Hibernate POJO object should have default constructor

**@Id** annotation on property defines whether hibernate will access fields or accessor methods for the columns, so the mapping needs to be defined accordingly

**@Access**

Override the how property of entity are accessed at runtime

If @Id is defined at field level then @Access also needs to be defined on field level, access comes into picture only at run time when loading or storing the entity

noop

If DB has columns which are not part of entity object then such properties are defined in hbm.xml

Note annotations on model object will cease to exists and needs to be defined in the hbm.xml

@ColumnTransformer

Used to modify value of fields before reading or writing to DB (Transformations happens @ DB level)

@Generated

Used to delegate to Hibernate to access DB updated fields after an INSERT/UPDATE

Generally, hibernate app needs to refresh instance to load DB generated values for fields after INSERT/UPDATE

eg. housekeeping columns updated by trigger

@ColumnDefault

Assing default value for a column during DDL

@Temporal

Hibernate by default uses TIMESTAMP temporal type if no @Temporal defined

@CreationTimestamp and @UpdateTimestamp

Tells hibernate to automatically create or update

@Enumerated

By default for enums the ordinal value is stored in DB in order to store the label name use EnumType.STRING

@MappedSuperClass

By default columns of superclass are not persisted in subclass tables, use this to persist superclass columns

Not applicable when using inheritance stratergies

@DiscriminatorColumn

used to rename discriminator columns else column name DTYPE will be used

@DiscriminatorValue

Use to name the discriminator values otherwise entity names are used

@DiscriminatorFormula

When discriminator column cannot be added to DB, it can be derived

**Access** - How hibernate gets or set a value

1. **Field access**
2. **Property access** (set on getter and not on setter)

The placement of @Id decides access types, but if @Access is specified it decides the access type

@Access annotation specifies

**@Column**

insertable/updatable: specifies if a column should be included in insert/update

nullable: allow null or not

**@Basic**

By default all properties are nullable

If optional is false and hibernate is generating the schema then field will created with not null constraint

optional=false ensures the property needs to set otherwise hibernate throws exception before hitting DB with SQL

**@GeneratedValue** strategies

**AUTO** - picks any strategy based on DB implementation

**IDENTITY** applicable for specific DB's like MYSQL (auto increment field)

**SEQUENCE** applicable for specific DB's like ORACLE, SequenceGenerator specified sequence name

**TABLE** DB table created with primary key and next primary key value

@TableGenerator specifies table name pk column name and value

if no table generator specified **hibernate\_sequences** table (sequence name as pk default as value) is created with next value (start value is 0/current value is 1)

It fails if more than 1 entity are used

if no sequence generator is specified then **hibernate\_sequence** table is created with next value (start value is 1, current value is 1)

It uses running sequence number when more than 1 entities are used

In case of IDENTITY creates auto increment primary key column

**Sequences are faster than identity as they are stored in memory whereas getting next value in an identity requires disk read**

**@Temporal**

Hibernate exhibits below behaviour when fetching from DB

java.util.Time return Data(1970-01-01) + Time and java.util.Date return Date + Time(00:00:00)

java.sql.Time returns time and java.sql.Date return date

use DATE and TIME in @Temporal to prevent above

**@Formula (hibernate)**

Runs SQL function on the column when selecting from DB

applicable for only selects not for inserts/updates [value is not included]

**Hibernate Types**

Java types are converted to hibernate types which then converted to DB types

1. **Value Types**

**Basic** - String, int, long

**Composite/Embedded** - Address, attributes which are not entities

**Collection** - List<String> or List<Address>

1. **Entity Types**

Corresponds to row in DB like User, Department

attributes which are entities themselves

**@AttributeOverrides**

Used to give names to the columns of embedded object if it is shared among multiple entities and each entity needs to have separate column name for the same field of the embedded object

**@ElementCollection**

**@CollectionTable** //used to provide mapping data (specify the table and join column name)

**@Column** specifies the value column name

Collections are stored in separate table with entity class id as foreign key

Applicable for List and Set only

For supporting Map add **@MapKeyColumn** which specifies the column name of key and @Column will contain value column name

Collection of embedded types **@AttributeOverride** continue to work but the embedded object will be stored in separate table and entity class id as foreign key

Embedded types columns are present in the same table as entity

**@Embedded** or **@Embeddable** are optional if either one is present

**Owning entities [Entity in which joinColumn is specified]**

**Source object [Table containing foreign key]**

**@OneToOne** //cascade type specifies when source is persisted target should also be persisted

**@JoinColumn** name specifies the source column name and referencedColumnName specifies target column name

In case of bi-direction one to one mapping @JoinColumn should not be speicifed on the target object

@OneToOne (mappedBy specifies target's column name in source object, points to field having @JoinColumn)

**@OneToMany**

In unidirectional mapping always on side is specified, other side is not specified

insertable=true and updatable=false is required on bi directional m:q!apping because the same column is annoted twice

In case of unidirectional one to many @OneToMany is specified on the target object and @JoinColumn is also specified with column name as source object's column name

Also, JoinColumn should not be nullable because if its nullable then referential integrity violation occurs

In bidirectional one to many the source object has the @ManyToOne mapping as well as the @JoinColumn

**@JoinTable**

If in a one to many relationships their multiple records in target table for which mapping to source table does not exists then join table is the best option

JoinColumn // target's column in join table

InverseJoinColumn // source’s column in join table

**@ManyToMany**

The @JoinTable mapping can be either on source or target object

Session // persistance manager

openSession() // persistance context is created

save() get() load() close()

**Transient**-------------------->**Persistent** ------------------------>**Detached**

saveOrUpdate() <------------------------

|                        saveOrUpdate() save() update()

|

| delete()

|

->**Removed**-----------------------------------> Deleted from DB

commit() close() flush()

load() method returns proxy object and when try to get the property, actual select is issued

returns row not found exception when try to get any property instead of null pointer in get()

flush is syncing persistence context and DB

flush happens during session.flush(), commit() and during select operation

Transaction ensures atomicity

**JPA API**

No need to mention mapping files as they are automatically scanned by JPA

persistance.xml contains met data info

persist() <--> save()

find() <--> get()

getReference() <--> load()

remove() <--> delete()

clear() detaches all entities in the persistence context

detach() detaches particular entity

merge() - reattach detached entity

if entity is no present in the context queries DB to get the entity and updates the entity and return this entity, if entity not present it calls persist () which will insert in DB

**@IdClass - JPA**

        Specify the id class, this is a separate class which only the id fields are present

        This id class is specified during get

In case of enumerations the ordinal value is stored in DB, if new enum added it may lead to incorrect value being populated

**@Enumerated** - JPA specify ordinal or string value

**@MappedSuperClass** // same as TABLE\_PER\_CLASS

        @Id is specified on the sub classes

        Specified on the abstract class so that the annotations specified in abtract class ae picked up in its sub classes

**@Inheritance** specified on base class

**@Id** specified on base class

**Inheritance strategies**

                 Collateral

                        |

----------------------------------------------------------

|                        |         |

Cash Collateral Security Guarantee

1. **SINGLE\_TABLE**

using discriminator **@DiscriminatorColumn**

Single table Collateral created with all columns belonging to subclasses with extra discriminator column

Does not satisfy 3NF and subclass specific not null constraints may be violated

1. **JOINED**

table for base class and sub class and using joins

Separate table for Collateral, Cashcollateral, Security and Guarantee each having columns specific to their class

When selecting collateral table hibernate does LEFT OUTER JOIN with of Collateral with (Cash collateral, Security and Guarantee)

When selecting Cashcollateral table hibernate does INNER JOIN with Collateral table

1. **TABLE\_PER\_CLASS**

table for all concrete classes and uses unions

CashCollateral, security and Guaranatee tables created, with @Id mapping should be defined on Collateral

For very complex queries create a view and map this as entity (read only)

**hbm2ddl**

**create //drop everything and create**

**validate // only validate the schema, if anything is missing throws error**

**update // if anything is missing creates it**

**create-drop // create everything and drop**

**HQL - Query**

**JPQL - Query/TypedQuery (No cast required)**

Positional parameter -?

In JPQL it is based and position needs to be mentioned like ?1 ?2

Named parameters

**SessionFactory <--> EntityMangerFactory**

**Session <--> EntityManager**

**Transaction <--> EntityTransaction**

**Hibernate findings**

In case of foreign key constraint, the foreign key should be the primary key of another table

@ManyToMany

Market <--> Instrument

For select fetches primary key of Market/Instrument

For each key it performs inner join of Market/Instrument with Market\_Instrument table (Lazy loading)

@OneToMany

Batch <--> Transaction

For each batch fetch transactions from Transaction (Lazy loading)

**Criteria API - build dynamic queries**

Criteria/CriteriaQuery

JPA uses the concept of Root, it creates CriteraQuery which is passed to Query

Crtierion(Where condition) are created from Restrictions