DAY 1 Maven Intro
:===

Build tool : Maven
===============

Build Process in java based project/application management:

-

develpers duty/responsibility to develop a java based business application:

1. write some source code(bunch of classes, inteface...)

2.add some external jarfiles to the classpath (dependencies of our application)
in JDBC dao project, driver jar file is the dependency of our project.

3.compile the code.

4.prepare some test cases. (Unit test, Junit, Mockito, Sonar)

5.add Junit/mockito related jar files inside the classpath

6.compile and run the test cases.

7.arrange our code in a standard folder structure

Java based Webapplication:
----

*.java
*.class
*.jar
*.html
*.css
*.js
.*xml files
*.mp3, mp4, jpg, gif

8.Do the packaging : build the jar, war file.

9.deploy this jar file/ war file to the server.

--if any mistake identified then developer performs above task again and again.

build process: keeping our project ready for execution/release is called build process.

--manually doing this build process will increase the burden on developer

--to automate this build process of java applicaiton , apache foundation released a tool called ANT (Another Neet Tool).

the problem with ANT tool is :

1. ANT does not have the capabilities of donwloading the required jar files from internate automatically.

2. It can not prepare,compile and run the test cases automatically.

3.In ANT we need to write a build.xml file which is very lenghty xml file.

As a solution Apache foundation released another build tool called Maven.

--Gradle

--we can develop java application using Maven by 2 ways:

1.using console based (here we need to download the maven s/w and install it in our
local computer ,and then we need to set path and some environment variable to
develop java application)

--after that using some command in command prompt we can generate the template of our java application without any IDE.

2.using IDE: here we don't need to install any kind of extra s/w.
STS has inbuilt support of Maven and Gradle

Maven terminology:
===============

1. artifact :-

--An artifact is an outcome in maven, it can be a file, .class file or a jar file, war file,ear file,etc.

2. archetype:-

--it is project template for creating similar type of project in maven.

3. Groupid:

--it is an Id used to identify the artifacts of a perticular organization (naming convention is similar to package name.) com.masai

4.artiactid : it is the id for the final outcome (artifactid name will be the root folder of our application)

5.pom.xml : (project object model) : all the information will be their in this file.

--in this model, entire application itself will be considered as an object.

it defines following properties for a project:

1. Name
2.version

3. packaging (jar, war, ear)

4.dependecnies : required jar files.

5.plugins: will inhance the functionality of our project.

docker
jenkins (CI/CD)

Maven Repository:

-- a repository is a store where maven maintains plugins, archetypes, and lots of jar files used for building different kinds of java projects.

maven repo are of 3 types:

1.central repo :- it is the maven's own repo in which it maintains all kinds of project related plugins, archetypes, jars etc.

https://repo1.maven.org/maven2

2.remote repo : this repo is maintained within the organization for sharing plugins, archetypes and jar files for multiple projects withing orgnizations. ex: masairepo

3.local repo: this repo will be created inside the developer computer. (.m2) is the name for this repo.

mysql-connector.jar :

pom.xml:

.m2 ---

maven build life cycle:

maven build life cycle contains diff phases:

1. validate: - in this phase it will verify the project diectory structure is valid or not. and it has pom.xml file is there or not.

2.compile: maven compiles all the source code of the project by downloading and adding requied jar files in the classpath.

3.test-compile: if we have written any unit test cases those code will be compiled.

4.test : maven will run all the test cases and it will show how many test cases are success and how many fails.

5.package : maven will bundle our java code into a jar file inside 'target' folder.

6.install : that jar file in step 5 will be stored in the localrepo.

7.deploy : maven stores the application jar file to the central repo.

**8.clean : here maven will delete and remove all the files that are generated in previous build. this phase is an isolated phase.

Note: if we execute any phase to build the maven project then maven will execute all the phases till that given phase.except phase 8.clean.

```
>mvn clean   :- remove and delete previous build
>mvn test   : till the test phase.
>mvn deploy:
>mvn validate:
```

--package.json ---- pom.xml

src/main/java : -- here we need to place our source code

src/main/resources : any xml file, properties files, text files

src/test/java : -- here we need to place our source code to unit testing

src/test/resources : any xml file, properties files, text files to unit testing

after creating a maven project we need to change the java version of the maven project from jdk 5 to jdk 8

search in google: "maven compiler source"

```
 <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
```

after pasting the above tag inside the pom.xml file we need to update our maven project:

right click on the project--->maven ---> update maven project.(select force update)

--to build our maven application : --->right click on the project----> Run as--->Maven build... -----> inside Goal type : clean package ----> run

DAY 2 Layered Arch and ORM
==

Layared Architecture in Business application:-
====================================

1.maintaining the business data in secure and easily retrival manner.

--the logic that we write to implement this part of business application is known as Data Accees Logic.

2.processing the data according to the business rule .

--the logic that we write to implement this part of business application is known as Business/Service logic.

3.presenting the data to the user in user-understandable format.

--the logic that we write to implement this part of business application is known as Presentation logic.

--the above 3 logics is required for almost every business application.

Note:- we can write all the 3 logics to develop a business application in one program/class itself , if we do so, the following problem we will face:-

1.all the logics to develop the application will be mixed up with each other (no clear code seperation).

2.modification done in one logic may affect the other logic .

3. logics will depend upon each other so, parellal developement will not be possible.

4.testing each logic is also will become complex..

--to solve this problem, a java based business application ,we devide into 3 logical partition .

and each part we say as a layer:-

1.Presentation Layer(UI layer)

2.Business Logic Layer (Service layer)

3.Data Access Layer

--a bussiness application will be devided into the logical partition depending upon the role played by each part.

--logical partition of a business application is known as layer.

Presentation Layer :-

--it is set of java classes, which are responsible for generating user input screen and response page(output screen) is knwon as PL.

--this layer provides the intraction with the end-user.

Business Logic Layer/Service Layer:-

--programatical implementation of business rule of a business organization is nothing but business logic .
--a collection of java classes whose methods have business logic to process the data according to the business rule is known as SL/BLL.

Data Access Layer :-

--a set of java classes whose methods r exclusivly meant for performing CRUD operation with the DB server is known as DAL.

using JDBC and DAO pattern

**Note:- to communicate among these layers loose coupling should be promoted.

Developing Data Access Layer using ORM (Object Relational mapping) approach:-
===

Java persistence:-

--the process of saving/storing java obj's state into the DB s/w is known as java persistence..

--for small application we can store business data (java object state) in the files using IO streams (serialization and deserailization approach).

--the logic that write to store java objs(which is holding business data) into the file using IO Streams is known as
 "IO stream based persistence logic".

--but in the realtime application, we store/save/persist the business data inside the database using JDBC

public String saveStudentDetails(Student student)

--the logic that we write to store java objs data into the DB using JDBC is known as "Jdbc based persistence logic"

limitation of JDBC based persistence logic:-

1.jdbc can't store the java objs into the table directly,becoz sql queires does not allows the java objs as input, here we need to convert obj data into the simple(atmoic) value to store them in a DB.

2.jdbc code is the DB dependent code becoz it uses DB s/w dependent queries. so our jdbc based persistence logic is not 100% portable across various DB s/w.

3.jdbc code having boiler plate code problem (writing the same code except sql queries in multiple classes of our application again and again)..

4.jdbc code throws lots of checked exceptions, programmer need to handle them.

5.After the select operation, we get the ResultSet object.this RS obj we can not transfer from one layer to another layer and to get the data from the ResultSet we need to now the structure of the ResultSet.

6.there is no any caching and transaction management support is available in jdbc.

etc ...

--to overcome the above limitations we need to use ORM approach.

ORM (Object - ralation mapping):- Java---->   relation
============================
--the process of mapping java classes with the DB tables ,, java class member variables with the DB table columns and
making the object of java class reperesents the DB table records having synchronization bt them is called a OR mapping.

```
student(roll, name, marks);

class Student{
roll;
name;
marks

}

Student s1=new Student(10,"Ram",500);
```

Student object----------->row of the student table

-here synchronization bt obj and table row is nothing but, the modification done in the obj will reflect the DB table row and vise-versa.

***the logic that we write to store java objs into the DB using ORM approach is called as ORM based persistence logic.

--there are variaous ORM s/w are available in the market, these s/w will act as f/w software to perform ORM based persistence logic.

ex:-

Hibernate ***
Toplink
Ibatis,
Eclipselink
etc...

f/w software :-

--it is a special type of s/w that provides abstraction layer on one or more existing core technology to simplyfy the process of application development.

--in java most of the f/w softwares comes in the form of jar files(one or more jar file)

--in order to use/work on these f/w softwares we need to add those jar files in our classpath.

--while working with the ORM based persistence logic we write all the logics in the form of objs without any sql quiery support. due to which our logic will become DB s/w independent logic.

--In ORM based logic, the ORM s/w takes objs as an input and gives objs as an output so no need to convert object data to the primitive values.

--ORM s/w addresses the mismatches bt object oriented representation of data and relational representation of data.

Object <----> tables

class

1.inheritence mismatches / IS-A mismatch

2.Granularity mismatch / has-A mismatch

3. assotiation mismatch / table relation mismatch

i.e for processing and presenting the data, we represent the data in form of ObjectOriented fashion
whereas for storing the data we represnet the data in the form of relational fashion(in the tables)

```
class Student{

int roll,
String name,
int marks,
Address addr; (city,state,pin)

}

class Address{


}
```

student table :-

```
roll int
name varchar
marks int
```

address table :-

--one obj of Student class will represnt one row of student table

ORM s/w addresses these mismathces in very easy manner.

POJO class:- java bean class:

Plain old java object

--it is a normal java class not bounded with any technology or f/w s/ws.
i.e a java class that is not implementing or extending technology/framework api related
classes or interfaces.

--a java class that can be compiled without adding any extra jar files in the classpath are known
as a POJO class.

POJI (plain old java interface)

Note:- every java bean class is a POJO but every POJO is not a java bean.

```
public class X {

public X(int x){

}

}
```

--the following class comes under the category of POJO class

```
class X implments Serializable, Runnable{

}
```

--this class does not comes under POJO

```
class X implements Servlet{

}
```

--above class is a POJO class but it is not a java bean class.becoz of parameterized constructor.

ORM s/w features:-
===============

1.it can persist/store java obj to the DB directly.

2.it supports POJO and POJI model

3.it is a lightweight s/w becoz to excute the ORM based application we need not install any kind of servers.

4.ORM persistance logic is DB independent. it is portable accross multiple DB s/ws.
(beocz here we deal with object, not with the sql queires)

5.prevent the developers from boiler plate code coding to perform CRUD operarions.

6.it generates fine tuned sql statements internally that improves the performance.

7.it provides caching mechanism (maintaing one local copy to inhance the performance)

8.it provides implicit connection pooling.

9.exception handling is optional becoz it throws unchecked exceptions.

10.it has a special Query language called JPQL (JPA query language) that totally depends upon the ' objects.   select eid,ename from employee.

```
sql> select roll, name, marks from student;

jpql> select roll, name,marks from Student;
```

--in sql we write the query in the term of tables and columns whereas in JPQL we write the Query in the term of classes and variables.

Java Persistence API: it is a standard api using which we can
work with any kind of ORM s/w.

Hibernate has thier own api also,

Hibernate : ORM ---> diff classes , diff methods, diff interfaces

Ibatis : ORM ---> diff classes , diff methods, diff interfaces

sun-microsystem : JPA api (standard api)

Hibernate : JPA api

Ibatis: JPA api

Hibernate and JPA:-

JPA is a specification and Hibernate is its one of the famous implementation.

Hibernate:- it is one of the ORM based framework s/w. other s/w are :-
toplink,ibatis,etc..

JPA:- (Java persistence api) :- it is an open specification given by Oracle corp, to develop any ORM based s/w .

JPA provides a standard api to work with any kind of ORM based s/w .

JPA api belongs from "javax.persistence" package.

--Hibernate is one of the most frequently used JPA implementation

--HB provides its own api to develop ORM based persistence logic , if we use those api then
our application will become vendor lock, ie we can not port our application accross multiple
ORM s/w.

--HB api comes in the form of "org.hibernate" package.

Note:- we get the JPA api , along with any ORM s/w , becoz all the ORM s/w implements
JPA specification.

java.sql
javax.sql   this jdbc api comes along with jdk installation

Jpa with Hibernate:-

JPA Application:-

any java application, that uses JPA api to perform persistnce operation (CRUD) operation with
the DB s/w is called as JPA application.

JPA architecture:-

Entity class or persistence class:-

--it is a class using which we map our table.

--if we are using the annotaion, then we need not map this class with the table inside the xml mapping file.

--an Entity class or persistence class is a java class that is developed corresponding to a table of DB.

--this class has many instance variables should be there as same as columns in the corresponding table

--we should take Entity class as a POJO class.

--we need to provide mapping information with the table in this class only using annotaitons.

Note:- when we gives this persistance /Entity class obj to the ORM s/w, then ORM s/w will
decide the destination DB s/w based on the configuration done in a xml file which is called as hibernate-configuration file.

Configuration file:-

--it is an xml file its name is "persistence.xml".

--this file must be created under src/META-INF folder in normal java application, where as in maven or gradle based application this file should be inside the src/main/resources/META-INF folder

--this file content will be used by ORM s/w (ORM engine) to locate the destination DB s/w.

--in this file generally 3 types of details we specify:-

1.DB connection details

2.ORM specific details (some instruction to the ORM s/w like dialect info,show_sql ,etc)

3. annotation based entity/persistence class name.(optional from latest hibernate version)

Note:- generally we take this file 1 per DB basis.

--we should always create this configuration file by taking support of example applications inside
the project folder of hibernate download zip file or by taking the reffernce from the Google.
ex:-

persistence.xml:-

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">


    <persistence-unit name="studentUnit" >

      <class>com.ratan.Student</class>

<properties>

           <property name="javax.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver" />
          <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/ratandb" />
            <property name="javax.persistence.jdbc.user" value="root" />
            <property name="javax.persistence.jdbc.password" value="root"
/>

   /*
           <property name="hibernate.connection.driver_class"
value="com.mysql.cj.jdbc.Driver"/>
            <property name="hibernate.connection.username" value="root"/>
            <property name="hibernate.connection.password"
value="root"/>
            <property name="hibernate.connection.url"
value="jdbc:mysql://localhost:3306/ratandb"/>
     */



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

the root tag is :-

<persistence> with some xml-namespace

--the child tag of <persistence> tag is <persistence-unit>

--this <persistence-unit> has 2 child tags:-

1. <class> tag ,:-using which we specify the Entity class name(fully qualified name) that used
annotations to map a table (optional from letest version of hibernate)

2.<properties> tag :- using this tag,we specify some configuration details to the ORM s/w

Persistence-unit:- it is a logical name of the configuation of our DB and some other details.

DAY 3 JPA and Hibernate
==
=

Develop Data Access Layer of a Business application using ORM approach.

limitations of JDBC approach

feature of ORM s/w

ORM s/w :

Hiberate
Toplink
Ibatis
EclipseLink

JPA --specification and api : it is implemented by all the ORM s/w

javax.persistence

org.hibernate

JPA architecture :

How to get the Hibernate s/w:
========================

1. download the hiberate s/w (zip file) and add the required jar file in the classpath of our project

2.maven approach:

hibernate-core jar file

persistence.xml : take this file from sample application or from hibernate docs..
and modify it accordingly.

ORM engine :-

--it is a specialized s/w written in java that performs translation of jpa calls into the sql call by using mapping annotation and configuration file details and send the mapped sql to the DB s/w using JDBC.

--ORM engine is provided by any ORM s/w.

steps to devlop the JPA application:-

1.create a maven project and add the hibernate-core dependency to the pom.xml.

2.add jdbc driver related dependency to the pom.xml

3.create a folder called "META-INF" inside src/main/resources folder, and create the "persistence.xml" file inside this folder by taking reference from Hibernate docs or from google.

example:

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">

    <persistence-unit name="studentUnit" >


<properties>

            <property name="javax.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver" />
           <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/ratandb" />
             <property name="javax.persistence.jdbc.user" value="root" />
             <property name="javax.persistence.jdbc.password" value="root"
/>


        </properties>

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

step 4:- create as many Entity/Perssitence classes as there r tables in the DB, apply the at least 2 annotations to these classes

@Entity :- on the top of the class
@Id :- on the top of PK mapped variable

--if we apply above 2 annotations then our java bean class will become Entity or Persistence class.

--inside these classes , we need to take variable corresponding to the columns of the tables.

11:06 am

step 5:- create a client application and activate ORM engine by using JPA api related following classes and interface and perform the DB operations.

1.Persistence class

2.EntityManagerFactory

3.EntityManager

--if we use Hibernate core api then we need to use

Configuration class

SessionFactory(I)

Session(I)

example :

Student.java: // Entity class

```
package com.masai;

import javax.persistence.Entity;
import javax.persistence.Id;

@Entity
public class Student {

      @Id
      private int roll;

      private int marks;
      private String name;

      public Student() {
            // TODO Auto-generated constructor stub
      }

      public Student(int roll, String name, int marks) {
```

```
		super();
		this.roll = roll;
		this.name = name;
		this.marks = marks;
	}

	public int getRoll() {
		return roll;
	}

	public void setRoll(int roll) {
		this.roll = roll;
	}

	public String getName() {
		return name;
	}

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

	public int getMarks() {
		return marks;
	}

	public void setMarks(int marks) {
		this.marks = marks;
	}

	@Override
	public String toString() {
		return "Student [roll=" + roll + ", name=" + name + ", marks="
+ marks + "]";
	}

}
```

Demo.java:

```
package com.masai;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class Demo {

	public static void main(String[] args) {

		EntityManagerFactory emf=
Persistence.createEntityManagerFactory("studentUnit");


		EntityManager em= emf.createEntityManager();


		Student student= em.find(Student.class, 20);
```

```
            if(student != null)
                  System.out.println(student);
            else
                  System.out.println("Student does not exist");

            em.close();


      }

}
```

Note:- when we call createEntityManagerFactory(-) method by suppliying persistence-unit name on the Persistence class,we will get the EntityManagerFactory object.

--this method loads the "persistence.xml" file into the memory

--EntityManagerFactory obj should be only one per application per DB.

this EMF obj contains :-

connection pool (readly available some jdbc connection obj)

some meta information

--by using this EMF class only we create the EntityManager object.
--EMF is a heavy weight object, it should be one per application

```
EntityManager em= emf.createEntityManager();
```

Note:- inside every DAO method(for every use case) we need to get the EntityManager obj
--after performing the DB opeation for that use-case we should close the EM obj.

EM should be one per use-case (one per DAO method)

JPA application ----------------->EntityManager(I) --------------------->ORM engine ------>JDBC----------->DB s/w

--in order to perform any DML (insert update delete) the method calls should be in a transactional area.

em.getTransaction(); method return
"javax.persistice.EntityTransaction(I) " object.

this EntityTransaction obj is a singleton object, i.e per EntityManager obj, only one Transaction object is created.

--to store the object we need to call persist(-) method on the EM object.

example:

Demo.java:

```
package com.masai;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class Demo {

	public static void main(String[] args) {

		EntityManagerFactory emf=
Persistence.createEntityManagerFactory("studentUnit");


		EntityManager em= emf.createEntityManager();

		Student student= new Student(30, "Ratan", 500);


//		EntityTransaction et= em.getTransaction();
//
//		et.begin();
//
//		em.persist(student);
//
//		et.commit();


		em.getTransaction().begin();

		em.persist(student);

		em.getTransaction().commit();




		System.out.println("done...");




		em.close();
	}

}
```

--to get the Object from the DB we need to call :- find(--) method of EM object

this find(--) method takes 2 parameter

1.the Classname of the Object which we want,

2.the ID value for which we want the object.

ex:-

Main.java:- for Read object

```
public class Main {

	public static void main(String[] args) {

		EntityManagerFactory
emf=Persistence.createEntityManagerFactory("studentUnit");

		EntityManager em= emf.createEntityManager();

		Student s= em.find(Student.class, 60);

		if(s != null)
			System.out.println(s);
		else
			System.out.println("Student does not exit..");

		em.close();

	}

}
```

Main.java :-(insert object)

```
public class Main {

	public static void main(String[] args) {

		EntityManagerFactory
emf=Persistence.createEntityManagerFactory("studentUnit");

		EntityManager em= emf.createEntityManager();
```

```
            Student s=new Student(35, "Arun", 780);

      /*    EntityTransaction et= em.getTransaction();

            et.begin();

            em.persist(s);

            et.commit();
      */

            em.getTransaction().begin();

            em.persist(s);

            em.getTransaction().commit();

            em.close();

            System.out.println("done");


      }

}
```

Main.java:- Delete:-

```
public class Main {

      public static void main(String[] args) {

            EntityManagerFactory
emf=Persistence.createEntityManagerFactory("studentUnit");

            EntityManager em= emf.createEntityManager();


            Scanner sc=new Scanner(System.in);

            System.out.println("Enter roll to delete ");
            int roll=sc.nextInt();

            Student student= em.find(Student.class, roll);

            if(student != null){

                  em.getTransaction().begin();

                  em.remove(student);

                  em.getTransaction().commit();


                  System.out.println("Student removed....");

            }else
```

```
                System.out.println("Student not found...");

            em.close();


            System.out.println("done");


      }

}
```

Main.java :- Update the marks:-
============================

```
public class Main {

      public static void main(String[] args) {

            EntityManagerFactory
emf=Persistence.createEntityManagerFactory("studentUnit");

            EntityManager em= emf.createEntityManager();


            Scanner sc=new Scanner(System.in);

            System.out.println("Enter roll to give grace marks ");
            int roll=sc.nextInt();

            Student student=em.find(Student.class, roll); //if it returns
the obj then the obj will be in p.state


            if(student == null){
                  System.out.println("Student does not exist..");
            }
            else
            {

                  System.out.println("Enter the grace marks");
                  int marks=sc.nextInt();

                  em.getTransaction().begin();

                  student.setMarks(student.getMarks()+marks);

                  em.getTransaction().commit();

                  System.out.println("Marks is graced...");

            }
            em.close();

            System.out.println("done");
      }

}
```

--in the above application we didn't call any update method, we just change the state of the persistence/entity obj
inside the transactional area, at the end of the tx , ORM engine will generate the update sql.

--this is known as the ORM s/w maintaining synchronization bt entity obj and the db table records.

--we have a method called merge() inside the EntityManager obj to update a record also.

Life-cycle of persistence/entity object:-

an entity obj has the 3 life-cycle state:-

1.new state/transient state

2.persistence state/managed state

3.detached state

1.new state/transient state:-

--if we create a object of persistence class and this class is not attached with the EM obj, then
this stage is known as new state/transient state

```
Student s=new Student(10,"ram",780);
```

2.persistence state:-

--if a persistence class obj or Entity obj is associated with EM obj, then this obj will be in persistence state.

ex:-

when we call a persist(-) method by supplying Student entity obj then at time student obj will be in persistence state

or

when we call find() method and this method returns the Student obj, then that obj will also be in persistence state.

Note:- when an entity class obj is in persisitence state ,it is in-sync with the DB table ,i.e
any change made on that obj inside the tx area will reflect to the table automatically.

ex:-

```
            Student s=new Student(150,"manoj",850); // here student obj is in transient state .

            em.getTransaction().begin();

            em.persist(s); // here it is in the persistence state

            s.setMarks(900);


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

detached state:-

--when we call close() method   or call clear() method on the EM obj, then all the associated entity obj will be in detached state.

--in this stage the entity objs will not be in-sync with the table.

Note:- we have a merge() method in EM obj, when we call this method by supplying any detached object then that detached object will bring back in the persistence state.

ex:-

Main.java:-

```
public class Main {

      public static void main(String[] args) {

            EntityManagerFactory
emf=Persistence.createEntityManagerFactory("studentUnit");

            EntityManager em= emf.createEntityManager();

            Student s= em.find(Student.class, 20); //persistence state
            em.clear(); //detached state

            em.getTransaction().begin();

            s.setMarks(500);

            //em.persist(s);// it will throw duplicate ID related
exception
            em.merge(s); //persistence state

            em.getTransaction().commit();

            em.close();

            System.out.println("done");
      }
}
```

em.persist()
em.find()------------>persistence state-----------em.close(), em.clear()---------->detached state---->em.merge()--->reflect in the table.

--after merge() method, we can not do modification on that object(it will not be
reflected.).

Note:- to see the ORM tool(HB) generated sql queries on the console add the following property inside the persistence.xml

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

12:30 pm

to create or update the table according to the entity class mapping information:-

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

create :- drop the existing table then create a fresh new table and insert the record.

update :- if table is not there then create a new table, and if table is already there it will perform insert operation only in the existing table.

some of the annotations of JPA:-

@Entity :- to make a java bean class as entity , i.e to map with a table

@Id :- to make a field as the ID field (to map with PK of a table)

@Table(name="mystudents") :- if the table name and the class name is different

@Column(name="sname") :- if the column name of table and corresponding variable of the class is diff.

@Transient : it will ignore the filed value.

DAY 4 JPA and EntityManager

===

POJO class
Java Bean class
Entity class

POJO class: the class which can be compiled without adding any extra jar files in the classpath, that class should not implements or extends any
framework api related classes or interfaces.

ex1:
```
class X{  //it is a valid POJO

public X(int i){

}

}
```

ex2:

```
class X implements Runnable{ //yes becoz is a part of Plain JDK, no need to add any jar file

}
```

ex3

```
class X implements Servlet{ // non-pojo

}


@Entity
class X{   // now it is a non-pojo class

@Id
private int roll


}
```

Java Bean class:
----

pure encapsulated reusable component,

1.the class should be public.

2.the class must of zero argument constructor

3.if any fields are their then those should be private.

4. expose the private fields through public getter setter methods.

5.that class should implement Serializable interface

6. this class may have parameterized constrcutor,

7.this class may override equals() , hashCode(), toString()

8.may have some other methods also.

```
public class X implements Serializable{ // it is a Java Bean class, POJO
class also.


}
```

Generators in JPA:-

--Generators are used to generate the ID filed value automatically.

---> txID

```
      @Id
      @GeneratedValue(strategy=GenerationType.AUTO)
      private int roll;
```

--here roll will be generated automatically for each row.

**Note:- if we use this @GeneratedValue annoation then we are not allowed to give the roll explicitly while inserting a record.

--so we should create a object by using zero argument constrcutor and set the each value by calling setter method.

AUTO :- internally underlaying ORM s/w creates a table called "hibernate_sequence" to maintain the Id value.

IDENTITY :- it is used for auto_increatement feature to auto generate the id value

SEQUENCE :- it is used for sequence feature to auto generate the id value

DAO pattern example with JPA:-
========================

EMUtil.java:-

```
public class EMUtil {


      private static EntityManagerFactory emf;

      static{
            emf=Persistence.createEntityManagerFactory("account-unit");
      }

      public static EntityManager provideEntityManager(){

            //EntityManager em= emf.createEntityManager();
            //return em;

            return emf.createEntityManager();
      }
}
```

Account.java:- (Entity class)

```
@Entity
public class Account  {

      @Id
      @GeneratedValue(strategy = GenerationType.AUTO)
      private int accno;
      private String name;
      private int balance;


      public Account() {
            // TODO Auto-generated constructor stub
      }


      public int getAccno() {
            return accno;
      }
```

```
	public void setAccno(int accno) {
		this.accno = accno;
	}


	public String getName() {
		return name;
	}


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


	public int getBalance() {
		return balance;
	}


	public void setBalance(int balance) {
		this.balance = balance;
	}


	public Account(int accno, String name, int balance) {
		super();
		this.accno = accno;
		this.name = name;
		this.balance = balance;
	}


	@Override
	public String toString() {
		return "Account [accno=" + accno + ", name=" + name + ", 
balance="
				+ balance + "]";
	}

}
```

AccountDao.java:-(interface)

```
public interface AccountDao {

	public boolean createAccount(Account account);

	public boolean deleteAccount(int accno);

	public boolean updateAccount(Account account);

	public Account findAccount(int accno);

}
```

AccountDaoImpl.java:-

```java
public class AccountDaoImpl implements AccountDao{

      @Override
      public boolean createAccount(Account account) {

            boolean flag= false;

            EntityManager em= EMUtil.provideEntityManager();

            em.getTransaction().begin();

            em.persist(account);
            flag=true;

            em.getTransaction().commit();


            em.close();

            return flag;
      }

      @Override
      public boolean deleteAccount(int accno) {
            boolean flag=false;


            EntityManager em= EMUtil.provideEntityManager();

            Account acc=em.find(Account.class, accno);

            if(acc != null){

                  em.getTransaction().begin();

                  em.remove(acc);
                  flag=true;

                  em.getTransaction().commit();
            }

            em.close();



            return flag;
      }

      @Override
      public boolean updateAccount(Account account) {

            boolean flag=false;


            EntityManager em= EMUtil.provideEntityManager();
            em.getTransaction().begin();

            em.merge(account);
            flag=true;

            em.getTransaction().commit();


            em.close();

            return flag;

      }

      @Override
      public Account findAccount(int accno) {
            /*Account account=null;

       EntityManager em=EMUtil.provideEntityManager();


            account = em.find(Account.class, accno);




            return account;*/

            return EMUtil.provideEntityManager().find(Account.class,
accno);

      }
}
```

persistence.xml:-

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
             version="2.0">


    <persistence-unit name="account-unit" >


<properties>


             <property name="hibernate.connection.driver_class"
value="com.mysql.cj.jdbc.Driver"/>
            <property name="hibernate.connection.username" value="root"/>
            <property name="hibernate.connection.password"
value="root"/>
            <property name="hibernate.connection.url"
value="jdbc:mysql://localhost:3306/ratandb"/>

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



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

DepositUseCase.java:-

```
public class DepositUseCase {


      public static void main(String[] args) {

            AccountDao dao=new AccountDaoImpl();

            /*Account acc1=new Account();
            acc1.setName("Ramesh");
            acc1.setBalance(880);



            boolean f= dao.createAccount(acc1);

            if(f)
                  System.out.println("Account created..");
            else
                  System.out.println("Not created...");*/


            Scanner sc=new Scanner(System.in);

            System.out.println("Enter Account number");
            int ano=sc.nextInt();

            Account acc= dao.findAccount(ano);

            if(acc == null)
                  System.out.println("Account does not exist..");
            else{


                  System.out.println("Enter the Amount to Deposit");
                  int amt=sc.nextInt();

                  acc.setBalance(acc.getBalance()+amt);

                  boolean f =dao.updateAccount(acc);

                  if(f)
```

```
                    System.out.println("Deposited Sucessfully...");
                else
                    System.out.println("Technical Error .....");

            }
        }

}
```

WithdrawUseCase.java:-

```
public class WithdrawUseCase {

    public static void main(String[] args) {

        AccountDao dao=new AccountDaoImpl();

        Scanner sc=new Scanner(System.in);

        System.out.println("Enter Account number");
        int ano=sc.nextInt();

        Account acc= dao.findAccount(ano);

        if(acc == null)
            System.out.println("Account does not exist..");
        else{

            System.out.println("Enter the withdrawing amount");
            int amt=sc.nextInt();

            if(amt <= acc.getBalance()){

                acc.setBalance(acc.getBalance()-amt);
                boolean f=dao.updateAccount(acc);
                if(f)
                    System.out.println("please collect the
cash...");
                else
                    System.out.println("Technical Error...");


            }else
                System.out.println("Insufficient Amount..");
        }
    }

}
```

limitation of EM methods in performing CRUD operations:-
----

persist();
find()
merge();
remove();

1.Retrieving Entity obj based on only ID field(PK field) @Id

2.multiple Entity obj retrival is not possible (multiple record)

3.bulk update and bulk delete is also not possible

4.to access Entity obj we can not specify some extra condition.

--to overcome the above limitation JPA has provided JPQL (java persistence query language).

Day 5