*1)* ***@Entity*** *🡪 javax.persistence.Entity 🡪 It is class level Annotation*

* *It belongs to JPA(Java Persistence API) it doesn’t belong to hibernate. Hibernate uses JPA annotations.*
* *It is given for class and basically wherever we use @Entity annotation it represents table in our database.*
* *This entity tells to map the specific class with database table.*
* *Entity classes are classes that are mapped with database table.*

*What is entity class?*

*🡪A class that is associated with data table is called as entity class.*

*2)* ***@ Id*** *🡪 javax.persistence.Id 🡪It is variable level annotation*

* *It is given to a Primary Key / ID type of value.*
* *This annotation needs mandatorily given, we cannot skip this.*

*\*(name) attribute is used to give custom name to table or any annotation provided.*

*\* It is always used with annotation.*

*3)* ***@ Table*** *🡪 javax.persistence.Table 🡪 It is class level annotation*

* *Also used to specify which class we want to create table for.*

*4)* ***@Column*** *🡪 javax.persistence.Column 🡪It is variable level annotation*

* *We can specify column name and give other attributes using this annotation*
* *@Column(name="columnName",unique=****true****,nullable=****false****)*

*5)* ***@GeneratedValue*** *🡪 It is variable level annotation*

* *We can specify* ***(strategy=GenerationType.Identity)*** *to specify autoincrement for PrimaryKey.*

1. *GenerationType.Identity 🡪 Generates and maintains separate sequence ID for particular table/entity class/POJO class*
2. *GenerationType.AUTO 🡪One common sequence is maintained for the table/entity class/POJO class in entire database database*

*6)* ***@cache****🡪It is class level annotation*

* *(usage=CacheConcurrencyStrategy.READ\_ONLY)* 🡪*provides read only access*

*7)* ***@OneToOne 🡪***

* *It is Object variable level annotation*
* *This annotation is applied to the relationship between Car and Engine. It specifies that each Car has one Engine, and each Engine belongs to one Car.*
* *It is used to model a one-to-one relationship.*
* *One entity is associated with another one entity*
* *By default Supports Eager Loading*
* ***(fetch=fetchType.EAGER)****,* ***(fetch=fetchType.LAZY)****, To force fully change the fetching type from default type*
* ***(cascade=CascadeType.ALL)****🡪If we perform any operations (save,get,delete) on a target entity then it gets applied / reflected on child entity as well. Eg:- If we are saving Car entity then automatically engine entity is also going to get saved.*

*8)* ***@OneToMany 🡪***

* *It is Object variable level annotation*
* *By default Supports Lazy Loading*
* ***(cascade=CascadeType.ALL)***
* ***(fetch=fetchType.EAGER)****,* ***(fetch=fetchType.LAZY)*** *🡪 To force fully change the fetching type from default type*

*9)* ***@NamedQueries 🡪***

*10)* ***@NamedNativeQueries 🡪***

**✅ @Entity**

🔍 **Purpose:**  
Marks a Java class as a **persistent entity**, allowing Hibernate to map it to a database table.

🤔 **Tricky/Indirect Questions:**

* How does Hibernate identify which classes should be persisted?
* Without XML config, how does ORM know to treat a class as a database table?
* How do you declare a class as a table-mapped entity in Hibernate?

⚙️ **Key Attributes:**

* name: *(Optional)* Custom logical entity name used internally by Hibernate.  
  ➤ *Note: Doesn’t change table name, only the entity reference name.*

🏷️ **Annotation Level:**  
Class-level.

🛠️ **Real-Time Example:**

java

@Entity

public class Product {

@Id

private int id;

private String name;

}

📝 **Extra Notes:**

* Must be used with @Id on at least one field.
* If omitted, the class won't be scanned or persisted by Hibernate.

🧪 **Similar Annotations:**

* @MappedSuperclass → for inheritance strategies.
* @Embeddable → for value objects that don't require separate tables.

🚨 **Possible Exceptions:**

* *AnnotationException*: Missing @Id in @Entity class leads to failure during persistence.

**✅ @Id**

🔍 **Purpose:**  
Specifies the **primary key** of the entity. Hibernate uses this to uniquely identify each row.

🤔 **Tricky/Indirect Questions:**

* How does Hibernate distinguish each row in the DB?
* What marks a field as the unique identifier for an entity?
* Without a unique constraint, how do you link a Java object to a table row?

⚙️ **Key Attributes:**

* None (used alone or with @GeneratedValue).

🏷️ **Annotation Level:**  
Field or Getter (depending on access strategy).

🛠️ **Real-Time Example:**

java

@Id

private int id;

📝 **Extra Notes:**

* Required for all entities.
* One and only one @Id per entity (unless using @EmbeddedId for composite key).

🧪 **Similar Annotations:**

* @EmbeddedId: For composite key objects.

🚨 **Possible Exceptions:**

* *AnnotationException*: No @Id or @EmbeddedId in entity class.

**✅ @GeneratedValue**

🔍 **Purpose:**  
Instructs Hibernate to **automatically generate** the value of the primary key.

🤔 **Tricky/Indirect Questions:**

* How does Hibernate handle primary key creation without manual input?
* What allows Hibernate to auto-increment an ID?
* How do you configure key generation strategy?

⚙️ **Key Attributes:**

* strategy:  
  ➤ AUTO *(default)* – Hibernate chooses based on dialect  
  ➤ IDENTITY – Uses DB auto-increment (e.g., MySQL)  
  ➤ SEQUENCE – Uses DB sequence (e.g., Oracle, PostgreSQL)  
  ➤ TABLE – Uses a table to simulate sequence
* generator:  
  ➤ *Name of a custom generator (if defined with @GenericGenerator)*

🏷️ **Annotation Level:**  
Field/Getter (must be with @Id)

🛠️ **Real-Time Example:**

java

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private int id;

📝 **Extra Notes:**

* The generation strategy must match your DB support (e.g., don’t use SEQUENCE with MySQL).
* Can be customized further using @SequenceGenerator or @TableGenerator.

🧪 **Similar Annotations:**

* @GenericGenerator → For custom generation logic (e.g., UUID).
* @SequenceGenerator, @TableGenerator

🚨 **Possible Exceptions:**

* *MappingException*: Using SEQUENCE with MySQL can cause generation errors.
* *IllegalArgumentException*: If an unknown strategy is passed.

**✅ @Table**

🔍 **Purpose:**  
Specifies the **exact table name** in the database to which the entity should be mapped. Used when the class name doesn't match the table name.

🤔 **Tricky/Indirect Questions:**

* What if your class name and table name are different?
* How can you map a class to a custom table name?
* Where do you define schema or catalog for an entity?

⚙️ **Key Attributes:**

* name: *(Required)* → Table name to map with
* schema: *(Optional)* → DB schema name
* catalog: *(Optional)* → Catalog name
* uniqueConstraints: *(Optional)* → Define unique constraints on columns

🏷️ **Annotation Level:**  
Class-level (used with @Entity)

🛠️ **Real-Time Example:**

java

@Entity

@Table(name = "product\_master", schema = "inventory")

public class Product {

@Id

private int id;

}

📝 **Extra Notes:**

* If not used, Hibernate uses class name as table name by default.
* Useful in multi-schema or legacy DB scenarios.

🧪 **Similar Annotations:**

* @SecondaryTable – to map to more than one table
* @JoinTable – for mapping associations like Many-to-Many

🚨 **Possible Exceptions:**

* *SQLGrammarException*: Incorrect table/schema names not existing in DB
* *AnnotationException*: When @Table is applied without @Entity

**✅ @Column**

🔍 **Purpose:**  
Maps a field or property to a **specific column** in the database table. Controls column name, nullability, length, etc.

🤔 **Tricky/Indirect Questions:**

* How do you customize the column name or type in the database?
* How do you control constraints like length or nullable in Hibernate?
* What if DB column is named differently from your field?

⚙️ **Key Attributes:**

* name: *(Optional)* → Custom column name
* nullable: *(Optional, default=true)* → Whether the column allows null
* length: *(Optional)* → Column length (used for String fields)
* unique: *(Optional)* → Whether the column must have unique values
* updatable / insertable: *(Optional)* → Exclude fields from INSERT/UPDATE

🏷️ **Annotation Level:**  
Field or Getter-level

🛠️ **Real-Time Example:**

java

@Column(name = "product\_name", length = 100, nullable = false)

private String name;

📝 **Extra Notes:**

* @Column is optional; by default, field name = column name.
* It gives more control over schema generation.

🧪 **Similar Annotations:**

* @JoinColumn – used for foreign key columns in relationships
* @Formula – used for calculated/derived fields

🚨 **Possible Exceptions:**

* *SchemaValidationException*: DB column and Java field type mismatch
* *ConstraintViolationException*: Violating length, uniqueness, or nullability rules

**✅ @OneToOne**

🔍 **Purpose:**  
Defines a **one-to-one relationship** between two entity classes, where one entity is directly associated with another.

🤔 **Tricky/Indirect Questions:**

* How do you map a one-to-one link between two tables?
* What annotation is used when each row in Table A maps to one row in Table B?
* How can you make two entities depend on each other?

⚙️ **Key Attributes:**

* mappedBy: *(Optional)* → In bidirectional mapping, tells which side owns the relationship
* cascade: *(Optional)* → Define cascade operations (PERSIST, ALL, etc.)
* fetch: *(Optional)* → Eager or Lazy loading
* orphanRemoval: *(Optional)* → Deletes child when unlinked from parent

🏷️ **Annotation Level:**  
Field or Getter

🛠️ **Real-Time Example:**

java

@OneToOne(cascade = CascadeType.ALL)

@JoinColumn(name = "passport\_id")

private Passport passport;

📝 **Extra Notes:**

* Requires a @JoinColumn for foreign key or mappedBy on inverse side.
* Default fetch type is *EAGER*.

🧪 **Similar Annotations:**

* @ManyToOne – for many-to-one mappings
* @OneToMany – opposite of one-to-many mapping

🚨 **Possible Exceptions:**

* *LazyInitializationException*: When LAZY-loaded child is accessed after session close
* *ConstraintViolationException*: When foreign key constraint fails

**✅ @ManyToOne**

🔍 **Purpose:**  
Used to map a **many-to-one relationship**, where *many instances of an entity* refer to *one instance of another entity*.

🤔 **Tricky/Indirect Questions:**

* How would you model "many employees reporting to one manager"?
* What annotation do you use when many records in Table A point to a single record in Table B?
* How do you manage foreign keys in parent-child relationships?

⚙️ **Key Attributes:**

* fetch: *(Optional)* – FetchType.LAZY (default) or FetchType.EAGER
* cascade: *(Optional)* – Defines cascading operations like PERSIST, MERGE
* optional: *(Optional, default = true)* – Specifies whether the association is optional

🏷️ **Annotation Level:**  
Field or Getter

🛠️ **Real-Time Example:**

java

@ManyToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL)

@JoinColumn(name = "dept\_id")

private Department department;

📝 **Extra Notes:**

* Requires @JoinColumn to specify the foreign key.
* Default fetch is *lazy*, unlike @OneToOne.

🧪 **Similar Annotations:**

* @OneToOne – maps one-to-one
* @OneToMany – inverse of many-to-one
* @JoinColumn – required with it

🚨 **Possible Exceptions:**

* *LazyInitializationException*: Accessing lazy object outside session
* *ConstraintViolationException*: Missing or invalid foreign key

**✅ @OneToMany**

🔍 **Purpose:**  
Defines a **one-to-many relationship**, where *one entity instance* is related to *many instances* of another entity.

🤔 **Tricky/Indirect Questions:**

* How do you model a Department having multiple Employees?
* What’s the reverse side of a @ManyToOne mapping?
* How can a single object manage a list of children?

⚙️ **Key Attributes:**

* mappedBy: *(Required in bidirectional mapping)* – Points to the owning side
* cascade: *(Optional)* – Defines cascade behavior
* fetch: *(Optional)* – Lazy or Eager loading
* orphanRemoval: *(Optional)* – Deletes children when removed from the collection

🏷️ **Annotation Level:**  
Field or Getter

🛠️ **Real-Time Example:**

java

@OneToMany(mappedBy = "department", cascade = CascadeType.ALL)

private List<Employee> employees;

📝 **Extra Notes:**

* Usually used with mappedBy to make it bidirectional.
* Cannot work alone — must be paired with @ManyToOne on the child side.

🧪 **Similar Annotations:**

* @ManyToOne – child side of the relationship
* @OneToOne – for one-to-one mapping

🚨 **Possible Exceptions:**

* *PersistentObjectException*: Attempting to persist detached children
* *LazyInitializationException*: Accessing the list after session close

**✅ @ManyToMany**

🔍 **Purpose:**  
Defines a **many-to-many relationship**, where *multiple records in one entity* are associated with *multiple records in another*.

🤔 **Tricky/Indirect Questions:**

* How would you model "students enrolled in multiple courses"?
* Can a single object have a list of another object, and vice versa?
* What relationship requires an intermediate table?

⚙️ **Key Attributes:**

* mappedBy: *(Optional)* – In bidirectional mappings
* cascade: *(Optional)* – Cascade options
* fetch: *(Optional)* – Fetch strategy

🏷️ **Annotation Level:**  
Field or Getter

🛠️ **Real-Time Example:**

java

@ManyToMany

@JoinTable(

name = "student\_course",

joinColumns = @JoinColumn(name = "student\_id"),

inverseJoinColumns = @JoinColumn(name = "course\_id")

)

private List<Course> courses;

📝 **Extra Notes:**

* Requires a **join table** to work.
* You must define both joinColumns and inverseJoinColumns.

🧪 **Similar Annotations:**

* @OneToMany, @ManyToOne for other associations
* @JoinTable – supports many-to-many mapping

🚨 **Possible Exceptions:**

* *MappingException*: If join table structure is incorrect
* *SQLIntegrityConstraintViolationException*: If duplicate insertions in join table

**✅ @JoinColumn**

🔍 **Purpose:**  
Specifies the **foreign key column** used for an association.

🤔 **Tricky/Indirect Questions:**

* How do you define the foreign key column name?
* What if you want to change the default FK column?
* How do you define custom join logic in associations?

⚙️ **Key Attributes:**

* name: *(Required)* – Name of the foreign key column
* referencedColumnName: *(Optional)* – Column in the target entity
* nullable: *(Optional)* – Whether FK can be null
* unique: *(Optional)* – Uniqueness constraint

🏷️ **Annotation Level:**  
Field or Getter

🛠️ **Real-Time Example:**

java

@ManyToOne

@JoinColumn(name = "dept\_id", referencedColumnName = "id")

private Department department;

📝 **Extra Notes:**

* Used in @OneToOne, @ManyToOne, @ManyToMany.
* referencedColumnName is optional unless referencing non-PK column.

🧪 **Similar Annotations:**

* @JoinTable – for mapping join tables
* @ForeignKey – legacy Hibernate annotation (deprecated)

🚨 **Possible Exceptions:**

* *AnnotationException*: Missing or conflicting mapping info
* *ConstraintViolationException*: Invalid FK values or constraint issues

**✅ @JoinTable**

🔍 **Purpose:**  
Defines the **join table** for a @ManyToMany or a bidirectional @OneToMany/@OneToOne mapping.

🤔 **Tricky/Indirect Questions:**

* How do you define a custom table to hold the relationship between two entities?
* What if the default join table is not acceptable?
* Where would you specify two foreign keys for two entities?

⚙️ **Key Attributes:**

* name: *(Required)* – Name of the join table
* joinColumns: *(Required)* – FK referencing the current entity
* inverseJoinColumns: *(Required)* – FK referencing the target entity
* uniqueConstraints, indexes: *(Optional)* – Additional table constraints

🏷️ **Annotation Level:**  
Field or Getter (used with @ManyToMany, sometimes @OneToOne)

🛠️ **Real-Time Example:**

java

@ManyToMany

@JoinTable(

name = "student\_course",

joinColumns = @JoinColumn(name = "student\_id"),

inverseJoinColumns = @JoinColumn(name = "course\_id")

)

private List<Course> courses;

📝 **Extra Notes:**

* Must be used when you want to control the structure of the **intermediate mapping table**.
* Often used with composite keys or legacy table names.

🧪 **Similar Annotations:**

* @JoinColumn – for single FK
* @MappedBy – to define the inverse side

🚨 **Possible Exceptions:**

* *MappingException*: If joinColumns or inverseJoinColumns are misconfigured
* *SQLIntegrityConstraintViolationException*: Bad relationship insertions

**✅ @Embeddable**

🔍 **Purpose:**  
Marks a class as **embeddable**, i.e., it can be embedded in an entity rather than being an entity itself.

🤔 **Tricky/Indirect Questions:**

* How do you store an address inside a user entity table?
* What do you use when you want to reuse a group of fields across entities?
* What’s a lightweight component class that doesn’t have its own table?

⚙️ **Key Attributes:**

* No attributes – just a marker annotation

🏷️ **Annotation Level:**  
Class Level

🛠️ **Real-Time Example:**

java

@Embeddable

public class Address {

private String city;

private String state;

private String zip;

}

📝 **Extra Notes:**

* No separate table is created.
* Can be reused across multiple entities.

🧪 **Similar Annotations:**

* @Embedded – used on the owning side
* @Entity – if it needs its own identity/table

🚨 **Possible Exceptions:**

* *AnnotationException*: Used on non-POJO classes
* *MappingException*: Invalid reuse without proper @Embedded

**✅ @Embedded**

🔍 **Purpose:**  
Used to **embed an @Embeddable class** into an entity. The embedded fields are stored in the **same table** as the entity.

🤔 **Tricky/Indirect Questions:**

* How can you flatten a class's fields into another table?
* How do you map reusable value objects like Address or ContactInfo?
* How to avoid separate tables for related data groups?

⚙️ **Key Attributes:**

* @AttributeOverrides: *(Optional)* – Overrides column names of fields inside the embeddable
* @AssociationOverrides: *(Optional)* – Used for embedded associations

🏷️ **Annotation Level:**  
Field or Getter

🛠️ **Real-Time Example:**

java

@Embedded

private Address address;

📝 **Extra Notes:**

* Automatically maps all fields from the embeddable class into the entity's table.
* Combine with @AttributeOverrides to customize column names.

🧪 **Similar Annotations:**

* @Embeddable – defines the embeddable class
* @ElementCollection – if list of embeddables is needed

🚨 **Possible Exceptions:**

* *PersistentObjectException*: If embeddable contains entity references without proper mapping
* *AnnotationException*: If used on incorrect type

**✅ @ElementCollection**

🔍 **Purpose:**  
Used to map **a collection of basic types or embeddable objects** (non-entity types).

🤔 **Tricky/Indirect Questions:**

* How do you store a list of strings or embeddable objects without making them entities?
* What if you want to store multiple addresses but don’t want a separate entity?

⚙️ **Key Attributes:**

* fetch: *(Optional)* – FetchType.LAZY (default)
* targetClass: *(Optional)* – Type of collection elements
* @CollectionTable: – To define the join table
* @Column: – To define the column of basic type

🏷️ **Annotation Level:**  
Field or Getter

🛠️ **Real-Time Example:**

java

@ElementCollection

@CollectionTable(name = "user\_phones", joinColumns = @JoinColumn(name = "user\_id"))

@Column(name = "phone\_number")

private List<String> phoneNumbers;

📝 **Extra Notes:**

* Creates a **separate table** to store the collection.
* Ideal for storing small value types like strings, dates, etc.

🧪 **Similar Annotations:**

* @OneToMany, @ManyToMany – for entity collections
* @Embedded, @Embeddable – for embedded objects

🚨 **Possible Exceptions:**

* *MappingException*: If used on an entity or mapped incorrectly
* *SQLGrammarException*: Missing collection table during access

**✅ @Id and @GeneratedValue**

🔍 **Purpose:**  
@Id marks the **primary key** field.  
@GeneratedValue provides **automatic ID generation strategy**.

🤔 **Tricky/Indirect Questions:**

* How does Hibernate know which column is the primary key?
* How are IDs generated automatically in entities?
* What if you don’t want to assign IDs manually?

⚙️ **Key Attributes (for @GeneratedValue):**

* strategy: GenerationType.IDENTITY, AUTO, SEQUENCE, TABLE
* generator: *(Optional)* – Custom generator name

🏷️ **Annotation Level:**  
Field or Getter

🛠️ **Real-Time Example:**

java

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

📝 **Extra Notes:**

* For custom sequences, combine with @SequenceGenerator.
* Use GenerationType.IDENTITY for auto-increment DB columns.

🧪 **Similar Annotations:**

* @SequenceGenerator, @TableGenerator – custom strategies
* @EmbeddedId, @IdClass – for composite keys

🚨 **Possible Exceptions:**

* *IdentifierGenerationException*: If strategy is invalid
* *PersistenceException*: If multiple @Id annotations are used

**✅ @SequenceGenerator**

🔍 **Purpose:**  
Defines a **sequence-based primary key generator** (used with @GeneratedValue(strategy = GenerationType.SEQUENCE)).

🤔 **Tricky/Indirect Questions:**

* How would you generate primary keys using a database sequence?
* What if you're using Oracle and want Hibernate to use a native sequence?
* How do you control the name and increment size of auto-generated IDs?

⚙️ **Key Attributes:**

* name: *(Required)* — Logical name of the generator (used in @GeneratedValue(generator = "name"))
* sequenceName: *(Required)* — Actual database sequence name
* initialValue: *(Optional)* — Starting value of the sequence
* allocationSize: *(Optional)* — Increment size

🏷️ **Annotation Level:**  
Field or Class Level (used along with @GeneratedValue)

🛠️ **Real-Time Example:**

java

@SequenceGenerator(name = "product\_seq", sequenceName = "product\_seq\_db", allocationSize = 1)

@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "product\_seq")

@Id

private Long id;

📝 **Extra Notes:**

* Works best with Oracle/PostgreSQL sequences.
* Efficient when used with allocationSize > 1 for batch insert optimization.

🧪 **Similar Annotations:**

* @TableGenerator — Uses a table instead of sequence
* @GeneratedValue — General ID generation

🚨 **Possible Exceptions:**

* *MappingException*: If the sequence does not exist in DB
* *JDBCException*: Invalid syntax for the underlying DB

**✅ @TableGenerator**

🔍 **Purpose:**  
Provides a way to **generate primary keys using a separate table** that keeps track of ID values.

🤔 **Tricky/Indirect Questions:**

* How do you generate IDs when the database doesn't support sequences?
* Can Hibernate simulate sequence behavior in MySQL?

⚙️ **Key Attributes:**

* name: *(Required)* — Logical name
* table: *(Required)* — Table name used for ID tracking
* pkColumnName: — Column for primary key name
* valueColumnName: — Column for next ID value
* initialValue, allocationSize: Optional tuning

🏷️ **Annotation Level:**  
Field or Class Level (with @GeneratedValue)

🛠️ **Real-Time Example:**

java

@TableGenerator(

name = "emp\_gen", table = "id\_gen\_table",

pkColumnName = "gen\_name", valueColumnName = "gen\_val",

pkColumnValue = "emp\_id", allocationSize = 1

)

@GeneratedValue(strategy = GenerationType.TABLE, generator = "emp\_gen")

@Id

private Long id;

📝 **Extra Notes:**

* Useful for **legacy DBs** without sequence support.
* Slightly **slower than SEQUENCE** strategy.

🧪 **Similar Annotations:**

* @SequenceGenerator
* @GeneratedValue

🚨 **Possible Exceptions:**

* *SQLGrammarException*: Table or columns missing
* *DataIntegrityViolationException*: Duplicate PK issues

**✅ @Temporal**

🔍 **Purpose:**  
Used to specify the **format** in which a java.util.Date or Calendar is stored in the DB.

🤔 **Tricky/Indirect Questions:**

* How do you map just the date without time into DB?
* Why is my date being stored with time in Hibernate?
* How do you convert java.util.Date into proper DB formats?

⚙️ **Key Attributes:**

* value: *(Required)* — Accepts TemporalType.DATE, TIME, TIMESTAMP

🏷️ **Annotation Level:**  
Field or Getter

🛠️ **Real-Time Example:**

java

@Temporal(TemporalType.DATE)

private Date birthDate;

📝 **Extra Notes:**

* Mandatory when using legacy java.util.Date.
* For Java 8+ types (LocalDate, etc.), use converters or updated support in newer Hibernate versions.

🧪 **Similar Annotations:**

* @Column with SQL type
* Java 8 DateTime mappings via attribute converters

🚨 **Possible Exceptions:**

* *MappingException*: Missing or invalid @Temporal on legacy Date
* *DateTimeException*: Inconsistent conversion logic

**✅ @Enumerated**

🔍 **Purpose:**  
Maps Java enums to DB columns — either by name or ordinal value.

🤔 **Tricky/Indirect Questions:**

* How do you store enums in a readable form in DB?
* What happens if the enum order changes in code?
* Why is Hibernate storing 0, 1, 2 instead of enum names?

⚙️ **Key Attributes:**

* value: *(Optional)* — Either EnumType.STRING (default in newer versions) or EnumType.ORDINAL

🏷️ **Annotation Level:**  
Field or Getter

🛠️ **Real-Time Example:**

java

@Enumerated(EnumType.STRING)

private RoleType role;

📝 **Extra Notes:**

* Prefer EnumType.STRING to avoid bugs when enum order changes.
* Works directly with @Column.

🧪 **Similar Annotations:**

* @Convert — for custom enum conversion

🚨 **Possible Exceptions:**

* *IllegalArgumentException*: Enum name not matching DB string
* *MappingException*: Enum not mapped properly

**✅ @Version**

🔍 **Purpose:**  
Enables **optimistic locking** by maintaining a version number or timestamp.

🤔 **Tricky/Indirect Questions:**

* How do you avoid lost updates in concurrent transactions?
* How does Hibernate prevent overwriting when two users update the same row?
* Can you lock entities without using DB-level locks?

⚙️ **Key Attributes:**

* None. Just mark a field as @Version

🏷️ **Annotation Level:**  
Field or Getter

🛠️ **Real-Time Example:**

java

@Version

private int version;

📝 **Extra Notes:**

* Hibernate automatically checks version during updates.
* If version has changed → throws OptimisticLockException.

🧪 **Similar Annotations:**

* @OptimisticLocking — Hibernate-specific fine-tuning

🚨 **Possible Exceptions:**

* *OptimisticLockException*: When concurrent updates conflict
* *StaleObjectStateException*: Version mismatch

**✅ @Lob**

🔍 **Purpose:**  
Maps **large objects** like text or binary (CLOB/BLOB) into the DB.

🤔 **Tricky/Indirect Questions:**

* How do you store large descriptions or documents in Hibernate?
* Which annotation maps a large file or paragraph to DB?

⚙️ **Key Attributes:**

* None. But works with @Column for size customization

🏷️ **Annotation Level:**  
Field or Getter

🛠️ **Real-Time Example:**

java

@Lob

private String productDescription;

📝 **Extra Notes:**

* Use with byte[], String, InputStream (BLOB or CLOB).
* Can store files, images, or long text.

🧪 **Similar Annotations:**

* @Basic(fetch = FetchType.LAZY) — for loading control

🚨 **Possible Exceptions:**

* *JDBCException*: DB doesn't support LOB for that column
* *OutOfMemoryError*: With huge LOB fetches

**✅ @Transient**

🔍 **Purpose:**  
Marks a field that should be **ignored by Hibernate** (not persisted in DB).

🤔 **Tricky/Indirect Questions:**

* What if you want to use a field only during runtime but not in DB?
* Why is one of my fields not saved?

⚙️ **Key Attributes:**

* None

🏷️ **Annotation Level:**  
Field or Getter

🛠️ **Real-Time Example:**

java

@Transient

private int tempCalculation;

📝 **Extra Notes:**

* Opposite of @Column
* Used for derived values or calculated fields

🧪 **Similar Annotations:**

* transient keyword in Java — also ignored by serialization

🚨 **Possible Exceptions:**

* None directly, but missing @Transient might cause unwanted persistence

**✅ @OneToOne**

🔍 **Purpose:**  
Maps a **one-to-one relationship** between two entity classes. One row in Table A maps to exactly one row in Table B.

🤔 **Tricky/Indirect Questions:**

* How do you model tightly coupled entities like User and UserProfile?
* If one entity must have exactly one corresponding entity, which mapping do you use?
* How would you configure a bi-directional one-to-one association?

⚙️ **Key Attributes:**

* mappedBy: Indicates **inverse** (non-owning) side  
  *→ Used only on the non-owning side in bidirectional mappings*
* cascade: Defines cascading operations like PERSIST, MERGE, etc.
* fetch: Defaults to FetchType.EAGER, but can be LAZY
* optional: Marks whether the association is optional (default = true)

🏷️ **Annotation Level:**  
Field or Getter

🛠️ **Real-Time Example:**

java

@Entity

public class User {

@Id

private Long id;

@OneToOne(cascade = CascadeType.ALL)

@JoinColumn(name = "profile\_id")

private UserProfile profile;

}

java

@Entity

public class UserProfile {

@Id

private Long id;

private String address;

}

📝 **Extra Notes:**

* Use @JoinColumn to specify the **foreign key**.
* mappedBy must match the property name on the owning side.
* Use orphanRemoval=true to delete orphaned entities.

🧪 **Similar Annotations:**

* @ManyToOne: When many users can share a profile (hypothetical)
* @OneToMany: For collection relationships

🚨 **Possible Exceptions:**

* *MappingException*: If both sides declare mappedBy or both omit it
* *ConstraintViolationException*: Missing FK if not handled properly

**✅ @OneToMany**

🔍 **Purpose:**  
Maps a **one-to-many relationship**, i.e., one entity is related to multiple entities.

🤔 **Tricky/Indirect Questions:**

* How do you associate a Category with multiple Products?
* How do you model parent-child relationships where parent has multiple children?
* How would you define a collection mapping in Hibernate?

⚙️ **Key Attributes:**

* mappedBy: Specifies the field in the **owning side**
* cascade: Cascade operations
* fetch: Default is LAZY
* orphanRemoval: Deletes children if removed from collection

🏷️ **Annotation Level:**  
Collection (e.g., List<Product>)

🛠️ **Real-Time Example:**

java

@Entity

public class Category {

@Id

private Long id;

@OneToMany(mappedBy = "category", cascade = CascadeType.ALL)

private List<Product> products;

}

java

@Entity

public class Product {

@Id

private Long id;

@ManyToOne

@JoinColumn(name = "category\_id")

private Category category;

}

📝 **Extra Notes:**

* @OneToMany **without mappedBy** requires a @JoinTable.
* Always combine with @ManyToOne on the other side to form a bidirectional mapping.

🧪 **Similar Annotations:**

* @ManyToOne: The owning side
* @ManyToMany: If the relation is not unique

🚨 **Possible Exceptions:**

* *TransientObjectException*: If children are not saved/persisted properly
* *LazyInitializationException*: If accessed outside of session

**✅ @ManyToOne**

🔍 **Purpose:**  
Maps a **many-to-one relationship**, where multiple entities are related to one parent entity.

🤔 **Tricky/Indirect Questions:**

* Which mapping would you use to associate multiple Orders with one Customer?
* How do you ensure referential integrity from child to parent?

⚙️ **Key Attributes:**

* cascade: Optional cascading
* fetch: Defaults to EAGER (can change to LAZY)
* optional: Whether null is allowed

🏷️ **Annotation Level:**  
Field or Getter

🛠️ **Real-Time Example:**

java

@Entity

public class Order {

@Id

private Long id;

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "customer\_id")

private Customer customer;

}

java

@Entity

public class Customer {

@Id

private Long id;

private String name;

}

📝 **Extra Notes:**

* It's the **owning side**, so @JoinColumn is required.
* Always paired with @OneToMany if bidirectional.

🧪 **Similar Annotations:**

* @OneToMany: On the inverse side
* @OneToOne: If the association is one-to-one

🚨 **Possible Exceptions:**

* *ConstraintViolationException*: On missing FK or nullability
* *ObjectNotFoundException*: Lazy loading failure

**✅ @ManyToMany**

🔍 **Purpose:**  
Defines a **many-to-many relationship**, e.g., students enrolled in multiple courses and courses having many students.

🤔 **Tricky/Indirect Questions:**

* How do you model associations like User ↔ Role?
* Can two entities be related without owning each other?
* How do you manage a mapping that needs a **join table**?

⚙️ **Key Attributes:**

* mappedBy: For inverse side
* cascade: Optional
* fetch: Default = LAZY

🏷️ **Annotation Level:**  
Collection

🛠️ **Real-Time Example:**

java

@Entity

public class Student {

@Id

private Long id;

@ManyToMany

@JoinTable(name = "student\_course",

joinColumns = @JoinColumn(name = "student\_id"),

inverseJoinColumns = @JoinColumn(name = "course\_id"))

private Set<Course> courses;

}

java

@Entity

public class Course {

@Id

private Long id;

@ManyToMany(mappedBy = "courses")

private Set<Student> students;

}

📝 **Extra Notes:**

* Requires a **JoinTable** to connect both sides.
* Watch out for **infinite recursion** in toString() if both sides refer each other.

🧪 **Similar Annotations:**

* @OneToMany with extra entity (for customized mapping)

🚨 **Possible Exceptions:**

* *SQLIntegrityConstraintViolationException*: If join table not properly mapped
* *LazyInitializationException*: If not fetched properly