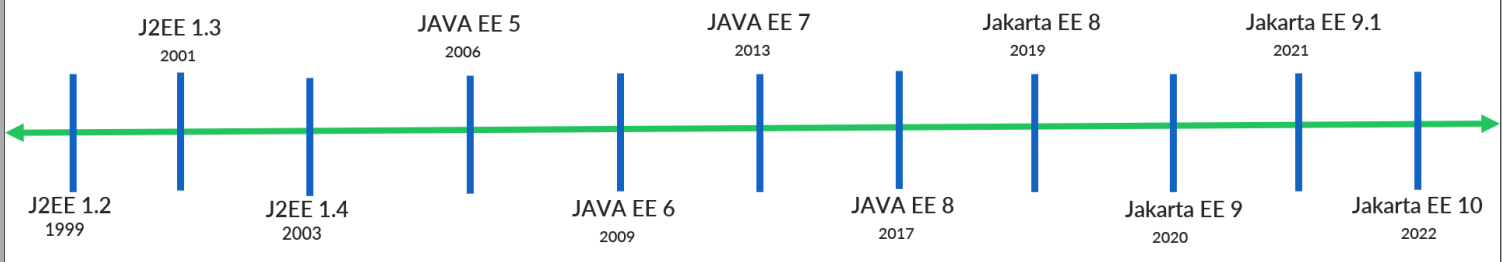
# **Section 1: Introduction to Spring Framework**

## **What is Spring ?**

* Spring Framework (viết tắt là Spring) là một framework phát triển, mạnh mẽ và rất linh hoạt, tập trung vào việc xây dựng các ứng dụng web bằng Java.
* Spring giúp lập trình Java nhanh hơn, dễ dàng hơn và an toàn hơn cho mọi người. Nó tập trung vào tốc độ, sự đơn giản và năng suất đã khiến nó trở thành khung công tác Java phổ biến nhất thế giới.
* Cho dù bạn đang xây dựng các microservice an toàn, tương tác, dựa trên đám mây cho web hay các luồng dữ liệu truyền trực tuyến phức tạp cho doanh nghiệp, thì Spring đều có các công cụ để trợ giúp.
* Ra đời như một giải pháp thay thế cho EJB vào đầu những năm 2000, Spring framework nhanh chóng vượt qua đối thủ nhờ sự đơn giản, nhiều tính năng và tích hợp thư viện của bên thứ ba.
* Nó phổ biến đến mức đối thủ cạnh tranh chính của nó đã bỏ cuộc khi Oracle dừng quá trình phát triển của Java EE 8 và cộng đồng đã tiếp nhận việc bảo trì nó thông qua Jakarta EE.
* Lý do chính của sự thành công của Spring framework là nó thường xuyên giới thiệu các tính năng/dự án dựa trên xu hướng thị trường mới nhất, nhu cầu của cộng đồng Dev. Ví dụ: SpringBoot
* Spring là mã nguồn mở. Nó có một cộng đồng lớn và tích cực cung cấp phản hồi liên tục dựa trên nhiều trường hợp sử dụng trong thế giới thực.

## **Jakarta EE Vs Spring**



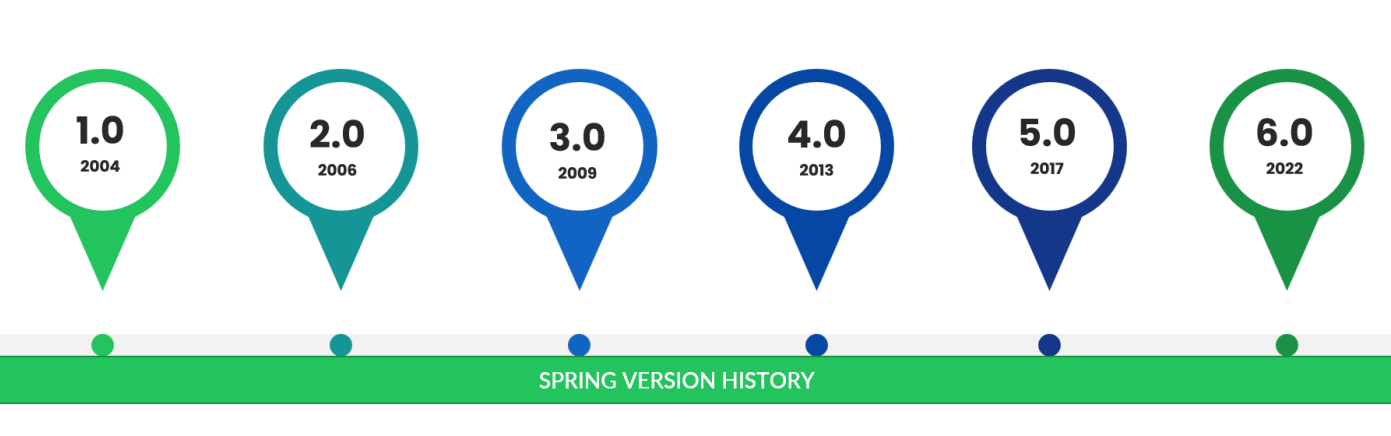
Java/Jakarta Enterprise Edition (EE) chứa Servlet, JSP, EJB, JMS, RMI, JPA, JSF, JAXB, JAX-WS, Web Sockets, v.v.

Các thành phần của Java/Jakarta Enterprise Edition (EE) như EJB, Servlet về bản chất là phức tạp do đó mọi người đều điều chỉnh Spring framework để phát triển ứng dụng web.

Java EE đã bỏ cuộc chạy đua với Spring framework, khi Oracle dừng quá trình phát triển của Java EE 8 và cộng đồng đã tiếp nhận việc bảo trì nó thông qua Jakarta EE.

Vì Oracle sở hữu nhãn hiệu cho tên "Java“, Java EE được đổi tên thành Jakarta EE. Tất cả các gói đều được cập nhật bằng javax." đến jakarta.” thay đổi namespace.

## **SPRING RELEASE TIMELINE**



Phiên bản đầu tiên của Spring được viết bởi Rod Johnson, người đã phát hành framework cùng với việc xuất bản cuốn sách Expert One-on-One J2EE Design and Development vào tháng 10 năm 2002

Spring ra đời vào năm 2003 như một phản ứng đối với sự phức tạp của các đặc tả J2EE ban đầu. Trong khi một số người coi Java EE và Spring là đối thủ cạnh tranh, thì trên thực tế, Spring là phần bổ sung cho Java EE. Mô hình lập trình Spring không bao gồm đặc tả nền tảng Java EE; thay vào đó, nó tích hợp với các thông số kỹ thuật riêng lẻ được lựa chọn cẩn thận từ EE

Spring tiếp tục đổi mới và phát triển. Ngoài Spring Framework, còn có các dự án khác, chẳng hạn như Spring Boot, Spring Security, Spring Data, Spring Cloud, Spring Batch, v.v.

## **Spring Core**

* Spring Core là trái tim của toàn bộ Spring. Nó chứa một số lớp, nguyên tắc và cơ chế khung cơ sở.
* Toàn bộ Spring Framework và các dự án khác của Spring được phát triển trên Spring Core.
* Spring Core chứa các thành phần quan trọng sau:
* loC (Inversion of Control)
* DI (Dependency Injection)
* Beans
* Context
* SpEL (Spring Expression Language)
* loC Container

## **INVERSION OF CONTROL & DEPENDENCY INJECTION**

* Inversion of Control (IoC) là Nguyên tắc thiết kế phần mềm, không phụ thuộc vào ngôn ngữ, không thực sự tạo đối tượng nhưng mô tả cách đối tượng được tạo.
* loC là nguyên tắc, trong đó luồng điều khiển của chương trình được đảo ngược: thay vì lập trình viên kiểm soát luồng chương trình, khung hoặc dịch vụ sẽ kiểm soát luồng chương trình.
* Dependency Injection là mô hình mà Inversion of Control đạt được.
* Thông qua Dependency Injection, trách nhiệm tạo các đối tượng được chuyển từ ứng dụng sang bộ chứa Spring loC. Nó làm giảm sự ghép nối giữa nhiều đối tượng vì nó được khung tự động đưa vào.

## **ADVANTAGES OF loC & DI**

Những lợi ích của Inversion of Control (IoC) và Dependency Injection (DI):

1. Giảm sự kết nối chặt chẽ: IoC và DI giúp giảm sự kết nối chặt chẽ giữa các thành phần. Với DI, các phụ thuộc của một lớp được tiêm vào từ bên ngoài thay vì được tạo hoặc quản lý bên trong. Điều này giảm sự kết nối trực tiếp giữa các thành phần và cho phép dễ dàng bảo trì, kiểm thử và thay đổi mã nguồn.
2. Mã nguồn có tính mô-đun và có thể tái sử dụng: IoC và DI thúc đẩy tính mô-đun và khả năng tái sử dụng mã nguồn. Bằng cách tách biệt việc tạo và quản lý phụ thuộc khỏi lớp chính, các thành phần có thể dễ dàng được sử dụng lại trong các ngữ cảnh hoặc dự án khác nhau. Điều này thúc đẩy việc tổ chức mã nguồn và tạo kiến trúc mô-đun và linh hoạt hơn.
3. Khả năng kiểm thử: DI cải thiện khả năng kiểm thử. Bằng cách tiêm các phụ thuộc, bạn có thể dễ dàng cung cấp các phiên bản giả định hoặc mô phỏng của các phụ thuộc trong quá trình kiểm thử. Điều này cho phép kiểm thử đơn vị các thành phần riêng lẻ mà không cần thiết lập phức tạp hoặc phụ thuộc vào các phụ thuộc thực tế. Việc kiểm thử trở nên đơn giản và tập trung vào các đơn vị mã nguồn cụ thể.
4. Tính linh hoạt và mở rộng: IoC và DI làm cho mã nguồn của bạn linh hoạt và dễ mở rộng. Bằng cách tách biệt các thành phần, bạn có thể dễ dàng thay thế các phụ thuộc bằng các phiên bản thay thế. Điều này cho phép dễ dàng tùy chỉnh, cấu hình và mở rộng ứng dụng. Nó cho phép tích hợp các tính năng hoặc công nghệ mới mà không cần sửa đổi đáng kể mã nguồn hiện có.
5. Khuyến khích các nguyên tắc tốt nhất: IoC và DI khuyến khích việc sử dụng các nguyên tắc tốt nhất trong phát triển phần mềm, chẳng hạn như Nguyên tắc Đơn trách nhiệm (SRP) và Nguyên tắc Đảo ngược phụ thuộc (DIP). Những nguyên tắc này thúc đẩy việc phân tách quyền trách nhiệm, trừu tượng hóa phụ thuộc và thiết kế mô-đun, dẫn đến mã nguồn sạch hơn, dễ bảo trì và có khả năng mở rộng hơn.
6. Tính dễ đọc và dễ bảo trì: Bằng cách rõ ràng khai báo các phụ thuộc và tách biệt việc quản lý chúng, DI cải thiện tính dễ đọc và dễ bảo trì của mã nguồn. Nó cung cấp thông tin rõ ràng và rõ ràng về các phụ thuộc cần thiết của một lớp, giúp các nhà phát triển dễ hiểu và chỉnh sửa mã nguồn. Điều này có thể dẫn đến sự hợp tác tốt hơn giữa các thành viên trong nhóm và việc bảo trì mã nguồn mượt mà hơn.

## **SPRING BEANS, CONTEXT, SpEL**

* Bất kỳ lớp Java bình thường nào được khởi tạo, lắp ráp và quản lý bởi bộ chứa Spring loC đều được gọi là Spring Bean.
* Các bean được tạo bằng siêu dữ liệu cấu hình mà bạn cung cấp cho container ở dạng Annotation và cấu hình XML.
* Spring loC Container quản lý vòng đời của phạm vi Spring Bean và đưa bất kỳ phụ thuộc cần thiết nào vào bean.
* Contexr giống như một vị trí bộ nhớ của ứng dụng của bạn, trong đó chúng tôi thêm tất cả các phiên bản đối tượng mà chúng tôi muốn framework quản lý. Theo mặc định, Spring không biết bất kỳ đối tượng nào bạn xác định trong ứng dụng của mình. Để cho phép Spring xem các đối tượng của bạn, bạn cần thêm chúng vào context.
* SpEL cung cấp một ngôn ngữ biểu thức mạnh mẽ để truy vấn và thao tác một biểu đồ đối tượng trong thời gian chạy như cài đặt và nhận các giá trị thuộc tính, gán thuộc tính, gọi phương thức BEAN, v.v.

## **SPRING IoC CONTAINER**

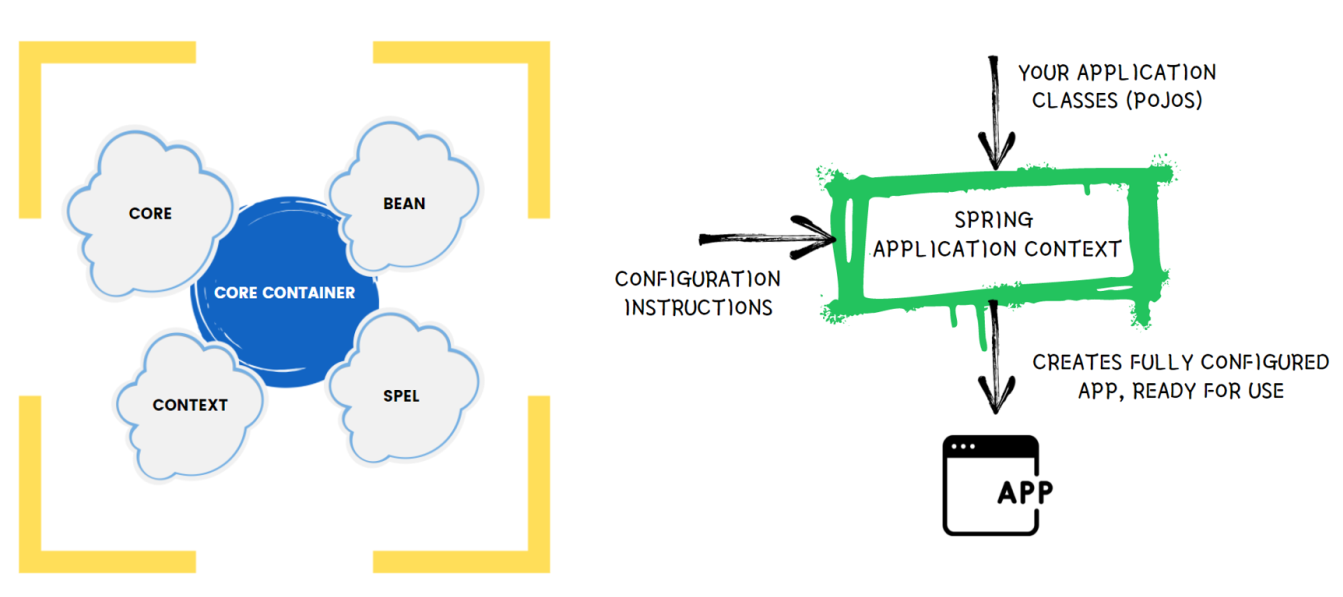
Spring loC Container

* IoC container chịu trách nhiệm
* Để khởi tạo lớp ứng dụng
* Để cấu hình đối tượng
* Để tập hợp các phụ thuộc giữa các đối tượng

Có hai loại bộ chứa IoC. Họ đang:

* org.springframework.beans.factory.BeanFactory
* org.springframework.context.ApplicationContext

The Spring container sử dụng dependency injection (DI) để quản lý các thành phần/đối tượng tạo nên một ứng dụng.



# **Section 2: Creating Beans inside Spring Context**

## **Maven**

Apache Maven là một công cụ quản lý dự án và xây dựng phần mềm tự động trong môi trường phát triển Java. Nó giúp quản lý các phụ thuộc của dự án, quản lý vòng đời phát triển và xây dựng ứng dụng.

Maven sử dụng mô hình quản lý dự án dựa trên khái niệm Project Object Model (POM), trong đó POM là một tệp cấu hình XML mô tả cấu trúc và các yêu cầu của dự án. POM mô tả các phụ thuộc của dự án, plugin sử dụng trong quá trình xây dựng, các thiết lập và các nhiệm vụ cần thiết để xây dựng, kiểm thử và triển khai ứng dụng.

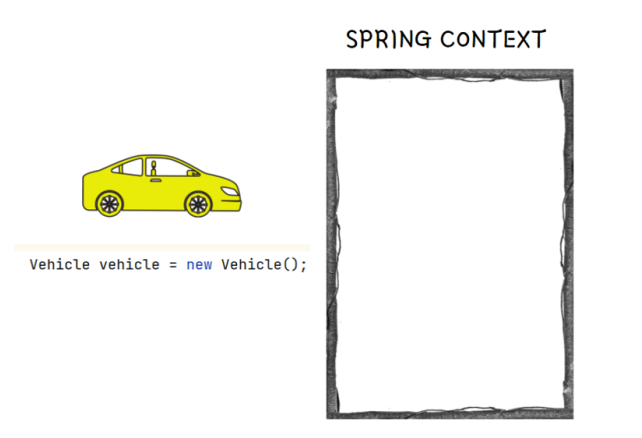
Một số tính năng chính của Maven bao gồm:

* Quản lý phụ thuộc: Maven tự động quản lý các phụ thuộc của dự án. Nó có thể tải về, cài đặt và định cấu hình các thư viện và công cụ cần thiết cho dự án, từ kho lưu trữ công cộng hoặc các kho lưu trữ nội bộ.
* Quản lý vòng đời: Maven cung cấp một chuỗi các nhiệm vụ xây dựng tiêu chuẩn, từ biên dịch mã nguồn, đóng gói tài nguyên, kiểm thử, tạo tài liệu đến triển khai ứng dụng. Nó tự động quản lý quá trình xây dựng và tạo ra sản phẩm cuối cùng.
* Plugin mở rộng: Maven hỗ trợ một loạt các plugin mở rộng cho các nhiệm vụ xây dựng khác nhau. Các plugin có thể được sử dụng để thực hiện các công việc tùy chỉnh, như đóng gói ứng dụng, chạy kiểm thử, tạo tài liệu, quản lý phiên bản, v.v.
* Tích hợp dễ dàng: Maven tích hợp tốt với các công cụ phát triển phổ biến khác như Eclipse, IntelliJ IDEA và NetBeans. Nó cung cấp các plugin và công cụ hỗ trợ để tích hợp quy trình phát triển Maven vào môi trường phát triển được sử dụng.

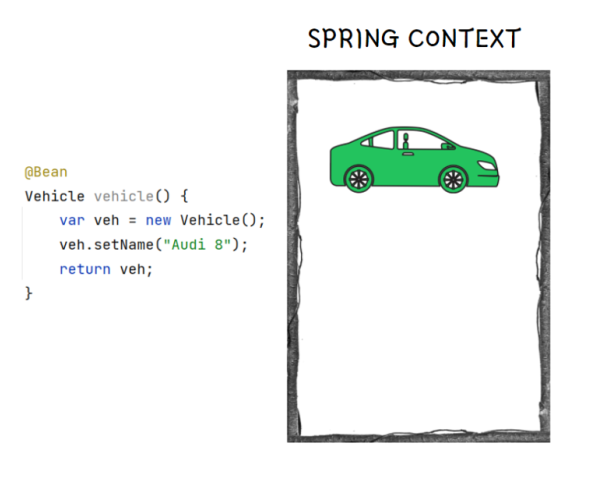
Maven là một công cụ mạnh mẽ giúp tăng hiệu suất và sự nhất quán trong quá trình phát triển phần mềm Java. Nó đơn giản hóa việc quản lý phụ thuộc, xây dựng và triển khai ứng dụng, đồng thời giúp tạo ra các dự án có cấu trúc chuẩn mà các nhà phát triển có thể dễ dàng làm việc và chia sẻ.

## **Adding new beans to spring context**

- Khi chúng ta tạo trực tiếp một đối tượng java với toán tử new() như hình bên dưới, thì Spring Context/Spring IoC Container của bạn sẽ không có bất kỳ manh mối nào về đối tượng.



- Annotation @Bean cho Spring biết rằng nó cần gọi phương thức này khi nó khởi tạo context và thêm object/value được trả về vào Spring context/Spring IoC container.



Ví dụ:  
@Configuration  
public class ProjectConfig {  
   
 @Bean  
 Vehicle vehicle() {  
 var veh = new Vehicle();  
 veh.setName("Audi 8");  
 return veh;  
 }

}

public class Example1 {  
  
 public static void main(String[] args) {  
  
 Vehicle vehicle = new Vehicle();  
 vehicle.setName("Honda City");  
 System.*out*.println("Vehicle name from non-spring context is: " + vehicle.getName());  
  
 var context = new AnnotationConfigApplicationContext(ProjectConfig.class);  
  
 Vehicle veh = context.getBean(Vehicle.class);  
 System.*out*.println("Vehicle name from Spring Context is: " + veh.getName());  
  
 }  
}

=> Kết quả trả về:

Vehicle name from non-spring context is: null

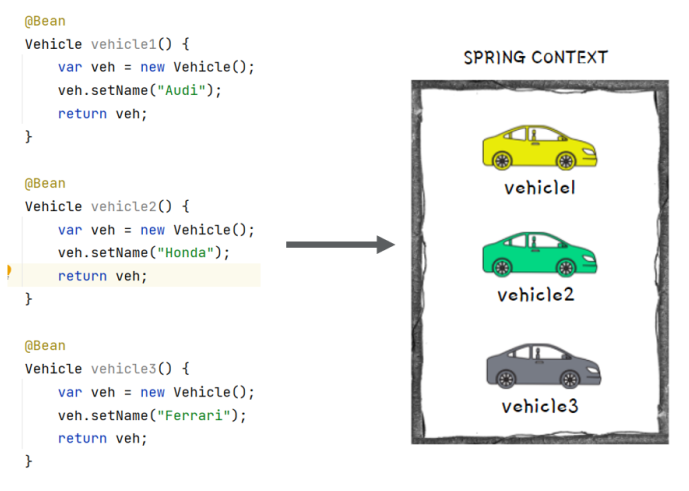
Vehicle name from Spring Context is: Audi 8

String value from Spring Context is: Hello World

Integer value from Spring Context is: 16

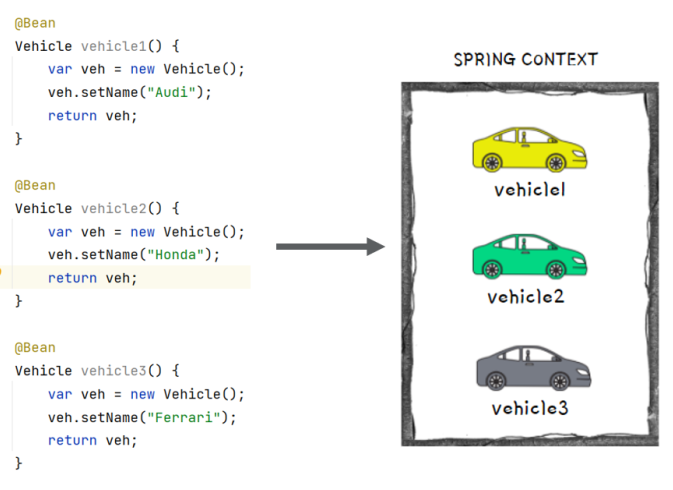
## **Understanding NoUniqueBeanDefinitionException in Spring**

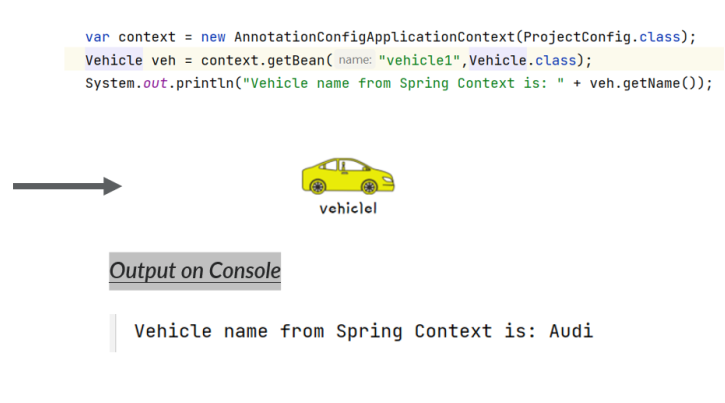
Khi chúng tôi tạo nhiều đối tượng cùng loại và cố gắng tìm nạp bean từ ngữ cảnh theo loại, thì Spring không thể đoán được trường hợp bạn đã khai báo mà bạn đề cập đến. Điều này sẽ dẫn đến NoUniqueBeanDefinitionException như hình bên dưới:





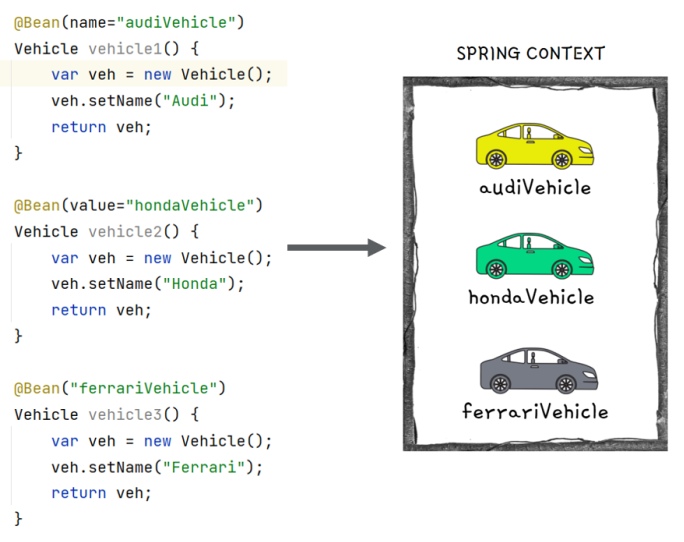
Để tránh NoUniqueBeanDefinitionException trong các tình huống này, chúng ta có thể tìm nạp bean từ ngữ cảnh bằng cách nhắc đến tên của nó như được hiển thị bên dưới:

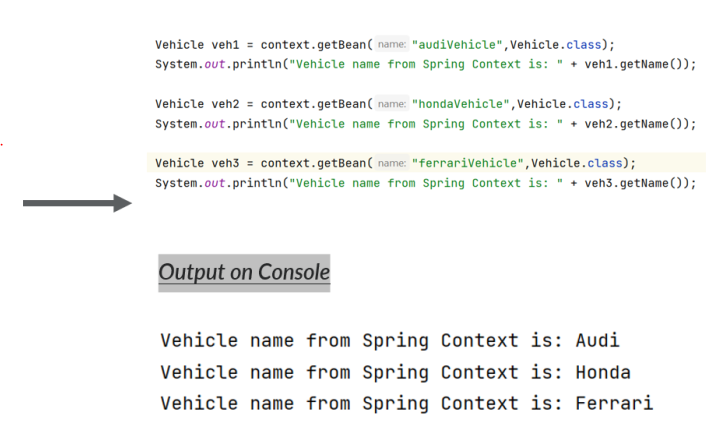




## **Providing a custom name to the bean**

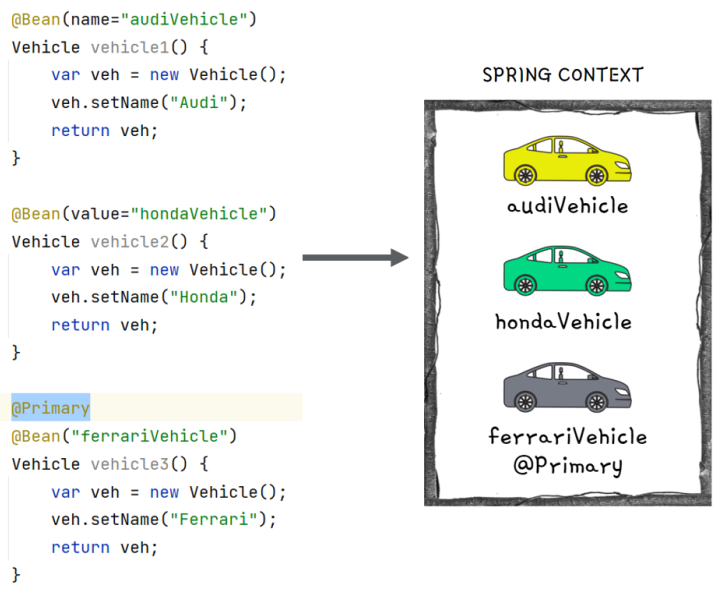
Theo mặc định, Spring sẽ coi tên phương thức là tên bean. Nhưng nếu chúng tôi có yêu cầu tùy chỉnh để xác định tên bean riêng biệt, thì chúng tôi có thể sử dụng bất kỳ phương pháp nào dưới đây với sự trợ giúp của annotation @Bean.

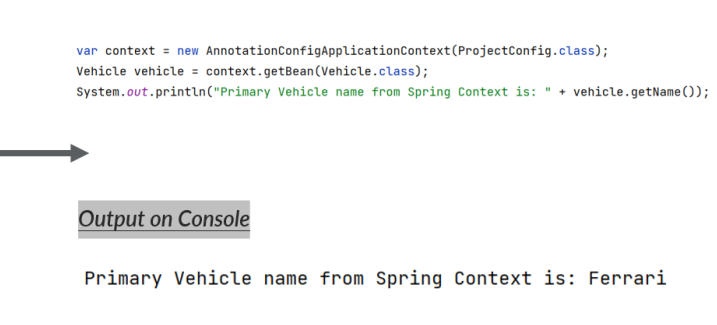




## **Understanding @Primary Annotation inside Spring**

Khi bạn có nhiều bean cùng loại trong Spring context, bạn có thể đặt một trong số chúng thành chính bằng cách sử dụng annotation @Primary. Bean chính là cái mà Spring sẽ chọn nếu nó có nhiều tùy chọn và bạn không chỉ định tên. Nói cách khác, đó là bean mặc định mà Spring Context sẽ xem xét trong trường hợp nhầm lẫn do có nhiều bean cùng loại.





## **Creating Beans using @Component annotation**

@Component là một trong những annotation khuôn mẫu được sử dụng phổ biến nhất bởi các nhà phát triển. Sử dụng điều này, chúng ta có thể dễ dàng tạo và thêm một bean vào Spring context bằng cách viết ít mã hơn so với tùy chọn @Bean. Với các annotation rập khuôn, chúng ta cần thêm annotation phía trên lớp mà chúng ta cần có một thể hiện trong Spring context.

Sử dụng annotation @ComponentScan trên lớp cấu hình, hướng dẫn Spring về nơi tìm các lớp mà bạn đã đánh dấu bằng annotation khuôn mẫu.





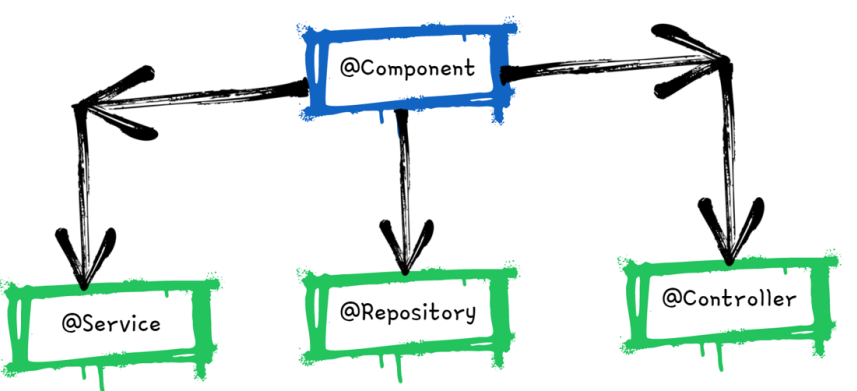
Khi một class được đánh dấu bằng @Component, Spring IoC Container sẽ quản lý và tạo ra một phiên bản của class đó như một bean trong quá trình khởi tạo ứng dụng. Bean này có thể được tiêm vào các thành phần khác thông qua Dependency Injection (DI).

Khi Spring IoC Container khởi tạo ứng dụng, nó sẽ quét các class và tìm các annotation @Component và các annotation liên quan khác để xác định các bean cần tạo và quản lý. Các bean này sẽ được đặt trong container và có thể được sử dụng trong toàn bộ ứng dụng.

## **Spring Stereotype Annotations - Chú thích khuôn mẫu spring**

+ Spring cung cấp **các annotation** đặc biệt gọi là Stereotype annotations sẽ giúp tạo các Spring bean một cách tự động trong ngữ cảnh ứng dụng.

+ Các **Stereotype Annotations** trong spring là @Component, @Service, @Repository và @Controller



* **@Component** được sử dụng chung trên bất kỳ lớp Java nào. Nó là cơ sở cho các annotation khác.
* **@Service** có thể được sử dụng trên các lớp bên trong lớp **service**, đặc biệt là khi chúng tôi viết logic nghiệp vụ và thực hiện lệnh gọi API bên ngoài.
* **@Repostiory** có thể được sử dụng trên các lớp xử lý mã liên quan đến các hoạt động liên quan đến truy cập Database như Insert, Update, Delete, v.v.
* **@Controller** có thể được sử dụng trên các lớp bên trong lớp Controller của các ứng dụng MVC.

## **Comparison between @Bean Vs @Component**

|  |  |
| --- | --- |
|  | |
| * Một hoặc nhiều thể hiện của lớp có thể được thêm vào Spring context * Chúng ta có thể tạo một thể hiện đối tượng của bất kỳ loại lớp nào, kể cả các thư viện hiện tại bên trong như String, v.v. * Thông thường chúng ta cần viết thêm mã như các phương thức riêng biệt để tạo các phiên bản bean * Developer sẽ có toàn quyền trong việc tạo và cấu hình bean * Khung công tác mùa xuân tạo bean dựa trên các hướng dẫn và giá trị do Developer cung cấp | * Chỉ một thể hiện của lớp có thể được thêm vào Spring context * Chúng ta chỉ có thể tạo một thể hiện đối tượng cho lớp ứng dụng được tạo bởi nhóm Dev * Các phiên bản Bean có thể được tạo với rất ít mã như sử dụng @Component ở đầu lớp * Developer sẽ không có bất kỳ quyền kiểm soát trong việc tạo và định cấu hình bean * Khung công tác mùa xuân chịu trách nhiệm tạo bean và đăng bài mà Nhà phát triển sẽ có quyền truy cập vào nó |

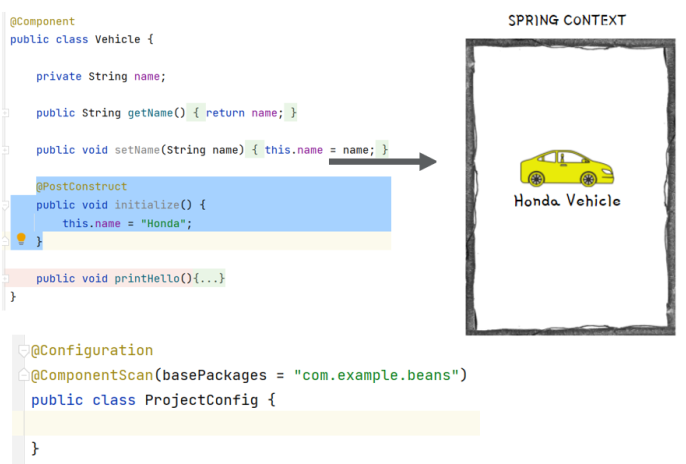
## **Understanding @PostConstruct Annotation**

Chúng tôi đã thấy rằng khi chúng tôi đang sử dụng các stereotype annotation, chúng tôi không có quyền kiểm soát trong khi tạo một bean. Nhưng điều gì sẽ xảy ra nếu chúng ta muốn thực hiện một số hướng dẫn sau khi Spring tạo bean. Tương tự, chúng ta có thể sử dụng annotation @PostConstruct.

@PostConstruct là một annotation được sử dụng để đánh dấu một phương thức trong một bean sẽ được thực hiện ngay sau khi việc khởi tạo bean và injection các giá trị phụ thuộc đã hoàn thành. Annotation này cho phép chúng ta thực hiện các tác vụ khởi tạo sau khi bean đã được khởi tạo nhưng trước khi bean được sử dụng.

Các phương thức được đánh dấu bằng @PostConstruct phải được public và không được chứa bất kỳ tham số nào. Spring IoC Container sẽ tìm và thực hiện các phương thức này tự động.

Spring mượn annotation @PostConstruct từ Java EE.



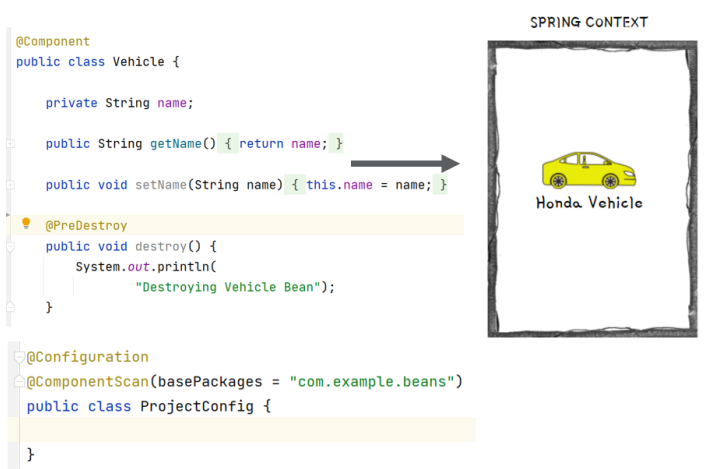


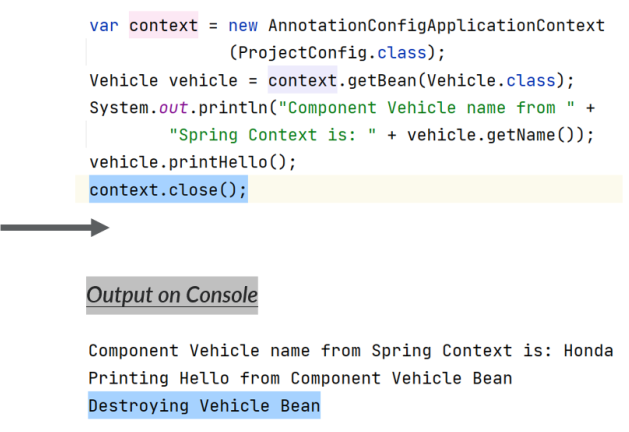
## **Understanding @PreDestroy Annotation**

Chú thích @PreDestory có thể được sử dụng trên các phương thức và Spring sẽ đảm bảo gọi phương thức này ngay trước khi xóa và hủy context.

Điều này có thể được sử dụng trong các tình huống mà chúng tôi muốn đóng tài nguyên bất kỳ, kết nối Database, v.v.

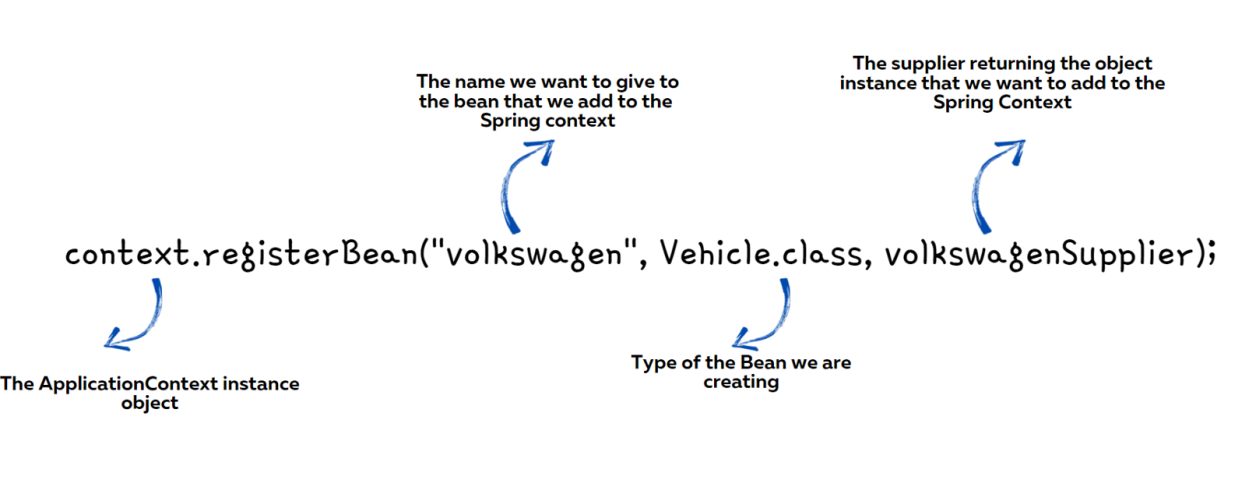
Spring mượn @PreDestory annotation cũng từ Java EE.

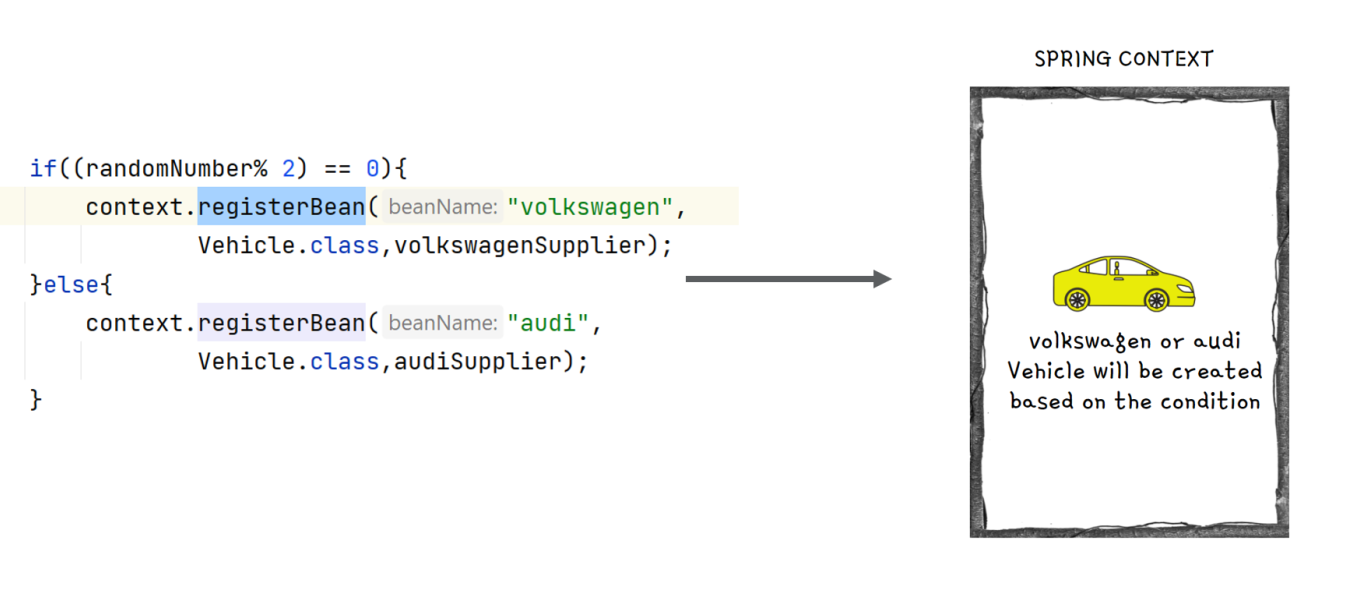




## **Creating Beans programmatically using registerBean()**

Đôi khi chúng ta muốn tạo các thể hiện mới của một đối tượng và thêm chúng vào Spring context dựa trên điều kiện lập trình. Tương tự như vậy, từ phiên bản Spring 5, một cách tiếp cận mới được cung cấp để tạo các bean theo chương trình bằng cách gọi phương thức resisterBean() có bên trong đối tượng ngữ cảnh.





public class Example7 {  
  
 public static void main(String[] args) {  
  
 var context = new AnnotationConfigApplicationContext(ProjectConfig.class);  
  
 Vehicle volkswagen = new Vehicle();  
 volkswagen.setName("Volkswagen");  
 Supplier<Vehicle> volkswagenSupplier = () -> volkswagen;  
  
  
 Supplier<Vehicle> audiSupplier = () -> {  
 Vehicle audi = new Vehicle();  
 audi.setName("Audi");  
 return audi;  
 };  
  
 Random random = new Random();  
 int randomNumber = random.nextInt(10);  
 System.*out*.println("randomNumber = " + randomNumber);  
  
 if((randomNumber% 2) == 0){  
 context.registerBean("volkswagen",  
 Vehicle.class,volkswagenSupplier);  
 }else{  
 context.registerBean("audi",  
 Vehicle.class,audiSupplier);  
 }  
 Vehicle volksVehicle = null;  
 Vehicle audiVehicle = null;  
 try {  
 volksVehicle = context.getBean("volkswagen",Vehicle.class);  
 }catch (NoSuchBeanDefinitionException noSuchBeanDefinitionException){  
 System.*out*.println("Error while creating Volkswagen vehicle");  
 }  
 try {  
 audiVehicle = context.getBean("audi",Vehicle.class);  
 }catch (NoSuchBeanDefinitionException noSuchBeanDefinitionException){  
 System.*out*.println("Error while creating Audi vehicle");  
 }  
  
 if(null != volksVehicle){  
 System.*out*.println("Programming Vehicle name from Spring Context is: " + volksVehicle.getName());  
 }else{  
 System.*out*.println("Programming Vehicle name from Spring Context is: " + audiVehicle.getName());  
 }  
  
 }  
}

// output

randomNumber = 5

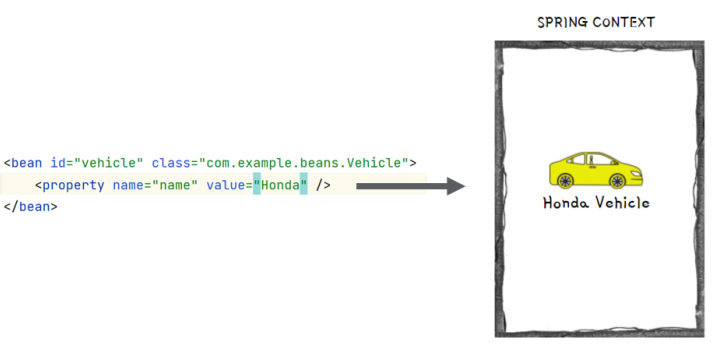
Error while creating Volkswagen vehicle

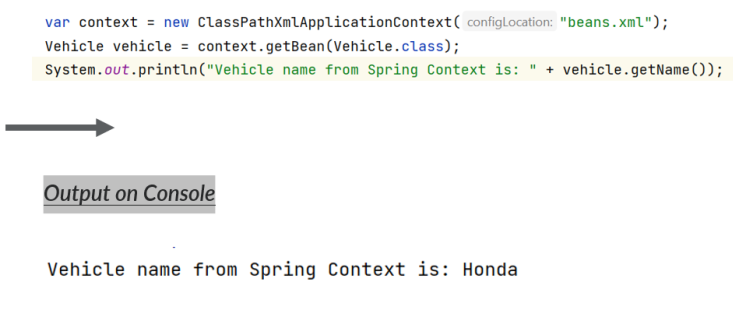
Programming Vehicle name from Spring Context is: Audi

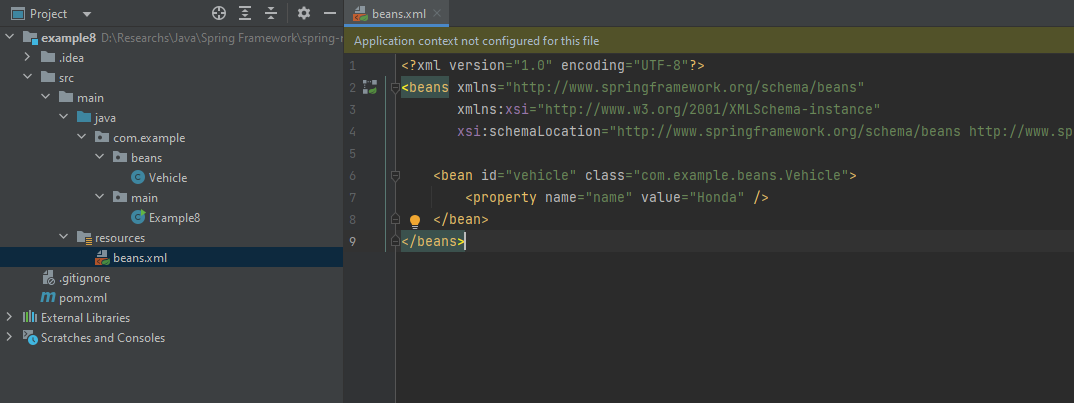
## **Creating Beans using XML Configurations (example)**

Trong các phiên bản đầu tiên của Spring, bean và các cấu hình khác thường được thực hiện bằng XML. Nhưng theo thời gian, nhóm Spring mang đến các cấu hình dựa trên annotation để giúp các nhà phát triển trở nên dễ dàng. Ngày nay, chúng ta chỉ có thể thấy các cấu hình XML trong các ứng dụng cũ hơn được xây dựng dựa trên các phiên bản đầu tiên của Spring.

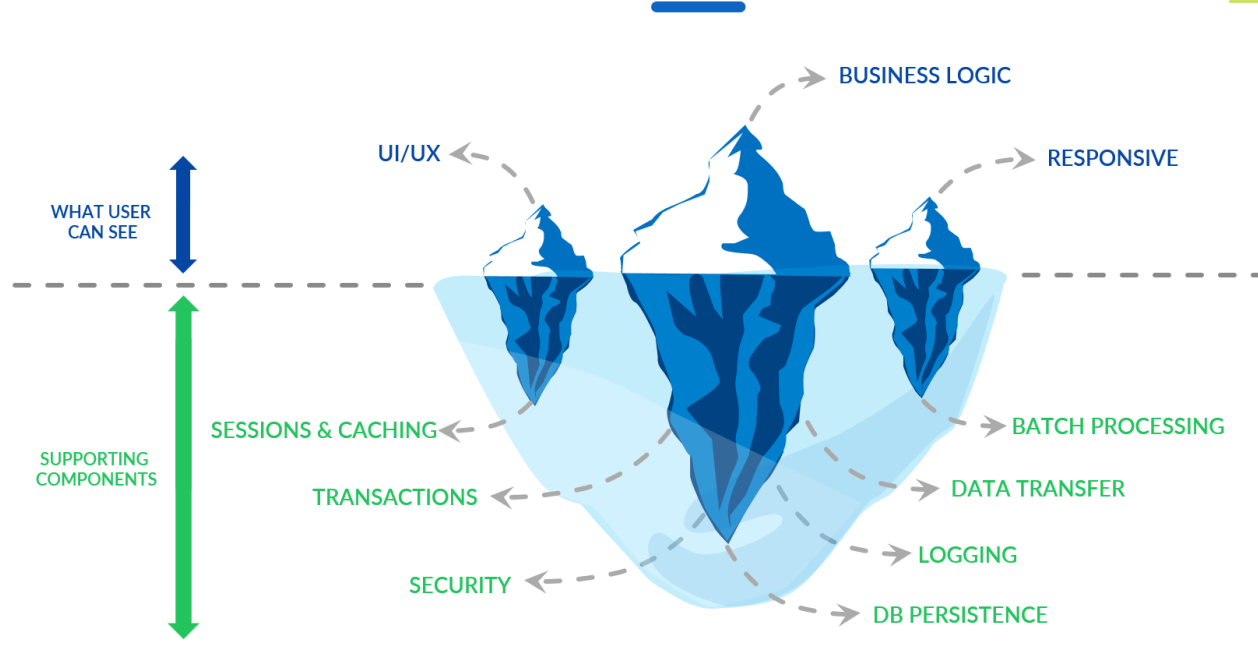
Bạn nên hiểu cách tạo một bean bên trong Spring context bằng cách sử dụng các cấu hình kiểu XML. Vì vậy, sẽ rất hữu ích nếu có một kịch bản mà bạn cần làm việc trong một dự án dựa trên các phiên bản ban đầu của Spring.





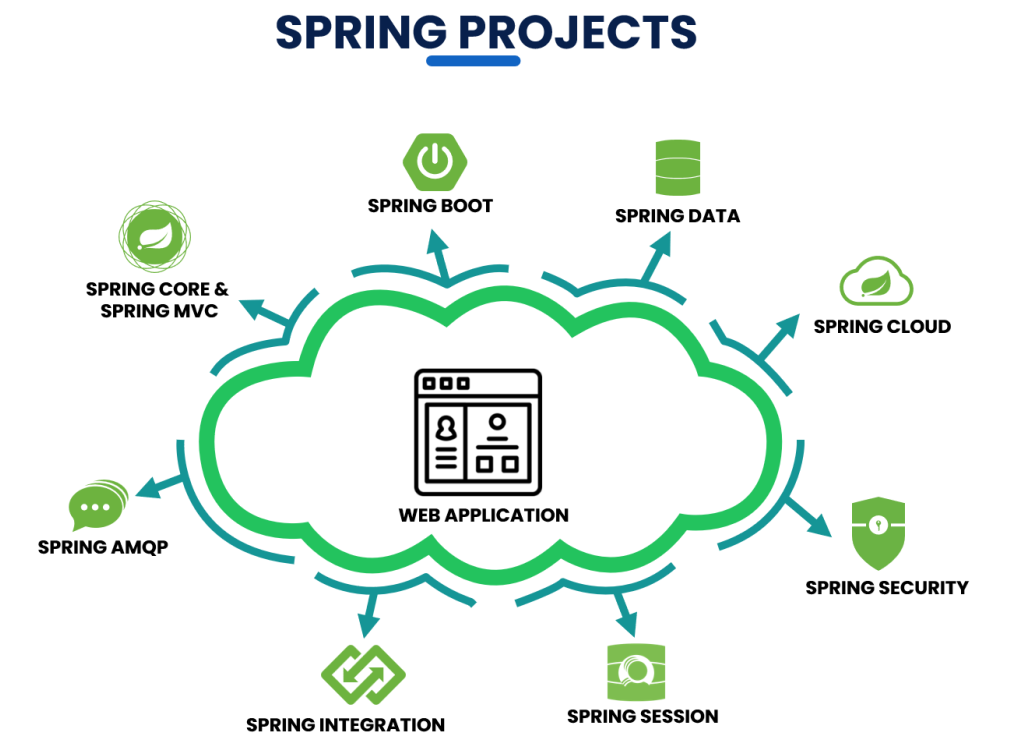


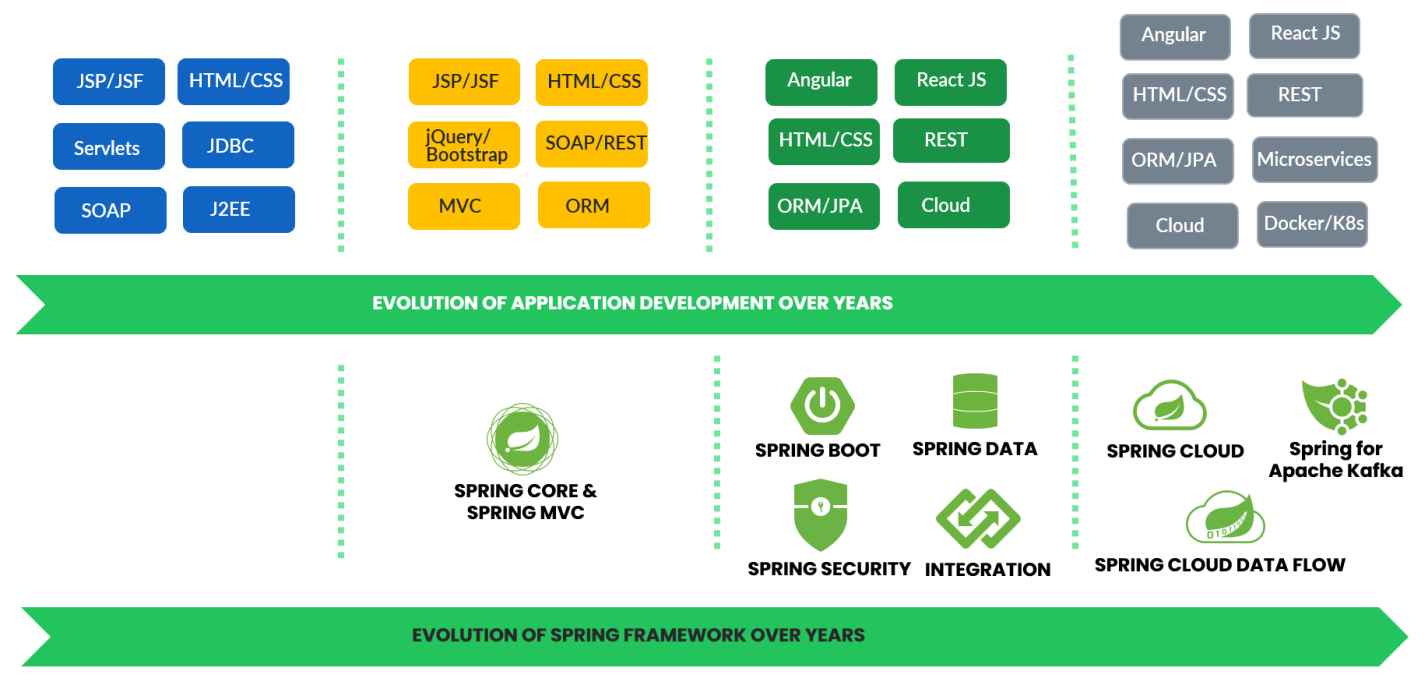
## **Why should we use frameworks**



|  |  |
| --- | --- |
|  | |
| * Tận dụng Security, Logging etc. from frameworks * Có thể dễ dàng mở rộng quy mô ứng dụng của mình * Ứng dụng sẽ hoạt động theo cách có thể dự đoán được * Tập trung nhiều hơn vào logic kinh doanh * Ít nỗ lực hơn và nhiều kết quả/doanh thu hơn | * Cần xây dựng mã cho Security, Logging, v.v. * Mở rộng quy mô của anh ấy không phải là một lựa chọn cho đến khi anh ấy kiểm tra mọi thứ * Ứng dụng có thể không hoạt động theo cách có thể dự đoán được * Tập trung nhiều hơn vào các thành phần hỗ trợ * Nhiều nỗ lực hơn và ít kết quả/doanh thu hơn |

## **Introduction to Spring Projects**





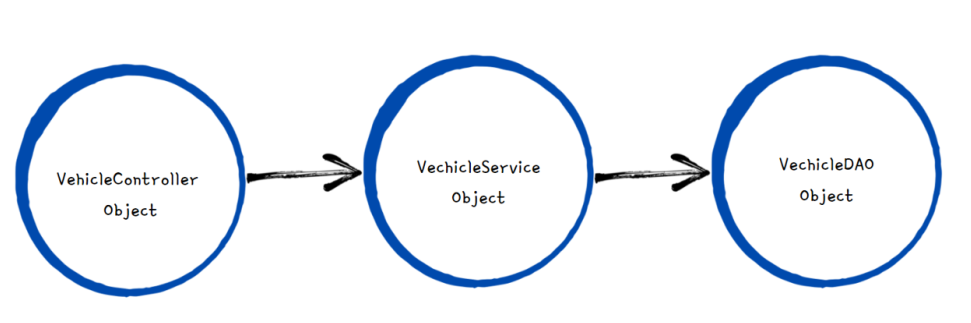
# **Section 3: Wiring Beans using @Autowiring**

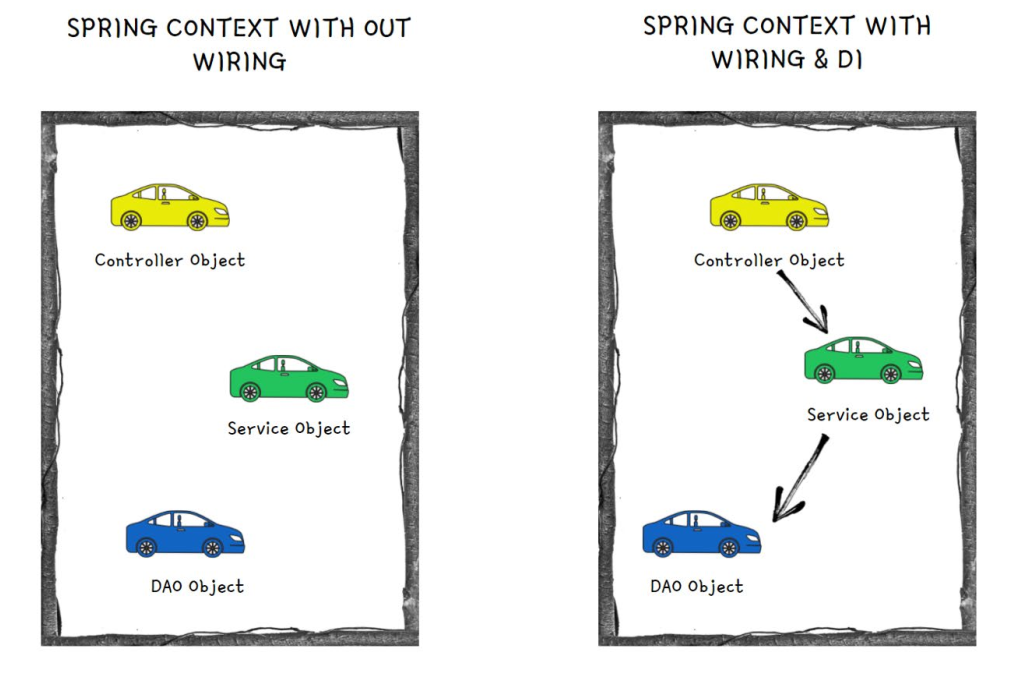
## **Introduction to wiring & auto-wiring inside Spring**

Trong Spring Framework, wiring là quá trình kết nối các thành phần (beans) với nhau để tạo ra sự phụ thuộc giữa chúng. Wiring có thể được thực hiện bằng cách chỉ định các liên kết giữa các beans trong cấu hình hoặc sử dụng tự động kết nối (auto-wiring) dựa trên quy tắc được xác định.

Wiring trong Spring có mục đích chính là đảm bảo rằng các beans có thể tương tác và làm việc với nhau một cách hợp lý. Khi các beans được kết nối, các phụ thuộc (dependencies) giữa chúng được giải quyết, cho phép chúng truy cập và sử dụng các tính năng, dịch vụ hoặc tài nguyên của nhau.

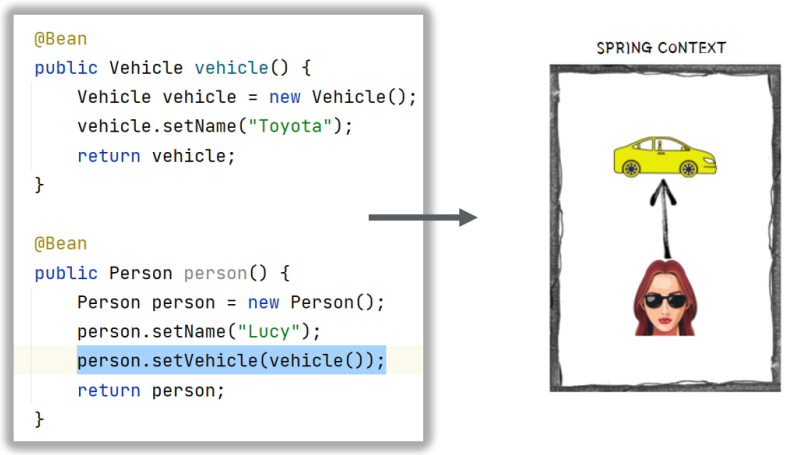
Auto-wiring là một tính năng của Spring Framework cho phép tự động kết nối các beans với nhau mà không cần chỉ định rõ ràng trong cấu hình. Spring sẽ tự động tìm kiếm các beans phù hợp dựa trên kiểu dữ liệu và tên biến và thực hiện kết nối tự động.

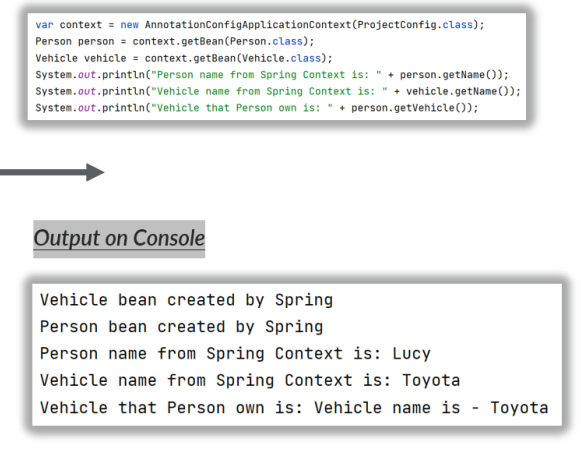




## **Wiring Beans using method call**

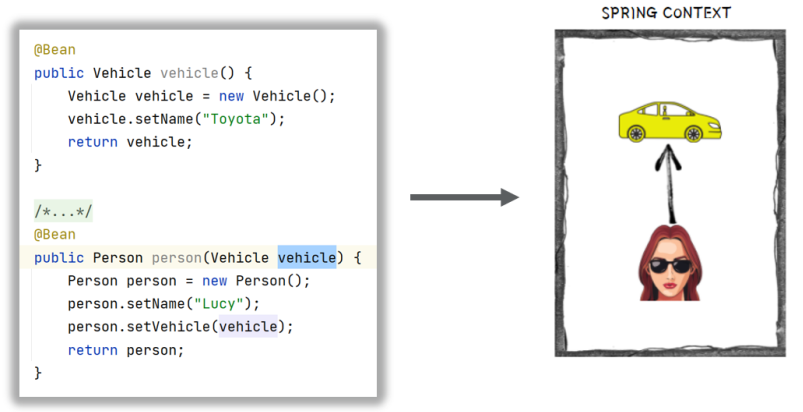
Hãy xem xét một kịch bản trong đó chúng ta có hai lớp java Person và Vehicle. Lớp Person có sự phụ thuộc vào Vehicle. Dựa trên đoạn mã dưới đây, chúng tôi chỉ tạo các bean bên trong Spring Context và sẽ không thực hiện nối dây. Do đó, cả hai Bean này đều có mặt trong Spring Context mà không biết về nhau.





## **Wiring Beans using method parameters**

Ở đây trong đoạn mã dưới đây, chúng tôi đang cố gắng kết nối hoặc thiết lập mối quan hệ giữa Person và Vehicle, bằng cách gọi phương thức vehicle() từ phương thức person (). Bây giờ bên trong Sprint Context, person sở hữu vehicle. Spring sẽ đảm bảo chỉ có 1 vehicle bean được tạo và vehicle bean cũng sẽ luôn được tạo trước vì person bean phụ thuộc vào nó.



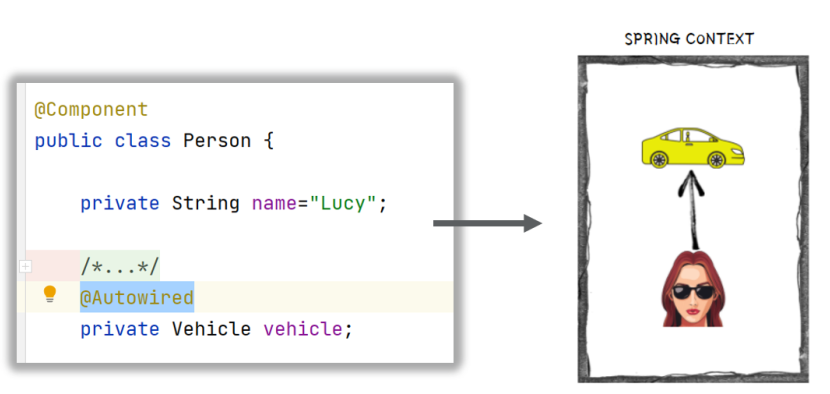


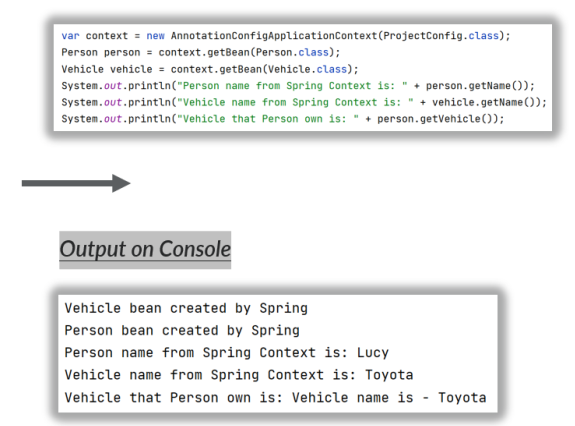
## **Wiring Beans using @Autowired on class fields (example 11)**

@Autowired annotation đánh dấu trên một trường, phương thức setter, hàm tạo được sử dụng để tự động kết nối các bean đang ‘injecting beans'(Objects) vào thời gian chạy theo cơ chế Spring Dependency Injection.

Với đoạn mã dưới đây, Spring injects/auto-wire kết vehicle bean với person bean thông qua class field và dependency injection.

Kiểu bên dưới không được khuyến nghị cho việc sử dụng production vì chúng tôi không thể đánh dấu các trường là final.



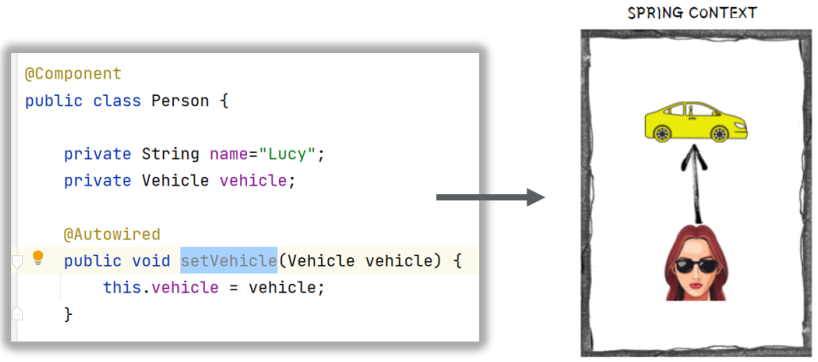


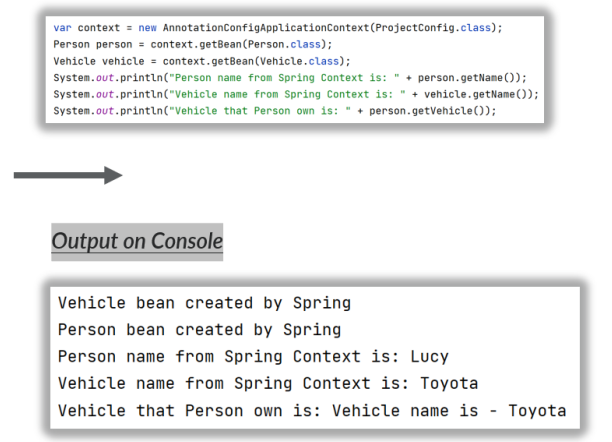
=> @Autowired(required = false) sẽ giúp tránh NoSuchBeanDefinitionException nếu bean không có sẵn trong quá trình Autowiring.

## **Wiring Beans using @Autowired on setter method**

Với đoạn mã dưới đây, Spring injects/auto-wire kết nối vehicle bean với person bean thông qua phương thức setter và dependency injection.

Kiểu bên dưới không được khuyến nghị cho việc sử dụng production vì chúng tôi không thể đánh dấu các trường là final và không thân thiện với người đọc.

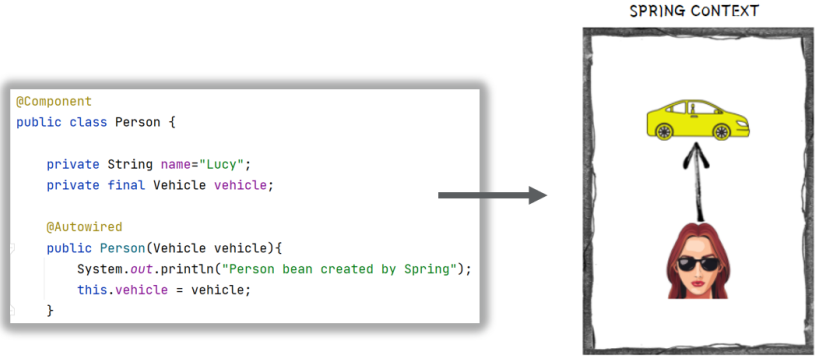


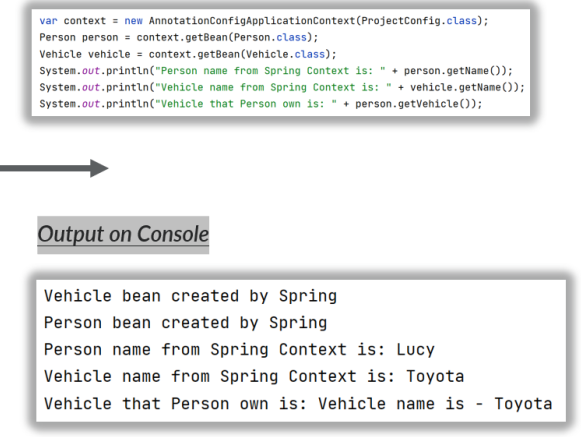


## **Wiring Beans using @Autowired on constructor**

Với đoạn mã dưới đây, Spring injects/auto-wire kết nối vehicle bean với person bean thông qua hàm constructor và dependency injection.

Từ phiên bản Spring 4.3, khi chúng ta chỉ có một hàm constructor trong lớp, việc viết annotation @Autowired là tùy chọn.



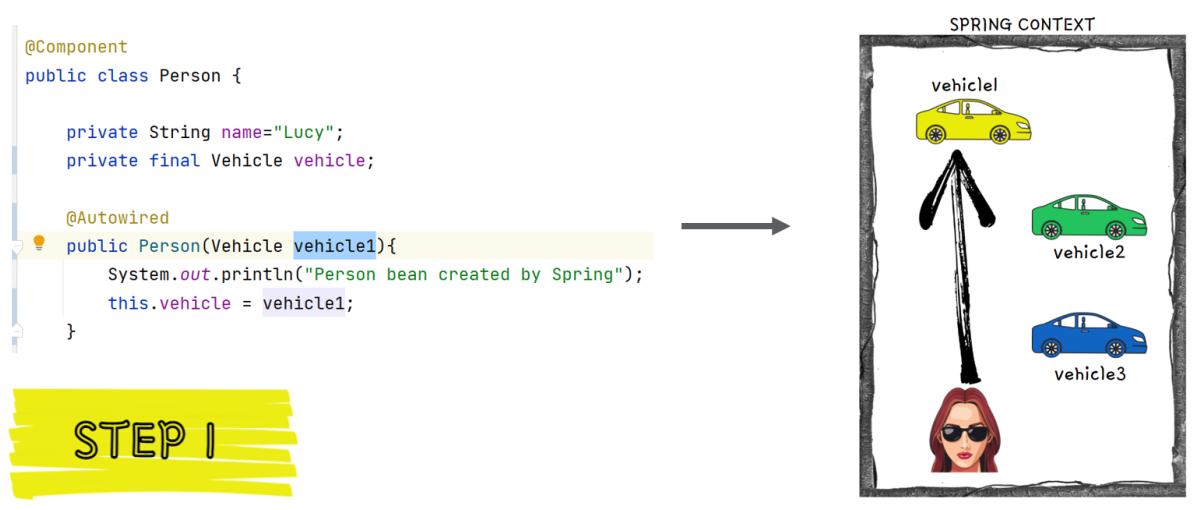


## **Deep dive of Autowiring inside Spring (example12)**

Theo mặc định, Spring cố gắng tự động kết nối với loại lớp. Nhưng cách tiếp cận này sẽ thất bại nếu cùng một loại lớp có nhiều bean.

Nếu Spring context có nhiều bean cùng loại như bên dưới, thì Spring sẽ cố gắng auto-wire dựa trên tên name/field mà chúng ta sử dụng trong khi định cấu hình autowiring annotation.

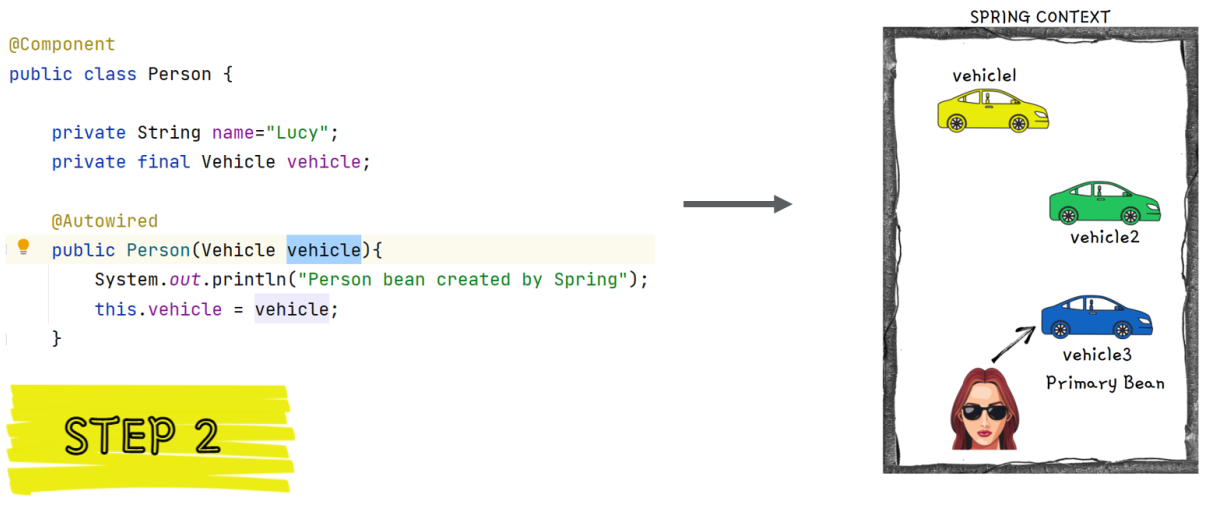
Trong trường hợp bên dưới, chúng tôi đã sử dụng ‘vehicle1’ làm tham số hàm constructor. Spring sẽ cố gắng auto-wire với bean có cùng tên như trong hình bên dưới.



public class ProjectConfig {  
  
 @Bean  
 Vehicle vehicle1() {  
 var veh = new Vehicle();  
 veh.setName("Audi");  
 return veh;  
 }  
  
 @Bean  
 Vehicle vehicle2() {  
 var veh = new Vehicle();  
 veh.setName("Honda");  
 return veh;  
 }  
  
 @Bean  
 Vehicle vehicle3() {  
 var veh = new Vehicle();  
 veh.setName("Ferrari");  
 return veh;  
 }  
  
}

Nếu tên name/field mà chúng ta sử dụng trong khi định cấu hình autowiring annotation không khớp với bất kỳ tên bean nào, thì Spring sẽ tìm bean đã cấu hình @Primary.

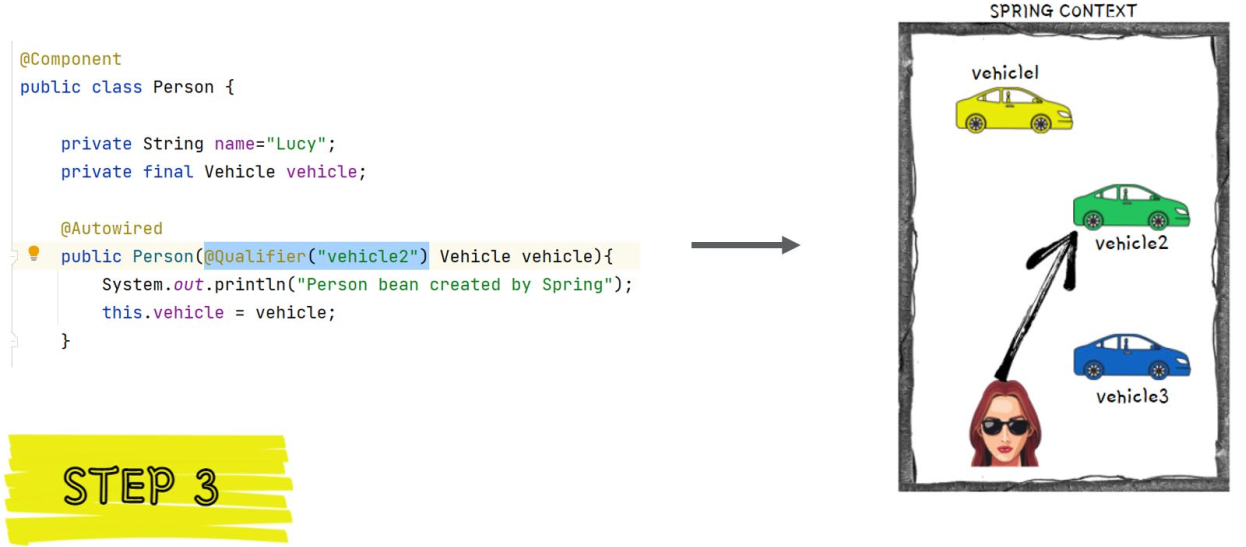
Trong trường hợp bên dưới, chúng tôi đã sử dụng ‘vehicle’ làm tham số hàm constructor. Spring sẽ cố gắng auto-wire với bean có cùng tên và vì nó không thể tìm thấy bean có cùng tên, nên nó sẽ tìm bean có cấu hình @Primary như trong hình bên dưới.



public class ProjectConfig {  
   
 @Bean  
 @Primary  
 Vehicle vehicle1() {  
 var veh = new Vehicle();  
 veh.setName("Audi");  
 return veh;  
 }  
  
 @Bean  
 Vehicle vehicle2() {  
 var veh = new Vehicle();  
 veh.setName("Honda");  
 return veh;  
 }  
  
 @Bean  
 Vehicle vehicle3() {  
 var veh = new Vehicle();  
 veh.setName("Ferrari");  
 return veh;  
 }  
  
}

+ Nếu tên name/field mà chúng ta sử dụng trong khi định cấu hình autowiring annotation không khớp với bất kỳ tên bean nào và ngay cả Primary bean cũng không được định cấu hình, thì Spring sẽ xem xét chú thích @Qualifier có được sử dụng với tên bean phù hợp với Spring context.

Trong kịch bản bên dưới, chúng tôi đã sử dụng ‘vehicle2’ với chú thích @Qualifier annotation. Spring sẽ cố gắng auto-wire với bean có cùng tên như trong hình bên dưới.



Chú ý nếu **@Primary** và **@Qualifier** cùng được sử dụng thì Spring sẽ chọn **@Qualifier**

## **Understanding & Avoiding Circular dependencies**

Trong Spring Framework, circular dependencies (phụ thuộc vòng) xảy ra khi hai hoặc nhiều beans phụ thuộc lẫn nhau theo cách tạo ra một chuỗi vòng lặp. Điều này có thể gây ra các vấn đề và khó khăn trong quá trình tạo và quản lý các beans.

Để hiểu và tránh circular dependencies trong Spring, hãy cùng tìm hiểu một số khái niệm quan trọng:

### **Circular Dependencies là gì?**

Circular dependencies xảy ra khi hai hoặc nhiều beans tạo thành một vòng phụ thuộc, tức là bean A phụ thuộc vào bean B, bean B phụ thuộc vào bean C, và bean C phụ thuộc lại vào bean A.

### **Vấn đề của Circular Dependencies:**

Circular dependencies có thể gây ra các vấn đề như:

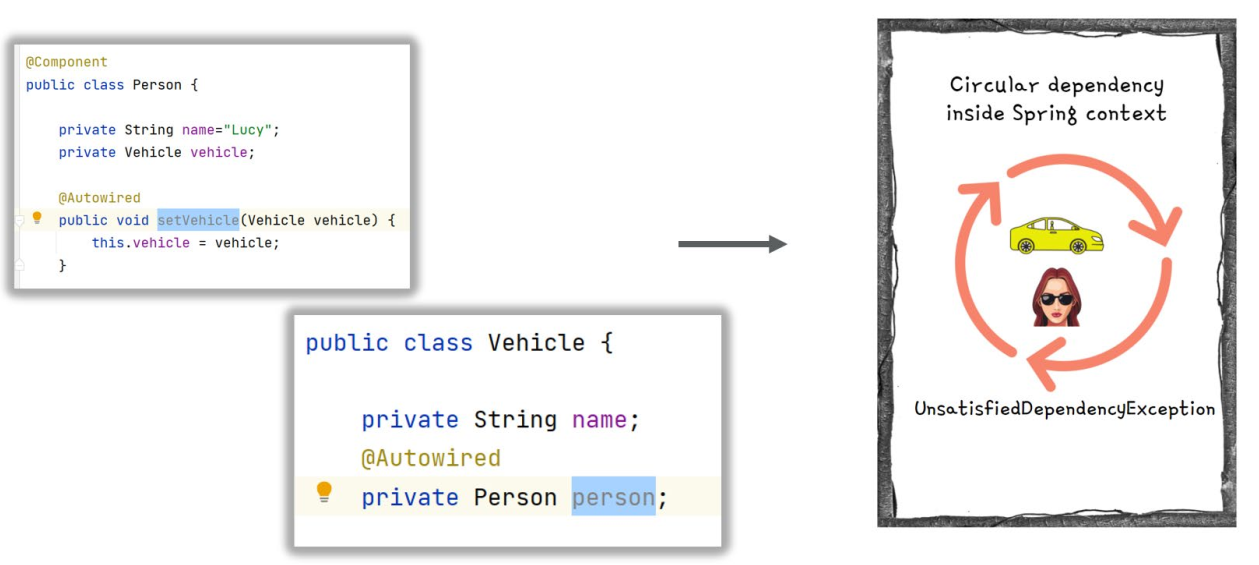
* Khó khăn trong việc khởi tạo beans: Spring không thể xác định thứ tự khởi tạo các beans trong một chuỗi phụ thuộc vòng.
* Rủi ro Deadlock: Nếu các phụ thuộc giữa các beans không được quản lý cẩn thận, có thể xảy ra deadlock khi các beans cố gắng khởi tạo và phụ thuộc vào nhau.
* Giảm khả năng kiểm tra và bảo trì: Vì các beans phụ thuộc lẫn nhau, việc kiểm tra, sửa lỗi và bảo trì có thể trở nên phức tạp hơn.

### **Cách tránh Circular Dependencies:**

Để tránh circular dependencies trong Spring, bạn có thể áp dụng các phương pháp sau:

* Sử dụng Dependency Injection qua Constructor: Thay vì sử dụng injection qua các thuộc tính hoặc setter method, hãy sử dụng injection qua constructor. Điều này giúp xác định rõ ràng các phụ thuộc và tránh vòng phụ thuộc.
* Sử dụng Setter Injection hoặc Field Injection: Nếu bạn không thể sử dụng constructor injection, hãy sử dụng setter injection hoặc field injection. Tuy nhiên, đảm bảo rằng không có phụ thuộc vòng trong quá trình injection.
* Sử dụng Lazy Initialization: Sử dụng lazy initialization để trì hoãn việc khởi tạo các beans và tránh circular dependencies trong quá trình khởi tạo ban đầu.
* Tái cấu trúc các beans: Xem xét lại cấu trúc của các beans và xác định xem có thể tách chúng thành các beans riêng lẻ để tránh circular dependencies.

Tuy nhiên, tránh hoàn toàn circular dependencies không phải lúc nào cũng thực tế hoặc dễ dàng. Trong một số trường hợp, bạn có thể cần xem xét lại thiết kế và sắp xếp lại các phụ thuộc để giảm thiểu tối đa circular dependencies.



## **Problem Statement for Assignment related to Beans, Autowiring and DI (example13)**



# **Section 4: Beans scope inside Spring framework**

## **Introduction to Bean Scopes inside Spring**

Trong Spring Framework, bean scope (phạm vi bean) là một khái niệm quan trọng để xác định thời gian tồn tại và sự sử dụng của một bean trong container. Spring hỗ trợ nhiều loại bean scopes khác nhau, mỗi loại phù hợp với các tình huống sử dụng khác nhau.

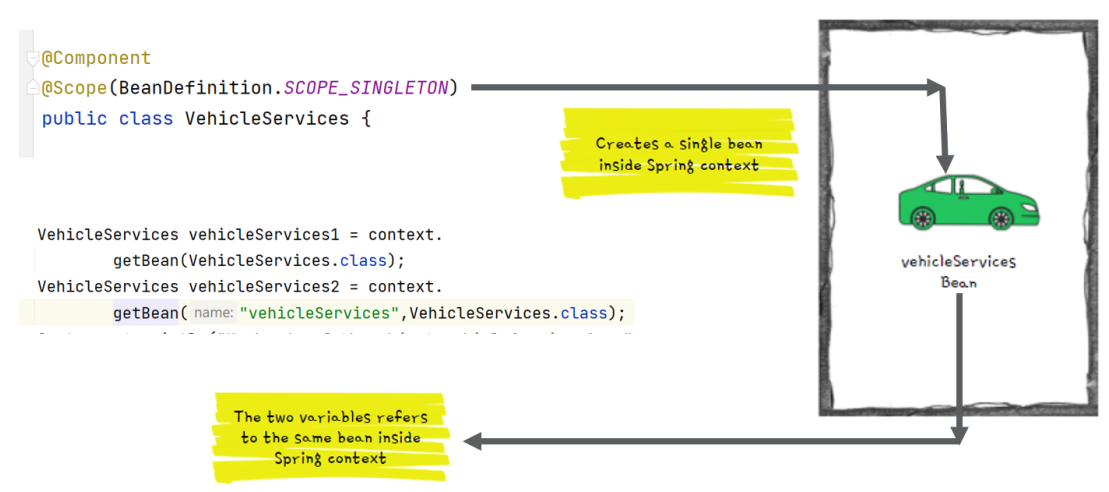
Dưới đây là một số loại bean scopes phổ biến trong Spring:

1. Singleton: Đây là mặc định scope của một bean trong Spring. Một bean với scope singleton chỉ có một phiên bản duy nhất trong toàn bộ container. Mỗi lần bean được yêu cầu, Spring sẽ cung cấp phiên bản đã tồn tại hoặc tạo mới nếu chưa có. Singleton scope đảm bảo rằng chỉ có một instance duy nhất của bean và được chia sẻ trong toàn bộ ứng dụng.
2. Prototype: Mỗi lần bean được yêu cầu, Spring sẽ tạo ra một phiên bản mới. Điều này đảm bảo rằng mỗi bean được sử dụng đều là một instance riêng biệt. Prototype scope thích hợp cho các bean có trạng thái (stateful) hoặc khi cần tạo ra nhiều phiên bản của bean.
3. Request: Mỗi bean trong scope này sẽ tồn tại trong suốt một request HTTP. Điều này có nghĩa là mỗi request sẽ có một instance riêng của bean. Scope này thường được sử dụng trong môi trường web để đảm bảo rằng các bean không chia sẻ giữa các request.
4. Session: Mỗi bean trong scope này tồn tại trong suốt một session HTTP. Điều này đảm bảo rằng các bean chỉ được chia sẻ giữa các yêu cầu của cùng một session. Scope này thích hợp cho các bean lưu trữ dữ liệu liên quan đến một phiên làm việc của người dùng trong ứng dụng web.
5. Global Session: Tương tự như session scope, nhưng được sử dụng trong môi trường portlet. Mỗi bean trong scope này tồn tại trong suốt một global session portlet.
6. Application: Bean có scope application tồn tại trong suốt vòng đời của ứng dụng. Một instance duy nhất của bean sẽ được tạo ra và được chia sẻ cho toàn bộ ứng dụng. Scope này thích hợp cho các bean không thay đổi và cần chia sẻ thông qua toàn bộ ứng dụng.
7. Websocket: Đây là scope mới được giới thiệu từ Spring 4.2, được sử dụng cho các bean liên quan đến WebSocket trong ứng dụng web.

## **Deepdive on Singleton Bean scope (example14)**

Singleton là phạm vi mặc định của bean trong Spring. Trong phạm vi này, đối với một bean duy nhất, chúng tôi luôn nhận được một trường hợp tương tự khi bạn refer hoặc autowire bên trong ứng dụng của mình.

Không giống như mẫu thiết kế Singleton nơi chúng ta chỉ có 1 phiên bản trong toàn bộ ứng dụng, bên trong **Singleton** **scope**, Spring sẽ đảm bảo chỉ có 1 phiên bản cho mỗi bean duy nhất. Ví dụ: nếu bạn có nhiều bean cùng loại, thì phạm vi Spring Singleton sẽ duy trì 1 phiên bản cho mỗi bean được khai báo cùng loại.



public class Example14 {  
  
 public static void main(String[] args) {  
 var context = new AnnotationConfigApplicationContext(ProjectConfig.class);  
 VehicleServices vehicleServices1 = context.getBean(VehicleServices.class);  
 VehicleServices vehicleServices2 = context.getBean("vehicleServices",VehicleServices.class);  
 System.*out*.println("Hashcode of the object vehicleServices1 : " +vehicleServices1.hashCode());  
 System.*out*.println("Hashcode of the object vehicleServices2 : " +vehicleServices2.hashCode());  
 if(vehicleServices1==vehicleServices2){  
 System.*out*.println("VehicleServices bean is a singleton scoped bean");  
 }  
 }  
}

//output

Person bean created by Spring

Hashcode of the object vehicleServices1 : 1606304070

Hashcode of the object vehicleServices2 : 1606304070

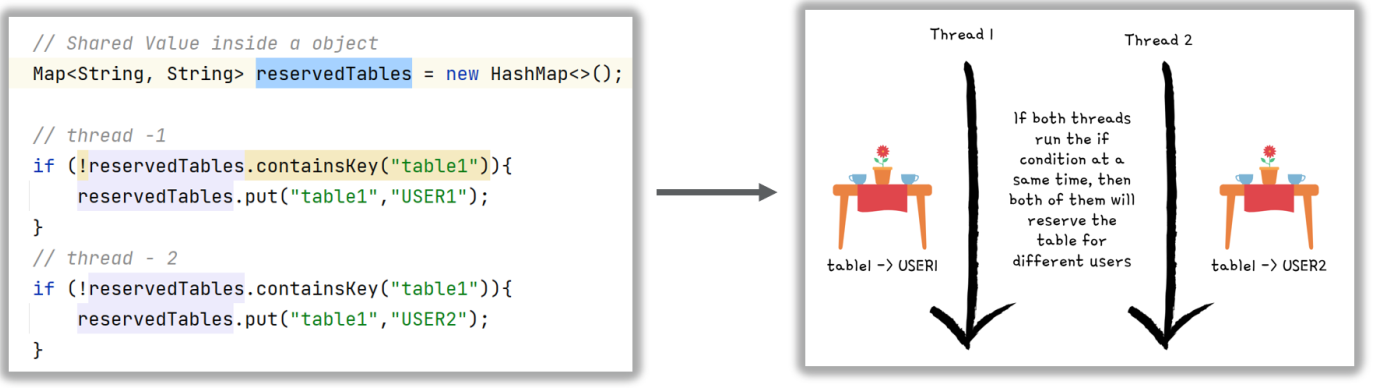
VehicleServices bean is a singleton scoped bean

## **3. What is a Race Condition**

Race condition (điều kiện đua) là một tình huống xảy ra trong lập trình đồng thời khi hành vi của chương trình phụ thuộc vào thứ tự hoặc thời gian thực thi của các luồng (threads) hoặc quá trình (processes) khác nhau. Nó xảy ra khi nhiều luồng hoặc quá trình truy cập vào dữ liệu hoặc tài nguyên chung cùng một lúc, và kết quả cuối cùng của chương trình phụ thuộc vào thứ tự cụ thể mà các luồng được lập lịch chạy.

Trong một race condition, việc thực thi và kết quả đúng của chương trình phụ thuộc vào việc xen kẽ (interleaving) hoặc lập lịch của các luồng, điều này có thể không đoán trước được và không xác định. Điều này có thể dẫn đến kết quả không mong muốn và sai sót, vì các luồng có thể gây ảnh hưởng lẫn nhau hoặc ghi đè lên dữ liệu của nhau.

Race condition thường xảy ra khi nhiều luồng hoặc quá trình thực hiện các hoạt động đọc-sửa-ghi (read-modify-write) trên dữ liệu chung mà không có các cơ chế đồng bộ hóa hoặc phối hợp đúng đắn. Ví dụ, nếu hai luồng cố gắng tăng một biến đếm chia sẻ cùng một lúc, giá trị cuối cùng của biến đếm có thể không chính xác do việc xen kẽ các hoạt động.



## **Singleton Beans use cases**

Vì cùng một phiên bản của singleton bean sẽ được sử dụng bởi nhiều luồng bên trong ứng dụng của bạn, nên điều rất quan trọng là các bean này là bất biến.

Phạm vi này phù hợp hơn cho các bean xử lý service layer, repository layer, business logics.

1. Xây dựng các singleton bean có thể thay đổi, sẽ dẫn đến các race conditions bên trong môi trường đa luồng.
2. Có nhiều cách để tránh các race condition do singleton beans có thể thay đổi với sự trợ giúp của đồng bộ hóa.
3. Nhưng nó không được khuyến khích, vì nó mang lại nhiều vấn đề phức tạp và hiệu suất bên trong ứng dụng của bạn. Vì vậy, vui lòng không cố gắng xây dựng các loại singleton bean có thể thay đổi.

Singleton bean scope trong Spring được sử dụng để đảm bảo rằng chỉ có một phiên bản duy nhất của một bean được tạo ra và chia sẻ trong toàn bộ ứng dụng. Đây là một trong những phạm vi bean phổ biến nhất trong Spring và được sử dụng trong nhiều trường hợp khác nhau. Dưới đây là một số use case phổ biến của Singleton bean scope:

* Cấu hình ứng dụng: Singleton scope được sử dụng cho các bean chịu trách nhiệm cấu hình và cung cấp dịch vụ cấu hình cho toàn bộ ứng dụng. Ví dụ: bean chịu trách nhiệm đọc cấu hình từ tệp cấu hình và cung cấp các giá trị cấu hình cho các thành phần khác trong ứng dụng.
* Kết nối cơ sở dữ liệu: Singleton scope thường được sử dụng cho các bean quản lý kết nối cơ sở dữ liệu. Thay vì tạo ra một kết nối mới cho mỗi yêu cầu, một bean Singleton có thể duy trì một kết nối cơ sở dữ liệu duy nhất và chia sẻ nó giữa các thành phần trong ứng dụng.
* Cache: Singleton scope cũng thích hợp cho các bean quản lý cache dữ liệu. Bean Singleton có thể duy trì một bộ nhớ cache và cung cấp dữ liệu cached cho các thành phần khác trong ứng dụng.
* Dịch vụ: Singleton scope thường được sử dụng cho các bean đại diện cho các dịch vụ cung cấp chức năng cho toàn bộ ứng dụng. Ví dụ: bean quản lý email service, logging service, authentication service, etc.
* Thành phần chia sẻ: Singleton scope được sử dụng cho các thành phần được chia sẻ trong toàn bộ ứng dụng. Ví dụ: bean chịu trách nhiệm quản lý danh sách người dùng đăng nhập, bean chịu trách nhiệm quản lý danh sách các phiên làm việc hiện tại của người dùng.

Lưu ý rằng việc sử dụng Singleton scope cần cẩn thận để đảm bảo tính nhất quán và an toàn khi truy cập đồng thời vào dữ liệu chia sẻ. Đồng thời, cần chú ý xử lý các vấn đề liên quan đến đồng bộ hóa và thread safety khi sử dụng Singleton bean trong môi trường đa luồng.

## **Deepdive of Eager and Lazy instantiation of Singleton scope (example15)**

Trong Spring, Singleton là một trong những phạm vi (scope) bean mặc định, có nghĩa là một instance của bean sẽ được tạo ra duy nhất và được chia sẻ trong toàn bộ ứng dụng. Tuy nhiên, có hai cách để thực hiện việc khởi tạo Singleton bean: eager (sẵn sàng) và lazy (lười biếng) instantiation.

Theo mặc định, Spring sẽ khởi tạo Singleton bean: eager (sẵn sàng) trong quá trình khởi động ứng dụng. Điều này được gọi là Eager instantiation.

1. **Eager Instantiation (Khởi tạo sẵn sàng):**

Trong eager instantiation, Singleton bean được tạo ra ngay khi ApplicationContext khởi động hoặc tạo ra một instance của bean nằm trong Singleton scope. Điều này có nghĩa là bean sẽ được tạo ra trước khi nó được yêu cầu lần đầu tiên và sẵn sàng sử dụng ngay từ đầu. Khi eager instantiation được sử dụng, tất cả các Singleton bean sẽ được tạo ra cùng với ApplicationContext và sẵn sàng để sử dụng trong toàn bộ vòng đời của ứng dụng.

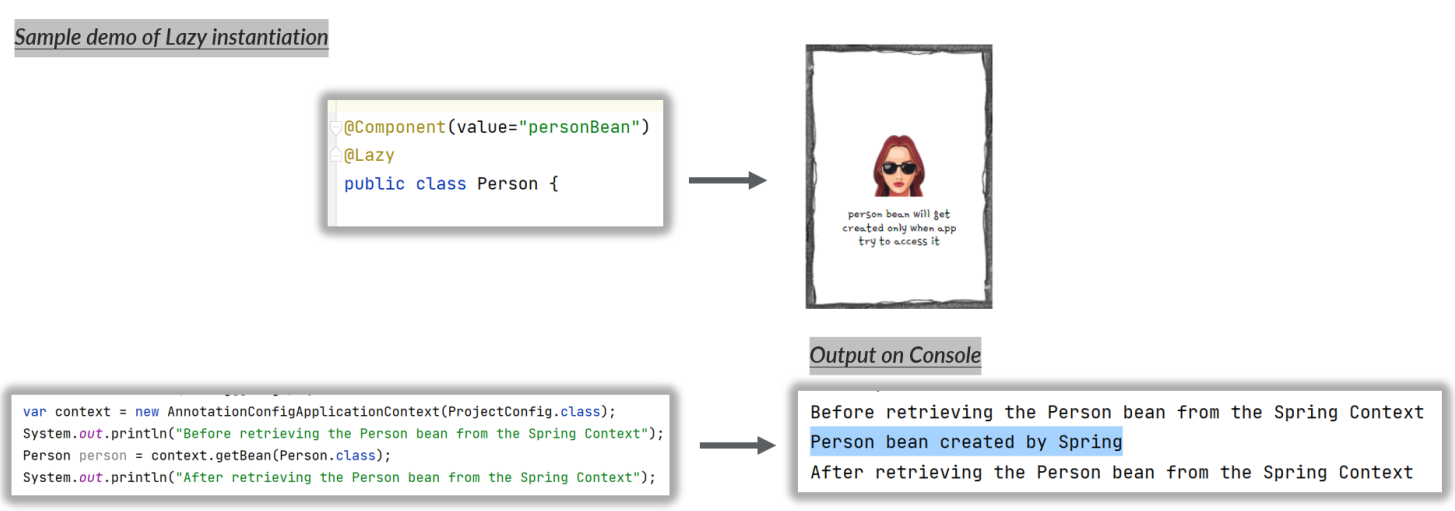
1. **Lazy Instantiation (Khởi tạo lười biếng):**

Trái ngược với eager instantiation, trong lazy instantiation, Singleton bean không được tạo ra ngay khi ApplicationContext khởi động. Thay vào đó, bean chỉ được khởi tạo khi nó được yêu cầu lần đầu tiên. Khi một bean Singleton được yêu cầu, nó sẽ được tạo ra và lưu trữ để sử dụng trong các yêu cầu tiếp theo. Lazy instantiation giúp giảm tải khởi động ban đầu của ứng dụng và chỉ tạo ra các bean cần thiết.

Sự lựa chọn giữa eager và lazy instantiation phụ thuộc vào yêu cầu của ứng dụng và tài nguyên hệ thống. Một số điểm cần lưu ý khi quyết định sử dụng eager hoặc lazy instantiation cho Singleton bean:

* Eager instantiation thích hợp trong trường hợp bean có khối lượng công việc nặng hoặc cần sử dụng ngay từ đầu của ứng dụng.
* Lazy instantiation thích hợp khi tài nguyên hệ thống quan trọng và cần được tiết kiệm, hoặc khi bean có thể không cần thiết trong mọi trường hợp sử dụng.

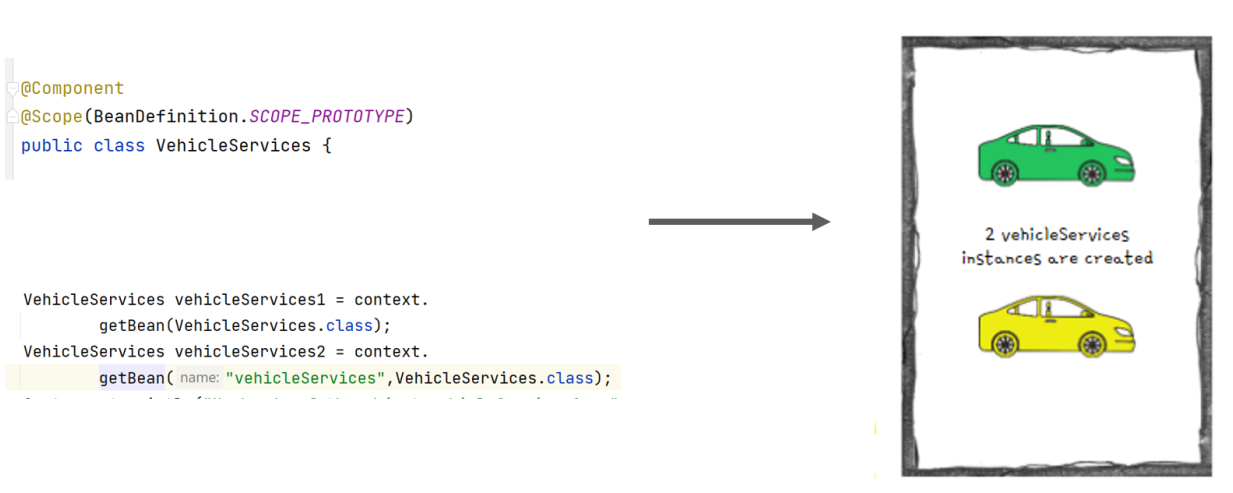
|  |  |
| --- | --- |
| **Eager Instantiation** | **Lazy Instantiation** |
| * Đây là hành vi mặc định bên trong Spring framework * Singleton bean sẽ được tạo trong quá trình khởi động ứng dụng * Máy chủ sẽ không khởi động nếu bean không thể tạo do bất kỳ ngoại lệ phụ thuộc nào * Spring context sẽ chiếm nhiều bộ nhớ nếu chúng ta cố gắng sử dụng eager cho tất cả các bean bên trong ứng dụng * Eager có thể được theo dõi cho tất cả các loại bean được yêu cầu rất phổ biến trong một ứng dụng | * Đây không phải là hành vi mặc định và cần định cấu hình rõ ràng bằng cách sử dụng @Lazy * Singleton bean sẽ được tạo khi ứng dụng cố gắng giới thiệu bean lần đầu tiên * Ứng dụng sẽ đưa ra exception runtime nếu việc tạo bean không thành công do bất kỳ ngoại lệ phụ thuộc nào * Hiệu suất sẽ bị ảnh hưởng nếu chúng ta cố gắng sử dụng lazy cho tất cả các bean bên trong ứng dụng * Lazy có thể theo dõi đối với các bean được sử dụng trong một tình huống rất xa bên trong một ứng dụng |



## **Deepdive of Prototype Bean scope (example16)**

Với prototype scope, mỗi khi chúng ta yêu cầu tham chiếu một bean, Spring sẽ tạo một thể hiện đối tượng mới và cung cấp tương tự.

Prototype scope hiếm khi được sử dụng bên trong các ứng dụng và chúng tôi chỉ có thể sử dụng phạm vi này trong các tình huống mà bean của bạn sẽ thường xuyên thay đổi trạng thái của dữ liệu, điều này sẽ dẫn đến các điều kiện tương tranh trong môi trường đa luồng. Sử dụng prototype scope sẽ không tạo ra bất kỳ race condition nào.



## **Singleton Beans Vs Prototype Beans**

|  |  |
| --- | --- |
| **Singleton Beans** | **Prototype Beans** |
| * Đây là phạm vi mặc định bên trong Spring framework * Cùng một thể hiện đối tượng sẽ được trả về mỗi khi chúng ta giới thiệu một bean bên trong mã * Chúng ta có thể cấu hình để tạo các bean trong quá trình khởi động hoặc khi tham chiếu lần đầu * Các đối tượng bất biến có thể không hoạt động đối với phạm vi Singleton * Scope được sử dụng phổ biến nhất | * Cần cấu hình rõ ràng bằng cách sử dụng @Scope(BeanDefinition.SCOPE\_PROTOTYPE) * Đối tượng mới sẽ được trả về mỗi khi chúng ta giới thiệu một bean bên trong mã * Spring luôn tạo đối tượng mới khi chúng ta cố gắng tham chiếu bean. Không thể eager instantiation * Các đối tượng có thể thay đổi có thể không hoạt động đối với phạm vi nguyên mẫu * Phạm vi rất hiếm khi được sử dụng |

# **Section 5: Aspect Oriented Programming (AOP) inside Spring framework**

## **Introduction to Aspect Oriented Programming (AOP)**

Một **Aspect** (khía cạnh) chỉ đơn giản là một đoạn mã mà Spring framework thực thi khi bạn gọi các phương thức cụ thể bên trong ứng dụng của mình.

Spring AOP cho phép Aspect-Oriented Programming (Lập trình hướng khía cạnh) trong các spring applications. Trong AOP, các khía cạnh cho phép mô đun hóa các mối quan tâm như transaction management, logging hay security xuyên suốt nhiều loại và đối tượng (thường được gọi là mối quan tâm xuyên suốt)

* AOP cung cấp cách để tự động thêm mối quan tâm xuyên suốt trước, sau hoặc xung quanh(before, after, around) logic thực tế bằng cách sử dụng các cấu hình đơn giản.
* AOP giúp phân tách và duy trì nhiều mã liên quan đến logic non-business như logging, auditing, security, transaction management.
* AOP là một mô hình lập trình nhằm mục đích tăng tính mô đun hóa bằng cách cho phép tách biệt các mối quan tâm xuyên suốt. Nó thực hiện điều này bằng cách thêm hành vi bổ sung vào mã hiện có mà không sửa đổi mã

## **AOP Jargons – Thuật ngữ AOP**

Khi chúng ta định nghĩa một Aspect hoặc thực hiện cấu hình, chúng ta cần tuân theo WWW (3Ws)

* WHAT -> Aspect
* WHEN -> Advice
* WHICH -> Pointcut

**WHAT – Cái gì** Mã hoặc logic nào mà chúng tôi muốn Sprint thực thi khi bạn gọi một phương thức cụ thể. Điều này được gọi là Aspect (Khía cạnh).

**WHEN – Khi nào** Spring cần thực thi Aspect đã cho. Ví dụ: trước hoặc sau khi gọi phương thức. Điều này được gọi là Advice.

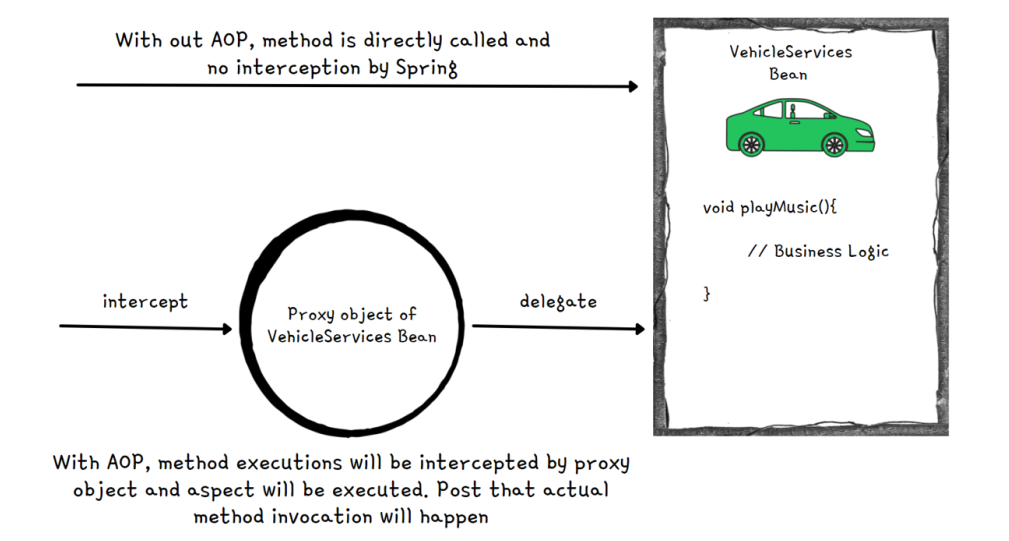
**WHICH - Cài mà** Phương thức bên trong Ứng dụng mà khung đó cần chặn và thực thi Aspect đã cho. Điều này được gọi là Pointcut.

* Join point xác định sự kiện kích hoạt việc thực thi một aspect. Trong Spring, sự kiện này luôn là một cuộc gọi phương thức.
* Target object là bean khai báo phương thức/pointcut bị chặn bởi một khía cạnh.

## **Weaving inside AOP**

Khi triển khai AOP bên trong application của mình bằng Spring framework, nó sẽ chặn(intercept) từng lệnh gọi phương thức và áp dụng logic được xác định trong Aspect.

Nhưng làm thế nào để điều này hoạt động? Spring thực hiện điều này với sự trợ giúp của đối tượng proxy. Vì vậy, chúng tôi cố gắng gọi một phương thức bên trong một bean, Spring thay vì trực tiếp đưa ra tham chiếu của bean thay vào đó, nó sẽ đưa ra một đối tượng proxy sẽ quản lý từng cuộc gọi đến một phương thức và áp dụng logic khía cạnh. Quá trình này được gọi là **Weaving**



## **Type of Advices inside AOP**

Một số loại Advice phổ biến trong AOP bao gồm:

**@Before:** Advice này chạy trước khi thực thi điểm liên kết và có thể được sử dụng để thực hiện một số cài đặt hoặc xác thực bổ sung trước khi thực hiện phương thức thực sự.

**@After:** Advice này chạy sau khi thực thi điểm liên kết, bất kể phương thức thực thi có thành công hay không. Advice sau có thể được sử dụng để thực hiện một số hoạt động dọn dẹp hoặc ghi nhật ký.

**@Around:** Advice này chạy quanh thực thi điểm liên kết và có thể sửa đổi hành vi của phương thức trước và sau khi thực thi. @Around là loại lời khuyên mạnh nhất và có thể được sử dụng để triển khai bộ nhớ cache, retry và các tính năng nâng cao khác.

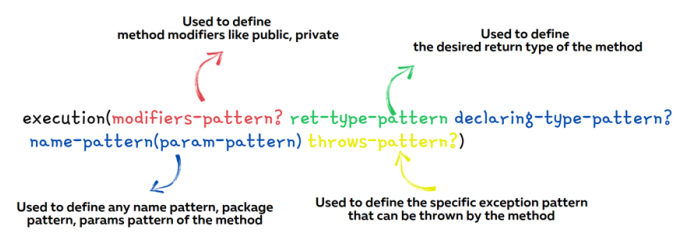
**@AfterReturning:** Advice này chạy sau khi thực thi thành công của điểm liên kết và có thể được sử dụng để thực hiện một số xử lý bổ sung trên giá trị trả về.

**@AfterThrowing:** Advice này chạy sau khi điểm liên kết ném ra một ngoại lệ và có thể được sử dụng để thực hiện một số xử lý lỗi hoặc ghi nhật ký.

## **Configuring Advices inside AOP**

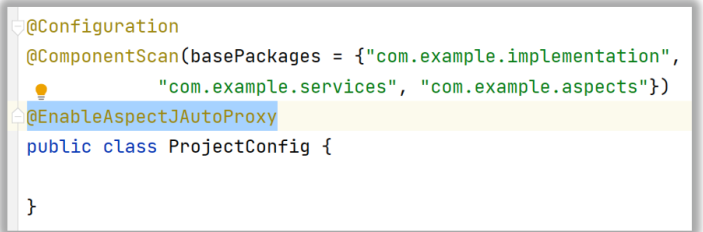
Execution expression là một phương pháp sử dụng trong Aspect-Oriented Programming (AOP) để xác định các điểm cắt (join points) trong mã ứng dụng. Nó cho phép bạn chỉ định các điểm cắt cụ thể mà các khía cạnh (aspects) sẽ được áp dụng.

Trong execution expression, bạn sử dụng biểu thức execution để xác định phương thức hoặc phương thức của một lớp cụ thể mà bạn muốn áp dụng advice. Cú pháp của execution expression thường được sử dụng như sau:



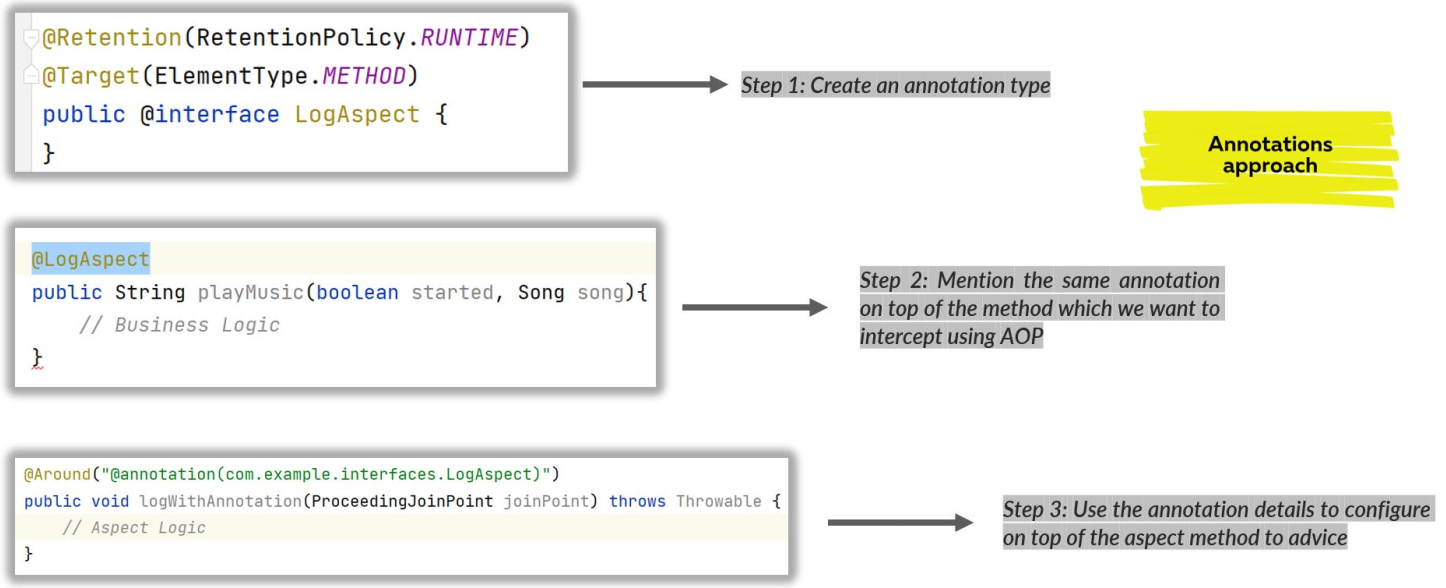
Ở đây, một số thành phần quan trọng của execution expression bao gồm:

* modifiers-pattern: Xác định các phạm vi truy cập của phương thức (ví dụ: public, protected, private).
* return-type-pattern: Xác định kiểu trả về của phương thức.
* declaring-type-pattern: Xác định lớp chứa phương thức.
* method-name-pattern: Xác định tên của phương thức.
* param-pattern: Xác định kiểu và số lượng tham số của phương thức.
* throws-pattern: Xác định các ngoại lệ mà phương thức có thể ném ra.





Ngoài ra, chúng ta có thể sử dụng kiểu Annotation để định cấu hình Advices bên trong AOP. Dưới đây là ba bước mà chúng tôi làm theo cho cùng,



## **Configuring @Around advice (example17)**

### **@Around với annatation**

Để cấu hình @Around advice trong Spring, bạn cần thực hiện các bước sau:

Bước 1: Annotation Configuration:

@Configuration  
@ComponentScan(basePackages = {"com.example.implementation",  
 "com.example.services", "com.example.aspects"})  
@EnableAspectJAutoProxy  
public class ProjectConfig {  
  
}

Bước 2: Tạo một **annatation**:

@Retention(RetentionPolicy.*RUNTIME*)  
@Target(ElementType.*METHOD*)  
public @interface LogAspect {  
}

Bước 3: Tạo một Aspect: Tạo một lớp Aspect chứa @Around advice và các thành phần AOP khác. Bạn có thể sử dụng các annotation như @Aspect để đánh dấu lớp là một Aspect và @Around để đánh dấu phương thức là advice loại @Around.

@Aspect  
@Component  
public class LoggerAspect {  
  
 private Logger logger = Logger.*getLogger*(LoggerAspect.class.getName());  
  
   
 @Around("@annotation(com.example.interfaces.LogAspect)")  
 public void logWithAnnotation(ProceedingJoinPoint joinPoint) throws Throwable {  
 logger.info(joinPoint.toString() + " method execution start");  
 }

}

Bước 4: Tạo một Service và sử dụng annotation

@Component  
public class VehicleServices {  
  
 @LogAspect  
 public String playMusic(){  
 return "";  
 }

}

=> Chạy chương trình

public class Example17 {  
  
 public static void main(String[] args) {  
 var context = new AnnotationConfigApplicationContext(ProjectConfig.class);  
 var vehicleServices = context.getBean(VehicleServices.class);  
 System.*out*.println(vehicleServices.playMusic());  
  
 }  
}

// Output  
Jun 28, 2023 9:41:22 PM com.example.aspects.LoggerAspect logWithAnnotation

INFO: execution(String com.example.services.VehicleServices.playMusic()) method execution start

null

### **@Around annotation áp dụng cho tất cả các phương thức trong tất cả các lớp**

@Aspect  
@Component  
@Order(2)  
public class LoggerAspect {  
  
 private Logger logger = Logger.*getLogger*(LoggerAspect.class.getName());  
  
 @Around("execution(\* com.example.services.\*.\*(..))")  
 public void log(ProceedingJoinPoint joinPoint) throws Throwable {  
 logger.info(joinPoint.getSignature().toString() + " method execution start");  
 Instant start = Instant.*now*();  
 joinPoint.proceed();  
 Instant finish = Instant.*now*();  
 long timeElapsed = Duration.*between*(start, finish).toMillis();  
 logger.info("Time took to execute the method : "+timeElapsed);  
 logger.info(joinPoint.getSignature().toString() + " method execution end");  
 }

}

## **Configuring @Before advice**

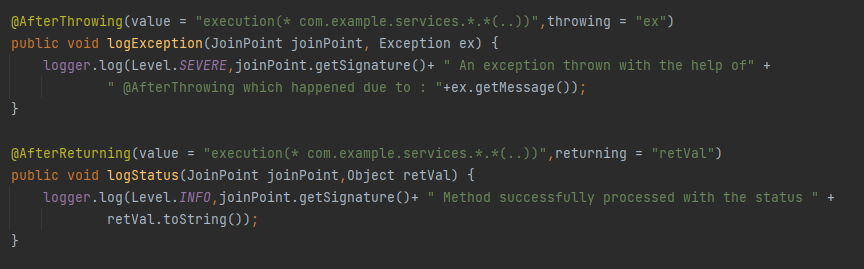
@Aspect  
@Component  
@Order(1)  
public class VehicleStartCheckAspect {  
  
 private Logger logger = Logger.*getLogger*(VehicleStartCheckAspect.class.getName());  
  
 @Before("execution(\* com.example.services.\*.\*(..)) && args(vehicleStarted,..)")  
 public void checkVehicleStarted(JoinPoint joinPoint, boolean vehicleStarted) throws Throwable {  
 if(!vehicleStarted){  
 throw new RuntimeException("Vehicle not started");  
 }  
 }  
}

Đây là một ví dụ về cách sử dụng advice @Before trong AOP để kiểm tra tham số của phương thức được gọi trước khi nó được thực thi. Nó được áp dụng cho tất cả các phương thức trong tất cả các lớp trong gói com.example.services bằng cách sử dụng pointcut expression "execution(\* com.example.services..(..))". Ngoài ra, advice này sử dụng một tham số boolean được đặt tên là vehicleStarted và kiểm tra xem giá trị của tham số này có là true hay không bằng cách sử dụng args(vehicleStarted,..).

Cụ thể, @Before là một loại advice trong AOP được sử dụng để thực thi code trước khi phương thức được gọi. Trong trường hợp này, @Before được kết hợp với pointcut expression "execution(\* com.example.services..(..))", mô tả điểm nơi mà advice sẽ được áp dụng, nghĩa là tất cả các phương thức trong tất cả các lớp trong gói com.example.services. Điều kiện kiểm tra giá trị của tham số vehicleStarted được thêm vào advice bằng cách sử dụng args(vehicleStarted,..), nghĩa là chỉ áp dụng cho các phương thức có tham số đầu tiên là vehicleStarted.

Trong ví dụ này, nếu giá trị của tham số vehicleStarted là false, một ngoại lệ RuntimeException sẽ được ném ra, ngăn chặn phương thức được gọi tiếp tục thực thi và thông báo rằng "Vehicle not started". Nếu giá trị của tham số vehicleStarted là true, phương thức được gọi sẽ tiếp tục thực thi. Đối tượng JoinPoint được truyền vào advice để cho phép advice truy cập thông tin về phương thức và tham số của nó.

## **Configuring @AfterThrowing and @AfterReturning advices**



Đây là một ví dụ về cách sử dụng advice @AfterThrowing và @AfterReturning trong AOP để ghi nhật ký thông tin khi một phương thức trong gói com.example.services ném ra một ngoại lệ hoặc trả về giá trị thành công.

Cụ thể, @AfterThrowing là một loại advice trong AOP được sử dụng để thực thi code sau khi phương thức ném ra ngoại lệ. Trong trường hợp này, @AfterThrowing được kết hợp với pointcut expression "execution(\* com.example.services..(..))", mô tả điểm nơi mà advice sẽ được áp dụng, nghĩa là tất cả các phương thức trong tất cả các lớp trong gói com.example.services. Tham số throwing được sử dụng để truyền đối tượng ngoại lệ được ném ra.

Trong ví dụ này, khi một phương thức ném ra một ngoại lệ, advice sẽ được gọi, và đối tượng ngoại lệ được truyền vào advice. Advice này sử dụng Logger để ghi nhật ký thông tin về ngoại lệ đã xảy ra, bao gồm cả tên phương thức và thông tin về ngoại lệ đó, bằng cách sử dụng phương thức getMessage() để lấy thông tin chi tiết về ngoại lệ.

@AfterReturning là một loại advice trong AOP được sử dụng để thực thi code sau khi phương thức đã trả về giá trị thành công. Trong trường hợp này, @AfterReturning được kết hợp với pointcut expression "execution(\* com.example.services..(..))", mô tả điểm nơi mà advice sẽ được áp dụng, nghĩa là tất cả các phương thức trong tất cả các lớp trong gói com.example.services. Tham số returning được sử dụng để truyền đối tượng giá trị trả về.

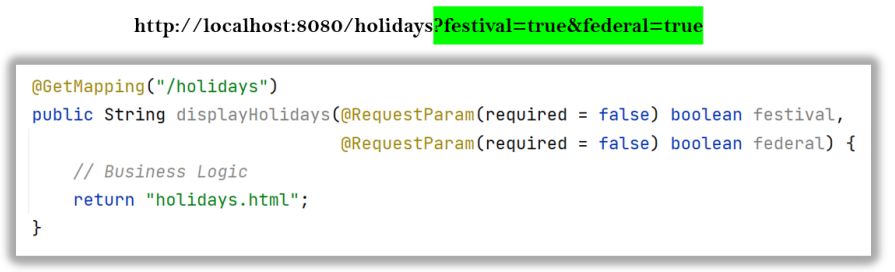
Trong ví dụ này, khi một phương thức trả về giá trị thành công, advice sẽ được gọi, và đối tượng giá trị trả về được truyền vào advice. Advice này sử dụng Logger để ghi nhật ký thông tin về tên phương thức và giá trị trả về, bằng cách sử dụng phương thức toString() để lấy thông tin chi tiết về giá trị trả về.

# **Section 6: Processing Query Params & Path Variables inside Spring**

## **Accepting Query Params using @RequestParam annotation(example22)**

Trong Spring, annotation @RequestParam được sử dụng để ánh xạ tham số truy vấn (query parameters) hoặc form data.

Ví dụ: nếu muốn nhận giá trị tham số từ URL được yêu cầu HTTP GET thì có thể sử dụng chú thích @RequestParam như trong ví dụ bên dưới.

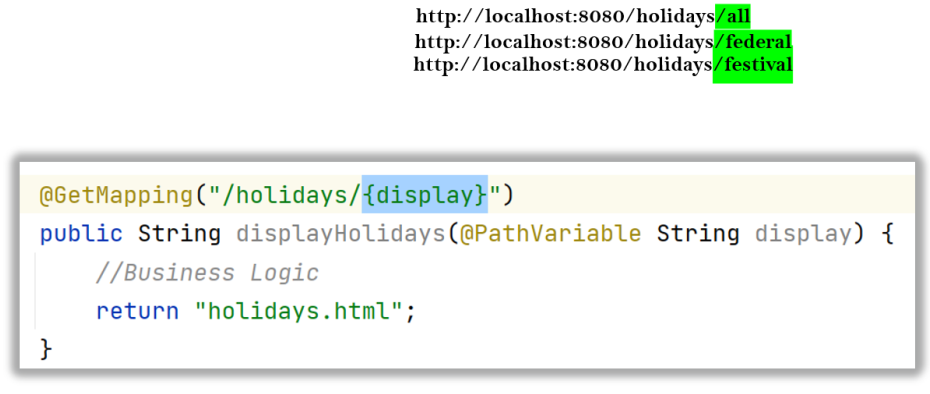


Annatation **@RequestParam** hỗ trợ các thuộc tính như name, required, value, defaultValue. Chúng tôi có thể sử dụng chúng trong ứng dụng của mình dựa trên các yêu cầ.

* value (hoặc name): Xác định tên của tham số truy vấn. Ví dụ: @RequestParam("paramName").
* required: Xác định xem tham số truy vấn có bắt buộc hay không. Mặc định là true. Nếu đặt thành false, và tham số truy vấn không được cung cấp, Spring sẽ gán giá trị null cho tham số đó.
* defaultValue: Xác định giá trị mặc định cho tham số truy vấn khi không được cung cấp. Ví dụ: @RequestParam(value = "paramName", defaultValue = "default").

## **Accepting Query Params using @RequestParam annotation**

Annotation @PathVariable được sử dụng để trích xuất giá trị từ URI. Nó phù hợp nhất cho dịch vụ web RESTful nơi URL chứa một số giá trị. Spring MVC cho phép chúng ta sử dụng nhiều annotation @PathVariable trong cùng một phương thức.



=> Nó cũng hỗ trợ các attribute như name, value, required giống **@RequestParam**

# **Section 7: Validating the input using Java Bean & Hibernate Validators**

## **Introduction to Java Bean Validations**

+ Bean Validation (https://beanvalidation.org/) là tiêu chuẩn để triển khai validation trong hệ sinh thái Java. Nó được tích hợp tốt với Spring và Spring Boot.

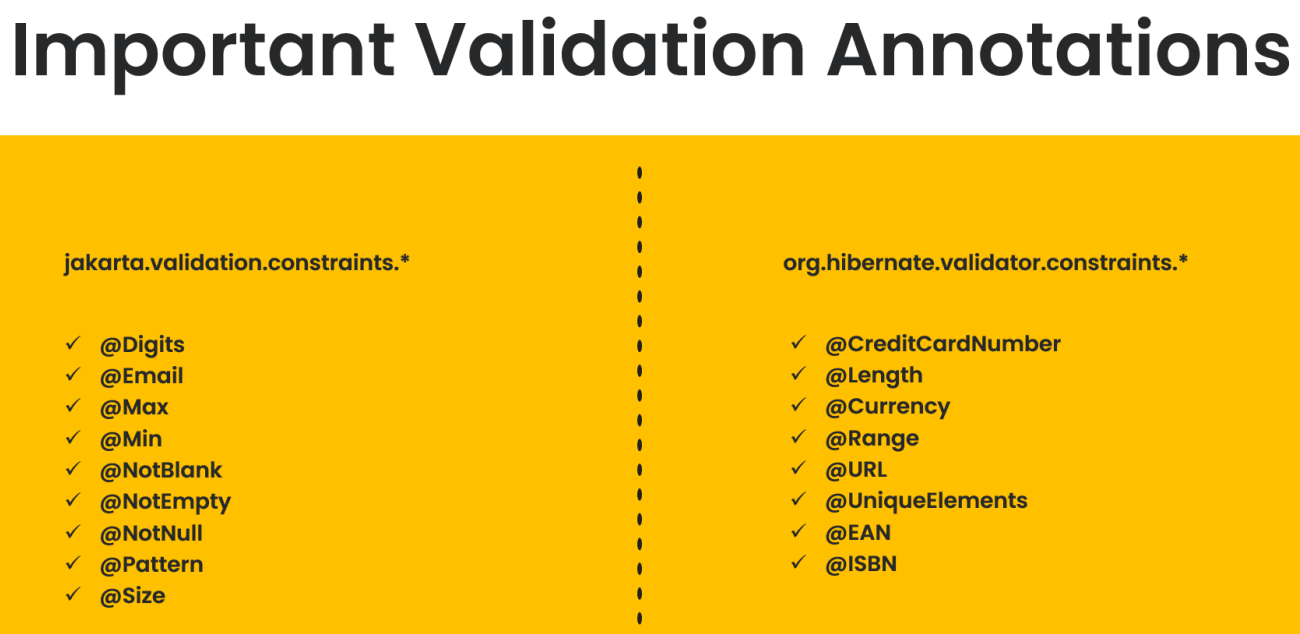
+ Dưới đây là maven dependency mà chúng ta có thể thêm vào để triển khai xác thực Bean trong bất kỳ dự án Spring/SpringBoot nào

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-validation</artifactId>  
</dependency>

* **Bean Validations**hoạt động bằng cách xác định các ràng buộc đối với các trường của một lớp bằng cách chú thích chúng bằng các annotation nhất định
* chúng ta có thể đặt annotation @Valid trên các trường và tham số phương thức để nói với Spring rằng chúng ta muốn một tham số phương thức hoặc trường được validate.
* Dưới đây là các package quan trọng nơi có thể xác định các annotation liên quan đến validate.

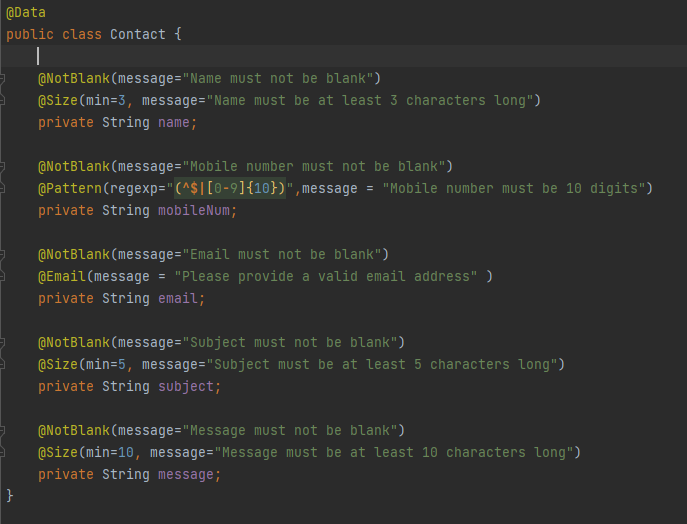
jakarta.validation.constraints.\*

org.hibernate.validator.constraints.\*



## **Adding Bean Validation annotations inside Contact POJO class(example24)**

Khi bạn muốn thực hiện kiểm tra hợp lệ dữ liệu trên một lớp POJO trong Spring, bạn có thể sử dụng các annotation từ Java Bean Validation API. Dưới đây là ví dụ về việc thêm các annotation kiểm tra hợp lệ vào một lớp POJO Contact:



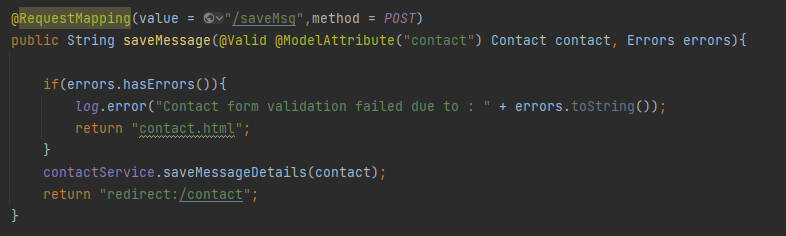
Trong ví dụ trên, chúng ta sử dụng các annotation từ Java Bean Validation API để thực hiện kiểm tra hợp lệ trên các trường của lớp Contact:

* @NotNull: Kiểm tra xem một trường đã cho có phải là null hay không nhưng cho phép các giá trị trống & phần tử bằng 0 bên trong các bộ sưu tập.
* @NotEmpty: Kiểm tra xem một trường đã cho có phải là null hay không và kích thước/độ dài của nó có lớn hơn 0 hay không.
* @NotBlank: Kiểm tra xem một trường đã cho có phải là null không và độ dài đã cắt có lớn hơn 0 hay không.

Khi bạn sử dụng các annotation kiểm tra hợp lệ như trên, bạn cần chắc chắn rằng bạn đã cấu hình một Bean Validator trong ứng dụng Spring của mình để kiểm tra và xử lý các lỗi kiểm tra hợp lệ.

## **Adding Bean Validation related changes inside Web Application**

Trong controller, sử dụng **@Valid** hoặc **@Validated** để áp dụng kiểm tra hợp lệ trên các đối tượng được truyền vào.



Ở ví dụ trên, chúng ta sử dụng @Valid để áp dụng kiểm tra hợp lệ trên đối tượng Contact từ yêu cầu POST, xử lý lỗi (nếu có), lưu thông tin liên hệ và chuyển hướng người dùng đến một trang hiển thị thành công hoặc trang.