Buổi 2: Tư duy OOP & Spring Core (DI/IoC)

Ôn lại buổi 1

Trước khi học IoC/DI, hãy chắc rằng bạn đã hoàn tất các điểm mấu chốt của Buổi 1:

- Đã cài đặt đầy đủ JDK 17, VS Code với các extension Java/Spring, Git và Postman/Thunder Client để không bị gián đoạn khi thực hành injection.
- Đã clone project `taxi-booking-backend`, tạo branch cá nhân, chạy ứng dụng thử trong VS Code và quen với Terminal tích hợp.
- Đã hiểu tối thiểu quy trình Git (add/commit/push) và có repo GitHub sẵn sàng để lưu các bài tập DI/IoC hôm nay.

Nếu gặp trục trặc, hãy quay lại checklist của Buổi 1 để xử lý trước khi tiếp tục.

Kiến thức

1. Ôn tập Interface và Abstract Class trong Java

1.1. Interface là gì?

Giải thích: Interface giống như một bản hợp đồng. Khi một class "ký" hợp đồng (implements interface), class đó phải thực hiện tất cả các điều khoản (methods) trong hợp đồng đó.

Ví dụ đơn giản:

```
// Định nghĩa hợp đồng: "Ai muốn làm Coach phải có method train()"
public interface Coach {
    // Phương thức train() - chưa có code, chỉ là khai báo
    // Các class implement PHẢI thực hiện method này
    String train();

    // Default method - ĐÃ CÓ CODE (từ Java 8+)
    // Các class implement KHÔNG CẦN phải thực hiện, nhưng có thể override
nếu muốn
    default String getDailyMotivation() {
        return "Hôm nay là một ngày tuyệt vời để luyện tập!";
    }

    // Static method - ĐÃ CÓ CODE (từ Java 8+)
    // Có thể gọi trực tiếp từ interface, không cần tạo đối tượng
    static String getCoachInfo() {
        return "Đây là thông tin về Coach interface";
    }
}

// Class TennisCoach "ký" hợp đồng Coach
public class TennisCoach implements Coach {
    // Phải thực hiện method train() vì đã ký hợp đồng
```

```
    @Override
    public String train() {
        return "Đánh bóng tennis 100 lần!";
    }

    // Có thể override default method nếu muốn (không bắt buộc)
    @Override
    public String getDailyMotivation() {
        return "Hôm nay hãy đánh bóng tennis thật tốt!";
    }
}

// Class FootballCoach cũng "ký" hợp đồng Coach
public class FootballCoach implements Coach {
    // Cũng phải thực hiện method train()
    @Override
    public String train() {
        return "Chạy 5 vòng sân!";
    }

    // Không override getDailyMotivation() → Sẽ dùng code mặc định từ
interface
}

// Sử dụng:
public class MainApp {
    public static void main(String[] args) {
        // Tạo đối tượng
        Coach tennisCoach = new TennisCoach();
        Coach footballCoach = new FootballCoach();

        // Gọi method bắt buộc (phải implement)
        System.out.println(tennisCoach.train());  // "Đánh bóng tennis 100
lần!"
        System.out.println(footballCoach.train());  // "Chạy 5 vòng sân!"

        // Gọi default method (đã có code trong interface)
        System.out.println(tennisCoach.getDailyMotivation());
        // "Hôm nay hãy đánh bóng tennis thật tốt!" (đã override)

        System.out.println(footballCoach.getDailyMotivation());
        // "Hôm nay là một ngày tuyệt vời để luyện tập!" (dùng code mặc
định)

        // Gọi static method (gọi trực tiếp từ interface, không cần đối
tượng)
        System.out.println(Coach.getCoachInfo());
        // "Đây là thông tin về Coach interface"
    }
}
```

Tại sao dùng Interface?

- Cho phép nhiều class khác nhau có cùng "hợp đồng" nhưng thực hiện khác nhau
- Giúp code linh hoạt: Bạn có thể thay đổi TennisCoach thành FootballCoach mà không cần sửa code nơi sử dụng

Lưu ý quan trọng về Interface:

1. **Interface không thể khởi tạo (không thể dùng `new`):**

```
// ❌ SAI - Không thể làm như này
Coach coach = new Coach();  // Lỗi! Interface không thể khởi tạo

// ✅ ĐÚNG - Phải tạo đối tượng từ class implement
Coach coach = new TennisCoach();  // OK
```

2. **Một class có thể implement nhiều Interface (multiple inheritance):**

```
public interface Coach {
    String train();
}

public interface Teacher {
    String teach();
}

// Class có thể implement nhiều interface
public class TennisCoach implements Coach, Teacher {
    @Override
    public String train() {
        return "Đánh bóng tennis!";
    }

    @Override
    public String teach() {
        return "Dạy kỹ thuật tennis!";
    }
}
```

3. **Interface có thể extend Interface khác:**

```
public interface Coach {
    String train();
}

// Interface có thể kế thừa từ interface khác
public interface ProfessionalCoach extends Coach {
    String getCertification();
}
```

```java
// Class implement ProfessionalCoach phải implement CẢ HAI method
public class TennisCoach implements ProfessionalCoach {
    @Override
    public String train() {
        return "Đánh bóng tennis!";
    }

    @Override
    public String getCertification() {
        return "ITF Certified";
    }
}
```

4. **Tất cả method trong Interface mặc định là `public`:**

```java
public interface Coach {
    // Không cần khai báo "public" - mặc định đã là public
    String train();  // Tương đương với: public String train();

    // Có thể khai báo rõ ràng, nhưng không cần thiết
    public String getDailyMotivation();  // Cũng được, nhưng dư thừa
}
```

5. **Tất cả field trong Interface mặc định là `public static final` (constant):**

```java
public interface Coach {
    // Mặc định là: public static final int MAX_TRAINING_HOURS = 8;
    int MAX_TRAINING_HOURS = 8;  // Không thể thay đổi giá trị (constant)

    // Có thể khai báo rõ ràng
    public static final String DEFAULT_MESSAGE = "Let's train!";
}
```

6. **Default method và Static method chỉ có từ Java 8+:**

- Nếu dùng Java 7 trở xuống, interface chỉ có thể có abstract method
- Từ Java 8+: Có thêm default method và static method
- Từ Java 9+: Có thêm private method trong interface

1.2. Abstract Class là gì?

Giải thích: Abstract Class giống như một bản thiết kế nhà chưa hoàn chỉnh. Nó có một số phần đã xây xong (method có code), một số phần chỉ có bản vẽ (abstract method - chưa có code).

Ví dụ đơn giản:

```java
// Abstract class: Bản thiết kế chung cho tất cả Vehicle
public abstract class Vehicle {
    // Method đã có code (concrete method) - CÓ THỂ override
    public void startEngine() {
        System.out.println("Khởi động động cơ...");
    }

    // Abstract method: Chỉ có khai báo, chưa có code
    // Mỗi loại xe PHẢI implement method này
    public abstract void move();
}

// Class Car kế thừa từ Vehicle
public class Car extends Vehicle {
    // Phải implement method move() vì nó là abstract
    @Override
    public void move() {
        System.out.println("Xe hơi chạy trên đường!");
    }

    // CÓ THỂ override concrete method startEngine() nếu muốn
    // Nếu không override, sẽ dùng code mặc định từ Abstract Class
    @Override
    public void startEngine() {
        System.out.println("Xe hơi: Bấm nút khởi động...");
        System.out.println("Vroom vroom!");
    }
}

// Class Boat cũng kế thừa từ Vehicle
public class Boat extends Vehicle {
    @Override
    public void move() {
        System.out.println("Thuyền chạy trên nước!");
    }

    // Không override startEngine() → Sẽ dùng code mặc định từ Abstract
Class
    // Khi gọi boat.startEngine() → Sẽ in "Khởi động động cơ..."
}

// Sử dụng:
public class MainApp {
    public static void main(String[] args) {
        Car car = new Car();
        car.startEngine();  // "Xe hơi: Bấm nút khởi động..." + "Vroom
vroom!" (đã override)
        car.move();         // "Xe hơi chạy trên đường!"

        Boat boat = new Boat();
        boat.startEngine(); // "Khởi động động cơ..." (dùng code mặc định)
        boat.move();       // "Thuyền chạy trên nước!"
```

```
    }
}
```

Lưu ý quan trọng:

- ✅ **Concrete method (method có code) trong Abstract Class CÓ THỂ override** - Class con có thể override để thay đổi hành vi
- ✅ **Abstract method PHẢI implement** - Class con bắt buộc phải có code cho method này
- ✅ **Nếu không override concrete method** - Class con sẽ dùng code mặc định từ Abstract Class

---

Abstract Class có thể implement Interface:

```
// Interface định nghĩa hợp đồng
public interface Drivable {
    void drive();
}

// Abstract Class vừa kế thừa từ class khác, vừa implement Interface
public abstract class Vehicle implements Drivable {
    // Method đã có code
    public void startEngine() {
        System.out.println("Khởi động động cơ...");
    }

    // Abstract method
    public abstract void move();

    // Có thể implement method từ Interface ngay trong Abstract Class
    // Hoặc để class con implement
    @Override
    public void drive() {
        startEngine();
        move();
        System.out.println("Đang lái xe...");
    }
}

// Class Car kế thừa từ Vehicle (đã implement Drivable)
public class Car extends Vehicle {
    @Override
    public void move() {
        System.out.println("Xe hơi chạy trên đường!");
    }

    // Có thể override method drive() nếu muốn
    // Nếu không, sẽ dùng code từ Abstract Class Vehicle
}

// Sử dụng:
public class MainApp {
```

```
    public static void main(String[] args) {
        Car car = new Car();
        car.drive();  // Gọi method từ Interface (đã implement trong
Abstract Class)
        // Output:
        // "Khởi động động cơ..."
        // "Xe hơi chạy trên đường!"
        // "Đang lái xe..."
    }
}
```

Tóm tắt:

- ✅ **Abstract Class CÓ THỂ implement Interface** - Dùng `implements` giống class thường
- ✅ **Abstract Class có thể implement method từ Interface** - Hoặc để class con implement
- ✅ **Class con kế thừa Abstract Class** - Tự động có tất cả method từ Interface (nếu Abstract Class đã implement)

So sánh Interface vs Abstract Class:

Đặc điểm	Interface	Abstract Class
Có thể có method có code?	Có (từ Java 8+): default method, static method	Có: concrete method
Có thể có method chưa có code?	Có: abstract method	Có: abstract method
Class con có thể override method có code?	Có (override default method)	Có (override concrete method)
Có thể implement Interface?	Không (interface không thể implement interface khác, chỉ extend)	Có (dùng `implements`)
Một class có thể implement/kế thừa bao nhiêu?	Nhiều (multiple inheritance)	Chỉ 1 (single inheritance)
Khi nào dùng?	Khi muốn nhiều class có cùng "hợp đồng"	Khi muốn chia sẻ code chung giữa các class

1.3. Sự khác biệt giữa `extends` và `implements`

Giải thích đơn giản:

- `extends` = "Kế thừa từ" - Giống như con kế thừa từ cha, nhận tất cả tài sản (code) của cha
- `implements` = "Thực hiện hợp đồng" - Giống như ký hợp đồng, phải thực hiện các điều khoản (methods) trong hợp đồng

Bảng so sánh:

Tiêu chí	`extends` (Kế thừa)	`implements` (Thực hiện)
Dùng với gì?	Class kế thừa từ Class hoặc Abstract Class	Class thực hiện Interface
Có thể kế thừa/thực hiện bao nhiêu?	Chỉ 1 (single inheritance)	Nhiều (multiple inheritance)
Nhận được gì?	Tất cả code (method, field) từ class cha	Chỉ nhận "hợp đồng" (phải tự viết code)
Có thể dùng với Interface?	Interface có thể `extends` Interface khác	Class `implements` Interface
Từ khóa	`extends`	`implements`

Ví dụ cụ thể:

```
// ========== EXTENDS (Kế thừa) ==========

// Class cha (có code sẵn)
public class Animal {
    // Method đã có code
    public void eat() {
        System.out.println("Động vật đang ăn...");
    }

    // Method đã có code
    public void sleep() {
        System.out.println("Động vật đang ngủ...");
    }
}

// Class con KẾ THỪA từ Animal
public class Dog extends Animal {
    // Dog tự động có method eat() và sleep() từ Animal
    // Không cần viết lại, có thể dùng luôn

    // Có thể thêm method riêng
    public void bark() {
        System.out.println("Gâu gâu!");
    }
}

// Sử dụng:
Dog dog = new Dog();
dog.eat();   // "Động vật đang ăn..." (dùng code từ Animal)
dog.sleep(); // "Động vật đang ngủ..." (dùng code từ Animal)
dog.bark();  // "Gâu gâu!" (method riêng của Dog)

// ========== IMPLEMENTS (Thực hiện) ==========

// Interface (chỉ có hợp đồng, chưa có code)
```

```java
public interface Flyable {
    // Chỉ có khai báo, chưa có code
    void fly();
}

// Class thực hiện Interface
public class Bird implements Flyable {
    // PHẢI tự viết code cho method fly()
    @Override
    public void fly() {
        System.out.println("Chim đang bay!");
    }
}

// Sử dụng:
Bird bird = new Bird();
bird.fly();  // "Chim đang bay!" (code tự viết)

// ========== KẾT HỢP CẢ HAI ==========

// Class có thể vừa extends, vừa implements
public class Eagle extends Animal implements Flyable {
    // Kế thừa eat() và sleep() từ Animal (không cần viết lại)

    // Phải implement fly() từ Interface Flyable
    @Override
    public void fly() {
        System.out.println("Đại bàng đang bay cao!");
    }

    // Có thể thêm method riêng
    public void hunt() {
        System.out.println("Đại bàng đang săn mồi!");
    }
}

// Sử dụng:
Eagle eagle = new Eagle();
eagle.eat();  // "Động vật đang ăn..." (kế thừa từ Animal)
eagle.sleep(); // "Động vật đang ngủ..." (kế thừa từ Animal)
eagle.fly();   // "Đại bàng đang bay cao!" (implement từ Flyable)
eagle.hunt();  // "Đại bàng đang săn mồi!" (method riêng)
```

Quy tắc quan trọng:

1. **Một class chỉ có thể `extends` 1 class:**

```java
// ❌ SAI - Không thể extends nhiều class
public class Dog extends Animal, Mammal {  // Lỗi!

// ✅ ĐÚNG - Chỉ extends 1 class
public class Dog extends Animal {  // OK
```

2. **Một class có thể `implements` nhiều Interface:**

```
// ✅ ĐÚNG - Có thể implements nhiều interface
public class Eagle extends Animal implements Flyable, Swimmable, Runnable
{
    // Phải implement tất cả methods từ các interface
}
```

3. **Thứ tự: `extends` trước, `implements` sau:**

```
// ✅ ĐÚNG - extends trước, implements sau
public class Eagle extends Animal implements Flyable {
}

// ❌ SAI - Không thể đảo ngược
public class Eagle implements Flyable extends Animal {  // Lỗi!
```

4. **Interface có thể `extends` Interface khác:**

```
// Interface có thể extends nhiều interface khác
public interface Flyable {
    void fly();
}

public interface Swimmable {
    void swim();
}

// Interface có thể extends nhiều interface
public interface SuperAnimal extends Flyable, Swimmable {
    void run();
}

// Class implement SuperAnimal phải implement TẤT CẢ methods
public class Duck implements SuperAnimal {
    @Override
    public void fly() { System.out.println("Bay!"); }

    @Override
    public void swim() { System.out.println("Bơi!"); }

    @Override
    public void run() { System.out.println("Chạy!"); }
}
```

Tóm tắt dễ nhớ:

- `extends` = "Nhận tài sản" (code) từ 1 class cha
- `implements` = "Ký hợp đồng" (phải tự viết code) từ nhiều interface
- **Có thể dùng cả hai:** `class A extends B implements C, D`

---

2. Inversion of Control (IoC) Container - "Kho chứa" Bean

2.1. IoC là gì?

Giải thích đơn giản: Inversion of Control (Đảo ngược điều khiển) có nghĩa là bạn không cần tự tạo đối tượng nữa. Thay vào đó, Spring Framework sẽ tự động tạo và quản lý các đối tượng cho bạn.

Ví dụ thực tế dễ hiểu:

Hãy tưởng tượng bạn đang xây một ngôi nhà:

- **Cách cũ (không có IoC):** Bạn phải tự đi mua gạch, tự trộn xi măng, tự xây từng viên gạch. Bạn phải làm TẤT CẢ mọi thứ.
- **Cách mới (có IoC):** Bạn chỉ cần nói "Tôi muốn một ngôi nhà", và có một "đội thợ" (Spring Container) sẽ tự động mua nguyên liệu, xây nhà, và đưa cho bạn ngôi nhà hoàn chỉnh.

Trong code Java:

Không có IoC (cách cũ - bạn phải tự làm tất cả):

```
public class MainApp {
    public static void main(String[] args) {
        // Bạn phải tự tạo đối tượng TennisCoach
        // Giống như tự đi mua gạch, tự xây nhà
        Coach tennisCoach = new TennisCoach();
        System.out.println(tennisCoach.train());
    }
}
```

Có IoC (Spring - Spring tự động làm cho bạn):

```
@SpringBootApplication
public class MainApp {
    public static void main(String[] args) {
        // Spring Container tự động tạo và quản lý đối tượng
        // Bạn không cần tự tạo nữa!
        // Giống như chỉ cần nói "Tôi muốn Coach", Spring sẽ tự động tạo
và đưa cho bạn
        SpringApplication.run(MainApp.class, args);
    }
}
```

Tóm lại: IoC = Bạn không cần tự tạo đối tượng, Spring sẽ tự động tạo và quản lý cho bạn.

2.2. IoC Container là gì?

Giải thích đơn giản: IoC Container (hay Spring Container) giống như một "kho chứa đồ" thông minh. Khi ứng dụng khởi động, nó sẽ:

1. **Tự động tạo các đối tượng** (gọi là Bean) và lưu vào "kho"
2. **Quản lý các đối tượng** (khi nào tạo, khi nào xóa)
3. **Tự động đưa đối tượng** vào nơi cần dùng (khi bạn yêu cầu)

Ví dụ cụ thể với Taxi Booking System:

Hãy tưởng tượng bạn có một nhà kho (IoC Container) chứa các "công cụ" (Bean) như: `BookingService`, `UserRepository`, `EmailService`...

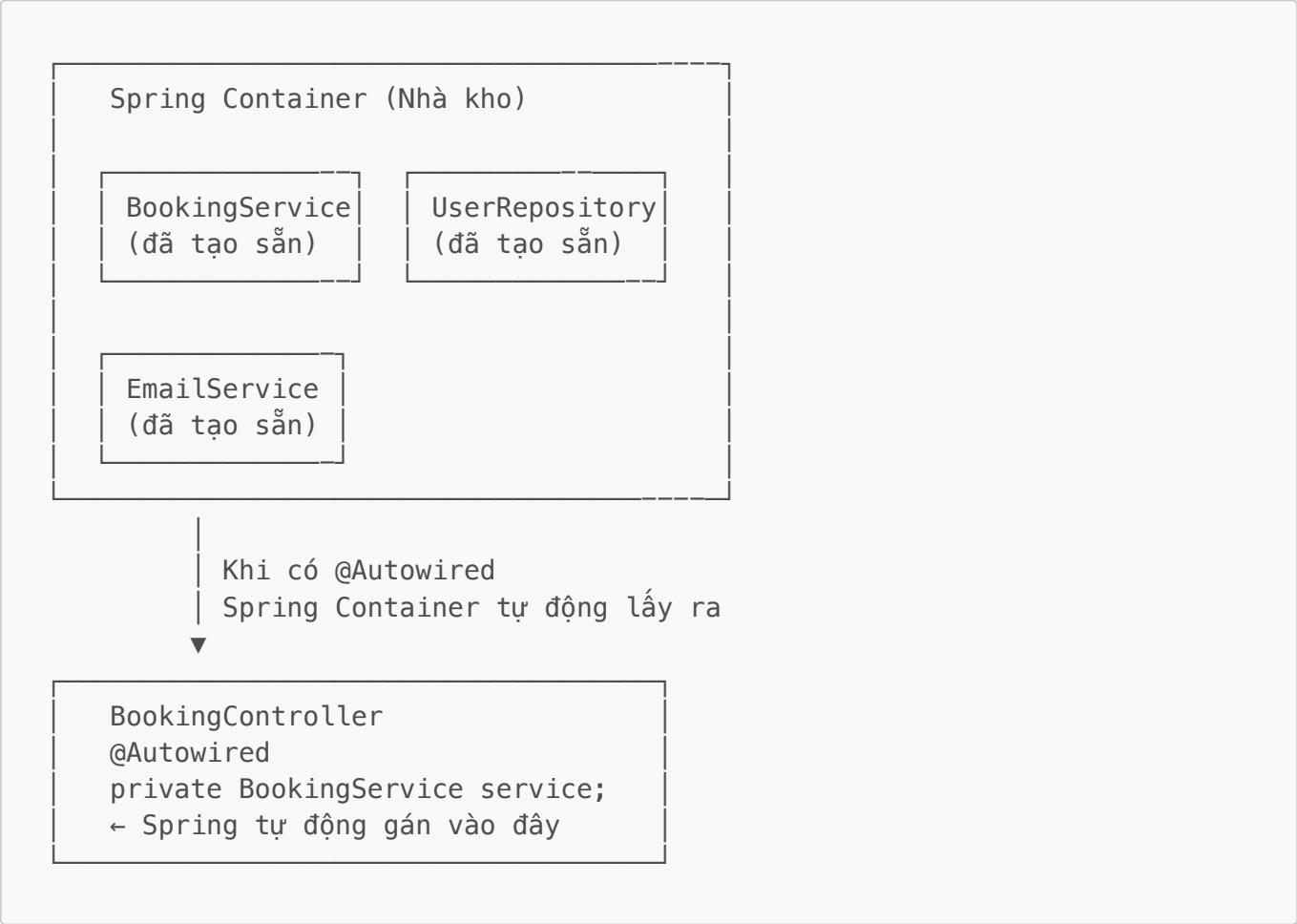
```
// Bước 1: Đánh dấu class là một Bean (để Spring Container quản lý)
// Giống như đặt một nhãn "Lưu vào kho" lên đồ vật
@Component
public class BookingService {
    // Class này sẽ được Spring Container tự động tạo và lưu vào "kho"

    public void createBooking() {
        System.out.println("Tạo booking...");
    }
}

// Bước 2: Sử dụng Bean trong class khác
@Component
public class BookingController {
    // Bạn muốn dùng BookingService, nhưng không cần tự tạo
    // Chỉ cần "yêu cầu" Spring Container đưa cho bạn
    @Autowired
    private BookingService bookingService;
    // Spring Container sẽ tự động:
    // 1. Tìm BookingService trong "kho"
    // 2. Lấy ra và gán vào biến bookingService này

    public void handleRequest() {
        // Bây giờ bạn có thể dùng bookingService ngay
        bookingService.createBooking();
    }
}
```

Quy trình hoạt động chi tiết (dễ hiểu):

1. **Ứng dụng khởi động:**
    - Spring Container được tạo ra (giống như mở cửa nhà kho)

2. **Spring Container quét tất cả class:**
    - Tìm các class có `@Component`, `@Service`, `@Repository`
    - Tạo đối tượng từ các class đó
    - Lưu vào "kho" (giống như sắp xếp đồ vào kệ)
3. **Khi gặp `@Autowired`:**
    - Spring Container hiểu: "Ồ, class này cần một đối tượng!"
    - Tìm trong "kho" xem có đối tượng nào phù hợp không
    - Lấy ra và tự động gán vào biến (giống như lấy đồ từ kệ và đưa cho bạn)

Ví dụ minh họa bằng hình ảnh:



Tóm lại: IoC Container = Một "nhà kho" tự động tạo, lưu trữ, và đưa các đối tượng (Bean) cho bạn khi cần.

---

3. Dependency Injection (DI) - Tự động cung cấp đối tượng

3.1. Dependency Injection là gì?

Giải thích đơn giản: Dependency Injection có nghĩa là "tự động cung cấp đối tượng cần thiết". Thay vì bạn phải tự tạo đối tượng, Spring Framework sẽ tự động tạo và đưa vào (gán vào) biến cho bạn.

Ví dụ thực tế dễ hiểu:

Hãy tưởng tượng bạn đang nấu ăn:

- **Cách cũ (không có DI):** Bạn muốn nấu mì → Bạn phải tự đi mua mì, tự mua thịt, tự mua rau, tự chuẩn bị tất cả nguyên liệu
- **Cách mới (có DI):** Bạn chỉ cần nói "Tôi muốn nấu mì", và có người sẽ tự động mang tất cả nguyên liệu đến cho bạn, bạn chỉ việc nấu thôi

Trong code Java:

Không có DI (cách cũ - phải tự tạo tất cả):

```
public class BookingService {
    // Bạn phải tự tạo đối tượng UserRepository
    // Giống như tự đi mua nguyên liệu
    private UserRepository userRepository = new UserRepository();

    public void createBooking() {
        // Sử dụng userRepository
        userRepository.save(user);
    }
}
```

Có DI (Spring - Spring tự động tạo và đưa cho bạn):

```
@Service
public class BookingService {
    // Bạn chỉ cần "yêu cầu" UserRepository
    // Spring sẽ tự động tạo và gán vào biến này
    // Giống như chỉ cần nói "Tôi cần UserRepository", Spring sẽ tự động
mang đến
    @Autowired
    private UserRepository userRepository;

    public void createBooking() {
        // Sử dụng userRepository (đã được Spring tự động gán vào)
        userRepository.save(user);
    }
}
```

Giải thích chi tiết:

- **Dependency (Phụ thuộc):** Là những đối tượng mà class này cần để hoạt động. Ví dụ: `BookingService` cần `UserRepository` để lưu dữ liệu → `UserRepository` là dependency của `BookingService`
- **Injection (Cung cấp/Tự động đưa vào):** Là việc Spring tự động tạo đối tượng và gán vào biến cho bạn, thay vì bạn phải tự tạo

Tại sao dùng DI?

- **Linh hoạt:** Dễ dàng thay đổi implementation (ví dụ: khi test, có thể thay `UserRepository` bằng `MockUserRepository` giả lập)
- **Dễ test:** Có thể tạo đối tượng giả (mock) để test mà không cần database thật
- **Giảm ràng buộc:** Class không cần biết cách tạo dependency, chỉ cần biết sử dụng (giống như bạn không cần biết cách làm mì, chỉ cần biết cách nấu)

3.2. Các Annotation quan trọng: @Component, @Service, @Repository

Giải thích: Các annotation này đều đánh dấu class là một Bean (để Spring Container quản lý). Chúng giống nhau về chức năng, nhưng khác nhau về ý nghĩa (semantic).

@Component

1. Định nghĩa:

- `@Component` là annotation cơ bản nhất trong Spring Framework
- Đánh dấu một class là một "component" (thành phần) mà Spring Container sẽ quản lý
- Khi Spring Container khởi động, nó sẽ tự động tìm tất cả class có `@Component` và tạo đối tượng (Bean) từ chúng

2. Cách thức hoạt động:

```
Bước 1: Ứng dụng khởi động
    ↓
Bước 2: Spring Container quét tất cả package
    ↓
Bước 3: Tìm các class có @Component
    ↓
Bước 4: Tạo đối tượng từ class đó (gọi constructor)
    ↓
Bước 5: Lưu đối tượng vào "kho" (IoC Container)
    ↓
Bước 6: Sẵn sàng để inject vào nơi khác
```

3. Trường hợp sử dụng thực tế:

- Dùng cho các class **không thuộc** Service, Repository, hoặc Controller
- Thường dùng cho các class tiện ích (utility classes), helper classes
- Ví dụ: EmailSender, NotificationService, PaymentGateway, Logger, Validator

4. Ví dụ minh họa:

Ví dụ đơn giản:

```java
@Component
public class EmailSender {
    public void sendEmail(String to, String message) {
```

```
        System.out.println("Gửi email đến: " + to);
        System.out.println("Nội dung: " + message);
    }
}

// Sử dụng trong class khác
@Service
public class BookingService {
    @Autowired
    private EmailSender emailSender;  // Spring tự động inject EmailSender

    public void notifyUser(String email) {
        emailSender.sendEmail(email, "Chuyến xe của bạn đã được xác
nhận!");
    }
}
```

Ví dụ trong Taxi Booking System:

```
// Class gửi thông báo SMS
@Component
public class SmsSender {
    public void sendSms(String phoneNumber, String message) {
        // Logic gửi SMS (giả lập)
        System.out.println("Gửi SMS đến: " + phoneNumber);
        System.out.println("Nội dung: " + message);
    }
}

// Class tính giá cước
@Component
public class PriceCalculator {
    private static final int BASE_PRICE = 10000;  // Giá cơ bản
    private static final int PRICE_PER_KM = 15000;  // Giá mỗi km

    public int calculatePrice(double distance) {
        return BASE_PRICE + (int)(distance * PRICE_PER_KM);
    }
}

// Sử dụng trong BookingService
@Service
public class BookingService {
    @Autowired
    private SmsSender smsSender;  // Spring tự động inject

    @Autowired
    private PriceCalculator priceCalculator;  // Spring tự động inject

    public Booking createBooking(BookingRequest request) {
        // Tính giá
        int price = priceCalculator.calculatePrice(request.getDistance());
```

```
        // Tạo booking
        Booking booking = new Booking();
        booking.setPrice(price);
        // ... logic khác

        // Gửi SMS thông báo
        smsSender.sendSms(request.getPhone(), "Đặt xe thành công!");

        return booking;
    }
}
```

Lưu ý quan trọng:

- `@Service`, `@Repository`, `@Controller` đều là "phiên bản đặc biệt" của `@Component`
- Về mặt kỹ thuật, chúng hoạt động giống hệt nhau
- Sự khác biệt chỉ là về **ý nghĩa** (semantic) để code dễ đọc và dễ hiểu hơn

---

@Service

1. Định nghĩa:

- `@Service` là annotation đánh dấu class là một Service (dịch vụ)
- Về mặt kỹ thuật, `@Service` = `@Component` (chỉ khác tên)
- Dùng để chỉ rõ class này xử lý **business logic** (logic nghiệp vụ)

2. Cách thức hoạt động:

- Hoạt động giống hệt `@Component`
- Spring Container tự động tạo Bean từ class có `@Service`
- Khác biệt duy nhất: Tên annotation rõ ràng hơn, giúp code dễ đọc

3. Trường hợp sử dụng thực tế:

- Dùng cho các class xử lý **business logic** (logic nghiệp vụ)
- Thường là các Service class: `BookingService`, `UserService`, `PaymentService`
- Service class thường:
  - Xử lý logic phức tạp
  - Gọi Repository để lưu/xóa dữ liệu
  - Gọi các Service khác
  - Xử lý validation, tính toán

4. Ví dụ minh họa:

Ví dụ đơn giản:

```
@Service
public class BookingService {
```

```
    @Autowired
    private UserRepository userRepository;  // Inject Repository

    public Booking createBooking(BookingRequest request) {
        // Business logic: Xử lý đặt xe
        // 1. Validate dữ liệu
        if (request.getDistance() <= 0) {
            throw new IllegalArgumentException("Khoảng cách phải lớn hơn
0");
        }

        // 2. Tính giá
        int price = calculatePrice(request.getDistance());

        // 3. Tạo booking
        Booking booking = new Booking();
        booking.setDistance(request.getDistance());
        booking.setPrice(price);
        booking.setStatus("PENDING");

        // 4. Lưu vào database
        return userRepository.save(booking);
    }

    private int calculatePrice(double distance) {
        return (int)(distance * 15000);  // 15000 VNĐ/km
    }
}
```

Ví dụ trong Taxi Booking System:

```
@Service
public class BookingService {
    @Autowired
    private BookingRepository bookingRepository;

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private EmailSender emailSender;

    /**
     * Tạo booking mới
     * Business logic: Validate → Tính giá → Tạo booking → Lưu DB → Gửi
email
     */
    public Booking createBooking(CreateBookingRequest request) {
        // 1. Validate user tồn tại
        User user = userRepository.findById(request.getUserId())
            .orElseThrow(() -> new UserNotFoundException("User không tồn
tại"));
```

```
        // 2. Validate khoảng cách
        if (request.getDistance() <= 0) {
            throw new InvalidDistanceException("Khoảng cách không hợp
lệ");
        }

        // 3. Tính giá cước
        int price = calculatePrice(request.getDistance(),
request.getVehicleType());

        // 4. Tạo booking
        Booking booking = new Booking();
        booking.setPassengerId(user.getId());
        booking.setPickupLocation(request.getPickupLocation());
        booking.setDropoffLocation(request.getDropoffLocation());
        booking.setDistance(request.getDistance());
        booking.setPrice(price);
        booking.setStatus(BookingStatus.PENDING);
        booking.setCreatedAt(LocalDateTime.now());

        // 5. Lưu vào database
        Booking savedBooking = bookingRepository.save(booking);

        // 6. Gửi email thông báo
        emailSender.sendEmail(user.getEmail(),
            "Đặt xe thành công! Mã booking: " + savedBooking.getId());

        return savedBooking;
    }

    /**
     * Tính giá cước dựa trên khoảng cách và loại xe
     */
    private int calculatePrice(double distance, String vehicleType) {
        int basePrice = 10000;  // Giá cơ bản
        int pricePerKm;

        switch (vehicleType) {
            case "4_SEATER":
                pricePerKm = 15000;
                break;
            case "7_SEATER":
                pricePerKm = 20000;
                break;
            case "MOTORBIKE":
                pricePerKm = 10000;
                break;
            default:
                pricePerKm = 15000;
        }

        return basePrice + (int)(distance * pricePerKm);
```

```
    }
}
```

So sánh @Component vs @Service:

- `@Component`: Dùng cho class tiện ích, helper
- `@Service`: Dùng cho class xử lý business logic
- Về mặt kỹ thuật: Giống hệt nhau
- Về mặt ý nghĩa: `@Service` rõ ràng hơn, code dễ đọc hơn

@Repository

1. Định nghĩa:

- `@Repository` là annotation đánh dấu class là một Repository (kho lưu trữ)
- Về mặt kỹ thuật, `@Repository` = `@Component` (chỉ khác tên)
- Dùng để chỉ rõ class này tương tác với **database**

2. Cách thức hoạt động:

- Hoạt động giống hệt `@Component`
- Spring Container tự động tạo Bean từ class có `@Repository`
- **Đặc biệt:** Spring tự động xử lý exception từ database (chuyển thành Spring exception)

3. Trường hợp sử dụng thực tế:

- Dùng cho các class/interface tương tác với database
- Thường là các Repository: `UserRepository`, `BookingRepository`, `PaymentRepository`
- Repository thường:
  - Kế thừa từ `JpaRepository` (Spring Data JPA)
  - Chứa các method: `save()`, `findById()`, `findAll()`, `delete()`
  - Có thể có custom query methods

4. Ví dụ minh họa:

Ví dụ đơn giản:

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    // JpaRepository đã cung cấp sẵn các method:
    // - save(User user)
    // - findById(Long id)
    // - findAll()
    // - delete(User user)
    // - ...

    // Custom query method
    User findByEmail(String email);
```

```
    boolean existsByEmail(String email);
}
```

Ví dụ trong Taxi Booking System:

```
// Entity User
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String email;
    private String password;
    private String fullName;
    private String phone;
    private String role;  // PASSENGER, DRIVER, ADMIN

    // Getters and Setters
}

// Repository
@Repository
public interface UserRepository extends JpaRepository<User, Long> {
    // Tìm user theo email
    Optional<User> findByEmail(String email);

    // Kiểm tra email đã tồn tại chưa
    boolean existsByEmail(String email);

    // Tìm user theo role
    List<User> findByRole(String role);

    // Tìm user theo phone
    Optional<User> findByPhone(String phone);
}

// Booking Repository
@Repository
public interface BookingRepository extends JpaRepository<Booking, Long> {
    // Tìm tất cả booking của một passenger
    List<Booking> findByPassengerId(Long passengerId);

    // Tìm booking theo status
    List<Booking> findByStatus(String status);

    // Tìm booking của driver
    List<Booking> findByDriverId(Long driverId);

    // Tìm booking đang chờ (PENDING) và chưa có driver
```

```
    List<Booking> findByStatusAndDriverIdIsNull(String status);
}
```

Sử dụng Repository trong Service:

```
@Service
public class BookingService {
    @Autowired
    private BookingRepository bookingRepository;  // Inject Repository

    @Autowired
    private UserRepository userRepository;

    public Booking createBooking(CreateBookingRequest request) {
        // Sử dụng Repository để lưu dữ liệu
        Booking booking = new Booking();
        // ... set các thuộc tính

        return bookingRepository.save(booking);  // Lưu vào database
    }

    public List<Booking> getBookingsByPassenger(Long passengerId) {
        // Sử dụng Repository để tìm dữ liệu
        return bookingRepository.findByPassengerId(passengerId);
    }
}
```

Lưu ý quan trọng:

- `@Repository` thường dùng với **interface** (không phải class)
- Interface này kế thừa từ `JpaRepository` (Spring Data JPA tự động implement)
- Spring tự động tạo implementation cho interface này khi ứng dụng khởi động

---

@Controller

1. Định nghĩa:

- `@Controller` là annotation đánh dấu class là một Controller (bộ điều khiển)
- Về mặt kỹ thuật, `@Controller` = `@Component` + khả năng xử lý HTTP request
- Dùng cho các class xử lý HTTP request từ client

2. Cách thức hoạt động:

```
Client gửi HTTP Request
    ↓
Spring DispatcherServlet nhận request
    ↓
```

```
Tìm Controller phù hợp (dựa vào @RequestMapping)
    ↓
Gọi method trong Controller
    ↓
Trả về View (HTML) hoặc ModelAndView
```

3. Trường hợp sử dụng thực tế:

- Dùng cho các class xử lý HTTP request
- Thường là các Controller: `BookingController`, `UserController`, `PaymentController`
- Controller thường:
  - Nhận request từ client
  - Gọi Service để xử lý logic
  - Trả về View (HTML) hoặc redirect

4. Ví dụ minh họa:

Ví dụ đơn giản:

```
@Controller
@RequestMapping("/bookings")
public class BookingController {
    @Autowired
    private BookingService bookingService;

    // Xử lý GET request: /bookings
    @GetMapping
    public String getAllBookings(Model model) {
        List<Booking> bookings = bookingService.getAllBookings();
        model.addAttribute("bookings", bookings);
        return "bookings/list";  // Trả về file HTML: bookings/list.html
    }

    // Xử lý GET request: /bookings/{id}
    @GetMapping("/{id}")
    public String getBooking(@PathVariable Long id, Model model) {
        Booking booking = bookingService.getBookingById(id);
        model.addAttribute("booking", booking);
        return "bookings/detail";  // Trả về file HTML:
bookings/detail.html
    }
}
```

So sánh @Controller vs @RestController:

- `@Controller`: Trả về View (HTML) - Dùng cho web application có giao diện
- `@RestController`: Trả về JSON - Dùng cho REST API (sẽ học ở Buổi 3)
- `@RestController` = `@Controller` + `@ResponseBody`

Lưu ý: Ở giai đoạn này, bạn chỉ cần biết có annotation này. Sẽ học chi tiết ở Buổi 3.

@Configuration và @Bean

1. Định nghĩa:

- `@Configuration`: Đánh dấu class là class cấu hình (chứa các method tạo Bean)
- `@Bean`: Đánh dấu method để Spring tạo Bean từ method đó
- Dùng khi bạn muốn tạo Bean **thủ công** (không dùng `@Component`)

2. Cách thức hoạt động:

```
Bước 1: Spring Container quét và tìm class có @Configuration
    ↓
Bước 2: Tìm các method có @Bean trong class đó
    ↓
Bước 3: Gọi method đó để tạo đối tượng
    ↓
Bước 4: Lưu đối tượng vào "kho" (IoC Container)
    ↓
Bước 5: Sẵn sàng để inject vào nơi khác
```

3. Trường hợp sử dụng thực tế:

- **Khi tạo Bean từ class của thư viện bên ngoài:**
  - Class không có `@Component` (không thể sửa code)
  - Ví dụ: `RestTemplate`, `ObjectMapper`, `HttpClient`
- **Khi cần logic phức tạp để tạo Bean:**
  - Cần đọc config từ file
  - Cần tính toán, xử lý trước khi tạo Bean
  - Cần tạo nhiều Bean phụ thuộc lẫn nhau
- **Khi muốn tạo Bean có điều kiện:**
  - Tạo Bean khác nhau tùy theo môi trường (dev, prod)
  - Tạo Bean chỉ khi thỏa mãn điều kiện nào đó

4. Ví dụ minh họa:

Ví dụ 1: Tạo Bean từ class thư viện bên ngoài

```java
@Configuration
public class AppConfig {

    // RestTemplate là class của Spring, không có @Component
    // Phải tạo Bean thủ công
    @Bean
```

```
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }

    // ObjectMapper là class của Jackson, không có @Component
    @Bean
    public ObjectMapper objectMapper() {
        ObjectMapper mapper = new ObjectMapper();
        mapper.setDateFormat(new SimpleDateFormat("yyyy-MM-dd"));
        return mapper;
    }
}

// Sử dụng
@Service
public class PaymentService {
    @Autowired
    private RestTemplate restTemplate;  // Spring tự động inject

    public void callPaymentAPI() {
        // Sử dụng restTemplate để gọi API
        String response =
restTemplate.getForObject("https://api.payment.com", String.class);
    }
}
```

Ví dụ 2: Tạo Bean với logic phức tạp

```
@Configuration
public class AppConfig {

    // Đọc config từ application.properties
    @Value("${app.environment}")
    private String environment;

    // Tạo Bean với logic phức tạp
    @Bean
    public Coach tennisCoach() {
        TennisCoach coach = new TennisCoach();

        // Thêm logic phức tạp
        if ("production".equals(environment)) {
            coach.setTrainingMode("STRICT");
        } else {
            coach.setTrainingMode("RELAXED");
        }

        return coach;
    }
}
```

Ví dụ 3: Tạo Bean phụ thuộc lẫn nhau

```
@Configuration
public class AppConfig {

    // Tạo Bean A
    @Bean
    public DatabaseConnection databaseConnection() {
        return new DatabaseConnection("localhost", 3306);
    }

    // Tạo Bean B phụ thuộc vào Bean A
    @Bean
    public UserRepository userRepository(DatabaseConnection dbConnection)
{
        // Spring tự động inject DatabaseConnection vào đây
        return new UserRepository(dbConnection);
    }

    // Tạo Bean C phụ thuộc vào Bean B
    @Bean
    public UserService userService(UserRepository userRepository) {
        // Spring tự động inject UserRepository vào đây
        return new UserService(userRepository);
    }
}
```

Ví dụ trong Taxi Booking System:

```
@Configuration
public class TaxiConfig {

    // Đọc config từ application.properties
    @Value("${taxi.payment.gateway.url}")
    private String paymentGatewayUrl;

    @Value("${taxi.payment.gateway.api-key}")
    private String apiKey;

    // Tạo Bean cho PaymentGateway
    @Bean
    public PaymentGateway paymentGateway() {
        PaymentGateway gateway = new PaymentGateway();
        gateway.setUrl(paymentGatewayUrl);
        gateway.setApiKey(apiKey);
        gateway.setTimeout(5000);  // 5 giây
        return gateway;
    }

    // Tạo Bean cho EmailService với config
```

```
    @Bean
    public EmailService emailService() {
        EmailService emailService = new EmailService();
        emailService.setSmtpHost("smtp.gmail.com");
        emailService.setSmtpPort(587);
        emailService.setFromEmail("noreply@taxi.com");
        return emailService;
    }

    // Tạo Bean cho RestTemplate (để gọi API bên ngoài)
    @Bean
    public RestTemplate restTemplate() {
        RestTemplate restTemplate = new RestTemplate();

        // Cấu hình timeout
        HttpComponentsClientHttpRequestFactory factory =
            new HttpComponentsClientHttpRequestFactory();
        factory.setConnectTimeout(3000);  // 3 giây
        factory.setReadTimeout(5000);      // 5 giây

        restTemplate.setRequestFactory(factory);
        return restTemplate;
    }
}

// Sử dụng
@Service
public class BookingService {
    @Autowired
    private PaymentGateway paymentGateway;  // Spring tự động inject

    @Autowired
    private EmailService emailService;

    public void processPayment(Booking booking) {
        // Sử dụng paymentGateway đã được cấu hình
        paymentGateway.charge(booking.getPrice());
    }
}
```

So sánh @Component vs @Bean:

Tiêu chí	@Component	@Bean
Dùng ở đâu?	Trên class	Trên method
Khi nào dùng?	Class của bạn	Class thư viện bên ngoài hoặc cần logic phức tạp
Cách tạo Bean	Spring tự động gọi constructor	Bạn tự viết method
Linh hoạt	Đơn giản, ít linh hoạt	Linh hoạt, có thể thêm logic

Tiêu chí	@Component	@Bean
Ví dụ	`@Component class EmailSender`	`@Bean public RestTemplate restTemplate()`

@Primary

1. Định nghĩa:

- `@Primary` đánh dấu Bean là Bean **ưu tiên** (primary bean)
- Khi có nhiều Bean cùng loại, Spring sẽ tự động chọn Bean có `@Primary`
- Dùng kèm với `@Component`, `@Service`, `@Repository`, hoặc `@Bean`

2. Cách thức hoạt động:

```
Bước 1: Spring Container tìm tất cả Bean cùng loại (cùng interface)
    ↓
Bước 2: Kiểm tra xem có Bean nào có @Primary không
    ↓
Bước 3: Nếu có → Chọn Bean có @Primary
    ↓
Bước 4: Nếu không có @Primary → Báo lỗi (nếu có nhiều hơn 1 Bean)
    ↓
Bước 5: Inject Bean đã chọn vào nơi cần dùng
```

3. Trường hợp sử dụng thực tế:

- **Khi có nhiều implementation của cùng một interface:**
  - Ví dụ: Có `EmailService`, `SmsService`, `PushNotificationService` đều implement `NotificationService`
  - Bạn muốn `EmailService` là mặc định
- **Khi có nhiều cách triển khai khác nhau:**
  - Ví dụ: `ProductionPaymentGateway` và `TestPaymentGateway` đều implement `PaymentGateway`
  - Ở môi trường production, dùng `@Primary` cho `ProductionPaymentGateway`

4. Ví dụ minh họa:

Ví dụ đơn giản:

```java
// Interface
public interface Coach {
    String train();
}
```

```
// Implementation 1
@Component
public class TennisCoach implements Coach {
    @Override
    public String train() {
        return "Đánh bóng tennis!";
    }
}

// Implementation 2 - Đánh dấu là Bean ưu tiên
@Component
@Primary  // Đánh dấu là Bean ưu tiên
public class FootballCoach implements Coach {
    @Override
    public String train() {
        return "Chạy 5 vòng sân!";
    }
}

// Sử dụng - Spring tự động chọn FootballCoach
@Service
public class MainApp {
    @Autowired
    private Coach coach;  // Spring sẽ tự động chọn FootballCoach (vì có
@Primary)

    public void test() {
        System.out.println(coach.train());  // "Chạy 5 vòng sân!"
    }
}
```

Ví dụ trong Taxi Booking System:

```
// Interface NotificationService
public interface NotificationService {
    void sendNotification(String recipient, String message);
}

// Implementation 1: Email
@Component
public class EmailNotificationService implements NotificationService {
    @Override
    public void sendNotification(String recipient, String message) {
        System.out.println("Gửi email đến: " + recipient);
        System.out.println("Nội dung: " + message);
    }
}

// Implementation 2: SMS - Đánh dấu là mặc định
@Component
@Primary  // SMS là phương thức thông báo mặc định
public class SmsNotificationService implements NotificationService {
```

```
    @Override
    public void sendNotification(String recipient, String message) {
        System.out.println("Gửi SMS đến: " + recipient);
        System.out.println("Nội dung: " + message);
    }
}

// Implementation 3: Push Notification
@Component
public class PushNotificationService implements NotificationService {
    @Override
    public void sendNotification(String recipient, String message) {
        System.out.println("Gửi push notification đến: " + recipient);
        System.out.println("Nội dung: " + message);
    }
}

// Sử dụng - Spring tự động chọn SmsNotificationService
@Service
public class BookingService {
    @Autowired
    private NotificationService notificationService;
    // Spring tự động chọn SmsNotificationService (vì có @Primary)

    public void notifyUser(String phone, String message) {
        notificationService.sendNotification(phone, message);
        // Sẽ gọi SmsNotificationService.sendNotification()
    }
}
```

Lưu ý quan trọng:

- Nếu có nhiều Bean cùng loại và **không có** `@Primary`, Spring sẽ báo lỗi
- Chỉ nên có **1 Bean** có `@Primary` cho mỗi loại
- Nếu muốn chọn Bean cụ thể, dùng `@Qualifier` thay vì `@Primary`

@Qualifier

1. Định nghĩa:

- `@Qualifier` chỉ định rõ Bean **cụ thể** nào cần dùng (khi có nhiều Bean cùng loại)
- Dùng kèm với `@Autowired` để chọn Bean chính xác
- Phải chỉ định tên Bean (tên class hoặc tên đã đặt)

2. Cách thức hoạt động:

```
Bước 1: Spring Container tìm tất cả Bean cùng loại
    ↓
Bước 2: Kiểm tra @Qualifier để biết tên Bean cần tìm
```

```
    ↓
Bước 3: Tìm Bean có tên khớp với @Qualifier
    ↓
Bước 4: Nếu tìm thấy → Inject Bean đó
    ↓
Bước 5: Nếu không tìm thấy → Báo lỗi
```

3. Trường hợp sử dụng thực tế:

- **Khi có nhiều Bean cùng loại và muốn chọn Bean cụ thể:**
  - Ví dụ: Có `EmailService` và `SmsService`, muốn dùng `EmailService` ở một chỗ, `SmsService` ở chỗ khác
- **Khi không muốn dùng `@Primary`:**
  - `@Primary` chỉ cho phép 1 Bean mặc định
  - `@Qualifier` cho phép chọn Bean khác nhau ở các nơi khác nhau
- **Khi muốn linh hoạt hơn:**
  - Có thể inject nhiều Bean cùng loại vào các biến khác nhau

4. Ví dụ minh họa:

Ví dụ đơn giản:

```java
// Interface
public interface Coach {
    String train();
}

// Implementation 1 - Đặt tên Bean là "tennisCoach"
@Component("tennisCoach")  // Tên Bean = "tennisCoach"
public class TennisCoach implements Coach {
    @Override
    public String train() {
        return "Đánh bóng tennis!";
    }
}

// Implementation 2 - Đặt tên Bean là "footballCoach"
@Component("footballCoach")  // Tên Bean = "footballCoach"
public class FootballCoach implements Coach {
    @Override
    public String train() {
        return "Chạy 5 vòng sân!";
    }
}

// Sử dụng - Chỉ định rõ Bean nào
@Service
```

```
public class MainApp {
    @Autowired
    @Qualifier("tennisCoach")  // Chỉ định dùng TennisCoach
    private Coach coach1;  // Sẽ là TennisCoach

    @Autowired
    @Qualifier("footballCoach")  // Chỉ định dùng FootballCoach
    private Coach coach2;  // Sẽ là FootballCoach

    public void test() {
        System.out.println(coach1.train());  // "Đánh bóng tennis!"
        System.out.println(coach2.train());  // "Chạy 5 vòng sân!"
    }
}
```

Ví dụ trong Taxi Booking System:

```
// Interface PaymentGateway
public interface PaymentGateway {
    void processPayment(double amount);
}

// Implementation 1: Stripe Payment
@Component("stripePaymentGateway")
public class StripePaymentGateway implements PaymentGateway {
    @Override
    public void processPayment(double amount) {
        System.out.println("Xử lý thanh toán qua Stripe: " + amount);
        // Logic xử lý Stripe
    }
}

// Implementation 2: PayPal Payment
@Component("paypalPaymentGateway")
public class PayPalPaymentGateway implements PaymentGateway {
    @Override
    public void processPayment(double amount) {
        System.out.println("Xử lý thanh toán qua PayPal: " + amount);
        // Logic xử lý PayPal
    }
}

// Implementation 3: VNPay Payment (mặc định cho Việt Nam)
@Component("vnpayPaymentGateway")
@Primary  // Mặc định dùng VNPay
public class VNPayPaymentGateway implements PaymentGateway {
    @Override
    public void processPayment(double amount) {
        System.out.println("Xử lý thanh toán qua VNPay: " + amount);
        // Logic xử lý VNPay
    }
}
```

```
// Sử dụng – Chọn PaymentGateway cụ thể
@Service
public class BookingService {
    // Mặc định dùng VNPay (có @Primary)
    @Autowired
    private PaymentGateway defaultPaymentGateway;

    // Chọn Stripe cho khách hàng quốc tế
    @Autowired
    @Qualifier("stripePaymentGateway")
    private PaymentGateway internationalPaymentGateway;

    // Chọn PayPal cho khách hàng Mỹ
    @Autowired
    @Qualifier("paypalPaymentGateway")
    private PaymentGateway usPaymentGateway;

    public void processPayment(Booking booking, String paymentMethod) {
        switch (paymentMethod) {
            case "STRIPE":

internationalPaymentGateway.processPayment(booking.getPrice());
                break;
            case "PAYPAL":
                usPaymentGateway.processPayment(booking.getPrice());
                break;
            default:
                defaultPaymentGateway.processPayment(booking.getPrice());
        }
    }
}
```

Lưu ý về tên Bean:

- Mặc định, tên Bean = tên class (chữ cái đầu viết thường)
- Ví dụ: `TennisCoach` → tên Bean = `"tennisCoach"`
- Có thể đặt tên tùy ý: `@Component("myCustomName")`

So sánh @Primary vs @Qualifier:

Tiêu chí	@Primary	@Qualifier
Cách hoạt động	Tự động chọn Bean mặc định	Chỉ định rõ Bean cụ thể
Số lượng	Chỉ 1 Bean có @Primary	Có thể có nhiều @Qualifier
Linh hoạt	Ít linh hoạt (chỉ 1 mặc định)	Linh hoạt (chọn khác nhau ở các nơi)
Khi nào dùng	Khi có 1 Bean chính, các Bean khác ít dùng	Khi cần dùng nhiều Bean cùng lúc

Tiêu chí	@Primary	@Qualifier
Ví dụ	`@Primary` cho EmailService (mặc định)	`@Qualifier("sms")` cho SmsService (khi cần)

@Value

1. Định nghĩa:

- `@Value` đọc giá trị từ file `application.properties` hoặc `application.yml`
- Gán giá trị vào biến, field, hoặc parameter
- Hỗ trợ giá trị mặc định nếu không tìm thấy trong config

2. Cách thức hoạt động:

```
Bước 1: Spring Container khởi động
    ↓
Bước 2: Đọc file application.properties
    ↓
Bước 3: Tìm giá trị theo key trong @Value("${key}")
    ↓
Bước 4: Nếu tìm thấy → Gán vào biến
    ↓
Bước 5: Nếu không tìm thấy → Dùng giá trị mặc định (nếu có) hoặc báo lỗi
```

3. Trường hợp sử dụng thực tế:

- **Đọc cấu hình từ file properties:**
  - Database URL, username, password
  - API keys, secrets
  - Timeout, retry count
  - Feature flags
- **Thay đổi giá trị mà không cần sửa code:**
  - Chỉ cần sửa file `application.properties`
  - Không cần rebuild code
- **Cấu hình khác nhau cho môi trường khác nhau:**
  - `application-dev.properties` cho development
  - `application-prod.properties` cho production

4. Ví dụ minh họa:

Ví dụ đơn giản:

```
// File: application.properties
app.name=Taxi Booking System
app.version=1.0.0
taxi.base.price=10000
taxi.price.per.km=15000

// Sử dụng trong code
@Component
public class AppConfig {
    // Đọc giá trị từ application.properties
    @Value("${app.name}")
    private String appName;  // Sẽ là "Taxi Booking System"

    @Value("${app.version}")
    private String appVersion;  // Sẽ là "1.0.0"

    @Value("${taxi.base.price}")
    private int basePrice;  // Sẽ là 10000

    // Có thể dùng giá trị mặc định nếu không tìm thấy
    @Value("${taxi.max.distance:50}")  // Nếu không có, dùng 50
    private int maxDistance;

    // Có thể dùng với các kiểu dữ liệu khác
    @Value("${app.debug:false}")  // Boolean, mặc định false
    private boolean debug;

    @Value("${app.timeout:5000}")  // Integer, mặc định 5000
    private int timeout;
}
```

Ví dụ trong Taxi Booking System:

```
// File: application.properties
# Database Configuration
spring.datasource.url=jdbc:mysql://localhost:3306/taxi_db
spring.datasource.username=root
spring.datasource.password=password123

# Taxi Configuration
taxi.base.price=10000
taxi.price.per.km=15000
taxi.price.per.km.7seater=20000
taxi.price.per.km.motorbike=10000
taxi.max.distance=100
taxi.cancellation.timeout.minutes=15

# Payment Configuration
payment.gateway.url=https://api.payment.com
payment.gateway.api.key=abc123xyz
payment.gateway.timeout=5000
```

```
# Email Configuration
email.smtp.host=smtp.gmail.com
email.smtp.port=587
email.from=noreply@taxi.com

// Sử dụng trong Service
@Service
public class BookingService {
    // Đọc giá cước từ config
    @Value("${taxi.base.price}")
    private int basePrice;

    @Value("${taxi.price.per.km}")
    private int pricePerKm;

    @Value("${taxi.price.per.km.7seater}")
    private int pricePerKm7Seater;

    @Value("${taxi.price.per.km.motorbike}")
    private int pricePerKmMotorbike;

    @Value("${taxi.max.distance:100}")  // Mặc định 100km
    private int maxDistance;

    public int calculatePrice(double distance, String vehicleType) {
        // Validate khoảng cách
        if (distance > maxDistance) {
            throw new IllegalArgumentException("Khoảng cách vượt quá giới
hạn: " + maxDistance + "km");
        }

        int pricePerKm;
        switch (vehicleType) {
            case "7_SEATER":
                pricePerKm = pricePerKm7Seater;
                break;
            case "MOTORBIKE":
                pricePerKm = pricePerKmMotorbike;
                break;
            default:
                pricePerKm = this.pricePerKm;
        }

        return basePrice + (int)(distance * pricePerKm);
    }
}

// Sử dụng trong PaymentService
@Service
public class PaymentService {
    @Value("${payment.gateway.url}")
    private String paymentGatewayUrl;
```

```
    @Value("${payment.gateway.api.key}")
    private String apiKey;

    @Value("${payment.gateway.timeout:5000}")  // Mặc định 5 giây
    private int timeout;

    public void processPayment(double amount) {
        // Sử dụng các giá trị từ config
        System.out.println("Gọi API: " + paymentGatewayUrl);
        System.out.println("API Key: " + apiKey);
        System.out.println("Timeout: " + timeout + "ms");
        // Logic xử lý thanh toán
    }
}

// Sử dụng trong EmailService
@Service
public class EmailService {
    @Value("${email.smtp.host}")
    private String smtpHost;

    @Value("${email.smtp.port}")
    private int smtpPort;

    @Value("${email.from}")
    private String fromEmail;

    public void sendEmail(String to, String subject, String body) {
        // Sử dụng các giá trị từ config
        System.out.println("Gửi email từ: " + fromEmail);
        System.out.println("SMTP Host: " + smtpHost);
        System.out.println("SMTP Port: " + smtpPort);
        // Logic gửi email
    }
}
```

Lưu ý quan trọng:

- Cú pháp: `@Value("${key}")` hoặc `@Value("${key:defaultValue}")`
- Giá trị mặc định: Dùng dấu `:` sau key, ví dụ: `@Value("${key:default}")`
- Có thể dùng với các kiểu dữ liệu: String, int, boolean, double, etc.
- File config phải đặt trong `src/main/resources/application.properties`

3.3. @Autowired - Tự động cung cấp đối tượng

1. Định nghĩa:

- `@Autowired` là annotation yêu cầu Spring Container tự động tìm và cung cấp (inject) đối tượng cần thiết
- Spring sẽ tự động tìm Bean phù hợp trong IoC Container và gán vào biến, constructor parameter, hoặc setter method

- Đây là cách Spring thực hiện Dependency Injection

2. Cách thức hoạt động:

```
Bước 1: Spring Container quét class có @Autowired
    ↓
Bước 2: Xác định loại đối tượng cần (dựa vào kiểu dữ liệu)
    ↓
Bước 3: Tìm Bean phù hợp trong IoC Container
    ↓
Bước 4: Kiểm tra @Qualifier (nếu có) để chọn Bean cụ thể
    ↓
Bước 5: Kiểm tra @Primary (nếu có nhiều Bean cùng loại)
    ↓
Bước 6: Tạo hoặc lấy Bean từ Container
    ↓
Bước 7: Gán vào biến/constructor/setter
```

3. Trường hợp sử dụng thực tế:

- **Inject Repository vào Service:**
  - Service cần Repository để lưu/xóa dữ liệu
- **Inject Service vào Controller:**
  - Controller cần Service để xử lý business logic
- **Inject các dependency phức tạp:**
  - Inject nhiều dependency vào một class
  - Inject dependency có dependency khác (dependency chain)

4. Ví dụ minh họa:

Ví dụ dễ hiểu:

- Bạn đang nấu ăn và cần dao → Bạn nói "Tôi cần dao" → Có người tự động mang dao đến cho bạn
- Trong code: Bạn cần `UserRepository` → Bạn đánh dấu `@Autowired` → Spring tự động tìm `UserRepository` trong kho và gán vào biến cho bạn

Các cách sử dụng @Autowired:

Cách 1: Field Injection (tự động gán vào biến) - Phổ biến nhất, dễ viết

Định nghĩa: Gán trực tiếp vào field (biến) của class

Cách hoạt động:

- Spring sử dụng Reflection để gán giá trị vào field
- Gán sau khi object được tạo

Trường hợp sử dụng:

- Code ngắn gọn, dễ viết
- Phù hợp cho prototype, test nhanh

Ví dụ:

```
@Service
public class BookingService {
    // Spring tự động tìm UserRepository trong kho
    // và gán vào biến userRepository này
    @Autowired
    private UserRepository userRepository;

    @Autowired
    private EmailService emailService;

    public void createBooking() {
        // Có thể dùng userRepository ngay
        userRepository.save(user);
        emailService.sendEmail("user@example.com", "Booking created");
    }
}
```

Ưu điểm:

- ✅ Code ngắn gọn, dễ viết
- ✅ Dễ đọc

Nhược điểm:

- ❌ Khó test (không thể truyền mock dễ dàng)
- ❌ Không thể dùng `final`
- ❌ Khó phát hiện lỗi (nếu thiếu dependency, chỉ báo lỗi khi runtime)

Cách 2: Constructor Injection (tự động gán qua constructor) - Khuyên dùng nhất

Định nghĩa: Gán qua constructor parameter

Cách hoạt động:

- Spring gọi constructor và truyền dependency vào
- Gán ngay khi object được tạo

Trường hợp sử dụng:

- **Khuyên dùng cho production code**
- Khi muốn đảm bảo dependency không null
- Khi muốn dùng `final` để immutable

Ví dụ:

```
@Service
public class BookingService {
    private final UserRepository userRepository;
    private final EmailService emailService;
    private final PaymentService paymentService;

    // Spring tự động tìm các dependency và truyền vào constructor
    // Khi tạo BookingService, Spring sẽ tự động gọi constructor này
    @Autowired  // Có thể bỏ @Autowired nếu chỉ có 1 constructor (từ
Spring 4.3+)
    public BookingService(
            UserRepository userRepository,
            EmailService emailService,
            PaymentService paymentService
    ) {
        this.userRepository = userRepository;
        this.emailService = emailService;
        this.paymentService = paymentService;
    }

    public void createBooking() {
        userRepository.save(user);
        emailService.sendEmail("user@example.com", "Booking created");
    }
}
```

Ưu điểm:

- ✅ Dễ test (có thể truyền mock vào constructor)
- ✅ Đảm bảo dependency không null (nếu thiếu, app không khởi động được)
- ✅ Có thể dùng `final` (immutable)
- ✅ Rõ ràng về dependencies của class

Nhược điểm:

- ❌ Code dài hơn một chút

Lưu ý: Từ Spring 4.3+, nếu class chỉ có 1 constructor, có thể bỏ `@Autowired`:

```
@Service
public class BookingService {
    private final UserRepository userRepository;

    // Không cần @Autowired nữa
    public BookingService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
}
```

Cách 3: Setter Injection (tự động gán qua setter) - Ít dùng

Định nghĩa: Gán qua setter method

Cách hoạt động:

- Spring gọi setter method và truyền dependency vào
- Gán sau khi object được tạo

Trường hợp sử dụng:

- Khi dependency là optional (có thể null)
- Khi cần thay đổi dependency sau khi object được tạo (hiếm)

Ví dụ:

```
@Service
public class BookingService {
    private UserRepository userRepository;

    // Spring tự động gọi method này và truyền UserRepository vào
    @Autowired
    public void setUserRepository(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    // Có thể có nhiều setter
    @Autowired(required = false)  // Optional - không bắt buộc
    public void setEmailService(EmailService emailService) {
        // Nếu không có EmailService, method này không được gọi
    }
}
```

Ưu điểm:

- ✅ Có thể làm dependency optional (`required = false`)

Nhược điểm:

- ❌ Khó test
- ❌ Không thể dùng `final`
- ❌ Có thể quên inject (nếu không có `@Autowired`)

Ví dụ trong Taxi Booking System:

```
@Service
public class BookingService {
    // Constructor Injection - Khuyên dùng
    private final BookingRepository bookingRepository;
    private final UserRepository userRepository;
```

```
    private final EmailService emailService;
    private final PaymentService paymentService;

    // Spring tự động inject tất cả dependencies
    public BookingService(
            BookingRepository bookingRepository,
            UserRepository userRepository,
            EmailService emailService,
            PaymentService paymentService
    ) {
        this.bookingRepository = bookingRepository;
        this.userRepository = userRepository;
        this.emailService = emailService;
        this.paymentService = paymentService;
    }

    public Booking createBooking(CreateBookingRequest request) {
        // Sử dụng các dependencies đã được inject
        User user = userRepository.findById(request.getUserId())
            .orElseThrow(() -> new UserNotFoundException());

        Booking booking = new Booking();
        // ... logic tạo booking

        Booking savedBooking = bookingRepository.save(booking);

        // Gửi email
        emailService.sendEmail(user.getEmail(), "Booking created");

        return savedBooking;
    }
}
```

So sánh 3 cách:

Tiêu chí	Field Injection	Constructor Injection	Setter Injection
Code	Ngắn gọn	Dài hơn	Trung bình
Test	Khó	Dễ	Khó
Final	❌ Không	✅ Có	❌ Không
Null safety	❌ Runtime error	✅ Compile error	❌ Runtime error
Khuyên dùng	⚠ Hạn chế	✅ **Khuyên dùng**	❌ Ít dùng
Optional	❌ Không	❌ Không	✅ Có

Kết luận: Nên dùng **Constructor Injection** cho production code.

3.4. IoC Container vs Dependency Injection - So sánh chi tiết

Câu hỏi thường gặp: IoC Container và Dependency Injection có gì khác nhau? Chúng có giống nhau không?

Trả lời ngắn gọn:

- **IoC Container** = Công cụ/Thiết bị (Cái gì đó)
- **Dependency Injection** = Cách thức/Hành động (Làm như thế nào)

Giải thích chi tiết:

1. IoC Container là gì?

- IoC Container là một "nhà kho" thông minh
- Nó là **công cụ** để tạo, lưu trữ và quản lý các đối tượng (Bean)
- Ví dụ: Giống như một nhà kho thật, nơi bạn lưu trữ đồ đạc

2. Dependency Injection là gì?

- Dependency Injection là **cách thức** mà IoC Container sử dụng để cung cấp đối tượng
- Nó là **hành động** đưa đối tượng vào nơi cần dùng
- Ví dụ: Giống như cách bạn lấy đồ từ kho và đưa cho người cần

Mối quan hệ giữa chúng:

```
IoC Container (Nhà kho)
    │
    │ sử dụng
    ▼
Dependency Injection (Cách lấy đồ từ kho và đưa cho người cần)
```

Ví dụ thực tế dễ hiểu:

Hãy tưởng tượng bạn có một nhà kho Amazon:

- **IoC Container** = Nhà kho Amazon (nơi lưu trữ hàng hóa)
    - Chức năng: Lưu trữ, quản lý hàng hóa
    - Có thể: Tạo hàng mới, lưu hàng, tìm hàng, quản lý hàng tồn kho
- **Dependency Injection** = Dịch vụ giao hàng của Amazon (cách đưa hàng đến tay bạn)
    - Chức năng: Lấy hàng từ kho và đưa đến nơi cần
    - Cách hoạt động: Khi bạn đặt hàng (`@Autowired`), nhân viên sẽ lấy hàng từ kho và giao đến cho bạn

Trong code Spring:

```
// ========== IoC Container (Nhà kho) ==========
// Spring Container tự động tạo và lưu các Bean vào "kho"
```

```java
@Component
public class BookingService {
    // Bean này được Spring Container tạo và lưu vào "kho"
}

@Component
public class UserRepository {
    // Bean này cũng được Spring Container tạo và lưu vào "kho"
}

// ========== Dependency Injection (Cách đưa hàng) ==========
// Spring Container sử dụng DI để đưa Bean vào nơi cần dùng

@Service
public class BookingController {
    // @Autowired = Yêu cầu Spring Container đưa BookingService cho tôi
    // Spring Container sẽ:
    // 1. Tìm BookingService trong "kho" (IoC Container)
    // 2. Lấy ra và gán vào biến này (Dependency Injection)
    @Autowired
    private BookingService bookingService;

    @Autowired
    private UserRepository userRepository;
}
```

Bảng so sánh chi tiết:

Tiêu chí	IoC Container	Dependency Injection
Là gì?	Công cụ/Thiết bị (Nhà kho)	Cách thức/Hành động (Dịch vụ giao hàng)
Chức năng	Tạo, lưu trữ, quản lý Bean	Đưa Bean vào nơi cần dùng
Khi nào hoạt động?	Khi ứng dụng khởi động	Khi gặp `@Autowired`
Làm gì?	Quét class → Tạo Bean → Lưu vào "kho"	Tìm Bean trong "kho" → Lấy ra → Gán vào biến
Ví dụ thực tế	Nhà kho Amazon	Dịch vụ giao hàng
Trong Spring	Spring Container	`@Autowired` annotation

Quy trình hoạt động kết hợp:

```
Bước 1: Ứng dụng khởi động
    |
    ▼
Bước 2: IoC Container được tạo (mở nhà kho)
    |
    ▼
```

```
Bước 3: IoC Container quét và tạo Bean (nhập hàng vào kho)
    │
    │   @Component
    │   public class BookingService { }
    │
    ▼
Bước 4: Bean được lưu vào "kho" (IoC Container)
    │
    │   [Kho chứa: BookingService, UserRepository, ...]
    │
    ▼
Bước 5: Khi gặp @Autowired (có người đặt hàng)
    │
    │   @Autowired
    │   private BookingService service;
    │
    ▼
Bước 6: Dependency Injection hoạt động (giao hàng)
    │
    │   IoC Container tìm BookingService trong "kho"
    │   → Lấy ra
    │   → Gán vào biến service
    │
    ▼
Bước 7: Bạn có thể sử dụng Bean ngay
    │
    │   service.createBooking(); // ✅ Hoạt động!
```

Tóm lại:

1. **IoC Container** = Nhà kho (nơi lưu trữ Bean)
2. **Dependency Injection** = Cách lấy hàng từ kho và đưa cho bạn
3. **Mối quan hệ:** IoC Container sử dụng Dependency Injection để cung cấp Bean
4. **Trong Spring:**
    - IoC Container = Spring Container (tự động tạo khi ứng dụng khởi động)
    - Dependency Injection = `@Autowired` (cách yêu cầu và nhận Bean)

Lưu ý quan trọng:

- Không thể có Dependency Injection mà không có IoC Container (giống như không thể giao hàng mà không có kho)
- IoC Container là nền tảng, Dependency Injection là cách sử dụng nền tảng đó
- Trong Spring, cả hai hoạt động cùng nhau một cách tự động

---

4. Bean Lifecycle - Vòng đời của Bean

4.1. Singleton vs Prototype

Giải thích: Khi Spring Container tạo Bean, nó có thể tạo 1 lần và dùng lại (Singleton) hoặc tạo mới mỗi lần cần (Prototype).

Singleton (mặc định):

- Spring tạo Bean 1 lần duy nhất khi ứng dụng khởi động
- Mọi nơi sử dụng đều dùng chung 1 đối tượng
- Giống như: Bạn có 1 chiếc xe, mọi người trong gia đình dùng chung

```
@Component  // Mặc định là Singleton
public class EmailService {
    public EmailService() {
        System.out.println("EmailService được tạo!");
    }

    public void sendEmail(String to) {
        System.out.println("Gửi email đến: " + to);
    }
}

// Sử dụng
@Service
public class BookingService {
    @Autowired
    private EmailService emailService1;  // Dùng chung 1 đối tượng

    @Autowired
    private EmailService emailService2;  // Cũng dùng chung 1 đối tượng
}
```

Kết quả: Console sẽ chỉ in "EmailService được tạo!" 1 lần, vì Spring chỉ tạo 1 đối tượng.

Prototype:

- Spring tạo Bean mới mỗi lần cần
- Mỗi nơi sử dụng có đối tượng riêng
- Giống như: Mỗi người có 1 chiếc xe riêng

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)  // Đánh dấu là Prototype
public class EmailService {
    public EmailService() {
        System.out.println("EmailService được tạo!");
    }

    public void sendEmail(String to) {
        System.out.println("Gửi email đến: " + to);
    }
}

// Sử dụng
@Service
public class BookingService {
```

```
    @Autowired
    private EmailService emailService1;  // Có 1 đối tượng riêng

    @Autowired
    private EmailService emailService2;  // Có 1 đối tượng riêng khác
}
```

Kết quả: Console sẽ in "EmailService được tạo!" 2 lần, vì Spring tạo 2 đối tượng riêng biệt.

Khi nào dùng Singleton?

- Khi Bean không có state (không lưu trữ dữ liệu riêng)
- Khi muốn tiết kiệm bộ nhớ
- **Hầu hết các trường hợp dùng Singleton** (mặc định)

Khi nào dùng Prototype?

- Khi Bean có state riêng (mỗi nơi sử dụng cần state khác nhau)
- Khi Bean không thread-safe (không an toàn khi nhiều thread cùng dùng)

4.2. Bean Lifecycle Hooks

Giải thích: Bean có vòng đời từ khi được tạo đến khi bị hủy. Bạn có thể can thiệp vào các giai đoạn này.

Các giai đoạn:

1. **Instantiation:** Spring tạo đối tượng (gọi constructor)
2. **Population:** Spring tự động cung cấp dependencies (@Autowired) - Gán các đối tượng cần thiết vào biến
3. **Initialization:** Bean đã sẵn sàng sử dụng
4. **Destruction:** Bean bị hủy (khi ứng dụng tắt)

Ví dụ sử dụng @PostConstruct và @PreDestroy:

```
@Component
public class DatabaseConnection {

    public DatabaseConnection() {
        System.out.println("1. Constructor được gọi – Đối tượng được
tạo");
    }

    @Autowired
    public void setConfig(Config config) {
        System.out.println("2. Dependencies được tự động cung cấp (gán
vào)");
    }

    @PostConstruct  // Chạy sau khi Bean đã được tạo và dependencies đã
được tự động cung cấp
```

```
    public void init() {
        System.out.println("3. @PostConstruct - Khởi tạo kết nối
database");
        // Thường dùng để: Kết nối database, load config, khởi tạo
resources
    }

    @PreDestroy  // Chạy trước khi Bean bị hủy (khi ứng dụng tắt)
    public void cleanup() {
        System.out.println("4. @PreDestroy - Đóng kết nối database");
        // Thường dùng để: Đóng kết nối, giải phóng resources
    }
}
```

Kết quả khi chạy ứng dụng:

```
1. Constructor được gọi - Đối tượng được tạo
2. Dependencies được tự động cung cấp (gán vào)
3. @PostConstruct - Khởi tạo kết nối database
... (ứng dụng chạy) ...
4. @PreDestroy - Đóng kết nối database (khi tắt ứng dụng)
```

---

5. VS Code Productivity - Làm việc hiệu quả với VS Code

5.1. Go to Definition (Đi đến định nghĩa)

Giải thích: Khi bạn thấy một class, method, hoặc biến và muốn xem nó được định nghĩa ở đâu, bạn có thể "nhảy" đến đó ngay lập tức.

Cách sử dụng:

- **Phím tắt:** `F12` hoặc `Cmd+Click` (macOS) / `Ctrl+Click` (Windows/Linux)
- **Ví dụ:** Click vào `@Autowired` → Nhấn `F12` → VS Code sẽ mở file định nghĩa của `@Autowired`

Thực hành:

1. Mở file Java có sử dụng `@Autowired`
2. Click vào `@Autowired`
3. Nhấn `F12` → Xem file định nghĩa được mở

5.2. Find References (Tìm tất cả nơi sử dụng)

Giải thích: Khi bạn muốn biết một class, method, hoặc biến được sử dụng ở những đâu trong project.

Cách sử dụng:

- **Phím tắt:** `Shift+F12`

- **Ví dụ:** Click vào class `TennisCoach` → Nhấn `Shift+F12` → VS Code sẽ hiển thị tất cả nơi sử dụng `TennisCoach`

Thực hành:

1. Mở file Java có class `TennisCoach`
2. Click vào tên class `TennisCoach`
3. Nhấn `Shift+F12` → Xem danh sách tất cả nơi sử dụng `TennisCoach`

5.3. Quick Fix (Sửa lỗi nhanh)

Giải thích: Khi VS Code phát hiện lỗi, nó thường có gợi ý cách sửa. Quick Fix giúp bạn sửa lỗi chỉ với 1 cú click.

Cách sử dụng:

- **Phím tắt:** `Cmd+.` (macOS) / `Ctrl+.` (Windows/Linux)
- **Ví dụ:**
  - Thiếu import → VS Code gợi ý "Add import"
  - Thiếu method → VS Code gợi ý "Create method"

Thực hành:

1. Tạo một class mới và sử dụng `@Autowired` nhưng chưa import
2. VS Code sẽ hiển thị lỗi đỏ
3. Click vào dòng lỗi → Nhấn `Cmd+.` / `Ctrl+.` → Chọn "Add import for org.springframework.beans.factory.annotation.Autowired"

5.4. IntelliSense (Gợi ý code tự động)

Giải thích: IntelliSense giống như một trợ lý thông minh. Khi bạn gõ code, nó tự động gợi ý những gì bạn có thể gõ tiếp theo.

Các tính năng:

- **Auto-complete:** Gợi ý tên method, biến khi bạn gõ
- **Parameter hints:** Hiển thị tham số cần truyền vào method
- **Quick info:** Hiển thị thông tin về method, class khi hover chuột

Ví dụ:

```
@Service
public class BookingService {
    @Autowired
    private UserRepository userRepository;

    public void test() {
        // Gõ "userRepository." → VS Code tự động gợi ý các method:
save(), findById(), findAll()...
```

```
        userRepository.  // ← Gõ dấu chấm, VS Code sẽ hiện danh sách
method
    }
}
```

Thực hành:

1. Tạo một class mới với `@Autowired UserRepository`
2. Gõ `userRepository.` → Xem VS Code gợi ý các method
3. Hover chuột vào method → Xem thông tin chi tiết

5.5. Peek Definition (Xem định nghĩa trong popup)

Giải thích: Peek Definition giống như "xem trước" định nghĩa mà không cần rời khỏi file hiện tại.

Cách sử dụng:

- **Phím tắt:** `Alt+F12`
- **Ví dụ:** Click vào `@Autowired` → Nhấn `Alt+F12` → Xem định nghĩa trong popup nhỏ, không cần mở file mới

So sánh với Go to Definition:

- **Go to Definition (`F12`):** Mở file mới, bạn phải quay lại file cũ
- **Peek Definition (`Alt+F12`):** Xem trong popup, không cần rời file hiện tại

Thực hành:

1. Mở file Java có sử dụng `@Autowired`
2. Click vào `@Autowired`
3. Nhấn `Alt+F12` → Xem định nghĩa trong popup
4. Nhấn `Esc` để đóng popup

Thực hành

Bài tập 1: Tạo Interface Coach và các Implementation

Mục tiêu: Hiểu cách sử dụng Interface và cách Spring quản lý các Bean

Yêu cầu:

1. **Tạo Interface Coach:**
    - Tạo package: `com.taxi.booking.coach`
    - Tạo interface `Coach` với method `String train()`
2. **Tạo các class implement Coach:**
    - `TennisCoach` implements `Coach` → Method `train()` trả về "Đánh bóng tennis 100 lần!"
    - `FootballCoach` implements `Coach` → Method `train()` trả về "Chạy 5 vòng sân!"

- `BasketballCoach` implements `Coach` → Method `train()` trả về "Ném bóng rổ 50 lần!"

3. **Đánh dấu các class là @Component:**
   - Thêm `@Component` vào mỗi class để Spring Container quản lý

Code mẫu:

```
// File: src/main/java/com/taxi/booking/coach/Coach.java
package com.taxi.booking.coach;

public interface Coach {
    String train();
}
```

```
// File: src/main/java/com/taxi/booking/coach/TennisCoach.java
package com.taxi.booking.coach;

import org.springframework.stereotype.Component;

@Component
public class TennisCoach implements Coach {
    @Override
    public String train() {
        return "Đánh bóng tennis 100 lần!";
    }
}
```

```
// File: src/main/java/com/taxi/booking/coach/FootballCoach.java
package com.taxi.booking.coach;

import org.springframework.stereotype.Component;

@Component
public class FootballCoach implements Coach {
    @Override
    public String train() {
        return "Chạy 5 vòng sân!";
    }
}
```

```
// File: src/main/java/com/taxi/booking/coach/BasketballCoach.java
package com.taxi.booking.coach;

import org.springframework.stereotype.Component;
```

```
@Component
public class BasketballCoach implements Coach {
    @Override
    public String train() {
        return "Ném bóng rổ 50 lần!";
    }
}
```

Kết quả mong đợi:

- ✅ Đã tạo interface `Coach` và 3 class implement
- ✅ Tất cả class đều có `@Component`
- ✅ Code compile không lỗi

Bài tập 2: Inject Coach vào MainApp và Test

Mục tiêu: Hiểu cách Spring tự động inject dependency

Yêu cầu:

1. **Tạo class MainApp:**
    - Tạo package: `com.taxi.booking`
    - Tạo class `MainApp` với `@Component`
    - Inject `Coach` vào MainApp bằng `@Autowired`
2. **Tạo method test:**
    - Tạo method `testCoach()` để gọi `coach.train()` và in kết quả
3. **Chạy ứng dụng và test:**
    - Chạy Spring Boot app
    - Gọi method `testCoach()` để xem kết quả

Code mẫu:

```
// File: src/main/java/com/taxi/booking/MainApp.java
package com.taxi.booking;

import com.taxi.booking.coach.Coach;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.stereotype.Component;

@SpringBootApplication
public class TaxiBookingBackendApplication {
    public static void main(String[] args) {
        SpringApplication.run(TaxiBookingBackendApplication.class, args);
```

```
    }
}

@Component
class CoachRunner implements CommandLineRunner {
    @Autowired
    private Coach coach;  // Spring sẽ tự động inject một trong các Coach
(TennisCoach, FootballCoach, hoặc BasketballCoach)

    @Override
    public void run(String... args) throws Exception {
        System.out.println("=== Kết quả training ===");
        System.out.println(coach.train());
    }
}
```

Lưu ý:

- Nếu có nhiều class implement `Coach`, Spring sẽ báo lỗi vì không biết chọn class nào
- Để fix, bạn có thể:
  - Chỉ giữ lại 1 class có `@Component` (comment `@Component` ở các class khác)
  - Hoặc dùng `@Primary` để đánh dấu class ưu tiên
  - Hoặc dùng `@Qualifier` để chỉ định class cụ thể

Kết quả mong đợi:

- ✅ Ứng dụng chạy thành công
- ✅ Console in ra kết quả training (ví dụ: "Đánh bóng tennis 100 lần!")

Bài tập 3: Practice Navigation trong VS Code

Mục tiêu: Làm quen với các tính năng navigation trong VS Code

Yêu cầu:

1. **Go to Definition:**
   - Mở file `MainApp.java`
   - Click vào `@Autowired` → Nhấn `F12` → Xem file định nghĩa
   - Click vào `Coach` → Nhấn `F12` → Xem interface `Coach`
2. **Find References:**
   - Click vào class `TennisCoach` → Nhấn `Shift+F12` → Xem tất cả nơi sử dụng `TennisCoach`
3. **Peek Definition:**
   - Click vào `@Component` → Nhấn `Alt+F12` → Xem định nghĩa trong popup
4. **Quick Fix:**
   - Xóa import `@Component` → VS Code sẽ báo lỗi

- Click vào dòng lỗi → Nhấn `Cmd+.` / `Ctrl+.` → Chọn "Add import"

5. **IntelliSense:**
   - Trong method `testCoach()`, gõ `coach.` → Xem VS Code gợi ý method `train()`
   - Hover chuột vào method `train()` → Xem thông tin chi tiết

Kết quả mong đợi:

- ✅ Đã thử tất cả các tính năng navigation
- ✅ Hiểu cách sử dụng các phím tắt
- ✅ Cảm thấy làm việc với VS Code nhanh hơn

Bài tập 4: Hiểu Bean Lifecycle (Singleton vs Prototype)

Mục tiêu: Hiểu sự khác biệt giữa Singleton và Prototype

Yêu cầu:

1. **Tạo class EmailService (Singleton - mặc định):**

```
@Component
public class EmailService {
    public EmailService() {
        System.out.println("EmailService được tạo! HashCode: " +
this.hashCode());
    }
}
```

2. **Inject EmailService vào 2 class khác nhau:**

```
@Component
class Service1 {
    @Autowired
    private EmailService emailService;

    public void printHashCode() {
        System.out.println("Service1 - EmailService HashCode: " +
emailService.hashCode());
    }
}

@Component
class Service2 {
    @Autowired
    private EmailService emailService;

    public void printHashCode() {
        System.out.println("Service2 - EmailService HashCode: " +
emailService.hashCode());
```

```
    }
}
```

3. **Chạy và quan sát:**
    - Console sẽ chỉ in "EmailService được tạo!" 1 lần
    - HashCode của emailService trong Service1 và Service2 giống nhau (cùng 1 đối tượng)
4. **Thử đổi sang Prototype:**
    - Thêm `@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)` vào `EmailService`
    - Chạy lại → Console sẽ in "EmailService được tạo!" nhiều lần
    - HashCode khác nhau (mỗi nơi có đối tượng riêng)

Kết quả mong đợi:

- ✅ Hiểu sự khác biệt giữa Singleton và Prototype
- ✅ Biết khi nào dùng Singleton, khi nào dùng Prototype

Bài tập 5: Sử dụng @PostConstruct và @PreDestroy

Mục tiêu: Hiểu vòng đời của Bean và cách can thiệp

Yêu cầu:

1. **Tạo class DatabaseConnection:**

```
@Component
public class DatabaseConnection {

    public DatabaseConnection() {
        System.out.println("1. Constructor - DatabaseConnection được
tạo");
    }

    @PostConstruct
    public void init() {
        System.out.println("2. @PostConstruct - Khởi tạo kết nối
database");
    }

    public void connect() {
        System.out.println("3. Kết nối database thành công!");
    }

    @PreDestroy
    public void cleanup() {
        System.out.println("4. @PreDestroy - Đóng kết nối database");
    }
}
```

2. **Sử dụng DatabaseConnection:**

```
@Component
class DatabaseRunner implements CommandLineRunner {
    @Autowired
    private DatabaseConnection dbConnection;

    @Override
    public void run(String... args) throws Exception {
        System.out.println("=== Sử dụng DatabaseConnection ===");
        dbConnection.connect();
    }
}
```

3. **Chạy ứng dụng:**
    - Quan sát thứ tự in ra console
    - Tắt ứng dụng (Ctrl+C) → Quan sát @PreDestroy được gọi

Kết quả mong đợi:

- ✅ Hiểu thứ tự vòng đời của Bean
- ✅ Biết cách sử dụng @PostConstruct và @PreDestroy

Dự án Taxi

Chưa áp dụng vào dự án Taxi

Giải thích: Buổi 2 tập trung vào lý thuyết cốt lõi về OOP và Spring Core (DI/IoC). Chúng ta cần hiểu rõ các khái niệm này trước khi áp dụng vào dự án Taxi.

Ở các buổi sau, chúng ta sẽ áp dụng:

- Buổi 3: Tạo REST API cơ bản cho Taxi Booking System
- Buổi 4: Xử lý request/response cho API đặt xe
- Buổi 6: Tạo Entity và Repository cho User, Booking

Tuy nhiên, bạn có thể thử nghiệm:

- Tạo các interface và class liên quan đến Taxi (ví dụ: `PaymentService`, `NotificationService`)
- Thử inject các service này vào các class khác
- Practice navigation trong VS Code với code Taxi (khi có)

Tổng kết buổi 2

Những gì đã học:

1. ✅ Hiểu Interface và Abstract Class trong Java

2. ✅ Hiểu Inversion of Control (IoC) Container
3. ✅ Hiểu Dependency Injection (DI) và các annotation quan trọng:
    - Annotation tạo Bean: `@Component`, `@Service`, `@Repository`, `@Controller`
    - Annotation cấu hình: `@Configuration`, `@Bean`
    - Annotation inject: `@Autowired`
    - Annotation chọn Bean: `@Primary`, `@Qualifier`
    - Annotation đọc config: `@Value`
4. ✅ Hiểu Bean Lifecycle: Singleton vs Prototype
5. ✅ Biết cách sử dụng `@PostConstruct` và `@PreDestroy`
6. ✅ Làm quen với VS Code Productivity tools: Go to Definition, Find References, Quick Fix, IntelliSense, Peek Definition

Kiến thức quan trọng:

- **IoC Container:** Spring tự động tạo và quản lý các Bean (giống như một "nhà kho" thông minh)
- **Dependency Injection:** Spring tự động cung cấp (gán) đối tượng cần thiết vào nơi cần dùng
- **Annotation tạo Bean:**
    - `@Component`: Annotation cơ bản nhất
    - `@Service`: Dùng cho business logic
    - `@Repository`: Dùng cho database access
    - `@Controller`: Dùng cho HTTP request handling
- **Annotation cấu hình:**
    - `@Configuration`: Đánh dấu class cấu hình
    - `@Bean`: Tạo Bean thủ công từ method
- **Annotation inject:**
    - `@Autowired`: Tự động cung cấp đối tượng cần thiết
- **Annotation chọn Bean:**
    - `@Primary`: Đánh dấu Bean ưu tiên (khi có nhiều Bean cùng loại)
    - `@Qualifier`: Chỉ định Bean cụ thể (khi có nhiều Bean cùng loại)
- **Annotation đọc config:**
    - `@Value`: Đọc giá trị từ `application.properties`
- **Singleton (mặc định):** Spring tạo Bean 1 lần, mọi nơi dùng chung
- **Prototype:** Spring tạo Bean mới mỗi lần cần

Chuẩn bị cho buổi 3:

- ✅ Đã hiểu cách Spring quản lý Bean
- ✅ Đã biết cách inject dependency
- ✅ Đã làm quen với VS Code navigation tools
- ✅ Sẵn sàng học Spring Boot và REST API

Kiểm tra lại trước buổi 3:

- ☐ Đã tạo được Interface Coach và các class implement
- ☐ Đã inject Coach vào MainApp thành công
- ☐ Đã thử các tính năng navigation trong VS Code (F12, Shift+F12, Alt+F12)
- ☐ Hiểu sự khác biệt giữa Singleton và Prototype
- ☐ Đã thử @PostConstruct và @PreDestroy

Bài tập về nhà (tùy chọn):

- Tạo thêm các interface và class khác (ví dụ: `PaymentService`, `NotificationService`)
- Thử inject nhiều dependency vào một class
- Thử dùng `@Primary` và `@Qualifier` để chọn Bean cụ thể
- Thử dùng `@Value` để đọc config từ `application.properties`
- Thử dùng `@Configuration` và `@Bean` để tạo Bean thủ công
- Đọc thêm về Spring Framework tại https://spring.io/projects/spring-framework
- Xem lại tất cả các annotation đã học: @Component, @Service, @Repository, @Controller, @Configuration, @Bean, @Autowired, @Primary, @Qualifier, @Value