ORM:

Advantages of ORM (Object-Relational Mapping)

1. **Object-oriented data handling**
    - You interact with database tables as **objects**, instead of writing raw SQL queries.
    - CRUD operations (Create, Read, Update, Delete) are simpler and more maintainable.

```java
// Using Hibernate (Java)
Session session = sessionFactory.openSession();
User u = session.get(User.class, 1); // Fetch user with id=1 as object
u.setName("Ahmad");
session.update(u); // Save changes
session.close();
```

2. **Automatic schema management**
    - Many ORMs can **create tables automatically** if they don't exist.
    - Keep the database schema synchronized with your object model.

```java
@Entity
class User {
    @Id
    private int id;
    private String name;
}
```

3. **Relationship management**

- Supports **one-to-one, one-to-many, many-to-many** relationships naturally.
- You can navigate relationships via objects instead of writing complex join queries.

```java
@OneToMany(mappedBy="user")
private List<Order> orders; // Fetch all orders for a user
```

4. **Transaction management**
   - ORMs provide easy handling of **transactions**, including `commit` and `rollback`.
   - Ensures data consistency and makes handling errors simpler.

```java
Transaction tx = session.beginTransaction();
try {
    session.save(u);
    tx.commit(); // changes are saved
} catch (Exception e) {
    tx.rollback(); // undo changes if error occurs
}
```

5. **Connection pooling**
   - ORMs often provide **built-in connection pooling**, which reuses database connections instead of opening/closing them repeatedly.
   - This improves performance and resource management.

```java
// Connections are reused automatically via connection pool
Session session = sessionFactory.openSession(); // doesn't open a new DB connection each time
```

6. **Database abstraction**
    - Code is less tied to a specific database vendor.
    - Switching databases requires minimal changes

Disadvantages of ORM

1. **Performance Overhead**
    - Abstracting SQL can introduce extra queries or slower performance for complex operations.
2. **Learning Curve**
    - Developers need to understand ORM framework specifics (like Hibernate annotations, session management).
3. **Complex Queries**
    - Writing advanced queries (joins, aggregates) can be harder than plain SQL.
4. **Not Always Optimal**
    - Automatically generated SQL may not be as efficient as hand-written SQL for large datasets.
5. **Overkill for Small Projects**
    - For very simple applications, ORM adds unnecessary complexity.

Use ORM when:

- You want object-oriented DB access, easy transaction and relationship management, and database independence.

Avoid ORM when:

- Performance-critical queries or very simple projects where SQL is straightforward.

Clarification on structure

- **ORM is not an architecture** like MVC.
- It is a **technique or tool** for working with databases in an object-oriented way.
- Usually, ORM is used **inside the model/data access layer** of an application.

Java Beans: (Facade) Design pattern

It's a Class/interface which simplifies the representation of other classes.
Properties:

- Default constructor.
- Private data members.
- Public getter setters
- Serializable.

Why Use JavaBeans

1. **Reusability** – Components can be reused across applications.
2. **Encapsulation and consistency** – Standardized property access.
3. **Persistence** – Serializable for storage or network transfer.

Facade is about simplification and encapsulation.

Serialization

Why Serialization is Needed

In object-oriented programming, every object instance has its own **state** (its data/fields), which lives in memory. These objects may reference other objects through pointers. The object information is **scattered in memory**, making it impossible to directly save or transmit across systems.

Serialization solves this problem by converting objects into a **linear sequence of bytes or characters**, which can be:

- Saved to a file for persistence.
- Sent over a network for distributed systems.
- Easily reconstructed later (deserialization).

This process ensures that an object's state can **survive beyond its in-memory existence** and be used in other contexts.

How Serialization Works

Serialization converts an object's state into a **format suitable for storage or transmission**, then deserialization reconstructs the object.

- **Without pointers:** Objects do not reference other objects. Serialization is straightforward: just convert the fields into a sequence.

```
class User implements Serializable {
    String name;
    int age;
}
```

- **With pointers (object references):** Objects reference other objects (nested objects, lists, or trees). Serialization needs to maintain these references so that the object graph can be reconstructed correctly.

```
class Address implements Serializable {
    String city;
}

class User implements Serializable {
    String name;
    Address address;
}
```

Binary Serialization

- Converts the object to **binary bytes**.
- **Advantages:**
    - Compact, fast, efficient for storage and network transmission.
- **Disadvantages:**
    - Not human-readable.
    - Harder to debug.
- **Usage:**
    - Network communication, distributed systems, storing large datasets.

Text (Character) Serialization

- Converts the object to **human-readable characters**, usually in **JSON, XML, or CSV**.
- **Advantages:**
    - Easy to read/debug.
    - Interoperable across platforms and languages.
- **Disadvantages:**
    - Bigger size, slower than binary.
- **Usage:**
    - REST APIs, configuration files, logs, cross-platform communication.