**Singleton Design Pattern**

Singleton đảm bảo chỉ duy nhất một thể hiện (instance) được tạo ra và nó sẽ cung cấp cho bạn một method để có thể truy xuất được thể hiện duy nhất đó mọi lúc mọi nơi trong chương trình.

Sử dụng khi:

* Đảm bảo rằng chỉ có một instance của lớp.
* Việc quản lý việc truy cập tốt hơn vì chỉ có một thể hiện duy nhất.
* Có thể quản lý số lượng thể hiện của một lớp trong giớn hạn chỉ định.

Implement:

**Eager initialization**

Singleton Class được khởi tạo ngay khi được gọi đến. Đây là cách dễ nhất nhưng nó có một nhược điểm mặc dù instance đã được khởi tạo mà có thể sẽ không dùng tới.

Ví dụ:

package com.gpcoder.patterns.creational.singleton;

public class EagerInitializedSingleton {

    private static final EagerInitializedSingleton INSTANCE = new EagerInitializedSingleton();

    // Private constructor to avoid client applications to use constructor

    private EagerInitializedSingleton() {

    }

    public static EagerInitializedSingleton getInstance() {

        return INSTANCE;

    }

}

**Static block initialization**

Cách làm tương tự như Eager initialization chỉ khác phần static block cung cấp thêm lựa chọn cho việc handle exception hay các xử lý khác.

Ví dụ:

package com.gpcoder.patterns.creational.singleton;

public class StaticBlockSingleton {

    private static final StaticBlockSingleton INSTANCE;

    private StaticBlockSingleton() {

    }

    // Static block initialization for exception handling

    static {

        try {

            INSTANCE = new StaticBlockSingleton();

        } catch (Exception e) {

            throw new RuntimeException("Exception occured in creating singleton instance");

        }

    }

    public static StaticBlockSingleton getInstance() {

        return INSTANCE;

    }

}

**Lazy Initialization**

Là một cách làm mang tính mở rộng hơn so với 2 cách làm trên và hoạt động tốt trong môi trường đơn luồng (single-thread).

package com.gpcoder.patterns.creational.singleton;

public class LazyInitializedSingleton {

    private static LazyInitializedSingleton instance;

    private LazyInitializedSingleton() {

    }

    public static LazyInitializedSingleton getInstance() {

        if (instance == null) {

            instance = new LazyInitializedSingleton();

        }

        return instance;

    }

}

Cách này đã khắc phục được nhược điểm của cách Eager initialization, chỉ khi nào getInstance() được gọi thì instance mới được khởi tạo. Tuy nhiên, cách này chỉ sử dụng tốt trong trường hợp đơn luồng (single-thread), trường hợp nếu có nhiều luồng (multi-thread) cùng chạy và cùng gọi hàm getInstance() tại cùng một thời điểm thì có thể có nhiều hơn 1 thể hiện của instance. Để khắc phục nhược điểm này chúng ta sử dụng Thread Safe Singleton.

**Thread Safe Singleton**

Cách đơn giản nhất là chúng ta gọi phương thức synchronized của hàm getInstance() và như vậy hệ thống đảm bảo rằng tại cùng một thời điểm chỉ có thể có 1 luồng có thể truy cập vào hàm getInstance() và đảm bảo rằng chỉ có duy nhất 1 thể hiện của class.

package com.gpcoder.patterns.creational.singleton;

public class ThreadSafeLazyInitializedSingleton {

    private static volatile ThreadSafeLazyInitializedSingleton instance;

    private ThreadSafeLazyInitializedSingleton() {

    }

    public static synchronized ThreadSafeLazyInitializedSingleton getInstance() {

        if (instance == null) {

            instance = new ThreadSafeLazyInitializedSingleton();

        }

        return instance;

    }

}

Cách này có nhược điểm là một phương thức synchronized sẽ chạy rất chậm và tốn hiệu năng, bất kỳ Thread nào gọi đến đều phải chờ nếu có một Thread khác đang sử dụng. Có những tác vụ xử lý trước và sau khi tạo thể hiện không cần thiết phải block. Vì vậy chúng ta cần cải tiến nó đi 1 chút với Double Check Locking Singleton.

**Double Check Locking Singleton**

Để implement theo cách này, chúng ta sẽ kiểm tra sự tồn tại thể hiện của lớp, với sự hổ trợ của đồng bộ hóa, hai lần trước khi khởi tạo. Phải khai báo volatile cho instance để tránh lớp làm việc không chính xác do quá trình tối ưu hóa của trình biên dịch.

package com.gpcoder.patterns.creational.singleton;

public class DoubleCheckLockingSingleton {

    private static volatile DoubleCheckLockingSingleton instance;

    private DoubleCheckLockingSingleton() {

    }

    public static DoubleCheckLockingSingleton getInstance() {

        // Do something before get instance ...

        if (instance == null) {

            // Do the task too long before create instance ...

            // Block so other threads cannot come into while initialize

            synchronized (DoubleCheckLockingSingleton.class) {

                // Re-check again. Maybe another thread has initialized before

                if (instance == null) {

                    instance = new DoubleCheckLockingSingleton();

                }

            }

        }

        // Do something after get instance ...

        return instance;

    }

}

**Bill Pugh Singleton Implementation**

Với cách làm này bạn sẽ tạo ra static nested class với vai trò 1 Helper khi muốn tách biệt chức năng cho 1 class function rõ ràng hơn. Đây là cách thường hay được sử dụng và có hiệu suất tốt

package com.gpcoder.patterns.creational.singleton;

public class BillPughSingleton {

    private BillPughSingleton() {

    }

    public static BillPughSingleton getInstance() {

        return SingletonHelper.INSTANCE;

    }

    private static class SingletonHelper {

        private static final BillPughSingleton INSTANCE = new BillPughSingleton();

    }

}

Khi Singleton được tải vào bộ nhớ thì SingletonHelper chưa được tải vào. Nó chỉ được tải khi và chỉ khi phương thức getInstance() được gọi. Với cách này tránh được lỗi cơ chế khởi tạo instance của Singleton trong Multi-Thread, performance cao do tách biệt được quá trình xử lý. Do đó, cách làm này được đánh giá là cách triển khai Singleton nhanh và hiệu quả nhất.

**Enum Singleton**

Khi dùng enum thì các params chỉ được khởi tạo 1 lần duy nhất, đây cũng là cách giúp bạn tạo ra Singleton instance.

package com.gpcoder.patterns.creational.singleton;

/\*\*

 \* Singleton implementation using enum initialization

 \*/

public enum EnumSingleton {

    INSTANCE;

}

Enum có thể sử dụng như một Singleton, nhưng nó có nhược điểm là không thể extends từ một lớp được, nên khi sử dụng cần xem xét vấn đề này.

Hàm constructor của enum là lazy, nghĩa là khi được sử dụng mới chạy hàm khởi tạo và nó chỉ chạy duy nhất một lần. Nếu muốn sử dụng như một eager singleton thì cần gọi thực thi trong một static block khi start chương trình.

**Factory Design Pattern**

Nhiệm vụ của Factory Pattern là quản lý và trả về các đối tượng theo yêu cầu, giúp cho việc khởi tạo đổi tượng một cách linh hoạt hơn.

Tạo đối tượng mà không để lộ logic tạo đối tượng ở phía người dùng và tham chiếu đến đối tượng mới được tạo ra bằng cách sử dụng một interface chung.

Factory Pattern được sử dụng khi có một class cha (super-class) với nhiều class con (sub-class), dựa trên đầu vào và phải trả về 1 trong những class con đó.

Một Factory Pattern bao gồm các thành phần cơ bản sau:

* **Super Class**: môt supper class trong Factory Pattern có thể là một **interface**, **abstract class** hay một **class** thông thường.
* **Sub Classes**: các sub class sẽ implement các phương thức của **supper class** theo nghiệp vụ riêng của nó.
* **Factory Class**: một class chịu tránh nhiệm khởi tạo các đối tượng **sub class** dựa theo tham số đầu vào. Lưu ý: lớp này là Singleton hoặc cung cấp một **public static method** cho việc truy xuất và khởi tạo đối tượng. Factory class sử dụng if-else hoặc switch-case để xác định class con đầu ra.

Ví dụ:

Supper Class:

public interface Bank {

    String getBankName();

}

Sub Classes:

package com.gpcoder.patterns.creational.factorymethod;

public class TPBank implements Bank {

    @Override

    public String getBankName() {

        return "TPBank";

    }

}

package com.gpcoder.patterns.creational.factorymethod;

public class VietcomBank implements Bank {

    @Override

    public String getBankName() {

        return "VietcomBank";

    }

}

Factory class:

public class BankFactory {

    private BankFactory() {

    }

    public static final Bank getBank(BankType bankType) {

        switch (bankType) {

        case TPBANK:

            return new TPBank();

        case VIETCOMBANK:

            return new VietcomBank();

        default:

            throw new IllegalArgumentException("This bank type is unsupported");

        }

    }

}

Bank type:

public enum BankType {

    VIETCOMBANK, TPBANK;

}

Client:

public class Client {

    public static void main(String[] args) {

        Bank bank = BankFactory.getBank(BankType.TPBANK);

        System.out.println(bank.getBankName()); // TPBank

    }

}

Như bạn thấy, phía client chỉ cần gọi duy nhất một phương thức BankFactory.getBank() là có thể sử dụng được dịch vụ của một ngân hàng bất kỳ.

Khi hệ thống muốn cung cấp thêm dịch vụ của một ngân hàng khác, chẳng hạn VietinBank, thì cần tạo thêm một class mới implement từ interface Bank, và thêm vào logic khởi tạo Bank trong Factory là xong. Nó không làm ảnh hưởng đến code ở phía Client.

**Chain of Responsibility**

Cho phép một đối tượng gửi một yêu cầu nhưng không biết đối tượng nào sẽ nhận và xử lý nó. Điều này được thực hiện bằng cách kết nối các đối tượng nhận yêu cầu thành một chuỗi (chain) và gửi yêu cầu theo chuỗi đó cho đến khi có một đối tượng xử lý nó.

Cài đặt:

Các thành phần tham gia mẫu Chain of Responsibility:

* **Handler** : định nghĩa 1 interface để xử lý các yêu cầu. Gán giá trị cho đối tượng successor (không bắt buộc).
* **ConcreteHandler** : xử lý yêu cầu. Có thể truy cập đối tượng successor (thuộc class Handler). Nếu đối tượng ConcreateHandler không thể xử lý được yêu cầu, nó sẽ gởi lời yêu cầu cho successor của nó.
* **Client** : tạo ra các yêu cầu và yêu cầu đó sẽ được gửi đến các đối tượng tiếp nhận.

Client gửi một yêu cầu để được xử lý gửi nó đến chuỗi (chain) các trình xử lý (handers), đó là các lớp mở rộng lớp Handler. Mỗi Hanlder trong chuỗi lần lượt cố gắng xử lý yêu cầu nhận được từ Client. Nếu trình xử lý đầu tiên (ConcreteHandler) có thể xử lý nó, thì yêu cầu sẽ được xử lý. Nếu không được xử lý thì sẽ gửi đến trình xử lý tiếp theo trong chuỗi (ConcreteHandler + 1).

Một ví dụ là ứng dụng phê duyệt xin nghỉ phép. Nếu xin nghỉ <=3 ngày thì Supervisor có thể phê duyệt (approve). Nếu xin nghỉ <=5 ngày thì DeliveryManager có thể approve. Nếu xin nghỉ >5 ngày thì phải được approve bởi Director. Quy trình này có thể linh động tùy theo quy mô phát triển của công ty.

LeaveRequest.java

package com.gpcoder.patterns.behavioral.cor.leaverequest;

public class LeaveRequest {

    private int days;

    public LeaveRequest(int days) {

        this.days = days;

    }

    public int getDays() {

        return days;

    }

}

Approver.java

package com.gpcoder.patterns.behavioral.cor.leaverequest;

public abstract class Approver {

    protected Approver nextApprover;

    public void approveLeave(LeaveRequest request) {

        System.out.println("Checking permission for " + this.getClass().getSimpleName());

        if (this.canApprove(request.getDays())) {

            this.doApproving(request);

        } else if (nextApprover != null) {

            nextApprover.approveLeave(request);

        }

    }

    public void setNext(Approver approver) {

        this.nextApprover = approver;

    }

    protected abstract boolean canApprove(int numberOfDays);

    protected abstract void doApproving(LeaveRequest request);

}

Supervisor.java

package com.gpcoder.patterns.behavioral.cor.leaverequest;

public class Supervisor extends Approver {

    @Override

    protected boolean canApprove(int numberOfDays) {

        return numberOfDays <= 3;

    }

    @Override

    protected void doApproving(LeaveRequest request) {

        System.out.println("Leave request approved for " + request.getDays() + " days by Supervisor");

    }

}

DeliveryManager.java

package com.gpcoder.patterns.behavioral.cor.leaverequest;

public class DeliveryManager extends Approver {

    @Override

    protected boolean canApprove(int numberOfDays) {

        return numberOfDays <= 5;

    }

    @Override

    protected void doApproving(LeaveRequest request) {

        System.out.println("Leave request approved for " + request.getDays() + " days by Delivery Manager");

    }

}

Director.java

package com.gpcoder.patterns.behavioral.cor.leaverequest;

public class Director extends Approver {

    @Override

    protected boolean canApprove(int numberOfDays) {

        return numberOfDays > 5;

    }

    @Override

    protected void doApproving(LeaveRequest request) {

        System.out.println("Leave request approved for " + request.getDays() + " days by Director");

    }

}

LeaveRequestWorkFlow.java

package com.gpcoder.patterns.behavioral.cor.leaverequest;

public class LeaveRequestWorkFlow {

    public static Approver getApprover() {

        Approver supervisor = new Supervisor();

        Approver manager = new DeliveryManager();

        Approver director = new Director();

        supervisor.setNext(manager);

        manager.setNext(director);

        return supervisor;

    }

}

App.java

package com.gpcoder.patterns.behavioral.cor.leaverequest;

public class App {

    public static void main(String[] args) {

        LeaveRequestWorkFlow.getApprover().approveLeave(new LeaveRequest(2));

        System.out.println("---");

        LeaveRequestWorkFlow.getApprover().approveLeave(new LeaveRequest(5));

        System.out.println("---");

        LeaveRequestWorkFlow.getApprover().approveLeave(new LeaveRequest(7));

    }

}

Sử dụng khi:

* Có nhiều hơn một đối tượng có khả thực xử lý một yêu cầu trong khi đối tượng cụ thể nào xử lý yêu cầu đó lại phụ thuộc vào ngữ cảnh sử dụng.
* Muốn gửi yêu cầu đến một trong số vài đối tượng nhưng không xác định đối tượng cụ thể nào sẽ xử lý yêu cầu đó.
* Khi cần phải thực thi các trình xử lý theo một thứ tự nhất định..
* Khi một tập hợp các đối tượng xử lý có thể thay đổi động: tập hợp các đối tượng có khả năng xử lý yêu cầu có thể không biết trước, có thể thêm bớt hay thay đổi thứ tự sau này.

**Adapter Pattern**

Giữ vai trò trung gian giữa hai lớp, chuyển đổi interface của một hay nhiều lớp có sẵn thành một interface khác, thích hợp cho lớp đang viết. Điều này cho phép các lớp có các interface khác nhau có thể dễ dàng giao tiếp tốt với nhau thông qua interface trung gian, không cần thay đổi code của lớp có sẵn cũng như lớp đang viết.

Một Adapter Pattern bao gồm các thành phần cơ bản sau:

* **Adaptee**: định nghĩa interface không tương thích, cần được tích hợp vào.
* **Adapter**: lớp tích hợp, giúp interface không tương thích tích hợp được với interface đang làm việc. Thực hiện việc chuyển đổi interface cho Adaptee và kết nối Adaptee với Client.
* **Target**: một interface chứa các chức năng được sử dụng bởi Client (domain specific).
* **Client**: lớp sử dụng các đối tượng có interface Target.

Sử dụng khi:

* Adapter Pattern giúp nhiều lớp có thể làm việc với nhau dễ dàng mà bình thường không thể. Một trường hợp thường gặp phải và có thể áp dụng Adapter Pattern là khi không thể kế thừa lớp A, nhưng muốn một lớp B có những xử lý tương tự như lớp A. Khi đó chúng ta có thể cài đặt B theo Object Adapter, các xử lý của B sẽ gọi những xử lý của A khi cần.
* Khi muốn sử dụng một lớp đã tồn tại trước đó nhưng interface sử dụng không phù hợp như mong muốn.
* Khi muốn tạo ra những lớp có khả năng sử dụng lại, chúng phối hợp với các lớp không liên quan hay những lớp không thể đoán trước được và những lớp này không có những interface tương thích.
* Cần phải có sự chuyển đổi interface từ nhiều nguồn khác nhau.
* Khi cần đảm bảo nguyên tắc [Open/ Close](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/#Open-Closed_principle_OCP) trong một ứng dụng.

Ví dụ:

Một người Việt muốn trao đổi với một người Nhật. Tuy nhiên, 2 người này không biết ngôn ngữ của nhau nên cần phải có một người để chuyển đổi từ ngôn ngữ tiếng Việt sang ngôn ngữ tiếng Nhật. Chúng ta sẽ mô hình hóa trường hợp này với Adapter Pattern như sau:

VietnameseTarget.java

package com.gpcoder.patterns.structural.adapter;

public interface VietnameseTarget {

    void send(String words);

}

JapaneseAdaptee.java

package com.gpcoder.patterns.structural.adapter;

public class JapaneseAdaptee {

    public void receive(String words) {

        System.out.println("Retrieving words from Adapter ...");

        System.out.println(words);

    }

}

TranslatorAdapter.java

package com.gpcoder.patterns.structural.adapter;

public class TranslatorAdapter implements VietnameseTarget {

    private JapaneseAdaptee adaptee;

    public TranslatorAdapter(JapaneseAdaptee adaptee) {

        this.adaptee = adaptee;

    }

    @Override

    public void send(String words) {

        System.out.println("Reading Words ...");

        System.out.println(words);

        String vietnameseWords = this.translate(words);

        System.out.println("Sending Words ...");

        adaptee.receive(vietnameseWords);

    }

    private String translate(String vietnameseWords) {

        System.out.println("Translated!");

        return "こんにちは";

    }

}

VietnameseClient.java

package com.gpcoder.patterns.structural.adapter;

public class VietnameseClient {

    public static void main(String[] args) {

        VietnameseTarget client = new TranslatorAdapter(new JapaneseAdaptee());

        client.send("Xin chào");

    }

}

Output của chương trình trên:

Reading Words ...

Xin chào

Translated!

Sending Words ...

Retrieving words from Adapter ...

こんにちは