Object-Relational Mapping (O/R Mapping) Implementation:

Object-Relational Mapping (ORM) is a technique that connects object-oriented programming with relational databases. Here's a comprehensive demonstration of ORM implementation using JPA (Java Persistence API) and Hibernate as the provider.

1. Basic Entity Mapping

```
@Entity
@Table(name = "employees")
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "emp_id")
    private Long id;

    @Column(name = "first_name", nullable = false, length = 50)
    private String firstName;

    @Column(name = "last_name", nullable = false, length = 50)
    private String lastName;

    @Column(nullable = false, unique = true)
    private String email;

    @Column(precision = 10, scale = 2)
    private BigDecimal salary;

    @Temporal(TemporalType.DATE)
    @Column(name = "hire_date")
    private Date hireDate;

    @Enumerated(EnumType.STRING)
    private EmployeeStatus status;

    @Lob
    private byte[] profilePicture;

    // Constructors, getters, setters
}

public enum EmployeeStatus {
    ACTIVE, ON_LEAVE, TERMINATED
}
```

2. Relationship Mapping
One-to-Many Relationship

```
@Entity
public class Department {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    @OneToMany(mappedBy = "department", cascade = CascadeType.ALL,
orphanRemoval = true)
    private List<Employee> employees = new ArrayList<>();

    // Helper method for bidirectional relationship
    public void addEmployee(Employee employee) {
        employees.add(employee);
        employee.setDepartment(this);
    }

    // Constructors, getters, setters
}

@Entity
public class Employee {
    // ... previous fields

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "dept_id")
    private Department department;

    // ... rest of the class
}
```

Many-to-Many Relationship

```
@Entity
public class Project {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @ManyToMany
    @JoinTable(
        name = "employee_project",
        joinColumns = @JoinColumn(name = "project_id"),
        inverseJoinColumns = @JoinColumn(name = "employee_id")
    )
    private Set<Employee> employees = new HashSet<>();

    // Constructors, getters, setters
}

@Entity
public class Employee {
    // ... previous fields

    @ManyToMany(mappedBy = "employees")
    private Set<Project> projects = new HashSet<>();

    // ... rest of the class
}
```

One-to-One Relationship

```
@Entity
```

```
public class EmployeeDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String address;
    private String phoneNumber;

    @OneToOne
    @JoinColumn(name = "employee_id")
    private Employee employee;

    // Constructors, getters, setters
}

@Entity
public class Employee {
    // ... previous fields

    @OneToOne(mappedBy = "employee", cascade = CascadeType.ALL)
    private EmployeeDetails details;

    // ... rest of the class
}
```

3. Inheritance Mapping Strategies
Single Table Inheritance

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "vehicle_type", discriminatorType =
DiscriminatorType.STRING)
public abstract class Vehicle {
    @Id
    @GeneratedValue
    private Long id;

    private String manufacturer;
    private String model;
    private Integer year;

    // Constructors, getters, setters
}

@Entity
@DiscriminatorValue("CAR")
public class Car extends Vehicle {
    private Integer numberOfDoors;
    private String fuelType;

    // Constructors, getters, setters
}

@Entity
@DiscriminatorValue("TRUCK")
public class Truck extends Vehicle {
    private Double loadCapacity;
    private Integer numberOfAxles;
    // Constructors, getters, setters
}
```

Joined Table Inheritance

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Payment {
    @Id
    @GeneratedValue
    private Long id;

    private Double amount;
    private LocalDateTime paymentDate;

    // Constructors, getters, setters
}

@Entity
public class CreditCardPayment extends Payment {
    private String cardNumber;
    private String expirationDate;
    private String cvv;

    // Constructors, getters, setters
}

@Entity
public class BankTransferPayment extends Payment {
    private String accountNumber;
    private String bankName;
    private String reference;

    // Constructors, getters, setters
}
```

4. Embedded Objects

```
@Embeddable
public class Address {
    private String street;
    private String city;
    private String state;
    private String zipCode;
    private String country;

    // Constructors, getters, setters
}

@Entity
public class Customer {
    @Id
    @GeneratedValue
    private Long id;

    private String name;
    private String email;
    @Embedded
    private Address shippingAddress;

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name = "street", column = @Column(name =
"billing_street")),
        @AttributeOverride(name = "city", column = @Column(name =
"billing_city"))
    })
    private Address billingAddress;

    // Constructors, getters, setters
}
```

5. Entity Lifecycle Callbacks

```
@Entity
@EntityListeners(AuditListener.class)
public class Product {
    @Id
    @GeneratedValue
    private Long id;

    private String name;
    private BigDecimal price;

    @Column(name = "created_at")
    private LocalDateTime createdAt;

    @Column(name = "updated_at")
    private LocalDateTime updatedAt;

    @PrePersist
    public void prePersist() {
        this.createdAt = LocalDateTime.now();
        this.updatedAt = LocalDateTime.now();
    }

    @PreUpdate
    public void preUpdate() {
        this.updatedAt = LocalDateTime.now();
    }

    // Constructors, getters, setters
}

// External listener class
public class AuditListener {
    @PrePersist
    public void beforePersist(Object entity) {
        System.out.println("About to persist: " + entity);
    }

    @PostPersist
    public void afterPersist(Object entity) {
        System.out.println("Persisted: " + entity);
    }
}
```

6. Using the ORM in Practice

```
public class OrmDemoService {
    @PersistenceContext
    private EntityManager entityManager;

    @Transactional
    public void performOrmOperations() {
        // Create and persist a new department
        Department itDept = new Department("IT");
        entityManager.persist(itDept);

        // Create and persist employees
        Employee emp1 = new Employee("John", "Doe",
"john.doe@example.com");
        Employee emp2 = new Employee("Jane", "Smith",
"jane.smith@example.com");

        itDept.addEmployee(emp1);
        itDept.addEmployee(emp2);

        // Update an employee
        emp1.setSalary(new BigDecimal("75000.00"));

        // Query employees
        TypedQuery<Employee> query = entityManager.createQuery(
            "SELECT e FROM Employee e WHERE e.department.name =
:deptName",
            Employee.class);
        query.setParameter("deptName", "IT");
        List<Employee> itEmployees = query.getResultList();

        // Remove an employee
        entityManager.remove(emp2);

        // Refresh state from database
        entityManager.refresh(emp1);

        // Detach an entity
        entityManager.detach(itDept);
    }

    @Transactional
    public Employee getEmployeeWithProjects(Long employeeId) {
        // Demonstrating fetch strategies
        Employee employee = entityManager.find(Employee.class,
employeeId);

        // Force initialization of lazy collection
        Hibernate.initialize(employee.getProjects());

        return employee;
    }
}
```

Key ORM Concepts Demonstrated:

Basic Entity Mapping: Mapping Java classes to database tables with @Entity, @Table, and field mappings.

Relationship Mapping:

One-to-Many (@OneToMany)

Many-to-One (@ManyToOne)

Many-to-Many (@ManyToMany)

One-to-One (@OneToOne)

Inheritance Strategies:

Single Table (all classes in one table with discriminator)

Joined (separate tables with joins)

Embeddable Objects: Using @Embeddable for reusable component mappings.

Entity Lifecycle:

Callback methods (@PrePersist, @PostLoad, etc.)

External entity listeners

Entity Manager Operations:

CRUD operations (persist, merge, remove, find)

Querying (JPQL)

Managing persistence context (detach, refresh)

Fetch Strategies: Controlling when related entities are loaded (eager vs lazy loading).

This implementation demonstrates how ORM bridges the gap between object-oriented programming and relational databases, allowing developers to work with objects while the ORM framework handles the database operations.