Hibernate Relationship Mappings

1. One-to-One (1-1) Mapping

 Definition: Each entity instance is associated with exactly one other entity instance.

Annotations Used

- `@OneToOne` → Defines 1-1 relationship
- `@JoinColumn` → Defines the foreign key

Example

```
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "address_id")  // FK in Employee table
    private Address address;
}

@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String city;
    private String state;
}
```

Use Case: Employee → Address, Person → Passport

2. One-to-Many (1-N) Mapping

 Definition: One entity is related to many other entities.

Annotations Used

- `@OneToMany` → Defines one-to-many relationship
- `mappedBy` → Defines the owning side
- `@JoinColumn` (optional if unidirectional)

Example

```
@Entity
public class Question {
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String qname;

    @OneToMany(mappedBy = "question", cascade = CascadeType.ALL)
    private List<Answer> answers;
}

@Entity
public class Answer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String answer;

    @ManyToOne
    @JoinColumn(name = "question_id")  // FK in Answer table
    private Question question;
}
```

Use Case: Question → Answers, Department → Employees

3. Many-to-One (N-1) Mapping

Definition: Many entities are related to one entity (reverse of 1-N).

Annotations Used

- `@ManyToOne` → Defines many-to-one relationship
- `@JoinColumn` → Defines the foreign key

Example

```
@Entity
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;

    @ManyToOne
    @JoinColumn(name = "dept_id")  // FK in Employee table
    private Department department;
}

@Entity
public class Department {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String deptName;
}
```

Use Case: Many Employees belong to One Department

4. Many-to-Many (N-N) Mapping

 Definition: Multiple entities are related to multiple other entities.

Annotations Used

- `@ManyToMany` → Defines many-to-many relationship
- `@JoinTable` → Defines a join (bridge) table

Example

```
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;

    @ManyToMany
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name = "student_id"),
        inverseJoinColumns = @JoinColumn(name = "course_id")
    )
    private List<Course> courses;
}

@Entity
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String title;

    @ManyToMany(mappedBy = "courses")
    private List<Student> students;
}
```

Use Case: Students ↔ Courses, Authors ↔ Books

Summary Table

Mapping Type	Cardinality	Example	Key Annotations
1-1	One ↔ One	Employee ↔ Address	`@OneToOne`, `@JoinColumn`

Mapping Type	Cardinality	Example	Key Annotations
1-N	One ↔ Many	Question ↔ Answers	`@OneToMany`, `mappedBy`
N-1	Many ↔ One	Employees ↔ Department	`@ManyToOne`, `@JoinColumn`
N-N	Many ↔ Many	Students ↔ Courses	`@ManyToMany`, `@JoinTable`

Why Mapping is Required in Hibernate

1. Object-Oriented vs Relational Mismatch

- Java works with **objects** (classes, fields, references).
- Databases work with **tables** (rows, columns, primary keys, foreign keys).
- This difference is called the **Object-Relational Impedance Mismatch**.

Example:

- In Java, `Employee` has a reference to `Department` as an object.
- In a database, `employee` table has a `dept_id` column (foreign key).
- Mapping bridges this gap by telling Hibernate **how an object reference translates to a foreign key**.

2. Automates Relationship Handling

Without mapping, developers would need to manually:

- Write **JOIN queries**
- Handle **foreign keys**
- Manage **insert/update/delete cascades**

Mapping lets Hibernate do this automatically based on annotations.

 Example:

- `@OneToMany` automatically means “create a foreign key in child table”.
- When you save a `Question`, Hibernate also saves its `Answers`.

3. Simplifies Code

With mapping, developers work with **Java objects only**, instead of thinking about SQL joins and FK constraints.

Example:

```
Question q = new Question();
Answer a = new Answer();
q.getAnswers().add(a);
a.setQuestion(q);
session.save(q);
```

Hibernate automatically persists both objects and the relationship.
No need to manually insert into the foreign key column.

4. Ensures Data Consistency

Mappings handle **cascades** and **orphan removal**, ensuring the database is consistent.

Example:

- If a `Department` is deleted, all its `Employees` can also be deleted if cascade is set.
- This prevents dangling rows.

5. Supports Complex Relations (N-N)

Some relations like **many-to-many** are tricky in plain SQL (requires a join table).
With mapping, Hibernate manages it automatically.

Example:
`Student ↔ Course` → Hibernate creates and maintains the `student_course` join table without manual queries.

6. Reusability & Flexibility

Mappings allow the same entity to be reused across multiple relationships without rewriting SQL.
Also, if DB schema changes slightly (column name, join table), only mapping annotations need updating — **no change in business logic code**.

Analogy for Students

Think of Hibernate mapping as a **translator**:

- Java speaks in **objects**
- Database speaks in **tables**
- Mapping is the **dictionary** that helps them understand each other.

In Summary:
Mapping is required in Hibernate to:

- Bridge the gap between Java objects and database tables
- Automate foreign key/relationship handling
- Reduce boilerplate SQL code
- Maintain data consistency
- Support complex relationships like Many-to-Many