provides a clean separation of concerns in a well-designed data application, and the database and application can each work with data in its native form.

The key feature of ORM is the mapping it uses to bind an object to its data in the database. Mapping expresses how an object and its properties and behaviors are related to one or more tables and their fields in the database. An ORM uses this mapping information to manage the process of converting data between its database and object forms, and generating the SQL for a relational database to insert, update, and delete data in response to changes the application makes to data objects.

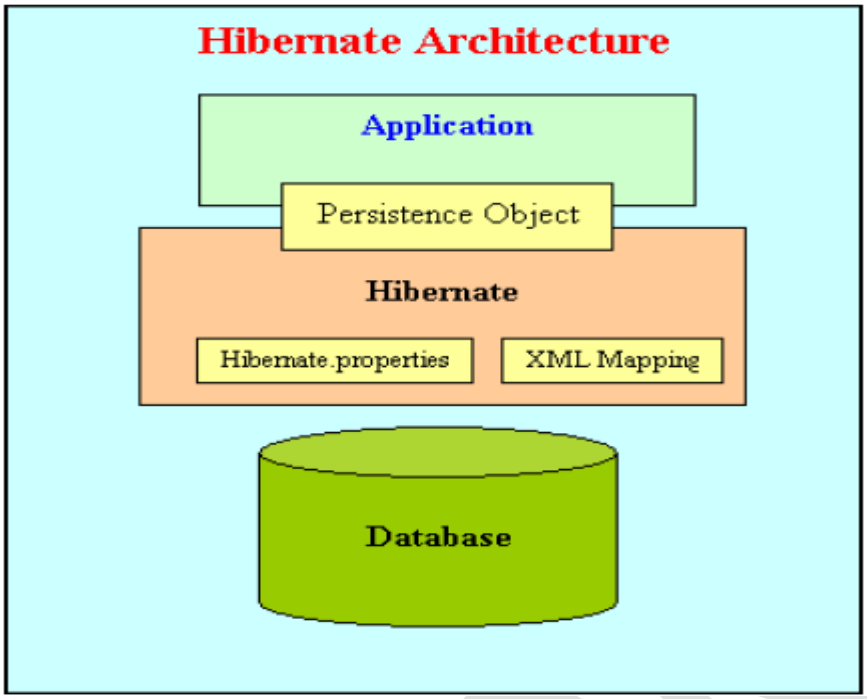
There are a number of benefits to using an ORM for development of databased applications

- **Productivity:** The data access code is usually a significant portion of a typical application, and the time needed to write that code can be a significant portion of the overall development schedule. When using an ORM tool, the amount of code is unlikely to be reduced—in fact, it might even go up—but the ORM tool generates 100% of the data access code automatically based on the data model you define, in mere moments.
- **Application design**: A good ORM tool designed by very experienced software architects will implement effective design patterns that almost force you to use good programming practices in an application. This can help support a clean separation of concerns and independent development that allows parallel, simultaneous development of application layers.
- **Code Reuse**: If you create a class library to generate a separate DLL for the ORM-generated data access code, you can easily reuse the data objects in a variety of applications. This way, each of the applications that use the class library need have no data access code at all.
- **Application Maintainability**: All of the code generated by the ORM is presumably well-tested, so you usually don't need to worry about testing it extensively. Obviously you need to make sure that the code does what you need, but a widely used ORM is likely to have code banged on by many developers at all skill levels. Over the long term, you can refactor the database schema or the model definition without affecting how the application uses the data objects.

Hibernate Architecture

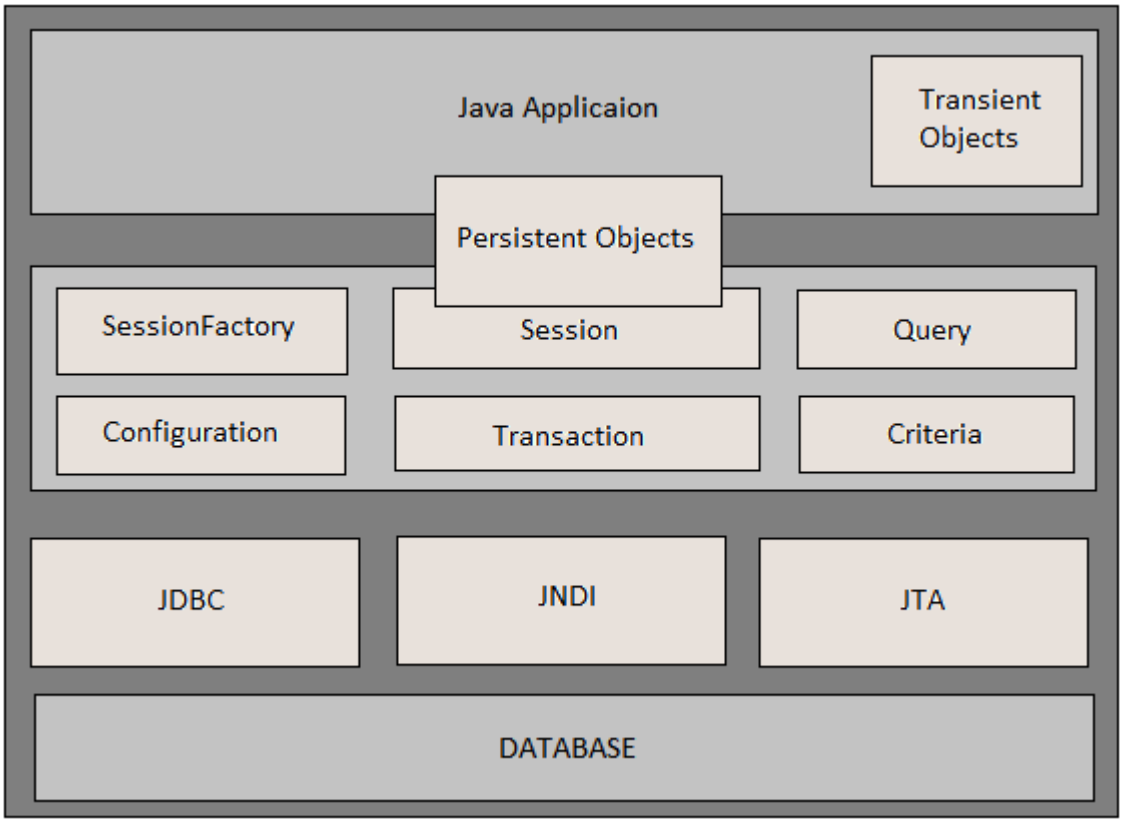
The Hibernate architecture is layered to keep us isolated from having to know the underlying APIs. Hibernate makes use of the database and configuration data to provide persistence services (and persistent objects) to the application.

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



Java program will expose the data in the form of objects, Using Hibernate we can store those objects into the database directly.

Internal Structure of Hibernate application



Configuration

Configuration object is created to load the configuration files like hibernatate.cfg.xml and hbm files (Java objects to database mapping). Since this step is loading of configurations, it generally happens at the application initialization time.

```
Configuration cfg  = new Configuration();
```

SessionFactory

We would always need one instance of Session factory per database that our application is interacting with. So if we have two different databases, we would create two session factory objects.

As Session factory is a heavy weight object, creation of the session factory object is an expensive operation and recommended to get it created at application start up.

Obtaining sessionFactory

```
==========================================================================
protected void setUp() throws Exception {
        // A SessionFactory is set up once for an application!
        final StandardServiceRegistry registry = new StandardServiceRegistryBuilder()
                        .configure() // configures settings from hibernate.cfg.xml
                        .build();
        try {
                sessionFactory = new MetadataSources( registry
).buildMetadata().buildSessionFactory();
        }
        catch (Exception e) {
                // The registry would be destroyed by the SessionFactory, but we had trouble
building the SessionFactory
                // so destroy it manually.
                StandardServiceRegistryBuilder.destroy( registry );
        }
}
```

The setUp method first builds a org.hibernate.boot.registry.StandardServiceRegistry instance which incorporates configuration information into a working set of Services for use by the SessionFactory. In this tutorial we defined all configuration information in hibernate.cfg.xml so there is not much interesting to see here.

Using the StandardServiceRegistry we create the org.hibernate.boot.MetadataSources which is the start point for telling Hibernate about your domain model. Again, since we defined that in hibernate.cfg.xml so there is not much interesting to see here.

org.hibernate.boot.Metadata represents the complete, partially validated view of the application domain model which the SessionFactory will be based on.

The final step in the bootstrap process is to build the SessionFactory. The SessionFactory is a thread-safe object that is instantiated once to serve the entire application.

The SessionFactory acts as a factory for org.hibernate.Session instances, which should be thought of as a corollary to a "unit of work".

Session

Session objects are created using Session factory and are a lightweight object. Session objects provide a connection with a relational database.

By design, we should create a new session object when a database interaction is needed. A session object represents a unit of work.

Session objects are not thread safe and hence should not keep open for a long time.

```
Session session = sessionFactory.openSession();
```

Transaction

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

```
Transaction tx = session.beginTransaction();//start
      //operations
tx.commit(); //end
```

Hibernate Application Requirements

Any hibernate application, for example consider even first hello world program must always contains 4 files totally.

- POJO class
- Mapping XML or Annotations
- Configuration XML
- One java file to write our business logic

Actually these are the minimum requirement to run any hibernate application, and in fact we may require any number of POJO classes and any number of mapping xml files (Number of POJO classes = that many number of mapping xmls), and only one configuration xml and finally one java file to write our logic.

Hibernate Persistent Classes or POJO classes

Persistent classes are those java classes whose objects have to be stored in the database tables. They should follow some simple rules of Plain Old Java Object programming model (POJO).

a) A persistence class should have a default constructor
b) A persistence class should have an id to uniquely identify the class objects.All attributes should be declared as private
c) Public getter and setters shold be defined to access the class attribues

==========Product.java==========

```
public class Product {
	private Integer pid;
	private String pname;
	private Double price;

	public Integer getPid() {
		return pid;
	}
	public void setPid(Integer pid) {
		this.pid = pid;
	}
	public String getPname() {
		return pname;
	}
	public void setPname(String pname) {
		this.pname = pname;
	}
	public Double getPrice() {
		return price;
	}
	public void setPrice(Double price) {
		this.price = price;
	}
}
```

==========

Hibernate configuration file

Hibernate works as an intermediate layer between java application and relational database. So hibernate needs some configuration setting related to the database and other parameters like mapping files.

A hibernate configuration file mainly contains three types of information

- Connection Properties related to the database.
- Hibernate Properties related to hibernate behavior.
- Mapping files entries related to the mapping of a POJO class and a database table.

Note: No. of hibernate configuration files in an application is dependence upon the no. of database uses. No. of hibernate configuration files are equals to the no. of database uses.

```
<hibernate-configuration>
      <session-factory>
            // Connection Properties
            <property name="connection.driver_class">driverClassName</property>
            <property name="connection.url">jdbcConnectionURL</property>
            <property name="connection.user">databaseUsername</property>
            <property name="connection.password">databasePassword</property>
            // Hibernate Properties
            <property name="show_sql">true/false</property>
            <property name="dialet">databaseDialetClass</property>
            <property name="hbm2ddl.auto">like create/update</property>
            // Mapping files entries
            <mapping resource="mappingFile1.xml" />
            <mapping resource="mappingFile2.xml" />
      </session-factory>
</hibernate-configuration>
```

Hibernate mapping file

Hibernate mapping file is used by hibernate framework to get the information about the mapping of a POJO class and a database table.

It mainly contains the following mapping information

- Mapping information of a POJO class name to a database table name.
- Mapping information of POJO class properties to database table columns.

Elements of the Hibernate mapping file:

1. **hibernate-mapping:** It is the root element.
2. **Class:** It defines the mapping of a POJO class to a database table.
3. **Id:** It defines the unique key attribute or primary key of the table.
4. **generator:** It is the sub element of the id element. It is used to automatically generate the id.
5. **property:** It is used to define the mapping of a POJO class property to database table column.

```
<hibernate-mapping>
     <class name="POJO class name" table="table name in database">
           <id name="propertyName" column="columnName" type="propertyType">
                 <generator class="generatorClass" />
           </id>
           <property name="propertyName1" column="colName1" type="propertyType " />
           <property name="propertyName2" column="colName2" type="propertyType " />
           ….
     </class>
</hibernate-mapping>
```

Hibernate First Application

To create Hibernate application we need to create below files

Product.java (POJO class)

Product.hbm.xml (Xml mapping file)

hibernate.cfg.xml (Xml configuration file)

ClientForSave.java (java file to write our hibernate logic)

==========Product.java==========

```
public class Product {

	private Integer productId;
	private String proName;
	private Double price;

	//Setters and getters

}
```

==========

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
	<class name="Products" table="PRODUCTS">
		<id name="productId" type="int" column="PID">
		</id>
		<property name="procName" column="PRODUCT_NAME" />
		<property name="price" column="PRICE" />
	</class>
</hibernate-mapping>
```

==========-Product.hbm.xml==========

In this mapping file, Product class is linked with PRODUCTS table in the database, and next is the id element, means in the database table what column we need to take as primary key column, that property name we need to give here, here property name “productId” which will mapped with “pid “ column in the table.

And “proName” is mapped with “pname” column of the PRODUCTS table, here we have not specified any column for the property price, this means that, our property name in the pojo class and the column name in the table both are same.

Remember: the first 3 lines is the DTD for the mapping file, as a programmer no need to remember but we need to be very careful while we are copying this DTD, program may not be executed if we write DTD wrong, actually we have separate DTD’s for Mapping xml and Configuration xml files.

==========**hibernate.cfg.xml**==========

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

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

```
		<!-- Related to Database information -->
		<property name="hibernate.connection.driver_class">
			oracle.jdbc.driver.OracleDriver
		</property>

		<property name="hibernate.connection.url">
			oracle:jdbc:thin:@localhost:1521/XE
		</property>
		<property name="hibernate.connection.username">
			hibernate
		</property>
		<property name="hibernate.connection.password">
			hibernate@123
		</property>

		<!-- Related to Hibernate Properties -->
		<property name="show_sql">true</property>
		<property name="hbm2ddl.auto">update</property>
		<property
name="dialect">org.hibernate.dialect.OracleDialect</property>

		<!-- List of XML mapping files -->
		<mapping resource="Prodcut.hbm.xml" />

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

==========SavePRoduct.java==========

```
public class SaveProduct {
	public static void main(String[] args) {
		Configuration cfg = new Configuration()
		cfg.configure("hibernate.cfg.xml");
		StandardServiceRegistry registry = new
StandardServiceRegistryBuilder().configure().build();
	SessionFactory sessionFactory = new MetadataSources( registry
).buildMetadata().buildSessionFactory();

		Session hsession = sessionFactory.openSession();
		Transaction tx = hsession.beginTransaction();

		Product p = new Product();
		p.setProductId(101);
		p.setProName("Apple");
		p.setPrice(35000.50);

		hsession.save(p);
		tx.commit();
		hsession.close();
		sessionFactory.close();
	}
}
```

==========SaveProduct.java==========

CURD Operations in Hibernate

Storing entity into table

In hibernate, we generally use one of below two versions of save() method:

```
public Serializable save(Object object) throws HibernateException

public Serializable save(String entityName,Object object) throws HibernateException
```

Both **save()** methods take a transient object reference (which must not be null) as an argument. Second method takes an extra parameter '***entityName***' which is useful in case you have mapped multiple entities to a Java class. Here you can specify which entity you are saving using **save()** method

Hibernate Select Query Example

1) Loading an hibernate entity using session.load() method

Hibernate's Session interface provides several load() methods for loading entities from your database. Each load() method requires the object's primary key as an identifier, and it is mandatory to provide it. In addition to the ID, Hibernate also needs to know which class or entity name to use to find the object with that ID. After the load()method returns, you need to cast the returned object to suitable type of class to further use it. It's all what load()method need from you to work it correctly.

Let's look at different flavors of load() method available in hibernate session:

1. public Object load(Class theClass, Serializable id) throws HibernateException
2. public Object load(String entityName, Serializable id) throws HibernateException
3. public void load(Object object, Serializable id) throws HibernateException

==========RetriveProduct.java file==========

```
public class SaveProduct {
	public static void main(String[] args) {
		Configuration cfg = new Configuration()
		cfg.configure("hibernate.cfg.xml");
		StandardServiceRegistryBuilder serviceRegistryBuilder = new
StandardServiceRegistryBuilder();

	serviceRegistryBuilder.applySettings(configuration.getProperties());
		ServiceRegistry serviceRegistry =
serviceRegistryBuilder.build();
		SessionFactory sessionFactory =
configuration.buildSessionFactory(serviceRegistry);
		Session hsession = sessionFactory.openSession();
		Transaction tx = hsession.beginTransaction();

		Object obj =  hsession.load(Product.class,new Integer(1));
		Prodcut p = (Product) obj;
		System.out.println(p);

		tx.commit();
		hsession.close();
		sessionFactory.close();
	}
}
```

2) Loading an hibernate entity using session.get() method

The get() method is very much similar to load() method. The get() methods take an identifier and either an entity name or a class. There are also two get() methods that take a lock mode as an argument, but we will discuss lock modes later. The rest get() methods are as follows:

1. public Object get(Class clazz, Serializable id) throws HibernateException
2. public Object get(String entityName, Serializable id) throws HibernateException

Difference between load() and get() method in hibernate session

The difference between both methods lies in return value "if the identifier does not exist in database". ***In case of get() method you will get return value as NULL if identifier is absent; But in case of load() method, you will get a runtime exception***

org.hibernate.ObjectNotFoundException: No row with the given identifier exists

Hibernate Update Query Example

To update entity in the database we have two methods in hibernate

```
public void update(Object obj)

public void saveOrUpdate(Object obj)
```

==========UpdateProduct.java file start==========

```
public class SaveProduct {
      public static void main(String[] args) {
            //Get the SessionFactory Object
            //Get the Session Object
            //Begin Transaction

      Product p =  (Product)hsession.get(Product.class,new Integer(1));
            p.setProductId(101);
            p.setPrice(35000.89);
            hsession.update(p);

            //commit the transaction
            //close operations
      }
}
```

==========UpdateProduct.java file end ==========

Using saveOrUpdate(Object obj) method

In the program replace update() method saveOrUpdate() method.

Differences between save(), persiste(), saveOrUpdate() and update() methods

save() method will store entity into database and returns Serializable Id

persist() method will store entity into database and will not return any value

saveOrUpdate() method, if entity already available in database it will update otherwise it will store entity into database

update() method is used to update the record, when we call update(), if that entity not available in database it will throw exception

Hibernate Delete Query Example

==========DeleteProduct.java file start ==========

```
public class SaveProduct {
            public static void main(String[] args) {
                  //Get the SessionFactory Object
                  //Get the Session Object
                  //Begin Transaction

      Product p =  (Product)hsession.get(Product.class,new Integer(1));
            p.setProductId(101);
            hsession.delete(p);

            //commit the transaction
            //close operations
            }
      }
```

==========DeleteProduct.java file end==========

Auto DDL in Hibernate

One Of the major advantage of hibernate is Auto DDL support. It will create the tables Automatically. To enable this feature we have to configure "**hibernate.hbm2ddl.auto"** property in Hibernate configuration file.

For this hbm.2ddl.auto property we can below 4 values

validate: validate the schema, makes no changes to the database.
update: update the schema.
create: creates the schema, destroying previous data.
create-drop: drop the schema at the end of the session.

configure this property in hibernate.cfg.xml file like below

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

<property name="hibernate.hbm2ddl.auto">validate</property>

<property name="hibernate.hbm2ddl.auto">update</property>

<property name="hibernate.hbm2ddl.auto">create-drop</property>

Enable Logging and Commenting

Hibernate can output the underlying SQL behind your HQL queries into your application's log file. This is especially useful if the HQL query does not give the results you expect, or if the query takes longer than you wanted. This is not a feature you will have to use frequently, but it is useful should you have to turn to your database administrators for help in tuning your Hibernate application.

1) Logging: The easiest way to see the SQL for a Hibernate HQL query is to enable SQL output in the logs with the "show_sql" property. Set this property to true in your hibernate.cfg.xml configuration file and Hibernate will output the SQL into the logs. When you look in your application's output for the Hibernate SQL statements, they will be prefixed with "Hibernate".

```
<property name="hibernate.show_sql"> true <property>
```

2) Commenting: Tracing your HQL statements through to the generated SQL can be difficult, so Hibernate provides a commenting facility on the Query object that lets you apply a comment to a specific query. The Query interface has a setComment() method that takes a String object as an argument, as follows:

```
public Query setComment(String comment)
```

Hibernate will not add comments to your SQL statements without some additional configuration, even if you use the setComment() method. You will also need to set a Hibernate property, hibernate.use_sql_comments, to true in your Hibernate configuration. If you set this property but do not set a comment on the query programmatically, Hibernate will include the HQL used to generate the SQL call in the comment. I find this to be very useful for debugging HQL.

Use commenting to identify the SQL output in your application's logs if SQL logging is enabled.

Note : Creating SessionFactory is a heavy weight operation, so it is not recommended to create more than one instance for SessionFactory in applications.

Singleton Design Pattern

Singleton pattern will ensure that there is only one instance of a class is created in the Java Virtual Machine. It is used to provide global point of access to the object

Singleton class rules

- Static member : This contains the instance of the singleton class.
- Private constructor : This will prevent anybody else to instantiate the Singleton class.
- Static public method : This provides the global point of access to the Singleton object and returns the instance to the client calling class.

```
public class SingletonExample {

    // Static member holds only one instance of the
    // SingletonExample class

    private static SingletonExample singletonInstance;

    // SingletonExample prevents any other class from instantiating
    private SingletonExample() {
    }

    // Providing Global point of access
    public static synchronized SingletonExample getSingletonInstance() {
```

```
        if (null == singletonInstance) {
            singletonInstance = new SingletonExample();
        }
        return singletonInstance;
    }

    public void printSingleton(){
        System.out.println("Inside print Singleton");
    }

}
```

Now Create HibernateUtil class which provides one instance of SessionFactory, from now onwards we will get SessionFactory object from this class.

```
public class HibernateUtil {

    private static SessionFactory sessionFactory = buildSessionFactory();
    private static SessionFactory buildSessionFactory() {
        try {
            if (sessionFactory == null) {
                Configuration configuration = new Configuration().configure(
                                                HibernateUtil.class.getResource("/hibernate.cfg.xml"));
                StandardServiceRegistryBuilder serviceRegistryBuilder = new StandardServiceRegistryBuilder();
                serviceRegistryBuilder.applySettings(configuration.getProperties());
                ServiceRegistry serviceRegistry = serviceRegistryBuilder.build();
                sessionFactory = configuration.buildSessionFactory(serviceRegistry);
            }
            return sessionFactory;
        } catch (Throwable ex){
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory(){
        return sessionFactory;
    }

    public static void close(){
        getSessionFactory().close();
    }

}
```

Lifecycle of POJO Class Objects

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

There are three types of Hibernate object states.

1. Transient Object State:
An object which is not associated with hibernate session and does not represent a row in the database is considered as transient. It will be garbage collected if no other object refers to it. An object that is created for the first time using the new() operator is in transient state. When the

object is in transient sate then it will not contain any identifier (primary key value). You have to use save, persist or saveOrUpdate methods to persist the transient object.

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

3. Detached Object State
Object which is just removed from hibernate session is called as detached object.The sate of the detached object is called as detached state.When the object is in detached sate then it contain identity but you can't do persistence operation with that identity.

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

Generators in Hibernate

Generator classes are used to generate the 'identifier or primary key value' for a persistent object while saving an object in database.

- Hibernate provides different primary key generator algorithms.
- All hibernate generator classes implements hibernate.id.IdentifierGenerator interface, and overrides the generate(SessionImplementor,Object) method to generate the 'identifier or primary key value'.
- If we want our own user defined generator, then we should implement IdentiferGenerator interface and override the generate()
- <generator /> tag (which is sub element of <id /> tag) is used to configure generator class in mapping file.

All generators implement the interface org.hibernate.id.IdentifierGenerator. This is a very simple interface; some applications may choose to provide their own specialized implementations. However, Hibernate provides a range of built-in implementations. There are shortcut names for the built-in generators:

increment	generates identifiers of type long, short or int that are unique only when no other process is inserting data into the same table. Do not use in a cluster.
identity	supports identity columns in DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL. The returned identifier is of type long, short or int.
sequence	uses a sequence in DB2, PostgreSQL, Oracle, SAP DB, McKoi or a generator in Interbase. The returned identifier is of type long, short or int
hilo	uses a hi/lo algorithm to efficiently generate identifiers of type long, short or int, given a table and column (by default hibernate_unique_key and next_hi respectively) as a source of hi values. The hi/lo algorithm generates identifiers that are unique only for a particular database.
seqhilo	uses a hi/lo algorithm to efficiently generate identifiers of type long, short or int, given a named database sequence.
uuid	uses a 128-bit UUID algorithm to generate identifiers of type string, unique within a network (the IP address is used). The UUID is encoded as a string of hexadecimal digits of length 32. guid uses a database-generated GUID string on MS SQL Server and MySQL.
native	picks identity, sequence or hilo depending upon the capabilities of the underlying database. assigned lets the application to assign an identifier to the object before save() is called. This is the default strategy if no <generator> element is specified.
select	retrieves a primary key assigned by a database trigger by selecting the row by some unique key and retrieving the primary key value
foreign	uses the identifier of another associated object. Usually used in conjunction with a <one-to-one> primary key association.

Composite Primary Key in Hibernate

- If the database table has more than one column as primary key then we call it as composite primary key.
- If the table has one primary key then in hibernate mapping file we need to configure this column by using <id> element.
- if the table has multiple primary key columns , in order to configure these primary key we need to use <composite-id=""> element in our hibernate hbm file

Let us consider there is a need for persisting a Book Object in a table BOOK_INFO.Book class contains properties such as isbn, bookName, authorName, category, price out of which isbn, bookName and authorName are part of a composite primary key fields in the table BOOK_INFO. Steps to implement this requirement is mentioned below.

To implement this application, follow the steps mentioned below.

==========Create Book.java==========

```
import java.io.Serializable;

public class Book implements Serializable {

	private int isbn;
	private String bookName;
	private String authorName;
	private String category;
	private Double price;

	private static final long serialVersionUID = 4430953665101945676L;
```

```
	// Setter and Getters

}
=================================================================

<?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="info.ashok.Book" table="BOOK">
		<composite-id>
			<key-property name="isbn" column="ISBN" />
			<key-property name="bookName" column="BOOKNAME" />
		</composite-id>
		<property name="authorName" column="AUTHORNAME" />
		<property name="price" column="PRICE" />
	</class>
</hibernate-mapping>

============================Test.java=============================================
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class Test {
	public static void main(String[] args) {
		SessionFactory factory = HibernateUtil.getSessionFactory();
		Session session = factory.openSession();
		Book book = new Book();
		book.setIsbn(101);
		book.setBookName("Head First Java");
		book.setAuthorName("Kathy");
		book.setPrice(new Double("500.00"));
		Transaction tx = session.beginTransaction();
		session.save(book);
		session.flush();
		Book book1 = new Book();
		book1.setIsbn(101);
		book1.setBookName("Head First Java");
		book1.setAuthorName("Kathy");
		Book book2 = (Book) session.load(Book.class, book1);
		System.out.println("Category : " + book2.getCategory() + "\nPrice :
"
				+ book2.getPrice().toString());
		tx.commit();
		session.close();
		factory.close();
	}
}
```

Composite Primary Key In Hibernate With Select Query

```
public class ForOurLogic4Load {

	public static void main(String[] args) {
		SessionFactory factory = HibernateUtil.buildSessionFactory();
```

```
		Session session = factory.openSession();

		Product p = new Product();
		p.setProductId(101);
		p.setPrice(25000);

		Object o = session.get(Product.class, p);
		// here p must be an serializable object,
		Product p1 = (Product) o;
		System.out.println("The price is: " + p1.getProName());

		System.out.println("Object Loaded successfully.....!!");
		session.close();
		factory.close();
	}

}
```

==========

Component Mapping in Hibernate

A Component mapping is a mapping for a class having a reference to another class as a member variable.

A component is a contained object that is persisted as a value type and not an entity reference. The term "component" refers to the object-oriented notion of composition and not to architecture-level components.

```
============================Customer.java=======================================
import java.util.Date;
public class Customer implements java.io.Serializable {

	private Integer custId;
	private String custName;
	private int age;
	private Address address;
	//setters and getters
}

============================== Address.java=====================================

public class Address implements java.io.Serializable {

	private String city;
	private String state;
	private String country;
	//setters and getters
}
================================================================================
```

In this case, Address.java is a “component” represent the “city”, “state” and "country" columns for Customer.java

```
======================== Customer.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.mkyong.customer.Customer" table="customer"
		catalog="mkyongdb">
```

```
		<id name="custId" type="java.lang.Integer">
			<column name="CUST_ID" />
			<generator class="identity" />
		</id>
		<property name="custName" type="string">
			<column name="CUST_NAME" length="10" not-null="true" />
		</property>

		<component name="Address" class="com.mkyong.customer.Address">
			<property name="city" type="string">
				<column name="CITY" not-null="true" />
			</property>
			<property name="state" type="string">
				<column name="STATE" not-null="true" />
			</property>
			<property name="country" type="string">
				<column name="COUNTRY" not-null="true" />
			</property>
		</component>
	</class>
</hibernate-mapping>

================================ App.java=====================================
public class App {
	public static void main(String[] args) {

	System.out.println("Hibernate component mapping");
	Session session = HibernateUtil.getSessionFactory().openSession();

	session.beginTransaction();

	Address address = new Address();
	address.setCity("HYD");
	address.setState("TG");
	address.setCountry("INDIA");

	  Customer cust = new Customer();
	  cust.setCustName("Sunil");
	  cust.setAddress(address);
	session.save(cust);
	session.getTransaction().commit();
	System.out.println("Done");
	}
}
================================================================================
```

Connection Pooling in Hibernate

By default, Hibernate uses JDBC connections in order to interact with a database. Creating these connections is expensive—probably the most expensive single operation Hibernate will execute in a typical-use case. Since JDBC connection management is so expensive that possibly you will advise to use a pool of connections, which can open connections ahead of time (and close them only when needed, as opposed to “when they’re no longer used”).

Thankfully, Hibernate is designed to use a connection pool by default, an internal implementation. However, Hibernate’s built-in connection pooling isn’t designed for production use. In production, you

would use an external connection pool by using either a database connection provided by JNDI or an external connection pool configured via parameters and classpath.

Hibernate supports a variety of connection pooling mechanisms. If you are using an application server, you may wish to use the built-in pool (typically a connection is obtaining using JNDI). If you can't or don't wish to use your application server's built-in connection pool,

Hibernate Supports for below Exxternal Connection pools

c3p0 - Distributed with Hibernate

Apache DBCP - Apache Pool

Proxool - Distributed with Hibernate

Configure C3P0 Connection Pool

```
<hibernate-configuration>

 <session-factory>
  <property
name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
  <property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/myschema</prope
rty>
  <property name="hibernate.connection.username">user</property>
  <property name="hibernate.connection.password">password</property>
  <property
name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
  <property name="show_sql">true</property>

  <property name="hibernate.c3p0.min_size">5</property>
  <property name="hibernate.c3p0.max_size">20</property>
  <property name="hibernate.c3p0.timeout">300</property>
  <property name="hibernate.c3p0.max_statements">50</property>
  <property name="hibernate.c3p0.idle_test_period">3000</property>

  . . . .
</session-factory>
================================================================================
```

Explanation about the properties:

hibernate.c3p0.min_size – Minimum number of JDBC connections in the pool. Hibernate default: 1
hibernate.c3p0.max_size – Maximum number of JDBC connections in the pool. Hibernate default: 100
hibernate.c3p0.timeout – When an idle connection is removed from the pool (in second). Hibernate default: 0, never expire.
hibernate.c3p0.max_statements – Number of prepared statements will be cached. Increase performance. Hibernate default: 0 , caching is disable.
hibernate.c3p0.idle_test_period – idle time in seconds before a connection is automatically validated. Hibernate default: 0

Configure Apache DBCP Connection Pool

Apache Connection Pool can be downloaded from http://commons.apache.org/dbcp/
In order to integrate this pool with Hibernate you will need the following jars: **commons-dbcp.jar** and **commons-pool-1.5.4.jar**.

Here's a sample configuration in hibernate.cfg.xml:

```
<hibernate-configuration>
 <session-factory>
  <property
name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
  <property
name="hibernate.connection.url">jdbc:mysql://localhost:3306/myschema</prope
rty>
  <property name="hibernate.connection.username">user</property>
  <property name="hibernate.connection.password">password</property>
  <property
name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
  <property name="show_sql">true</property>

  <property name="hibernate.dbcp.initialSize">8</property>
  <property name="hibernate.dbcp.maxActive">20</property>
  <property name="hibernate.dbcp.maxIdle">20</property>
  <property name="hibernate.dbcp.minIdle">0</property>

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

==========

Batch Processing

Many applications within the enterprise domain require bulk processing to perform business operations in mission critical environments. These business operations include automated, complex processing of large volumes of information that is most efficiently processed without user interaction. These operations typically include time based events (e.g. month-end calculations, notices or correspondence), periodic application of complex business rules processed repetitively across very large data sets (e.g. Insurance benefit determination or rate adjustments), or the integration of information that is received from internal and external systems that typically requires formatting, validation and processing in a transactional manner into the system of record. Batch processing is used to process billions of transactions every day for enterprises.

Suppose there is one situation in which we have to insert 1000000 records in to database in a time. So what to do in this situation...

==========

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<1000000; i++ )
{
    Student student = new Student(.....);
    session.save(student);
}
tx.commit();
session.close();
```

==========

By default, Hibernate will cache all the persisted objects in the session-level cache and ultimately your application would fall over with an OutOfMemoryException somewhere around the 50,000th row. You can resolve this problem if you are using batch processing with Hibernate.

To use the batch processing feature, first set hibernate.jdbc.batch_size as batch size to a number either at 20 or 50 depending on object size. This will tell the hibernate container that every X rows to be inserted as batch. To implement this in your code we would need to do little modification as follows:

==========

```
Session session = SessionFactory.openSession();

Transaction tx = session.beginTransaction();
for ( int i=0; i<1000000; i++ )
{
    Student student = new Student(.....);
    session.save(employee);
    if( i % 50 == 0 ) // Same as the JDBC batch size
    {
        //flush a batch of inserts and release memory:
        session.flush();
        session.clear();
    }
}
tx.commit();
session.close();
```

==========

Above code will work fine for the INSERT operation, but if you want to make UPDATE operation then you can achieve using the following code:

==========

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
ScrollableResults studentCursor = session.createQuery("FROM
STUDENT").scroll();
int count = 0;
while(studentCursor .next())
 {
   Student student = (Student) studentCursor.get(0);
   student.setName("DEV");
   seession.update(student);
   if ( ++count % 50 == 0 ) {
      session.flush();
      session.clear();
   }
}
tx.commit();
session.close();
```

==========

If you are undertaking batch processing you will need to enable the use of JDBC batching. This is absolutely essential if you want to achieve optimal performance. Set the JDBC batch size to a reasonable number (10-50).

```
hibernate.jdbc.batch_size 50
```

==========

SessionFactory sf = HibernateUtil.getSessionFactory();

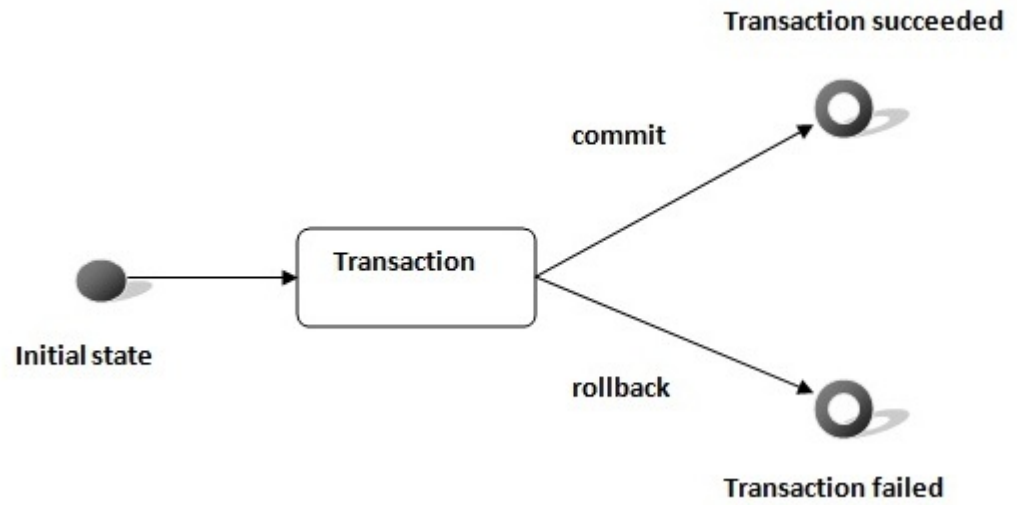
```
  Session session = sf.openSession();
  Transaction transaction = session.beginTransaction();

  for ( int i=0; i<100000; i++ )
  {
            String studentName = "DINESH " + i;
            int rollNumber = 9 + i;
            String course = "MCA " + i;
            Student student = new Student();
            student.setStudentName(studentName);
            student.setRollNumber(rollNumber);
            student.setCourse(course);
            session.save(student);
          if( i % 50 == 0 )
          {
```

```
            session.flush();
            session.clear();
        }
}
transaction.commit();
session.close();
sf.close();
```

Transaction management in Hibernate

A **transaction** simply represents a unit of work. In such case, if one step fails, the whole transaction fails (which is termed as atomicity). A transaction can be described by ACID properties (Atomicity, Consistency, Isolation and Durability).



Transaction Interface in Hibernate

In hibernate framework, we have **Transaction** interface that defines the unit of work. It maintains abstraction from the transaction implementation (JTA,JDBC).

A transaction is associated with Session and instantiated by calling **session.beginTransaction()**.

The methods of Transaction interface are as follows:

1. **void begin()** starts a new transaction.
2. **void commit()** ends the unit of work unless we are in FlushMode.NEVER.
3. **void rollback()** forces this transaction to rollback.
4. **void setTimeout(int seconds)** it sets a transaction timeout for any transaction started by a subsequent call to begin on this instance.
5. **boolean isAlive()** checks if the transaction is still alive.
6. **void registerSynchronization(Synchronization s)** registers a user synchronization callback for this transaction.
7. **boolean wasCommited()** checks if the transaction is commited successfully.

8. **boolean wasRolledBack()** checks if the transaction is rolledback successfully.

Example :

```
Session session = null;
Transaction tx = null;

try {
      session = sessionFactory.openSession();
      tx = session.beginTransaction();
      // some action

      tx.commit();

} catch (Exception ex) {
      ex.printStackTrace();
      tx.rollback();
} finally {
      session.close();

}
```

==========

Annotations

Java Annotations allow us to add metadata information into our source code, although they are not a part of the program itself. Annotations were added to the java from JDK 5. Annotation has no direct effect on the operation of the code they annotate (i.e. it does not affect the execution of the program)

Annotations are given by SUN as replacement to the use of xml files in java
Every annotations is internally an Interface, but the key words starts with @ symbol

What's the use of Annotations?

1) Instructions to the compiler: There are three built-in annotations available in Java (@Deprecated, @Override & @SuppressWarnings) that can be used for giving certain instructions to the compiler. For example the @override annotation is used for instructing compiler that the annotated method is overriding the method. More about these built-in annotations with example is discussed in the next sections of this article.

2) Compile-time instructors: Annotations can provide compile-time instructions to the compiler that can be further used by software build tools for generating code, XML files etc.

3) Runtime instructions: We can define annotations to be available at runtime which we can access using java reflection and can be used to give instructions to the program at runtime. We will discuss this with the help of an example, later in this same post.

Annotations basics
An annotation always starts with the symbol @ followed by the annotation name. The symbol @ indicates to the compiler that this is an annotation.

For e.g. @Override

Where we can use annotations?

Annotations can be applied to the classes, interfaces, methods and fields. For example the below annotation is being applied to the method.

Built-in Annotations in Java

Java has three built-in annotations:

@Override

@Deprecated

@SuppressWarnings

1) @Override:

While overriding a method in the child class, we should use this annotation to mark that method. This makes code readable and avoid maintenance issues, such as: while changing the method signature of parent class, you must change the signature in child classes (where this annotation is being used) otherwise compiler would throw compilation error. This is difficult to trace when you haven't used this annotation.

```
public class Parent {

    public void justaMethod() {
        System.out.println("Parent class method");
    }
}

public class Child extends Parent {

    @Override
    public void justaMethod() {
        System.out.println("Child class method");
    }
}
```

2) @Deprecated

@Deprecated annotation indicates that the marked element (class, method or field) is deprecated and should no longer be used. The compiler generates a warning whenever a program uses a method, class, or field that has already been marked with the @Deprecated annotation. When an element is deprecated, it should also be documented using the Javadoc @deprecated tag, as shown in the following example. Make a note of case difference with @Deprecated and @deprecated. @deprecated is used for documentation purpose.

```
@Deprecated
public void anyMethodHere(){
    // Do something
}
```

Now, whenever any program would use this method, the compiler would generate a warning

3) @SuppressWarnings

This annotation instructs compiler to ignore specific warnings. For example in the below code, I am calling a deprecated method (lets assume that the method deprecatedMethod() is marked with @Deprecated annotation) so the compiler should generate a warning, however I am using @SuppressWarnings annotation that would suppress that deprecation warning.

```
@SuppressWarnings("deprecation")
    void myMethod() {
        myObject.deprecatedMethod();
}
```

Let us see few points regarding annotations in hibernate

- ✓ In hibernate annotations are given to replace hibernate mapping [xml] files
- ✓ While working with annotations in hibernate, we do not require any mapping files, but hibernate xml configuration file is must
- ✓ Hibernate borrowed annotations from java persistence API but hibernate itself doesn't contain its own annotations

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

@**Table** annotation specifies the table name where data of this entity is to be persisted. If you are not using @Table annotation in Entity class, hibernate will use the class name as the table name by default.

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

@GeneratedValue annotation is used to specify the primary key generation strategy to use. If the strategy is not specified by default *AUTO* will be used**.**

@**Column** annotation specifies the details of the column for this property or field. If @Column annotation is not specified, property name will be used as the column name by default. We can use column annotation with the following most commonly used attributes

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

==========Product.java==========

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

@Entity
@Table(name = "PRODUCTS")
public class Product {

	@Id
	@Column(name = "proid")
	private Integer productId;

	@Column(name = "proName", length = 10)
	private String proName;

	@Column(name = "price")
	private Double price;

	// setters and getters
}
```

HQL (Hibernate Query Language)

So far we done the operations on single object (single row), here we will see modifications, updates on multiple rows of data (multiple objects) at a time. In hibernate we can perform the operations on a single row (or) multiple rows at a time, if we do operations on multiple rows at once, then we can call this as bulk operations.

HQL is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties

HQL queries are translated by Hibernate into conventional SQL queries

Advantages Of HQL

- ✓ HQL is database independent, means if we write any program using HQL commands then our program will be able to execute in all the databases with out doing any further changes to it
- ✓ HQL supports object oriented features ike ***Inheritance***, ***polymorphism***, ***Associations***(Relation ships)
- ✓ HQL is initially given for selecting object from database and in hibernate 3.x we can do DML operations (insert, update…) too

Example :

```
// In SQL
sql> select * from Product
Note: Product is the table name...!!!
```

```
// In HQL
hql> select p from Product p
   [ or ]
   from Product p
Note: here p is the reference...!!
```

- If we want to load the **Partial Object** from the database that is only selective properties (selected columns) of an objects then we need to replace column names with POJO class variable names.

```
// In SQL
sql> select pid,pname from Product
Note: pid, pname are the columns Product is the table

// In HQL
hql> select p.productid,p.productName from Product p
   [ or ]
   from Product p ( we should not start from, from key word here because
we selecting the columns hope you are getting me )

Note: here p is the reference...!!
        productid,productName are POJO variables
```

- It is also possible to **load** or **select** the object from the database **by passing run time values** into the query, in this case we can use either " ? " symbol or label in an HQL command, the index number of " ? " will starts from zero but not one (Remember this, little important regarding interview point of view)

Example

```
// In SQL
sql> select * from Product where pid=?
Note: Product is the table

// In HQL
hql> select p from Product p where p.productid=?
   [ or ]
   select p from Product p where p.productid=:java4s
   [ or ]
   from Product p where p.productid=?
   [ or ]
   from Product p where p.productid=:java4s

Note: Here p is the reference...!!
```

==========Student.java==========

```
@Entity
@Table(name=”STUDENTS”)
public class Student {
	// data members

	@Id
	@Column(name=”student_id”)
	private int studentId;
	@Column(name=”first_name”)
	private String firstName;
	@Column(name=”last_name”)
	private String lastName;
	@Column(name=”roll_no”)
	private String rollNo;

	// Generate Setters and Getters
}
```

==========Create MyApp.java==========

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

		// Create the student object.
		Student student = new Student();
		// Setting the object properties.
		student.setFirstName("Vivek");
		student.setLastName("Solenki");
		student.setClassName("MCA ");
		student.setRollNo("MCA/07/70");
		student.setAge(27);
		// Get the session object.
		Session session = HibernateUtil.getSessionFactory().openSession();
		// Start hibernate transaction.
		session.beginTransaction();
		// Persist the student object.
		session.save(student);

		// Update the student object.
		Query query1 = session.createQuery("update Student"
				+ " set className = 'MCA final'"
				+ " where rollNo = 'MCA/07/70'");
		query1.executeUpdate();

		// select a student record
		Query query2 = session
				.createQuery("FROM Student where rollNo =
'MCA/07/70'");
		Student stu1 = (Student) query2.uniqueResult();
		System.out.println("First Name: " + stu1.getFirstName());
		System.out.println("Last Name: " + stu1.getLastName());
		System.out.println("Class: " + stu1.getClassName());
		System.out.println("RollNo: " + stu1.getRollNo());
		System.out.println("Age: " + stu1.getAge());

		// select query using named parameters
		Query query3 = session
```

```
                        .createQuery("FROM Student where rollNo = :rollNo");
                query3.setParameter("rollNo", "MCA/07/70");
                Student stu2 = (Student) query3.uniqueResult();

                System.out.println("First Name: " + stu2.getFirstName());
                System.out.println("Last Name: " + stu2.getLastName());
                System.out.println("Class: " + stu2.getClassName());
                System.out.println("RollNo: " + stu2.getRollNo());
                System.out.println("Age: " + stu2.getAge());

                // select query using positional parameters
                Query query4 = session.createQuery("FROM Student where rollNo = ?");
                query4.setString(0, "MCA/07/70");
                Student stu3 = (Student) query4.uniqueResult();
                System.out.println("First Name: " + stu3.getFirstName());
                System.out.println("Last Name: " + stu3.getLastName());
                System.out.println("Class: " + stu3.getClassName());
                System.out.println("RollNo: " + stu3.getRollNo());
                System.out.println("Age: " + stu3.getAge());

                // delete a student record
                Query query5 = session
                        .createQuery("delete Student where rollNo =
'MCA/07/70'");
                query5.executeUpdate();
                // Commit hibernate transaction.
                session.getTransaction().commit();
                // Close the hibernate session.
                session.close();
        }
}
```

==========

Pagination in Hibernate

Pagination through the result set of a database query is a very common application pattern. Typically, you would use pagination for a web application that returned a large set of data for a query. The web application would page through the database query result set to build the appropriate page for the user. The application would be very slow if the web application loaded all of the data into memory for each user. Instead, you can page through the result set and retrieve the results you are going to display one chunk at a time.

There are two methods on the Query interface for paging: setFirstResult() and setMaxResults(). ThesetFirstResult() method takes an integer that represents the first row in your result set, starting with row 0. You can tell Hibernate to only retrieve a fixed number of objects with the setMaxResults() method. Your HQL is unchanged—you need only to modify the Java code that executes the query.

```
Query query = session.createQuery("from Student");
query.setFirstResult(1);
query.setMaxResults(5);
List results = query.list();
displayStudentsList(results);
```

Hibernate Criteria Query Language (HCQL)

HCQL stands for Hibernate Criteria Query Language. As we discussed HQL provides a way of manipulating data using objects instead of database tables. Hibernate also provides more object oriented alternative ways of HQL. Hibernate Criteria API provides one of these alternatives. HCQL is mainly used in search operations and works on filtration rules and logical conditions.

Unlike HQL, Criteria is only for selecting the data from the database, that to we can select complete objects only not partial objects, in fact by combining criteria and projections concept we can select partial objects too. We can't perform non-select operations using this criteria. Criteria is suitable for executing dynamic queries too, let us see how to use this criteria queries in the hibernate..

The Criteria API allows you to build up a criteria query object programmatically; the org.hibernate.Criteria interface defines the available methods for one of these objects. The Hibernate Session interface contains severalcreateCriteria() methods. Pass the persistent object's class or its entity name to the createCriteria() method, and Hibernate will create a Criteria object that returns instances of the persistence object's class when your application executes a criteria query.

The simplest example of a criteria query is one with no optional parameters or restrictions—the criteria query will simply return every object that corresponds to the class.

```
Criteria crit = session.createCriteria(Student.class);
List<Student> results = crit.list();
```

Using Restrictions with Criteria

The Criteria API makes it easy to use restrictions in your queries to selectively retrieve objects; for instance, your application could retrieve only products with a price over $30. You may add these restrictions to a Criteria object with the add() method. The add() method takes an org.hibernate.criterion.Criterion object that represents an individual restriction. You can have more than one restriction for a criteria query.

Hibernate criteria restrictions query example

Restrictions class provides the methods to restrict the search result based on the restriction provided.

Commonly used Restriction class methods:

1. Restrictions.eq: Make a restriction that the property value must be equal to the specified value.

Syntax: Restrictions.eq("property", specifiedValue)

2. Restrictions.lt: Make a restriction that the property value must be less than the specified value.

Syntax: Restrictions.lt("property", specifiedValue)

3. Restrictions.le: Make a restriction that the property value must be less than or equal to the specified value.

Syntax: Restrictions.le("property", specifiedValue)

4. Restrictions.gt: Make a restriction that the property value must be greater than the specified value.

Syntax: Restrictions.gt("property", specifiedValue)

5. Restrictions.ge: Make a restriction that the property value must be greater than or equal to the specified value.

Syntax: Restrictions.ge("property", specifiedValue)

6. Restrictions.like: Make a restriction that the property value follows the specified like pattern.

Syntax: Restrictions.like("property", "likePattern")

7. Restrictions.between: Make a restriction that the property value must be between the start and end limit values.

Syntax: Restrictions.between("property", startValue, endValue)

8. Restrictions.isNull: Make a restriction that the property value must be null.

Syntax: Restrictions.isNull("property")

9. Restrictions.isNotNull: Make a restriction that the property value must not be null.

Syntax: Restrictions.isNotNull("property")

i) Restrictions.eq() Example

To retrieve objects that have a property value that "**equals**" your restriction, use the `eq()` method on `Restrictions`, as follows:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.eq("description","Mouse"));
List<Product> results = crit.list()
```

Above query will search all products having description as "Mouse".

ii) Restrictions.ne() Example

To retrieve objects that have a property value "`not equal to`" your restriction, use the `ne()` method on`Restrictions`, as follows:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.ne("description","Mouse"));
List<Product> results = crit.list()
```

Above query will search all products having description anything but not "Mouse".

iii) Restrictions.like() and Restrictions.ilike() Example

Instead of searching for exact matches, we can retrieve all objects that have a property matching part of a given pattern. To do this, we need to create an SQL LIKE clause, with either the `like()` or the `ilike()` method. The`ilike()` method is case-insensitive.

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.like("name","Mou%",MatchMode.ANYWHERE));
List<Product> results = crit.list();
```

Above example uses an org.hibernate.criterion.MatchMode object to specify how to match the specified value to the stored data. The MatchMode object (a type-safe enumeration) has four different matches:
ANYWHERE: Anyplace in the string
END: The end of the string
EXACT: An exact match
START: The beginning of the string

v) Restrictions.isNull() and Restrictions.isNotNull() Example

The `isNull()` and `isNotNull()` restrictions allow you to do a search for objects that have (or do not have) null property values.

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.isNull("name"));
List<Product> results = crit.list();
```

v) Restrictions.gt(), Restrictions.ge(), Restrictions.lt() and Restrictions.le() Examples

Several of the restrictions are useful for doing math comparisons. The greater-than comparison is gt(), the greater-than-or-equal-to comparison is ge(), the less-than comparison is lt(), and the less-than-or-equal-to comparison isle(). We can do a quick retrieval of all products with prices over $25 like this, relying on Java's type promotions to handle the conversion to Double:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.gt("price", 25.0));
List<Product> results = crit.list();
```

vi) Combining Two or More Criteria Examples

Moving on, we can start to do more complicated queries with the Criteria API. For example, we can combine AND and OR restrictions in logical expressions. When we add more than one constraint to a criteria query, it is interpreted as an AND, like so:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.lt("price",10.0));
crit.add(Restrictions.ilike("description","mouse", MatchMode.ANYWHERE));
List<Product> results = crit.list();
```

If we want to have two restrictions that return objects that satisfy either or both of the restrictions, we need to use the or() method on the Restrictions class, as follows:

```
Criteria crit = session.createCriteria(Product.class);
Criterion priceLessThan = Restrictions.lt("price", 10.0);
Criterion mouse = Restrictions.ilike("description", "mouse", MatchMode.ANYWHERE);
LogicalExpression orExp = Restrictions.or(priceLessThan, mouse);
crit.add(orExp);
List results=crit.list();
```

Obtaining a Unique Result

Sometimes you know you are going to return only zero or one object from a given query. This could be because you are calculating an aggregate or because your restrictions naturally lead to a unique result. If you want obtain a single Object reference instead of a List, the uniqueResult() method on the Criteria object returns an object or null. If there is more than one result, the uniqueResult() method throws a HibernateException.

The following short example demonstrates having a result set that would have included more than one result, except that it was limited with the setMaxResults() method:

```
Criteria crit = session.createCriteria(Product.class);
Criterion price = Restrictions.gt("price",new Double(25.0));
crit.setMaxResults(1);
Product product = (Product) crit.uniqueResult();
```

Hibernate criteria ordering query example

Order class provides the methods for performing ordering operations.

Methods of Order class:

1. Order.asc: To sort the records in ascending order based on the specified property.

Syntax: Order.asc("property")

2. Order.desc: To sort the records in descending order based on the specified property.

Syntax: Order.desc("property")

Hibernate criteria projections query example

Instead of working with objects from the result set, you can treat the results from the result set as a set of rows and columns, also known as a projection of the data. This is similar to how you would use data from a SELECT query with JDBC.

To use projections, start by getting the org.hibernate.criterion.Projection object you need from theorg.hibernate.criterion.Projections factory class. The Projections class is similar to the Restrictions class in that it provides several static factory methods for obtaining Projection instances. After you get a Projection object, add it to your Criteria object with the setProjection() method. When the Criteria object executes, the list contains object references that you can cast to the appropriate type.

Example 1 : Single Aggregate (Getting Row Count)

```
Criteria crit = session.createCriteria(Product.class);
crit.setProjection(Projections.rowCount());
List<Long> results = crit.list();
```

Projections class provides the methods to perform the operation on a particular column.

Other aggregate functions available through the Projections factory class include the following:

1. **avg(String propertyName)**: Gives the average of a property's value
2. **count(String propertyName)**: Counts the number of times a property occurs
3. **countDistinct(String propertyName)**: Counts the number of unique values the property contains
4. **max(String propertyName)**: Calculates the maximum value of the property values
5. **min(String propertyName)**: Calculates the minimum value of the property values
6. **sum(String propertyName)**: Calculates the sum total of the property values

Getting Selected Columns

Another use of projections is to retrieve individual properties, rather than entities. For instance, we can retrieve just the name and description from our product table, instead of loading the entire object representation into memory.

```
Criteria crit = session.createCriteria(Product.class);
ProjectionList projList = Projections.projectionList();
projList.add(Projections.property("name"));
projList.add(Projections.property("description"));
crit.setProjection(projList);
crit.addOrder(Order.asc("price"));
List<object[]> results = crit.list();
```

Hibernate Native SQL Queries

Hibernate does provide a way to use native SQL statements directly through Hibernate. One reason to use native SQL is that your database supports some special features through its dialect of SQL that are not supported in HQL. Another reason is that you may want to call stored procedures from your Hibernate application.

You can modify your SQL statements to make them work with Hibernate's ORM layer. You do need to modify your SQL to include Hibernate aliases that correspond to objects or object properties. You can specify all properties on an object with {objectname.*}, or you can specify the aliases directly with {objectname.property}. Hibernate uses the mappings to translate your object property names

into their underlying SQL columns. This may not be the exact way you expect Hibernate to work, so be aware that you do need to modify your SQL statements for full ORM support. You will especially run into problems with native SQL on classes with subclasses—be sure you understand how you mapped the inheritance across either a single table or multiple tables, so that you select the right properties off the table.

Underlying Hibernate's native SQL support is the org.hibernate.SQLQuery interface, which extends theorg.hibernate.Query interface.

Your application will create a native SQL query from the session with the createSQLQuery() method on the Session interface.

```
public SQLQuery createSQLQuery(String queryString) throws HibernateException
```

Hibernate Named Queries

Named queries in hibernate is a **technique to group the HQL statements in single location**, and lately refer them by some name whenever need to use them. It **helps largely in code cleanup**because these HQL statements are no longer scattered in whole code.

Apart from above, below are some minor **advantages** of named queries

1. **Fail fast**: Their syntax is checked when the session factory is created, making the application fail fast in case of an error.
2. **Reusable**: They can be accessed and used from several places

Hibernate mapping file example

```
<hibernate-mapping>

	<query name="Give Query Name">
		from Product p where p.price = :price
	</query>

</hibernate-mapping>
```

```
Query qry = session.getNamedQuery("Name we given in hibernate
mapping xml");
qry.setParameter("java4s", new Integer(1022));
List l = qry.list();
```

Syntax Of hibernate mapping file [For Native SQL]

```
<hibernate-mapping>

	<sql-query name="Give Query Name">
		select * from PRODUCTS
	</sql-query>

</hibernate-mapping>
```

Named queries with Annotations

```
@Entity
@Table(name = "DEPARTMENT", uniqueConstraints = {@UniqueConstraint(columnNames = "ID"),
                    @UniqueConstraint(columnNames = "NAME") })
@NamedQueries
(
    {
        @NamedQuery(name=DepartmentEntity.GET_DEPARTMENT_BY_ID, query=DepartmentEntity.GET_DEPARTMENT_BY_ID_QUERY),
        @NamedQuery(name=DepartmentEntity.UPDATE_DEPARTMENT_BY_ID, query=DepartmentEntity.UPDATE_DEPARTMENT_BY_ID_QUERY)
    }
)
public class DepartmentEntity implements Serializable {

    static final String GET_DEPARTMENT_BY_ID_QUERY = "from DepartmentEntity d where d.id = :id";
    public static final String GET_DEPARTMENT_BY_ID = "GET_DEPARTMENT_BY_ID";

    static final String UPDATE_DEPARTMENT_BY_ID_QUERY = "UPDATE DepartmentEntity d SET d.name=:name where d.id = :id";
    public static final String UPDATE_DEPARTMENT_BY_ID = "UPDATE_DEPARTMENT_BY_ID";

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID", unique = true, nullable = false)
    private Integer id;

    @Column(name = "NAME", unique = true, nullable = false, length = 100)
    private String name;
      //setters and getters
}
```

Working with Procedures in Hibernate

What is a Stored Procedure?

A stored procedure or in simple a proc is a named PL/SQL block which performs one or more specific task. This is similar to a procedure in other programming languages.
A procedure has a header and a body. The header consists of the name of the procedure and the parameters or variables passed to the procedure. The body consists or declaration section, execution section and exception section similar to a general PL/SQL Block.

A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage.
We can pass parameters to procedures in three ways.
1) IN-parameters
2) OUT-parameters
3) IN OUT-parameters

Syntax to create a procedure is:

CREATE [OR REPLACE] PROCEDURE proc_name [list of parameters]
IS
Declaration section
BEGIN
Execution section
EXCEPTION
Exception section
END;

IS - marks the beginning of the body of the procedure and is similar to DECLARE in anonymous PL/SQL Blocks. The code between IS and BEGIN forms the Declaration section.

The syntax within the brackets [ ] indicate they are optional. By using CREATE OR REPLACE together the procedure is created if no other procedure with the same name exists or the existing procedure is replaced with the current code.

Procedure With IN Paramters

CREATE OR REPLACE PROCEDURE INSERTEMPRECORDPROC(

EID IN NUMBER,

ENAME IN VARCHAR2,

ESAL IN NUMBER,

EGENDER IN CHARACTER)

AS

BEGIN

INSERT INTO EMPLOYEES (EMP_ID,EMP_NAME,EMP_SALARY,EMP_GENDER) VALUES (EID,ENAME,ESAL,EGENDER);

END INSERTEMPRECORDPROC;

```
public class CallProcDemo{
public static void main(String[] args) {
            getAllEmpRecords(101,'Ashok',25000,'M');
      }

      public void insertEmpRecord(Integer id,String name,Double
salary,Character gender) {
            try {
                  SessionFactory sf = HibernateUtil.getSessionFactory();
                  Session hsession = sf.openSession();
                  Transaction tx = hsession.beginTransaction();

            StoredProcedureQuery query = hsession

      .createStoredProcedureQuery("INSERTEMPRECORDPROC");

            query.registerStoredProcedureParameter(0, Integer.class,
                              ParameterMode.IN);

            query.registerStoredProcedureParameter(1, String.class,
                              ParameterMode.IN);

            query.registerStoredProcedureParameter(2, Integer.class,
                              ParameterMode.IN);

            query.registerStoredProcedureParameter(3, Character.class,
                              ParameterMode.IN);

                  query.setParameter(0, id);
                  query.setParameter(1, name);
                  query.setParameter(2, salary);
                  query.setParameter(3, gender);

                  query.executeUpdate();
```

```
                tx.commit();

                hsession.close();
                sf.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
      }

}
```

Procedure with IN OUT parameters - GetEmpName By ID

CREATE OR REPLACE PROCEDURE GETEMPRECORDBYID(eid in number,ename out varchar2) AS

BEGIN

SELECT EMP_NAME INTO ENAME FROM EMPLOYEES WHERE EMP_ID=EID;

END GETEMPRECORDBYID;

```
public class CallProcDemo {

      public static void main(String[] args) {
            getEmpRecordById(101);
      }
      public static void void getEmpRecordById(Integer empId) {
            try {
                SessionFactory sf = HibernateUtil.getSessionFactory();
                Session hsession = sf.openSession();
                Transaction tx = hsession.beginTransaction();
                StoredProcedureQuery spQuery = hsession

      .createStoredProcedureQuery("GETEMPRECORDBYID");
                spQuery.registerStoredProcedureParameter(0,
Integer.class,
                            ParameterMode.IN);
                spQuery.registerStoredProcedureParameter(1, String.class,
                            ParameterMode.OUT);
                spQuery.setParameter(0, empId);
                spQuery.execute();
                String name = (String)
spQuery.getOutputParameterValue(1);
                System.out.println(name);
                tx.commit();
                hsession.close();
                sf.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
      }
}
```

==========

Procedure with OUT Paramter as RefCursor - GETALLEMPS

CREATE OR REPLACE

PROCEDURE GETALLEMPS(

EMPS OUT SYS_REFCURSOR)

AS

BEGIN

*OPEN EMPS FOR SELECT * FROM EMPLOYEES;*

END GETALLEMPS;

```
public class CallProcDemo {

	public static void main(String[] args) {
		getAllEmpRecords();
	}

	public void getAllEmpRecords() {
		try {
			SessionFactory sf = HibernateUtil.getSessionFactory();
			Session hsession = sf.openSession();
			Transaction tx = hsession.beginTransaction();
			StoredProcedureQuery spQuery = hsession
					.createStoredProcedureQuery("GETALLEMPS");
			spQuery.registerStoredProcedureParameter(1, Class.class,
					ParameterMode.REF_CURSOR);
			spQuery.execute();
			List empList = spQuery.getResultList();
			Iterator itr = empList.iterator();
			while (itr.hasNext()) {
				Object[] objArr = (Object[]) itr.next();
				for (Object obj : objArr) {
					System.out.print(obj + "\t");
				}
				System.out.println();
			}
			System.out.println(empList.size());
			tx.commit();
			hsession.close();
			sf.close();
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

==========

Filters in Hibernate

With Hibernate3 there is a new way to filtering the results of searches. Sometimes it is required to only process a subset of the data in the underlying Database tables. Hibernate filters are very useful in those situations. Other approaches for these kind of problems is to use a database view or use a WHERE clause in the query or Hibernate Criteria API.

But Hibernate filters can be enabled or disabled during a Hibernate session. Filters can be parameterized also. This way one can manage the 'visibility' rules within the Integration tier. They

can be used in the scenarios where you need to provide the capability of security roles, entitlements or personalization.

When to Use Hibernate Filters

Let's take an example, consider a web application that does the reporting for various flights. In future course there is a change in requirement such that flights are to be shown as per their status (on time, delayed or cancelled).

This can also be done using a WHERE clause within the SQL SELECT query or Hibernate's HQL SELECT query. For a small application it is okay to do this, but for a large and complex application it might be a troublesome effort. Moreover it will be like searching each and every SQL query and making the changes in the existing code which have been thoroughly tested.

This can also be done using Hibernate's Criteria API but that also means changing the code at numerous places that is all working fine. Moreover in both the approaches, one need to be very careful so that they are not changing existing working SQL queries in inadvertent way.

Filters can be used like database views, but parameterized inside the application. This way they are useful when developers have very little control over DB operations. Here I am going to show you the usage of Hibernate filters to solve this problem. When the end users select the status, your application activates the flight's status for the end user's Hibernate session. Any SQL query will only return the subset of flights with the user selected status. Flight status is maintained at two locations- Hibernate Session and flight status filter.

One of the other use cases of filters I can think of is in the user's view of the organization data. A user can only view the data that he/she is authorized to. For example an admin can see data for all the users in the organization, manager can see the data for all the employees reporting to him/her in his/her group while an employee can only see his/her data. If a user moves from one group to another-with a very minimal change using Hibernate Filters this can be implemented.

How to Use Hibernate Filters

Hibernate filters are defined in Hibernate mapping documents (hbm.xml file)-which are easy to maintain. One can programmatically turn on or off the filters in the application code. Though filters can't be created at run time, they can be parameterized which makes them quite flexible in nature. We specify the filter on the column which is being used to enable/disable visibility rules. Please go thru the example application in which the filter is applied on flight status column and it must match a named parameter. After that at run time we specify one of the possible values.

```
============================hbm file=======================================
<?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.javacodegeeks.Student" table="student"
catalog="tutorials">
		<id name="movieId" type="java.lang.Integer">
			<column name="MOVIE_ID" />
			<generator class="identity" />
		</id>
		<property name="movieName" type="string">
			<column name="MOVIE_NAME" length="10" not-null="true"
				unique="true" />
		</property>
```

```
            <property name="releasedYr" type="string">
                  <column name="RELEASED_YEAR" length="20" not-null="true"/>
            </property>

             <filter name="movieFilter" condition="RELEASED_YEAR >=
:movieReleasedFilter"/>
      </class>

      <filter-def name="movieFilter">
            <filter-param name="movieReleasedFilter" type="java.lang.Integer" />
      </filter-def>

</hibernate-mapping>
```

==========

Now attach the filters to class or collection mapping elements. You can attach a single filter to more than one class or collection. To do this, you add a <filter> XML element to each class or collection. The <filter> XML element has two attributes viz. name and condition. The name references a filter definition (in the sample application it's : statusFilter) while condition is analogous to a WHERE clause in HQL. Please go thru the complete hibernate mapping file from the HibernateFilters.zip archive.

Note: Each <filter> XML element must correspond to a <filter-def> element. It is possible to have more than one filter for each filter definition, and each class can have more than one filter. Idea is to define all the filter parameters in one place and then refer them in the individual filter conditions.

In the java code, we can programmatically enable or disable the filter. By default the Hibernate Session doesn't have any filters enabled on it.

The Session interface contains the following methods:

```
    public Filter enableFilter(String filterName)

    public Filter getEnabledFilter(String filterName)

    public void disableFilter(String filterName)
```

The Filter interface contains some of the important methods:

```
    public Filter setParameter(String name, Object value)

    public Filter setParameterList(String name, Collection values)

    public Filter setParameterList(String name, Object[] values)
```

setParameter() method is mostly used. Be careful and specify only the type of java object that you have mentioned in the parameter at the time of defining filter in the mapping file.

The two setParameterList() methods are useful for using IN clauses in your filters. If you want to use BETWEEN clauses, use two different filter parameters with different names.

At the time of enabling the filter on session-use the name that you have provided in the mapping file for the filter name for the corresponding column in the table. Similarly condition name should contain one of the possible values for that column. This condition is being set on the filter.

```
==================== HibernateFilterDemo.java==============================
import java.util.List;

import org.hibernate.Filter;
import org.hibernate.Query;
import org.hibernate.Session;

import info.ashok.HibernateUtil;

public class App {
	public static void main(String[] args) {
		Session session = HibernateUtil.getSessionFactory().openSession();

		session.beginTransaction();

		Filter filter = session.enableFilter("movieFilter");
		filter.setParameter("movieReleasedFilter", 2015);

		Query query = session.createQuery("from movie");

		List<?> list = query.list();

		for (int i = 0; i < list.size(); i++) {

			Movie mv = (Student) list.get(i);
			System.out.println(mv);
		}

		session.getTransaction().commit();
	}
}
```

==========

Hibernate Filters with Annotations

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

import org.hibernate.annotations.Filter;
import org.hibernate.annotations.FilterDef;
import org.hibernate.annotations.ParamDef;

@Entity
@Table(name = "movie")
@FilterDef(name = "movieFilter", parameters = @ParamDef(name = "yearFilter", type
= "java.lang.String"))
@Filter(name = "movieFilter", condition = "released_in_year = :yearFilter")
public class Movie {

 @Id
 @GeneratedValue(strategy = GenerationType.IDENTITY)
 private long id;

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

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

```
 private String year;

 public long getId() {
  return id;
 }

 public void setId(long id) {
  this.id = id;
 }

 public String getName() {
  return name;
 }

 public void setName(String name) {
  this.name = name;
 }

 public String getYear() {
  return year;
 }

 public void setYear(String year) {
  this.year = year;
 }

 @Override
 public String toString() {
  return "Movie [id=" + id + ", name=" + name + ", year=" + year + "]";
 }

}

}
```

Hibernate Cache Mechanism

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.

First level cache - This is enabled by default and works in session scope
Second level cache - This is apart from first level cache which is available to be used globally in session factory scope
Query cache – It is used cache the the queries

Fist level cache in hibernate is enabled by default and you do not need to do anything to get this functionality working. In fact, you canot disable it even forcefully.

Its easy to understand the first level cache if we understand the fact that it is associated with Session object. As we know session object is created on demand from session factory and it is lost, once the session is closed. Similarly, first level cache associated with session object is available only till session object is live. It is available to session object only and is not accessible to any other session object in any other part of application.

Understanding Hibernate First Level Cache with Example

Caching is a 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 achieves the second goal by implementing first level cache.

Fist level cache in hibernate is enabled by default and you do not need to do anything to get this functionality working. In fact, you can not disable it even forcefully.

Its easy to understand the first level cache if we understand the fact that it is associated with Session object. As we know session object is created on demand from session factory and it is lost, once the session is closed. Similarly, first level cache associated with session object is available only till session object is live. It is available to session object only and is not accessible to any other session object in any other part of application.

Hibernate first level cache

- ✓ First level cache is associated with "session" object and other session objects in application can not see it.
- ✓ The scope of cache objects is of session. Once session is closed, cached objects are gone forever.
- ✓ First level cache is enabled by default and you can not disable it.
- ✓ When we query an entity first time, it is retrieved from database and stored in first level cache associated with hibernate session.
- ✓ If we query same object again with same session object, it will be loaded from cache and no sql query will be executed.
- ✓ 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.
- ✓ The whole session cache can be removed using clear() method. It will remove all the entities stored in cache.

==========

```
SessionFactory sf = HibernateUtil.getSessionFactory();
Session hsession = sf.openSession();
Transaction tx = hsession.beginTransaction();

Employee emp1 = (Employee) hsession.load(Employee.class, 1);
System.out.println(emp1);

Employee emp2 = (Employee) hsession.load(Employee.class, 1);
System.out.println(emp2);

tx.commit();
hsession.close();
sf.close();
```

==========

In Above program when we try to load Employee object twice in same session, we can see that second "session.load()" statement does not execute select query again and load the department entity directly.

Removing cache objects from first level cache example

Though we cannot disable the first level cache in hibernate, but we can certainly remove some of objects from it when needed. This is done using two methods :

public void evict(Object obj)

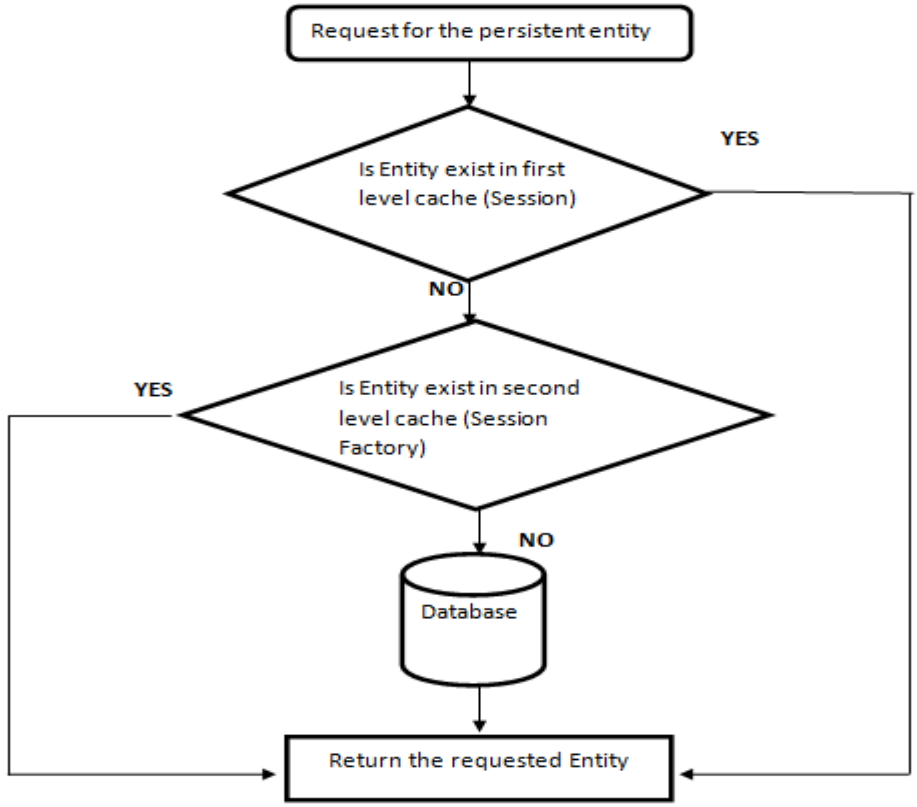
public void clear()

Here evict() is used to remove a particular object from cache associated with session, and clear() method is used to remove all cached objects associated with session. So they are essentially like remove one and remove all.

Second level cache

In Hibernate second level cache means Session Factory level cache. Hibernate Session is by default first level cache of persistent data. Sometimes we required to configure a cluster or you can say JVM(Session Factory) level cache. Cache is used to improve the performance of application by reducing your database hit. To configure second level cache, you have to put some additional effort as this is not provided by default. To use second level cache you have to use at least one Cache provider.

How second level cache works



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).

If cached copy of entity is present in first level cache, it is returned as result of load method.

If there is no cached entity in first level cache, then second level cache is looked up for cached entity.

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.

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.

Second level cache validate itself for modified entities, if modification has been done through hibernate session APIs.

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.

Every fresh session having its own cache memory, Caching is a mechanism for storing the loaded objects into a cache memory. The advantage of cache mechanism is, whenever again we want to load the same object from the database then instead of hitting the database once again, it loads from the local cache memory only, so that the no. of round trips between an application and a database server got decreased. It means caching mechanism increases the performance of the application.

Second level cache in the hibernate implemented by **4** vendors…

- Easy Hibernate [EHCache] Cache from hibernate framework
- Open Symphony [OS] cache from Open Symphony
- SwarmCache
- TreeCache from JBoss

How to enable second level cache in hibernate

To enable second level cache in the hibernate, then the following **3** changes are required

1. Add second level cache related properties in hibernate.cfg.xml file
2. Create EHCache.xml (in classpath folder)
3. Add @Cache annotation in POJO class

==========config file changes==========

```
<property name="hibernate.cache.use_second_level_cache">true</property>

<property name="hibernate.cache.region.factory_class">
                org.hibernate.cache.ehcache.EhCacheRegionFactory
</property>

<propertyname="net.sf.ehcache.configurationResourceName">
      ehcache.xml
</property>
```

==========ehcache.xml==========

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache>
      <defaultCache maxElementsInMemory="10000" eternal="false"
timeToIdleSeconds="100" timeToLiveSeconds="1000" />

</ehcache>
```

==========Employee.java==========

```
@Entity
@Table(name = "EMPLOYEES")
@Cache(usage=CacheConcurrencyStrategy.READ_ONLY)
public class Employee {
	//body
}
```

==========

For both options, caching strategy can be of following types:

none : No caching will happen.

read-only : If your application needs to read, but not modify, instances of a persistent class, a read-only cache can be used.

read-write : If the application needs to update data, a read-write cache might be appropriate.

nonstrict-read-write : If the application only occasionally needs to update data (i.e. if it is extremely unlikely that two transactions would try to update the same item simultaneously), and strict transaction isolation is not required, a nonstrict-read-write cache might be appropriate.

transactional : The transactional cache strategy provides support for fully transactional cache providers such as JBoss TreeCache. Such a cache can only be used in a JTA environment and you must specify hibernate.transaction.manager_lookup_class.

Example

==========Main.java==========

```
import org.hibernate.Session;

import com.javamakeuse.poc.pojo.Employee;
import com.javamakeuse.poc.util.HibernateUtility;

public class Main {
	public static void main(String[] args) {
		Session session =
HibernateUtility.getSessionFactory().openSession();

		session.beginTransaction();

		Employee Employee = null;

		Employee = (Employee) session.load(Employee.class, 1l);

		System.out.println("Employee from the Database => "+Employee);
		System.out.println();

		System.out.println("Going to print Employee *** from First
Level Cache");
		// second time loading same entity from the first level cache
		Employee = (Employee) session.load(Employee.class, 1l);
		System.out.println(Employee);

		// removing Employee object from the first level cache.
		session.evict(Employee);
		System.out.println("Object removed from the First Level
Cache");
		System.out.println();
```

```
            System.out.println("Going to print Employee *** from Second
level Cache");
            Employee = (Employee) session.load(Employee.class, 1l);
            System.out.println(Employee);
            session.getTransaction().commit();

            // loading object in another session
            Session session2 =
HibernateUtility.getSessionFactory().openSession();
            session2.beginTransaction();
            System.out.println();
            System.out
                        .println("Printing Employee *** from Second level
Cache in another session");
            Employee = (Employee) session2.load(Employee.class, 1l);
            System.out.println(Employee);
            session2.getTransaction().commit();

      }
}
```

==========

Query caching

The Query Cache does not cached the entities unlike Second Level Cache, It cached the queries and the return identifiers of the entities. Once it cached the identifiers than, It took help from Hibernate Second level cache to load the entities based on the identifiers value. So Query cache always used with Second level cache to get better result, because only query cache will cached the identifiers not the complete entities.

Enabling Query cache

```
<property key="hibernate.cache.use_query_cache">true</property>
```

and where queries are defined in our code, add the method call **setCacheable(true)** to the queries that should be cached:

```
Session hsession = sessionFactory.openSession();

Query query = hsession.createQuery("...")

query.setCacheable(true);

query.list();
```

==========Client.java==========

```
public class Main {
 public static void main(String[] args) {
  System.out.println(getEmp(1));
  System.out
    .println("Going to fetch Country from Query Cache*************");
  System.out.println(getEmp(1));
 }

 public static Country getEmp(long id) {

  Session session = HibernateUtility.getSessionFactory().openSession();

  Query query = session
    .createQuery("from Employee c where c.empId = :empId ");
```

```
  query.setParameter("empId", id);
  query.setMaxResults(1);
  query.setCacheable(true);
  return query.list() == null ? null : (Employee) query.list().get(0);

 }
}
```

In the above program, although we are calling getCountry(1) methods twice in main method, but it will generate only one sql query and next time it will fetched the records from the query cached and second level cache.

Relationships in Hibernate

The mapping of associations between entity classes and the relationships between tables is the soul of ORM. Following are the four ways in which the cardinality of the relationship between the objects can be expressed. An association mapping can be unidirectional as well as bidirectional.

Using hibernate, if we want to put relationship between two entities [Objects of two POJO classes], then in the database tables, there must exist foreign key relationship, we call it as Referential integrity.

The main advantage of putting relationship between objects is, we can do operation on one object, and the same operation can transfer onto the other object in the database [remember, object means one row in hibernate terminology]

While selecting, it is possible to get data from multiple tables at a time if there exits relationship between the tables, nothing but in hibernate relationships between the objects.

Using hibernate we can put the following 4 types of relationships

One-To-One

One-To-Many

Many-To-One

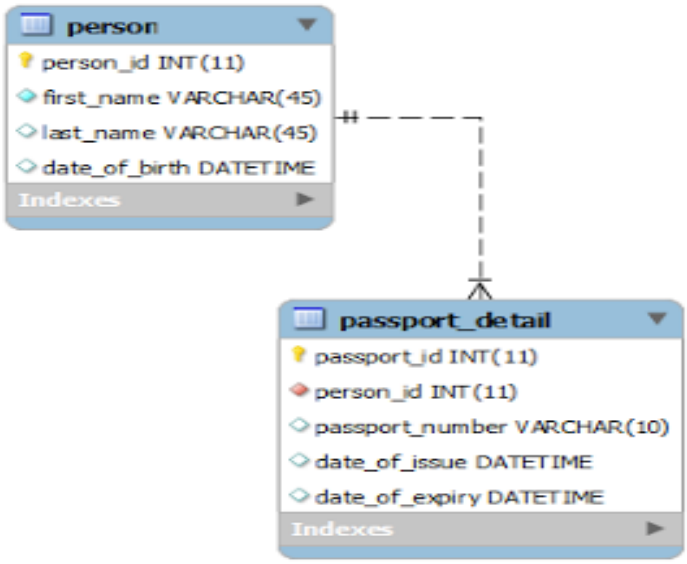
Many-To-Many

One to One Relation ship

One-to-One association means, Each row of table is mapped with exactly one and only one row with another table

For example, we have a passport_detail and person table each row of person table is mapped with exactly one and only one row of passport_detail table. One person has only one passport.

One-One relationship ER Diagram



==========Person.java==========

```
@Entity
@Table(name = "person")
public class Person {

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "person_id", unique = true, nullable = false)
	private Integer personId;

	@Column(name = "first_name")
	private String firstName;

	@Column(name = "last_name")
	private String lastName;

	@Temporal(TemporalType.DATE)
	@Column(name = "date_of_birth")
	private Date dateOfBirth;

	@OneToOne(mappedBy = "person", cascade = CascadeType.ALL)
	private PassportDetail passportDetail;

	//getters and setters

 }
```

==========PassportDetails.java==========

```
@Entity
@Table(name = "passport_details")
public class PassportDetails {

	@Id
	@GeneratedValue
	@Column(name = "PASSPORT_ID")
	private Integer passportId;

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

```
	private String passportNo;

	@OneToOne
	@JoinColumn(name = "PERSON_ID")
	private Person person;

	//getters and setters

}
```

```
public class OneToOneDemo {

	public static void main(String[] args) {
		OneToOneDemo otd = new OneToOneDemo();
		otd.insertRecord();
		//otd.retriveRecord();
	}
	public void retriveRecord() {
		try {
			SessionFactory sf = HibernateUtil.getSessionFactory();
			Session hsession = sf.openSession();
			Transaction tx = hsession.beginTransaction();
			Person p = hsession.get(Person.class, new Integer(25));
			tx.commit();
			hsession.close();
			sf.close();
		} catch (Exception e) {
			e.printStackTrace();
		}
	}

	public void insertRecord() {
		SessionFactory sf = HibernateUtil.getSessionFactory();
		Session hsession = sf.openSession();
		Transaction tx = hsession.beginTransaction();
		Person person = new Person();
		person.setFirstName("Tony");
		person.setLastName("Leo");
		SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
		PassportDetails passportDetail = new PassportDetails();
		try {
			person.setDateOfBirth(sdf.parse("1990-10-10"));
			passportDetail.setDateOfIssue(sdf.parse("2010-10-10"));
			passportDetail.setDateOfExpiry(sdf.parse("2020-10-09"));
		} catch (Exception e) {
			e.printStackTrace();
		}

		passportDetail.setPassportNo("Q12345678");

		person.setPassportDetails(passportDetail);
		passportDetail.setPerson(person);

		// saving person it will generate two insert query.
		System.out.println(hsession.save(person));
		tx.commit();
		hsession.close();
		sf.close();
	}
}
```

==========

That's it, Now run the PersonPassportService class you will get output at your console where two insert query will executed one for Person and one for PasspportDetail table.

One-To-Many Relationship

One to many association means each row of a table can be related to many rows in the relating tables.

For example, we have an author and book table, we all know that a particular book is written by a particular author. An author of a book is always one and only one person, co-author of a book may be multiple but author always only one. So many books can be written by an author, here many books an author relationship is one-to-many association example.
an author has many books that is. 1----------* books(one-to-may association)
Here is the example of author and book table for our One-To-Many association example

Author table:

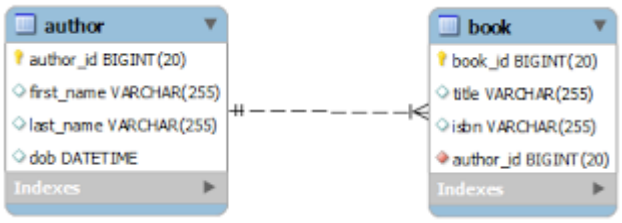
author_id	first_name	last_name	dob
101	Joshua	Bloch	1961-08-28
102	Watts	S. Humphrey	1927-07-04

Book table:

book_Id	title	isbn	author_id
1001	A Discipline for Software Engineering	0201546108	102
1002	Managing the Software Process	8177583301	102
1003	Effective Java	9780321356680	101
1004	Java Puzzlers	032133678X	101

From the above two tables we have seen that author id 101 has written two books(1003,1004) and author 102 is also written two books(1001,1002) but the same book is not written by each other. So it is clear that an author has many books. Lets start implementation of these in hibernate practically.

Here is an One-To-Many association mapping of an author and book table ER diagram:



==========Author.java==========

```
@Entity
@Table(name = "author")
public class Author {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "author_id", unique = true, nullable = false)
	private long authorId;

	@Column(name = "first_name")
	private String firstName;

	@Column(name = "last_name")
	private String lastName;

	@Temporal(TemporalType.DATE)
	@Column(name = "dob")
	private Date dateOfBirth;

	@OneToMany(mappedBy = "author", cascade = CascadeType.ALL)
	private Set<Book> books = new HashSet<>();

	//getters and settes

}
```

```
@Entity
@Table(name = "book")
public class Book {

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "book_id", unique = true, nullable = false)
	private long bookId;

	@Column(name = "title")
	private String title;

	@Column(name = "isbn")
	private String isbn;

	@ManyToOne
	@JoinColumn(name = "author_id", nullable = false)
	private Author author;

	//setters and getters

}
```

```
public class OneToMany {
	public static void main(String[] args) {
		 insertRecord();
		//retriveBooks();
		//deleteBook();
	}
	public static void insertRecord() {
		SessionFactory sf = HibernateUtil.getSessionFactory();
		Session hsession = sf.openSession();
		Transaction tx = hsession.beginTransaction();

		Author author = new Author();
		author.setAuthorName("Ramu");
		author.setAuthorDob(new Date());
```

```
		Book book1 = new Book();
		book1.setBookName("Design Patterns");
		book1.setIsbn("DP1234");
		book1.setAuthor(author);

		Book book2 = new Book();
		book2.setBookName("Spring In Action");
		book2.setIsbn("SP0987");
		book2.setAuthor(author);

		Set<Book> books = new HashSet<Book>();
		books.add(book1);
		books.add(book2);

		author.setBooks(books);
		hsession.save(author);

		tx.commit();
		hsession.close();
		sf.close();
	}

	public static void retriveBooks() {
		SessionFactory sf = HibernateUtil.getSessionFactory();
		Session hsession = sf.openSession();
		Author author = hsession.get(Author.class, new Integer(33));
		System.out.println(author);
		hsession.close();
		sf.close();
	}

	public static void deleteBook() {
		SessionFactory sf = HibernateUtil.getSessionFactory();
		Session hsession = sf.openSession();
		Transaction tx = hsession.beginTransaction();

		Author auth = new Author();
		auth.setAuthorId(33);
		Query query = hsession.createQuery("from Author where
authorId=39");
		hsession.delete(query.getResultList().get(0));

		tx.commit();
		hsession.close();
		sf.close();
	}

}
```

Many To Many Relationship

Many-To-Many association means, One or more row(s) of a table are associated one or more row(s) in another table. For example an employee may assigned with multiple projects, and a project is associated with multiple employees

Employee table:

employee_id	first_name	last_name	doj
101	Tony	Jha	2014-08-28
102	Zeneva	S. Humphrey	2014-07-04

Project table:

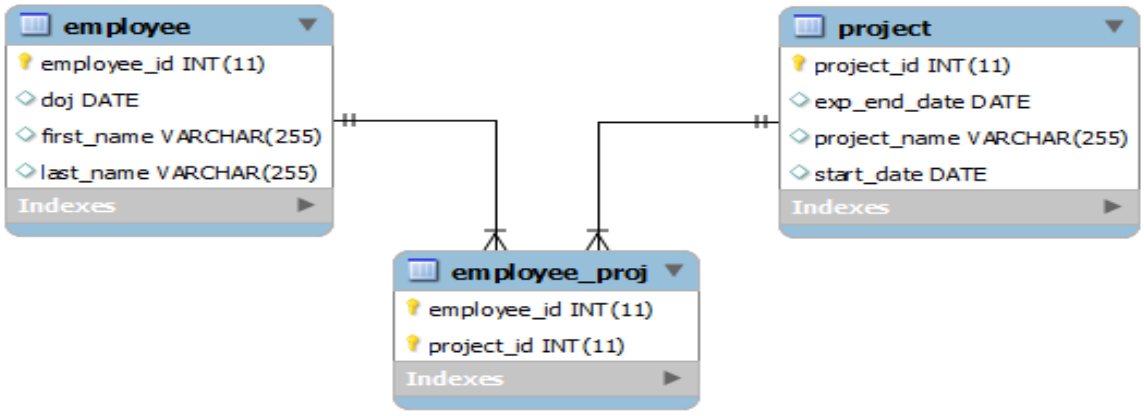
project_id	project_name	start_date	exp_end_date
101	MARS	2013-01-01	2015-08-28
102	SBA	2014-10-02	2016-07-04

Employee_Proj table:

employee_id	project_id
101	101
101	102
102	102

From the above Employee_Proj table it's clear that an employee 101 is associated with project(101,102) and project 102 is associated with employee(101,102). So it's clear that employee and project relationship is fall in Many-To-Many association categories.

Here is Many-To-Many association mapping table ER diagram.




```
=======================Employee.java=====================================
@Entity
@Table(name = "employee")
public class Employee {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "employee_id")
	private int employeeId;

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

```
	private String firstName;

	@Column(name = "last_name")
	private String lastName;

	@Column(name = "doj")
	@Temporal(TemporalType.DATE)
	private Date doj;

	@ManyToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
	@JoinTable(name = "employee_proj", joinColumns = { @JoinColumn(name =
"employee_id", nullable = false, updatable = false) }, inverseJoinColumns =
{ @JoinColumn(name = "project_id", nullable = false, updatable = false) })
	private Set<Project> projects;

	//setters and getters
}
```

```
@Entity
@Table(name = "project")
public class Project {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "project_id")
	private int projectId;

	@Column(name = "project_name")
	private String projectName;

	@Column(name = "start_date")
	@Temporal(TemporalType.DATE)
	private Date startDate;

	@Column(name = "exp_end_date")
	@Temporal(TemporalType.DATE)
	private Date expectedEndDate;

	@ManyToMany(fetch = FetchType.LAZY, mappedBy = "projects")
	private Set<Employee> employees;

		//setters and getters
}
```

```
public class ManyToManyDemo {
	public static void main(String[] args) {
		Employee employee = new Employee();

		SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
		try {

			SessionFactory sf = HibernateUtil.getSessionFactory();
			Session hsession  = sf.openSession();
			Transaction tx = hsession.beginTransaction();

			// employee details
			employee.setDoj(sdf.parse("2014-08-28"));
			employee.setFirstName("Tony");
			employee.setLastName("Jha");

			// project details
			Project mars = new Project();
			mars.setProjectName("MARS");
			mars.setStartDate(sdf.parse("2013-01-01"));
```

```
                mars.setExpectedEndDate(sdf.parse("2015-08-28"));

                // project 2 details
                Project sba = new Project();
                sba.setProjectName("SBA");
                sba.setStartDate(sdf.parse("2014-10-02"));
                sba.setExpectedEndDate(sdf.parse("2016-07-04"));

                Set<Project> projects = new HashSet<>();
                projects.add(mars);
                projects.add(sba);

                employee.setProjects(projects);

                hsession.save(employee);

                tx.commit();
                hsession.close();
                sf.close();
            } catch (ParseException e) {
                e.printStackTrace();
            }
        }
}
```

Fetching Strategies in Hibernate

The fetch type essentially decides whether or not to load all of the relationships of a particular object/table as soon as the object/table is initially fetched.

There are two types of the fetching strategies in the hibernate.

1. Lazy Fetch type – Load the child entities when needed
2. Eager Fetch type – Load all child entities immediately when parent object is loaded

Lazy Fetch Type

Hibernate defaults to a lazy fetching strategy for all entities and collections. Suppose you have a parent and that parent has a collection of children. Now hibernate can lazy load these children which means that hibernate does not load all the children while loading the parent. Instead it loads children only when it is requested to do so. It prevents a huge load since entity is loaded only once they are required by the program. Hence it increase the performance.

Syntax :

```
@OneToMany(mappedBy = "author", fetch = FetchType.LAZY)
private Set<Book> books = new HashSet<>();
```

Eager Fetch Type

In hibernate 2 this is default behavior of the to retrieving an object from the database.

Join Fetch strategy the eager fetching of associations. The purpose of Join Fetch strategy is optimization in terms of time. I mean even associations are fetched right at the time of fetching parent object. So in this case we don't make database call again and again . So this will be much

faster. Agreed that this will bad if we are fetching too many objects in a session because we can get java heap error.

Syntax :

```
@OneToMany(mappedBy = "author", fetch = FetchType.EAGER)
private Set<Book> books = new HashSet<>();
```

Cascade Types in Hibernate

Cascading means that if we do any operation on an object it acts on the related objects as well. If you insert, update or delete an object the related objects (mapped objects) are inserted, updated or deleted.

Hibernate does not navigate to object associations when you are performing an operation on the object. We have to do the operations like save, update or delete on each and every object explicitly. To overcome this problem Hibernate is providing Cascade Types.

To enable cascading for related association, the cascade attribute has to be configured for association in mapping metadata.

There are following types of cascade:

CascadeType.PERSIST : means that save() or persist() operations cascade to related entities.

CascadeType.MERGE : means that related entities are merged into managed state when the owning entity is merged.

CascadeType.REFRESH : does the same thing for the refresh() operation.

CascadeType.REMOVE : removes all related entities association with this setting when the owning entity is deleted.

CascadeType.DETACH : detaches all related entities if a “manual detach” occurs.

CascadeType.ALL : is shorthand for all of the above cascade operations.

The cascade configuration option accepts an array of CascadeTypes; thus, to include only refreshes and merges in the cascade operation for a One-to-Many relationship as in our example, you might see the following:

```
@OneToMany(cascade={CascadeType.REFRESH, CascadeType.MERGE}, fetch =
FetchType.LAZY)
@JoinColumn(name="AUTHOR_ID")

private Set<Book> books;
```

Working with Blob and Clob in Hibernate

Sometimes, Our data is not limited to strings and numbers. We need to store a large amount of data in Database table like documents, raw files, XML documents and photos etc. To store these files or photos our table column datatype should be BLOB or CLOB.

To Support for BLOB or CLOB hibernate provided @Lob annotation

@Lob saves the data in BLOB or CLOB

CLOB(Character Large Object): If data is text and is not enough to save in VARCHAR, then that data should be saved in CLOB.

BLOB(Binary Large Object): In case of double byte character large data is saved in BLOB data type.

In case of Character[],char[] and String data is saved in CLOB. And the data type Byte[], byte[] will be stored in BLOB.

Example

==========Employee.java==========

```
import java.util.Date;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Lob;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.Version;

@Entity
@Table(name = "EMPLOYEE")
public class Employee {

	@Id
	@GeneratedValue
	@Column(name = "EMP_ID")
	private Integer empId;

	@Column(name = "EMP_NAME")
	private String empName;

	@Lob
	@Column(name = "EMP_IMAGE")
	private byte[] empPhoto;

	@Version
	@Temporal(TemporalType.TIMESTAMP)

	private Date update_dt;

}
```

==========ImageDemo.java==========

```
public class ImageDemo {

	public static void main(String[] args) throws Exception {
		insertImage();
		readImage();
	}

	public static void readImage() throws Exception {

		SessionFactory sf = HibernateUtil.getSessionFactory();
		Session hsession = sf.openSession();
		Transaction tx = hsession.beginTransaction();

		Employee emp = hsession.get(Employee.class, 1);

		FileOutputStream fos = new FileOutputStream("image2.jpeg");
		fos.write(emp.getEmpPhoto());
		fos.flush();
		fos.close();

		tx.commit();
		hsession.close();
		sf.close();

	}

	public static void insertImage() throws Exception {

		SessionFactory sf = HibernateUtil.getSessionFactory();
		Session hsession = sf.openSession();
		Transaction tx = hsession.beginTransaction();

		File f = new File("image1.jpeg");
		byte[] brr = new byte[(int) f.length()];

		FileInputStream fis = new FileInputStream(f);
		fis.read(brr);

		Employee emp = new Employee();
		emp.setEmpName("Ashok");
		emp.setEmpPhoto(brr);

		hsession.save(emp);

		tx.commit();
		hsession.close();
		sf.close();


	}
```

==========

Versioning and Timestamping in Hibernate

Let's assume that, two users are working on a project and they both are currently viewing a bug or an enhancement in a project management application. Let's say one user knows that it is a duplicate bug while the other does not know and thinks that this needs to be done. Now one user changes its status as **IN PROGRESS** and the other user marks it as **DUPLICATE** at the same time but the request with status as **DUPLICATE** goes a bit early to the server and the request with status IN PROGRESS arrives later. What will be the current status?

It will be IN PROGRESS as the later request will overwrite the previous status. These kind of scenarios are common when multiple users work on same application and should be prevented.

In general, following is the scenario which should be prevented

- Two transactions read a record at the same time.
- One transaction updates the record with its value.
- Second transaction, not aware of this change, updates the record according to its value.

End Result is, the update of first transaction is completely lost.

Solution : Hibernate has a provision of version based control to avoid such kind of scenarios. Under this strategy, a version column is used to keep track of record updates. This version may either be a timestamp or a number.

If it is a number, Hibernate automatically assigns a version number to an inserted record and increments it every time the record is modified.

If it is a timestamp, Hibernate automatically assigns the current timestamp at which the record was inserted / updated.

Hibernate keeps track of the latest version for every record (row) in the database and appends this version to the where condition used to update the record. When it finds that the entity record being updated belongs to the older version, it issues an **org.hibernate.StaleObjectStateException** and the wrong update will be cancelled.

How to Implement Versioning??

There are only three steps required for versioning to work for an entity. They are :

1. Add a column with any name (usually it is named as Version) in the database table of the entity. Set its type either to a number data type (int, long etc.) or a timestamp as you want to store its value.

2. Add a field of the corresponding type to entity class. If we have made the version column in the database table of a number type , then this field should be either int, long etc. and if the column type in the database is of type timestamp, then this field should be a java.sql.Timestamp.

3. Annotate the above field in your entity class with @Version or provide the definition of version column in the <class> tag if you are using XML based entity definitions.

Example

==========Defect.java==========

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

@Entity
@Table(name = "Defects")
public class Address {
	@Id
	@GeneratedValue
	@Column(name=”Id”)
	private Integer id;
	@Column(name = "Type")
	private String type;
	@Column(name = "Description")
	private String description;
	@Column(name = "Status")
	private String status;
	@Version
	private Integer version;

	// getter and setter methods
}
```

==========VersionInsertDemo.java==========

```
public class VersionInsertDemo {

	public static void main(String[] args) throws Exception {

		SessionFactory sf = HibernateUtil.getSessionFactory();
		Session hsession = sf.openSession();
		Transaction tx = hsession.beginTransaction();

		Defect defect = new Defect();
		defect.setType("Bug");
		defect.setDescription("Core defect");
		defect.setStatus("DUPLICATE");
		//We did not set the version field value

		hsession.save(defect);

		tx.commit();
		hsession.close();
		sf.close();
	}
}
```

==========

Executing the above code will generate a database record as shown below

	Id	Type	Description	Status	Version
▶	1	Bug	Core defect	DUPLICATE	0

The value 0 in the version column is automatically generated by Hibernate. **Note that we did not set version in the above code**. Below is the query generated by Hibernate while saving the record.

```
Hibernate: insert into defect(Description, Status, Type, version) values (?, ?, ?, ?)
```

See the field version is included in the insert query generated by Hibernate though we did not set its value.

Now let's make a change in this record and update it using the below code :

==========VersionUpdateDemo.java==========

```
public class VersioningUpdateDemo {

	public static void main(String[] args) throws Exception {

		SessionFactory sf = HibernateUtil.getSessionFactory();
		Session hsession = sf.openSession();
		Transaction tx = hsession.beginTransaction();

		// fetch record with id as 1
		Defect defect = session.get(Ticket.class, 1);
		// change a field value
		defect.setStatus("CLOSED");
		// We did not set the version field value again
		session.update(defect);

		tx.commit();
		hsession.close();
		sf.close();
	}
}
```

==========

Executing the above code will generate a database record as shown below

	Id	Type	Description	Status	Version
▶	1	Bug	Core defect	CLOSED	1

The value 1 in the version column is automatically generated by Hibernate. **Note that we did not update version in the above code**.

How versioning will solve the above scenario?

Two Users are viewing at a defect ticket at the same time. We will call them User1 and User2. Let's say the defect is a new entry in database table. Post versioning implemented, it will have version as 0. User1 and User2 change the status of a bug at the same time but the request of User1 arrives a bit early to the server. User1 has changed the status to DUPLICATE, the query issued is :

update defect set Description=?, Status=DUPLICATE, Type=?, version=1 where id=? and version=0

Now the request of User2 to change the status to IN PROGRESS arrives, the query issued will be :

update defect set Description=?, Status=IN PROGRESS, Type=?, version=1 where id=? and version=0

Note the version in the WHERE clause is 0 since both the users were looking at version 0 of the defect. Now there is NO record matching the id of that defect and version since the version has been changed by the previous update. Hibernate is smart enough to detect an object mismatch and issues

a org.hibernate.StaleObjectStateException thus preventing the update on an already updated record.

- If the type of version column in database and the field type in java class are set to timestamp, then every time a record is updated, the current timestamp is set as the version value.
- When timestamp is set as the type of value for version column, then its value can indicate the time at which the entity was updated.
- It is not mandatory to name the java field representing version as version itself.

Timestamping

Storing the creation timestamp or the timestamp of the last update is a common requirement for modern applications. It sounds like a simple requirement, but for a huge application, we don't want to set a new update timestamp in every use case that changes the entity.

We need a simple, fail-safe solution that automatically updates the timestamp for each and every change. As so often, there are multiple ways to achieve that:

- We can use a database update trigger that performs the change on a database level. Most DBAs will suggest this approach because it's easy to implement on a database level. But Hibernate needs to perform an additional query to retrieve the generated values from the database.
- We can use an entity lifecycle event to update the timestamp attribute of the entity before Hibernate performs the update.
- We can use an additional framework, like Hibernate Envers, to write an audit log and get the update timestamp from there.
- We can use the Hibernate-specific **@CreationTimestamp** and **@UpdateTimestamp** annotations and let Hibernate trigger the required updates.

It's obvious that the last option is the easiest one to implement if we can use Hibernate-specific features. So let's have a more detailed look at it.

@CreationTimestamp and *@UpdateTimestamp*

Hibernate's *@CreationTimestamp* and *@UpdateTimestamp* annotations make it easy to track the timestamp of the creation and last update of an entity.

When a new entity gets persisted, Hibernate gets the current timestamp from the VM and sets it as the value of the attribute annotated with *@CreationTimestamp*. After that, Hibernate will not change the value of this attribute.

The value of the attribute annotated with *@UpdateTimestamp* gets changed in a similar way with every SQL Update statement. Hibernate gets the current timestamp from the VM and sets it as the update timestamp on the SQL Update statement.

Supported attribute types

We can use the *@CreationTimestamp* and *@UpdateTimestamp* with the following attribute types:

- *java.time.LocalDate* (since Hibernate 5.2.3)
- *java.time.LocalDateTime* (since Hibernate 5.2.3)
- *java.util.Date*
- *java.util.Calendar*
- *java.sql.Date*
- *java.sql.Time*
- *java.sql.Timestamp*

Example

==========Employee.java==========

```
import java.time.LocalDateTime;

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

import org.hibernate.annotations.CreationTimestamp;
import org.hibernate.annotations.UpdateTimestamp;

@Entity
@Table(name = "EMPLOYEE")
public class Employee {

	@Id
	@GeneratedValue
	@Column(name = "EMP_ID", updatable = false, nullable = false)
	private Long empId;

	@Column(name = "EMP_NAME")
	private String empName;

	@CreationTimestamp
	@Column(name = "CREATE_DT")
	private LocalDateTime createDateTime;

	@UpdateTimestamp
	@Column(name = "UPDATE_DT")
	private LocalDateTime updateDateTime;

	// setters and getters


}
```

==========

When we persist a new *Employee*, Hibernate will get the current time from the VM and store it as the creation and update timestamp.

Hibernate Validator framework

A good data validation strategy is an important part of every application development project. Being able to consolidate and generalize validation using a proven framework can significantly improve the reliability of your software, especially over time

Hibernate Validator provides a solid foundation for building lightweight, flexible validation code for Java SE and Java EE applications. Hibernate Validator is supported by a number of popular frameworks, but its libraries can also be used in a standalone implementation. Standalone Java SE validation components can become an integral part of any complex heterogeneous server-side application. In order to follow this introduction to using Hibernate Validator to build a standalone component, you will need to have JDK 6 or higher installed. All use cases in the article are built using Validator version 5.0.3. You should download the Hibernate Validator 5.0.x binary distribution package, where directory \hibernate-validator-5.0.x.Final\dist contains all the binaries required for standalone implementation.

- **@Size** : This annotation is used to set the size of the field. It has three properties to configure, the `min`, `max` and the `message` to be set.
- **@Min** : This annotation is used to set the min size of a field
- **@NotNull** : With this annotation you can make sure that the field has a value.
- **@Length** : This annotation is similar to **@Size**.
- **@Pattern** : This annotation can be used when we want to chack a field against a regular expression. The `regex` is set as an attribute to the annotation.
- **@Range** : This annotation can be used to set a range of min and max values to a field.

```
============================Form.java=========================================
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;
import org.hibernate.validator.constraints.Length;
import org.hibernate.validator.constraints.Range;

public class Form {

	@Size(min = 5, max = 10, message = "Your name should be between 5 - 10
characters.")
	private String name;
	@Min(value = 5, message = "Please insert at least 5 characters")
	private String lastname;
	@NotNull(message = "Please select a password")
	@Length(min = 5, max = 10, message = "Password should be between 5 - 10
characters")
	private String password;
	@Pattern(regexp = "[0-9]+", message = "Wrong zip!")
	private String zip;
	@Pattern(regexp = ".+@.+\\..+", message = "Wrong email!")
	private String email;
	@Range(min = 18, message = "You cannot subscribe if you are under 18 years
old.")
	private String age;

	// Setters and getters
}

==================================000=======================================
```