## Question 1 answer:

Object-Relational Mapping (ORM) is a technique that enables developers to interact with databases using object-oriented programming principles. It abstracts the complexities of SQL queries, providing a more intuitive way to handle database operations through objects. ORM helps in reducing boilerplate code, managing relationships between entities, and maintaining data integrity. It ensures better maintainability and portability by providing a consistent API regardless of the underlying database system.

## Question 2 answer:

Spring Data JPA is a powerful abstraction built on top of JPA and Hibernate. It simplifies the implementation of data access layers by reducing the need for boilerplate code. With features like method query derivation and integrated support for various databases (like H2 and MySQL), developers can quickly build robust applications. Spring Data JPA offers enhanced productivity through repository interfaces and custom query support. It supports both in-memory (H2) and external databases like MySQL, making development flexible and efficient.

## Question 3 answer:

Core objects in the Hibernate framework include SessionFactory, Session, TransactionFactory, Transaction, and ConnectionProvider. SessionFactory is a factory for creating Session objects and is a heavyweight object used across the application. Session is a lightweight object used to interact with the database and execute CRUD operations. TransactionFactory and Transaction help manage transactions, and ConnectionProvider abstracts the underlying JDBC connection management.

## Question 4 answer:

Hibernate allows ORM implementation through two configurations: XML and Annotation. In XML configuration, mapping is done using .hbm.xml files and settings in hibernate.cfg.xml. The persistence class is mapped to the database table manually using XML tags. In annotation-based configuration, classes use @Entity, @Table, @Id, and @Column annotations to define mappings. Both approaches require loading the configuration, building the SessionFactory, opening a session, beginning a transaction, performing the operation, and committing the transaction. Annotation configuration is more concise and preferred in modern applications.

## Question 5 answer:

Java Persistence API (JPA) is a specification for ORM in Java. It defines a standard API for accessing relational databases. Hibernate is an implementation of JPA and also offers extra features beyond the JPA specification. Spring Data JPA builds on top of JPA and Hibernate, providing a higher-level abstraction. It automates the implementation of repositories and simplifies CRUD operations. While JPA defines the 'what', Hibernate and Spring Data JPA define the 'how'. Spring Data JPA also integrates smoothly with the Spring ecosystem.

## Question 6 answer:

DML (Data Manipulation Language) operations in Spring Data JPA can be implemented using JpaRepository. To perform operations, you can use methods like findById(), save(), and deleteById(). Hibernate logs and ddl-auto configurations can be used to track database actions and schema generation. You can define custom query methods like findByName or findByCode directly in the repository interface. Spring Data JPA handles the implementation automatically, making it efficient and reducing boilerplate code.

## Exercise: Country Management Using Spring Data JPA

Create a service to manage country data using Spring Data JPA. This includes creating a repository interface for the Country entity, and implementing methods to:  
1. Find a country based on its country code using a method like findByCountryCode(String code).  
2. Add a new country using the save() method from JpaRepository.  
3. Implement service and controller layers to expose these operations through REST APIs.

**Solutions:**

**Country Entity:**

Import jakarta.persistence.Entity;

Import jakarta.persistence.Id;

Import jakarta.persistence.Table;

@Entity

@Table(name = “countries”)

Public class Country {

@Id

Private String countryCode;

Private String countryName;

// Getters and setters

Public String getCountryCode() {

Return countryCode;

}

Public void setCountryCode(String countryCode) {

This.countryCode = countryCode;

}

Public String getCountryName() {

Return countryName;

}

Public void setCountryName(String countryName) {

This.countryName = countryName;

}

}

**Repository:**

Import org.springframework.data.jpa.repository.JpaRepository;

Public interface CountryRepository extends JpaRepository<Country, String> {

Country findByCountryCode(String countryCode);

}

**Country service:**

Import org.springframework.beans.factory.annotation.Autowired;

Import org.springframework.stereotype.Service;

Import java.util.Optional;

@Service

Public class CountryService {

@Autowired

Private CountryRepository countryRepository;

Public Country findByCode(String code) {

Return countryRepository.findByCountryCode(code);

}

Public Country addCountry(Country country) {

Return countryRepository.save(country);

}

}

**Country controller:**

Import org.springframework.beans.factory.annotation.Autowired;

Import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping(“/countries”)

Public class CountryController {

@Autowired

Private CountryService countryService;

@GetMapping(“/{code}”)

Public Country getCountryByCode(@PathVariable String code) {

Return countryService.findByCode(code);

}

@PostMapping(“/add”)

Public Country addNewCountry(@RequestBody Country country) {

Return countryService.addCountry(country);

}

}

Now, we can add the country, or fetch country by country code

**POST**

{

“countryCode”: “IN”,

“countryName”: “India”

}

**GET**

{

“countryCode”: “IN”,

“countryName”: “India”

}