TESTYANTRA

SOFTWARE SOLUTIONS (INDIA) PVT. LTD.

HIBERNATE

EXPERIENTIAL
learning factory

Hibernate

- **What is HIBERNATE ?**

- General meaning of hibernate is “sleep mode” , “log off” or “switch off”.
- Hibernate is an open source persistent framework created by Gavin King in 2001.
- Hibernate is an **O**bject-**R**elational **M**apping (ORM) solution or tool for JAVA.
- Hibernate is a collection of jar files with dedicated functionality.
- It is a powerful, high performance Object-Relational Persistence and Query service for any Java Application.
- Basically hibernate is use to connect with database.
- Hibernate maps Java classes to database tables and from Java data types to SQL data types.

Advantages of Hibernate

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

Disadvantages with JDBC

- JDBC is a DB specific.
- JDBC does not support caching.
- In JDBC it is the responsibility of developer to handle JDBC result set and convert to java object.
- JDBC does not supports “Lazy-Loading”.
- JDBC is a technology.
- In JDBC User is responsible for creation and closing the connections.
- JDBC does not support Association.
- Wont generate PK automatically.

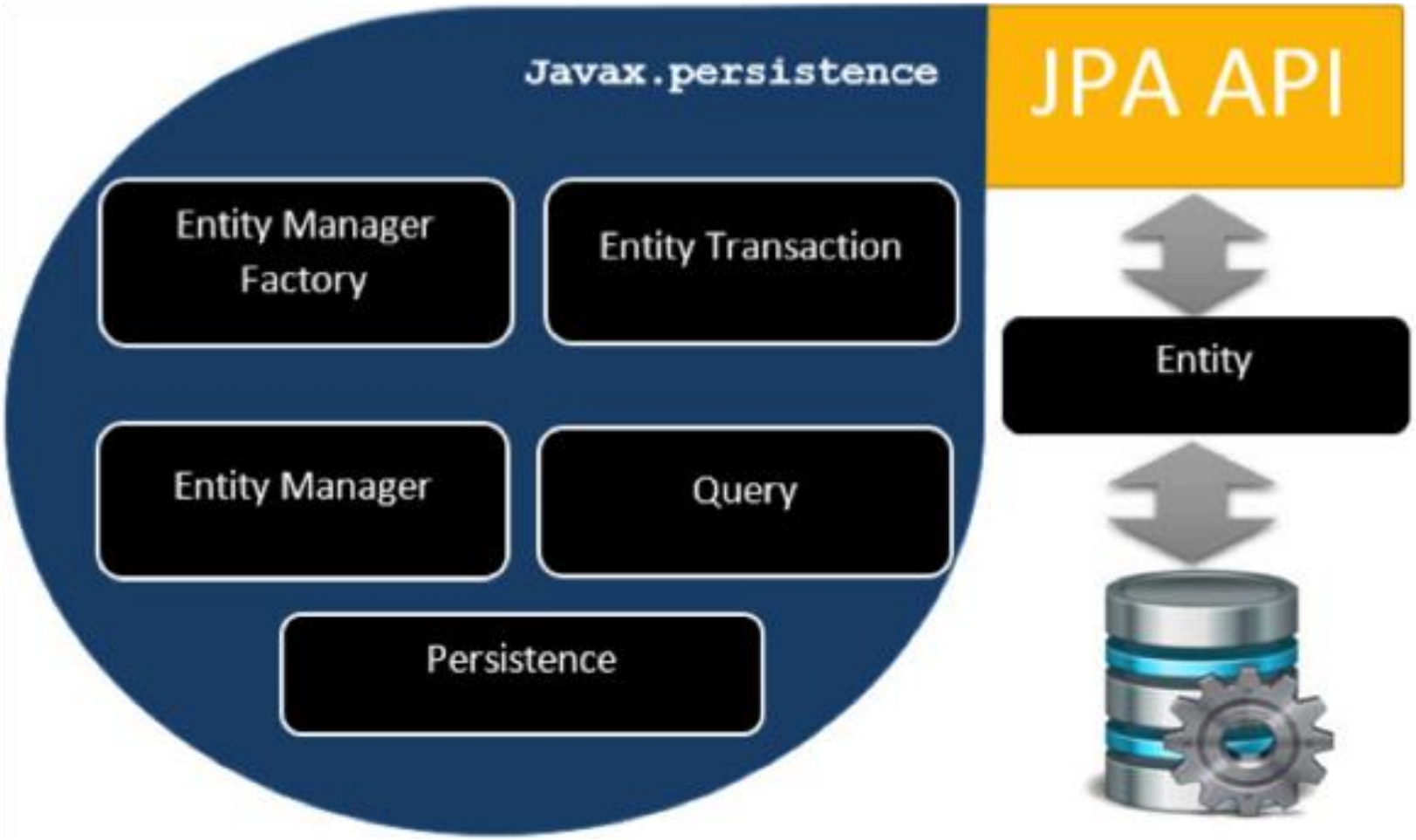
ORM- Object Relational Mapping

- When we work with an object-oriented system, there is a mismatch between the object model and the relational database.
- RDBMS represent data in a tabular format whereas object-oriented languages, such as Java represent it as an interconnected graph of objects.
- ORM stands for **O**bject-**R**elational **M**apping (ORM) .ORM is a kind of persistence s/w , which acts like a bridge b/w java application and databases and transfer to from data in the form of objects.
- ORM internally uses JDBC, for communicating a java application with a database in order to transfer object b/w them.

- **What is JPA ?**

- JPA – Stands for Java Persistence API
- JPA is the Java standard for mapping Java objects to a relational database.
- Mapping Java objects to database tables and vice versa is called Object-relational mapping (ORM).
- The Java Persistence API (JPA) is one possible approach to ORM. Via JPA the developer can map, store, update and retrieve data from relational databases to Java objects and vice versa.
- JPA can be used in Java-EE and Java-SE applications.
- JPA is a specification and several implementations are available. Popular implementations are Hibernate, EclipseLink and Apache OpenJPA.

Hibernate Architecture



Elements Of Hibernate

- For creating the first hibernate application, we must know the elements of Hibernate architecture.

They are as follows:

- EntityManagerFactory
- EntityManager
- EntityTransaction
- Query

Configuration

- Hibernate should know what are the java classes relates the database tables and also set of configuration settings related to database
- All these information is usually supplied as an XML file
- The XML file name always should be **persistence.xml**
- The configuration file contains information about the database and mapping file
- This XML should be in **“src/main/resources/META-INF/”** folder

Hibernate Properties

- **Some Hibernate Properties:**

- javax.persistence.jdbc.driver – Qualified JDBC driver class name
- javax.persistence.jdbc.url – DBURL to the database
- javax.persistence.jdbc.user – username of the database
- javax.persistence.jdbc.password – password of the database
- hibernate.hbm2ddl.auto - This command is use to execute ddl statement.
- org.hibernate.show_sql - to display the sql queries in the console.
- hibernate.dialect – The dialect specifies the type of database used in hibernate so that hibernate generate appropriate type of SQL statements.

persistence.xml


```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
        version="2.1">
        <persistence-unit name="employee-unit">
                <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
                <properties>
                        <property name="javax.persistence.jdbc.driver"
                                value="com.mysql.jdbc.Driver" />
                        <property name="javax.persistence.jdbc.url"
                                value="jdbc:mysql://localhost:3306/db_name" />
                        <property name="javax.persistence.jdbc.user" value="root" />
                        <property name="javax.persistence.jdbc.password"
                                value="root" />
                </properties>
        </persistence-unit>
</persistence>
```

Persistence

- Persistence class is used to load the XML file to JPA
- It is concrete class which has default constructer and many static methods.

public EntityManagerFactory createEntityManagerFactory(String persistenceUnit) ;

This method will search for the given persistenceUnit name in persistence.xml file present in src/main/resources/META-INF/

If it find the persistenceUnit it will load the configuration and create EntityManagerFactory object and return it to the user.

One persistence-unit will be there for each database and hence one EntityMangerFactory

EntityMangerFactory

- Persistence object is used to create a EntityMangerFactory object which in turn configures hibernate for the application using the supplied configuration file and allows for a Session object to be instantiated.
- The EntityMangerFactory is a heavyweight object; it is usually created during application start up and kept for later use.
- EntityMangerFactory object will be created per database using a separate persistence-unit element in persistence.xml file. So, when multiple databases are used, then multiple EntityMangerFactory objects should be created.
- The EntityMangerFactory is a factory class used to get EntityManger object
- EntityMangerFactory is a thread safe.

EntityManager

- The EntityManager object provides an interface between the application and data stored in the database
- EntityManager life cycle is bound to the beginning and end of the transaction
- JPA with Hibernate EntityManager is not a thread safe
- It is factory of EntityTransaction, Query and Criteria
- It holds a first-level cache (mandatory) of data
- The EntityManager interface provides methods to insert, update and delete the object

EntityManager

- A EntityManager is used to get a physical connection with a database.
- The EntityManager object is lightweight and designed to be instantiated each time an interaction is needed with the database.
- Persistent objects are saved and retrieved through a EntityManager object.
- The EntityManager objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed them as needed.
- The main function of the EntityManager is to offer, create, read, and delete operations for instances of mapped entity classes.

EntityTransaction

- EntityTransaction represents a unit of work with the database and most of the RDBMS supports transaction functionality.
- Transactions in Hibernate are handled by an underlying transaction manager and transaction (from JDBC).
- In hibernate framework, we have EntityTransaction Interface that defines the unit of work.
- The EntityTransaction interface provides methods for transaction management.

Sample:

```
EntityTransaction txn = entityManager.getTransaction();
Txn.begin();
// some persistent operation
txn.commit();
```

- **JPQL-Hibernate Query Language :**

- Hibernate Query Language (JPQL) is same as SQL (Structured Query Language) but it doesn't depends on the table of the database.

- Instead of table name, we use class name in JPQL. So it is database independent query language.

- Although you can use SQL statements directly with Hibernate using Native SQL, but recommend to use JPQL whenever possible to avoid database portability hassles, and to take advantage of Hibernates SQL generation and caching strategies.

- Keywords like SELECT, FROM, and WHERE, etc., are not case sensitive, but properties like table and column names are case sensitive in JPQL.

Example :

```
String JPQL = "FROM Employee";
Query query = entityManager.createQuery(JPQL);
List results = query.list();
```

- **Advantage of JPQL :**

There are many advantages of JPQL. They are as follows:

- database independent
- supports polymorphic queries
- easy to learn for Java Programmer

- **FROM CLAUSE** :

- If you want to load a complete persistent objects into memory following is the simple **syntax** of using **FROM** clause.

- **Syntax :**

```
String JPQL = “FROM Employee”;
Query query = session.createQuery(JPQL);
List results = query.list();
```

- If you need to fully qualify a class name in JPQL , just specify the packages and class name as follows :

```
String JPQL = “FROM  Employee”;
Query query = session.createQuery(JPQL);
 List Results = query.list();
```

- **SELECT CLAUSE** :

- Select Clause - provides more control over the result set then the FROM clause.

- If you want to obtain few properties of objects instead of the complete object use the select clause.

Syntax With Select Clause

```
String JPQL = “Select E.firstname FROM Employee E”;
Query query = session.createQuery(JPQL);
List results = query.list();
```

- **WHERE CLAUSE :**

- If you want to narrow the specific objects that are returned from storage you can use the Where Clause.

- **Syntax with Where clause :**

```
String JPQL = “FROM Employee E WHERE E.id = 10 ”;
Query query = session.createQuery(JPQL);
List results = query.list();
```

- **Named Parameters :**

- Hibernate supports named parameters in its JPQL queries.

- This makes writing JPQL queries that accept input from the user easy and you do not have to defend against SQL injection attacks.

- Following is the simple syntax of using named parameters –

```
String JPQL = "FROM Employee E WHERE E.id = :employee_id";
Query query = session.createQuery(JPQL);
query.setParameter("employee_id",10);
List results = query.list();
```

Difference Between SQL And JPQL

SQL	JPQL
Structured Query Language	Hibernate Query Language.
SQL uses the Concepts of relational database management to manage the data	JPQL is a combination of object-oriented programming with relational database concepts
SQL manipulates data stored in tables and modifies its rows and columns	It is similar to SQL, It represents SQL queries in the form of objects
SQL is concerned about the relationship that exists between two tables	JPQL considers the relation between two objects

- JPQL Queries are in the form of objects that are translated into SQL Queries in the target Database.
- JPQL is also Case Insensitive.
- JPQL supports concepts like polymorphism, inheritance, association,etc.
- It is a powerful and easy to learn.
- JPQL works with classes and their properties to finally be mapped to a table structure.

Native SQL

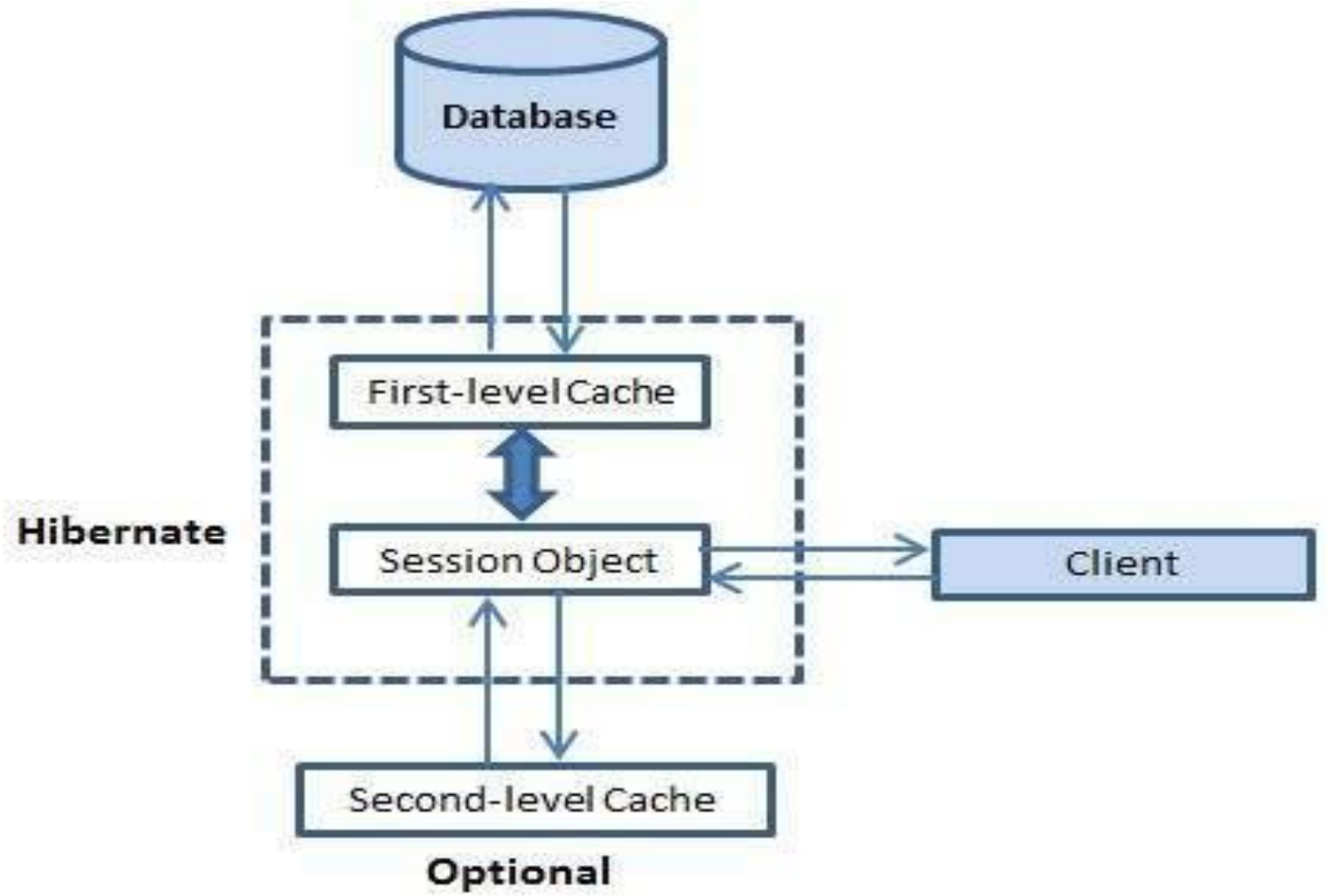
- By using Native SQL, we can perform both select, non-select operations on the data.
- In face Native SQL means using the direct SQL command specific to the particular (current using) database and executing it with using hibernate.
- We can use the database specific keywords (commands), to get the data from the database.
- While migrating a JDBC program into hibernate, the task becomes very simple because JDBC uses direct SQL commands and hibernate also supports the same commands by using this Native SQL.
- The **main draw back of Native SQL is**, some times it makes the hibernate application as database dependent one.

Caching

- Caching is a mechanism to enhance the performance of a system. It is a buffer memory that lies between the application and the database.
- Cache memory stores recently used data items in order to reduce the number of database hits as much as possible.
- Caching is important to Hibernate as well.

Caching

- It utilizes a multilevel caching scheme as explained below −



First Level Cache

- The first-level cache is the Session cache and is a mandatory cache through which all requests must pass.
- The Session object keeps an object under its own power before committing it to the database.
- If you issue multiple updates to an object, Hibernate tries to delay doing the update as long as possible to reduce the number of update SQL statements issued. If you close the session, all the objects being cached are lost and either persisted or updated in the database.
- 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.

Second Level Cache

- This is apart from first level cache which is available to be used globally in session factory scope.
- Above statement means, **second level cache is created in session factory scope** and is **available to be used in all sessions** which are created using that particular session factory.
- Hibernate uses first-level cache by default and you have nothing to do to use first-level cache. Not all classes benefit from caching, so it's important to be able to disable the second-level cache.
- The Hibernate second-level cache is set up in two steps. First to decide which concurrency strategy to use.
- After that, configure cache expiration and physical cache attributes using the cache provider.

Second Level Cache

- **To enable second level cache :**

@Cacheable
@Cache(usage = CacheConcurrencyStrategy.***READ_ONLY)***

- **Concurrency Strategies :**

- A concurrency strategy is a mediator, which is responsible for storing items of data in the cache and retrieving them from the cache.

- To enable a second-level cache, one should decide, for each persistent class and collection, which cache concurrency strategy to use.

- **Transactional** − Use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions, in the rare case of an update.

Second Level Cache

- **Read-write** − Again use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions, in the rare case of an update.

- **Nonstrict-read-write** − This strategy makes no guarantee of consistency between the cache and the database. Use this strategy if data hardly ever changes and a small likelihood of stale data is not of critical concern.

- **Read-only** − A concurrency strategy suitable for data, which never changes. Use it for reference data only.

Hibernate –Annotations

- **@Entity Annotation:**

- The annotations are contained in the **javax.persistence** package.

- The **@Entity** annotation marks the class as an entity bean, so it must have a no-argument constructor that is visible with at least protected scope.

- **@Table Annotation :**

- The @Table annotation allows you to specify the details of the table that will be used to persist the entity in the database.

- The @Table annotation provides four attributes, allowing you to override the name of the table, its catalogue, and its schema, and enforce unique constraints on columns in the table.

Hibernate –Annotations

- **@Id and @Generated Value Annotations:**

- Each entity bean will have a primary key, which annotated on the class with the **@Id** annotation.
- The primary key can be a single field or a combination of multiple fields depending on your table structure.
- By default, the @Id annotation will automatically determine the most appropriate primary key generation strategy to be used to override this by applying the **@Generated Value** annotation, which takes two parameters **strategy** and **generator**.

- **@Column Annotation :**

- The @Column annotation is used to specify the details of the column to which a field or property will be mapped.

Hibernate –Annotations

- You 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.

Hibernate- Annotation

- **@Embeddable Annotation :**

- This annotation is used to specify the fields in the class can be the composite key for some Entity class. And this annotation is used to the class.

- **@EmbeddedId Annotation :**

- This annotation is used to specify the composite key and the field should be the reference variable of one of the class which is annotated with @Embeddable.

Mapping

- **Association Mapping Types :**

1. One-to-One : Mapping one-to-one relationship using Hibernate
2. One-to-Many : Mapping one-to-many relationship using Hibernate
3. Many-to-One : Mapping many-to-one relationship using Hibernate
4. Many-to-Many :Mapping many-to-many relationship using Hibernate

OneToOne

- One-to-one mappings represent simple pointer references between two Java objects.
- In Java, a single pointer stored in an attribute represents the mapping between the source and target objects.
- Relational database tables implement these mappings using foreign keys.
- Various supported techniques for one to one mapping

1. Using foreign key association
2. Using common join table
3. Using shared primary key

OneToOne…

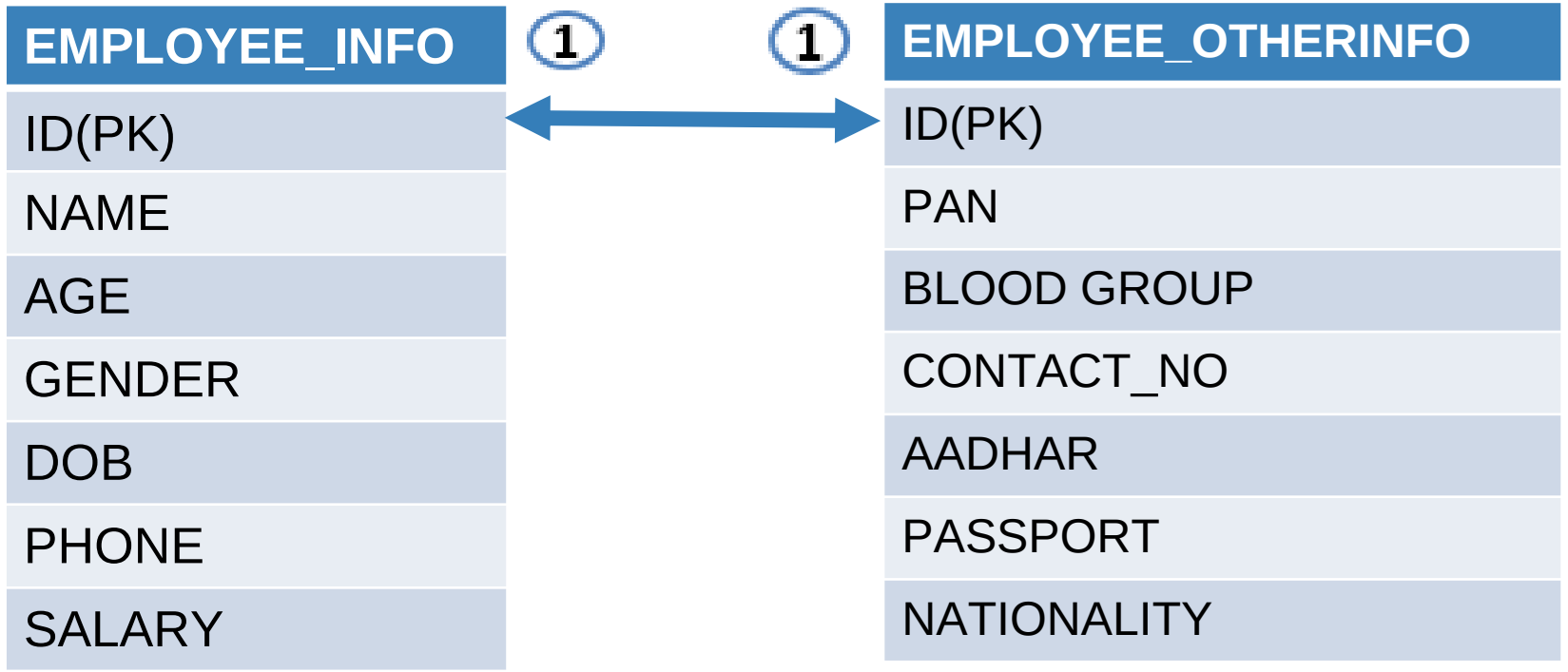
- In hibernate there are 3 ways to create one-to-one relationships between two entities. Either way we have to use **@OneToOne** annotation.

- **Example:-**

```
@Entity
@Table(name="employee_info")
public class EmployeeInfoBean
                @Id
                @OneToOne(cascade = CascadeType.ALL)
                @JoinColumn(name="id", referencedColumnName="id")
                 private EmployeeOtherInfo otherinfo;
```

OneToOne…

- Employee Info Class :



OneToOne…

- Above Table Structure shows one to one mapping with primary key association
- In this kind of association, a primary key column is created in both the tables
- This column will store the primary key for EmployeeInfo Table. Just like shows in the above table structure.
- To make such association , refer the EmployeeOtherInfo Entity in EmployeeInfo class.

Example : EmployeeInfo Class

```
@OneToOne
@JoinColumn
private EmployeeOtherInfo otherinfo;
```

OneToMany

- **Hibernate one to many mapping -** is made between two entities
- where first entity can have relation with multiple second entity instances
- but second can be associated with only one instance of first entity
- Its **1 to N** relationship
- For example, in any company an employee can have multiple address but in one address will be associated with one and only one employee.

OneToMany…

- **When to use one to many mapping :**

✓ Hibernate one to many mapping solutions

✓ Hibernate one to many mapping with foreign key association .

✓ Hibernate one to many mapping with join column .

Example :

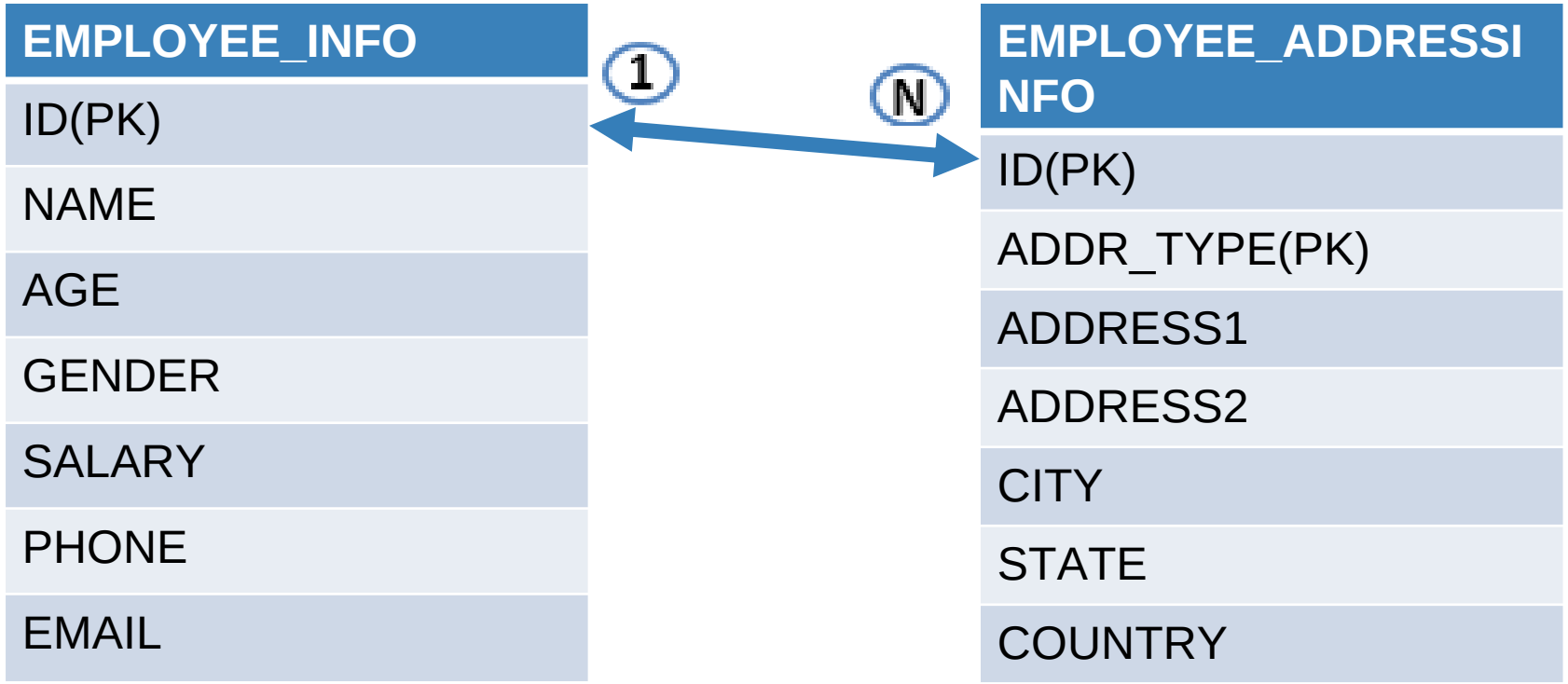
Following Tables shows the relation between two Entity classes-

EMPLOYEE_INFO Relation with EMPLOYEE_ADDRESSINFO.

One Employee having many address so ONE TO MANY RELATION.(1:N)

OneToMany…

- ONE TO MANY : 1 : N Relationship



OneToMany…

- Above Table Structure shows one to one mapping with primary key association
- In this kind of association, a primary key column is created in both the tables
- This column will store the primary key for EmployeeInfo Table into EmployeeAddressInfo Table. Just like shows in the above table structure.
- To make such association, refer the List of EmployeeAddressInfo Entity in EmployeeInfo class.

Example : EmployeeInfo Class

```
@OneToMany
@JoinColumn
private List<EmployeeAddressInfo> addressInfos;
```

ManyToOne

- **Hibernate many to one mapping -** is made between two entities
- where first entity will have relation with one second entity instances
- but second can be associated with many instance of first entity
- Its **N to 1** relationship
- For example, one Gmail account will be associated with only one user but one user can have multiple Gmail account.

ManyToOne…

- **When to use many to one mapping :**

✓ Hibernate many to one mapping solutions

✓ Hibernate many to one mapping with foreign key association

✓ Hibernate many to one mapping with join column

Example :

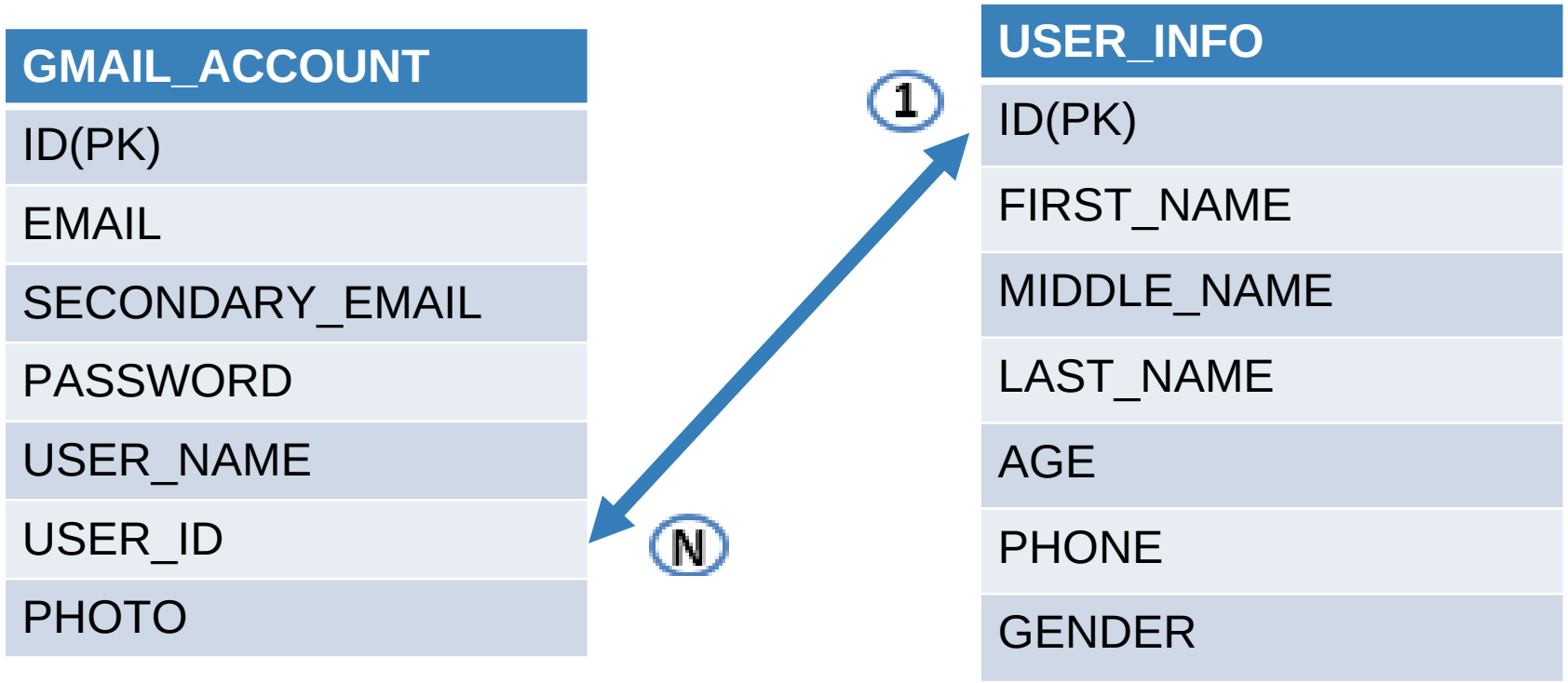
Following Tables shows the relation between two Entity classes-

USER_INFO Relation with GAMIL_ACCOUNT.

Many Gmail account associated with one User MANY TO ONE RELATION.(N:1)

ManyToOne…

- MANY TO ONE : 1 : N Relationship



ManyToOne…

- Above Table Structure shows one to one mapping with primary key association
- In this kind of association, a primary key column is created in both the tables
- This column will store the primary key for UserInfo Table to GmailAccount just like shows in the above table structure.
- To make such association, refer the UserInfo Entity in GmailAccount class.

Example : GmailAccount Class

```
@ManyToOne
@JoinColumn
private UserInfo info;
```

ManyToMany

- **Hibernate many to many mapping -** is made between two entities
- where first entity will have relation with many second entity instances
- and second will be associated with many instance of first entity
- Its **N to N** relationship
- For example, one student will be associated with many courses and one course can have multiple students.

ManyToMany…

- **When to use many to many mapping :**

✓ Hibernate many to many mapping solutions

✓ Hibernate many to many mapping with foreign key association .

✓ Hibernate many to many mapping with join table .

Example :

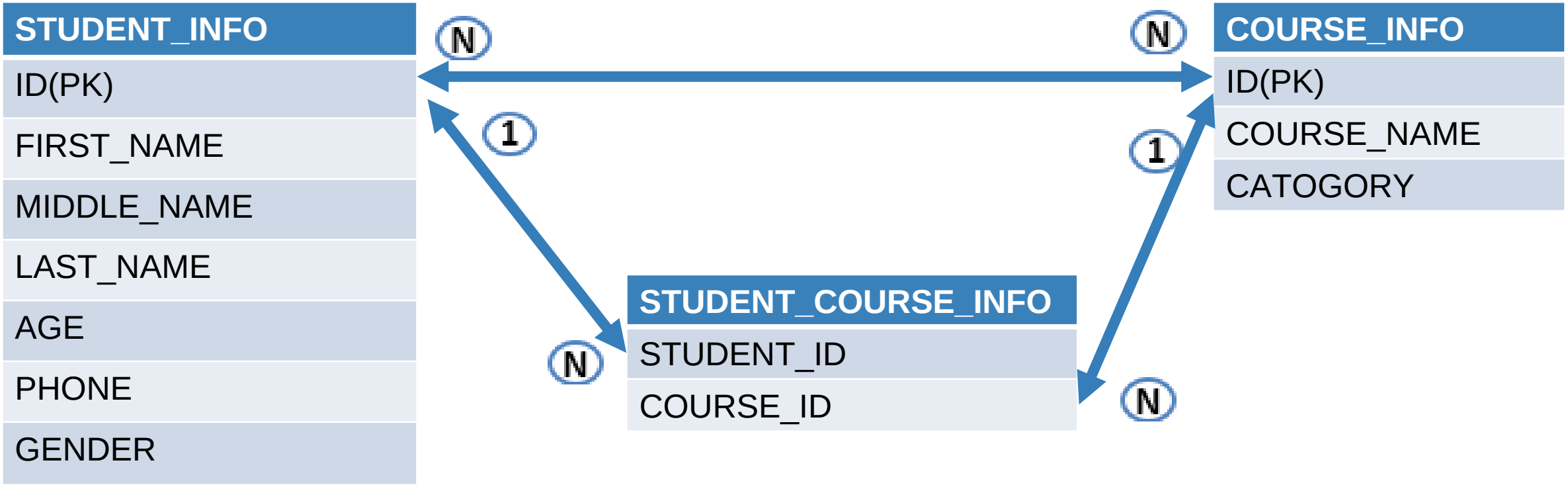
Following Tables shows the relation between two Entity classes-

STUDENT_INFO Relation with COURSE_INFO.

Many Students associated with many Courses, MANY TO MANY RELATION.(N:N)

ManyToMany…

- MANY TO MANY : N : N Relationship



ManyToMany…

- Above Table Structure shows many to many mapping
- In this kind of association, new table will be created to define the relation between these two tables, and that
- This will store the primary key from StudentInfo Table to StudentCourseInfo and primary key from CourseInfo to StudentCourseInfo just like shows in the above table structure.
- To make such association, refer the List of CourseInfo Entity in StudentInfo class.

Example : StudentInfo Class

```
@ManyToMany
@JoinTable(name = “student_course_info”,
joinColumns = @JoinColumn(name=“id”, referencedCoulmnName = “student_id”),
inverseJoinColumn = @JoinColumn(name=“id”, referencedCoulmnName = “course_id”)
private UserInfo info;
```

TESTYANTRA

SOFTWARE SOLUTIONS (INDIA) PVT. LTD.

Thank You !!!

No.01, 3rd Cross Basappa Layout, Gavipuram Extension, Kempegowda Nagar, Bengaluru, Karnataka 560019

praveen.d@testyantra.com

www.testyantra.com

EXPERIENTIAL learning factory