**Spring Boot và Cấu trúc dự án**

1. **Tổng quan về Spring Boot**
2. **Java Spring Boot là gì?**

- Là một module của Spring framework.

- Cung cấp các tính năng RAD (Rapid Application Development - Phát triển ứng dụng nhanh chóng) cho phép tạo ứng dụng độc lập dựa trên Spring nhanh chóng.

- Mục đích: loại bỏ cấu hình phức tạp của Spring (không yêu cầu cấu hình XML, nâng cao hiệu suất làm việc).

1. **Ưu điểm**

- Có đầy đủ các tính năng của Spring framework.

- Đơn giản hóa cấu hình và xây dựng được các ứng dụng độc lập có khả năng chạy bằng java-jar nhờ các dependency starter.

- Dễ dàng deploy: các ứng dụng server được nhúng trực tiếp vào ứng dụng, tránh những khó khăn khi triển khai lên môi trường production và không cần thiết phải tải file WAR.

1. **Đặc tính cơ bản của Spring Boot**

- Cung cấp sẵn một lớp (Spring Application) chứa hàm main, là lớp mở đầu cho toàn bộ chương trình.

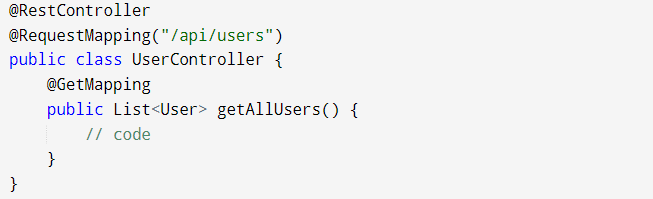
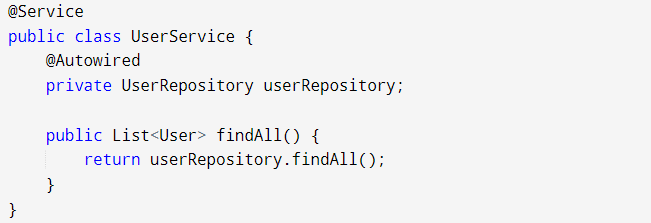
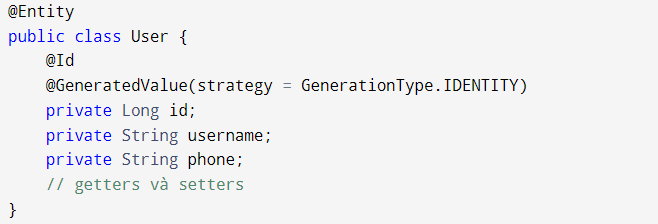
- Giảm bớt độ phức tạp trong việc cấu hình ứng dụng trong trường hợp sử dụng nhiều môi trường.

- Externalized Configuration cho phép có khả năng cấu hình được từ bên ngoài

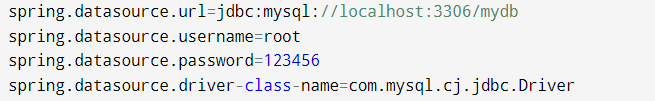
- Tất cả tính năng log nội bộ của Spring Boot đều sử dụng common log và được quản lý mặc định.

1. **Cấu trúc dự án Spring Boot**
2. **Cấu trúc dự án Spring Boot**
3. **Theo mô hình MVC**

- **src/main/java**: thư mục chứa mã nguồn Java của dự án, các thư mục con như:

* **Tên package của dự án**: thư mục gốc cho tất cả các class và packages khác của dự án, bao gồm controller, service,.... Đặt tên theo tiêu chuẩn domain ngược, ví dụ: com.mypackage, org.demo,...
* **controller**: chứa các lớp Controller, có nhiệm vụ xử lý các yêu cầu HTTP, ánh xạ chúng đến phương thức xử lý tương ứng, và điều hướng dữ liệu đến View hoặc trả về phản hồi. Ví dụ: 
* **service**: chứa các lớp Service, nơi xử lý các logic nghiệp vụ của ứng dụng. Các lớp trong tầng này thường gọi đến các lớp trong repository và cung cấp dữ liệu cho controller. Ví dụ: 
* **repository**: chứa các lớp Repository dùng để truy xuất dữ liệu từ CSDL.
* **model** hoặc **entity**: chứa các lớp Model hoặc Entity, biểu diễn các đối tượng dữ liệu (bản sao của các bảng trong CSDL). Ví dụ: 

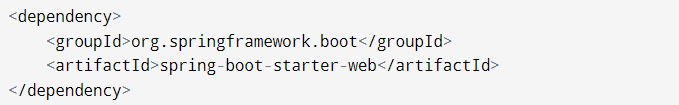
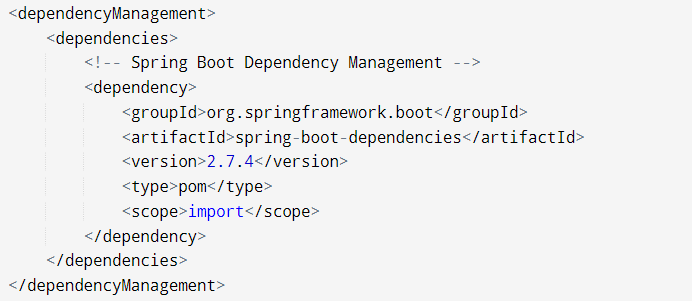
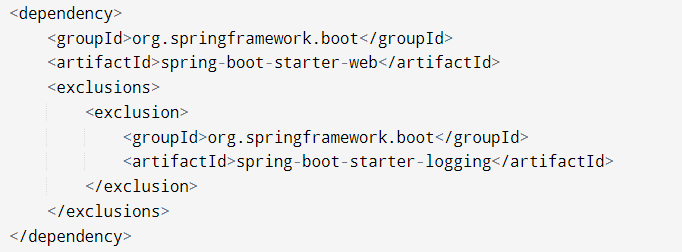
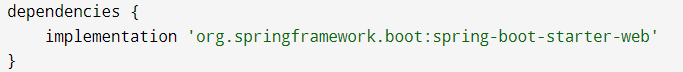
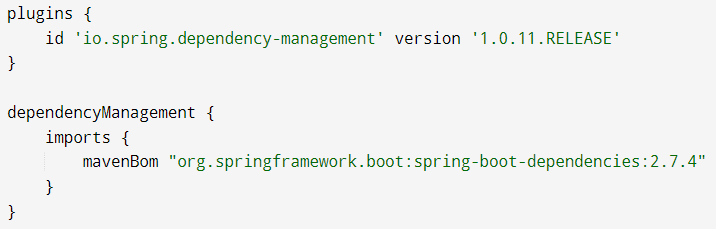
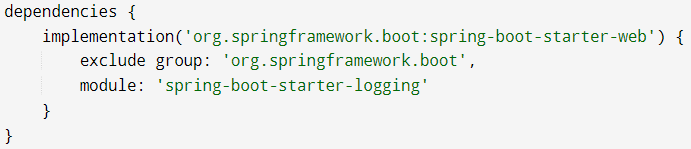
- **src/main/resources**: chứa các file cấu hình và các tài nguyên tĩnh khác:

* **application.properties** hoặc **application.yml**: File cấu hình chính của Spring Boot, nơi khai báo các thông số cấu hình như kết nối cơ sở dữ liệu, cổng server, logging... Ví dụ (application.properties**)**:
* Cấu hình server: server.port=8080
* Cấu hình kết nối CSDL: 
* Cấu hình JPA/Hibernate: 
* Cấu hình logging: 
* **static**: chứa các tài nguyên tĩnh như file CSS, JavaScript, hình ảnh,...
* **messages**: chứa các file cấu hình **messages.properties** cho việc xử lý đa ngôn ngữ.
* **data.sql** hoặc **schema.sql**: file SQL để khởi tạo hoặc nhập dữ liệu mẫu vào cơ sở dữ liệu khi ứng dụng khởi động.

- **src/test/java**: chứa các lớp phục vụ cho unit test, thường có cấu trúc tương tự src/main/java:

* **controller**: các lớp kiểm thử controller.
* **service**: các lớp kiểm thử logic nghiệp vụ.
* **repository**: các lớp kiểm thử truy xuất dữ liệu từ cơ sở dữ liệu.

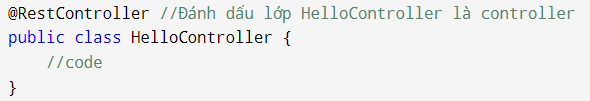
- **pom.xml** (với Maven) hoặc **build.gradle** (với Gradle): file cấu hình dependencies và build dự án.

* **pom.xml**:
* **Spring Boot Starter POMs**: tập hợp phụ thuộc được cấu hình sẵn cho các chức năng phổ biến. Ví dụ với ứng dụng web: 
* **Dependency Management Section**: Spring Boot sử dụng một phần dependencyManagement trong POM để quản lý phiên bản của các phụ thuộc, tránh việc phải xác định phiên bản cho từng phụ thuộc 
* **Exclusions**: loại bỏ các phụ thuộc không cần thiết từ các starter POMs bằng cách sử dụng phần tử <exclusions> 
* **build.gradle:**
* **Spring Boot Starter Dependencies**: Sử dụng các starter dependencies trong phần dependencies. Ví dụ: 
* **Dependency Management Plugin**: Gradle sử dụng plugin io.spring.dependency-management để hỗ trợ quản lý phụ thuộc tương tự như Maven 
* **Exclusions**: loại bỏ các phụ thuộc không cần thiết, ví dụ: 

1. **Theo tính năng**

- Mỗi tính năng tương ứng với package phù hợp. Ví dụ, **user package** chứa UserController, UserService,...

1. **Controller đơn giản xử lý request, response**
2. **Một số annotation**
3. **@RestController**

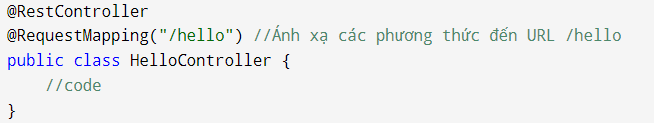
**-** Đánh dấu class là controller để xử lý các yêu cầu REST, ví dụ: 

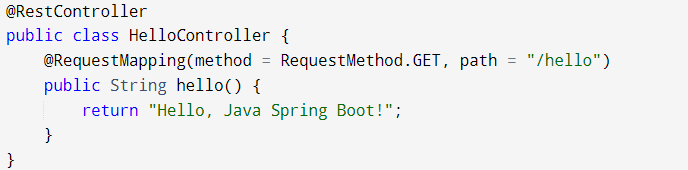
- Kết hợp **@Controller** và **@ResponseBody**: tự động chuyển đổi đối tượng được trả về thành JSON hoặc XML.

1. **@RequestMapping**

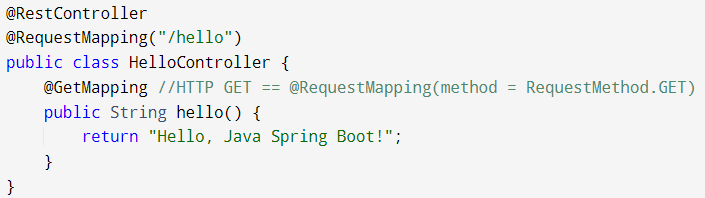
- Ánh xạ các yêu cầu HTTP tới các phương thức xử lý của controller

- Có thể áp dụng trên class hoặc phương thức

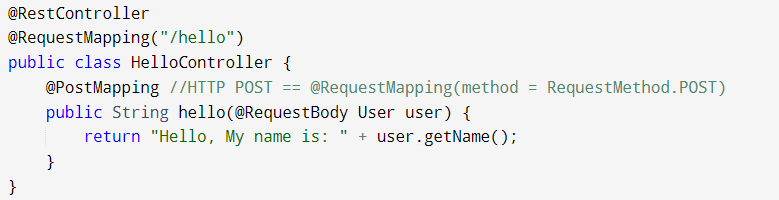
- Trên class: 

- Trên phương thức: 

1. **@GetMapping**

- Dành cho các HTTP GET, ví dụ: 

1. **@PostMapping**

- Dành cho HTTP POST, ví dụ: 

1. **@RequestBody**

- Chuyển đổi JSON từ body của request thành kiểu đối tượng (data binding)

1. **Cấu hình dự án với application.properties**
2. **application.properties là gì?**

- File application.properties được dùng để cấu hình ngữ cảnh ứng dụng (application context). Ngữ cảnh được tự động tạo ra khi ứng dụng Spring Boot được khởi động.

- Là nơi chứa cấu hình mặc định của ứng dụng.

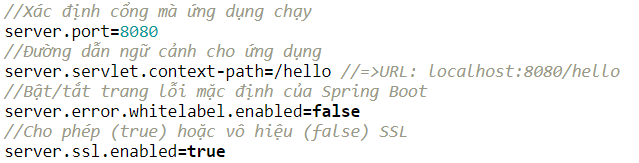
- Các cấu hình trong application.properties có độ ưu tiên thấp hơn khi ngữ cảnh ứng dụng được ghi đè ở nhiều nơi (tham số dòng lệnh, biến môi trường,...). Ví dụ:

* Tham số dòng lệnh: --server.port=8081
* Trong application.properties: server.port=8080

=> Ứng dụng chạy trên cổng 8081

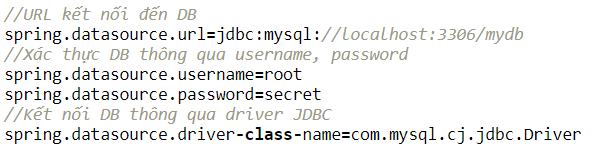
1. **Cấu hình ứng dụng**
2. **Cấu hình Server (Server configuration)**

- Cấu hình liên quan đến máy chủ nhúng mà Spring Boot sử dụng.

- Ví dụ: 

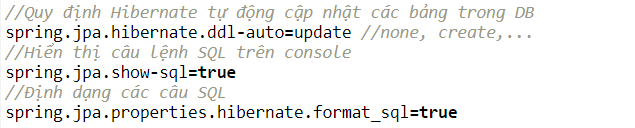
1. **Cấu hình cơ sở dữ liệu (Data source configuration)**

- Thiết lập thông tin kết nối cơ sở dữ liệu (CSDL) khi sử dụng JPA hoặc JDBC.

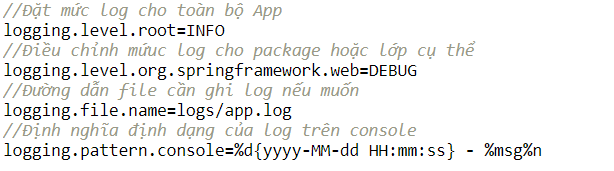
- Ví dụ: 

1. **Cấp hình JPA/Hibernate**

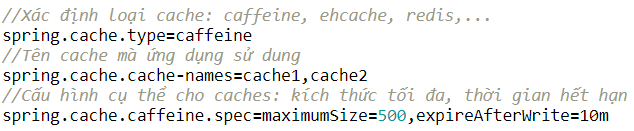
- Kiểm soát cách xử lý cơ sở dữ liệu và các đối tượng.

- Ví dụ: 

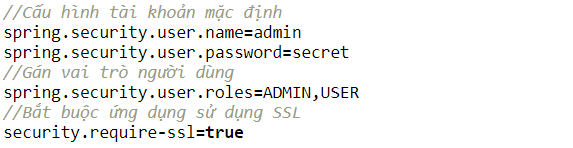
1. **Cấu hình logging (Logging configuration)**

- Điều chỉnh cách ghi log, ví dụ: 

1. **Cấu hình cache**

- Điều chỉnh cách ứng dụng xử lý bộ nhớ đệm, ví dụ: 

1. **Cấu hình bảo mật (Security configuration)**

- Thiết lập bảo mật ứng dụng, ví dụ: 

1. **Một số cấu hình khác**

- Cấu hình Mail: thiết lập việc gửi email thông qua server SMTP

- Cấu hình Thymeleaf (nếu sử dụng làm công cụ template engine)

- Cấu hình Web: điều chỉnh các thành phần như MVC, tài nguyên tĩnh, session

- Cấu hình File Upload

- Cấu hình CORS (Cross-Origin Resource Sharing): cấu hình cho phép chia sẻ tài nguyên giữa các nguồn khác nhau

1. **Cơ bản về MySQL**
2. **Select**

- Truy xuất dữ liệu từ cơ sở dữ liệu, kết quả trả về được lưu trong bảng kết quả, được gọi là tập kết quả.

- Cú pháp:

* Lấy dữ liệu theo các cột cụ thể: 
* Hoặc lấy tất cả các cột: 

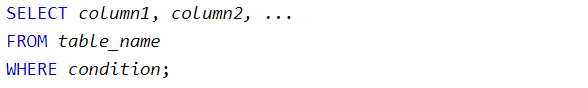
- Ví dụ: 

- Từ khóa **DISTINCT** để trả về các bản ghi là duy nhất, ví dụ: 

- Các từ khóa cho phép thực hiện tính toán trên dữ liệu trả về như: **MAX, MIN, COUNT, ...**

1. **Where**

-Từ khóa **WHERE** được sử dụng để lọc các bản ghi dữ liệu thõa mãn theo điều kiện xác định.

- Cú pháp: 

- Các toán tử: **=**, **<**, **<**, **>=**, **<=**, <>, BETWEEN, LIKE, IN,...

- Các toán tử logic: AND, OR, NOT

1. **Insert**

- Câu lệnh **INSERT INTO** dùng đề chèn thêm bản ghi mới vào bảng trong CSDL.

- Cú pháp: (chỉ rõ từng cột cần thêm dữ liệu) 

hoặc: (thêm vào tất cả các cột ứng với các giá trị truyền vào, số lượng giá trị truyền vào phải bằng số cột) 

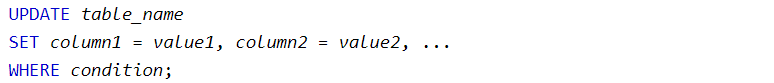
- Ví dụ: 

1. **Update**

- Cho phép sửa đổi bản ghi tồn tại trong bảng trong CSDL.

- Nên kết hợp với từ khóa **WHERE** để đảm bảo chỉ các bản ghi thõa mãn điều kiện mới được cập nhật.

- Nếu không có mệnh đề **WHERE**, tất cả các bản ghi sẽ được cập nhật.

- Cú pháp: 

1. **Delete**

- Xóa bản ghi tồn tại trong bảng trong CSDL.

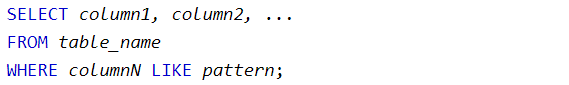
- Nên kết hợp mệnh đề **WHERE**, nếu không tất cả các bản ghi sẽ bị xóa

- Cú pháp: 

1. **Like**

- Kết hợp với WHERE để tìm kiếm một mẫu cụ thể.

- Các ký tự đại diện thường dùng như: **dấu phần trăm (%)** - đại diện cho bất kỳ chuỗi ký tự nào - và **dấu gạch dưới ( \_ )** - đại diện cho một ký tự duy nhất.

- Cú pháp: 

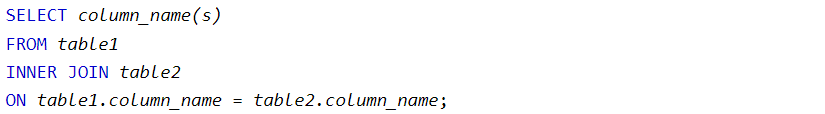
- Ví dụ, chọn tất cả khách hàng có tên bắt đầu bằng ‘a’: 

1. **Join**

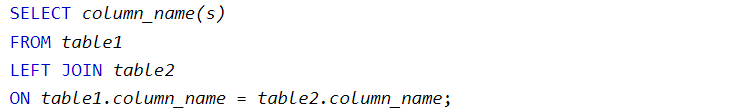
- Mệnh đề **JOIN** được sử dụng để kết hợp các hàng từ hai hoặc nhiều bảng, dựa trên một cột có liên quan giữa chúng.

- Có các loại như: **INNER JOIN**, **LEFT JOIN**, **RIGHT JOIN**, **CROSS JOIN**

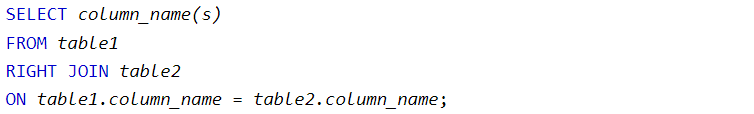
1. **Inner join**

- Chọn các bản ghi có giá trị trùng khớp trong cả hai bảng, cú pháp: 

1. **Left join**

- Trả về tất cả các bản ghi từ bảng bên trái (table1) và các bản ghi khớp (nếu có) từ bảng bên phải (table2), cú pháp: 

1. **Right join**

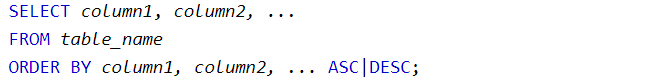
- Trả về tất cả các bản ghi từ bảng bên phải (table2) và các bản ghi khớp (nếu có) từ bảng bên trái (table1), cú pháp: 

1. **Cross join**

- Trả về tất cả bản ghi từ hai bảng (table1 và table2), cú pháp: 

1. **Order by**

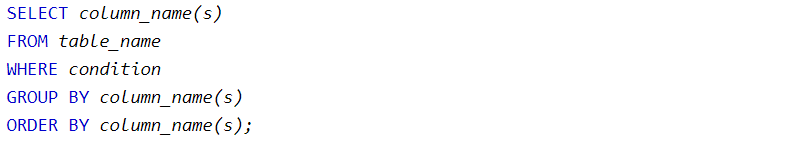
- Sử dụng để sắp xếp tập kết quả theo thứ tự tăng (**ASC**) hoặc giảm dần (**DESC**).

- Cú pháp: 

- Ví dụ, sắp xếp các bản ghi theo cột “Quốc gia”: 

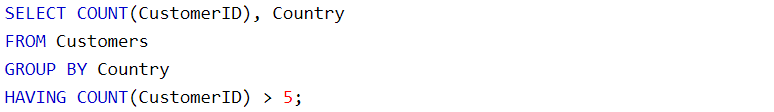
1. **Group by**

- Nhóm các hàng có cùng giá trị vào các hàng tóm tắt, thường được dùng với các hàm tổng hợp (**COUNT(), MAX(), MIN(), SUM(), AVG()**) để nhóm tập kết quả theo cột.

- Cú pháp: 

1. **Has**

- **HAS** được sử dụng với các hàm tổng hợp, cú pháp: 

- Ví dụ, liệt kê số lượng KH ở mỗi quốc gia thõa mãn điều kiện quốc gia đó có nhiều hơn 5 khách hàng: 

1. **Design pattern cơ bản trong Java**
2. **Các mẫu khởi tạo (Creational Pattern)**

- Cung cấp giải pháp để tạo ra các object và che dấu logic tạo ra nó, thay vì sử dụng từ khóa **new**.

- Gồm 5 mẫu: Factory Method, Abstract Factory, Builder, Prototype, Singleton.

1. **Singleton**

- Đảm bảo một lớp chỉ có một thể hiện, đồng thời cung cấp điểm truy cập toàn cục thông qua thể hiện này. => Kiểm soát quyền truy cập vào một số tài nguyên được chia sẻ, vi phạm nguyên tắc trách nhiệm đơn lẻ.

**- Đặc điểm thiết kế:**

* Hàm tạo mặc định là riêng tư
* Một phương thức tĩnh hoạt động như một hàm tạo. Một đối tượng được tạo ra từ nó và lưu trong một trường tĩnh. Mọi lệnh gọi đến hàm tạo này đều trả về đối tượng được lưu trong bộ nhớ đệm. => Tạo mới hoặc trả về đối tượng hiện có.

**- Khả năng áp dụng:**

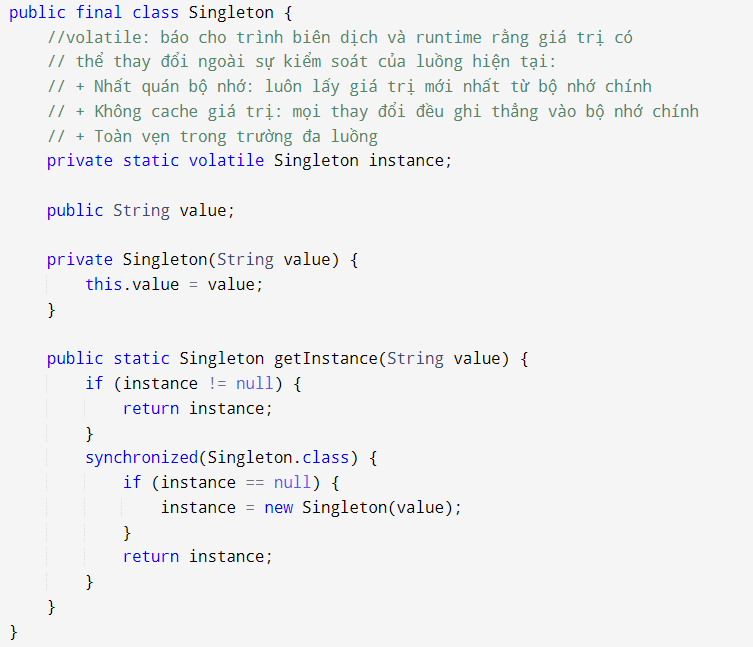
* Một lớp trong chương trình chỉ cần có một thể hiện duy nhất. Ví dụ, một đối tượng truy cập CSDL duy nhất.
* Cần kiểm soát chặt chẽ hơn các biến toàn cục.

**- Ưu điểm:**

* Một lớp chỉ có một thể hiện.
* Tạo ra điểm truy cập toàn cầu.
* Đối tượng singleton chỉ được khởi tạo khi được yêu cầu lần đầu tiên.

**- Nhược điểm:**

* Vi phạm nguyên tắc trách nhiệm đơn lẻ.
* Có thể che dấu thiết kế kém.
* Yêu cầu xử lý trong môi trường đa luồng để một luồng không tạo ra đối tượng đơn lẻ nhiều lần.
* Có thể khó kiểm thử đơn vị mã máy khách (vì nhiều khuôn khổ dựa trên tính kế thừa).

- An toàn luồng với lazy loading: Cần đồng bộ hóa các luồng trong lần đầu tạo đối tượng Singleton. 

1. **Abstract Factory**

- Cho phép tạo ra các họ đối tượng liên quan mà không cần chỉ định các lớp cụ thể.

**- Khả năng áp dụng:**

* Mã cần hoạt động với nhiều họ sản phẩm liên quan khác nhau, nhưng không muốn nó phụ thuộc vào các lớp cụ thể của các sản phẩm đó (chưa được biết trước hoặc cho phép mở rộng trong tương lai).
* Có một lớp với nhiều phương thức Factory làm mờ đi nhiệm vụ chính của lớp đó (Trong một thiết kế tốt, mỗi lớp chỉ chịu trách nhiệm cho một việc).

**- Ưu điểm:**

* Các sản phẩm nhận được từ nhà máy đều tương thích với nhau.
* Tránh sự kết hợp chặt chẽ giữa sản phẩm cụ thể và mã của khách hàng.
* Tuân thủ nguyên tắc trách nhiệm duy nhất: trích xuất mã tạo đối tượng ra khỏi lớp chính và đặt vào nơi chuyên chịu trách nhiệm khởi tạo.
* Tuân thủ nguyên tắc mở/đóng: thêm chức năng mới mà không cần thay đổi mã hiện có.

**- Nhược điểm:** mã có thể phức tạp hơn mức cần thiết.

1. **Builder**

- Xây dựng các đối tượng phức tạp từng bước 1. Cho phép tạo ra các loại và biểu diễn khác nhau của một đối tượng bằng cách sử dụng cùng một mã.

- Sử dụng mẫu này chỉ có ý nghĩa khi sản phẩm khá phức tạp và yêu cầu cấu hình mở rộng.

**- Khả năng áp dụng:**

* Sử dụng mẫu Builder để loại bỏ hàm tạo thu gọn (nạp chồng nhiều hàm tạo với tham số khác nhau).
* Khi muốn mã có thể tạo ra các biểu diễn khác nhau của một số sản phẩm (nhà đá và nhà gỗ).
* Cần xây dựng các đối tượng phức tạp.

**- Ưu điểm:**

* Xây dựng các đối tượng từng bước, trì hoãn các bước hoặc chạy các bước theo cách đệ quy.
* Có thể sử dụng lại cùng một mã xây dựng khi xây dựng nhiều dạng biểu diễn sản phẩm khác nhau.
* Tuân thủ nguyên tắc trách nhiệm đơn lẻ: tách biệt mã xây dựng phức tạp khỏi logic khác của sản phẩm.

**- Nhược điểm:** độ phức tạp tổng thể tăng lên vì yêu cầu tạo nhiều lớp mới.

1. **Factory Method**

- Cung cấp giao diện để tạo các đối tượng trong lớp cha, nhưng cho phép các lớp con thay đổi loại đối tượng sẽ được tạo ra.

- Mẫu này gợi ý thay thế các lệnh gọi xây dựng đối tượng trực tiếp bằng lệnh gọi đến một phương thức factory đặc biệt.

- Có thể ghi đè factory method trong lớp con và thay đổi lớp sản phẩm được tạo ra.

- Các lớp con chỉ có thể trả về các đối tượng khác nhau nếu chúng có chung một lớp cơ sở hoặc giao diện. Phương thức nhà máy trong lớp cơ sở phải có kiểu trả về được khai báo là giao diện này.

- Factory method không phải lúc nào cũng tạo ra các thể hiện mới. Nó cũng có thể trả về các đối tượng hiện có từ bộ nhớ đệm, nhóm đối tượng hoặc nguồn khác.

- Khả năng áp dụng:

* Sử dụng khi không biết trước chính xác loại và sự phụ thuộc của các đối tượng mà mã của bạn phải làm việc.
* Khi bạn muốn cung cấp cho người dùng thư viện hoặc khung của mình một cách để mở rộng các thành phần bên trong.
* Khi bạn muốn tiết kiệm tài nguyên hệ thống bằng cách sử dụng lại các đối tượng hiện có thay vì phải xây dựng lại chúng mỗi lần.

- Ưu điểm:

* Trách được sự kết hợp chặt chẽ giữa người sáng tạo và sản phẩm cụ thể.
* Nguyên tắc trách nhiệm đơn lẻ.
* Nguyên tắc mở/đóng .

- Nhược điểm: hệ thống trở nên phức tạp hơn.

1. **Các mẫu cấu trúc (Structural Pattern)**

- Dùng để thiết lập, định nghĩa quan hệ giữa các đối tượng.

- Gồm: Adapter, Bridge, Composite, Decorator, Facade, Flyweight và Proxy.

1. **Adapter**

- Mẫu thiết kế cho phép các đối tượng có giao diện không tương thích có thể cộng tác với nhau (chuyển đổi interface của một lớp thành interface khác mà client mong đợi).

- Khả năng áp dụng:

* Khi muốn sử dụng một số lớp hiện có nhưng giao diện của lớp đó không tương thích với phần còn lại của mã.
* Khi bạn muốn tái sử dụng một số lớp con hiện có thiếu một số chức năng chung không thể thêm vào lớp cha.

- Ưu điểm:

* Nguyên tắc trách nhiệm đơn lẻ.
* Nguyên tắc mở/đóng.

- Nhược điểm:

* Độ phức tạp chung của mã tăng lên.

1. **Decorator**

- Mở rộng chức năng của đối tượng một cách linh hoạt mà không cần thay đổi cấu trúc của nó. Bằng cách đặt các đối tượng này bên trong các đối tượng bao bọc đặc biệt chứa các hành vi đó.

- Khả năng áp dụng:

* Khi cần có khả năng gán các hành vi bổ sung cho các đối tượng khi chạy mà không làm hỏng mã sử dụng các đối tượng này.
* Khi việc mở rộng hành vi của đối tượng bằng cách kế thừa trở nên khó khăn hoặc không thể thực hiện được.

- Ưu điểm:

* Mở rộng hành vi của một đối tượng mà không cần tạo lớp con mới.
* Thêm hoặc xóa trách nhiệm khỏi một đối tượng khi chạy.
* Có thể kết hợp nhiều hành vi bằng cách gói một đối tượng vào nhiều trình trang trí.
* Nguyên tắc trách nhiệm đơn lẻ.

- Nhược điểm:

* Khó để xóa lớp bao cụ thể khỏi ngăn xếp lớp bao.
* Khó để triển khai một trình trang trí theo cách mà hành vi của nó không phụ thuộc vào thứ tự trong ngăn xếp trình trang trí.
* Mã cấu hình ban đầu của các lớp không đẹp.

1. **Các mẫu tương tác (Behavioral Pattern)**

- Dùng trong thực hiện các hành vi của đối tượng, sự giao tiếp giữa các object với nhau.

- Gồm: Interpreter, Template Method, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy và Visitor.

1. **Observer**

- Xác định cơ chế để đăng ký thông báo cho nhiều đối tượng về bất kỳ sự kiện nào xảy ra với đối tượng mà chúng đang quan sát.

- Các đối tượng riêng lẻ có thể đăng ký hoặc hủy đăng ký luồng sự kiện đến từ đối tượng được quan sát.

- Khả năng áp dụng:

* Khi việc thay đổi trạng thái của một đối tượng có thể yêu cầu phải thay đổi các đối tượng khác và tập hợp các đối tượng thực tế không được biết trước hoặc thay đổi một cách linh hoạt.
* Khi một số đối tượng trong ứng dụng của bạn phải quan sát những đối tượng khác, nhưng chỉ trong thời gian giới hạn hoặc trong những trường hợp cụ thể.

- Ưu điểm:

* Tuân thủ nguyên tắc mở/đóng.
* Có thể thiết lập mối quan hệ giữa các đối tượng khi chạy.

- Nhược điểm: Người đăng ký được thông báo theo thứ tự ngẫu nhiên.

1. **Strategy**

- Mẫu thiết kế cho phép xác định một họ thuật toán, đưa từng thuật toán vào một lớp riêng biệt và làm cho các đối tượng có thể hoán đổi cho nhau.

- Khả năng áp dung:

* Khi muốn sử dụng các biến thể khác nhau của một thuật toán trong một đối tượng và có thể chuyển đổi từ thuật toán này sang thuật toán khác trong thời gian chạy.
* Khi có nhiều lớp tương tự nhau chỉ khác nhau ở cách thực hiện một số hành vi.
* Sử dụng mẫu để tách biệt logic của một lớp khỏi các chi tiết triển khai của thuật toán có thể không quan trọng trong bối cảnh của logic đó.
* Khi một lớp có câu lệnh điều kiện lớn chuyển đổi giữa các biến thể khác nhau của cùng một thuật toán.

- Ưu điểm:

* Hoán đổi các thuật toán được sử dụng bên trong một đối tượng khi chạy.
* Tách biệt các chi tiết triển khai của thuật toán khỏi mã sử dụng thuật toán đó.
* Có thể thay thế kế thừa bằng thành phần.
* Nguyên tắc mở/đóng.

- Nhược điểm:

* Mô hình trở nên phức tạp nếu có ít thuật toán và hiếm khi thay đổi.
* Khách hàng cần nhận thức được sự khác biệt giữa các chiến lược để có thể lựa chọn phù hợp.

1. **Dependency Injection và Inversion of Control**

- Inversion of Control (IoC) là hướng đi, còn Dependency Injection (DI) là một hiện thực cụ thể.

1. **Inversion of Control (IoC)**

- Là một nguyên tắc thiết kế mà các đối tượng không kiểm soát việc tạo hoặc quản lý các phụ thuộc của chúng, thay vào đó, chúng nhận các phụ thuộc từ bên ngoài.

- IoC giúp cải thiện khả năng mở rộng và bảo trì của mã nguồn bằng cách tách biệt việc khởi tạo và logic nghiệp vụ. Trong IoC, trách nhiệm cung cấp các phụ thuộc được giao cho một IoC container.

- Các loại IoC phổ biến:

* **Dependency Injection (DI)**: Là cách phổ biến để thực hiện IoC. Các phụ thuộc của đối tượng được tiêm vào đối tượng thông qua phương thức khởi tạo, phương thức cập nhật hoặc gán trực tiếp vào thuộc tính tĩnh của đối tượng.
* **Event-based IoC**: Đối tượng đăng ký các sự kiện với một event listener. Khi một sự kiện xảy ra, container sẽ gọi các phương thức tương ứng.

- IoC container quản lý vòng đời của các đối tượng và tự động tiêm phụ thuộc. Các lớp được khai báo với annotation như @Autowired, @Component, @Service, @Repository, @Controller,... các phụ thuộc cần thiết sẽ tự động được tiêm vào khi khởi tạo lớp đó.

**- Ưu điểm:**

* Các phụ thuộc được liên kết động khi chạy, tăng tính kinh hoạt.
* Quản lý và cập nhật mã dễ dàng.
* Đơn giản hóa quá trình thử nghiệm, sử dụng các đối tượng giả thay thế các triển khai thực tế.
* Thúc đẩy sự kết hợp lỏng lẻo giữa các thành phần, tăng tính module.

**- Nhược điểm:**

* Dự án phức tạp hơn.
* Khó nắm bắt hoạt động của hệ thống do IoC che khuất luồng thực thi.
* Chi phí hiệu suất tiềm ẩn do một số khuôn khổ sử dụng phản xạ để khởi tạo động.

1. **Dependency Injection (DI)**

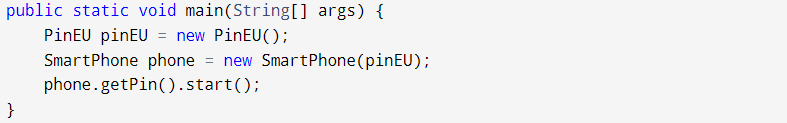
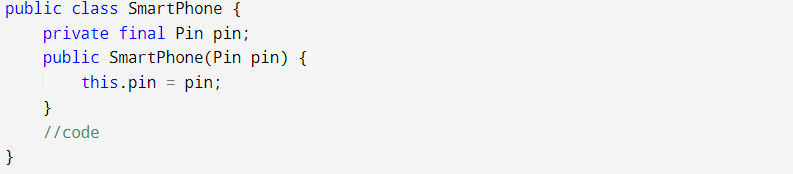
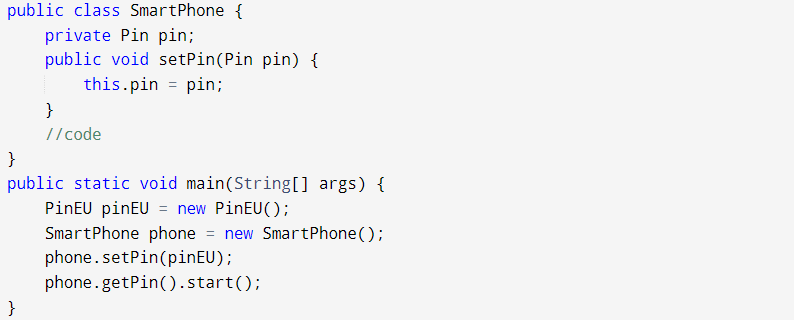
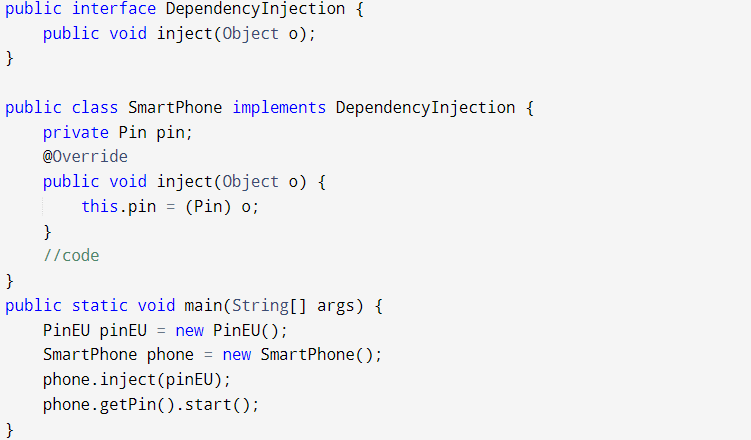
- DI là một cơ chế giúp đạt được **Inversion of Control**

- Là một kỹ thuật giúp quản lý các phụ thuộc giữa các thành phần hệ thống, làm cho mã dễ hiểu, trực quan hơn, dễ bảo trì và nâng cấp mã.

- Dependency xảy ra khi lớp A sử dụng một số chức năng của lớp B.

- Dependency Injection (DI) là một kỹ thuật trong lập trình phần mềm cho phép "tiêm" các phụ thuộc vào một thành phần thay vì định nghĩa chúng bên trong thành phần đó. Dependency là một loại quan hệ giữa 2 thành phần mà trong đó một thành phần hoạt động độc lập và thành phần còn lại phụ thuộc bởi thành phần kia.

- Ba loại DI gồm:

* **Constructor injection**: Các dependency được cung cấp thông qua hàm khởi tạo. Các phụ thuộc được lưu trữ bởi đối tượng. 
* **Setter injection**: Các dependency sẽ được truyền vào thông qua các hàm setter. Có tính linh hoạt hơn so với constructor injection (có thể thiết lập hoặc thay đổi các phụ thuộc sau khi đối tượng đã được tạo. 
* **Interface injection**: Ít phổ biến hơn. Cần có một giao diện được dùng để định nghĩa một phương thức chung để tiêm phụ thuộc. Các lớp cần phụ thuộc phải implement giao diện này và triển khai phương thức tiêm phụ thuộc. 

**- Ưu điểm**:

* Linh hoạt và giảm ràng buộc: nhận phụ thuộc từ bên ngoài thay vì phải tạo ra các phụ thuộc.
* Tăng khả năng kiểm tra.
* Tăng tính mô-đun, dễ bảo trì.
* Khả năng mở rộng và tái sử dụng.

**- Nhược điểm**:

* Ràng buộc về kiểu dữ liệu: cần phải cấu hình lại DI nếu các phụ thuộc thay đổi kiểu dữ liệu.
* Quản lý và cấu hình phức tạp.
* Tăng độ phức tạp nếu sử dụng sai.
* Nguy cơ tiềm ẩn vòng lặp phụ thuộc: người dùng có thể tạo ra vòng lặp phụ thuộc giữa các thành phần, gây ra vấn đề về thứ tự khởi tạo và tăng độ phức tạp của hệ thống.