JAVA: Database Operations

Agenda

- Why database connectivity?
- JDBC
  - What & Why
  - Architecture
  - Steps
- ORM
  - What & Why
  - Sample POJO to SQL Table
  - ORM Implementations
- JPA
  - What & Why
  - E2E Picture - JPA + ORM implementation + JDBC
  - JPA Details
    - Entity
    - Mappings
    - JPA & JAVA Collections
    - Steps
    - DB Operations

JDBC: What & Why

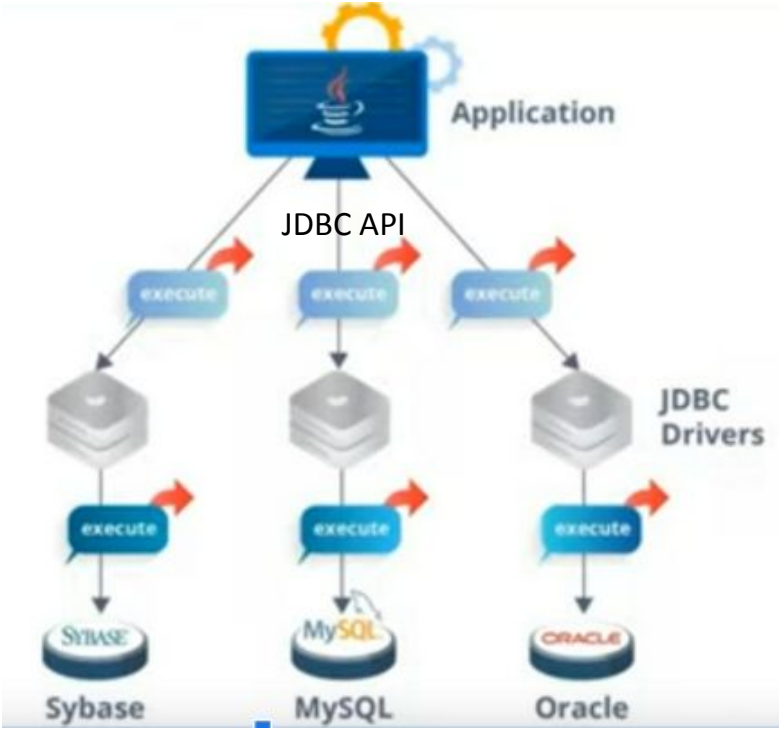
What:

JDBC stands for Java Database Connectivity. JDBC is a **standard Java API** which defines how a client may access a database in a database independent way.

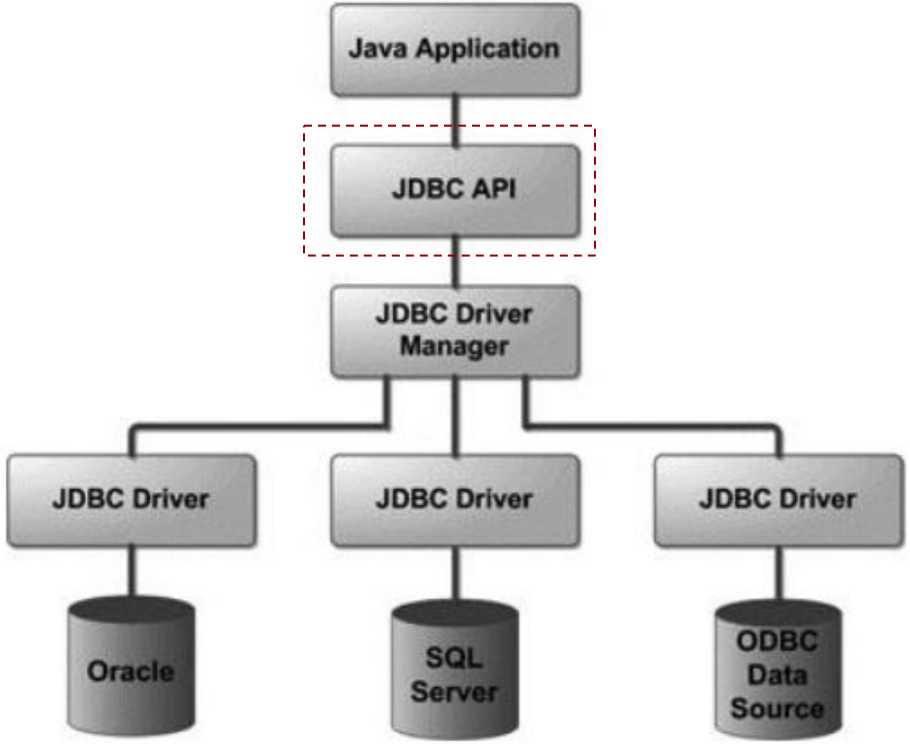
JDBC follows **ADAPTER** design pattern to hide database specific details and provides abstraction to JAVA program. Database specifics are there with drivers (adapters). It is a part of JavaSE (Java Standard Edition).

Why JDBC

1. **Loose coupling** between JAVA program with a specific database. For example for a java application to change database from ORACLE to MYSQL, NO code needs to be changed, just need to change driver name.
2. Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).



JDBC Architecture



JDBC API

- JDBC API: This provides the application-to-JDBC Manager connection.
- JDBC Driver API: This supports the JDBC Manager-to-Driver Connection.

JDBC driver manager ensures that the correct driver is used to access each data source.
The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

JDBC drivers implement the defined interfaces in the JDBC API, for interacting with specific database server.
Provided by third party database vendor/providers.
Third party vendors implements the java.sql.Driver (defined by JDBC specs) interface in their database driver.

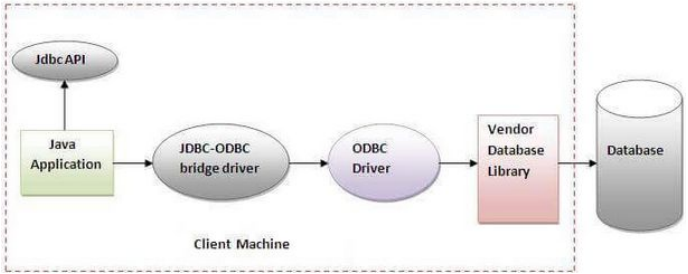
4 categories - Types 1, 2, 3, and 4
Type 1: JDBC-ODBC Bridge Driver
Type 2: JDBC-Native API
Type 3: JDBC-Net Pure Java
Type 4: 100% Pure Java

JDBC Driver

https://www.javatpoint.com/jdbc-driver

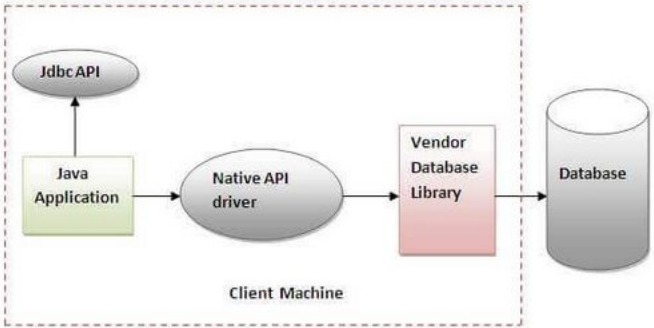
1) JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. Lowest Performance. This is now discouraged because of thin driver.



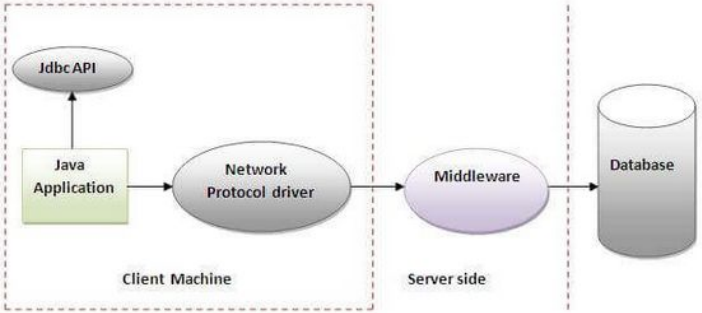
2) Native-API driver

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. The Vendor client library needs to be installed on client machine. Performance better than 1st.



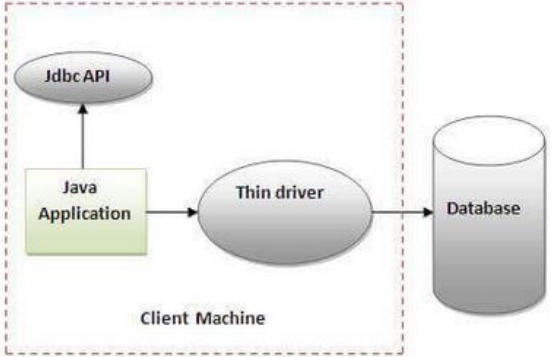
3) Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protoco

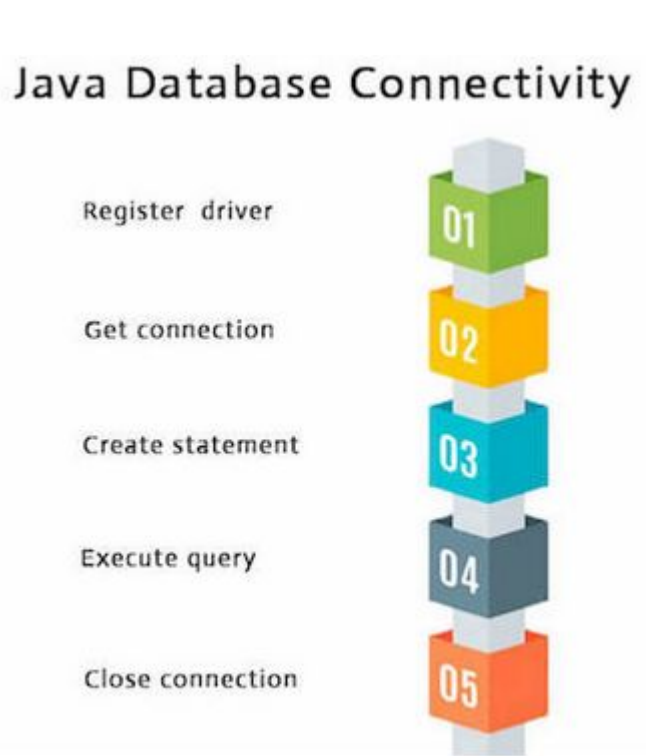


4) Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. **Best performance and preferred (if available)**



Steps



https://www.javatpoint.com/steps-to-connect-to-the-database-in-java

Class.forName("oracle.jdbc.driver.OracleDriver");

This method is used to dynamically load the driver class.

Note: Since JDBC 4.0, explicitly registering the driver is optional. We just need to put vender's Jar in the classpath, and then JDBC driver manager can detect and load the driver automatically.

public static Connection getConnection(String url,String name,String password)

Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","password");

public Statement createStatement()throws SQLException

Statement stmt=con.createStatement();

public ResultSet executeQuery(String sql)throws SQLException

ResultSet rs=stmt.executeQuery("select * from emp");

public void close()throws SQLException

con.close();

Steps: Example & Important Class/Interface

Example

```
import java.sql.*;
class MysqlCon{
public static void main(String args[]){
try{
Class.forName("com.mysql.jdbc.Driver");
Connection con=DriverManager.getConnection(
"jdbc:mysql://localhost:3306/sonoo","root","root");
//here sonoo is database name, root is username and password
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
con.close();
}catch(Exception e){ System.out.println(e);}
}
}
```

DriverManager: This class **manages a list of database drivers**. Matches connection requests from the java application with the proper database driver using communication subprotocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.

Driver: This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.

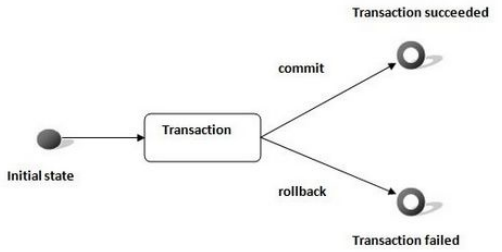
Connection: This interface with all methods for contacting a database. The connection object represents **communication context**, i.e., all communication with database is through connection object only.

Statement: You use objects created from this interface **to submit the SQL statements** to the database. Some derived interfaces accept parameters in addition to executing stored procedures.

ResultSet: These class objects **hold data retrieved from a database** after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.

SQLException: This class handles any errors that occur in a database application

JDBC: Transaction



In JDBC, **Connection interface** provides methods to manage transaction.

Method	Description
void setAutoCommit(boolean status)	It is true bydefault means each transaction is committed bydefault.
void commit()	commits the transaction.
void rollback()	cancels the transaction.

```
import java.sql.*;
class FetchRecords{
public static void main(String args[])throws Exception{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
con.setAutoCommit(false);

Statement stmt=con.createStatement();
stmt.executeUpdate("insert into user420 values(190,'abhi',40000)");
stmt.executeUpdate("insert into user420 values(191,'umesh',50000)");

con.commit();
con.close();
}}
```

With ORM, JAVA developers needn't directly use JDBC

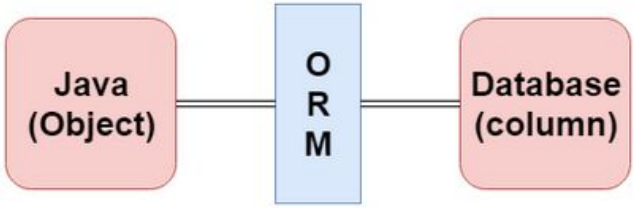
More Details: https://www.javatpoint.com/java-jdbc

https://www.youtube.com/watch?v=JR7-EdxDSf0

ORM: What & Why

What:

It stands for **Object Relation Mapping.** ORM is a **technique** for converting data between Java objects and relational databases (table). In simple words, we can say that the **ORM implements responsibility of mapping the object to relational model and vice-versa**. the ORM tool does mapping in such a way that **model class** becomes a table in the database and each instance becomes a row of the table.



Why ORM

1. Java Applications works on Objects, but to do any database operations with JDBC alone they need deal with conversion of Java Objects to/from SQL statements/outputs themselves.

   ORM provide technique for JAVA objects to work directly with objects instead of using SQL statements.

Query example: Application to form query & needs to map response to

```
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3));
```

ORM: Class <-> Table

Class name = Table name
Class Variable name = Column name
Objects = Row

JAVA Class

```
@Entity
@Table(name="student")
public class StudentEntity {

  @Id
  private int s_id;
  private String s_name;
  private int s_age;
```

Objects

```
StudentEntity s1=new StudentEntity();
s1.setS_id(101);
s1.setS_name("Gaurav");
s1.setS_age(24);

StudentEntity s2=new StudentEntity();
s2.setS_id(102);
s2.setS_name("Ronit");
s2.setS_age(22);

StudentEntity s3=new StudentEntity();
s3.setS_id(103);
s3.setS_name("Rahul");
s3.setS_age(26);
```

RDBMS Table

S_ID	S_NAME	S_AGE
101	Gaurav	24
102	Ronit	22
103	Rahul	26

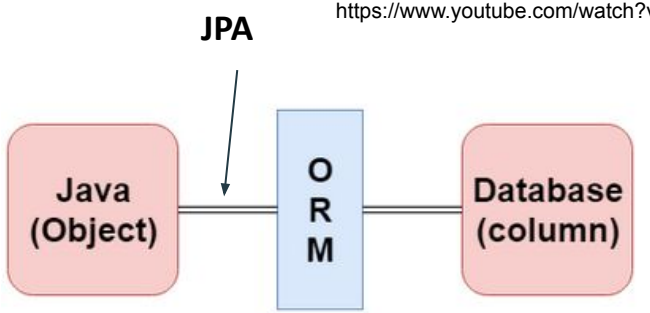
ORM: Implementations

JPA: What & Why

https://www.youtube.com/watch?v=otinfgwkMbY

What:

It stands for **Java Persistence API.** JPA is just a specification that facilitates object-relational mapping to manage relational data in Java applications. As JPA is just a specification, it doesn't perform any operation by itself. It requires an implementation. So, ORM tools like Hibernate, TopLink and iBatis implements JPA specifications for data persistence.



Why JPA

1. To provide loose coupling between application and ORM implementations in order to provide application flexibility to migrate to different ORM tools without code change. Also with this app developer only need to learn JPA, as JPA has provided abstraction over specific implementations.

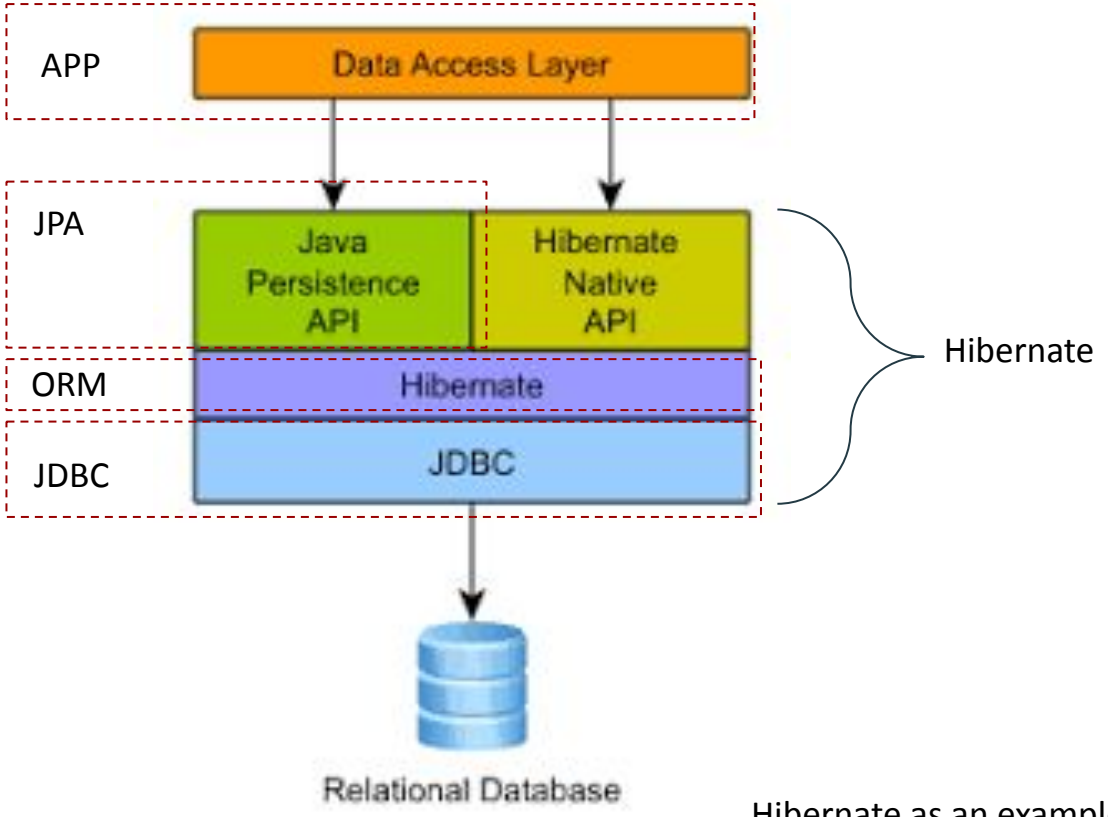
Hibernate

```
SessionFactory factory = meta.getSessionFactoryBuilder().build();
Session session = factory.openSession();
Transaction t = session.beginTransaction();

  Employee e1=new Employee();
  e1.setId(101);
  e1.setFirstName("Gaurav");
  e1.setLastName("Chawla");

  session.save(e1);
  t.commit();
  System.out.println("successfully saved");
  factory.close();
  session.close();

}
```

iBatis

```
public class UserDaoIbatis implements UserDao
{
  @Override
  public UserTEO addUser(UserTEO user, SqlMapClient sqlmapClient) {
    try
    {
      Integer id = (Integer)sqlmapClient.queryForObject("user.getMaxId");
      id = id == null ? 1 : id + 1;
      user.setId(id);
      user.setStatus(1);
      sqlmapClient.insert("user.addUser", user);
      user = getUserById(id, sqlmapClient);
      return user;
    }
    catch(Exception e)
    {
      e.printStackTrace();
    }
    return null;
  }
```

Working together: JPA + ORM + JDBC



Hibernate as an example

Entity

An entity represents a table in a relational database, and each entity instance corresponds to a row in that table. **The primary programming artifact of an entity is the entity class**

A Java class can be easily transformed into an entity class. For transformation the basic requirements are: -

- No-argument Constructor
- Annotation

@Entity - This is a marker annotation which indicates that this class is an entity. This annotation must be placed on the class name.

@Id - This annotation is placed on a specific field that holds the persistent identifying properties. This field is treated as a primary key in database.

```
import javax.persistence.*;
@Entity
public class Student {
  @Id
  private int id;
  private String name;
  private long fees;
  public Student() {}
  public Student(int id)
  {
    this.id = id;
    }
  public int getId()
  {
    return id;
    }
  public void setId(int id)
  {
    this.id = id;
    }
  public String getName()
  {
    return name;
    }
  public void setName(String name)
  {
    this.name = name;
    }
```

Mapping: OneToOne

4 types

1. OneToOne
2. OneToMany
3. ManyToOne
4. ManyToMany

In this example, we will create a One-To-One relationship between a Student and Library in such a way that one student can be issued only one type of book.

```
package com.javatpoint.mapping;
import javax.persistence.*;

@Entity
public class Library {
  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
private int b_id;
private String b_name;

@OneToOne
private Student stud;
```

```
import javax.persistence.*;

@Entity
public class Student {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private int s_id;
  private String s_name;
  public int getS_id() {
    return s_id;
  }
  public void setS_id(int s_id) {
    this.s_id = s_id;
  }
  public String getS_name() {
    return s_name;
  }
  public void setS_name(String s_name) {
    this.s_name = s_name;
  }
```

- Student table - This table contains the student details. To fetch data, run **select * from student** query in MySQL.

S_ID	S_NAME
1	Vipul
2	Vimal

- Library table - This table represents the mapping between student and library. To fetch data, run **select * from library** query in MySQL.

B_ID	B_NAME	STUD_S_ID
101	Data Structure	1 [->]
102	DBMS	2 [->]

Note: Instead of Library it should be Book class in this example.

Mapping: OneToMany

In this example, we will create a **One-To-Many** relationship between a Student and Library in such a way that one student can be issued more than one type of book.

```java
import javax.persistence.*;

@Entity
public class Student {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private int s_id;
  private String s_name;

  @OneToMany(targetEntity=Library.class)
  private List books_issued;

  public List getBooks_issued() {
    return books_issued;
  }
  public void setBooks_issued(List books_issued) {
    this.books_issued = books_issued;
  }
  public int getS_id() {
    return s_id;
  }
```

```java
@Entity
public class Library {
  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
private int b_id;
private String b_name;

public Library(int b_id, String b_name) {
  super();
  this.b_id = b_id;
  this.b_name = b_name;

}

public Library() {
  super();
  // TODO Auto-generated constructor stub
}

public int getB_id() {
  return b_id;
}
```

- Student table - This table contains the student details. To fetch data, run **select * from student** query in MySQL.

S_ID	S_NAME
1	Vipul

- Library Table - This table contains the library book details. To fetch data, run **select * from library** query in MySQL.

B_ID	B_NAME
101	Data Structure
102	DBMS

- Student_library table - This table represents the mapping between student and library table. To fetch data, run **select * from student_library** query in MySQL.

Student_S_ID	books_issued_B_ID
1 [->]	101 [->]
1 [->]	102 [->]

Mapping: ManyToOne

In this example, we will create a Many-To-One relationship between a Student and Library in such a way that more than one student can issued the same book.

```java
import javax.persistence.*;

@Entity
public class Student {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private int s_id;
  private String s_name;

  @ManyToOne
  private Library lib;

  public int getS_id() {
    return s_id;
  }
  public void setS_id(int s_id) {
    this.s_id = s_id;
  }
  public String getS_name() {
    return s_name;
  }
```

```java
@Entity
public class Library {
  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private int b_id;
  private String b_name;

public Library(int b_id, String b_name) {
  super();
  this.b_id = b_id;
  this.b_name = b_name;

}

public Library() {
  super();
  // TODO Auto-generated constructor stub
}
```

- Library table - This table contains the library details. To fetch data, run **select * from library** query in MySQL.

B_ID	B_NAME
101	Data Structure

- Student table - This table represents the mapping between student and library. To fetch data, run **select * from student** query in MySQL.

S_ID	S_NAME	LIB_B_ID
1	Vipul	101 [->]
2	Vimal	101 [->]

Mapping: ManyToMany

In this example, we will create a Many-To-Many relationship between a Student and Library in such a way that any number of students can be issued any type of books.

```java
import javax.persistence.*;

@Entity
public class Student {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private int s_id;
  private String s_name;

  @ManyToMany(targetEntity=Library.class)
  private List lib;
```

```java
import javax.persistence.*;

@Entity
public class Library {
  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
private int b_id;
private String b_name;

@ManyToMany(targetEntity=Student.class)
private List stud;
```

After the execution of the program, three tables are generated under MySQL workbench.

- Student table - This table contains the student details. To fetch data, run **select * from student** query in MySQL.

S_ID	S_NAME
1	Vipul
2	Vimal

- Library table - This table contains the library details. To fetch data, run **select * from library** query in MySQL.

B_ID	B_NAME
101	Data Structure
102	DBMS

- Library_student - This table contains the library details. To fetch data, run **select * from library_student** query in MySQL.

Library_B_ID	stud_S_ID
101 [->]	1 [->]
102 [->]	1 [->]
101 [->]	2 [->]
102 [->]	2 [->]

JPA & JAVA Collections

```java
import javax.persistence.*;
@Entity

public class Employee {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private int e_id;
  private String e_name;

  @ElementCollection
  private List<Address> address=new ArrayList<Address>();


  public int getE_id() {
    return e_id;
  }
```

```java
@Embeddable
public class Address {

  private int e_pincode;
  private String e_city;
  private String e_state;
  public int getE_pincode() {
    return e_pincode;
  }
  public void setE_pincode(int e_pincode) {
    this.e_pincode = e_pincode;
  }
  public String getE_city() {
    return e_city;
  }
```

This is for List, Set is similar

- Employee table - This table contains the employee details. To fetch data, run select * from employee

E_ID	E_NAME
1	Vijay
2	John

- Employee_address table - This table represents the mapping between employee and address table. query in MySQL.

E_CITY	E_PINCODE	E_STATE	Employee_E_ID
Noida	201301	Uttar Pradesh	1 [->]
Jaipur	302001	Rajasthan	2 [->]

JPA & JAVA Collections

```
import javax.persistence.*;
@Entity

public class Employee {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private int e_id;
  private String e_name;

  @ElementCollection
  private Map<Integer,Address> map=new HashMap<Integer,Address>();
```

```
@Embeddable
public class Address {

  private int e_pincode;
  private String e_city;
  private String e_state;
  public int getE_pincode() {
    return e_pincode;
  }
```

- Employee table - This table contains the employee details. To fetch data, run **select * from employe**

E_ID	E_NAME
1	Vijay
2	Vijay
3	William
4	Rahul

- Employee_map table - This table represents the mapping between employee and address table. manner.To fetch data, run select * from employee_map query in MySQL.

MAP_KEY	E_CITY	E_PINCODE	E_STATE	Employee_E_ID
3	Chandigarh	133301	Punjab	3 [->]
2	Jaipur	302001	Rajasthan	2 [->]
1	Noida	201301	Uttar Pradesh	1 [->]
4	Patna	80001	Bihar	4 [->]

Steps

- Create an entity class
- Now, map the entity class and other databases confiuguration in Persistence.xml file.
- Create a class which persist object using JPA's entity manager.

Persistence.xml

```
<persistence>
<persistence-unit name="Student_details">

  <class>com.javatpoint.jpa.student.StudentEntity</class>

<properties>
<property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
<property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/studentdata"/>
<property name="javax.persistence.jdbc.user" value="root"/>
<property name="javax.persistence.jdbc.password" value=""/>
<property name="eclipselink.logging.level" value="SEVERE"/>
<property name="eclipselink.ddl-generation" value="create-or-extend-tables"/>
</properties>

  </persistence-unit>
</persistence>
```

Steps

```
import com.javatpoint.jpa.student.*;
import javax.persistence.*;
public class PersistStudent {

public static void main(String args[])
{
        EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("Student_details");
        EntityManager em=emf.createEntityManager();

        em.getTransaction().begin();

        StudentEntity s1=new StudentEntity();
        s1.setS_id(101);
        s1.setS_name("Gaurav");
        s1.setS_age(24);

        StudentEntity s2=new StudentEntity();
        s2.setS_id(102);
        s2.setS_name("Ronit");
        s2.setS_age(22);

        StudentEntity s3=new StudentEntity();
        s3.setS_id(103);
        s3.setS_name("Rahul");
        s3.setS_age(26);

        em.persist(s1);
        em.persist(s2);
        em.persist(s3);

        em.getTransaction().commit();

        emf.close();
        em.close();

        }
}
```

DB Operations

Remove

```
  public static void main(String args[])
  {
  EntityManagerFactory emf=Persistence.createEntityManagerFactory("Student_details");
  EntityManager em=emf.createEntityManager();
em.getTransaction().begin();

  StudentEntity s=em.find(StudentEntity.class,102);
em.remove(s);
em.getTransaction().commit();
emf.close();
em.close();

  }
}
```

Query

```
StudentEntity s=em.find(StudentEntity.class,101);
```

Insert

```
    em.persist(s1);
    em.persist(s2);
    em.persist(s3);

em.getTransaction().commit();
```

Update??

Thanks

SQL Command Overview

DML (Data Manipulation Language): In DML, we try to manipulate the data. Commands like ***INSERT*** and ***UPDATE*** are part of this sub-language.

DDL (Data Definition Language): In DDL, we define the data structure and relation between them. Commands like ***CREATE*** are part of this.

DCL (Data Control Language): In DCL, we control access to data. This is used for permission management and who can access the data. Commands like ***GRANT*** and ***REVOKE*** are part of this sub-language.

TCL (Transaction Control Language): In TCL we control the flow in a transaction. ***COMMIT*** and ***ROLLBACK*** are part of this sub language.

DQL (Data Query Language): In DQL, we query the data. ***SELECT*** command is part of DQL.

Below is the table of commands that exist in these sub-languages.

DDL	DML	DQL	TCL	DCL
CREATE	INSERT	SELECT	COMMIT	GRANT
DROP	UPDATE		ROLLBACK	REVOKE
RENAME	DELETE		SAVEPOINT	
TRUNCATE				
ALTER				