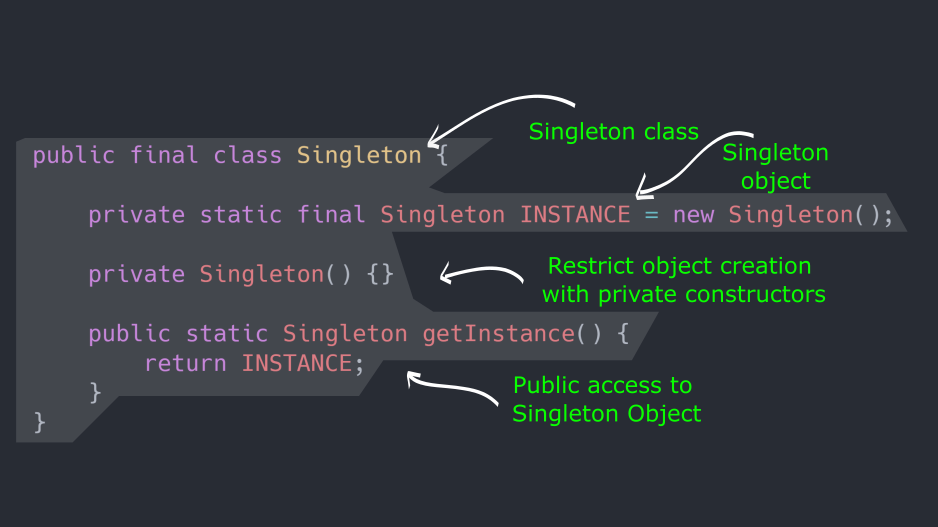
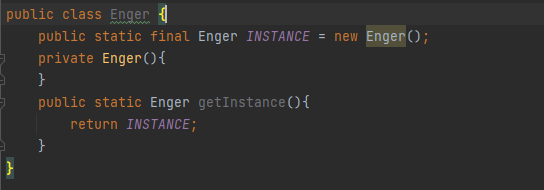
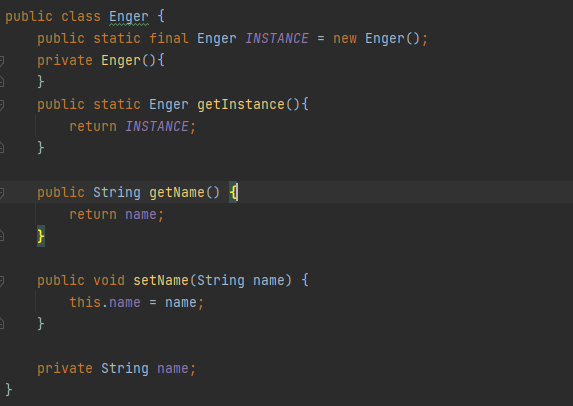
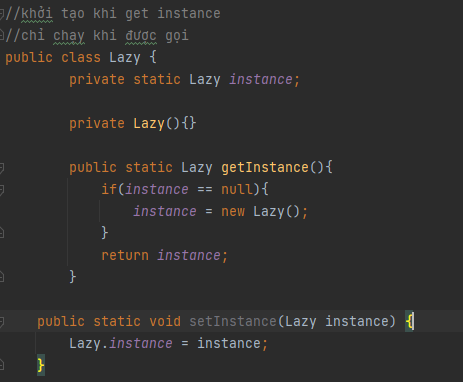
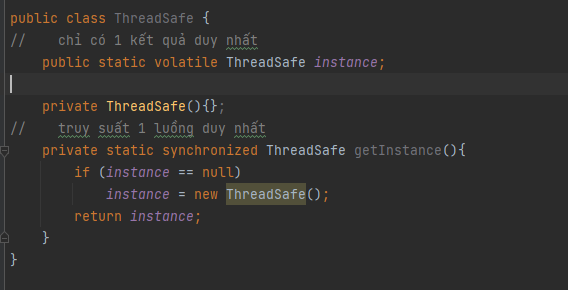
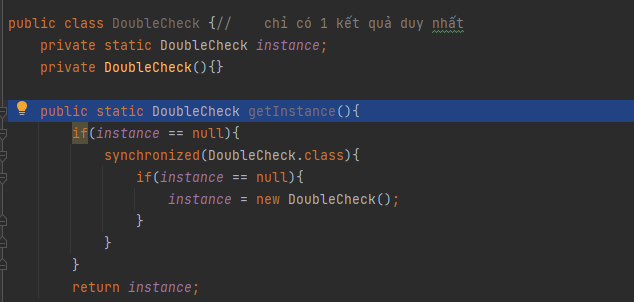
# Design Pattern

* giúp mở rộng code, tối ưu hoá, sử dụng lại code, dễ đọc, dễ nâng cấp
* 23 mẫu - 3 nhóm
  + Creation pattern - khởi tạo 5 mẫu khởi tạo các đối tượng
  + Structural pattern - cấu trúc 7 mẫu - thiết lập định nghĩa các đối tượng
  + Behavioral pattern - nhóm tương tác 11 mẫu - giao tiếp giữa các object
* phân biệt interface và abtract( chưa rõ ràng) class

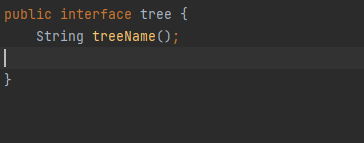
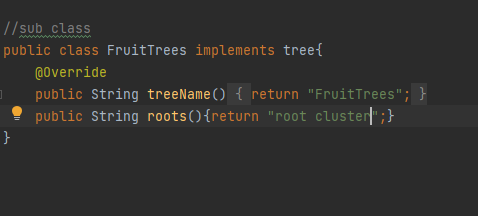
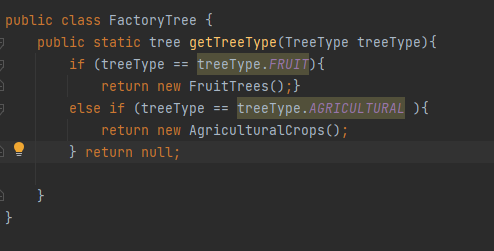
|  |  |  |
| --- | --- | --- |
|  | interface | abtract |
|  |  | Không tạo được đối tượng |
| method | Tất cả các method trong interface đều là abstract | Từ khoá abstract |
|  |  | Tái sử dụng code, sửa code |
|  | Được kế thừa nhiều cái( đa kế thừa) | Chỉ được kế thừa 1( Đơn kế thừa) |

## Creation pattern

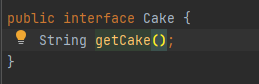
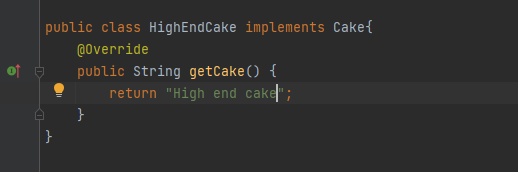
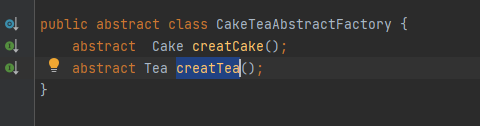
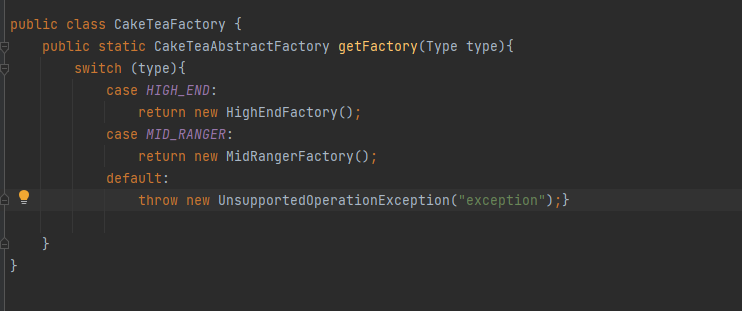
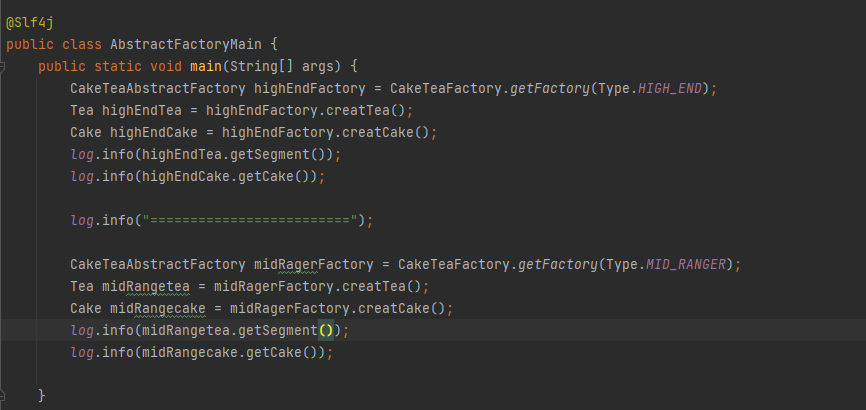
### **Singleton**

* **Khái niệm**: trả về duy nhất trường được định nghĩa trong class
* 
  + **Nguyên tắc:** 
    - Hàm tạo hay còn gọi là constructor có thể truy cập với private tức chỉ có thể truy xuất trong nội bộ của class. 
  + **Các cách dung singleton**:
    - Đây là cách dễ nhất nhưng nó có một nhược điểm là mặc dù instance đã được khởi tạo nhưng có thể sẽ không dùng tới 
    - Khắc phục cách 1, chỉ khi được gọi mới run, sử dụng trong trường hợp đơn luồng. Trường hợp dùng 2 luồng 
    - đảm bảo rằng chỉ có duy nhất 1 thể hiện của class Tuy nhiên một menthod synchronized sẽ chạy rất chậm và tốn hiệu năng vì vậy chúng ta cần cải tiến nó đi 1 chút. 
    - sử dụng cho đa luồng 
  + **các trường hợp sử dụng**:
    - Đảm bảo chỉ có một thể hiện(instance) duy nhất của một class.
    - Có thể quản lý số lượng thể hiện của một lớp trong giới hạn ta muốn.

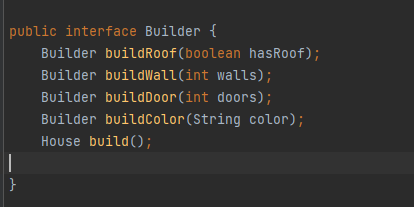
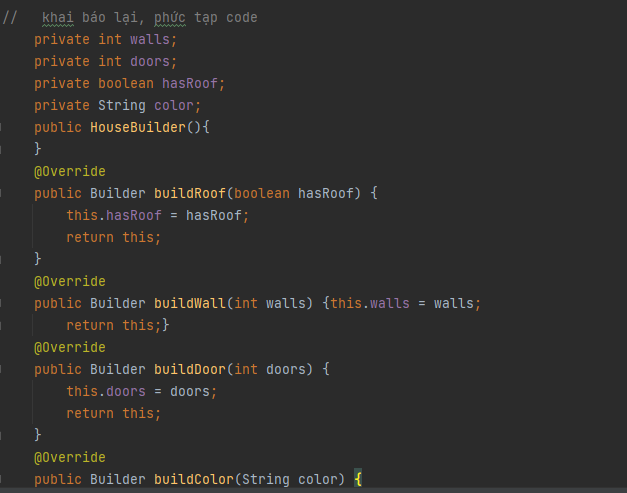
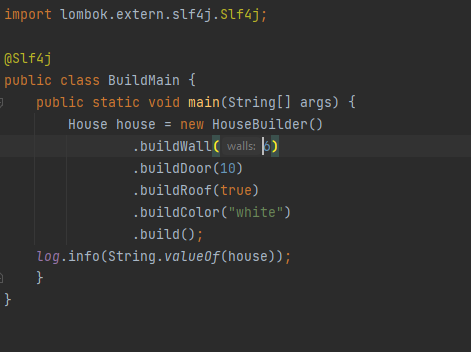
### Factory method

* **Khái niệm**:
  + Nhiệm vụ của nó là quản lý và trả về những đối tượng theo đúng những gì được yêu cầu, sẽ giúp việc khởi tạo đối tượng linh hoạt hơn.
  + Factory làm nhiệm vụ sản xuất hàng hóa thì Factory Patterns cũng vậy nhưng giúp chúng ta sản xuất đối tượng. Nó sẽ che giấu đi logic tạo ra những đối tượng đó giống như việc chúng ta chỉ có thể nhìn thấy nguyên liệu đi vào và sản phẩm đi ra từ nhà máy mà không thể biết sản phẩm phải trải qua những quy trình ra sao.
* **Trường hợp sử dụng**: Được sử dụng trong trường hợp có một class cha với nhiều class con kế thừa và chúng ta cần tùy theo trường hợp mà khởi tạo một trong những class con đó.
* **Các thành phần của Factory pattern**:
  + Super Class: môt supper class trong Factory Pattern có thể là một interface, abstract class hoặc cũng có thể là một class thông thường. 
  + Sub Classes: các class con sẽ kế thừa từ class cha theo nghiệp vụ riêng của nó.
  + 
  + Factory Class: một class chịu tránh nhiệm khởi tạo các đối tượng class con dựa theo tham số đầu vào. (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.)
  + 

### Abstract Factory Method

* **Khái niệm:** Abstract Factory pattern là một trong những Creational pattern. Nó sẽ tạp ra những Super-Factory để tạo ra những factory khác
* **Các thành phần của Abstract Factory Method:** 
  + AbstractFactory: Là một Interface hoạc abstract class chứa các phưng thức tạo ra các đối tượng abstract.
  + 
  + ConcreteFactory (hay mình thường nghĩ nó là Sub factory :v): cài đặt các phương thức để tao ra các đối tượng cụ thể.
  + 
  + AbstractProduct(hay các supper-class) : Là dạng interface hoặc abstract class để định nghĩa ra đối tượng cha hay đối tượng abstract.
  + 
  + Product (hay các sub-class): Là các đối tượng cụ thể cài đặt các method từ AbstractProduct.
  + 
  + Client: Đại diện cho nơi sử dụng AbstractFactory và các AbstractProduct.
  + 
* **Tại sao nên sử dụng**:
  + Giống như Factory method thông thường thì nó cũng che giấu logic khởi tạo đối tượng. Độc lập hoàn toàn giữa nơi sử dụng và nơi khởi tạo => Hệ thống dễ dàng mở rộng vì việc khởi tạo và sử dụng tách biệt nhau.
  + So với Factory method thì khi quá lớn sẽ dễ dàng quản lý và mở rộng hơn.
  + Như đã nói thì sẽ dễ dàng xây dựng nên một module hay một hệ thống độc lập để xử dụng trong các module và hệ thống khác.

### Builder Design Pattern

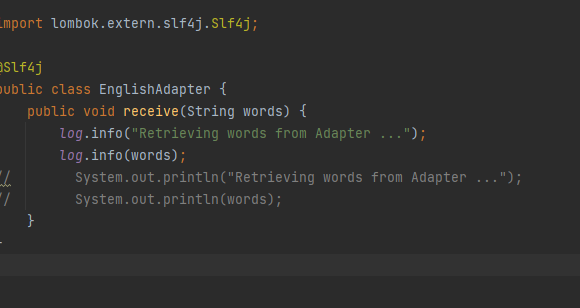
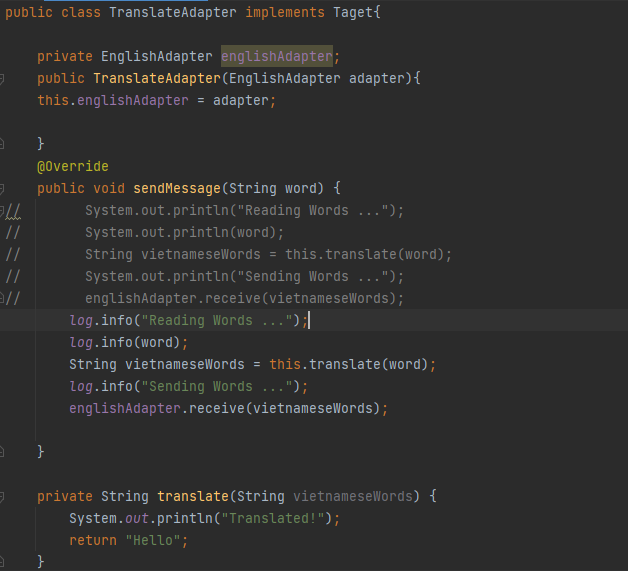
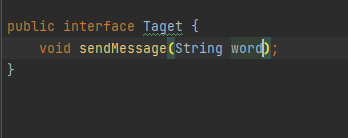
* **Khái niệm:** Builder pattern là mẫu thiết kế đối tượng được tạo ra để xây dựng một đôi tượng phức tạp bằng cách sử dụng các đối tượng đơn giản và sử dụng tiếp cận từng bước, việc xây dựng các đối tượng đôc lập với các đối tượng khác.
* Có ba vấn đề chính với Factory Pattern và Abstract Factory Pattern khi Object có nhiều thuộc tính:
  + Quá nhiều tham số phải truyền vào từ phía client tới Factory Class.
  + Một số tham số có thể là tùy chọn nhưng trong Factory Pattern, chúng ta phải gửi tất cả tham số, với tham số tùy chọn nếu không nhập gì thì sẽ truyền là null.
  + Nếu một Object có quá nhiều thuộc tính thì việc tạo sẽ phức tạp.
* Builder Pattern được xây dựng để khắc phục một số nhược điểm của Factory Pattern và Abstract Factory Pattern khi mà Object có nhiều thuộc tính.
* **Các thành phần của Builder:** 
  + Product : chính là đối tượng chúng ta cần tạo, có nhiều thuộc tính phức tạp.
  + 
  + Builder : là abstract class hoặc interface khai báo phương thức dùng để tạo đối tượng.
  + 
  + ConcreteBuilder : kế thừa Builder và cài đặt chi tiết cách tạo ra đối tượng. Nó sẽ xác định và nắm giữ các thể hiện mà nó tạo ra, đồng thời nó cũng cung cấp phương thức để trả các các thể hiện mà nó đã tạo ra trước đó.
  + 
  + Client : Đại diện cho nơi mà chúng ta sẽ khởi tạo ra đối tượng.
  + 

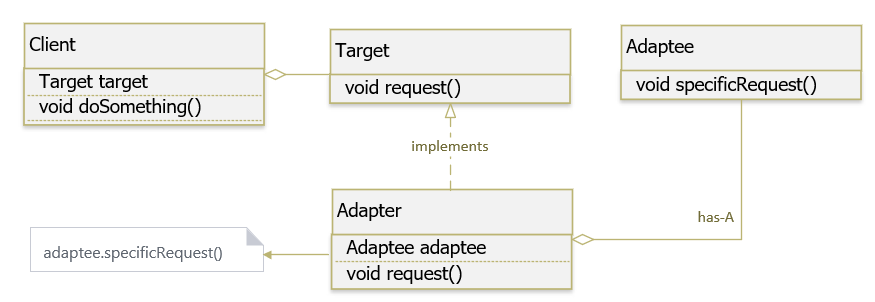
## Structural design patterns

### Adapter

* **Khái niệm:** Adapter Pattern cho phép các inteface (giao diện) không liên quan tới nhau có thể làm việc cùng nhau. Đối tượng giúp kết nối các interface gọi là Adapter.

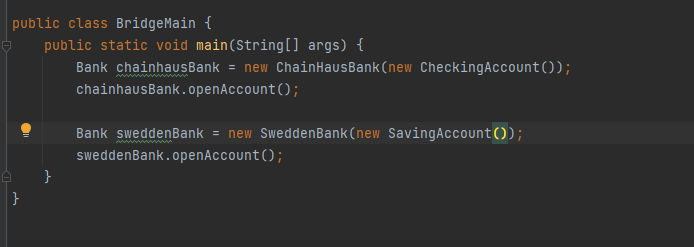
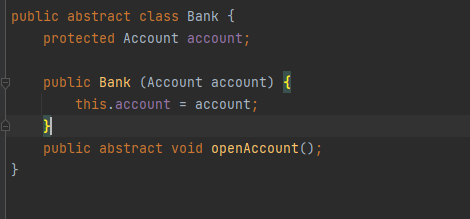
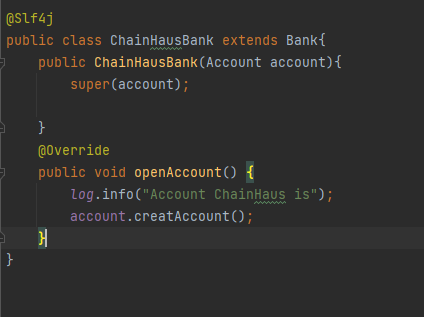
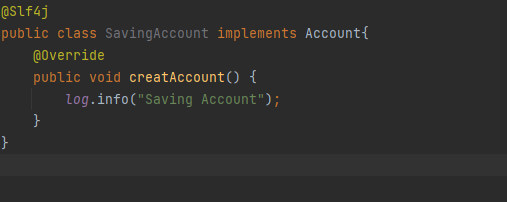


* Các thành phần cơ bản của adapter
  + Adaptee: định nghĩa interface không tương thích, cần được tích hợp vào.
  + 
  + Adapter: lớp tích hợp, giúp interface không tương thích tích hợp được với interface đang làm việc. Thực hiện việc chuyển đổi interface cho Adaptee và kết nối Adaptee với Client.
 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

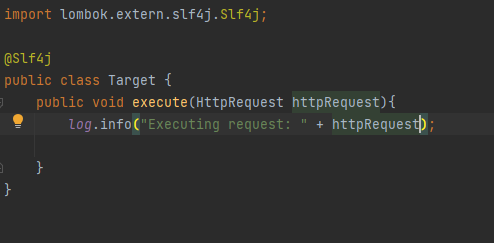
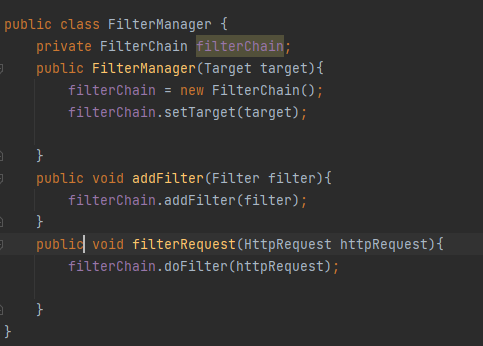
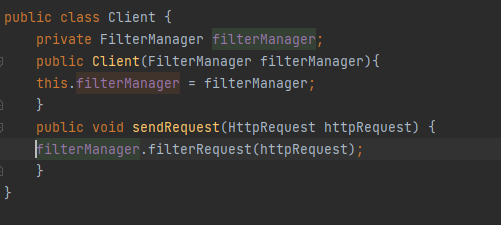


* Các loại adapter
  + Object Adapter – Composition : trong loại này, một lớp mới (Adapter) sẽ tham chiếu đến một (hoặc nhiều) đối tượng của lớp có sẵn với. Các bạn có thể hiểu Adaptee sẽ là một thuộc tích của Adapter.
  + Class Adapter – Inheritance: Ở dạng Adapter sẽ kế thừa từ cả Adapter và Target và thực hiện cài đặt những interface mà người dùng mong muốn (Target).

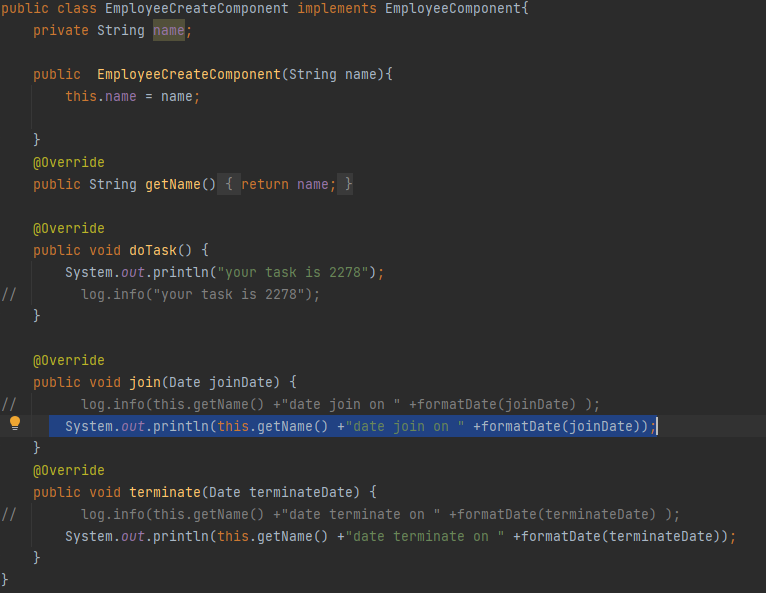
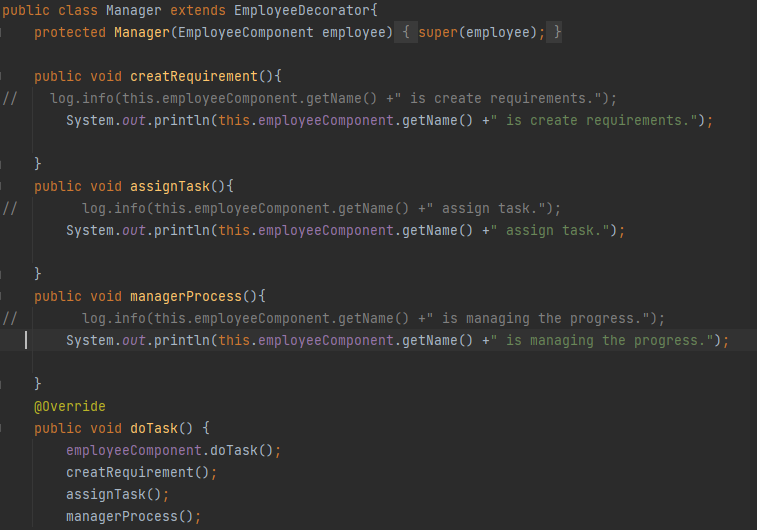
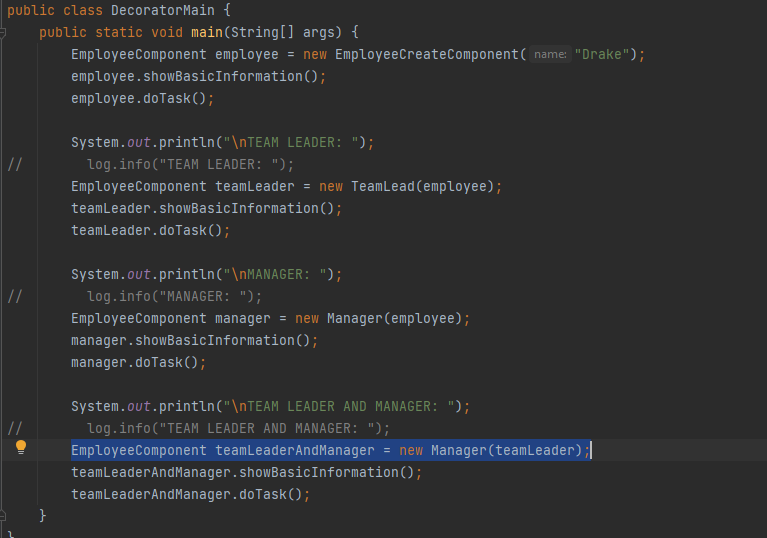
### Bridge

* **Khái niệm:** Bridge Pattern là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern). Ý tưởng của nó là tách tính trừu tượng (abstraction) ra khỏi tính hiện thực (implementation) của nó. Từ đó có thể dễ dàng chỉnh sửa hoặc thay thế mà không làm ảnh hưởng đến những nơi có sử dụng lớp ban đầu
* **Bridge Pattern** khá giống với mẫu Adapter Pattern ở chỗ là sẽ nhờ vào một lớp khác để thực hiện một số xử lý nào đó. Tuy nhiên, ý nghĩa và mục đích sử dụng của hai mẫu thiết kế này hoàn toàn khác nhau:
  + **Adapter Pattern** hay còn gọi là Wrapper pattern được dùng để biến đổi một class/ interface sang một dạng khác có thể sử dụng được. **Adapter Pattern** giúp các lớp không tương thích hoạt động cùng nhau mà bình thường là không thể.
  + **Bridge Pattern** được sử dụng được sử dụng để tách thành phần trừu tượng (abstraction) và thành phần thực thi (implementation) riêng biệt.
  + **Adapter Pattern** làm cho mọi thứ có thể hoạt động với nhau sau khi chúng đã được thiết kế (đã tồn tại). **Bridge Pattern** nên được thiết kế trước khi phát triển hệ thống để Abstraction và Implementation có thể thực hiện một cách độc lập.
* **Các thành phần của bridge**
  + Client: đại diện cho khách hàng sử dụng các chức năng thông qua Abstraction.
  + Abstraction : định ra một abstract interface quản lý việc tham chiếu đến đối tượng hiện thực cụ thể (Implementor).
  + 
  + Refined Abstraction (AbstractionImpl) : hiện thực (implement) các phương thức đã được định ra trong Abstraction bằng cách sử dụng một tham chiếu đến một đối tượng của Implementer.
  + 
  + Implementor : định ra các interface cho các lớp hiện thực. Thông thường nó là interface định ra các tác vụ nào đó của Abstraction.
  + 
  + ConcreteImplementor : hiện thực Implementor interface.

### Filter Pattern

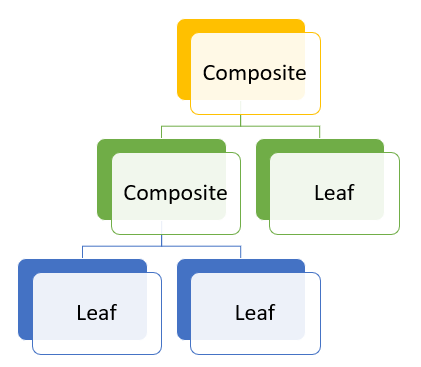
* **Khái niệm:** 
  + Intercepting filter pattern là một Java EE pattern, được sử dụng khi muốn thực hiện một vài xử lý trước (pre-processing) khi request được ứng dụng đích (target) xử lý hoặc sau (post-processing) khi response được trả về từ target.
  + Các Filter được định nghĩa và áp dụng trên yêu cầu (request) khi chuyển request đến ứng dụng đích thực tế (target). Các Filter có thể thực hiện xác thực (authentication), ủy quyền (authorization), nén dữ liệu (compressing), ghi nhật ký (logging) hoặc theo dõi yêu cầu (tracking) và sau đó chuyển yêu cầu đến các trình xử lý tương ứng.
  + Các Filter được thực thi một cách trong suốt, phía client và target không hề biết sự tồn tại của nó
* Thành phần của Filter Pattern
  + Filter : chịu trách nhiệm thực hiện một vài xử lý trước khi request được target xử lý hoặc sau khi response được trả về từ target.
  + 
  + Target : là một đối tượng xử lý lý chính, một trình xử lý yêu cầu.
  + 
  + Filter chain : chứa một chuỗi các Filter sẽ được thực hiện trên target theo thứ tự được xác định.
  + 
  + Filter manager : quản lý các Filter và Filter Chain.
  + 
  + Client : đối tượng gửi request đến target hoặc nhận response từ target.
  + 

### Decorator Pattern

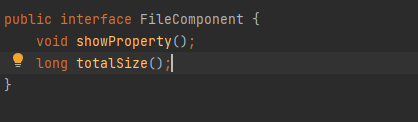
* **Khái niệm:** 
  + Nó cho phép người dùng thêm chức năng mới vào đối tượng hiện tại mà không muốn ảnh hưởng đến các đối tượng khác. Kiểu thiết kế này có cấu trúc hoạt động như một lớp bao bọc (wrap) cho lớp hiện có. Mỗi khi cần thêm tính năng mới, đối tượng hiện có được wrap trong một đối tượng mới (decorator class).
  + Decorator pattern sử dụng composition thay vì inheritance (thừa kế) để mở rộng đối tượng. Decorator pattern còn được gọi là Wrapper hay Smart Proxy.
* **Các thành phần Decorator Pattern**
  + Component: là một interface quy định các method chung cần phải có cho tất cả các thành phần tham gia vào mẫu này.
  + 
  + ConcreteComponent : là lớp hiện thực (implements) các phương thức của Component.
  + 
  + Decorator : là một abstract class dùng để duy trì một tham chiếu của đối tượng Component và đồng thời cài đặt các phương thức của Component interface.
  + 
  + ConcreteDecorator : là lớp hiện thực (implements) các phương thức của Decorator, nó cài đặt thêm các tính năng mới cho Component.
  + 
  + Client : đối tượng sử dụng Component.
  + 
* **Sử dụng Decorator Pattern**
  + Khi muốn thêm tính năng mới cho các đối tượng mà không ảnh hưởng đến các đối tượng này.
  + Khi không thể mở rộng một đối tượng bằng cách thừa kế (inheritance). Chẳng hạn, một class sử dụng từ khóa final, muốn mở rộng class này chỉ còn cách duy nhất là sử dụng decorator.
  + Trong một số nhiều trường hợp mà việc sử dụng kế thừa sẽ mất nhiều công sức trong việc viết code. Ví dụ trên là một trong những trường hợp như vậy.

### Composite Pattern

* **Khái niệm**: Composite Pattern là một sự tổng hợp những thành phần có quan hệ với nhau để tạo ra thành phần lớn hơn. Nó cho phép thực hiện các tương tác với tất cả đối tượng trong mẫu tương tự nhau.



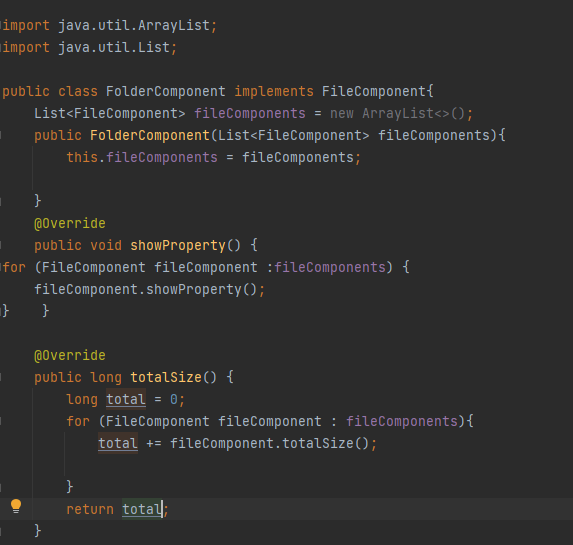
* **Các thành phần cơ bản của Composite Pattern:**
  + Base Component : là một interface hoặc abstract class quy định các method chung cần phải có cho tất cả các thành phần tham gia vào mẫu này.



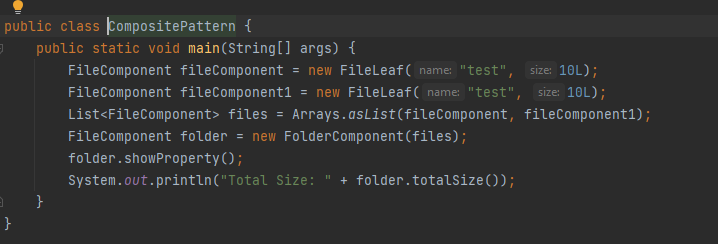
* + Leaf : là lớp hiện thực (implements) các phương thức của Component. Nó là các object không có con.



* + Composite : lưu trữ tập hợp các Leaf và cài đặt các phương thức của Base Component. Composite cài đặt các phương thức được định nghĩa trong interface Component bằng cách ủy nhiệm cho các thành phần con xử lý.



* + Client: sử dụng Base Component để làm việc với các đối tượng trong Composite.



* **Lợi ích của composite pattern:** Cung cấp cùng một cách sử dụng đối với từng đối tượng riêng lẻ hoặc nhóm các đối tượng với nhau.
* **Sử dụng Composite khi nào:**
  + Composite Pattern chỉ nên được áp dụng khi nhóm đối tượng phải hoạt động như một đối tượng duy nhất (theo cùng một cách).
  + Composite Pattern có thể được sử dụng để tạo ra một cấu trúc giống như cấu trúc cây.