What is API in Java

An API stands for **Application Programming Interface**. An API in Java is a collection of prewritten packages, classes, and interfaces with their respective methods, fields, and constructors. Sometimes, it is required for a programmer to use certain technologies without much concern about internal implementations.

API is useful in these situations. They make it easier for the developers to build applications by using predefined operations in APIs. There is more than 4500 APIs available in Java.

One example of Java API is REST API. It is a web standard architecture and uses the HTTP protocol for data communication.

Overall, API supports the development process. Furthermore, API will reduce the code length and improve the code reusability. They also help to access remote resources and are used for communication between services.

What is Framework in Java

Java framework is a collection of classes of predefined code that allows the developers to add them to their own programs to solve a problem. It provides the required functionalities to build and deploy an application.

A Java framework provides functionalities as a part of a larger software platform. It can consist of support programs, compilers, code libraries, toolsets and APIs that support the development of the entire project.

Framework is different from a usual library due to a number of reasons.

1. Firstly, it provides inversion of control. This means the flow of the program is controlled by the framework.
2. Secondly, a Java framework is extensible. Therefore, the programmer can extend the framework by overriding the methods or by adding specialized code that performs specific functionalities.
3. Thirdly, there is a non-modifiable framework code. Therefore, programmers can extend the framework without changing the code. Overall, a Java

framework provides multiple advantages. It increases efficiency and makes the application more secure.

Difference Between API and Framework in Java

API	Framework
API in Java is a set of subroutine definitions, communication protocols, and tools for building software.	Framework in Java is an abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application specific software
API works as an interface between applications	Java Framework is used to design applications such as MVC web applications. They provide the model to develop the application.

Examples

REST is an example for a Java API. Spring is an example for a Java framework.

Conclusion

Both API and Framework in Java help to build robust applications. The difference between API and Framework in Java is that Java API is an interface to a set of components that encapsulates functionalities while a framework is a set of classes, tools, and related components that help to develop the project.

Hibernate is used to overcome the of limitations of JDBC like:

1. JDBC code is dependent upon the Database software being using. i.e. Our persistence logic is dependent because of using JDBC. Here we are inserting a record into Employee table but our query is Database software dependent i.e. Here we are using MySQL. But if we change our Database then this query wont work.

```
    Connection con= null;
    PreparedStatement pstmt=null;
    ResultSet rs=null;
    String qry="select * from .usertable where email=? and password=?";
try {
        Class.forName("com.mysql.jdbc.Driver");
        con=DriverManager.getConnection("jdbc:mysql://localhost:3306?user=root&password=root");
        pstmt=con.prepareStatement(qry);
        pstmt.setString(1, "iamadamhussain@gmail.com");
        pstmt.setString(2, "");
        rs=pstmt.executeQuery();
        while (rs.next()) {
        String nm=rs.getString(2);

        }}
        catch (Exception e) {
        }
```

2. If working with JDBC, changing of Database in middle of the project is very costly.
3. JDBC code is not portable code across the multiple database softwares.
4. In JDBC, Exception handling is mandatory. Here We can see that we are handling lots of Exception for connection.
5. While working with JDBC, There is no support Object level relationship.

6. In JDBC, there occurs a Boiler plate problem i.e. For each and every project we have to write the below code.That increases the code length and reduce the readability.

To overcome from the above problems we use **ORM tool** i.e. nothing but **Hibernate framework**. By using Hibernate we can avoid all the above problems

Hibernate is a framework which provides some **abstraction layer** means programmer dont have to worry about the implementations, Hibernate do implementations for you internally like **Establishing a connection with the database, writing query to perform CRUD operations etc**.

It is a **java framework** which is used to develop persistence logic.

Persistence logic means to store and process the data for long use.

Hibernate is a open source, non-invasive, light-weight java ORM(Object relational mapping) framework to develop objects which is independent of the database software and make independent persistence logic in all JAVA, JEE.

Hibernate invented by Gavin King in 2001. He also invented JBoss server and JPA.

Non-invasive means:

- The classes of Hibernate application development are loosely coupled classes with respect to Hibernate API i.e. Hibernate class need not to implement hibernate API interfaces and need not to extend from Hibernate API classes.

Functionalities supported by Hibernate framework

1. Hibernate framework support **Auto DDL** operations. In JDBC manually we have to create table and declare the data-type for each and every column. But

Hibernate can do **DDL operations** for you internally like creation of table,drop a table,alter a table etc.

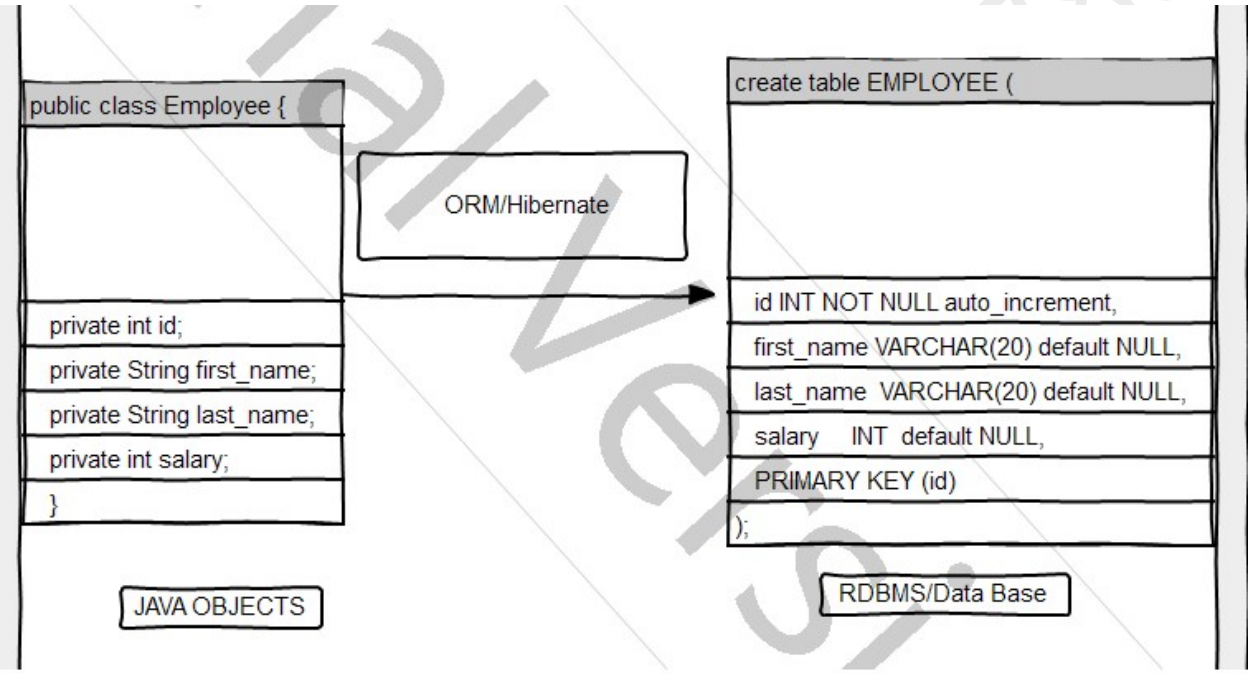
2. Hibernate supports **Auto Primary key generation**. It means in JDBC we have to manually set a primary key for a table. But Hibernate can this task for you.
3. Hibernate framework is independent of Database because it supports **HQL (Hibernate Query Language)** which is not specific to any database, whereas JDBC is database dependent.
4. In Hibernate, **Exception Handling is not mandatory**, whereas In JDBC exception handling is mandatory.
5. Hibernate supports **Cache Memory** whereas JDBC does not support cache memory.
6. Hibernate is a **ORM tool** means it support Object relational mapping. Whereas JDBC is not object oriented moreover we are dealing with values means primitive data. In hibernate each record is represented as a Object but in JDBC each record is nothing but a data which is nothing but primitive values.
7. Hibernate supports Inheritance, Associations, Collections
8. Hibernate supports a special query language(HQL) which is Database vendor independent.
9. Hibernate supports annotations, apart from XML.
10.

What are the different logics available in Enterprise Application?
Presentation Logic= Logic used to present the output/input.
Application/Controlling Logic= Logic used t o control the flow of application.
Business Logic=Programmatical implementation of business rules is nothing but business logic. Data Access Logic=Logic used to contact the Database.

What is persistence in a java based enterprise application?

The process of storing enterprise data in to relational database is known as persistence

What is ORM?
object/relational mapping is the automated (and transparent) persistence of objects in a Java application to the tables in a relational database, using metadata that describes the mapping between the objects and the database.
ORM, in essence, works by (reversibly) transforming data from one representation to another.
Orm is also called as object role modeling/object reatonal mapping.



Java ORM Frameworks:

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

1. Enterprise JavaBeans Entity Beans
2. Java Data Objects
3. TopLink
4. Spring DAO
5. Hibernate
6. And many more
7.

What are the Simple Hibernate Application Requirements?
1 Entity class/POJO class
2. Mapping file(Required if you are not using annotations)
3. Configuration file(hibernate.cfg.xml
4. DAO class (Where we write our logic to work with database)

What are the Steps to develop hibernate applications? -
Step 1: Develop persistent/domain/entity/pojo class for each table of the relational model
Step 2: For each entity develop a mapping file or Hibernate Annotation
Step 3: Develop the configuration file(hibernate.cfg.xml)
Step 4: Add hibernate framework jar files i n the classpath
Step 5: Make use of hibernate API and perform persistent operations

How to Make use of hibernate **API** to perform persistent'operations?
STEP1:
Create Configuration object
Configuration configuration = new Configuration();
STEP2:
Read configuration file along with mapping files using configure() rnethod of Configuration Object
configuration.configure();
STEP3:
Build a SessionFactory from Configuration
SessionFactory factory = **configuration.bhildSessionFactory();**
ST P4:
Get Session from SessionFactory object
Session session = factory.openSession();
STEP 5:
BEGIN Transacton
Transaction transaction=session.beginTransaction();
transaction.begin();

STEP6:
Perform persistence operations
Save/delete/read/update

STEP7: commit transaction and close session.

What is hibernate configuration file?

It is an XML file in which database connection details (username, password, url, driver class name) and ,Hibernate Properties(dialect, show-sql, second-level-cache ... etc) and Mapping file name(s) are specified to the hibernate

Hibernate uses this file to establish connection to the particular database server .
Standard for this file is <hibernate.cfg.xml>
We must create one configuration file for each database we are going to use, suppose if we want to connect wi th 2 databases, like Oracle, MySql, then we must create 2 configuration files.

No. of databases **we** are using = That many number of configuration files

We can write this configuration in 2 ways ...

1. XML file
2. Properties file(old style)

We don't have annotations to write configuration details. Actually in hibernate 1.x, 2.x we defined this configuration by using .properties file, **but from 3.x XML came into picture.**
XmL files are always recommended to use

Example Of Configuration xml:

```
<!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>
<!-- database information ?createDatabaseIfNotExist=true-->
<property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/raj</property>
<property name="hibernate.connection.user">root</property>
<property name="hibernate.connection.password">root</property>

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

<property name="hbm2ddl.auto">update</property>
<mapping class="com.rani.dto.UserDto"/>
</session-factory>
</hibernate-configuration>
```

2)Pojo class/Entity class

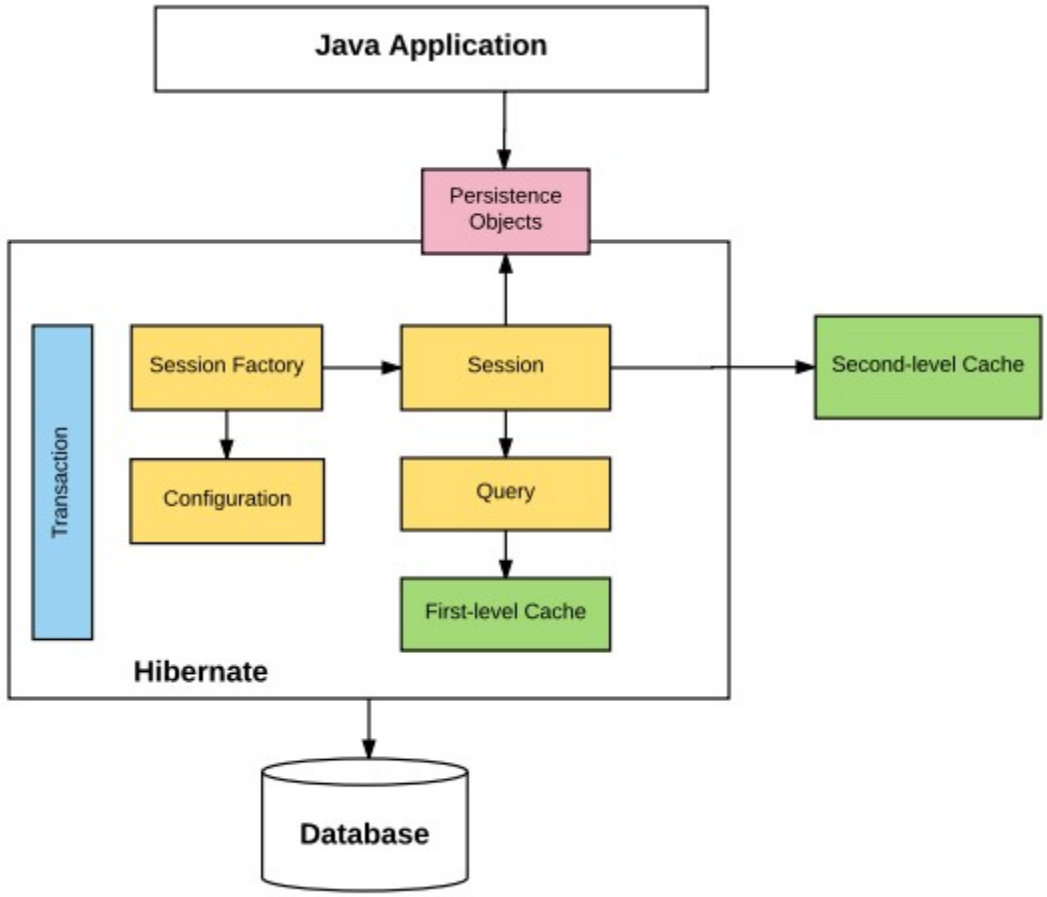
```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="user")
public class UserDto {
    @Id
    @GeneratedValue
    private int id;

    private  String email;

    private String name;
    private String password;
    public int getId() {
```

3)TestDao class or Testmain which has method

```
UserDto userDto=new UserDto();
userDto.setName("Adam");
userDto.setEmail("iamadamhussain@gmail.com");
userDto.setPassword("yourpassword");

Configuration configuration=new Configuration();
configuration.configure();
SessionFactory  factory=configuration.buildSessionFactory();

Session session=factory.openSession();
Transaction transaction=session.beginTransaction();

session.save(userDto);
transaction.commit();
session.close();
```

The above class following steps

Hibernate Architecture



The above diagram as follows :

Configuration:

Configuration is a class which is present in **org.hibernate.cfg** package. It activates Hibernate framework. It reads both configuration file and mapping files.

- It activate Hibernate Framework
- **Configuration cfg=new Configuration();**
- It read both cfg file and mapping files
- **cfg.configure();**
- It checks whether the config file is syntactically correct or not.
- If the config file is not valid then it will throw an exception. If it is valid then it creates a meta-data in memory and returns the meta-data to object to represent the config file.

SessionFactory:

- SessionFactory is an Interface which is present in org.hibernate package and it is used to create Session Object.
- It is immutable and thread-safe in nature.
- buildSessionFactory() method gathers the meta-data which is in the cfg Object.
- From cfg object it takes the JDBC information and create a JDBC Connection.
- **SessionFactory factory=cfg.buildSessionFactory();**

Session:

- Session is an interface which is present in org.hibernate package. Session object is created based upon SessionFactory object i.e. factory.
- It opens the Connection/Session with Database software through Hibernate Framework.
- It is a light-weight object and it is not thread-safe.
- Session object is used to perform CRUD operations.
- **Session session=factory.buildSession();**

Transaction:

- Transaction object is used whenever we perform any operation and based upon that operation there is some change in database.
- Transaction object is used to give the instruction to the database to make the changes that happen because of operation as a permanent by using commit() method.
- **Transaction tx=session.beginTransaction();**
- **tx.commit();**

In HQL see in Later example

Query:

- Query is an interface that present inside org.hibernate package.
- A Query instance is obtained by calling Session.createQuery().
- This interface exposes some extra functionality beyond that provided by Session.iterate() and Session.find():

1. A particular page of the result set may be selected by calling setMaxResults(), setFirstResult().
2. Named query parameters may be used.

Query query=session.createQuery();

Dialect means **"the variant of a language**"
There may a chances for some point hibernate has to use database specific SQL. Hibernate uses "dialect" configuration to know which database you are using so that it can switch to the database specific SQL generator code wherever/whenever necessary.
Dialect is a communicator to JDBC from Hibernate. So then Hibernate should have one Dialect to communicate with different Database vendor

RDBMS	Dialect
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
PostgreSQL	org.hibernate.dialect.PostgreSQLDialect
MySQL	org.hibernate.dialect.MySQLDialect
MySQL with InnoDB	org.hibernate.dialect.MySQLInnoDBDialect
MySQL with MyISAM	org.hibernate.dialect.MySQLMyISAMDialect
Oracle (any version)	org.hibernate.dialect.OracleDialect
Oracle 9i/10g	org.hibernate.dialect.Oracle9Dialect
Sybase	org.hibernate.dialect.SybaseDialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect
Microsoft SQL Server	org.hibernate.dialect.SQLServerDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Informix	org.hibernate.dialect.InformixDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Ingres	org.hibernate.dialect.IngresDialect
Progress	org.hibernate.dialect.ProgressDialect
Mckoi SQL	org.hibernate.dialect.MckoiDialect
Interbase	org.hibernate.dialect.InterbaseDialect
Pointbase	org.hibernate.dialect.PointbaseDialect
FrontBase	org.hibernate.dialect.FrontbaseDialect

Firebird	org.hibernate.dialect.FirebirdDialect

What is hibernate mapping file?
In this file hibernate application developer specify the mapping from entity class name to table name and entity properties names to table column names. i.e. mapping object oriented data to relational data .
Standard name for this file is **<domain-object-name.hbm.xml>**
In general, for each domain object we create one mapping file.
Number of Entity classes = that many number of mapping xmls.
Mapping can be done using annotations also. If we use annotations for mapping then we no need to write mapping file.
From hibernate 3.x version on wards it provides support for annotation, So mapping can be done in two ways
o XML
o Annotations

example **Of** Mapping xml:

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

<class name="FULLY QUALIFIED CLASS NAME" table="DB table name">
<id name="JAVA FILED NAME" column="DB COLUMN NAME">
<generator class="increment"></generator>
</id>

<property name=" JAVA FILED NAME " column=" DB COLUMN NAME "  />
<property name=" JAVA FILED NAME " column=" DB COLUMN NAME "/>

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

Generator classes in Hibernate

The <generator> subelement of id used to generate the unique identifier for the objects of persistent class. There are many generator classes defined in the Hibernate Framework.

All the generator classes implements the **org.hibernate.id.IdentifierGenerator interface**. The application programmer may create one's own generator classes by implementing the IdentifierGenerator interface. Hibernate framework provides many built-in generator classes:

1. assigned
2. increment
3. sequence
4. hilo
5. native
6. identity
7. seqhilo
8. uuid
9. guid
10. select
11. foreign
12. sequence-identity

increment	generates identifiers of type long, short or int that are unique only when no other process is inserting data into the same table.
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.

Caching mech

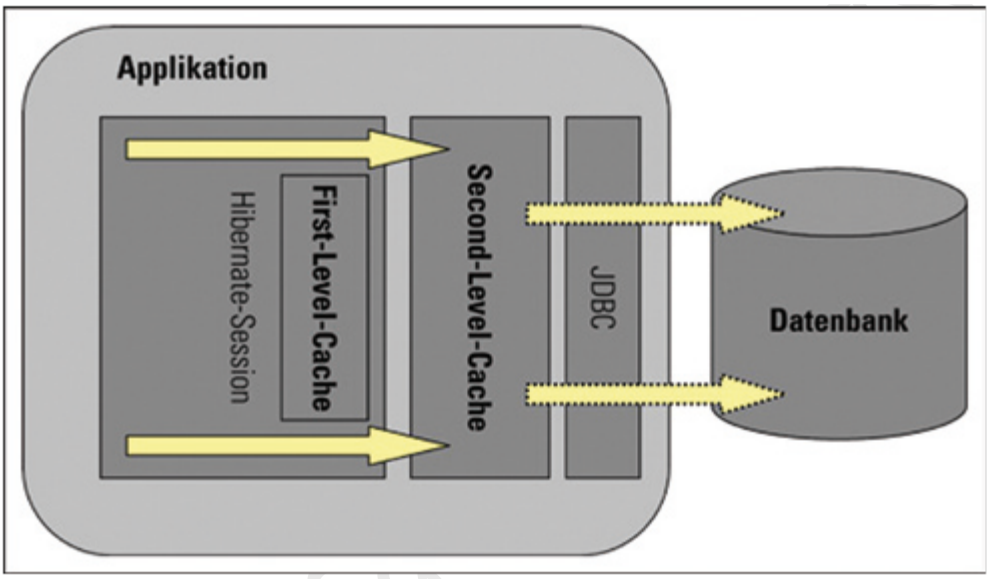
Caching is facility provided by ORM frameworks which help users to get fast running web application, while help framework itself to reduce number of queries made to database in a single transaction. Hibernate also provide this caching functionality, in two layers.

The first-level cache: The first level cache type is the **session cache**. The session cache caches object within the current session but this is not enough for long level i.e. session factory scope.
The second-level cache: The second-level cache is called 'second-level' because there is already a cache operating for you in Hibernate for the duration you have a session open. A Hibernate Session is a transaction-level cache of persistent data. It is possible to configure a SessionFactory-level cache on a class-by-class and collection-by-collection basis.

Hibernate uses first-level cache by default and you have nothing to do to use first-level cache. Let's go straight to the optional second-level cache. Not all classes benefit from caching, so it's important to be able to disable the second-level cache.

Note: **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.
It also means that **once session factory is closed, all cache associated with it die**



Hibernate Relational Associations

Two important features of Hibernate are Inheritance (Hierarchical) Mapping (supported by table-per-subclass etc.) and Relational Associations (supported by one-to-many etc.)

In database terminology, an **association denotes a relationship between two tables**. This tutorial explains the way how Hibernate writes Java bean classes to implement this relation.

Some table associations require an extra **join table** with primary and foreign keys which Hibernate also supports. As in SQL, it is not a good idea to have nullable foreign keys and this is not a requirement for Hibernate.

The associations may be unidirectional or bidirectional.

There are two types of Assoaction

1)IS-A relationship

2)Has-A relationship

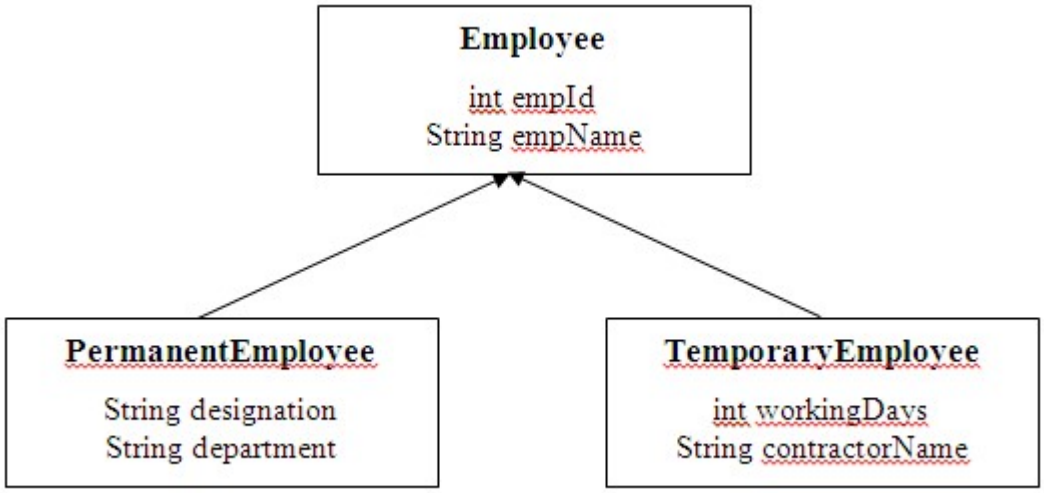
Inheritance Hierarchical Mapping

Java, being an OOPs language, supports inheritance for reusability of classes. Two types of reusability exists – "is-a" and "has-a" relationship. The relational model supported by Hibernate is "has-a" relationship. How Hibernate writes tables for the Java classes involved in inheritance?

1)IS-A Relationship

Hibernate comes with a provision to create tables and populate them as per the Java classes involved in inheritance. Hibernate offers basically **3 different approaches** to map hierarchical classes (classes involved in inheritance) with database tables.

Here, **we have classes but no tables**. Tables are created by Hibernate automatically. There are **3 styles** to do this job. Explained in this tutorial "Inheritance Hierarchical Mapping".



Observe, in the above hierarchy, **three classe**s are involved where **Employee** is the super class and **PermanentEmployee** and **TemporaryEmployee** are subclasses with their own properties declared as instance variables. Now the question is how many tables are required and moreover how to link the tables so that **PermanentEmployee** gets four properties of **empId** and **empName** (from super class), **designation** and **department**.

The three approaches adopted by Hibernate are

1. **table-per-class-hierarchy:** Only **one table is created** for all the classes involved in hierarchy. Here we maintain an extra **discriminator field** in table to differentiate between PermanentEmployee and TemporaryEmployee.
2. **table-per-subclass: One table for each class is created**. The above hierarchy gets three tables. Here, foreign key is maintained between the tables.
3. **table-per-concrete-class: One table for each concrete class (subclass) is created but not of super class**. The above hierarchy gets two tables. As a special case, the super class can be an abstract or interface. Here, foreign key is not maintained.

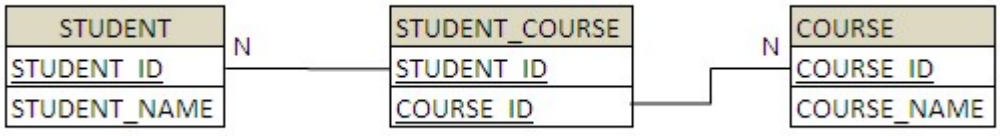
2)HAS-A Relationship:

The 4 associations Hibernate supports are

1. One to One
2. One to Many
3. Many to One
4. Many to Many

Hibernate Many-To-Many Mapping(Uni directional)

To create this relationship you need to have
a *STUDENT*, *COURSE* and *STUDENT_COURSE* table. The relational model is shown below.



To create the *STUDENT*, *COURSE* and *STUDENT_COURSE* table you need to create the following Java Class files.

```
import java.util.HashSet;
import java.util.Set;
```

```
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
import javax.persistence.Table;
@Entity
@Table(name = "STUDENT")
public class Student {
@Id
@GeneratedValue
		@Column(name = "STUDENT_ID")
		private long studentId;
@Column(name = "STUDENT_NAME", nullable = false, length = 100)
private String studentName;
@ManyToMany(cascade = CascadeType.ALL)
		@JoinTable(name = "STUDENT_COURSE",
 joinColumns = { @JoinColumn(name = "STUDENT_ID") },
 inverseJoinColumns = { @JoinColumn(name = "COURSE_ID") })
private Set<Course> courses = new HashSet<Course>(0);


		public Student() {
		}
public long getStudentId() {
```

```
                return this.studentId;
        }
public void setStudentId(long studentId) {
                this.studentId = studentId;
        }
        public String getStudentName() {
                return this.studentName;
        }
public void setStudentName(String studentName) {
                this.studentName = studentName;
        }
public Set<Course> getCourses() {
                return this.courses;
        }

        public void setCourses(Set<Course> courses) {
                this.courses = courses;
        }
}
```

The *@ManyToMany* annotation is used to create the many-to-many relationship between the *Student* and *Course* entities. The *@JoinTable* annotation is used to create the *STUDENT_COURSE*link table and *@JoinColumn* annotation is used to refer the linking columns in both the tables.

Course class is used to create the *COURSE* table.

```
@Entity
@Table(name="COURSE")
```

```
public class Course {
@Id
	@GeneratedValue
	@Column(name="COURSE_ID")
	private long courseId;
@Column(name="COURSE_NAME", nullable=false)
private String courseName;

	public long getCourseId() {
		return this.courseId;
	}

	public void setCourseId(long courseId) {
		this.courseId = courseId;
	}

		public String getCourseName() {
		return this.courseName;
	}

	public void setCourseName(String courseName) {
		this.courseName = courseName;
	}

}
```

This side is much simpler than the owner side, as we only need to specify the **mappedBy** attribute of the **@ManyToMany**annotation.

Hibernate.cfg.xml refer to class notes

Create the *Main* class to run the example.

```
public class Main {

	public static void main(String[] args) {

		Session session = HibernateUtil.getSessionFactory().openSession();
		Transaction transaction = null;
		try {
			transaction = session.beginTransaction();

			Set<Course> courses = new HashSet<Course>();
			courses.add(new Course("java"));
			courses.add(new Course("J2ee"));

			Student student1 = new Student("Adam", courses);
			Student student2 = new Student("Hussain", courses);
			session.save(student1);
			session.save(student2);
```

```
                    transaction.commit();
                } catch (HibernateException e) {
                    transaction.rollback();
                    e.printStackTrace();
                } finally {
                    session.close();
                }

        }
}
```


What is MVC?

MVC is an architecture Design patterns that separates business logic, presentation and data. In MVC,

- M stands for Model
- V stands for View
- C stands for controller.

Model Layer:

- This is the data layer which consists of the business logic of the system.
- It consists of all the data of the application

- It also represents the state of the application.
- It consists of classes which have the connection to the database.
- The controller connects with model and fetches the data and sends to the view layer.
- The model connects with the database as well and stores the data into a database which is connected to it.

View Layer:

- This is a presentation layer.
- It consists of HTML, JSP, etc. into it.
- It normally presents the UI of the application.
- It is used to display the data which is fetched from the controller which in turn fetching data from model layer classes.
- This view layer shows the data on UI of the application.

Controller Layer:

- It acts as an interface between View and Model.
- It intercepts all the requests which are coming from the view layer.
- It receives the requests from the view layer and processes the requests and does the necessary validation for the request.
- This requests is further sent to model layer for data processing, and once the request is processed, it sends back to the controller with required information and displayed accordingly by the view.
- The diagram is represented below:

-

The advantages of MVC are:

- This model is suitable for large applications.
- It allows use of reusable software components to design the business logic.

- It easy to maintain and test.

For Particular Example Please refer to class Notes.

Hibernate Query Language (HQL)

The Hibernate ORM framework provides its own query language called Hibernate Query Language or HQL for short. It is very powerful and flexible and has the following characteristics:

- **SQL similarity:** HQL's syntax is very similar to standard SQL. If you are familiar with SQL then writing HQL would be pretty easy:
  from SELECT, FROM, ORDER BY to arithmetic expressions and aggregate functions, etc.
- **Fully object-oriented:** HQL doesn't use real names of table and columns. It uses class and property names instead. HQL can understand inheritance, polymorphism and association.
- **Case-insensitive for keywords:** Like SQL, keywords in HQL are case-insensitive. That means SELECT, select or Select are the same.
- **Case-sensitive for Java classes and properties:** HQL considers case-sensitive names for Java classes and their properties, meaning Person and person are two different objects.

Why to use HQL?

- **Full support for relational operations:** HQL allows representing SQL queries in the form of objects. Hibernate Query Language uses Classes and properties instead of tables and columns.

- **Return result as Object:** The HQL queries return the query result(s) in the form of object(s), which is easy to use. This elemenates the need of creating the object and populate the data from result set.

- **Polymorphic Queries:** HQL fully supports **polymorphic queries**. Polymorphic queries results the query results along with all the child objects if any.

- **Easy to Learn:** Hibernate Queries are easy to learn and it can be easily implemented in the applications.

- **Support for Advance features:** HQL contains many advance features such as pagination, fetch join with dynamic profiling, Inner/outer/full joins, Cartesian products. It also supports Projection, Aggregation (max, avg) and grouping,

Ordering, Sub queries and SQL function calls.

- **Database independent:** Queries written in HQL are database independent (If database supports the underlying feature).

Understanding HQL Syntax
Any Hibernate Query Language may consist of following elements:

- Clauses
- Aggregate functions
- Subqueries

Clauses in the HQL are:

- **from**
- **select**
- **where**
- **order by**
- **group by**

Aggregate functions are:

- **avg(...)**,
- **sum(...)**,
- **min(...)**,
- **max(...)**
- **count(*)**
- count(...),
- count(distinct ...),
- count(all...)

Subqueries
Subqueries are nothing but its a query within another query. Hibernate supports Subqueries if the underlying database supports it.

Some of the commonly supported clauses in HQL are:

1. **HQL From**: HQL From is same as select clause in SQL, from Employee is same as select * from Employee. We can also create alias such as from Employee emp or from Employee as emp.

2. **HQL Join** : HQL supports inner join, left outer join, right outer join and full join. For example, select e.name, a.city from Employee e INNER JOIN e.address a. In this query, Employee class should have a variable named address. We will look into it in the example code.
3. **Aggregate Functions**: HQL supports commonly used aggregate functions such as count(*), count(distinct x), min(), max(), avg() and sum().
4. **Expressions**: HQL supports arithmetic expressions (+, -, *, /), binary comparison operators (=, >=, <=, <>, !=, like), logical operations (and, or, not) etc.
5. HQL also supports ordre by and group by clauses.
6. HQL also supports sub-queries just like SQL queries.
7. HQL supports DDL, DML and executing store procedures too.

There is two types of query parameters binding in the ***Hibernate Query***. One is ***positioned parameter*** and another one is ***named parameter***.
But, ***hibernate***recommend to use the named parameters since it is more flexible and powerful compare to the ***positioned parameter***.

Named parameters :

Named parameters are as name itself suggests, the query string will be using the parameters in the variable name. That can be replaced at runtime and one advantage of using ***named parameter*** is, the same named parameter can be used many times in the same query.

Named Parameters Binding:
This is the most common and user friendly way. It use colon followed by a parameter name **(:example)** to define a named parameter. See examples…

Example 1 – setParameter
The **setParameter** is smart enough to discover the parameter data type for you.

```
String hql ="from Student student where student.rollNumber= :rollNumber";

Query query = session.createQuery(hql);

query.setParameter("rollNumber", "3");

List result = query.list();
```

Example 2 – setString

You can use **setString** to tell Hibernate this parameter date type is String.

```
String hql = "from Student student where student.studentName= :studentName";

Query query = session.createQuery(hql);

query.setString("studentName", "Sweety Rajput");

List result = query.list();
```

Positioned parameter :

Positional parameters

It's use question mark (?) to define a named parameter, and you have to set your parameter according to the position sequence. See example…

```
String hql = "from Student student where student.course= ? and student.studentName = ?";

Query query = session.createQuery(hql);
```

```
query.setString(0, "MCA");

query.setParameter(1, "Dinesh Rajput")

List result = query.list();
```

In Hibernate parameter binding, i would recommend always go for “Named parameters”, as it’s more easy to maintain, and the compiled SQL statement can be reuse (if only bind parameters change) to increase the performance.

How to execute HQL in Hibernate

Basically, it’s fairly simple to execute HQL in Hibernate. Here are the steps:

- Write your HQL:

  ```
  String hql = "Your Query Goes Here";
  ```

- Create a Query from the Session:

  ```
  Query query = session.createQuery(hql);
  ```

- Execute the query: depending on the type of the query (listing or update), an appropriate method is used:
  - For a listing query (SELECT):

    ```
    List listResult = query.list();
    ```

  - For an update query (INSERT, UPDATE, DELETE):

    ```
    int rowsAffected = query.executeUpdate();
    ```

- Extract result returned from the query: depending of the type of the query, Hibernate returns different type of result set. For example:
  - Select query on a mapped object returns a list of those objects.

- Join query returns a list of arrays of Objects which are aggregate of columns of the joined tables. This also applies for queries using aggregate functions (count, sum, avg, etc).

Now, let's go through various concrete examples.

List Query Example

The following code snippet executes a query that returns all Category objects:

```
String hql = "from Category";
Query query = session.createQuery(hql);
List<Category> listCategories = query.list();

for (Category aCategory : listCategories) {
   System.out.println(aCategory.getName());
}
```

Note that in HQL, we can omit the SELECT keyword and just use the FROM instead.

Search Query Example

The following statements execute a query that searches for all products in a category whose name is 'Computer':

```
String hql = "from Product where category.name = 'Computer'";
Query query = session.createQuery(hql);
List<Product> listProducts = query.list();

for (Product aProduct : listProducts) {
   System.out.println(aProduct.getName());
}
```

The cool thing here is Hibernate automatically generates JOIN query between the Product and Category tables behind the scene. Thus we don't have to use explicit JOIN keyword:

```
from Product where category.name = 'Computer'
```

If you are new to Hibernate and want to learn more, read this book: **Hibernate Made Easy: Simplified Data Persistence with Hibernate and JPA (Java Persistence API) Annotations**

Using Named Parameters Example

You can parameterize your query using a colon before parameter name, for example **:id** indicates a placeholder for a parameter named **id**. The following example demonstrates how to write and execute a query using named parameters:

```
String hql = "from Product where description like :keyword";

String keyword = "New";
Query query = session.createQuery(hql);
query.setParameter("keyword", "%" + keyword + "%");

List<Product> listProducts = query.list();

for (Product aProduct : listProducts) {
   System.out.println(aProduct.getName());
}
```

The above HQL searches for all products whose description contains the specified keyword:

```
from Product where description like :keyword
```

Then use the **setParameter(name, value)** method to set actual value for the named parameter:

```
query.setParameter("keyword", "%" + keyword + "%");
```

Note that we want to perform a LIKE search so the percent signs must be used outside the query string, unlike traditional SQL.

Insert - Select Query Example

HQL doesn't support regular INSERT statement (you know why - because the session.save(Object) method does it perfectly). So we can only write INSERT … SELECT query in HQL. The following code snippet executes a query that inserts all rows from Category table to OldCategory table:

```
String hql = "insert into Category (id, name)"
      + " select id, name from OldCategory";

Query query = session.createQuery(hql);

int rowsAffected = query.executeUpdate();
if (rowsAffected > 0) {
```

```
    System.out.println(rowsAffected + "(s) were inserted");
}
```

Note that HQL is object-oriented, so Category and OldCategory must be mapped class names (not real table names).

Update Query Example

The UPDATE query is similar to SQL. The following example runs a query that updates price for a specific product:

```
String hql = "update Product set price = :price where id = :id";

Query query = session.createQuery(hql);
query.setParameter("price", 488.0f);
query.setParameter("id", 43l);

int rowsAffected = query.executeUpdate();
if (rowsAffected > 0) {
   System.out.println("Updated " + rowsAffected + " rows.");
}
```

Delete Query Example

Using DELETE query in HQL is also straightforward. For example:

```
String hql = "delete from OldCategory where id = :catId";

Query query = session.createQuery(hql);
query.setParameter("catId", new Long(1));

int rowsAffected = query.executeUpdate();
if (rowsAffected > 0) {
   System.out.println("Deleted " + rowsAffected + " rows.");
}
```

Hibernate Criteria

Hibernate Criteria API provides object oriented approach for querying the database and getting results. We can't use Criteria in Hibernate to run update or delete queries or any DDL statements. Hibernate Criteria query is only used to fetch the results from the database using object oriented approach.

Some of the common usage of Hibernate Criteria API are;

1. Hibernate Criteria API provides Projection that we can use for aggregate functions such as sum(), min(), max() etc.
2. Hibernate Criteria API can be used with ProjectionList to fetch selected columns only.
3. Criteria in Hibernate can be used for join queries by joining multiple tables, useful methods for Hibernate criteria join are createAlias(), setFetchMode() and setProjection()
4. Criteria in Hibernate API can be used for fetching results with conditions, useful methods are add() where we can add Restrictions.
5. Hibernate Criteria API provides addOrder() method that we can use for ordering the results.

```
SessionFactory sessionFactory = HibernateUtil.getSessionFactory();
                    Session session = sessionFactory.getCurrentSession();
Transaction tx = session.beginTransaction();
```

```
//Get All Employees
                    Criteria criteria = session.createCriteria(Employee.class);
                    List<Employee> empList = criteria.list();
                    for(Employee emp : empList){
                            System.out.println("ID="+emp.getId()+",
Zipcode="+emp.getAddress().getZipcode());
                    }
```

```
// Get with ID, creating new Criteria to remove all the settings
```

```
criteria = session.createCriteria(Employee.class)
.add(Restrictions.eq("id", new Long(3)));
        Employee emp = (Employee) criteria.uniqueResult();
System.out.println("Name=" + emp.getName() + ", City="
                                + emp.getAddress().getCity());
```

Difference Between Hibernate Save And Persist Methods

If our generator class is assigned, then there is no difference between save() and persist() methods. Because generator 'assigned' means, as a programmer we need to give the primary key value to save in the database right [Hope you know this generators concept]

- In case of other than assigned generator class, suppose if our generator class name is Increment means hibernate it self will assign the primary key id value into the database right [other than assigned generator, hibernate only used to take care the primary key id value remember , so in this case if we call save() or persist() method then it will insert the record into the database normally

But here thing is, save() method can return that primary key id value which is generated by hibernate and we can see it by

```
long s = session.save(k);
```

In this same case, persist() will never give any value back to the client.

Difference Between Merge And Update Methods In Hibernate

```
1. SessionFactory factory = cfg.buildSessionFactory();
2. Session session1 = factory.openSession();
3. Student s1 = null;
4. Object o = session1.get(Student.class, new Integer(101));
5. s1 = (Student)o;
6. session1.close();

7. s1.setMarks(97);

8. Session session2 = factory.openSession();
9. Student s2 = null;
10. Object o1 = session2.get(Student.class, new Integer(101));
11. s2 = (Student)o1;
12. Transaction tx=session2.beginTransaction();
13. session2.merge(s1);
```

See from line numbers 3 – 6, we just loaded one object s1 into session1 cache and closed session1 at line number 6, so now object s1 in the session1 cache will be destroyed as session1 cache will expires when ever we say session1.close()

- Now s1 object will be in some RAM location, not in the session1 cache
- here s1 is in detached state, and at line number 7 we modified that detached object s1, now if we call update() method then hibernate will throws an error, because we can update the object in the session only
- So we opened another session [session2] at line number 8, and again loaded the same student object from the database, but with name s2
- so in this session2, we called **session2.merge(s1)**; now into s2 object s1 changes will be merged and saved into the database

Hope you are clear…, actually update and merge methods will come into picture when ever we loaded the same object again and again into the database

Hibernate Eager vs Lazy Fetch Type

The relationships are defined through joins in database. Hibernate represents joins in the form of associations like One-to-One, One-to-Many and Many-to-One. It is required to define Fetch Type when you use any of these associations. Fetch Type decides on whether or not to load all the data belongs to associations as soon as you fetch data from parent table. Fetch type supports two types of loading: Lazy and Eager. By default, Fetch type would be Lazy.

FetchType.LAZY: It fetches the child entities lazily, that is, at the time of fetching parent entity it just fetches proxy (created by cglib or any other utility) of the child entities and when you access any property of child entity then it is actually fetched by hibernate.

FetchType.EAGER: it fetches the child entities along with parent.

Lazy initialization improves performance by avoiding unnecessary computation and reduce memory requirements.

Eager initialization takes more memory consumption and processing speed is slow.

Having said that, depends on the situation either one of these initialization can be used.

So, the long story short here is:

FetchType.LAZY = Doesn't load the relationships unless explicitly "asked for" via getter
FetchType.EAGER = Loads ALL relationships