Chap 01. 디자인 패턴

> GoF 왈, "특정한 상황에서 일반적 설계문제를 해결하기 위해 상호교류하는 수정 가능한 객체와 클래스들에 대한 설명이다.”
>
> 소프트웨어 설계에서 자주 발생하는 문제를 해결하기 위한 재사용 가능한 설계 템플릿
>
> 상호 작용하는 객체와 클래스의 구조와 관계를 정의하여 문제 해결

01. 싱글톤 패턴

Abstract

개념

- 하나의 클래스에 오직 하나의 인스턴스만 가지는 패턴
- 하나의 클래스를 기반으로 단 하나의 인스턴스를 만들어 이를 기반으로 로직을 만드는데 사용
- 데이터베이스 연결 모듈에 많이 사용

장점

- 1개의 인스턴스 만들어 놓고 다른 모듈들이 공유 → 인스턴스 생성 시 비용 감소

단점

- 의존성 높아짐

JAVA에서의 실글톤 패턴

```
class Singleton {
    private static class singleInstanceHolder {
        private static final Singleton INSTANCE = new Singleton();
    }
    public static Singleton getInstance() {
        return singleInstanceHolder.INSTANCE;
    }
}
```

```
public class HelloWorld{
     public static void main(String []args){
        Singleton a = Singleton.getInstance();
        Singleton b = Singleton.getInstance();
        System.out.println(a.hashCode());
        System.out.println(b.hashCode());
        if (a == b){
         System.out.println(true);
        }
     }
}
```

싱글톤 패턴 구현 7가지 방법

방법1) 단순한 메서드 호출

- 싱글톤 패턴 생성 여부 확인 → 싱글톤 없으면 새로 만듬/ 싱글톤 있으면 만들어진 인스턴스 반환

```
/*
기본 싱글톤 예제 코드
인스턴스 없을 경우 만들고 있으면 존재하는 인스턴스 반환

*/

public class Singleton{
    private static Siongleton instance:

    private Singleton() {

    }

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

```
/*
멀티 스레드 만족x 예제
*/


public class Hyo{
    private static String hyo="효선";
```

```
    public static void main(String[] args){
        Hyo a = new Hyo();

        //스레드 01
        new Thread(() -> {
            for (int i=0; i<10; i++){
                a.say("자바자바");
            }
        }).start();

        //스레드 02
        new Thread(() -> {
            for (int i=0; i<10; i++){
                a.say("파이썬파이썬");
            }
        }).start();
    }

    public void say(String song){
        hyo=song;
        try{
            long sleep=(long) (Math.random()*100);
            Thread.sleep(sleep);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        if (!hyo.equals(song)) {
            System.out.println(song+"|"+hyo);
        }
    }
}


//output
자바자바|파이썬파이썬
파이썬파이썬|자바자바
자바자바|파이썬파이썬
파이썬파이썬|자바자바
자바자바|파이썬파이썬
파이썬파이썬|자바자바
자바자바|파이썬파이썬
파이썬파이썬|자바자바
자바자바|파이썬파이썬
파이썬파이썬|자바자바
자바자바|파이썬파이썬
파이썬파이썬|자바자바
자바자바|파이썬파이썬
```

문제점

1. 원장성 결여
2. 멀티 스레드 불가 (JAVA는 멀티스레드 언어)

   멀티 스레드일 경우 인스턴스 2개 이상 만들어짐 → 싱글톤x

방법2) synchronized

- synchronized 키워드 : 인스턴스를 반환하기 전까지 격리 공간에 놓기 위해 잠금
- 최초로 접근한 스레드가 해당 메서드 호출시 다른 스레드가 접근하지 못하도록 잠금(lock)
- 방법1) 문제점 해결

```
/*
기본 싱글톤 예제 코드
인스턴스 없을 경우 만들고 있으면 존재하는 인스턴스 반환
synchronized 키워드 추가
getInstance 호출할때마다 멀티스레드에서 사용 가능

*/

public class Singleton02_01{
    private static Singleton02_01 instance;

    private Singleton(){

    }

    public static synchronized Singleton02_01 getInstance(){
        if (instance == null){
            instance=new Singleton02_01();
        }
        return instance;
    }
}
```

```
/*
synchronized 키워드 추가
getInstance 호출할때마다 멀티스레드에서 사용 가능

*/

public class Hyo{
    private static String hyo="효선";

    public static void main(String[] args){
        Hyo a = new Hyo();

        //스레드 01
        new Thread(() -> {
            for (int i=0; i<10; i++){
```

```
                a.say("자바자바");
            }
        }).start();

        //스레드 02
        new Thread(() -> {
            for (int i=0; i<10; i++){
                a.say("파이썬파이썬");
            }
        }).start();
    }

    public synchronized void say(String song){
        hyo=song;
        try{
            long sleep=(long) (Math.random()*100);
            Thread.sleep(sleep);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        if (!hyo.equals(song)) {
            System.out.println(song+"|"+hyo);
        }
    }
}
```

문제점

1. 여러개의 모듈들이 getinstance 메서드를 통해서 호출하는데 그때마다 매번 호출

   → lock이 걸려져 있는 메서드를 계속 호출하므로 성능저하 발생

방법3) 정적 멤버 / 방법4) 정적 블록

- 최초에 클래스 로딩때 미리 인스턴스 생성하여 이용
- 클래스 로딩과 동시에 싱글톤 인스턴스 만듬
- 모듈들이 싱글톤 인스턴스 요청시 그냥 만들어진 인스턴스 반환

```
/*
정적 멤버 기본 예제
*/

//정적 멤버
public class Singleton03_01 {
      //최초 선언
    private final static Singleton03_01 instance = new Singleton03_01();

    private Singleton03_01() {
```

```
    }

    public static Singleton03_01 getInstance() {
        return instance;
    }
}

//정적 블록
public class Singleton03_01 {
        //최초 선언
    private final Singleton03_01 instance=null;

    static{
        instance=new Singleton03_01();
    }

    private Singleton03_01() {

    }

    public static Singleton03_01 getInstance() {
        return instance();
    }
}
```

문제점

1. 자원낭비

방법5) 정적 멤버와 Lazy Holder(중첩 클래스)

- singleInstanceHolder라는 내부 클래스를 하너 더 만듬

  → Singleton클래스가 최초에 로딩되더라고 함께 초기화 되지 않고 getInstance()가 호출될때 singleInstanceHolder 클래스가 로딩되어 인스턴스 생성

- 모듈들이 필요로 할때만 정적 멤버로 선언

```
class Singleton05_01 {
    private static class singleInstanceholder {
        private static final Singleton05_01 INSTANCE = new Singleton05_01();

    }

    public static Singleton05_01 getInstance() {
        return singleInstanceholder.INSTANCE;
    }
}
```

방법6) 이중 확인 (DCL)

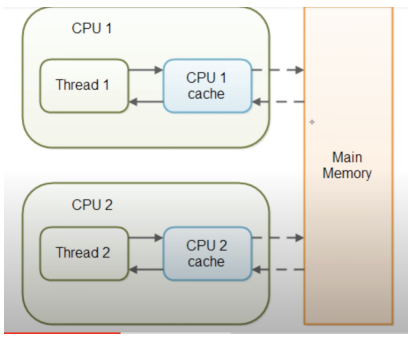
- 이중 확인 잠금(Double checked locking)
- 인스턴스 생성 여부를 싱글톤 패턴 잠금 전에 한번, 객체를 생성하기 전에 하번 2번 체크

  → 인스턴스가 존재하지 않을 때만 잠금 걸수 있음

```
/*
이중 확인
인스턴스가 null일 경우 synchronized로 인스턴스 없을 경우 할당
인스턴스 없는 지 확인 2번
*/


public class Singleton06_01 {

    private volatile Singleton06_01 instance;

    private Singleton06_01() {

    }

    public Singleton06_01 getInstance() {
        if (instance == null) {
            synchronized (Singleton06_01.class) {
                if (instance == null) {
                    instance = new Singleton06_01();
                }
            }
        }
        return instance;
    }
}
```

문제점

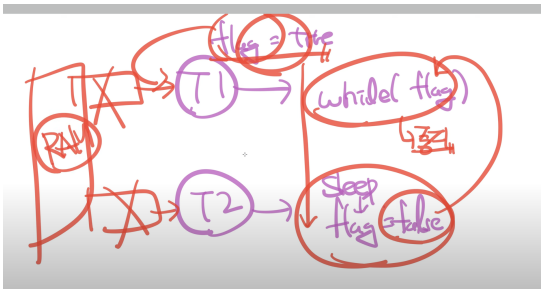
- java에서는 스레드 2개 열리면 변수를 메모리(RAM)으로부터 가져오지 않고 캐시메모리에서 각각의 캐시메모리를 기반으로 가져옴



*volatile 사용

- volatile 사용시 캐시메모리가 아닌 메인메모리(RAM)에서 변수 가지고옴

  → 스레드들이 같은 변수 이용 가능



방법7) enum

- enum의 인스턴스는 기본적으로 스레드 세이프(thread safe) 한 점이 보장됨

```
public enum SingletonEnum{
    INSTANCE;
    public void oportCloud(){

    }
}
```

방법5) 정적 멤버와 Lazy Holder(중첩 클래스)

방법7) enum

방법5가 가장 많이 사용되고 7번은 이펙티브 자바를 쓴 조슈아 브로크가 추천

mongoose의 싱글톤 패턴

```
/*
방법5 사용
mongoDB 연결 코드
*/


import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoDatabase;

public class MongoDBConnection {
```

```
    // 내부 클래스(singleInstanceHolder)로 싱글톤 인스턴스 지연 초기화
    private static class singleInstanceHolder {
        private static final MongoDBConnection INSTANCE = new MongoDBConnection();
    }

    private MongoClient mongoClient;
    private MongoDatabase database;

    // private 생성자: 외부에서 인스턴스화 할 수 없음
    private MongoDBConnection() {
        // MongoDB 연결 설정
        connect();
    }

    // getInstance() 메서드를 통해 싱글톤 인스턴스 반환
    public static MongoDBConnection getInstance() {
        return singleInstanceHolder.INSTANCE;
    }

    // MongoDB에 연결하는 메서드
    private void connect() {
        String connectionString = "mongodb://localhost:27017";
        mongoClient = MongoClients.create(connectionString);
        database = mongoClient.getDatabase("mydatabase");
        System.out.println("MongoDB connected");
    }

    // 데이터베이스 객체 반환 메서드
    public MongoDatabase getDatabase() {
        return database;
    }

    // 연결을 종료 메서드
    public void close() {
        if (mongoClient != null) {
            mongoClient.close();
        }
    }

    public static void main(String[] args) {
        // 예제 사용법
        MongoDBConnection connection = MongoDBConnection.getInstance();
        MongoDatabase db = connection.getDatabase();

        // 여기에서 데이터베이스 작업 수행

        // 작업이 끝나면 연결 종료
        connection.close();
    }
}
```

mysql의 싱글톤 패턴

```
/*
방법5 사용
mysql 연결 코드
*/

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class MySQLConnection {
    // 내부 클래스 (Lazy Holder)를 사용하여 싱글톤 인스턴스를 지연 초기화합니다.
    private static class ConnectionHolder {
        private static final MySQLConnection INSTANCE = new MySQLConnection();
    }

    private Connection connection;

    // private 생성자: 외부에서 인스턴스화 할 수 없습니다.
    private MySQLConnection() {
        // MySQL 연결 설정
        connect();
    }

    // getInstance() 메서드를 통해 싱글톤 인스턴스를 반환합니다.
    public static MySQLConnection getInstance() {
        return ConnectionHolder.INSTANCE;
    }

    // MySQL에 연결하는 메서드
    private void connect() {
        try {
            String url = "jdbc:mysql://localhost:3306/mydatabase"; // MySQL URL
            String user = "username"; // MySQL 사용자 이름
            String password = "password"; // MySQL 비밀번호
            connection = DriverManager.getConnection(url, user, password);
            System.out.println("MySQL connected");
        } catch (SQLException e) {
            e.printStackTrace();
            throw new RuntimeException("Failed to connect to MySQL", e);
        }
    }

    // 데이터베이스 연결 객체를 반환하는 메서드
    public Connection getConnection() {
        return connection;
    }
```

```
    // 연결을 종료하는 메서드
    public void close() {
        if (connection != null) {
            try {
                connection.close();
                System.out.println("MySQL connection closed");
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        // 예제 사용법
        MySQLConnection mySQLConnection = MySQLConnection.getInstance();
        Connection connection = mySQLConnection.getConnection();

        // 여기에서 데이터베이스 작업 수행

        // 작업이 끝나면 연결을 종료합니다.
        mySQLConnection.close();
    }
}
```

싱글톤 패턴의 단점

1. 테스트 독립성의 문제
   - 단위 테스트는 서로 독립적으로 실행
   - 싱글톤은 인스턴스 1개 → 테스트간 상태 공유 불가

     → 한 테스트가 다른 테스트에 영향
2. 테스트 격리의 어려움
   - 단위 테스트를 격리된 상태로 유지하기 위해서는 각 테스트마다 새로운 인스턴스 생성해야함
   - 싱글톤은 하나의 인스턴스 공유 → 격리 어려움
3. 모킹의 어려움
   - 싱글톤 객체를 쉽게 모킹(Mock)할 수 없음

     → **의존성 주입 필요**

*모킹

- 테스트 자동화에서 사용되는 기법
- 실제 객체 대신에 가짜 객체(Mock)를 만들어서 테스트 수행
- 이 가짜 객체는 실제 객체와 동일한 인터페이스를 가지며, 실제 객체의 동작을 흉내

  → 이를 통해 외부 의존성을 제거하고, 테스트 대상 코드의 특정 부분을 독립적으로 테스트 할 수 있음

의존성 주입

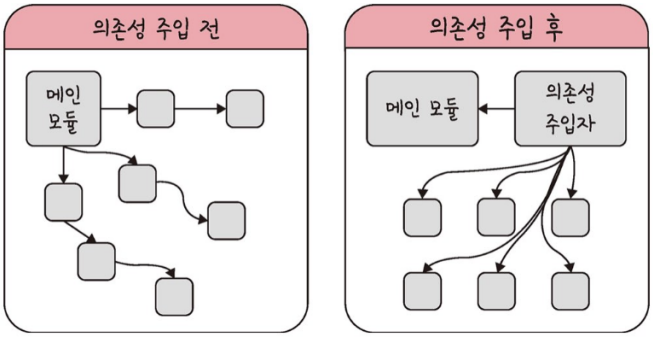
- **의존성(Dependency)**: 객체가 사용하는 다른 객체를 의미

  ex) 서비스 객체가 데이터베이스 객체를 필요로 한다면 데이터베이스 객체는 서비스 객체의 의존성

- **주입(Injection)**: 의존성을 외부에서 제공하는 행위 의미

  의존성 주입은 일반적으로 생성자 주입(Constructor Injection), 세터 주입(Setter Injection), 인터페이스 주입(Interface Injection)의 세 가지 방식

- 모듈간의 결합을 느슨하게 해줌
- A가 B에 의존성이 있다 = B의 변경 사항에 대해 A 또한 변해야 한다.



- 메인 모듈이 직접 하위 모듈에 의존성 주지 않고 중간에 의존성 주입자를 넣어 간접적으로 의존성 주입

spring framework에서의 의존성 주입

- 의존성 주입을 매우 쉽게 사용할 수 있게 해주는 강력한 프레임워크
- `@Autowired` 애노테이션을 사용하여 `UserService`의 생성자에 `EmailService`를 주입

스프링 설정 파일 (XML 또는 Java Config)

```
@Configuration
public class AppConfig {
    @Bean
    public EmailService emailService() {
        return new EmailService();
    }

    @Bean
    public UserService userService() {
        return new UserService(emailService());
    }
}
```

사용 클래스

```
public class UserService {
    private final EmailService emailService;

    @Autowired
    public UserService(EmailService emailService) {
        this.emailService = emailService;
    }

    public void registerUser(String email) {
        emailService.sendConfirmation(email);
    }
}
```

메인 클래스

```
public class Application {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class
        UserService userService = context.getBean(UserService.class);
        userService.registerUser("test@example.com");
    }
}
```

결론

<장점>

의존성 주입은 모듈 간의 결합도를 낮추고, 테스트 용이성과 유연성을 높이는 소프트웨어 설계 원칙.

이를 통해 모듈을 쉽게 교체하고, 독립적이고 예측 가능한 테스트를 수행 가능

<담점>

클래스 수가 증가하여 복잡성이 높아질 수 있으며, 약간의 런타임 페널티가 있을 수 있음.

의존성 주입 원칙을 잘 준수하여 상위 모듈과 하위 모듈의 독립성을 유지하고, 추상화에 의존하도록 설계하는 것이 중요

02. 팩토리 패턴

개념

- 객체를 생성하는 부분을 추상화하여, 객체 생성 코드를 사용하는 코드로부터 분리
- 상위 클래스와 하위 클래스 간의 느슨한 결합 유지, 객체 생성 로직을 하위 클래스에서 결정하도록 함

1. 객체 생성의 추상화

- 객체를 생성하는 부분을 별도의 클래스나 메서드로 추상화 → 캡슐화
- 객체 생성에 대한 구체적인 내용은 하위 클래스에서 결정, 상위 클래스는 뼈대만 결정

2. 느슨한 결합

- 상위 클래스와 하위 클래스가 분리되어 있어 상위 클래스는 객체 생성 방식에 대해 알필요x
- 상위 클래스는 더 많은 유연성 가짐

3. 유지보수성 증가

- 객체 생성 로직이 별도로 분리되어 있어, 코드 리팩토링 시 한 곳만 수정

Java 실습 예제

*기본적인 자바 상속 구조

```
/*
팩토리 패턴
상위 : 커피 공정
하위 : 라떼, 에스프레소
*/

// CoffeeType 열거형 정의
enum CoffeeType {
    LATTE,
    ESPRESSO
}

// 상위 : 추상 클래스 Coffee
abstract class Coffee {
    protected String name;

    public String getName() {
        return name;
    }
}

// 하위 : 구체적인 커피 클래스 Latte
class Latte extends Coffee {
    public Latte() {
        name = "Latte";
```

```
    }
}

// 하위 : 구체적인 커피 클래스 Espresso
class Espresso extends Coffee {
    public Espresso() {
        name = "Espresso";
    }
}

// CoffeeFactory 클래스
class CoffeeFactory {
    public static Coffee createCoffee(CoffeeType type) {
        switch (type) {
            case LATTE:
                return new Latte();
            case ESPRESSO:
                return new Espresso();
            default:
                throw new IllegalArgumentException("Invalid coffee type: " + type);
        }
    }
}

// Main 클래스
public class Main {
    public static void main(String[] args) {
        // CoffeeFactory를 사용하여 Latte 객체 생성
        Coffee coffee = CoffeeFactory.createCoffee(CoffeeType.LATTE);
        System.out.println(coffee.getName()); // 출력: Latte
    }
}
```

03. 전략 패턴

개념

- 객체의 행위를 바꾸고 싶을때, 행위를 직접 수행하지 않고 전략이라고 부르는 캡슐화한 알고리즘을 컨텍스트 안에서 바꿔주면서 상호교체가 가능하게 만드는 패턴
- = 정책 패턴

1. 전략

- 실행 할 알고리즘이나 행위를 캡슐화한 인터페이스 정의

2. 구체적인 전략

- 전략 인터페이스를 구현할 클래스
- 각기 다른 알고리즘 정의

3. 컨텍스트

- 전략 객체를 사용하여 특정 행위를 수행하는 클래스
- 전략 객체를 변경하여 행위를 동적으로 바꿀 수 있음

JAVA 예제

```
/*
전략 패턴
쇼핑 카트에서 아이템을 담아 결제할 때, LUNACard 또는 KAKAOCard라는 두 가지 전략을 사용하여 결제하는 예제.

 */


//전략 인터페이스 정의
interface PaymentStrategy {
    void pay(int amount);
}

// 구체적인 전략 클래스
class KAKAOCardStrategy implements PaymentStrategy {
    private String name;
    private String cardNumber;
    private String cvv;
    private String dateOfExpiry;

    public KAKAOCardStrategy(String name, String cardNumber, String cvv, String dateOfExpiry
        this.name = name;
        this.cardNumber = cardNumber;
        this.cvv = cvv;
        this.dateOfExpiry = dateOfExpiry;
    }

    @Override
    public void pay(int amount) {
        System.out.println(amount + " paid using KAKAOCard.");
    }
}

class LUNACardStrategy implements PaymentStrategy {
    private String emailId;
    private String password;
```

```
    public LUNACardStrategy(String email, String password) {
        this.emailId = email;
        this.password = password;
    }

    @Override
    public void pay(int amount) {
        System.out.println(amount + " paid using LUNACard.");
    }
}

//아이템 클래스 정의

class Item {
    private String name;
    private int price;

    public Item(String name, int price) {
        this.name = name;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public int getPrice() {
        return price;
    }
}

//컨텍스트 클래스 (ShoppingCart)
class ShoppingCart {
    private List<Item> items;

    public ShoppingCart() {
        this.items = new ArrayList<>();
    }

    public void addItem(Item item) {
        this.items.add(item);
    }

    public void removeItem(Item item) {
        this.items.remove(item);
    }

    public int calculateTotal() {
        int sum = 0;
        for (Item item : items) {
            sum += item.getPrice();
```

```
        }
        return sum;
    }

    public void pay(PaymentStrategy paymentMethod) {
        int amount = calculateTotal();
        paymentMethod.pay(amount);
    }
}


public class Main {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();

        Item item1 = new Item("ItemA", 100);
        Item item2 = new Item("ItemB", 300);

        cart.addItem(item1);
        cart.addItem(item2);

        // LUNACard로 결제
        cart.pay(new LUNACardStrategy("kundol@example.com", "pukubababo"));

        // KAKAOCard로 결제
        cart.pay(new KAKAOCardStrategy("Ju hongchul", "123456789", "123", "12/01"));
    }
}
```

✅ 팩토리 패턴 vs 전략 패턴

	팩토리 패턴	전략 패턴
목적	객체 생성 로직을 캡슐화하여 클라이언트 코드와 객체 생성 방식을 분리	객체의 행위를 동적으로 변경
사용 시점	객체 생성 로직을 분리하고, 다양한 객체를 생성할 필요가 있을때 사용	행위(알고리즘)을 런타임에 변경하고자 할때 사용
구조	전략(알고리즘)을 정의하는 인터페이스 + 이를 구현하는 구체적인 클래스 + 전략을 사용하는 컨텍스트	객체 생성을 위한 팩토리 인터페이스 or 클래스 + 이를 구현하는 구체적인 팩토리 클래스 + 생성되는 객체의 공통 인터페이스 또는 클래스

04. 옵저버 패턴

개념

- 객체의 상태 변화를 관찰하는 옵저버들이 주체 객체의 상태 변화를 감지하고, 변화가 있을 때마다 특정 메서드를 통해 통지를 받는 패턴
- 주로 이벤트 기반 시스템과 MVC 패턴에서 많이 사용됨
- 상속과 인터페이스 구현 개념 활용하여 설계

1. 주체(Subject):

- 상태 변화를 관찰하는 객체입니다. 상태가 변하면 모든 옵저버에게 통지를 보냅니다.

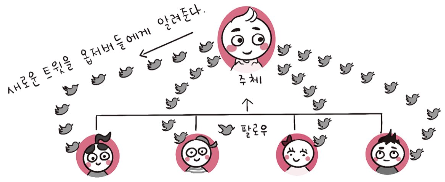
2. 옵저버(Observer):

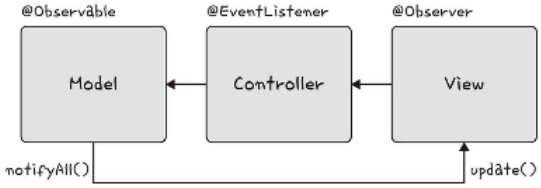
- 주체의 상태 변화를 감지하고, 변경이 있을 때 통지를 받는 객체입니다.

3. 알림(Notification):

- 주체 객체가 상태 변화를 옵저버들에게 알리는 행위입니다.

패턴 이해





주체(Model)에서 변경사항이 생겨 update() 메서드로 옵저버(View)에 알려주고 이를 기반으로 Controller 등이 작동

JAVA 예제

```
/*
옵저버 패턴

*/

//subject 인터페이스 정의
//옵저버를 등록, 제거 및 알림을 보내는 메서드 정의
import java.util.ArrayList;
import java.util.List;

interface Subject {
    void register(Observer obj);
```

```
    void unregister(Observer obj);
    void notifyObservers();
    Object getUpdate(Observer obj);
}

//Observer 인터페이스 정의
interface Observer {
    void update();
}

//Concrete Subject 클래스
class Topic implements Subject {
    private List<Observer> observers;
    private String message;

    public Topic() {
        this.observers = new ArrayList<>();
    }

    @Override
    public void register(Observer obj) {
        if (!observers.contains(obj)) {
            observers.add(obj);
        }
    }

    @Override
    public void unregister(Observer obj) {
        observers.remove(obj);
    }

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

    @Override
    public Object getUpdate(Observer obj) {
        return this.message;
    }

    public void postMessage(String msg) {
        System.out.println("Message sent to Topic: " + msg);
        this.message = msg;
        notifyObservers();
    }
}

//Concreate Observer 클래스
```

```
class TopicSubscriber implements Observer {
    private String name;
    private Subject topic;

    public TopicSubscriber(String name, Subject topic) {
        this.name = name;
        this.topic = topic;
    }

    @Override
    public void update() {
        String msg = (String) topic.getUpdate(this);
        System.out.println(name + ":: got message >> " + msg);
    }
}

public class Main {
    public static void main(String[] args) {
        Topic topic = new Topic();

        Observer a = new TopicSubscriber("a", topic);
        Observer b = new TopicSubscriber("b", topic);
        Observer c = new TopicSubscriber("c", topic);

        topic.register(a);
        topic.register(b);
        topic.register(c);
    }
}
```

*상속 vs 구현

상속 : 자식 클래스가 부모 클래스의 메서드 등을 상속 받아 사용. 자식 클래스에서 추가 및 확장 가능

→ 재사용성, 중복성의 최소화

→ 일반 클래스, abstract 클래스 기반 구현

구현 : 부모 인터페이스를 자식 클래스에서 재정의하여 구현하는 것. 부모 클래스의 메서드를 반드시 재정의 해야함\

→ 여러 인터페이스 구현 가능 (like 다중 상속)

→ 인터페이스 기반 구현

05. 프록시 패턴과 프록시 서버

개념

- 대상 객체(주채)에 접근하기 전에 그 접근을 가로채어 필터링하거나 수정하는 계층을 제공하는 패턴

- 객체의 속성이나 동작을 보완하여 보안, 데이터 검증, 캐싱 로깅 등의 기능을 추가할 수 있음

1. 프록시 개체

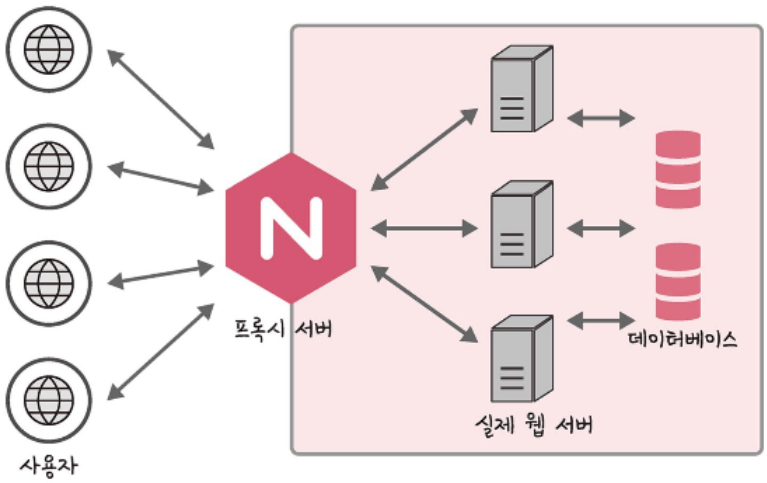
- 대상 객체에 대한 접근 제어 객체
- 주체와 도일한 인터페이스를 구현하여 주체의 기능을 대신하거나 확장할 수 있음

2. 대상 객체(주체)

- 실제로 작업을 수행하는 객체
- 클라이언트는 주체 대신 프록시 객체를 통해 주체에 접근

프록시 서버

- 클라이언트와 실제 서버 사이에서 중개 역할을 하는 서버
- 보안 강화, 캐싱, 로깅 등의 기능 제공 가능
- ex) NGINX

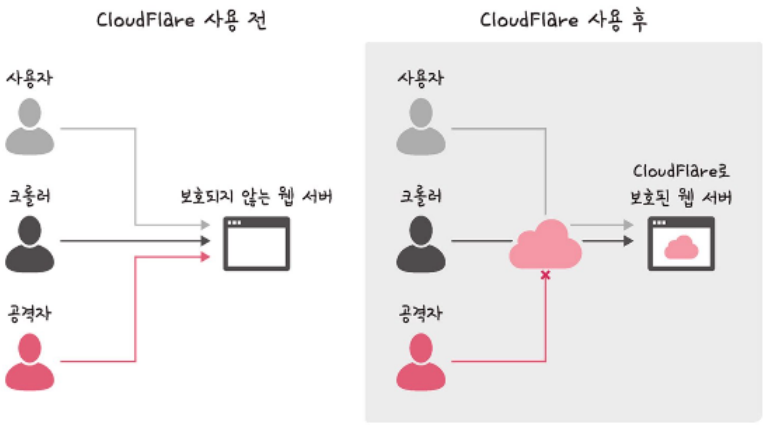


→ 앞단에 nginx, 뒤에 node.js

→ 보안 강화

→ 정적 자원을 gzip압축하거나 메인 서버 앞단에서의 로깅 가능

- ex) CloudFlare



→ DDOS 공격 방어나 HTTPS 구축에 쓰임

→ 서비스 배포 이후 의심스러운 트래픽 많이 발생되면 판단후 CAPTCHA 등을 기반으로 일정 부분 막아줌

→ https를 구축할때 별도의 인증서 설치 없이 좀 더 쉽게 https 구축 가능

- ex) CORS와 프론트엔드의 프록시 서버

  : cors는 서버가 웹 브라우저에서 리소스를 로드할 때 다른 오리진을 통해 로드하지 못하게 하는 http 헤더 기반 메커니즘

→ 프록시 서버를 둬서 프론트엔드 서버에서 요청되는 오리진을 백엔드 서버로 바꿔서 cors 에러 해결

06. 이터레이터 패턴

개념

- 컬렉션의 요소들에 순차적으로 접근할 수 있는 방법을 제공하는 패턴.
- 컬렉션의 내부 구조를 누출하지 않고도 요소들을 순회할 수 있음

1. 이터레이터

- 컬렉션의 요소들을 순회하는 객체
- next(), hasNext() 등의 메서드를 제공하여 순회를 제어

2. 컬렉션(collection)

- 이터레이터를 제공하는 개체
- iterator()메서드를 통해 이터레이터 객체 반환

장점

- 일관된 접근 방식 : 다양한 컬렉션 구조에 대해 동일한 방식으로 접근 가능
- 캡슐화 : 컬렉션의 내부 구조르 노출하지 않음

Java 예제

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

interface IteratorPattern {
    boolean hasNext();
    Object next();
}

class NameIterator implements IteratorPattern {
    List<String> names;
    int index = 0;

    public NameIterator(List<String> names) {
        this.names = names;
    }

    @Override
    public boolean hasNext() {
        return index < names.size();
    }

    @Override
    public Object next() {
        if (this.hasNext()) {
            return names.get(index++);
        }
        return null;
    }
}

class NameRepository {
    List<String> names = new ArrayList<>();
```

```
    public void addName(String name) {
        names.add(name);
    }

    public IteratorPattern getIterator() {
        return new NameIterator(names);
    }
}

public class Main {
    public static void main(String[] args) {
        NameRepository nameRepository = new NameRepository();
        nameRepository.addName("Alice");
        nameRepository.addName("Bob");
        nameRepository.addName("Charlie");

        IteratorPattern iterator = nameRepository.getIterator();
        while (iterator.hasNext()) {
            String name = (String) iterator.next();
            System.out.println("Name: " + name);
        }
    }
}
```

07. 노출 패턴

개념

- 즉시 실행 함수를 통해 private, public 같은 접근 제어자를 만드는 패턴

1.즉시 실행 함수

- 함수를 정의 하자마자 바로 실행하는 함수

2.private 변수/함수

- 즉시 실행 함수 내에서 정의된 변수와 함수로 외부에서 접근 불가

3.public 변수/함수

- 즉시 실행 함수는 반환하는 객체에 포함된 변수와 함수로 외부에서 접근 가능

장점

- 캡슐화

- 명확한 인터페이스 : 공개하는 부분 명확하게 지정

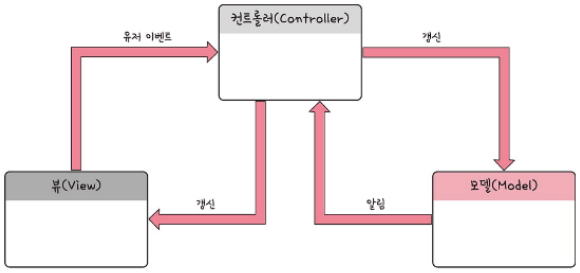
JAVA 예제

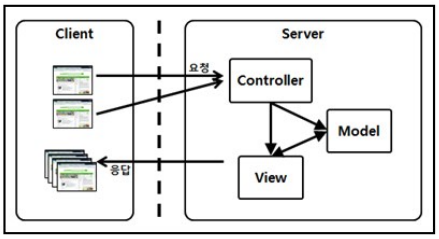
```
class Module {
    // private 변수와 메서드
    private int privateVariable = 1;
    private int privateMethod() {
        return 2;
    }

    // public 변수와 메서드
    public int publicVariable = 3;
    public int publicMethod() {
        return 4;
    }

    // 접근 제어자에 따른 메서드 노출
    public int getPrivateVariable() {
        return privateVariable;
    }

    public int callPrivateMethod() {
        return privateMethod();
    }
}

public class Main {
    public static void main(String[] args) {
        Module module = new Module();

        // public 변수와 메서드 접근
        System.out.println("Public Variable: " + module.publicVariable); // 3
        System.out.println("Public Method: " + module.publicMethod()); // 4

        // private 변수와 메서드 접근 (공개된 메서드를 통해)
        System.out.println("Private Variable: " + module.getPrivateVariable()); // 1
        System.out.println("Private Method: " + module.callPrivateMethod()); // 2

        // 직접 접근 불가
        // System.out.println(module.privateVariable); // 컴파일 오류
        // System.out.println(module.privateMethod()); // 컴파일 오류
    }
}
```

08. MVC 패턴

개념

모델 + 뷰 + 컨트롤러로 이루어진 패턴





→ 애플리케이션 구성 요소를 3가지로 역할 분리하여 집중하여 개발할 수 있도록

모델

- 애플리케이션의 데이터인 데이터베이스, 상수, 변수 등을 뜻함

  ex) 글자 내용, 글자 위치, 글자 포맷....
- 뷰에서 데이터를 생성하거나 수정하면 컨트롤러를 통해 모델을 생성하거나 갱신

뷰

- inputbox, checkout, textarea 등 사용자 이터페이스 요소를 나타냄
- 모델을 기반으로 사용자가 볼 수 있는 화면
- 모델이 가지고 있는 정보 따로 저장 x → 단순히 화면 표시하는 정보만 가지고 있어야함
- 변경 발생 시 컨트롤러에 전달

컨트롤러

- 하나 이상의 모델과 하나 이상의 뷰 연결해주는 다리 역할
- 이벤트 등 메인 로직 담당
- 모델과 뷰의 생명 주기 관리
- 비즈니스 로직 처리

장점

- 나눠진 모듈 → 가독성 높이고, 비즈니스 로직 분리 → 협업 수월

JAVA 예제

model

```
// Model: User 클래스
public class User {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public String getName() {
        return name;
    }

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

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

view

```
// View: UserView 클래스
public class UserView {
    public void printUserDetails(String userName, String userEmail) {
        System.out.println("User: ");
        System.out.println("Name: " + userName);
        System.out.println("Email: " + userEmail);
    }
}
```

controller

```
// Controller: UserController 클래스
public class UserController {
    private User model;
    private UserView view;
```

```
    public UserController(User model, UserView view) {
        this.model = model;
        this.view = view;
    }

    public void setUserName(String name) {
        model.setName(name);
    }

    public String getUserName() {
        return model.getName();
    }

    public void setUserEmail(String email) {
        model.setEmail(email);
    }

    public String getUserEmail() {
        return model.getEmail();
    }

    public void updateView() {
        view.printUserDetails(model.getName(), model.getEmail());
    }
}
```

main

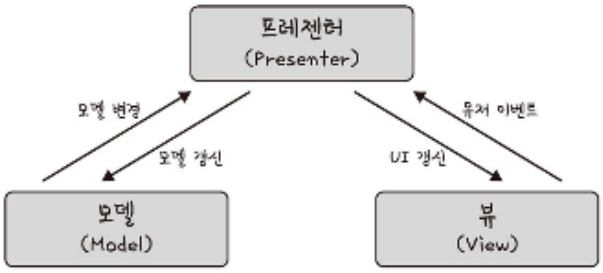
```
// Main 클래스
public class Main {
    public static void main(String[] args) {
        // 모델 객체 생성
        User model = new User("John Doe", "john.doe@example.com");

        // 뷰 객체 생성
        UserView view = new UserView();

        // 컨트롤러 객체 생성
        UserController controller = new UserController(model, view);

        // 초기 상태 출력
        controller.updateView();

        // 모델 데이터 변경
        controller.setUserName("Jane Smith");
        controller.setUserEmail("jane.smith@example.com");

        // 변경된 상태 출력
        controller.updateView();
```

```
    }
}
```

09. MVP 패턴

개념

MVC에서 파생. 컨트롤러가 프레젠터로 교체된 패턴



→ 뷰와 프레젠터는 1:1 관계 → mvc 패턴보다 강한 결합

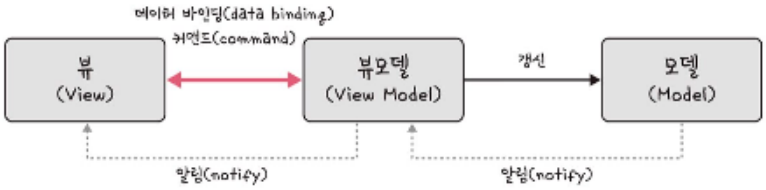
MVC vs MVP

	MVC	MVP
역할	컨트롤러가 뷰와 모델 연결 뷰의 이벤트 처리	프레젠터가 뷰와 모델 연결 모든 비즈니스 로직 담당 뷰에 대한 참조 가지며 뷰는 프레젠터를 호출하여 사용자 입력 처리
뷰의 역할	뷰가 상대적으로 더 많은 역할 사용자 이벤트 직접 처리 가능	뷰는 단순히 사용자 인터페이스 표시, 사용자 입력을 프레젠터에 전달하는 역할

10. MVCC 패턴

개념

view + view model + model



Veiw Model

- 뷰를 더 추상화한 계층
- MVC 패턴과 다르게 커맨드와 데이터 바인딩을 가짐
- 뷰와 뷰모델 사이의 양방향 데이터 바인딩을 지원
- UI를 별도의 코드 수정 없이 재사용 가능, 단위 테스팅이 쉬움
- 프론트에서 많이 사용됨

**모르는 용어??

- **데이터 바인딩**: 뷰와 뷰모델 사이의 데이터 동기화를 자동화하여, 뷰모델의 속성이 변경되면 뷰가 자동으로 업데이트되고, 뷰의 입력이 뷰모델로 자동으로 전달됩니다. 이는 수동으로 UI를 업데이트하는 번거로움을 줄여줍니다.
- **커맨드**: 뷰에서 발생하는 사용자 입력을 처리하는 메서드를 캡슐화하여, 뷰모델이 사용자 입력을 처리할 수 있게 합니다. 이는 뷰와 비즈니스 로직을 분리하여 코드의 가독성과 유지보수성을 높입니다.