Unit 3- Hibernate Framework

What is ORM?

ORM (Object-relational mapping) is a programming technique for mapping application domain model objects to relational database tables. Hibernate is a Java-based ORM tool that provides a framework for mapping application domain objects to relational database tables and vice versa.

What is the Java Persistence API (JPA)?

The Java Persistence API (JPA) is a Java specification for accessing, persisting, and managing data between Java objects/classes and a relational database. JPA acts as a bridge between object-oriented domain models and relational database systems, making it easier for developers to work with data in their applications.

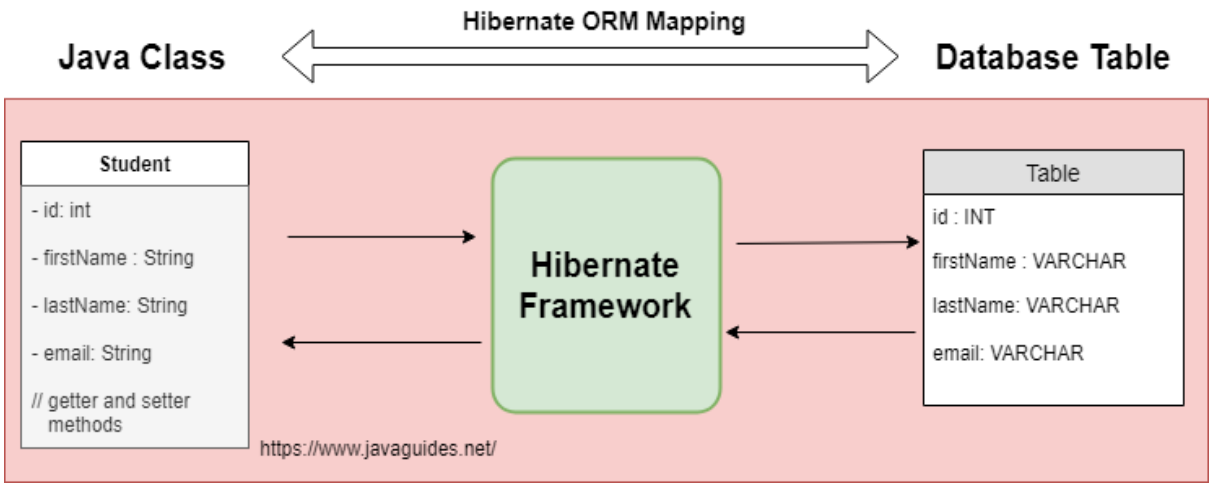
JPA is not an implementation but a specification. Various ORM tools, such as Hibernate, EclipseLink, and Apache OpenJPA, provide implementations of the JPA specification. This allows developers to switch between these implementations if needed without changing the application code that uses JPA.

What is the Hibernate Framework?

Hibernate is a Java-based ORM tool that provides a framework for mapping application domain objects to relational database tables and vice versa.

Hibernate is the most popular JPA implementation and one of the most popular Java ORM frameworks. Hibernate is an additional layer on top of JDBC and enables you to implement a database-independent persistence layer. It provides an object-relational mapping implementation that maps your database records to Java objects and generates the required SQL statements to replicate all operations to the database.

Example: The diagram below shows an Object-Relational Mapping between the Student Java class and the student's table in the database.



Key Features of Hibernate

Transparent Persistence: Hibernate manages the persistence of objects without requiring significant changes to how those objects are designed.

Database Independence: Applications built with Hibernate are portable across databases with minimal changes.

Performance Optimization: Features like caching and lazy loading help optimize performance by reducing database access.

Powerful Query Language: Hibernate Query Language (HQL) offers an object-oriented extension to SQL, easing data manipulation and retrieval.

Automatic Schema Generation: Hibernate can generate database schemas based on the object model, simplifying initial setup and migrations.

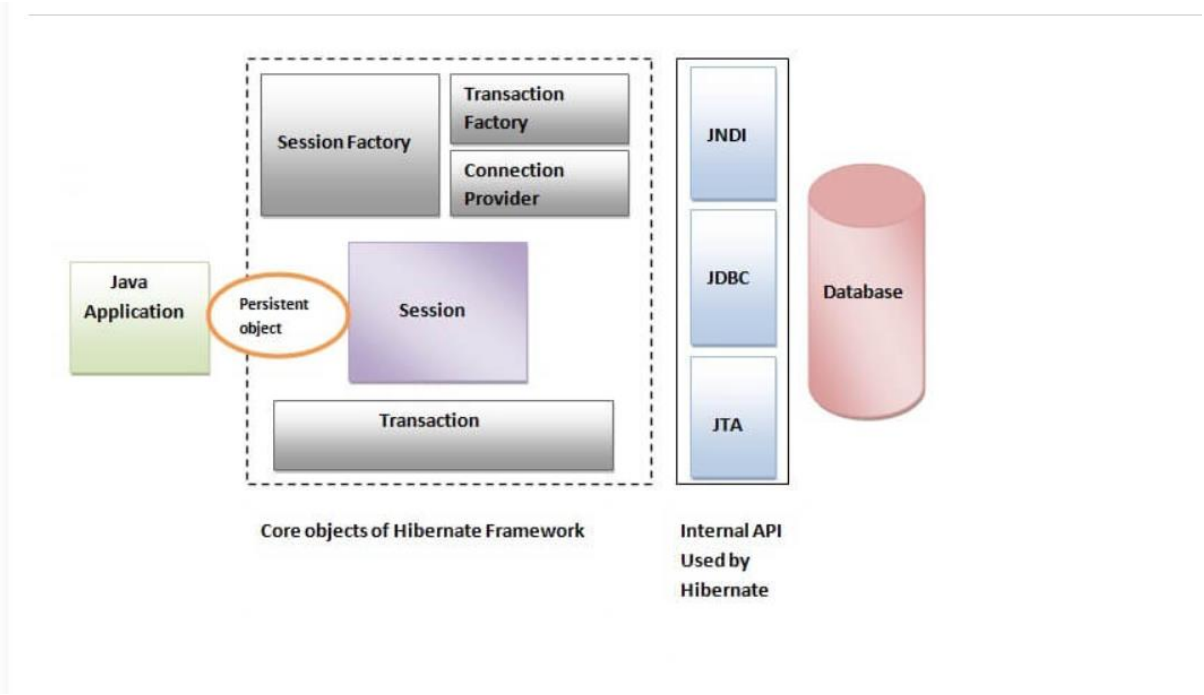
How does Hibernate relate to JDBC?

Hibernate internally uses JDBC for all database communications.

Hibernate acts as an additional layer on top of JDBC and enables you to implement a database-independent persistence layer:

Hibernate Architecture

The Hibernate architecture includes many objects such as persistent object, session factory, transaction factory, connection factory, session, transaction etc.



SessionFactory

The SessionFactory is a factory of session and client of ConnectionProvider. It holds second level cache (optional) of data. The org.hibernate.SessionFactory interface provides factory method to get the object of Session.

Session

The session object provides an interface between the application and data stored in the database. It is a short-lived object and wraps the JDBC connection. It is factory of Transaction, Query and Criteria. It holds a first-level cache (mandatory) of data. The org.hibernate.Session interface provides methods to insert, update and delete the object. It also provides factory methods for Transaction, Query and Criteria.

Transaction

The transaction object specifies the atomic unit of work. It is optional. The org.hibernate.Transaction interface provides methods for transaction management.

ConnectionProvider

It is a factory of JDBC connections. It abstracts the application from DriverManager or DataSource. It is optional.

TransactionFactory

It is a factory of Transaction. It is optional.

What are the advantages of Hibernate over JDBC?

Simplified Code: Hibernate significantly reduces boilerplate code required in JDBC, making the codebase cleaner and more readable.

Advanced Mapping Features: Unlike JDBC, Hibernate fully supports object-oriented features such as inheritance, associations, and collections.

Transaction Management: Hibernate seamlessly handles transaction management, requiring transactions for most operations, which contrasts with JDBC's manual transaction handling through commit and rollback.

Exception Handling: Hibernate abstracts boilerplate try-catch blocks by converting JDBC's checked SQLExceptions into unchecked JDBCException or HibernateException, simplifying error handling.

Object-Oriented Query Language: HQL (Hibernate Query Language) offers an object-oriented API, which aligns it more with Java programming concepts than JDBC's need for native SQL queries.

Caching for Performance: Hibernate's support for caching enhances performance, a feature not available with JDBC, where queries are directly executed without caching.

Database Synchronization: Hibernate can automatically generate database tables, offering greater flexibility than JDBC, which requires pre-existing tables.

Flexible Connection Management: Hibernate allows for both JDBC-like connections and JNDI DataSource connections with pooling, which is essential for enterprise applications and not supported by JDBC.

ORM Tool Independence: By supporting JPA annotations, Hibernate-based applications are not tightly bound to Hibernate and can switch ORM tools more easily than JDBC-based applications, which are closely coupled with the database.

Introduction to HQL

HQL is designed to query the object model and not the relational model, making it database-independent and more aligned with the application's object-oriented nature.

Key Features of HQL

- **Object-Oriented**: Queries are written in terms of the entity model, not the database schema.
- **Database Independent**: Queries are independent of the underlying database.
- **Powerful and Flexible**: Supports CRUD operations, joins, aggregations, and subqueries.

Basic HQL Syntax

SELECT Query

```
String hql = "FROM Student";
Session session = HibernateUtil.getSessionFactory().openSession();
List<Student> students = session.createQuery(hql, Student.class).getResultList();
students.forEach(System.out::println);
session.close();
```

This query retrieves all `Student` entities from the database.

WHERE Clause

```
String hql = "FROM Student S WHERE S.firstName = :firstName";
Session session = HibernateUtil.getSessionFactory().openSession();
Query<Student> query = session.createQuery(hql, Student.class);
query.setParameter("firstName", "John");
List<Student> students = query.getResultList();
students.forEach(System.out::println);
session.close();
```

This query retrieves `Student` entities with a specific first name.

ORDER BY Clause

```
String hql = "FROM Student S ORDER BY S.lastName ASC";
Session session = HibernateUtil.getSessionFactory().openSession();
List<Student> students = session.createQuery(hql, Student.class).getResultList();
students.forEach(System.out::println);
session.close();
```

This query retrieves all `Student` entities and sorts them by last name in ascending order.

JOIN Clause

Assume we have another entity `Course` and a `ManyToMany` relationship with `Student`.

```
@Entity
@Table(name = "course")
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;

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

    // Constructors, getters, setters, and toString() method
}
```

```
@Entity
@Table(name = "student")
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String firstName;
    private String lastName;
    private String email;

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

    // Constructors, getters, setters, and toString() method
}
```

```
String hql = "SELECT S FROM Student S JOIN S.courses C WHERE C.name = :courseName";
Session session = HibernateUtil.getSessionFactory().openSession();
Query<Student> query = session.createQuery(hql, Student.class);
query.setParameter("courseName", "Mathematics");
List<Student> students = query.getResultList();
students.forEach(System.out::println);
session.close();
```

This query retrieves `Student` entities enrolled in a specific course.

Advanced HQL Features

Aggregations

HQL supports aggregate functions like `COUNT`, `SUM`, `AVG`, `MAX`, and `MIN`.

```
String hql = "SELECT COUNT(S.id) FROM Student S";
Session session = HibernateUtil.getSessionFactory().openSession();
Long count = (Long) session.createQuery(hql).getSingleResult();
System.out.println("Total Students: " + count);
session.close();
```

This query counts the total number of `Student` entities.

Group By and Having

```
String hql = "SELECT S.lastName, COUNT(S.id) FROM Student S GROUP BY S.lastName HAVING COUNT(S.id) > 1";
Session session = HibernateUtil.getSessionFactory().openSession();
List<Object[]> results = session.createQuery(hql).getResultList();
for (Object[] result : results) {
    System.out.println("Last Name: " + result[0] + ", Count: " + result[1]);
}
session.close();
```

This query groups `Student` entities by last name and retrieves the count of students with the same last name, only if the count is greater than one.

Subqueries

```
String hql = "FROM Student S WHERE S.id IN (SELECT C.student.id FROM Course C WHERE C.name = :courseName)";
Session session = HibernateUtil.getSessionFactory().openSession();
Query<Student> query = session.createQuery(hql, Student.class);
query.setParameter("courseName", "Mathematics");
List<Student> students = query.getResultList();
students.forEach(System.out::println);
session.close();
```

This query retrieves `Student` entities enrolled in a specific course using a subquery.

HQL Functions and Operators

String Functions

- `concat()`
- `length()`
- `substring()`
- `lower()`
- `upper()`
- `trim()`

HCQL

The criteria query API lets us build nested, structured query expressions in Java, providing a compile-time syntax checking that is not possible with a query language like HQL or SQL.

1. The simplest example of a criteria query is one with no optional parameters or restrictions—the criteria query will simply return every object that corresponds to the class.

```
Criteria crit = session.createCriteria(Product.class);
List<Product> results = crit.list();
```

2. Using Restrictions

The Criteria API makes it easy to use restrictions in your queries to selectively retrieve objects; for instance, your application could retrieve only products with a price over $30. You may add these restrictions to a Criteria object with the add() method.

2.1. Restrictions.eq()

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.eq("description","Mouse"));
List<Product> results = crit.list()
```

2.2. Restrictions.ne()

To retrieve objects that have a property value "`not equal to`" your restriction, use the `ne()` method on `Restrictions`, as follows:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.ne("description","Mouse"));
List<Product> results = crit.list()
```

2.3. Restrictions.like() and Restrictions.ilike()

Instead of searching for exact matches, we can retrieve all objects that have a property matching part of given pattern. To do this, we need to create an SQL LIKE clause, with either the `like()` or the `ilike()` method. The `ilike()` method is case-insensitive.

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.like("name","Mou%",MatchMode.ANYWHERE));
List<Product> results = crit.list();
```

2.4. Restrictions.isNull() and Restrictions.isNotNull()

The `isNull()` and `isNotNull()` restrictions allow you to do a search for objects that have (or do not have) null property values.

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.isNull("name"));
List<Product> results = crit.list();
```

2.5. Restrictions.gt(), Restrictions.ge(), Restrictions.lt() and Restrictions.le()

Several of the restrictions are useful for doing math comparisons. The greater-than comparison is `gt()`, the greater-than-or-equal-to comparison is `ge()`, the less-than comparison is `lt()`, and the less-than-or-equal-to comparison is `le()`.

We can do a quick retrieval of all products with prices over $25 like this, relying on Java's type promotions to handle the conversion to `Double`:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.gt("price", 25.0));
List<Product> results = crit.list();
```

2.6. Combining Two or More Restrictions

Moving on, we can start to do more complicated queries with the Criteria API. For example, we can combine AND and OR restrictions in logical expressions. When we add more than one constraint to a criteria query, it is interpreted as an AND, like so:

```
Criteria crit = session.createCriteria(Product.class);
crit.add(Restrictions.lt("price",10.0));
crit.add(Restrictions.ilike("description","mouse", MatchMode.ANYWHERE));
List<Product> results = crit.list();
```

If we want to have two restrictions that return objects that satisfy either or both of the restrictions, we need to use the `or()` method on the Restrictions class, as follows:

```
Criteria crit = session.createCriteria(Product.class);
Criterion priceLessThan = Restrictions.lt("price", 10.0);
Criterion mouse = Restrictions.ilike("description", "mouse", MatchMode.ANYWHERE);
LogicalExpression orExp = Restrictions.or(priceLessThan, mouse);
crit.add(orExp);
List results=crit.list();
```

3. Pagination

One common application pattern that criteria can address is pagination through the result set of a database query. There are two methods on the `Criteria` interface for paging, just as there are for Query: `setFirstResult()` and `setMaxResults()`.

The `setFirstResult()` method takes an integer that represents the first row in your result set, starting with row 0. You can tell Hibernate to retrieve a fixed number of objects with the `setMaxResults()` method. Using both of these together, we can construct a paging component in our web or Swing application.

```
Criteria crit = session.createCriteria(Product.class);
crit.setFirstResult(1);
crit.setMaxResults(20);
List<Product> results = crit.list();
```

4. Fetch a Single Result

5. Distinct Results

6. Sorting

7. Associations or JOINS

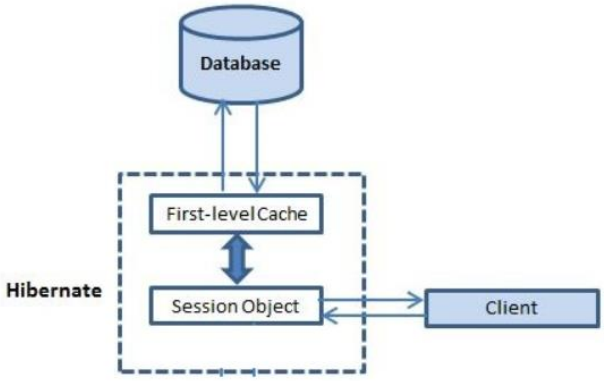
Hibernate Caching

Caching is a facility provided by ORM frameworks that **helps the users to get fast-running web applications** while **helping the framework itself to reduce the number of queries** made to the database in a single transaction. Hibernate achieves the second goal by implementing the first-level cache.

1. Available Only through *Session* Object

First level cache in hibernate is enabled by default and we do not need to do anything to get this functionality working. In fact, *we can not disable it even forcefully.*

> **object**. As we know the *Session* object is created on-demand from SessionFactory and **it is lost, once the current session is closed**. Similarly, the first-level cache associated with the *Session* object is available only till the session object is live.
>
> *The **first-level cache is available to Session object only** and is not accessible to any other session object in any other part of the application.*

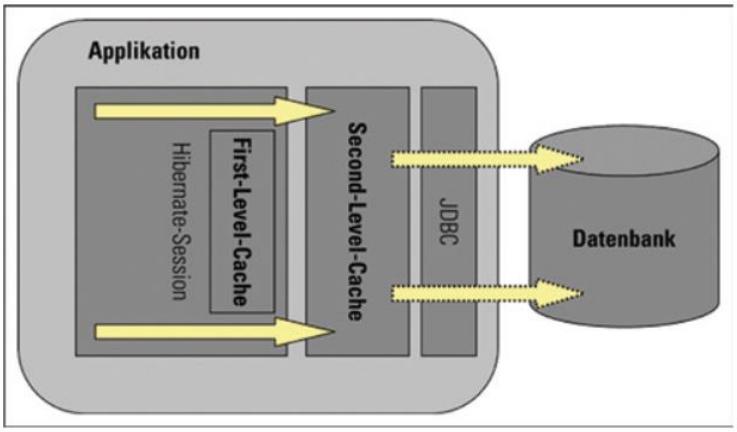


1. Caching in Hibernate

Hibernate also provides caching functionality in two layers.

- **First-level Cache**: Session object holds the first level cache data. It is enabled by default. The first level cache data will not be available to entire application. An application can use many session object.
- **Second-level Cache**: SessionFactory object holds the second level cache data. The data stored in the second level cache will be available to entire application. But we need to enable it explicitely.

- the entities stored in the second level cache will be available to all the sessions created using that particular session factory.
- **once the *SessionFactory* is closed, all cache associated with it die** and the cache manager also closes down.
- if we have two instances of *SessionFactory* (highly discouraged), we will have two cache managers in our application and while accessing cache stored in a physical store, we might get unpredictable results like cache-miss.



Hibernate first and second level cache

2. How does Second Level Cache Work in Hibernate?

Let us write all the facts point by point to better understand the internal working related to second-level caches.

1. Whenever hibernate session tries to load an entity, the very first place it looks for a cached copy of the entity in first-level cache (associated with a particular hibernate *Session*).
2. If a cached copy of the entity is present in first-level cache, it is returned as the result of *load()* method.
3. If there is no cached entity in the first-level cache, then the second-level cache is looked up for the cached entity.
4. If second-level cache has the cached entity, it is returned as the result of *load()* method. But, before returning the entity, it is stored in first level cache also so that the next invocation to *load()* method for that entity will return the entity from the first level cache itself, and there will not be need to go to the second level cache again.
5. If the entity is not found in first level cache and second level cache also, then a database query is executed and the entity is stored in both cache levels, before returning as the response to load() method.
6. Second-level cache validates itself for modified entities if the modification has been done through hibernate session APIs.
7. If some user or process makes changes directly in the database, there is no way that the second-level cache update itself until "*timeToLiveSeconds*" duration has passed

for that cache region. In this case, it is a good idea to invalidate the whole cache and let hibernate build its cache once again. You can use *sessionFactory.evictEntity()* in a loop to invalidate the whole Hibernate second-level cache.

Thankfully, Hibernate is designed to use a connection pool by default, an internal implementation. However, **Hibernate's built-in connection pooling isn't designed for production use**. In production, we would use an external connection pool by using either a database connection provided by JNDI or an external connection pool configured via parameters and classpath.

C3P0 is an example of an **external connection pool**.

To *configure c3p0 with hibernate*, we need to add Hibernate's c3p0 connection provider hibernate-c3p0 as dependency in the pom.xml. Please note that the version of the hibernate-c3p0 dependency should match Hibernate's compatible version.

Different Types of Association Mappings

The below table shows how we can select the side of the relationship that should be made the owner of a bi-directional association. **Remember that to make one entity as *owner of the association*, we must mark the other entity as being *mapped by* the owner entity.**

Association Types	Options/Usage
One-to-one	**Either end can be made by the owner**, but one (and only one) of them should be; if we don't specify this, we will end up with a circular dependency.
One-to-many	The ***many* end must be made the owner** of the association.
Many-to-one	This is the same as the one-to-many relationship viewed from the opposite perspective, so the same rule applies: the **many end must be made the owner** of the association.
Many-to-many	**Either end of the association can be made the owner.**

One-To-Many Relationship

In a one-to-many relationship, an entity has a reference to one or many instances of another entity.

A common example is the relationship between a *Department* and its *Employee*s. Each *Department* has many *Employee*s, but each *Employee* belongs to one *Department* only.

Let's take a look at how to define a one-to-many unidirectional association:

```
@Entity
public class Department {

    @Id
    private Long id;

    @OneToMany
    @JoinColumn(name = "department_id")
    private List<Employee> employees;
}

@Entity
public class Employee {

    @Id
    private Long id;
}
```

Here, the *Department* entity has a reference to a list of *Employee* entities. The *@OneToMany* annotation specifies that this is a one-to-many association. The *@JoinColumn* annotation specifies the foreign key column in the *Employee* table referencing the *Department* table.

Many-To-One Relationship

In a many-to-one relationship, many instances of an entity are associated with one instance of another entity.

For example, let's consider *Student* and *School.* Each *Student* can be enrolled in one *School* only, but each *School* can have multiple *Student*s.

```
@Entity
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToOne
    @JoinColumn(name = "school_id")
    private School school;
}

@Entity
public class School {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
}
```

In this case, we have a many-to-one unidirectional association between *Student* and *School* entities. The *@ManyToOne* annotation specifies that each student can be enrolled in only one school, and the *@JoinColumn* annotation specifies the foreign key column name to join the *Student* and *School* entities.

One-To-One Relationship

In a one-to-one relationship, an instance of an entity is associated with only one instance of another entity.

A common example is the relationship between an *Employee* and a *ParkingSpot*.
Each *Employee* has a *ParkingSpot*, and each *ParkingSpot* belongs to one *Employee*.

```
@Entity
public class Employee {

    @Id
    private Long id;

    @OneToOne
    @JoinColumn(name = "parking_spot_id")
    private ParkingSpot parkingSpot;

}

@Entity
public class ParkingSpot {

    @Id
    private Long id;

}
```

Here, the *Employee* entity has a reference to the *ParkingSpot* entity. The *@OneToOne* annotation specifies that this is a one-to-one association. The *@JoinColumn* annotation specifies the foreign key column in the *Employee* table that references the *ParkingSpot* table

In a many-to-many relationship, many instances of an entity are associated with many instances of another entity.

Suppose we have two entities – *Book* and *Author*. Each *Book* can have multiple *Author*s, and each *Author* can write multiple *Book*s. In JPA, this relationship is represented using the *@ManyToMany* annotation.

Let's take a look at how to define a many-to-many unidirectional association:

```
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String title;

    @ManyToMany
    @JoinTable(name = "book_author",
            joinColumns = @JoinColumn(name = "book_id"),
            inverseJoinColumns = @JoinColumn(name = "author_id"))
    private Set<Author> authors;

}

@Entity
public class Author {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
}
```

Here, we can see a many-to-many unidirectional association between *Book* and *Author* entities. The *@ManyToMany* annotation specifies that each *Book* can have multiple *Author*s, and each *Author* can write multiple *Book*s. The *@JoinTable* annotation specifies the name of the join table and the foreign key columns to join the *Book* and *Author* entities.

Bidirectional Associations

A bidirectional association is a relationship between two entities where each entity has a reference to the other.

In order to define bidirectional associations, we use the *mappedBy* attribute in the *@OneToMany* and *@ManyToMany* annotations. However, it's important to note that only relying on unidirectional associations may not be sufficient, as bidirectional associations provide additional benefits.

One-To-Many Bidirectional Association

In a one-to-many bidirectional association, an entity has a reference to another entity. Additionally, the other entity has a collection of references to the first entity.

For instance, a *Department* entity has a collection of *Employee* entities. Meanwhile, an *Employee* entity has a reference to the *Department* entity it belongs.

Let's take a look at how to create a one-to-many bidirectional association:

```
@Entity
public class Department {

    @OneToMany(mappedBy = "department")
    private List<Employee> employees;

}

@Entity
public class Employee {

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;

}
```

In the Department entity, we use the *@OneToMany* annotation to specify the relationship between the *Department* entity and the *Employee* entity. The *mappedBy* attribute specifies the name of the attribute in the *Employee* entity that owns the relationship. In this case, the *Department* entity doesn't own the relationship, so we specify *mappedBy = "department"*.

In the *Employee* entity, we use the *@ManyToOne* annotation to specify the relationship between the *Employee* entity and the *Department* entity. The *@JoinColumn* annotation specifies the name of the foreign key column in the *Employee* table referencing the *Department* table.

Many-To-Many Bidirectional Association

When dealing with a many-to-many bidirectional association, it's important to understand that each entity involved will have a collection of references to the other entity.

To illustrate this concept, let's consider the example of a *Student* entity that has a collection of *Course* entities and a *Course* entity that in turn has a collection of *Student* entities. By establishing such a bidirectional association, we enable both entities to be aware of each other and make it easier to navigate and manage their relationship.

Here's an example of how to create a many-to-many bidirectional association:

```
@Entity
public class Student {

    @ManyToMany(mappedBy = "students")
    private List<Course> courses;

}

@Entity
public class Course {

    @ManyToMany
    @JoinTable(name = "course_student",
        joinColumns = @JoinColumn(name = "course_id"),
        inverseJoinColumns = @JoinColumn(name = "student_id"))
    private List<Student> students;

}
```

In the *Student* entity, we use the *@ManyToMany* annotation to specify the relationship between the *Student* entity and the *Course* entity. The *mappedBy* attribute specifies the attribute's name in the *Course* entity that owns the relationship. In this case, the *Course* entity owns the relationship, so we specify *mappedBy = "students"*.

In the Course entity, we use the @ManyToMany annotation to specify the relationship between the Course entity and the Student entity. The @JoinTable annotation specifies the name of the join table that stores the relationship.

Unidirectional vs. Bidirectional Association

	Unidirectional Association	Bidirectional Association
Definition	A relationship between two tables where one table has a foreign key that references the primary key of another table.	A relationship between two tables where both tables have a foreign key that references the primary key of the other table.
Navigation	Only navigable in one direction - from the child table to the parent table.	Navigable in both directions - from either table to the other.
Performance	Generally faster due to simpler table structure and fewer constraints.	Generally slower due to additional constraints and table structure complexity.
Data Consistency	Ensured by the foreign key constraint in the child table referencing the primary key in the parent table.	Ensured by the foreign key constraint in the child table referencing the primary key in the parent table.
Flexibility	Less flexible as changes in the child table may require changes to the parent table schema.	More flexible as changes in either table can be made independently without affecting the other.