**BẢNG ĐÁNH GIÁ ĐIỂM THUYẾT TRÌNH**

**MÔN LẬP TRÌNH JAVA NÂNG CAO**

**LỚP CTK44-PM**

|  |  |
| --- | --- |
| **Thông tin đề tài thuyết trình** | **Thông tin sinh viên thực hiện đánh giá** |
| Tên đề tài: Khái niệm IoC, DI (Dependency Injection) và Spring Bean | MSSV của sinh viên đánh giá: 2012378 |
| Điểm tổng kết: 9.5 Ngày trình bày: 05/05/2023 | Họ tên sinh viên & ký xác nhận: Đặng Ngọc Thắng |
| ***Thông tin sinh viên thuyết trình*** |
| MSSV: 2011379  Họ tên: Triệu Trọng Hậu |

1. **ĐÁNH GIÁ THUYẾT TRÌNH**

**Thang điểm: Kém 0-5; Trung bình: 5-7; Khá 7-8; Tốt 8-9; Xuất sắc: 9-10**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Bảng chấm điểm nhóm thuyết trình** | | | | | | |
| Các chủ đề chính có được xác định? (A) | Slide thuyết trình có thiết kế tốt (B) | Giao tiếp với người nghe bằng ánh mắt (C) | Giao tiếp với người nghe bằng giọng nói (D) | Khả năng trình bày nội dung (E) | Thảo luận & Hỏi đáp (F) | Điểm trung bình = (A+B+C+D+E+F)/6 |
| **10** | **9** | **9** | **10** | **10** | **9** | **9.5** |

1. **BÀI TẬP NẮM KIẾN THỨC**

Dựa vào nội dung vừa trình bày, trả lời các câu hỏi sau:

1. Trình bày tóm tắt nội dung anh/ chị đã nắm được về chủ đề vừa trình bày?
2. Inversion of Control (IoC) / Dependency inversion

* **Inversion of Control (IoC - Đảo ngược điều khiển)** là một nguyên lý thiết kế trong công nghệ phần mềm trong đó các thành phần nó dựa vào để làm việc bị đảo ngược quyền điều khiển khi so sánh với lập trình hướng thủ thục truyền thống.
* Khi áp dụng cho các đối tượng lớp (dịch vụ) có thể gọi nó là Dependency inversion (đảo ngược phụ thuộc), để diễn giải trước tiên cần nắm rõ khái niệm Dependency (phụ thuộc)
* **Thiết kế truyền thống - tham chiếu trực tiếp đến Dependency**

+ Có lớp class A có sử dụng một chức năng (gọi hàm ào đó) của class B, lớp class B lại tham chiếu và gọi các chức năng có trong class C. Ta thấy class A dựa vào class B để hoạt động, class B dựa vào class C. Nếu vậy khi thiết kế theo cách thông thường, viết code thì class A có tham chiếu trực tiếp (cứng) đến class B và trong class B có tham chiếu đến class.

* **Thiết kế theo cách đảo ngược phụ thuộc Inverse Dependency**

+ Cách viết code này, ở thời điểm thực thi thì class A vẫn gọi được hàm có class B, class B vẫn gọi hàm có class C nghĩa là kết quả không đổi. Tuy nhiên, khi thiết kế ở thời điểm viết code (trong code) class A không tham chiếu trực tiếp đến class B mà nó lại sử dụng interface (hoặc lớp abstract) mà classB triển khai. Điều này dẫn tới sự phụ thuộc lỏng lẻo giữa classA và classB

+ Khi thực thi, classB có thể được thay thế bởi bất kỳ lớp nào triển khai từ giao điện interface B, classB cụ thể mà classA sử dụng được quyết định và điểu khiển bởi interface B (điều này có nghĩa tại sao gọi là đảo ngược phụ thuộc)

1. **Dependency Injection (DI)**

* Dependency Injection là một Design Pattern hay hiểu đơn giản là một phương pháp lập trình
* Là một thiết kế để hiệu quả cao hơn khi code
* Vấn đề:

+ Các class không nên phụ thuộc vào các kế thừa cấp thấp

+ Nên phụ thuộc vào Abstraction (lớp trừu tượng)

* DI là việc các Object nên phụ thuộc vào các Abstract class và thể hiện chi tiết của nó sẽ được Inject vào đối tượng lúc Runtime
* Có 3 cách để Inject Dependency vào một đối tượng:

+ Constructor Injection

+ Setter Injection

+ Interface Injection

* Dependency injection (DI) nó là một hình thức cụ thể của Inverse of Control (Dependency Inverse) đã nói ở trên. DI thiết kế sao cho các dependency (phụ thuộc) của một đối tượng CÓ THỂ được đưa vào, tiêm vào đối tượng đó (Injection) khi nó cần tới (khi đối tượng khởi tạo). Cụ thể cần làm:
* Xây dựng các lớp (dịch vụ) có sự phụ thuộc nhau một cách lỏng lẻo, và dependency có thể tiêm vào đối tượng (injection) - thường qua phương thức khởi tạo constructor, property, setter
* Xây dựng được một thư viện có thể tự động tạo ra các đối tượng, các dependency tiêm vào đối tượng đó, thường là áp dụng kỹ thuật Reflection của C# (xem thêm lớp type): Thường là thư viện này quá phức tạp để tự phát triển nên có thể sử dụng các thư viện có sẵn.
* Giả sử có lớp Car có chức năng (phương thức) Beep() - để phát ra tiếng còi xe, mà để phát ra tiếng còi - nó lại dựa vào vào lớp Horn chuyên tạo ra tiếng còi - lúc đó ta nói lớp Car có một phụ thuộc (dependency Horn) là lớp Horn, Horn là dependency của Car.
* Muốn lớp Car hoạt động thì nó phải có đối tượng (dịch vụ) từ Horn. Vậy khi thiết kế, thường có hai cách:
* Trong lớp Car thiết kế code mà nó phụ thuộc cứng vào lớp Horn - tự khởi tạo Horn, cách thiết kế này không có khả năng áp dụng kỹ thuật DI
* Trong lớp Car, dependency Horn không do Car trực tiếp khởi tạo mà nó được đưa vào qua phương thức khởi tạo, qua setter, qua gán property. Các thiết kế này linh hoạt và có KHẢ NĂNG để áp dụng DI

1. **Spring Bean**

* Trong Spring, các đối tượng tạo thành xương sống của ứng dụng của bạn và được quản lý bởi Spring IoC container được gọi là các bean. Bean là một đối tượng được khởi tạo, lắp ráp và quản lý bởi bộ chứa Spring IoC.
* Trong Spring Boot, bạn có thể khai báo các Spring Bean bằng cách sử dụng các annotation như @Component, @Service, @Repository, @Controller, hoặc @Configuration. Sau đó, Spring Boot sẽ tự động quản lý việc khởi tạo và phân phối các Bean này đến các thành phần khác trong ứng dụng của bạn.
* Dưới đây là danh sách các annotation phổ biến được sử dụng để khai báo Spring Bean trong Spring Framework:

**+ @Component**: Được sử dụng để đánh dấu một class là một Spring Bean.

**+ @Service**: Được sử dụng để đánh dấu một class là một Service Bean.

**+ @Repository**: Được sử dụng để đánh dấu một class là một Repository Bean.

**+ @Controller**: Được sử dụng để đánh dấu một class là một Controller Bean.

**+ @Configuration**: Được sử dụng để đánh dấu một class là một Configuration Bean.

**+ @Bean**: Được sử dụng để đánh dấu một method là một factory method để tạo ra một Bean.

**+ @Scope**: Được sử dụng để chỉ định phạm vi của Bean.

**+ @Autowired**: Được sử dụng để inject các Bean dependencies vào Bean hiện tại.

**+ @Value**: Được sử dụng để inject các giá trị từ các property file hoặc Environment vào Bean.

* **Bean Scope:**

**+ Singleton**: container chỉ khởi tạo 1 instance của bean và trả về chính nó nếu như có yêu cầu.

**+ Prototype**: mỗi khi có yêu cầu thì container sẽ tạo ra một instance mới và trả về.

**+ Request**: khởi tạo instance cho một HTTP Request

**+ Session**: khởi tạo instance cho một HTTP Session

**+ Application**: khởi tạo instance cho một vòng đời của ServletContext

**+ WebSocket**: khởi tạo instance cho một Websocket Session

* **Vòng đời của Bean**

Vòng đời của bean bao gồm các bước sau:

+ Bean Definition : khởi tạo bean thông qua sử dụng Annotation hoặc XML

+ Bean Instantiation : Spring khởi tạo các đối tượng Bean giống như khởi tạo đối tượng Java thông thường và đưa nó vào ApplicationContext

+ Populating Bean properties : Spring thực hiện scan các bean thực thi các Aware interfaces và thực hiện set các giá vào các property như id, scope và giá trị mặc định như khai báo của bean đấy

+ Pre-Initialization : Các phương thức postProcessBeforeInitialization() bắt đầu thực thi và @PostConstruct thực thi sau ngay nó

+ AfterPropertiesSet : Spring thực thi các phương thức afterPropertiesSet() của beans mà có implement InitializingBean

+ Custom Initialization : Spring kích hoạt các method khởi tạo với các thuộc tính được define ở trong initMethod trong @Bean annotations

+ Post-initialization : BeanPostProcessors của Spring hoạt động lần thứ 2. Lần này nó kích hoạt các phương thức postProcessAfterInitialization()

+ Ready : các Bean đã được khởi tạo và inject vào trong các dependencies

+ Pre-Destroy : Spring kích hoạt @PreDestroy annotated methods ở bước này

+ Destroy : Spring thực thi the destroy() methods

+ Custom Destruction : chúc ta có thể tuỳ chỉnh các thời điểm huỷ bằng thuộc tính destroyMethod ở trong @Bean annotation và Spring sẽ chạy nó trong giai đoạn cuối.

1. Anh/ chị hãy đưa ra các nhận xét và góp ý cho sinh viên vừa trình bày?

* Bạn trình bày quá đỉnh <3, không có nhận xét