JPA	Hibernate
JPA is described in **javax.persistence** package.	Hibernate is described in **org.hibernate** package.
It describes the handling of relational data in Java applications.	Hibernate is an Object-Relational Mapping (ORM) tool that is used to save the Java objects in the relational database system.
It is not an implementation. It is only a Java specification.	Hibernate is an implementation of JPA. Hence, the common standard which **is** given by JPA is followed by Hibernate.
It is a standard API that permits to perform database operations.	It is used in mapping Java data types with SQL data types and database tables.
As an object-oriented query language, it uses **Java Persistence Query Language (JPQL)** to execute database operations.	As an object-oriented query language, it uses **Hibernate Query Language (HQL)** to execute database operations.
To interconnect with the entity manager factory for the persistence unit, it uses **EntityManagerFactory** interface. Thus, it gives an entity manager.	To create Session instances, it uses **SessionFactory** interface.
To make, read, and remove actions for instances of mapped entity classes, it uses **EntityManager** interface. This interface interconnect**s** with the persistence condition.	To make, read, and remove actions for instances of mapped entity classes, it uses **Session** interface. It acts as a runtime interface between a Java application and Hibernate.

JPA using Hibernate: Introduction

Definition

Java Persistence API (JPA) is a specification for accessing, persisting, and managing data between Java objects/classes and a relational database. Hibernate is an ORM (Object-Relational Mapping) framework that implements JPA, providing a way to map Java objects to database tables.

Necessity

- **Abstraction over JDBC**: JPA abstracts the boilerplate code required for database operations, making development faster and cleaner.
- **Object-Relational Mapping**: It bridges the gap between object-oriented programming and relational databases, allowing for the use of objects to represent data in a database.
- **Consistency**: JPA standardizes the way data persistence is handled, ensuring consistency across different projects and teams.

Features/Advantages

- **Simplified Database Access**: Using JPA and Hibernate, developers can perform complex database operations with simple Java methods.
- **Automatic Table Creation**: Hibernate can generate database tables based on entity mappings.
- **Lazy Loading**: Hibernate supports lazy loading, which fetches related data only when it's accessed, improving performance.
- **Caching**: Hibernate provides first-level and second-level caching to reduce database access.

Start Over Here

Student.java

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity // This marks the class as a JPA entity.
public class Student {
    @Id // This marks the field as a primary key.
    @GeneratedValue(strategy = GenerationType.IDENTITY) // This specifies that
the ID should be generated automatically.
    private Long id;
```

```
    private String name;

    // Getters and setters
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Main.java

```
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class Main {
    public static void main(String[] args) {
        EntityManagerFactory emf =
Persistence.createEntityManagerFactory("example-unit"); // Create
EntityManagerFactory
        EntityManager em = emf.createEntityManager(); // Create EntityManager

        em.getTransaction().begin(); // Begin transaction
        Student student = new Student();
        student.setName("John Doe");
        em.persist(student); // Persist entity
        em.getTransaction().commit(); // Commit transaction

        Student foundStudent = em.find(Student.class, student.getId()); //
Find entity
        System.out.println("Found student: " + foundStudent.getName());

        em.close(); // Close EntityManager
        emf.close(); // Close EntityManagerFactory
    }
}
```

Explanation of Code

1. **Student.java**: This class is a simple JPA entity with id and name fields. The annotations @Entity, @Id, and @GeneratedValue are used to define the entity and its primary key.
2. **build.gradle**: This file contains the necessary Gradle configuration. It includes dependencies for Spring Boot, Spring Data JPA, and H2 database.
3. **application.properties**: This file contains the configuration for the H2 database, which is an in-memory database used for testing purposes.
4. **Main.java**:
    - **EntityManagerFactory emf = Persistence.createEntityManagerFactory("example-unit");**: This line creates an EntityManagerFactory using the specified persistence unit.
    - **EntityManager em = emf.createEntityManager();**: This line creates an EntityManager from the factory.
    - **em.getTransaction().begin();**: This line begins a transaction.
    - **em.persist(student);**: This line persists the Student entity.
    - **em.getTransaction().commit();**: This line commits the transaction.
    - **Student foundStudent = em.find(Student.class, student.getId());**: This line retrieves the Student entity by its ID.
    - **em.close(); emf.close();**: These lines close the EntityManager and EntityManagerFactory, respectively.

Understanding Persistence Unit in JPA

A **persistence unit** is a logical grouping that defines the set of all entity classes that are managed by EntityManager instances in an application. It is defined in the persistence.xml file, which is part of the configuration required by JPA to specify database connection details, JPA provider settings, and other properties.

Student.java

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
```

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

    // Getters and setters
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

2. Managing Entities (Persisting, Updating, Deleting)

Main.java

```java
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class Main {
    public static void main(String[] args) {
        // Create EntityManagerFactory
        EntityManagerFactory emf =
Persistence.createEntityManagerFactory("example-unit");
        EntityManager em = emf.createEntityManager();

        // Persisting a new student
        em.getTransaction().begin();
        Student student = new Student();
        student.setName("John Doe");
        em.persist(student);
        em.getTransaction().commit();

        // Updating an existing student
        em.getTransaction().begin();
        Student foundStudent = em.find(Student.class, student.getId());
```

```
        foundStudent.setName("Jane Doe");
        em.getTransaction().commit();

        // Deleting a student
        em.getTransaction().begin();
        em.remove(foundStudent);
        em.getTransaction().commit();

        // Close EntityManager and EntityManagerFactory
        em.close();
        emf.close();
    }
}
```

3. Querying Entities

Main.java (Continued)

```
import javax.persistence.Query;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Create EntityManagerFactory
        EntityManagerFactory emf =
Persistence.createEntityManagerFactory("example-unit");
        EntityManager em = emf.createEntityManager();

        // Persisting some students for query examples
        em.getTransaction().begin();
        Student student1 = new Student();
        student1.setName("John Doe");
        em.persist(student1);

        Student student2 = new Student();
        student2.setName("Jane Doe");
        em.persist(student2);

        Student student3 = new Student();
        student3.setName("Alice Smith");
        em.persist(student3);

        em.getTransaction().commit();

        // Querying all students
        Query query = em.createQuery("SELECT s FROM Student s");
        List<Student> students = query.getResultList();
        for (Student student : students) {
            System.out.println("Student Name: " + student.getName());
```

```
        }

        // Querying students with a specific name
        query = em.createQuery("SELECT s FROM Student s WHERE s.name =
:name");
        query.setParameter("name", "Jane Doe");
        Student singleStudent = (Student) query.getSingleResult();
        System.out.println("Found student: " + singleStudent.getName());

        // Close EntityManager and EntityManagerFactory
        em.close();
        emf.close();
    }
}
```

Explanation

1. **Entity Definition (Student.java)**:
    - The @Entity annotation marks the class as a JPA entity.
    - The @Id annotation marks the primary key field.
    - The @GeneratedValue annotation specifies the generation strategy for the primary key.
2. **Managing Entities (Main.java)**:
    - **Persisting**: Creating and saving a new entity using em.persist().
    - **Updating**: Retrieving an entity using em.find(), modifying it, and committing the transaction.
    - **Deleting**: Removing an entity using em.remove().
3. **Querying Entities (Main.java)**:
    - Using JPQL (Java Persistence Query Language) to query all students and specific students by name.
    - em.createQuery() is used to create the query.
    - query.getResultList() retrieves multiple results.
    - query.getSingleResult() retrieves a single result.

Entity Relationships in Hibernate-Specific JPA

In JPA, entity relationships represent how entities relate to each other. There are four main types of relationships:

1. **One-to-One**

2. **One-to-Many**
3. **Many-to-One**
4. **Many-to-Many**

We will use a simple Student example to explain these relationships.

1. One-to-One Relationship

A one-to-one relationship is where one entity is associated with exactly one instance of another entity.

Example: Each student has one address.

Student.java

```
import javax.persistence.*;

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

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "address_id", referencedColumnName = "id")
    private Address address;

    // Getters and setters
    // ...
}
```

Address.java

```
import javax.persistence.*;

@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String street;
    private String city;

    // Getters and setters
    // ...
}
```

2. One-to-Many Relationship

A one-to-many relationship is where one entity is associated with multiple instances of another entity.

Example: One student can have multiple courses.

Student.java

```
import javax.persistence.*;
import java.util.List;

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

    @OneToMany(mappedBy = "student", cascade = CascadeType.ALL)
    private List<Course> courses;

    // Getters and setters
    // ...
}
```

Course.java

```
import javax.persistence.*;

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

    @ManyToOne
    @JoinColumn(name = "student_id", nullable = false)
    private Student student;

    // Getters and setters
    // ...
}
```

3. Many-to-One Relationship

A many-to-one relationship is where multiple instances of an entity are associated with one instance of another entity.

Example: Multiple courses belong to one student (as shown in the one-to-many example above).

4. Many-to-Many Relationship

A many-to-many relationship is where multiple instances of one entity are associated with multiple instances of another entity.

Example: Students can enroll in multiple courses, and courses can have multiple students.

Student.java

```
import javax.persistence.*;
import java.util.List;

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

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

    // Getters and setters
    // ...
}
```

Course.java

```
import javax.persistence.*;
import java.util.List;

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

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

```
    // Getters and setters
    // ...
}
```

Explanation

1. **One-to-One**:
    - @OneToOne annotation defines the one-to-one relationship.
    - @JoinColumn specifies the foreign key column in the owning entity.
2. **One-to-Many**:
    - @OneToMany annotation defines the one-to-many relationship.
    - mappedBy indicates the field that owns the relationship in the other entity.
3. **Many-to-One**:
    - @ManyToOne annotation defines the many-to-one relationship.
    - @JoinColumn specifies the foreign key column in the owning entity.
4. **Many-to-Many**:
    - @ManyToMany annotation defines the many-to-many relationship.
    - @JoinTable specifies the join table that holds the foreign keys of both entities.

Relationships

Necessity and Use of Different Types of Relationships in JPA

Entity relationships in JPA (and ORM in general) model real-world associations between objects in a relational database context. Understanding why and how to use these relationships is crucial for effective database design and application development.

1. One-to-One Relationship

Necessity: A one-to-one relationship is used when an entity is associated with exactly one instance of another entity. This can help in breaking down complex entities into simpler, more manageable parts.

Example Use Case: A Student entity that has an associated Address.

- **Why**: It keeps the student information separate from the address information, promoting modularity and reuse.
- **Database Schema**: There will be a foreign key in one table referring to the primary key of another table.

Example:

- **Student Table**: id, name, address_id
- **Address Table**: id, street, city

2. One-to-Many Relationship

Necessity: A one-to-many relationship is used when one entity is associated with multiple instances of another entity. This is common in hierarchical structures.

Example Use Case: A Student entity that has multiple Course entities.

- **Why**: It allows one student to be enrolled in multiple courses.
- **Database Schema**: The Course table will have a foreign key referring to the Student table.

Example:

- **Student Table**: id, name
- **Course Table**: id, course_name, student_id

3. Many-to-One Relationship

Necessity: A many-to-one relationship is essentially the reverse of a one-to-many relationship. It's used when multiple instances of an entity are associated with one instance of another entity.

Example Use Case: Multiple Course entities belong to one Student.

- **Why**: It allows us to track which student is taking which courses.
- **Database Schema**: Similar to the one-to-many relationship, the Course table has a foreign key to the Student table.

Example:

- **Student Table**: id, name
- **Course Table**: id, course_name, student_id

4. Many-to-Many Relationship

Necessity: A many-to-many relationship is used when multiple instances of one entity are associated with multiple instances of another entity. This is common in many real-world scenarios.

Example Use Case: Students enrolling in multiple courses, and each course having multiple students.

- **Why**: It models complex relationships where entities are interlinked.
- **Database Schema**: Requires a join table (also known as a junction table) to manage the associations.

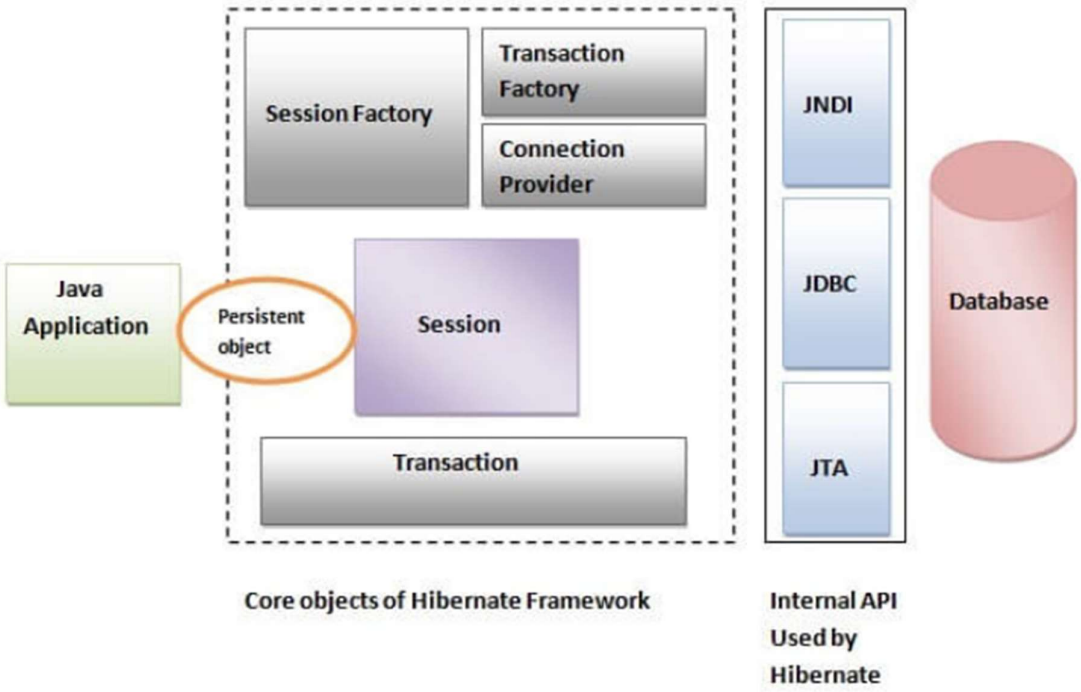
Example:

- **Student Table**: id, name
- **Course Table**: id, course_name
- **Student_Course Table**: student_id, course_id

How Relationships Work with Databases

1. **One-to-One**:
    - Uses a foreign key in the Student table that references the primary key in the Address table.
    - Ensures that each student can only have one address and vice versa.
2. **One-to-Many**:
    - Uses a foreign key in the Course table that references the primary key in the Student table.
    - Allows multiple courses to be linked to a single student.
3. **Many-to-One**:
    - Similar to one-to-many but from the perspective of the many-side entity.
    - Multiple courses refer to a single student, enforcing that each course is linked to a specific student.
4. **Many-to-Many**:
    - Uses a join table that holds foreign keys from both Student and Course tables.
    - Facilitates the linking of multiple students to multiple courses and vice versa.

Hibernate Architecture



Hibernate framework uses many objects such as session factory, session, transaction etc. alongwith existing Java API such as JDBC (Java Database Connectivity), JTA (Java Transaction API) and JNDI (Java Naming Directory Interface).

Elements of Hibernate Architecture

For creating the first hibernate application, we must know the elements of Hibernate architecture. They are as follows:

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.

Spring Core Introduction / Overview

Spring Core is the foundation of the Spring Framework, providing fundamental features for dependency injection and application context management. It is essential for creating scalable, maintainable, and modular Java applications. Here, we'll cover the key concepts, components, and features of Spring Core.

Definition

Spring Core: The core module of the Spring Framework that provides the fundamental features like Dependency Injection (DI) and Inversion of Control (IoC). It is responsible for managing the lifecycle and configuration of application objects.

Key Concepts

1. **Dependency Injection (DI)**: A design pattern used to implement IoC, where the control of object creation and dependency management is given to a container or framework. This promotes loose coupling and easier testing.
2. **Inversion of Control (IoC)**: A principle where the control flow of the application is inverted, and the control is given to a container or framework. This container manages the lifecycle and configuration of objects.

3. **Application Context**: The central interface in Spring for providing configuration information to the application. It is an extension of the BeanFactory interface, providing more enterprise-specific functionality.

Components and Features

1. **Beans**: Objects that are managed by the Spring IoC container. They are created, configured, and managed by the Spring container.
2. **BeanFactory**: The basic container that provides the configuration framework and basic functionality for managing beans.
3. **ApplicationContext**: A more advanced container that extends BeanFactory and adds more enterprise-specific features such as event propagation, declarative mechanisms to create a bean, and various means to look up.
4. **Configuration Metadata**: Metadata used by the Spring container to configure beans. This metadata can be provided in various forms, such as XML, annotations, or Java code.

Why Spring Core?

- **Loose Coupling**: By using DI and IoC, Spring promotes loose coupling between the components, making the application more modular and easier to maintain.
- **Testability**: Spring makes it easier to write unit tests by allowing you to inject mock dependencies.
- **Modularity**: Encourages modular development by promoting separation of concerns.
- **Integration**: Spring integrates well with other frameworks and technologies, making it suitable for enterprise application development.

Example with Explanation

Let's create a simple example to understand how Spring Core works. We will create a Student service that depends on a StudentRepository.

1. Maven Configuration (pom.xml)

Add the following dependencies in your pom.xml:

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.3.9</version>
  </dependency>
</dependencies>
```

2. Spring Configuration (Java-based)

We will use Java-based configuration to define our beans.

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {

    @Bean
    public StudentRepository studentRepository() {
        return new StudentRepository();
    }

    @Bean
    public StudentService studentService() {
        return new StudentService(studentRepository());
    }
}
```

3. Service and Repository Classes

```java
// Student.java
public class Student {
    private int id;
    private String name;

    // Constructors, getters, and setters
}

// StudentRepository.java
public class StudentRepository {
    public Student getStudentById(int id) {
        return new Student(id, "John Doe");
    }
}

// StudentService.java
public class StudentService {
    private final StudentRepository studentRepository;

    public StudentService(StudentRepository
studentRepository) {
```

```
        this.studentRepository = studentRepository;
    }

    public Student findStudent(int id) {
        return studentRepository.getStudentById(id);
    }
}
```

4. Main Class to Run the Application

```
import org.springframework.context.ApplicationContext;
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);
        StudentService studentService = context.getBean(StudentService.class);

        Student student = studentService.findStudent(1);
        System.out.println("Student Name: " + student.getName());
    }
}
```

Explanation

1. **Configuration**: AppConfig class is marked with @Configuration indicating that it is a source of bean definitions. The @Bean annotation tells Spring that the method will return an object that should be registered as a bean in the Spring application context.
2. **Dependency Injection**: The StudentService class is dependent on StudentRepository. In AppConfig, we define the beans and inject the dependencies. This allows the StudentService to use the StudentRepository without needing to instantiate it manually.
3. **Application Context**: In the Main class, we use AnnotationConfigApplicationContext to load the Spring context from the AppConfig class. The context is used to get the StudentService bean and call its method.

Spring Container

The Spring Container is a key component of the Spring Framework that is responsible for managing the lifecycle and configuration of application objects (beans). It is based on the concepts of Inversion of Control (IoC) and Dependency Injection (DI), which allow for a clean separation of configuration and application logic.

Key Concepts of Spring Container

1. Inversion of Control (IoC)

IoC is a design principle in which the control of object creation, configuration, and management is transferred from the application code to the container. In Spring, the IoC container takes responsibility for these tasks.

2. Dependency Injection (DI)

DI is a design pattern that allows dependencies to be injected into an object, rather than the object creating the dependencies itself. This promotes loose coupling and easier testing.

Types of Spring Containers

1. **BeanFactory**:
    - **Basic IoC container**.
    - Lazily initializes beans (beans are created only when requested).
    - Suitable for simple applications.
2. **ApplicationContext**:
    - **Advanced IoC container** with more enterprise-specific functionalities.
    - Eagerly initializes beans (beans are created at startup).
    - Provides additional features like event propagation, declarative mechanisms to create a bean, different scopes, and more.

Core Components

1. BeanFactory

- Basic container.
- Uses XmlBeanFactory, DefaultListableBeanFactory, and other implementations.
- Configures beans using XML.

```
BeanFactory factory = new XmlBeanFactory(new
ClassPathResource("beans.xml"));
MyBean myBean = (MyBean) factory.getBean("myBean");

```

2. ApplicationContext

- Provides all functionalities of BeanFactory along with enterprise-specific features.
- Uses ClassPathXmlApplicationContext, FileSystemXmlApplicationContext, AnnotationConfigApplicationContext, etc.

```
ApplicationContext context = new
ClassPathXmlApplicationContext("beans.xml");
MyBean myBean = context.getBean(MyBean.class);
```

•

Key Features

- **Bean Definition**: Describes the configuration of beans.
- **Bean Scopes**: Defines the lifecycle and visibility of beans (singleton, prototype, request, session, etc.).
- **Bean Lifecycle**: Methods like init-method and destroy-method manage bean initialization and destruction.
- **Dependency Injection**: Configures bean dependencies using constructor injection, setter injection, and field injection.
- **Event Handling**: Manages custom events and built-in events like ContextRefreshedEvent, ContextClosedEvent.
- **AOP Integration**: Supports Aspect-Oriented Programming.

Detailed Explanation with Examples

1. Bean Definition and Configuration

XML Configuration:

```
<!-- beans.xml -->
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="student" class="com.example.Student">
        <property name="id" value="1"/>
        <property name="name" value="John Doe"/>
    </bean>
</beans>
```

Java-based Configuration:

```
@Configuration
public class AppConfig {

    @Bean
    public Student student() {
        return new Student(1, "John Doe");
    }
}
```

2. Bean Scopes

- **Singleton**: Single instance per Spring container.
- **Prototype**: New instance each time requested.
- **Request**: One instance per HTTP request.
- **Session**: One instance per HTTP session.

Example:

```
<bean id="student" class="com.example.Student" scope="prototype">
    <property name="id" value="1"/>
    <property name="name" value="John Doe"/>
</bean>
```

3. Dependency Injection

Constructor Injection:

```
public class Student {
    private int id;
    private String name;

    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

Setter Injection:

```
public class Student {
    private int id;
    private String name;

    public void setId(int id) {
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Java Configuration:

```
@Configuration
public class AppConfig {
```

```
    @Bean
    public Student student() {
        return new Student(1, "John Doe");
    }

    @Bean
    public School school() {
        return new School(student());
    }
}
```

4. Bean Lifecycle

XML Configuration:

<bean id="student" class="com.example.Student" init-method="init" destroy-method="destroy">

<property name="id" value="1"/>

<property name="name" value="John Doe"/>

</bean>

Java Configuration:

```
@Configuration
public class AppConfig {

    @Bean(initMethod = "init", destroyMethod = "destroy")
    public Student student() {
        return new Student(1, "John Doe");
    }
}
```

Summary

- **Spring Container** is responsible for managing the lifecycle and configuration of application beans.
- Provides advanced features through ApplicationContext compared to the basic BeanFactory.
- Supports various configurations, dependency injection, scopes, lifecycle management, and event handling.
- Promotes loose coupling and easier testing, which are key for developing robust and scalable applications.

Dependency Injection in Spring

Dependency Injection (DI) is a design pattern used to implement IoC (Inversion of Control). It allows the creation of dependent objects outside of a class and provides those objects to a class through different methods. This pattern helps in promoting loose coupling and enhancing testability and maintainability of code.

In Spring, DI can be implemented in three ways:

1. **Constructor Injection**
2. **Setter Injection**
3. **Field Injection**

We'll explain each of these with examples using a Student and School scenario.

1. Constructor Injection

Constructor Injection involves injecting dependencies through a class constructor. This method ensures that a class is always instantiated with its dependencies.

Example:

Student Class:

```
public class Student {
    private int id;
    private String name;

    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    // Getters and setters (optional)
    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }
}
```

School.java

```
public class School {
    private Student student;

    public School(Student student) {
        this.student = student;
    }

    // Getters and setters (optional)
    public Student getStudent() {
        return student;
    }
```

```
}
```

Configuration with XML

Configuration with Java

```
@Configuration
public class AppConfig {

    @Bean
    public Student student() {
        return new Student(1, "John Doe");
    }

    @Bean
    public School school() {
        return new School(student());
    }
}
```

2. Setter Injection

Setter Injection involves injecting dependencies via public setter methods. This method provides more flexibility as dependencies can be set or changed after object creation.

Example:

Student Class:

```
public class Student {
    private int id;
    private String name;

    public void setId(int id) {
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
    }

    // Getters (optional)
    public int getId() {
        return id;
    }

    public String getName() {
        return name;
```

```
    }
}
```

School.java

```
public class School {
    private Student student;

    public void setStudent(Student student) {
        this.student = student;
    }

    // Getter (optional)
    public Student getStudent() {
        return student;
    }
}
```

Configuration with XML

Configuration with Java

```
@Configuration
public class AppConfig {

    @Bean
    public Student student() {
        Student student = new Student();
        student.setId(1);
        student.setName("John Doe");
        return student;
    }

    @Bean
    public School school() {
        School school = new School();
        school.setStudent(student());
        return school;
    }
}
```

Metadata / Configuration in Spring

In Spring, metadata and configuration are essential for defining the beans and their dependencies. Spring offers multiple ways to configure beans and dependencies:

1. **XML Configuration**
2. **Java-Based Configuration (using @Configuration and @Bean annotations)**
3. **Annotation-Based Configuration (using @Component, @Autowired, etc.)**

Each method has its use cases, and understanding them helps in choosing the right approach based on the application requirements.

1. XML Configuration

XML Configuration was the traditional way of configuring Spring applications. It involves defining beans and their dependencies in an XML file.

Example:

applicationContext.xml:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="student" class="com.example.Student">
    <property name="id" value="1"/>
    <property name="name" value="John Doe"/>
  </bean>

  <bean id="school" class="com.example.School">
    <property name="student" ref="student"/>
  </bean>

</beans>
```

Student.java

```
package com.example;
```

```
public class Student {
    private int id;
    private String name;

    // Getters and setters
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

School.java

```
package com.example;

public class School {
    private Student student;

    // Getter and setter
    public Student getStudent() {
        return student;
```

```
    }

    public void setStudent(Student student) {
        this.student = student;
    }
}
```

Main.java

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
        School school = context.getBean(School.class);
        System.out.println("Student ID: " + school.getStudent().getId());
        System.out.println("Student Name: " + school.getStudent().getName());
    }
}
```

2. Java-Based Configuration

Java-Based Configuration uses @Configuration and @Bean annotations to define beans and their dependencies. It provides a type-safe and refactor-friendly way to configure Spring applications.

Example:

AppConfig.java:

```
package com.example;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
@Configuration
public class AppConfig {

    @Bean
    public Student student() {
        return new Student(1, "John Doe");
    }

    @Bean
    public School school() {
        School school = new School();
        school.setStudent(student());
        return school;
    }
}
```

Student.java

```
package com.example;

public class Student {
    private int id;
    private String name;

    // Constructor
    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }

    // Getters
    public int getId() {
```

```
        return id;
    }

    public String getName() {
        return name;
    }
}
```

School.java

```
package com.example;

public class School {
    private Student student;

    // Getter and setter
    public Student getStudent() {
        return student;
    }

    public void setStudent(Student student) {
        this.student = student;
    }
}
```

Main.java

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
```

```
        School school = context.getBean(School.class);
        System.out.println("Student ID: " + school.getStudent().getId());
        System.out.println("Student Name: " + school.getStudent().getName());
    }
}
```

3. Annotation-Based Configuration

Annotation-Based Configuration uses annotations such as @Component, @Autowired, @Service, @Repository, etc., to define beans and their dependencies.

Example:

Student.java:

```
package com.example;

import org.springframework.stereotype.Component;

@Component
public class Student {
    private int id = 1;
    private String name = "John Doe";

    // Getters
    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }
}
```

School.java

```
package com.example;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class School {

   @Autowired
   private Student student;

   // Getter
   public Student getStudent() {
      return student;
   }
}
```

AppConfig.java

```
package com.example;

import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan(basePackages = "com.example")
public class AppConfig {
}
```

Main.java

```
package com.example;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
```

```
public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
        School school = context.getBean(School.class);
        System.out.println("Student ID: " + school.getStudent().getId());
        System.out.println("Student Name: " + school.getStudent().getName());
    }
}
```

Summary

- **XML Configuration**: Traditional way, involves defining beans in an XML file.
- **Java-Based Configuration**: Uses @Configuration and @Bean annotations, providing a type-safe way to configure beans.
- **Annotation-Based Configuration**: Uses annotations such as @Component, @Autowired, etc., for defining and injecting dependencies directly in the code.

1. Creational Design Patterns

Example: Student Management System

We'll demonstrate **Factory**, **Abstract Factory**, **Builder**, **Prototype**, and **Singleton** in one example.

```java
// Factory Pattern
public interface Student {
    void showDetails();
}

public class HighSchoolStudent implements Student {
    @Override
    public void showDetails() {
        System.out.println("High School Student");
    }
}

public class CollegeStudent implements Student {
    @Override
    public void showDetails() {
        System.out.println("College Student");
    }
}

public class StudentFactory {
```

```
    public Student getStudent(String studentType) {
        if (studentType == null) {
            return null;
        }
        if (studentType.equalsIgnoreCase("HighSchool")) {
            return new HighSchoolStudent();
        } else if (studentType.equalsIgnoreCase("College")) {
            return new CollegeStudent();
        }
        return null;
    }
}

// Abstract Factory Pattern
public interface Teacher {
    void showDetails();
}

public class HighSchoolTeacher implements Teacher {
    @Override
    public void showDetails() {
        System.out.println("High School Teacher");
    }
}

public class CollegeTeacher implements Teacher {
    @Override
    public void showDetails() {
        System.out.println("College Teacher");
    }
}

public interface AbstractFactory {
    Student getStudent(String studentType);
    Teacher getTeacher(String teacherType);
}

public class EducationFactory implements AbstractFactory {
    @Override
    public Student getStudent(String studentType) {
        return new StudentFactory().getStudent(studentType);
    }

    @Override
    public Teacher getTeacher(String teacherType) {
        if (teacherType == null) {
```

```
            return null;
        }
        if (teacherType.equalsIgnoreCase("HighSchool")) {
            return new HighSchoolTeacher();
        } else if (teacherType.equalsIgnoreCase("College")) {
            return new CollegeTeacher();
        }
        return null;
    }
}

// Builder Pattern
public class StudentBuilder {
    private String id;
    private String name;
    private String grade;

    public StudentBuilder setId(String id) {
        this.id = id;
        return this;
    }

    public StudentBuilder setName(String name) {
        this.name = name;
        return this;
    }

    public StudentBuilder setGrade(String grade) {
        this.grade = grade;
        return this;
    }

    public Student build() {
        return new Student(this);
    }
}

class Student {
    private String id;
    private String name;
    private String grade;

    public Student(StudentBuilder builder) {
        this.id = builder.id;
        this.name = builder.name;
        this.grade = builder.grade;
```

```
    }

    @Override
    public String toString() {
        return "Student [id=" + id + ", name=" + name + ", grade=" +
grade + "]";
    }
}

// Prototype Pattern
public class PrototypeStudent implements Cloneable {
    private String id;
    private String name;

    public PrototypeStudent(String id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return new PrototypeStudent(id, name);
    }

    @Override
    public String toString() {
        return "PrototypeStudent [id=" + id + ", name=" + name +
"]";
    }
}

// Singleton Pattern
public class StudentRegistry {
    private static StudentRegistry instance;

    private StudentRegistry() {}

    public static StudentRegistry getInstance() {
        if (instance == null) {
            instance = new StudentRegistry();
        }
        return instance;
    }

    public void registerStudent(String id, String name) {
```

```
        System.out.println("Registered Student: " + id + " - " +
name);
    }
}

// Main class to test
public class CreationalPatternDemo {
    public static void main(String[] args) throws
CloneNotSupportedException {
        // Factory Pattern
        StudentFactory studentFactory = new StudentFactory();
        Student student1 = studentFactory.getStudent("HighSchool");
        student1.showDetails();

        // Abstract Factory Pattern
        AbstractFactory factory = new EducationFactory();
        Student student2 = factory.getStudent("College");
        Teacher teacher1 = factory.getTeacher("HighSchool");
        student2.showDetails();
        teacher1.showDetails();

        // Builder Pattern
        Student student3 = new StudentBuilder()
                .setId("3")
                .setName("Alice")
                .setGrade("A")
                .build();
        System.out.println(student3);

        // Prototype Pattern
        PrototypeStudent prototypeStudent1 = new
PrototypeStudent("4", "Bob");
        PrototypeStudent prototypeStudent2 = (PrototypeStudent)
prototypeStudent1.clone();
        System.out.println(prototypeStudent1);
        System.out.println(prototypeStudent2);

        // Singleton Pattern
        StudentRegistry registry = StudentRegistry.getInstance();
        registry.registerStudent("5", "Charlie");
    }
}
```

2. Structural Design Patterns

Example: Student Management System

We'll demonstrate **Adapter**, **Bridge**, **Composite**, **Decorator**, **Facade**, **Flyweight**, and **Proxy** in one example.

```
// Adapter Pattern
public interface StudentDetails {
    void showDetails(String studentId);
}

public class LegacyStudentSystem {
    public void getStudentInfo(String studentId) {
        System.out.println("Fetching student details for ID: " +
studentId);
    }
}

public class StudentDetailsAdapter implements StudentDetails {
    private LegacyStudentSystem legacyStudentSystem;

    public StudentDetailsAdapter(LegacyStudentSystem
legacyStudentSystem) {
        this.legacyStudentSystem = legacyStudentSystem;
    }

    @Override
    public void showDetails(String studentId) {
        legacyStudentSystem.getStudentInfo(studentId);
    }
}

// Bridge Pattern
public interface StudentReport {
    void generateReport();
}

public class DetailedReport implements StudentReport {
    @Override
    public void generateReport() {
        System.out.println("Generating detailed student report...");
    }
}
```

```java
public class SummaryReport implements StudentReport {
    @Override
    public void generateReport() {
        System.out.println("Generating summary student report...");
    }
}

public abstract class Report {
    protected StudentReport studentReport;

    protected Report(StudentReport studentReport) {
        this.studentReport = studentReport;
    }

    public abstract void showReport();
}

public class StudentReportBridge extends Report {
    protected StudentReportBridge(StudentReport studentReport) {
        super(studentReport);
    }

    @Override
    public void showReport() {
        studentReport.generateReport();
    }
}

// Composite Pattern
import java.util.ArrayList;
import java.util.List;

public interface StudentComponent {
    void showDetails();
}

public class StudentLeaf implements StudentComponent {
    private String name;

    public StudentLeaf(String name) {
        this.name = name;
```

```
    }

    @Override
    public void showDetails() {
        System.out.println(name);
    }
}

public class StudentComposite implements StudentComponent {
    private List<StudentComponent> students = new ArrayList<>();

    @Override
    public void showDetails() {
        for (StudentComponent student : students) {
            student.showDetails();
        }
    }

    public void addStudent(StudentComponent student) {
        students.add(student);
    }

    public void removeStudent(StudentComponent student) {
        students.remove(student);
    }
}

// Decorator Pattern
public interface Student {
    String getDescription();
}

public class BasicStudent implements Student {
    @Override
    public String getDescription() {
        return "Student";
    }
}

public abstract class StudentDecorator implements Student {
    protected Student decoratedStudent;
```

```
    public StudentDecorator(Student decoratedStudent) {
        this.decoratedStudent = decoratedStudent;
    }

    public String getDescription() {
        return decoratedStudent.getDescription();
    }
}

public class HonorsStudent extends StudentDecorator {
    public HonorsStudent(Student decoratedStudent) {
        super(decoratedStudent);
    }

    @Override
    public String getDescription() {
        return super.getDescription() + ", Honors";
    }
}

// Facade Pattern
public class StudentFacade {
    private LegacyStudentSystem legacyStudentSystem;
    private StudentReport studentReport;

    public StudentFacade() {
        legacyStudentSystem = new LegacyStudentSystem();
        studentReport = new DetailedReport();
    }

    public void showStudentDetails(String studentId) {
        legacyStudentSystem.getStudentInfo(studentId);
        studentReport.generateReport();
    }
}

// Flyweight Pattern
import java.util.HashMap;
import java.util.Map;

public interface StudentFlyweight {
    void showDetails(String grade);
```

```
}

public class ConcreteStudent implements StudentFlyweight {
    private String name;

    public ConcreteStudent(String name) {
        this.name = name;
    }

    @Override
    public void showDetails(String grade) {
        System.out.println("Student: " + name + ", Grade: " +
grade);
    }
}

public class StudentFlyweightFactory {
    private static final Map<String, StudentFlyweight> studentMap =
new HashMap<>();

    public static StudentFlyweight getStudent(String name) {
        StudentFlyweight student = studentMap.get(name);

        if (student == null) {
            student = new ConcreteStudent(name);
            studentMap.put(name, student);
        }
        return student;
    }
}

// Proxy Pattern
public interface StudentProxy {
    void showDetails();
}

public class RealStudent implements StudentProxy {
    private String name;

    public RealStudent(String name) {
        this.name = name;
    }
```

```
    @Override
    public void showDetails() {
        System.out.println("Student: " + name);
    }
}

public class StudentProxyImpl implements StudentProxy {
    private RealStudent realStudent;
    private String name;

    public StudentProxyImpl(String name) {
        this.name = name;
    }

    @Override
    public void showDetails() {
        if (realStudent == null) {
            realStudent = new RealStudent(name);
        }
        realStudent.showDetails();
    }
}

// Main class to test
public class StructuralPatternDemo {
    public static void main(String[] args) {
        // Adapter Pattern
        StudentDetails adapter = new StudentDetailsAdapter(new
LegacyStudentSystem());
        adapter.showDetails("1");

        // Bridge Pattern
        Report detailedReport = new StudentReportBridge(new
DetailedReport());
        detailedReport.showReport();

        // Composite Pattern
        StudentComponent student1 = new StudentLeaf("John");
        StudentComponent student2 = new StudentLeaf("Alice");
        StudentComposite group = new StudentComposite();
        group.addStudent(student1);
```

```
        group.addStudent(student2);
        group.showDetails();

        // Decorator Pattern
        Student student = new BasicStudent();
        System.out.println(student.getDescription());
        student = new HonorsStudent(student);
        System.out.println(student.getDescription());

        // Facade Pattern
        StudentFacade facade = new StudentFacade();
        facade.showStudentDetails("2");

        // Flyweight Pattern
        StudentFlyweight flyweightStudent1 =
StudentFlyweightFactory.getStudent("Bob");
        StudentFlyweight flyweightStudent2 =
StudentFlyweightFactory.getStudent("Bob");
        flyweightStudent1.showDetails("A");
        flyweightStudent2.showDetails("B");

        // Proxy Pattern
        StudentProxy proxy = new StudentProxyImpl("Charlie");
        proxy.showDetails();
    }
}
```

3. Behavioral Design Patterns

Example: Student Management System

We'll demonstrate **Chain of Responsibility**, **Command**, **Interpreter**, **Mediator**, **Memento**, **Observer**, **State**, **Strategy**, **Template**, and **Visitor** in one example.

```
// Chain of Responsibility Pattern
public abstract class StudentHandler {
    protected StudentHandler nextHandler;

    public void setNextHandler(StudentHandler nextHandler) {
        this.nextHandler = nextHandler;
    }

    public void handleRequest(String request) {
```

```
        if (nextHandler != null) {
            nextHandler.handleRequest(request);
        }
    }
}

public class HighSchoolHandler extends StudentHandler {
    @Override
    public void handleRequest(String request) {
        if (request.equalsIgnoreCase("HighSchool")) {
            System.out.println("Handling HighSchool request.");
        } else {
            super.handleRequest(request);
        }
    }
}

public class CollegeHandler extends StudentHandler {
    @Override
    public void handleRequest(String request) {
        if (request.equalsIgnoreCase("College")) {
            System.out.println("Handling College request.");
        } else {
            super.handleRequest(request);
        }
    }
}

// Command Pattern
public interface Command {
    void execute();
}

public class AddStudentCommand implements Command {
    private StudentReceiver receiver;

    public AddStudentCommand(StudentReceiver receiver) {
        this.receiver = receiver;
    }

    @Override
    public void execute() {
        receiver.addStudent();
    }
}
```

```java
public class StudentReceiver {
    public void addStudent() {
        System.out.println("Student added.");
    }
}

public class CommandInvoker {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void executeCommand() {
        command.execute();
    }
}

// Interpreter Pattern
import java.util.HashMap;
import java.util.Map;

public interface Expression {
    int interpret(Map<String, Integer> context);
}

public class NumberExpression implements Expression {
    private int number;

    public NumberExpression(int number) {
        this.number = number;
    }

    @Override
    public int interpret(Map<String, Integer> context) {
        return number;
    }
}

public class VariableExpression implements Expression {
    private String variableName;

    public VariableExpression(String variableName) {
        this.variableName = variableName;
    }
```

```
    @Override
    public int interpret(Map<String, Integer> context) {
        return context.get(variableName);
    }
}

// Mediator Pattern
public class StudentMediator {
    private List<StudentColleague> colleagues;

    public StudentMediator() {
        colleagues = new ArrayList<>();
    }

    public void addColleague(StudentColleague colleague) {
        colleagues.add(colleague);
    }

    public void mediate(String message, StudentColleague originator) {
        for (StudentColleague colleague : colleagues) {
            if (colleague != originator) {
                colleague.receive(message);
            }
        }
    }
}

public abstract class StudentColleague {
    protected StudentMediator mediator;

    public StudentColleague(StudentMediator mediator) {
        this.mediator = mediator;
    }

    public abstract void receive(String message);
}

public class ConcreteStudentColleague extends StudentColleague {
    public ConcreteStudentColleague(StudentMediator mediator) {
        super(mediator);
    }

    @Override
    public void receive(String message) {
        System.out.println("Received message: " + message);
    }
```

```
}

// Memento Pattern
public class StudentMemento {
    private String state;

    public StudentMemento(String state) {
        this.state = state;
    }

    public String getState() {
        return state;
    }
}

public class StudentOriginator {
    private String state;

    public void setState(String state) {
        this.state = state;
    }

    public StudentMemento saveStateToMemento() {
        return new StudentMemento(state);
    }

    public void getStateFromMemento(StudentMemento memento) {
        state = memento.getState();
    }
}

// Observer Pattern
import java.util.ArrayList;
import java.util.List;

public interface Observer {
    void update();
}

public class StudentObserver implements Observer {
    private String name;

    public StudentObserver(String name) {
        this.name = name;
    }
```

```
    @Override
    public void update() {
        System.out.println(name + " updated.");
    }
}

public class Subject {
    private List<Observer> observers = new ArrayList<>();

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update();
        }
    }
}

// State Pattern
public interface State {
    void doAction(Context context);
}

public class StartState implements State {
    @Override
    public void doAction(Context context) {
        System.out.println("Student is in start state.");
        context.setState(this);
    }
}

public class StopState implements State {
    @Override
    public void doAction(Context context) {
        System.out.println("Student is in stop state.");
        context.setState(this);
    }
}

public class Context {
    private State state;

    public void setState(State state) {
        this.state = state;
```

```
    }

    public State getState() {
        return state;
    }
}

// Strategy Pattern
public interface StudyStrategy {
    void study();
}

public class IntenseStudyStrategy implements StudyStrategy {
    @Override
    public void study() {
        System.out.println("Studying intensely.");
    }
}

public class RelaxedStudyStrategy implements StudyStrategy {
    @Override
    public void study() {
        System.out.println("Studying in a relaxed manner.");
    }
}

public class StudentContext {
    private StudyStrategy strategy;

    public void setStrategy(StudyStrategy strategy) {
        this.strategy = strategy;
    }

    public void study() {
        strategy.study();
    }
}

// Template Pattern
public abstract class StudentTemplate {
    public final void dailyRoutine() {
        wakeUp();
        study();
        goSleep();
    }
```

```
    protected abstract void study();

    private void wakeUp() {
        System.out.println("Waking up.");
    }

    private void goSleep() {
        System.out.println("Going to sleep.");
    }
}

public class HighSchoolStudentTemplate extends StudentTemplate {
    @Override
    protected void study() {
        System.out.println("Studying high school subjects.");
    }
}

public class CollegeStudentTemplate extends StudentTemplate {
    @Override
    protected void study() {
        System.out.println("Studying college subjects.");
    }
}

// Visitor Pattern
public interface StudentElement {
    void accept(Visitor visitor);
}

public class StudentElementConcrete implements StudentElement {
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

public interface Visitor {
    void visit(StudentElementConcrete element);
}

public class ConcreteVisitor implements Visitor {
    @Override
    public void visit(StudentElementConcrete element) {
        System.out.println("Visiting student element.");
    }
```

```
}

// Main class to test
public class BehavioralPatternDemo {
    public static void main(String[] args) {
        // Chain of Responsibility Pattern
        StudentHandler highSchoolHandler = new HighSchoolHandler();
        StudentHandler collegeHandler = new CollegeHandler();
        highSchoolHandler.setNextHandler(collegeHandler);

        highSchoolHandler.handleRequest("College");

        // Command Pattern
        StudentReceiver receiver = new StudentReceiver();
        Command command = new AddStudentCommand(receiver);
        CommandInvoker invoker = new CommandInvoker();
        invoker.setCommand(command);
        invoker.executeCommand();

        // Interpreter Pattern
        Map<String, Integer> context = new HashMap<>();
        context.put("a", 1);
        context.put("b", 2);

        Expression a = new VariableExpression("a");
        Expression b = new VariableExpression("b");
        Expression plus = new NumberExpression(a.interpret(context) +
b.interpret(context));
        System.out.println(plus.interpret(context));

        // Mediator Pattern
        StudentMediator mediator = new StudentMediator();
        StudentColleague student1 = new
ConcreteStudentColleague(mediator);
        StudentColleague student2 = new
ConcreteStudentColleague(mediator);
        mediator.addColleague(student1);
        mediator.addColleague(student2);
        student1.receive("Hello from student1");

        // Memento Pattern
        StudentOriginator originator = new StudentOriginator();
        StudentMemento memento = new StudentMemento("State1");
        originator.setState("State2");
        System.out.println(originator.getStateFromMemento(memento).getS
tate());
```

```
        // Observer Pattern
        Subject subject = new Subject();
        Observer observer1 = new StudentObserver("John");
        Observer observer2 = new StudentObserver("Alice");
        subject.addObserver(observer1);
        subject.addObserver(observer2);
        subject.notifyObservers();

        // State Pattern
        Context contextState = new Context();
        State startState = new StartState();
        State stopState = new StopState();

        startState.doAction(contextState);
        System.out.println(contextState.getState().toString());
        stopState.doAction(contextState);
        System.out.println(contextState.getState().toString());

        // Strategy Pattern
        StudentContext contextStrategy = new StudentContext();
        StudyStrategy intenseStudy = new IntenseStudyStrategy();
        StudyStrategy relaxedStudy = new RelaxedStudyStrategy();

        contextStrategy.setStrategy(intenseStudy);
        contextStrategy.study();
        contextStrategy.setStrategy(relaxedStudy);
        contextStrategy.study();

        // Template Pattern
        StudentTemplate highSchool = new HighSchoolStudentTemplate();
        highSchool.dailyRoutine();

        StudentTemplate college = new CollegeStudentTemplate();
        college.dailyRoutine();

        // Visitor Pattern
        StudentElement element = new StudentElementConcrete();
        Visitor visitor = new ConcreteVisitor();
        element.accept(visitor);
    }
}
```