javax.persistence.* → JPA implementation.
JPA → Specification
hibernate → implementation

Hibernate

→ An ORM tool
→ Used in data application layer
→ implements JPA

Problems:-
i) mapping member variables to columns
ii) mapping relationship
iii) handling data types
iv) managing changes to object state

Steps
i) create hibernate configuration file - to tell which database to connect to.
ii) model object - what needs to be saved.
iii) service method to create the model object
iv) No database design
v) No need to write DAO method to save the object. using SQL queries

Sapient

i) hibernate.cfg.xml → Default configuration file name. We can change the name but we have to make changes.

a) Connection.driver_class → which database it has to connect to and driver loading

b) connection.url → url of the database

c) Connection.username → username

d) Connection.password → password of DB

e) Connection.pool_size → Connection pool size

f) dialect → which DB language it will use to write query. Every DB will have some different way of writing query.

g) cache.providerclass → for second level cache

f) show_sql → printout all the sql that hibernate generates in the console.

g) hbm2ddl.auto → It will tell whether to create the table automatically if it is not present

h) mapping class → mapping of the class file. Project ka file (model class)

Sapient kit

@Entity – It tells the hibernate, to treat the class as an entity and it needs to save it.
All the classes with @Entity has to be listed in the hibernate.cfg.xml ~~in the entity~~ under the mapping.
Class level annotation.

@Id → the particular field is the primary key of the table.
Field annotation.

Steps to use Hibernate API

i) Create a session factory. → using the hibernate.cfg.xml we will be creating the session factory.
ii) create a session from session factory inorder to save, update or any operations
iii) use the session to save model object.

Sapient

hbm2ddl.auto

i) create - It recreates the entire schema everytime.

ii) update - It ~~only makes~~ will update the table if it finds it or it will create a new table if it does not find it.

@Entity (name = "Saurabh") → It will create the table with name Saurabh in the DB.

@Column (name = "Saurabh-id") → It will create the column in the table with name Saurabh-id.

We can either put the annotation on top of fields or on top of the GETTER of the fields

@TABLE (name = "Saurabh-table") → This will create the table with the name Saurabh-table.

Difference between @Entity (name = " ") and @Table (name = " ") is that, @Entity will change the entity name but @Table will change the table name only, this will come into picture when we use the HQL.

@Basic → fields annotation
It will treat the field as basic and create the data type as default data type.
String → varchar
Int → integer

@Transient → field annotation
It will not save the field in DB.
→ We can also mark the field as static or transient instead of @Transient.

@Temporal → fields annotation

It is applied on top of fields with data type as Date, using this annotation we can say that we ~~need only~~ want to save only date, time or both.

@Lob → fields annotation

It means that the field is a large Object.

If it is on top of String then hibernate understands that it is CLOB

If it is on top of byte then hibernate understand that it is BLOB

Video - 8

Session.get(User.class, 1) → It is used to retrieve data from the database. 1 is the userId which is being saved in DB.

- Class name.
- Primary Key mentioned in class.

Sapient

Video - 9.

i) @GeneratedValue → It is used to generate the UserId automatically without the user need to explicitly pass the value.
Field annotation.

ii) @GeneratedValue(strategy = GenerationType.AUTO)
→ We will let hibernate to decide which strategy it has to use in order to generated the ids.

iii) GenerationType.identity → hibernate will use the identity column of the DB. Some of the DB provides the identity column.

iv) GenerationType.sequence → It use the sequence object in order to maintain sequences.

v) GenerationType.Table - You can have a seperate table and hibernate will use that table inorder to generate a ~~seperate~~ primary key.

Sapient

Video 10

@Embeddable → Class annotation.

It tells that not to create a seperate table for this class. This class will be embedded inside other.

ex:-

```
class User{
    String name;
    @Embedded → It tells that this field is embeddable.
    Address addr;
    //
}

@Embeddable
class Address{
    String city;
}
```

Table: User

Name	City

Sapient

Video 11

@AttributeOverride → If suppose we want to have two different address in the User class, eg homeAddress and OfficeAddress then it won't be possible as both the column will have same name. that's why we can override the property and can give different name to the different column.

@EmbeddedId

→ EmbeddedID can be used to form the compaite key. All the fields of the address table can be used combined as the primary key

Video 12

@ElementCollection – It can be used to define that the particular field is a collection.

UserId	UserName
1	Saurabh
2	Prakash

UserId	City	PinCode
1	Patna	5600
2	Bihar	2800
1	Siwan	4000

org.hibernate.annotations.* → This is hibernate implementation.

Video - 13

By default if we use collection, then the @name of the table will be TableName_FieldName.

If we want to change the name of the second table then we have to use @JoinTable annotation.

We can use @JoinColumn inside @JoinTable to change the primary key name of the ~~field~~ new table.

@CollectionId → This is hibernate annotation and not from JPA.

This is used to generate the primary key in a table, even if the table is not having any primary or unique key in the table.

Video - 14

Sapient

Lazy loading meaning loading from DB.
Eager loading meaning loading from proxy obj

Video-14 (Imp) → (By default)

Lazy Initialization - It means that we do not initialize the entire object, we only initialize the first level member variables, then we initialize the list only when we access it.
List and Collection value will be retrieved from DB.

for example:-

```
class User{
    String name;
    String id;
    Collection<Address> add = new ArrayList<>();
}

class Address{
    String city;
}

class Main{
    psvm(){
        user=Session.get(User.class, 1); // only user object will be loaded collection not.
        user.getAdd(); // Only when we use get then collection will be loaded.
    }
```

Sapient

Eager initialization - It will load the collection of address as well as at the same time.

Hibernate uses Proxy Class inorder to achieve lazy loading

Hibernate will create a proxy class of the entity class, and when we try to do Session.get(), it will hit the proxy class and get the data. It wont hit the actual Database to get the first level value (like name, id), but inorder to get the second level value (like List<Address> add) it will hit the dB inorder to get its value.

In Order, to get the value of the collection as eagerly we have to set the

@ElementCollection(fetch = FetchType.EAGER)

Sapient

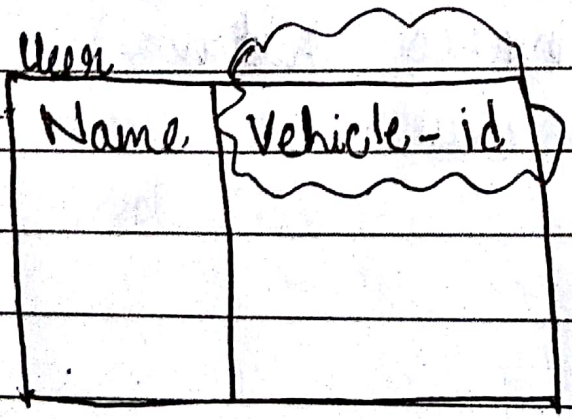
Video - 15 (One to One Mapping)

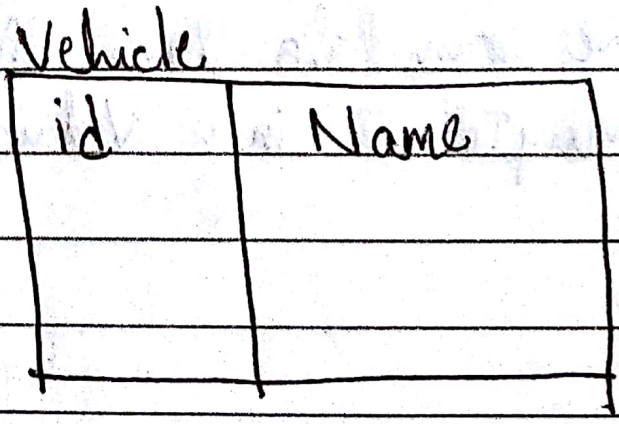
@OneToOne → We have to anotate the feild with @OneToOne, to say that it is one to one.

```
class User{
    String name;
    @OneToOne
    Vehicle v;
}

class Vehicle{
    @Id @GeneratedValue
    int id;

    String name;
}
```





Sapient

Video - 16 (One To Many)

@OneToMany - It helps in mapping one to many relationships.

For example:- A user can have multiple vehicles.

User

UserId	UserName

Vehicle

VehicleID	VehicleName

User-Vehicle

User-UserId	Vehicle-VehicleID

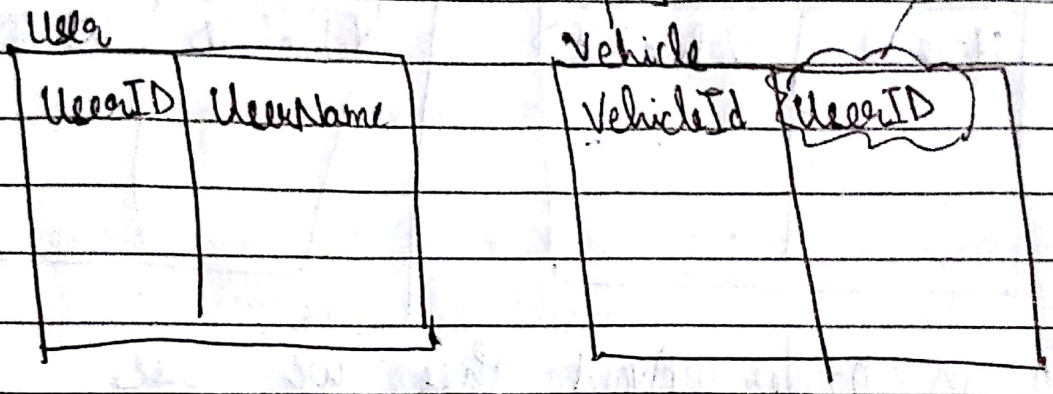
We can also have the inverse relation @ManyToOne in Vehicle class.

@Video - 17 (ManyToMany And MappedBy)

@MappedBy → @MappedBy is used to identify the inverse side of a relationship.

```
Class User {
    int userId;
    String UserName;
    @OneToMany(mappedBy = User)
    Vehicle vehicle
}
```

```
Class Vehicle {
    int vehicleID;
    @ManyToOne
    @JoinColumn(name = User_id)
    private UserDetails user;
}
```



As vehicle can belong to only one user, so we can have an extra column in the Vehicle table.

Sapient

@ManyToMany → We need to have a seperate table.

If we will give @ManyToMany in both table then two tables will be generated which will ~~beside~~ have same mapping but order different.

ex :-

User – Vehicle

UserId	VehicleID

Vehicle – User

VehicleID	UserID

So in orderToAvoid this, we use MappedBy inside one of the @ManyToMany Annotation

Sapient

Video - 18

@NOTFound → field annotation
If data is not found then you have to perform an action.

e.g - @NOTFOUND (action = NOTFOUNDException.IGNORE)

Hibernate Collections :-
↳ Bag semantic = List / Arraylist (No order)
↳ Bag semantic with ID - " " (No order)
↳ List semantic " " (Ordered)
↳ Set Semantic - Set
↳ Map Semantic - Map.

Cascade Types → It basically ~~set~~ Symbolizes that all the related fields should be saved or deleted from the corresponding table.
We dont have to explicitly mention session.save() for all the related fields or we dont have to explicitly delete from the related fields.

Sapient

@OneToMany(cascade = CascadeType.persist)
Collection<Vehicle> vehicle = new ArrayList<>();

CascadeType.persist → It is for saving
remove → It is for deleting
All → If you want to perform
all the operations like
create, ~~delete~~ delete, update
etc.

Video-19 Implementing Inheritance.

Hibernate by default implements a strategy which is known as "single table strategy". It will create ONE table for all the child class.

```
Class Vehicle {
    @Id
    @GeneratedValues
    String id;
    String name;
}
```

```
class TwoWheeler implements Vehicle {
    String steeringHandle;
}
```

```
Class FourWheeler implements Vehicle {
    String steeringWheel;
}
```

↳ Descriminator Type

DType	Id	Name	SteeringHandle	Steering wheel
Vehicle	1	Car		
TwoWheeler	2	Hero	Bike Steering	
FourWheeler	3	Porche		Car Steering

Video-20 (Single Table Strategy)

@Inheritance → Class level
If we do not define any value then it will automatically go for single table strategy

or

we can write

@Inheritance (strategy = InheritanceType.SINGLE_TABLE)

@DescriminatorColumn → Class level (Top level class)
This is used to change the name of the descriminator column. By default it will take the classname.

@DescriminatorValue → Class level (Child + Top class)
This is used to change the name of the values populated in descriminator column.

Sapient

Video-21 (Table per class) (It duplicate the properties of parent class in the child class)

We dont need a discriminator

@Inheritance(strategy = InheritanceType.

class Vehicle) ← Same class defination as in video-19

}

Vehicle

Id	Name
1	Vespa

TwoWHEELER

ID	Name	SteeringHandle
2	TVS	handle

FourWHEELER → //Repeating from Parent class

ID	Name	SteeringWHEEL
3	Maruti	wheel

Sapient

Video - 22 (Inheritance with Joined Strategy)

@Inheritance(Strategy = InheritanceType.Joined)

Vehicle

Id	Name
1	Veepa
2	TVS
3	Maruti

TWO WHEELER (No Name Column)

ID	SteeringHandle
2	Handle

FOUR WHEELER

ID	SteeringWHEEL
3	WHEEL

Video-23 (CRUD operation)

C → create → session.save(object);

R → read → session.get(ClassName, primary key);

D → delete → session.delete(object);

e.g → User user = (User) session.get(User.class, 6);

session.delete(user);

U → update → session.update(object);

e.g → User user = (User) session.get(User.class, 5);

user.setUserName("updated");

session.update(user);

Sapient

Video-24 (Transient, persistent & Detached)

Transient → means that object is not saved in DB

Persistent → means that object will be saved in DB.

Detached Object → session.close()

when we do session.close() it means that hibernate is not going to track any changes and any changes made after the close statement will not get persisted in DB.

for example :-

```
User user = new User();                                   → Transient
user.setName("saurabh");
SessionFactory sf = new Configuration().configure().buildSessionFactory();
Session session = sf.openSession();
session.beginTransaction();

session.save(user);                                       } Persistent
user.setName("Prakash");
user.setName("Prakash2");
session.close();                                          } Detached
```

Sapient

Video - 05 (State changes)

Create

new()

Transient → Persistent → Detached

session.save() session.close()

Read

get()

Persistent → Detached

session.close()

Delete

Transient ← Persistent → Detached

session.delete() session.close

Sapient

Video - 26 (Persisting Detached Object)

@org.hibernate.annotations.Entity(selectBeforeUpdate=true)

This annotation fire or trigger a select query Before performing update query. If there is any change then only it will run update query else it wont run any update query.

//Another example

User user = (User) session.get(User.class, 1);

↳ session.close

user.setUserName("Ram"); ← No session open at this point.

```
Session = SessionFactory.openSession();
Session.beginTransaction();
session.update(user);
Session.getTransaction.commit();
Session.close();
```

Sapient

Video-27 (Writing HQL)

HQL is a query which we use in hibernate

→ In HQL we mention classname and fields name instead of table name and columnname.

→ We dont use "Select *" in hQL

→ We dont get resultSet we get list

→ we use :- Query query = session.~~get~~createQuery ("from User");

Sapient

Video-28 (Select and Paginitation in HQL)

Pagination

query.setFirstResult(n) → This is the starting point (5) the nth record from which we want the data. (starting point)

query.setMaxResults(n) → The no. of records that we need. (n no. of records)

Output - 5
6
7
8

~~lazy load~~

We can also write

Query query = session.createQuery("select userName from PersonGrad");

If we want to retrieve specific field from the the class (~~column~~ table)

Sapient

Video 29 (Parameter Binding)

We should not do append parameter as it might lead to sql injection attacks. So inorder to avoid sql injection attacks we should do parameter binding.

1st way

```
Query q = session.createQuery("select name from
                PersonCRUD where id >? and name
                = ?");
q.setInteger(0, 5);
q.setString (1, "Ram");
```

2nd way

```
Query q = session.createQuery("select name from
                PersonCRUD where id=:id and name
                = :name");
q.setInteger("id", 5);
q.setString("name", "Ram");
```

Sapient

Video 30 (Named Query)

Named Query - It is used to consolidate all the query at one place and organize the query and maintain them.

It helps to write query at entity level.

@NamedQuery(name="", query="")

→ Another advantage of named query is that we can use the SQL query ~~or~~ (i.e. ~~no~~ we can write tablename instead of classname)

@NamedNativeQuery(name="", query="", resultClass=)

→ We can use @NamedNativeQuery to ~~write PL~~ run PL/SQL

→ We do write HQL in named query

Sapient

Video 31 (Criteria API)

There are 3 ways that we can execute query in ~~DB~~ Hibernate :-

1) session.get(User.class, 5) → This is very basic way.

2) session.createQuery("~~select~~ from User where id = :id");
~~sess~~ query.setInteger("id", 5);

// Using HQL.

3) Criteria API →
Criteria objects are some what like where clause.

Criteria criteria = session.createCriteria(User.class);
criteria.add(Restrictions.eq("UserName", "Saurabh"));
List<User> u = (List<User>) criteria.list();
It will fetch all the details of the UserName = "saurabh";

Sapient

Video - 32 (Understanding Restrictions)

We can add any number of criterias added

```
Criteria c = session.createCriteria(User.class);
c.add(Restrictions.eq("userid", 5))
 .add(Restrictions.gt("userid", 6));
```

We can also use 'OR' condition

```
c.add(Restrictions.or(Restrictions.eq("userid",5),
                      Restrictions.eq("userId",7)));
```

Video-33 (Projections)

→ Projections are used for aggregation fution
→ Projections are part of criteria.
→ Projections can be used to retrieve a single property from the class
→ or to find the max, min or any aggregation fution.

e.g. -

```
Criteria c = Session.createCriteria(User.class)
             .setProjection(Projections.property("id"));
```

QueryByExample - too many property and too many criteria to satisfy.

→ Example wont consider the primary key

→ Example wont consider the null column

→ We can also use "excludeproperty" in example, if we dont want any property to be considered.

Sapient

Video - 34 (Caching) → By default

→ first level cache → Session (eg - update query or select query)

→ Second level cache → for data to be for long time (Session factor)

→ We can implement 2nd level cache in different ways.

→ across session in an application

→ across applications

→ across clusters.

Video-35 (2nd level cache)

We can use different 2nd class to implement 2nd level cache -

i) EHCache.class

ii) OSCache.class

There are 3 steps to perform 2nd level cache :-

i) ~~mak~~ add 2 properties in hibernate.cfg.xml

ii) Add jars (EHCacheImplementor or OSCacheImpl)

iii) Mark all the classes whom you want to make them 2nd level cache enabled

i) <property name = "hibernate.cache.use_second_level_cache> true </property>

<property name = "cache.provider_class">
org.hibernate.cache.EhCacheProvider
</property>

ii) download Jar

iii) Mark the entity class as

@Cacheable

@Cache (usage = CacheConcurrencyStrategy.READ_ONLY)

Video-31 (Query Cache)

We in a way say that there are 3 types of cache.

i) Session level
ii) SessionFactory "
iii) Query level

In order to make query level cache, we need to do 3 things

i) 2 properties in hibernate.cfg.xml
ii) Jar download
iii) set → query.setCachable(true);

i) Properties

<property name="cache.provider_class">org.hibernate.cache.EhCacheProvider</property>

<property name="hibernate.cache.use_query_cache">true</property>

ii) Jar download

iii) Query q = session.createQuery("select name from User where id=1");
q.setCachable(true);
List l = q.list();

Sapient