⇒ Hibernate Basic

Preledge

Hibernate framework is an open-source Java-based framework. Hibernate framework shields developers from 'Messy' SQL and allows the developer to concentrate and work with the object model. It is a tool for object-relational mapping. The technique for mapping a model in an object-oriented domain to a relational type of database is ORM, Object Relational Mapping. Hibernate is free software.

Data Persistance

↳ means phenomenon of making a data permenant to the storage system

↳ Implimenting a proper data persistance strategy will make the applicatn scalable, mentainable and efficient

↳ Persistance phenomenon consists

- ↳ data (what to persist)
- ↳ medium (How to persist)
- ↳ storage (where to persist)

Disadvantages :-

Java I/O APIs pretty much covers all the functionalities as a data persistence medium.

- But working with the File system is very difficult and inefficient in handling large and complex data.
- And using Java I/O also need lower-level details of the data to be retrieved, stored, or manipulated.

Serialization too has its own disadvantages:

- Since storing and retrieval of the entire object graph is done at once, it is not a suitable approach while working with a large amount of data.
- Concurrent access is not possible.
- It provides no query capabilities.
- The data cannot be retrieved without de-serialization.

The **data** that needs to be persisted can be:

- Raw data: which is collected from a file or any other source in the form of bytes.
- Java object: which is the data contained in the object of a Java class.

To persist the data, Java provides **mediums** like:

- I/O Streams and Serialization
- JDBC
- ORM Frameworks like Hibernate

1) Java Input Output

The Java Input-Output(I/O) API provides classes for performing input and output operations on raw data.

- These classes are available in the java.io package.
- Java I/O API is built on four abstract classes. This depends upon the type of data it can handle (byte/character).
  - InputStream and OutputStream: deals with bytes.
  - Reader and Writer: deals with character.

2) Serialization

Serialization helps in sending Java objects through the network and this also can be used to store these Java objects in a file.

- An object can be marked serializable by implementing the java.io.Serializable interface.
- Serializable objects can be converted into a stream of bytes.
- This stream of bytes can be written into a file.
- These bytes can be read back to re-create the object.
- Deserialization is the process of retrieving an object from the byte streams.

JDBC

JDBC or 'Java Database Connectivity' is a Java Core API for performing database interaction.

- Using JDBC API, a Java application can access a variety of databases such as MySQL, Oracle, etc.
- JDBC follows a relational database-oriented approach to work with the data using SQL queries.

Disadvantages :-

The problem with Serialization is solved by JDBC, but it does not store the Java objects directly. The data from the objects need to be converted into a SQL query and then executed, for persistence.

- SQL code has to be embedded within Java Programs which makes it non-portable.
- JDBC API allows the developer to fire the SQL queries from the Java code. This means the developer needs to know the specific SQL constructs for the Relational Database Management System (RDBMS) used.
- Also, it is the responsibility of the programmer to make sure that the data model and the object model are synchronized properly.

⇒ Object Relational Impedence mismatch

↳ Refer to those technical and conceptual mismatch that arises when we try to map Java object with tables.

↳ The mismatches that we have are:

- ↳ Granularity: mismatch in no. of classes and no. of tables
- ↳ Inheritance/Subtype: Inheritance not available in RDBMS.
- ↳ Associations: In object model, we can have reference variable of a class inside another class and In relational model we have foregin keys
- ↳ Identity: In object model, we use '==' or .equals() and in relational model, we have primary keys.
- ↳ Data Navigation: we use (.) dot operator but we use Joins

⇒ Object Relational mapping (ORM)

// trys to resolve impedance mismatches

- ↳ ORM resolves ORIM
- ↳ ORM handles lower level interaction with database
- ↳ ORM helps to get rid of "Messy SQL"
- ↳ ORM helps developers to work on business logic and object model.
- ↳ ORM is database independent and that's why portable as well
- ↳ ORM is preffered elegent persistance solution for enterprise applications.

⇒ Introduction to Hibernate framework

// pure java persistance framework that supports ORM

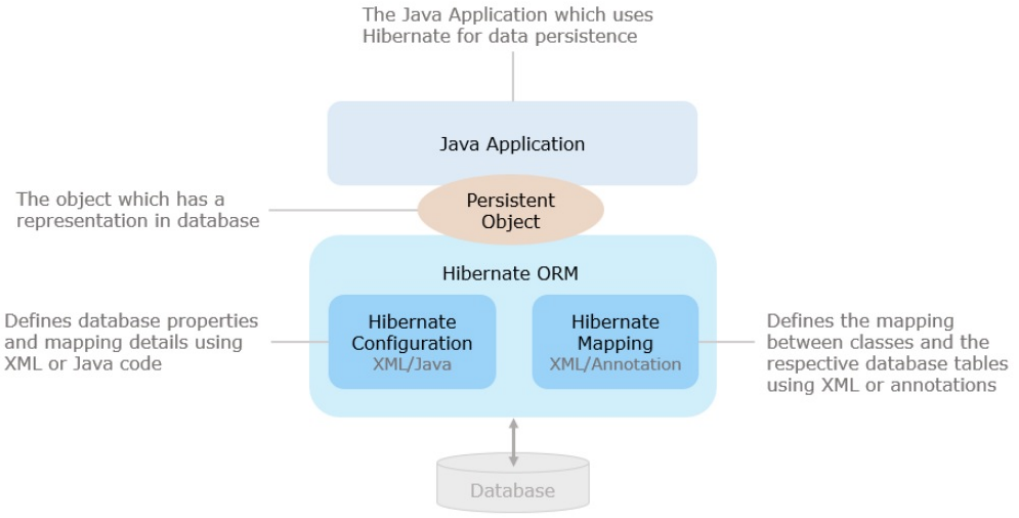
- Hibernate provides an implementation for JPA Specification.
- Hibernate is a powerful ORM solution that maps user-defined Java classes to DB tables.
- Hibernate has a strong query language which is called Hibernate Query Language. It supports native SQL as well.
- Hibernate reduces the number of lines in the code by keeping object-table mapping itself and gives the result to an application as Java objects. It ensures the programmer doesn't have to manually handle persistent data, this way reducing the time of development and cost of maintenance.
- Hibernate uses SQL based schema for mapping object model to the relational model.
- Hibernate allows developers to emphasize the domain model and not on the persistence plumbing (e.g.: connection management).

// features
- ↳ Object Relational mapping
- ↳ Scalability & Reliability (stability & quality)
- ↳ Extensible
- ↳ High Performance (due to fetch strategy, optimistic locking, caching etc)
- ↳ Idiomatic Persistance

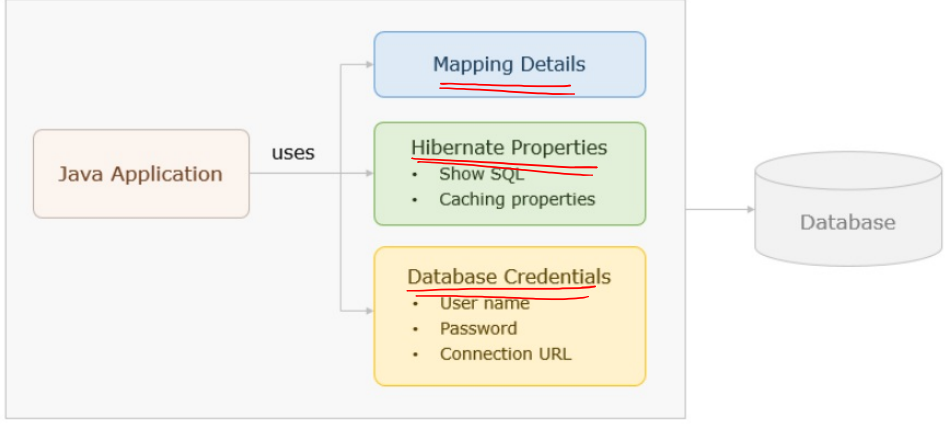
→ Benifits
- ↳ Light weight
- ↳ Open source
- ↳ vendor independent
- ↳ non-invasive
  (doesn't force to impliment or extends a class)

⇒ Hibernate Architecture

High level view



→ To retrieve data from database using hibernate, these are 3 components are needed :-



→ There are 2 ways to provide hibernate mapping

↳ Using XML

- The external hbm.xml file is generated.
- HBM stands for Hibernate Mapping.
- The mapping file by convention is named <EntityClassName>.hbm.xml .

↳ Using Annotations

- Use annotations in the entity class.
- Annotations are available in javax.persistence package.

1) using XML :-

```
<hibernate-mapping>
    <class name="com.infybank.Customer" table="CUSTOMER">
        <id name="customerId" type="java.lang.Integer" column="ID" >
        </id>
        <property name="customerName" type="java.lang.String" column="NAME"/>
        <property name="dateOfBirth" type="java.util.Calendar" column="DOB"/>
    </class>
</hibernate-mapping>
```

Mapping Customer class to CUSTOMER table

Represents primary key column

Customer class attributes are mapped to columns of the CUSTOMER table

2) using annotation

```
// Add necessary import statements
@Entity
@Table(name="CUSTOMER")
public class Customer {
    @Id
    @Column(name="ID")
    private Integer customerId;

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

    @Temporal(TemporalType.DATE)
    @Column(name="DOB")
    private Calendar dateOfBirth;

    @Transient
    private Integer age;

    // getter and setter methods
}
```

@Entity for mapping this entity class with the database table

@Table to specify the table name explicitly as "customer".
If this annotation is not mentioned, by default the table name is same as the class name (Customer)

@Id to specify customerId property as the primary key of the table

@Column to specify the column name as "name" for the "customerName" property.
If this annotation is not used, by default the column name in the table will be same as the property name (customerName)

@Temporal to convert the date from Java object to compatible database type

@Transient to specify that these values are never stored in the database

↳ hibernate configuration

Hibernate has to know prior as to where it can find the information of the mappings that is present between the Java classes and the related database tables. Hibernate would also need to know the information related to the database and its related parameters. This can be provided using the **Hibernate Configuration.**

Hibernate Configuration has the below information.

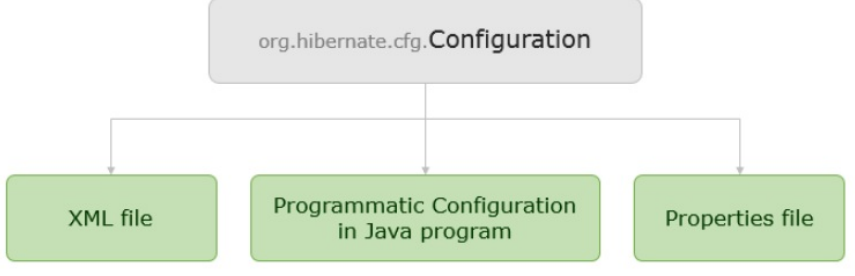
- Database connection settings
- Hibernate properties
- Mapping Resource details

CONFIGURATION PROPERTIES	DEFINITION
hibernate.connection.drive_class	The database driver class
hibernate.connection.url	The URL of the database instance
hibernate.connection.username	The database username
hibernate.connection.password	The database password
mapping class	Name of the entity class
mapping resource	Name of the .hbm.xml file

gmp

CONFIGURATION PROPERTIES	DEFINITION
hibernate.show_sql	Displays the SQL statements in the console
hibernate.connection.provider_class	The class name that implements Hibernate's ConnectionProvider interface
hibernate.cache.provider_class	Specifies the class name that provides CacheProvider interface
hibernate.hbm2ddl.auto	Automatically creates the schema, updates it, or on startup or shutdown, it will drop the schema. The acceptable values are create, update, create-drop
hibernate.cache.use_second_level_cache	Specifies whether the second-level cache can be used or not. It accepts the values as true or false
hibernate.connection.pool_size	Specifies the maximum number of pooled connections

↳ There are 3 ways to provide configuration details :-



↳ Types

1) XML file

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

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

        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/hibernatedb
        </property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">root</property>

        <property name="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect
        </property>

        <mapping class="com.infybank.Customer "/>
    </session-factory>
</hibernate-configuration>
```

Doctype for Hibernate Configuration

These are database connection properties

dialect property specify the name of the SQL dialect to be used for generating SQL statements

class attribute is used, if mapping details are through annotations. **resource** attribute should be used, if the mapping details are through ".hbm.xml" file as <mapping resource="Customer.hbm.xml"/>

2) Programatic configuration

```
Configuration configuration = new Configuration();

configuration.setProperty("hibernate.connection.driver_class",
        "com.mysql.jdbc.Driver ");
configuration.setProperty("hibernate.connection.url",
        "jdbc:mysql://localhost:3306/hibernatedb");
configuration.setProperty("hibernate.connection.username","root");
configuration.setProperty("hibernate.connection.password","root");

configuration.setProperty("hibernate.dialect",
        "org.hibernate.dialect.MySQL5Dialect");

configuration.addAnnotatedClass(com.infybank.Customer.class);

    //  Or

configuration.addResource("Customer.hbm.xml");
```

Creating an instance of **Configuration** class

Database Connection properties

hibernate.dialect property specifying SQL dialect to be used for generating SQL statements

addAnnotatedClass() is used if the mapping is provided through annotation

addResource() is used if the mapping is defined through XML file

3) hibernate.properties file

```
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost:3306/hibernatedb
hibernate.connection.username=root
hibernate.connection.password=root

hibernate.dialect=org.hibernate.dialect.MySQL5Dialect

....
```

Database Connection properties

hibernate.dialect property specifying SQL dialect to be used for generating SQL statements

```
Configuration configuration = new Configuration();

configuration.addResource("Customer.hbm.xml");
```

The **properties** file will be automatically loaded by Configuration class

Add this code to provide Hibernate mapping file in the Java application

⇒ Hibernate API's

The following are the important **Hibernate APIs** :

- org.hibernate.SessionFactory
- org.hibernate.Session
- org.hibernate.Transaction
- org.hibernate.connection.ConnectionProvider
- org.hibernate.TransactionFactory

Note :- StandardServiceRegisteryAPI → ServiceRegisteryAPI → SessionFactory → Session → Transaction

Transient object :- is an object which has been initialised using new operator and has no database representation.

Persistence object :- is represented in database as records or primary key value.

1) SessionFactory API :

- SessionFactory API is available in org.hibernate package, is used to create a number of database connection requests.
- Using the SessionFactory object, Hibernate Session objects are created.
- Ideally, there should be one SessionFactory instance defined for each database.

2) Session API:

- The Session API, available in org.hibernate package, is the primary runtime interface between the Java application and the RDBMS.
- A session is a single-threaded, transient object which denotes a conversational state between the persistence store and application.
- Session manages and provides the methods to interact with the Persistence Context.
- SessionFactory creates session objects and it contains a first-level cache to store persistent objects.

3) ConnectionProvider and TransactionFactory APIs are not visible to the Java application, but a developer can extend or implement them as per the requirements.

ConnectionProvider API:

- A factory and a pool of many JDBC connections.
- Available in org.hibernate.connection package.
- This is an abstraction of the application from the DataSource or DriverManager.

TransactionFactory API:

- Available in org.hibernate package.
- This represents a factory of Transaction instances.

4.) Transaction API:

- A Transaction is a single-threaded, transient object to specify one single unit of work.
- Transaction API is available in org.hibernate package.
- A session might make several transaction instances whenever required.
- It creates an abstraction of the application from the JDBC, JTA, or CORBA transaction.
- Session API requires Transaction API for implementing CRUD (Create, Read, Update, Delete) operations.

methods:-
- transaction.begin()
- transaction.commit()
- transaction.rollback()

5.) ServiceRegistry API:

- The main purpose of a service registry is to hold, manage, and provide access to services.
- This is the central service API.
- In Hibernate this API helps in managing and providing access to standard services like DialectFactory and ConnectionProvider.
- This is available in the org.hibernate.service package.

6.) StandardServiceRegistryBuilder API:

- This class is required to build standard ServiceRegistry instances.
- In Hibernate this API is used to create ServiceRegistry API instances.
- This is available org.hibernate.boot.registry.

Ex:- Steps for a Java application in order to connect to the database using Hibernate.

```
//add necessary import statements
class HibernateDemo{
        public static void main(String args[]){

Configuration configuration = new Configuration().configure();

ServiceRegistry serviceRegistry = new ServiceRegistryBuilder()
.applySettings(configuration.getProperties())
.buildServiceRegistry();

SessionFactory factory = configuration
.buildSessionFactory(serviceRegistry);

//Creating a session instance by using SessionFactory API
Session session = factory.openSession();
//Beginning a transaction
Transaction tx = session.beginTransaction();
//…
//Implement any of the database operation like reading a record
//  …
tx.commit();
    }
}
```

- Loads hibernate configurations and related mappings
- Configuration instance is loaded with the properties to instantiate SessionFactory
- ServiceRegistryBuilder API is used to create the instance of ServiceRegistry using the configuration properties
- Creating an instance of **SessionFactory** API using **buildSessionFactory()** method of Configuration API
- Creating an instance of **Session** from **SessionFactory** API
- Creating an instance of **Transaction** from **Session** API
- **commit()** Used to save the changes in the database table

⇒ CRUD operations using Hibernate

- To create/update :- save(), persist(), update(), saveOrUpdate()
- To Read :- get(), load(), refresh()
- To delete :- delete()

1)
- Serializable save (Object object) → does return identifier
  - Persists the entity object to the database table.
  - The primary key value of the persisted record is returned.
- void persist (Object object) → doesn't return the identifier
  - Persists the entity object to the database.

2) • Object get(Class , Serializable idValue) → return null if not found the object

An instance of the entity class can be retrieved with the given primary key value. If no such record is present in the database table or persistence context, the method returns **NULL** value.

• Object load(Class class, Serializable idValue) → return ObjectNotFoundException if not found the object

Retrieves a proxy instance of the entity class with the given primary key value. If the requested row is unavailable in the database table, **ObjectNotFoundException** is thrown. When using load(), Hibernate does not hit the database but initializes the entity object by doing **lazy loading**(defer initialization of an object until the point at which it is needed). This enhances performance.

3) In Hibernate, updating records in the database can be achieved in the following way:

- The Entity instance is obtained from the database using get() or load().
- Set the new values for the properties for the retrieved instance.
- Invoke void update(Object object).
- The changes are committed by an active transaction using the commit() method.

4) Records can be removed from the database table with the help of the delete method.

- The entity instance is retrieved from the database using **get() or load()** method.
- Invoke **void delete(Object object)** to remove the corresponding row from the database table.
- The changes are committed by an active transaction using the **commit()** method.

Note :- **void saveOrUpdate(Object object):** saveOrUpdate() method calls either save() or update() on the basis of identifier exists or not in the mapped table. This method internally invokes save(), if the object does not exist. This method internally invokes the update() method, if the object already exists.

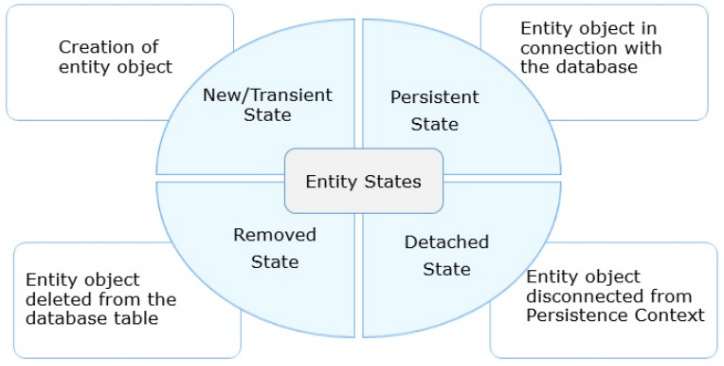
Ex :- Interface Customer with CRUD operations

```
package com.infybank;
public interface ICustomer {
     public void addCustomer(Customer custObj);
     public void readCustomer(int customerId);
     public void updateCustomer(Integer customerId,String customerName);
     public void deleteCustomer(Integer customerId);
}
```

Implimentation of CRUD

```
public class CustomerDAO implements ICustomer {
     /* Method to create a customer record */
     public void addCustomer(Customer c1) {
          Session session = HibernateUtil.getSessionFactory().openSession();
          Transaction tx = session.beginTransaction();
          session.save(c1);
          tx.commit();
          session.close();
     }

     /* Method to read customer details */
     public void readCustomer(int customerId) {
          Session session = HibernateUtil.getSessionFactory().openSession();
          Customer cust = (Customer) session.get(Customer.class, customerId);
          System.out.println("Name : " + cust.getCustomerName());
          session.close();

     }

     /* Method to update a customer record */
     public void updateCustomer(Integer customerId,String name) {
          Session session = HibernateUtil.getSessionFactory().openSession();
          Transaction tx = session.beginTransaction();
          Customer cust = (Customer) session.get(Customer.class, customerId);
          cust.setCustomerName(name);
          session.update(cust);
          tx.commit();
          session.close();

     }
     /* Method to delete a customer from the records */
     public void deleteCustomer(Integer customerId) {

          Session session = HibernateUtil.getSessionFactory().openSession();
          Transaction tx = session.beginTransaction();
          Customer cust = (Customer) session.get(Customer.class, customerId);

          session.delete(cust);
          tx.commit();
          session.close();

     }
}
```

⇒ Hibernate Entity State



Hibernate Entity States

New/Transient state : when the entity instance is just been created. (it is not related to persistance context/heap memory)

Persistance State : when entity instance is associated with persistance context and and linked with coorosponding database table.

Following operations brings an entity instance to a **Persistent** state:

- By persisting a new record into the database using **persist() or save()** method.
- By fetching an entity record from the database using **get() or load()** method.
- By updating the entity record in the database using **saveOrUpdate() , update()** or **merge()** method(explained later).

Note :- to check if an entity instance is in persistance state

```
Boolean b = session.contains(entity);
```

b = TRUE	b = FALSE
entity is in persistent state	entity is not in persistent state

A Persistance context is a cache which remembers all the modification and state change made to object in a particular unit of work

```
//Persistent Context 1 Starts                                   // Start of persistence context
Session session1 = HibernateUtil.getSessionFactory().openSession();
Customer customer1 = (Customer)session1.get(Customer.class,1001);
Customer customer2 = (Customer)session1.get(Customer.class,1001);

if(customer1 == customer2) {                                    // yes same object bcz same session
    System.out.println("Objects point to same reference");
}
session1. close();                                              // End of persistence context
//Persistent Context 1 Ends
//Persistent Context 2 Starts                                   // Start of persistence context
Session session2 = HibernateUtil.getSessionFactory().openSession();
Customer customer3 = (Customer)session1.get(Customer.class,1001);

if(customer1 == customer3) {                                    // not same bcz different sessions
    System.out.println("Objects point to same reference");
}
else{
    System.out.println("Objects do not point to same reference");
}
//Persistent Context 2 Ends
session2.close();                                               // End of persistence context
```

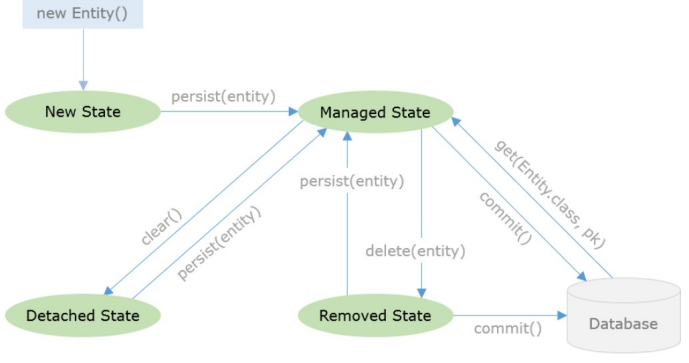
session started → session1 = HibernateUtil.getSessionFactory().openSession();
Check if references are pointing to the same object → if(customer1 == customer2)
session ends → session1. close();
session started again → session2 = HibernateUtil.getSessionFactory().openSession();
Check if references are pointing to the same object → if(customer1 == customer3)
session ends again → session2.close();

Detached State :- when entity is disconnected from persistance context but the coorosponding record is still available in the database.

An entity instance can be in the detached state for the following reasons:

- If an entity is detached specifically from the persistence context using **evict(entity)** method.
- If the entire persistence context is cleared using **clear()** method.
- If the persistence context is closed using **close()** method.

Removed State :- when entity is disconnected from persistence context and coorosponding data is deleted from database as well.



Note :- When an entity instance is removed from the persistence context but the physical object is still present in the heap memory. It will be garbage collected only if referenced to **NULL**.

1) **Object merge (Object object):** merge() can be utilized to update existing values. It takes the detached entity values and updates the persistent entity. This method will copy the state of the given object with the same primary key value as the persistent object.

```
public  void updateCustomer(Integer customerId, String name,Session session ){
     Transaction tx = session.beginTransaction();
     Customer customer = (Customer)session.get(Customer.class, customerId);
     customer.setCustomerName(name);
     //Using merge instead of update()
     Customer cust = session.merge(customer);

     tx.commit();

}
```

Used to update existing values, however this method creates a copy from the passed entity object and return it. The returned object (cust) is part of persistent context and tracked for any changes, where as passed object (Customer) is not tracked

2) Using the **refresh()** method, the latest data available in the database table can be loaded. This is useful especially when we are aware that the database state has been changed since the data was the last read. Refreshing enables the current database state to be taken into the entity instance and the persistence context.

Starting the Transaction

```
session.beginTransaction();
Customer entity = (Customer)session.get(Customer.class, 1002);
```

Reading the record

Hibernate compares values of the Customer table and Customer entity object. If it finds any difference it will update the Customer object with the latest values

```
session.refresh(entity);
```

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

Committing the changes to the database

⇒ Identifier Generation Strategy

It helps to generate a unique identifier value automatically for every record inserted into the database.

Strategy	Description	When to use
Increment	Takes the maximum value from the primary key column and increments it	Used in application where concurrent insertion is not required
Identity	Used the identity column of the underlying database	Used when the underlying database supports identity column, for example, DB2, MySQL, MS SQL Server, Sybase, and Hypersonic SQL
Sequence	Use the Sequence of the underlying database	Used when the underlying database supports sequence, for example, DB2, PostgreSQL, Oracle, and SAP DB
Hilo	User two values Hi and Lo to generate the unique ID. This reduces the frequent access of database for primary key generation	Used in the application that demands a huge number of insert operations
Native	It may internally use Sequence or Identity or Hilo	Used when an application needs portability between different databases
Assigned	Here the application should assign the key value before persisting. This is the default strategy.	Used when the identifier value needs to be assigned explicitly by the application

1) Increment Strategy

Hibernate takes the maximum value of primary key column and increment it

```
@Entity
@GenericGenerator(name = "idgen", strategy = "increment")
public class Customer {
    @Id
    @GeneratedValue(generator = "idgen")
    private Integer customerId;
    private String customerName;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    private String address;
    private Long phoneNo;
    //getters and setters
}
```

- @GenericGenerator(name = "generatorName", strategy = "strategyName")
  - Class-level annotation, to be applied to the entity class **Customer**. This can be used as an attribute level annotation also.
  - generatorName can be any user-defined name. Here "idgen" is used as the generator name.
  - strategyName specifies one of the primary key generation strategy types. Here it is an "increment" strategy.
- @GeneratedValue(generator = "generatorName")
  - Field level annotation, to be applied for the primary key attribute **customerId**.
  - generatorName should be the same as the name attribute defined in @GenericGenerator, which is "idgen" in the code snippet.

2.) Identity Strategy

In Identity strategy, Hibernate uses the identity column of the underlying database. An integer column having a primary key or unique constraint with the AUTO_INCREMENT feature is known as the IDENTITY column of a table. This strategy is database dependent, for example, it works on MySQL but not in Oracle.

```
@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer customerId;
    private String customerName;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    private String address;
    private Long phoneNo;
    //getters and setters
}
```

- @GenericGenerator is not used for this strategy as the Hibernate depends on the underlying database to generate the value.
- @GeneratedValue defines the strategy type as IDENTITY.

3) Assigned Strategy

In some scenerios, when identifier wants to provide the id value.

```
@Entity
@Table(name="customer")
@GenericGenerator(name = "generatorName",strategy = "assigned")
public class Customer{
    @Id
    @GeneratedValue(generator = "generatorName")
    private Integer customerId;
    private String customerName;
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    private String address;
    private Long phoneNo;
        //getters and setters
}
```

⇒ Association Mapping

To provide locker facility we provide association mapping

```java
@Entity
public class Customer {
  @Id
  private Integer customerId;
  private String customerName;
  @Temporal(TemporalType.DATE)
  private Calendar dateOfBirth;
  private String address;
  private Long phoneNo;
  @OneToOne(cascade = CascadeType.ALL)
  @JoinColumn(name = "lockerId", unique = true)
  private Locker locker;
  // constructors
  // getters and setters
}
```

```java
@Entity
@Table(name="locker")
public class Locker {
    @Id
    private String lockerId;
    private String lockerType;
    private Double rent;
    // constuctors
    // getters and setters
}
```

Note:-

Some of the possible CRUD operations which you can perform on the associated entities (Customer.java and Locker.java) are given below:

- Persisting source and target entity instances together - Adding a new customer with the details of the new locker allocated.
- Persisting only source entity instance - Adding a new customer without any locker details.
- Associating an existing source entity instance with a new target entity instance - Allocating a new locker to an existing customer.
- Fetching source and target entity instances together - Retrieving the details of a customer along with the corresponding locker details if available.
- Removing an existing source entity instance along with the associated target entity instance - Deleting the details of an existing customer and the allocated locker details from the database tables.
- Removing only the source entity instance - Deleting the details of a customer while retaining the corresponding locker details.

1.) The foreign key column (LOCKERID) of the CUSTOMER table references the primary key column of the LOCKER table, so the Customer entity can be the *source* and the Locker entity can be the *target*. To map the foreign key column, we need to associate source entity with the target entity i.e. the Customer class has a reference of Locker.

2.)

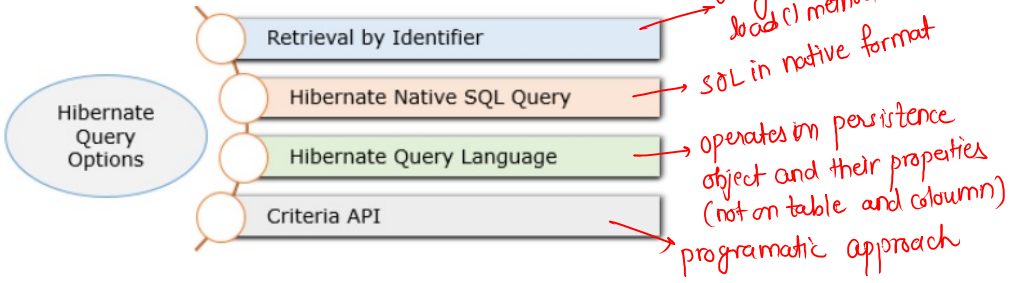
- @OneToOne(cascade = CascadeType.ALL)

This annotation is applied to the reference attribute of Locker in the Customer entity which indicates that the relationship has one to one cardinality. The **cascade** attribute of the annotation is mandatory. This attribute transfers operation (such as insert, update, delete) done on the Customer object to its Locker object.

- @JoinColumn(name = "lockerId", unique = true)

The annotation has two attributes, **name** attribute specifies the name of the foreign key column in the Customer table and the **unique** attribute assures unique values in the foreign key column to achieve one to one mapping.

⇒ Querying Mechanisms in hiberate



using get() and load() method

SQL in native format

operates on persistence object and their properties (not on table and column)

programmatic approach

Query API :- works for HQL
(calling of createQuery())
↳ methods are :-
list() , executeUpdate(), uniqueResult()

Running Native SQL queries from Hibernate definitely defeats the purpose of ORM. Hibernate provides **Hibernate Query Language(HQL)** which follows the Object-Oriented paradigm and fulfills ORM specification.

- HQL is very similar to SQL but can be written in fewer words.
- Helps to retrieve data based on non-key properties.
- Applied on persistent objects and their attributes.
- Offers efficient caching and SQL generation strategies.
- Portable as it does not deal with database tables and columns directly.
- HQL uses class name and properties instead of table name and column.
- Understands Object Oriented concepts like - inheritance, polymorphism, association, aggregation, and composition.

SQL Query API :- works for Native SQL Query mechanism
(calling of createSQLQuery())

disadvantages

- It might cause portability issues.
- No Object Oriented feature is used.

example

When createSQLQuery() called, it returns SQLQuery object

Passing sql query as a parameter of createSQLQuery()

```
SQLQuery query = session.createSQLQuery("select d.customerId,d.customerName from customer d where d.age>50");

List<Customer> list= query.list();
```

Return the query results as a List

Database table name (customer) and column names(customerId,customerName) are used for querying

Ex1

When createQuery() is called,it returns Query object

```
String queryString =
                "from Customer c where c.age >= 50 " );
Query query= session.createQuery(queryString);
List<Customer>list = query.list();
```

Entity class names (eg: Customer) and property names are used for querying

Passing an Object oriented representation of a precompiled query as a parameter of createQuery()

Return the query results as a List

Ex2

```
String queryString = "select c.city from Customer c where c.customerId >= 1002";
Query query = session.createQuery(queryString);
List<String> list = query.list();
```

Ex3

```
String queryString =
       "select c.customerId,c.customerName from Customer c where c.customerId >=  1002 ";
Query query = session.createQuery(queryString);
List<Object[]> custArray = query.list();
```

→ HQL supports Security

Customer name accepted from the user is passed as parameter and assuming that the value passed is "1 = 1"

SQL Injection Attack

```
public void listEmployee(String custName) {

    String badParameter= "'"+ custName +" '";

    Query query=session.createQuery
         ("from Customer where customerName ="+ badParameter +"");

    List<Customer> list = query.list();
}
```

Retrieve the list of customers whose name matches the input passed by the user

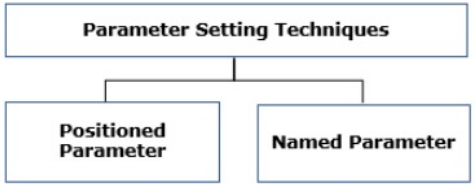
This list will list details of all the Customers

Parameter binding

SQL Injection Attack: In the above query, '1'='1' always evaluates to TRUE, and also the additional "or" condition, causes where clause to always evaluate to TRUE. Hence when the database interprets this query, it retrieves all the records of the Customer table and the attacker gets access to all customer's details.

One of the root cause for the SQL Injection attack is dynamic query creation by concatenating a constant base query string and a user input string.

⇒ Solution is parameter setting / binding



- Positioned Parameter: It uses a question mark (?) to define a named parameter in the query, and according to the position sequence, the value is set to the parameter.
- Named Parameter: It uses a colon, for example :id to define a named parameter in the query.

→ positioned parameter technique (? wali)

Position index starts from 0

```
Query query = session.createQuery
                    ("select c.customerName from Customer c where c.customerName=?");
query.setString(0, "Scott");
Query query = session.createQuery
                    ("select c.customerName from Customer c where c.customerName=? And city=?");
```

Setting the value for the parameter at index 1

```
query.setString(0, "Scott");
query.setString(1, "Mangalore");
```

→ namedparameter technique (: wali)

'name' is the name of named parameter here and it can be any valid identifier

```
Query query = session.createQuery
     ("select c.customerName from Customer c where c.customerName=:name");
query.setParameter("name", "Scott");
```

Note:
1. It is advisable to use only one parameter setting technique for each query.
2. Named parameters are preferred as it is easier to maintain and use

Ex1

```
@NamedQuery(name="Customer.retrieve",
            query = "from Customer c where c.customerName=:name")
```

Convention : **ClassName.operation_name**

Ex2

```
@NamedQueries({
 @NamedQuery(name="Customer.update", query = "update Customer set city='Mysore'
                                              where customerName='Brijesh'"),
 @NamedQuery(name="Customer.retrieve", query = "from Customer c where
                                                c.customerName='Scott' ")
 })
```

If more than one queries are needed to be managed as named query for the same entity instance, then the preferred way for data retrieval is having **"Named Queries"**

→ for multiple queries

When an application uses the same query at many places it's really tiresome to manage and write the same query again and again. It is an overhead for the application to compile and create the execution plan of the same query again and again. Hibernate ORM gives a solution for this problem in the form of "Named Query".

- Named queries allow managing queries, used repeatedly.
- The query is maintained along with the entity instance.
- Compiled and kept ready with the execution plan.
- Named queries are defined at the entity level using @NamedQuery.
- There can be one or more named queries defined using @NamedQueries.

→ HQL aggregate functions

Aggregate functions supported by Hibernate:

- avg()
- max()
- min()
- sum()
- count()
- count(*)

While using aggregate functions following points are worth remembering:

- The queries return a unique result.
- The result should be typecasted to the proper data type.

Ex:

```
Integer max = null;
Query query = session.createQuery
                    ("select max(c.points) from Customer c") ;
List<Integer> list = query.list();
max = list.get(0);
```

Retrieve the maximum of points from Customer table

The queries return an unique result

→ SQL clauses in HQL

↳ group by and having clause

ex:

group by

```
Query query = session.createQuery
                    ("select city,count(*) from Customer group by city");
List<Object[]> list=query.list();
```

Retrieve the number of customers for each city

having

```
Query query  = session.createQuery
                    ("select city,count(*) from Customer group by city having count(*)>1 ");
List<Object[]> list=query.list();
```

→ like clause

```
Query query  = session.createQuery("from Customer e where e.customerName like '%Ra%'" );
List<Customer> list=query.list();
```

Retrieve the customer details whose name starts with a pattern using like clause

→ Asc and desc clause

```
Query query  = session.createQuery("select customerId,customerName,city
                                    ,pointsfrom Customer order by customerId desc");
List<Customer> list=query.list();
```

Retrieve the customer details based on the descending order of the customerId

Sorting the Customer in descending order

→ HQL functions.

FUNCTIONS	DEFINITION
length(stringValue)	Computes the number of characters
upper(stringValue)	Converts the letters of string to uppercase
lower(stringValue)	Converts the letters of string to lowercase
concat(stringVal1,stringVal2)	Concatenates two given strings
substring(stringVal1,position)	Breaks the string into two substrings from the given position to the last index
substring(stringVal1,pos1,pos2)	Breaks the string into two substrings from pos1 to pos2

→ Joins

Implicit

JOIN is a strategy to retrieve data by linking between two associated entities.

Two types: Implicit and Explicit.

Implicit join: Does not use JOIN keyword. Associations are de-referenced using "." notation.

Let us observe the Infy Bank Customer and Address POJO class. Here each Customer is linked to one Address.

Ex:

```
Query query = session.createQuery
    ("select c.customerId,c.address.addressId from Customer c where c.address is not null");
List<Object[]> list = query.list();
```

Implicit join for retrieving the customerId and addressId from Customer where customer has a valid address

Explicit

Explicit Join: JOIN keyword is shown explicitly. Type of JOIN strategies supported by Hibernate:

- JOIN or INNER JOIN
- OUTER JOIN
  - LEFT OUTER JOIN
  - RIGHT OUTER JOIN

Ex:

```
Query query  = session.createQuery
    ("select c.customerId,a.addressId from Customer c Inner Join c.address a'");
List<Object[]> list=query.list();
```

Inner Join on Customer and Address