Spring Framework Fundamental (Part 1)

Thang Nguyen Van

September 2022

Nash Tech.

Agenda

1. **Spring Framework Eco-system**
2. **Object Initialization Problem**
   - Restaurant Business
3. **Spring ApplicationContext & Beans System**
   - Service Locator pattern
4. **Spring Dependencies Injection with @Component & @Autowire**
   - Dependency Injection Capabilities
   - @ComponentScan

Agenda

5. **Dependency Resolution Process**
   - Circular Dependency
   - @Lazy
   - Setter Workaround
6. **Bean Selection**
   - @Primary
   - @Qualifier
7. **Extra Injection, Resource, Generics**
   - @Value
   - Configuration Properties Binding
   - Generics Collection Fill up

Agenda

8. **Bean Scope**
   - Singleton
   - Prototype
   - Web Scope
9. **Bean Life-Cycle Hook**
   - Post-Construct
   - Pre-Destroy
10. **Spring Profile Selection**
    - @Profile

01. Spring Framework eco-system

Spring Framework eco-system

Spring Core

- At the heart of Spring Framework, Spring Core provides:
  - Dependency Injection (**DI**) capabilities,
  - Aspect Oriented Programming (**AOP**) paradigm,
  - Spring Expression Language (**SpEL**)

Spring Boot

- Provides **Automatic Configuration** capabilities
- Out-of-the-box **Starter Packages**
- Ready-to-use, stand alone, **observable**, production-ready application

Spring Data

- Provides ease when interacting with Database Management System
- Common query pattern
- Standard API (**JPA**), Interface for 3rd party Database-Object-Mapper (like **Hibernate**…)

Spring Security

- Comprehensive **Authentication & Authorization** suite
- **OAuth2** authorization flow compatible, **OAuth2 client/OAuth2 resource server**

Spring Framework eco-system

Spring Cloud

- Provide common pattern & cloud integration packages to build **Distributed** & **Cloud Ready/Compatible/Native** application
- Cloud Routing/Discovery pattern
- Observabilities exposable library

Spring for Apache Kafka

- Integration with **Apache Kafka** as a messaging backer service
- Use Kafka as an **Event Sourcing** provider
- Use Kafka to build **Stream Processing** application

Spring for GraphQL

- If you want to build a backend in GraphQL flavor, **Spring for GraphQL** support standard structure, API, configuration suite to do that
- Dynamic Graph resolver
- Integrated with many Data Source systems

Spring Webflux Reactive

- When you work with high concurrency workloads, **Spring Webflux** provide Non-blocking, Asynchronous way to utilize system resources
- Provide high throughput & reduce system bottleneck

02. Object Initialization Problem

Simple Restaurant

- We startup our own business with a restaurant
- We serve traditional foods
- We hire a chef
- Everything went well
- And we have this

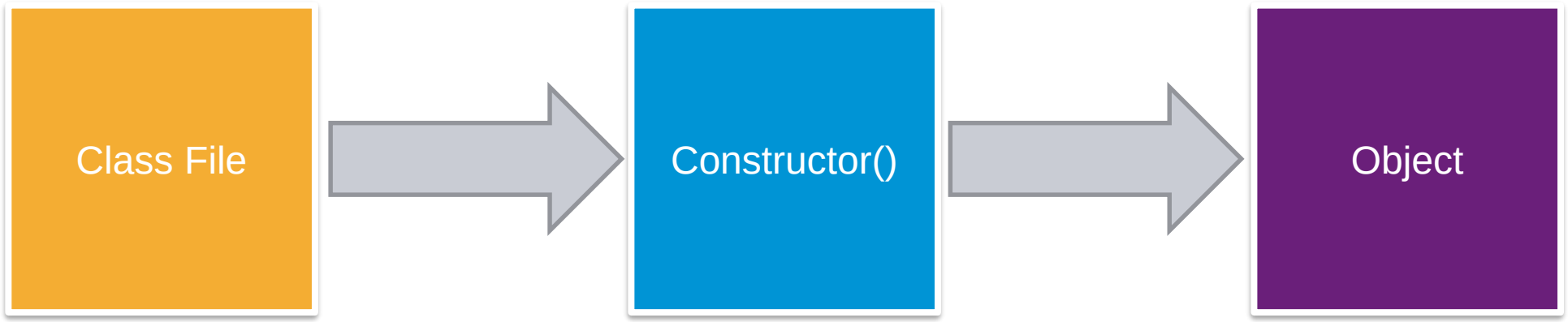
Simple Restaurant

```
import java.util.Arrays;
import java.util.List;

public class Chef {
    public List<String> cook() {
        return Arrays.asList("KFC", "Lotteria", "Texas Chicken");
    }
}
```

Simple Restaurant

```
import java.util.List;

public class Restaurant {

    public List<String> serve() {
        Chef chef = new Chef(); // Object initialization before using
        return chef.cook();
    }
}
```

Simple Restaurant



Complex Restaurant

- Bravo, our business grew up
- We now expand our restaurant and serve international flavors
- We now have Vietnamese Receipt & Thai Receipt
- We hire one more Chef, now we have 2, each one can cook one Receipt

Complex Restaurant

```
public abstract class ReceiptFlavor {
    public abstract List<String> getReceipt();
}
```

```
public class VietnameseReceiptFlavor extends ReceiptFlavor {
    @Override
    public List<String> getReceipt() {
        return Arrays.asList("sweet", "warm", "delicious");
    }
}
```

```
public class ThaiReceiptFlavor extends ReceiptFlavor {
    @Override
    public List<String> getReceipt() {
        return Arrays.asList("spicy", "hot", "sour");
    }
}
```

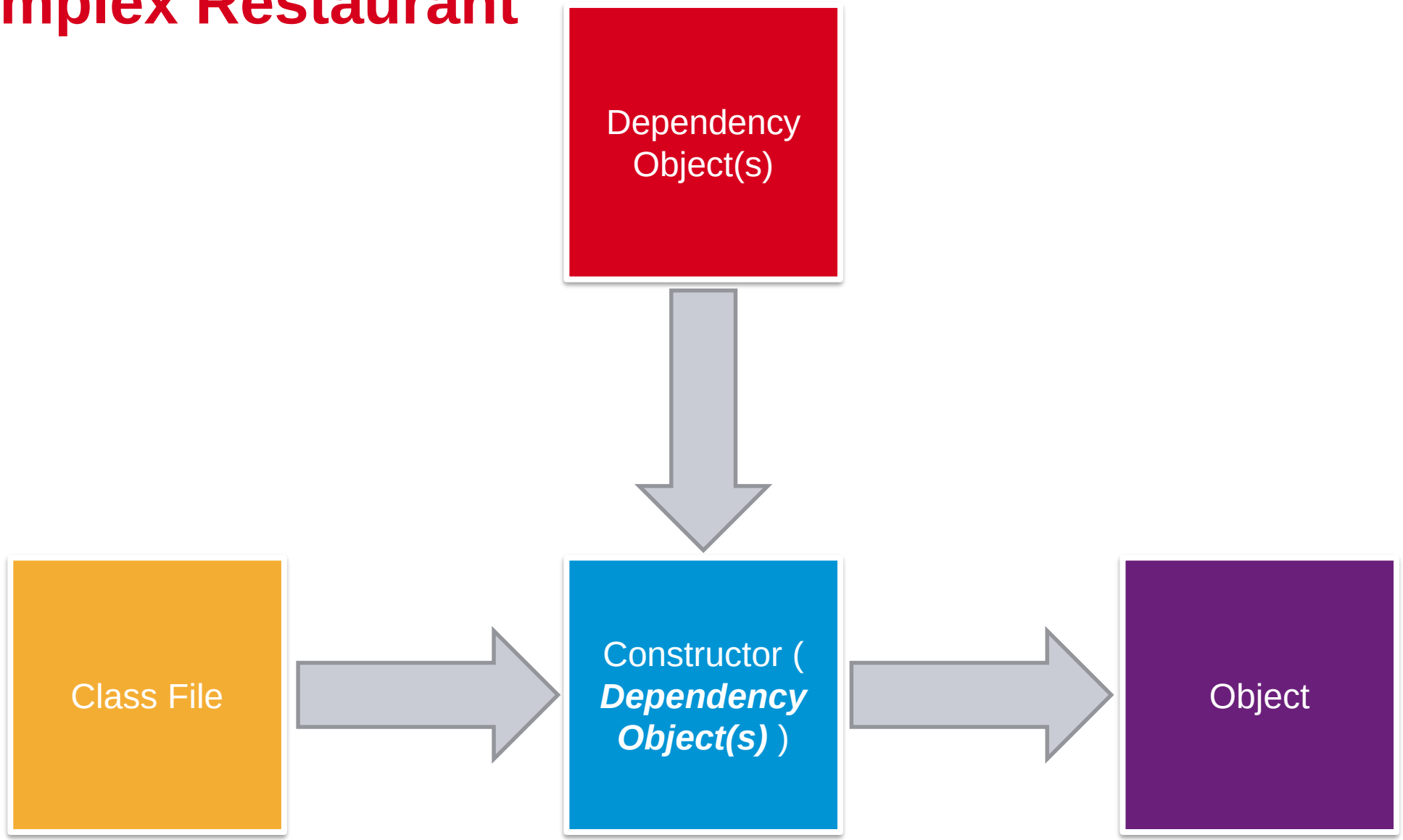
Complex Restaurant

```
import java.util.List;

public class NationalChef {

    private ReceiptFlavor receiptFlavor;

    public NationalChef(ReceiptFlavor receiptFlavor) {
        this.receiptFlavor = receiptFlavor;
    }

    public List<String> cook() {
        return this.receiptFlavor.getReceipt();
    }
}
```

Complex Restaurant

```
public class InternationalRestaurant {
    public List<String> serveVietnameseDishes() {
        List<String> dishes;

        ReceiptFlavor vietnameseFlavor = new VietnameseReceiptFlavor(); // We have to initialize ReceiptFlavor first
        NationalChef vietnameseChef = new NationalChef(vietnameseFlavor); // Initialization of NationalChef depends on Receipt Flavor
        dishes = vietnameseChef.cook();

        return dishes;
    }

    public List<String> serveThaiDishes() {
        List<String> dishes;

        ReceiptFlavor thaiFlavor = new ThaiReceiptFlavor(); // We have to initialize ReceiptFlavor first
        NationalChef thaiChef = new NationalChef(thaiFlavor); // Initialization of NationalChef depends on Receipt Flavor
        dishes = thaiChef.cook();

        return dishes;
    }
}
```

Complex Restaurant

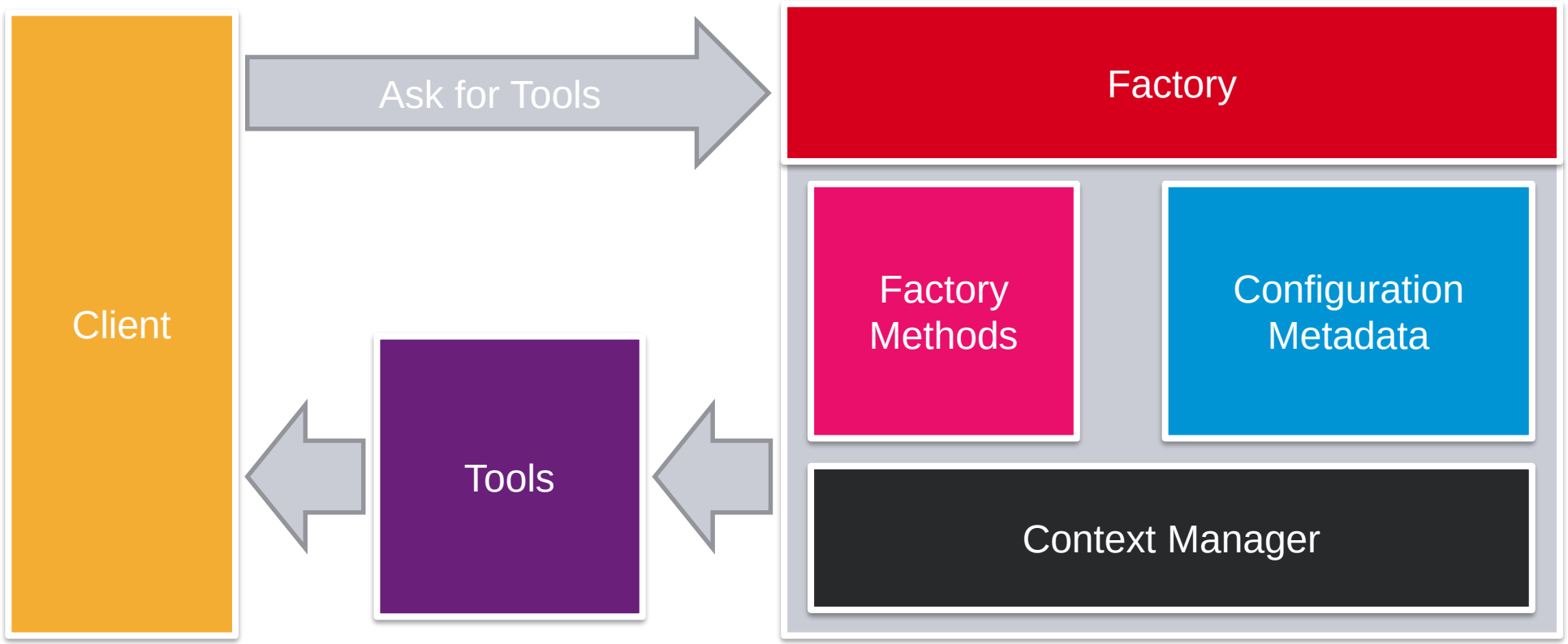


02.01 Factory Method

Factory Method

- Let Factory build the Tools that you need
- Just tell them what Tools that you want

Factory Method



Factory Method

```
public class FactoryMethod {
    public NationalChef giveMeAChef(String flavor) {
        if ("VietnameseFlavor".equals(flavor)) {
            ReceiptFlavor vietnameseFlavor = new VietnameseReceiptFlavor();
            NationalChef vietnameseChef = new NationalChef(vietnameseFlavor);
            return vietnameseChef;
        } else if ("ThaiFlavor".equals(flavor)) {
            ReceiptFlavor thaiFlavor = new ThaiReceiptFlavor();
            NationalChef thaiChef = new NationalChef(thaiFlavor);
            return thaiChef;
        }

        return null;
    }
}
```

Factory Method

```
public class InternationalRestaurant {
    private FactoryMethod factoryMethod;

    public InternationalRestaurant(FactoryMethod factoryMethod) {
        this.factoryMethod = factoryMethod;
    }

    public List<String> serveVietnameseDishes() {
        List<String> dishes;

        NationalChef vietnameseChef = factoryMethod.giveMeAChef("VietnameseFlavor");
        dishes = vietnameseChef.cook();

        return dishes;
    }

    public List<String> serveThaiDishes() {
        List<String> dishes;

        NationalChef thaiChef = factoryMethod.giveMeAChef("ThaiFlavor");
        dishes = thaiChef.cook();

        return dishes;
    }
}
```

03. Spring ApplicationContext & Beans System

Revise the Factory Method

```
public class InternationalRestaurant {
   private ApplicationContext applicationContext;

   public InternationalRestaurant(ApplicationContext applicationContext) {
      this.applicationContext = applicationContext;
   }

   public List<String> serveVietnameseDishes() {
      List<String> dishes;

      NationalChef vietnameseChef = (NationalChef) applicationContext.getBean("VietnameseChef");
      dishes = vietnameseChef.cook();

      return dishes;
   }

   public List<String> serveThaiDishes() {
      List<String> dishes;

      NationalChef thaiChef = (NationalChef) applicationContext.getBean("ThaiChef");
      dishes = thaiChef.cook();

      return dishes;
   }
}
```

ApplicationContext

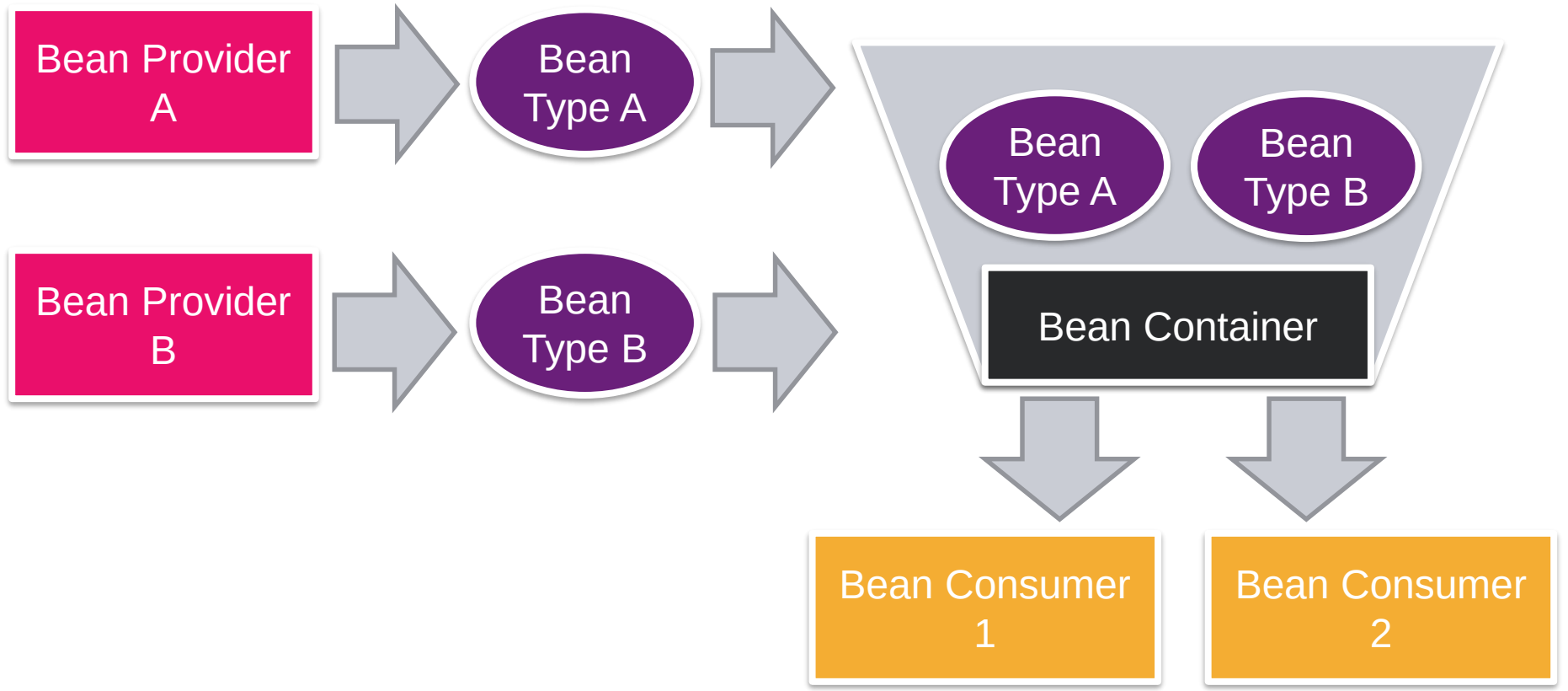
- Just like the **Factory Method** model
- Tell **ApplicationContext** which Tools that you need
- **ApplicationContext** will give you the Tools

03.01 Beans System

Beans

- A **Bean** is a Java Object, but **ApplicationContext** (Beans Container) **constructs & manages** it for you
- When you need a **Bean**, just ask **ApplicationContext,** tell it which **Bean** that you need
- But where do Beans come from, how **ApplicationContext** have those?

Beans


Bean Provider A
Bean Type A
Bean Provider B
Bean Type B
Bean Type A
Bean Type B
Bean Container
Bean Consumer 1
Bean Consumer 2

03.02 Method-Based Bean Provider

Method-Based Bean Provider

```
@Configuration
public class ReceiptConfiguration {

    @Bean(name = "ThaiReceiptFlavor")
    public ReceiptFlavor getThaiReceiptFlavor() {
        return new ThaiReceiptFlavor();
    }

    @Bean(name = "VietnameseReceiptFlavor")
    public ReceiptFlavor getVietnameseReceiptFlavor() {
        return new VietnameseReceiptFlavor();
    }
}
```

Method-Based Bean Provider

```
@Configuration
public class ChefConfiguration {

   @Bean(name = "ThaiChef")
   public NationalChef getThaiChef(
         @Autowired @Qualifier("ThaiReceiptFlavor") ReceiptFlavor receiptFlavor
   ) {
      return new NationalChef(receiptFlavor);
   }

   @Bean(name = "VietnameseChef")
   public NationalChef getVietnameseChef(
         @Autowired @Qualifier("VietnameseReceiptFlavor") ReceiptFlavor receiptFlavor
   ) {
      return new NationalChef(receiptFlavor);
   }
}
```

Method-Based Bean Provider

- To mark a method as a **Method-Based Bean Provider**, you mark it with **@Bean** Annotation
- To mark a **Class** that contains one or multiple **Method-Based Bean Provider(s)**, you mark it with **@Configuration** Annotation (this is for Spring Component Scanner, as well as for declarative, readable purpose)
- When you have multiple Bean Providers that produce **Beans of the same Type**, you should better give each Bean Provider an **explicit Bean Name**
- Method-Based Bean Provider be also called as **Java-Style Configuration**

03.03 Class-Based Bean Provider

Class-Based Bean Provider

```
@Component("ThaiReceiptFlavor")
public class ThaiReceiptFlavor extends ReceiptFlavor {
    @Override
    public List<String> getReceipt() {
        return Arrays.asList("spicy", "hot", "sour");
    }
}
```

```
@Component("VietnameseReceiptFlavor")
public class VietnameseReceiptFlavor extends ReceiptFlavor {
    @Override
    public List<String> getReceipt() {
        return Arrays.asList("sweet", "warm", "delicious");
    }
}
```

Class-Based Bean Provider

- To mark a class as a **Class-Based Bean Provider**, you mark it with **@Component** Annotation
- **@Component** Annotation is for Spring Component Scanner, as well as for declarative, readable purpose
- When you have multiple Bean Providers that produce **Beans of the same Type**, you should better give each Bean Provider an **explicit Bean Name**
- Class-Based Bean Provider be also called as **Annotation-Style Configuration**

03.04 Register Bean Provider with ApplicationContext

Register Bean Provider when defining Context

```
public class Main {
    public static void main(String... args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(
                ChefConfiguration.class,
                ReceiptConfiguration.class
        );

        InternationalRestaurant restaurant = new InternationalRestaurant(context);
        System.out.println(restaurant.serveThaiDishes());
        System.out.println(restaurant.serveVietnameseDishes());
    }
}
```

Register Bean Provider with Master Config File

```
public class Main {
    public static void main(String... args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(MasterConfigurationFile.class);

        InternationalRestaurant restaurant = new InternationalRestaurant(context);
        System.out.println(restaurant.serveThaiDishes());
        System.out.println(restaurant.serveVietnameseDishes());
    }
}
```

```
@Configuration
@Import({ReceiptConfiguration.class, ChefConfiguration.class})
public class MasterConfigurationFile {
}
```

Register Bean Provider by @ComponentScan

- This is the most common Style to register Bean Providers with ApplicationContext
- Let talk about @**ComponentScan** in next slides, along side with Dependency Injection

04. Spring Dependencies Injection with @Component & @Autowire

Take a look at Service Locator pattern

```
public class InternationalRestaurant {
   private ApplicationContext applicationContext;

   public InternationalRestaurant(ApplicationContext applicationContext) {
      this.applicationContext = applicationContext;
   }

   public List<String> serveVietnameseDishes() {
      List<String> dishes;

      NationalChef vietnameseChef = (NationalChef) applicationContext.getBean("VietnameseChef");
      dishes = vietnameseChef.cook();

      return dishes;
   }

   public List<String> serveThaiDishes() {
      List<String> dishes;

      NationalChef thaiChef = (NationalChef) applicationContext.getBean("ThaiChef");
      dishes = thaiChef.cook();

      return dishes;
   }
}
```

Take a look at Service Locator pattern

- Are you tired when passing **ApplicationContext** around?
- Are you tired to call **ApplicationContext.getBean(…)** whenever you need a Bean to use?
- Let Inject Beans automatically by using @**Component** & @**Autowire**

Dependency Injection with Annotation

```
@Component
public class InternationalRestaurant {

    NationalChef vietnameseChef;
    NationalChef thaiChef;

    public InternationalRestaurant(
            @Autowired @Qualifier("VietnameseChef") NationalChef vietnameseChef,
            @Autowired @Qualifier("ThaiChef") NationalChef thaiChef
    ) {
        this.vietnameseChef = vietnameseChef;
        this.thaiChef = thaiChef;
    }

    public List<String> serveVietnameseDishes() {
        return vietnameseChef.cook();
    }

    public List<String> serveThaiDishes() {
        return thaiChef.cook();
    }
}
```

04.01 Enable Spring Component Auto Scan mode

Enable Spring Component Auto Scan mode

- Before your classes enjoying the magic of Dependency Injection, let enable **Component Scan mode** first:

```
@Configuration
@ComponentScan(basePackages = "com.thangok.injectstyle")
//@Import({ReceiptConfiguration.class, ChefConfiguration.class})
public class MasterConfigurationFile {
}
```

Enable Spring Component Auto Scan mode

- When you have @**ComponentScan** apply on a **Package**, Spring Component Scanner will do this for you:
  - All **Method-Based Bean Providers** (method that marked with @**Bean** inside classes that marked with @**Configuration**) of this **Package** and its **Sub-Packages** will automatically scanned and **registered** with **ApplicationContext**
  - All **Class-Based Bean Providers** (classes that marked with @**Component**) of this **Package** and its **Sub-Packages** will automatically scanned and **registered** with **ApplicationContext**
- At the same time, Spring Component Scanner will bring Spring Components **join** into **ApplicationContext**:
  - All Spring Components (classes that marked with @**Component**) of this **Package** and its **Sub-Packages** will automatically scanned and **joined** into **ApplicationContext**
  - All Spring Components after joined into **ApplicationContext** will have **Dependency Injection Capability**

Dependency Injection Capability

```
@Component
public class InternationalRestaurant {

    NationalChef vietnameseChef;
    NationalChef thaiChef;

    public InternationalRestaurant(
            @Autowired @Qualifier("VietnameseChef") NationalChef vietnameseChef,
            @Autowired @Qualifier("ThaiChef") NationalChef thaiChef
    ) {
        this.vietnameseChef = vietnameseChef;
        this.thaiChef = thaiChef;
    }

    public List<String> serveVietnameseDishes() {
        return vietnameseChef.cook();
    }

    public List<String> serveThaiDishes() {
        return thaiChef.cook();
    }
}
```

Spring Components

- Are classes that marked with @**Component**, inside packages that enabled @**ComponentScan**
- It is also a **Bean Provider** (Class-Based Bean Provider)
- @**Configuration** is also a @**Component** (@Configuration extends @Component)
- It has some common Sub-Annotation Types (Stereo Type) that you usually work with:
  - @Controller
  - @Service
  - @Repository

04.02 Dependency Injection via Class Constructor

Class Constructor Injection

```
@Component
public class InternationalRestaurant {

   NationalChef vietnameseChef;
   NationalChef thaiChef;

   public InternationalRestaurant(
         @Autowired @Qualifier("VietnameseChef") NationalChef vietnameseChef,
         @Autowired @Qualifier("ThaiChef") NationalChef thaiChef
   ) {
      this.vietnameseChef = vietnameseChef;
      this.thaiChef = thaiChef;
   }
}
```

Class Constructor Injection

- To instruct ApplicationContext to inject bean via Class Constructor, let do:
  - Mark injecting Constructor with @**Autowire** (Optional)
  - Using @**Qualifier** when you have multiple **Beans of the same Type**
- When you really need a Class with **multiple Constructors**, one for Auto Injection, one for another purpose, let do:
  - Mark **one** Constructor with @**Autowire** to register this constructor should be injected by **ApplicationContext**
- If you need design some class flexible for injection, Eg: you have **multiple Constructors**, and let **ApplicationContext** choose the one it can Inject, base on the Beans in Container that it has at the time Dependencies Resolution occurs, let do:
  - Mark **two or more** Constructor with @**Autowire(required=false)**
  - Remember **required=false,** because you left the decision to **ApplicationContext**

Constructor Injection in case multiple constructors

```
@Component
public class InternationalRestaurant {

   NationalChef vietnameseChef;
   NationalChef thaiChef;

   @Autowired
   public InternationalRestaurant(
         @Autowired @Qualifier("VietnameseChef") NationalChef vietnameseChef,
         @Autowired @Qualifier("ThaiChef") NationalChef thaiChef
   ) {
      this.vietnameseChef = vietnameseChef;
      this.thaiChef = thaiChef;
   }

   public InternationalRestaurant(NationalChef vietnameseChef) {
      this.vietnameseChef = vietnameseChef;
   }
}
```

Constructor Injection in case multiple constructors

```
@Component
public class InternationalRestaurant {
    NationalChef vietnameseChef;
    NationalChef thaiChef;

    @Autowired(required = false)
    public InternationalRestaurant(
            @Autowired @Qualifier("VietnameseChef") NationalChef vietnameseChef,
            @Autowired @Qualifier("ThaiChef") NationalChef thaiChef
    ) {
        this.vietnameseChef = vietnameseChef;
        this.thaiChef = thaiChef;
    }

    @Autowired(required = false)
    public InternationalRestaurant(
            NationalChef vietnameseChef
    ) {
        this.vietnameseChef = vietnameseChef;
    }
}
```

04.03 Dependency Injection via Class Fields

Class Fields Injection

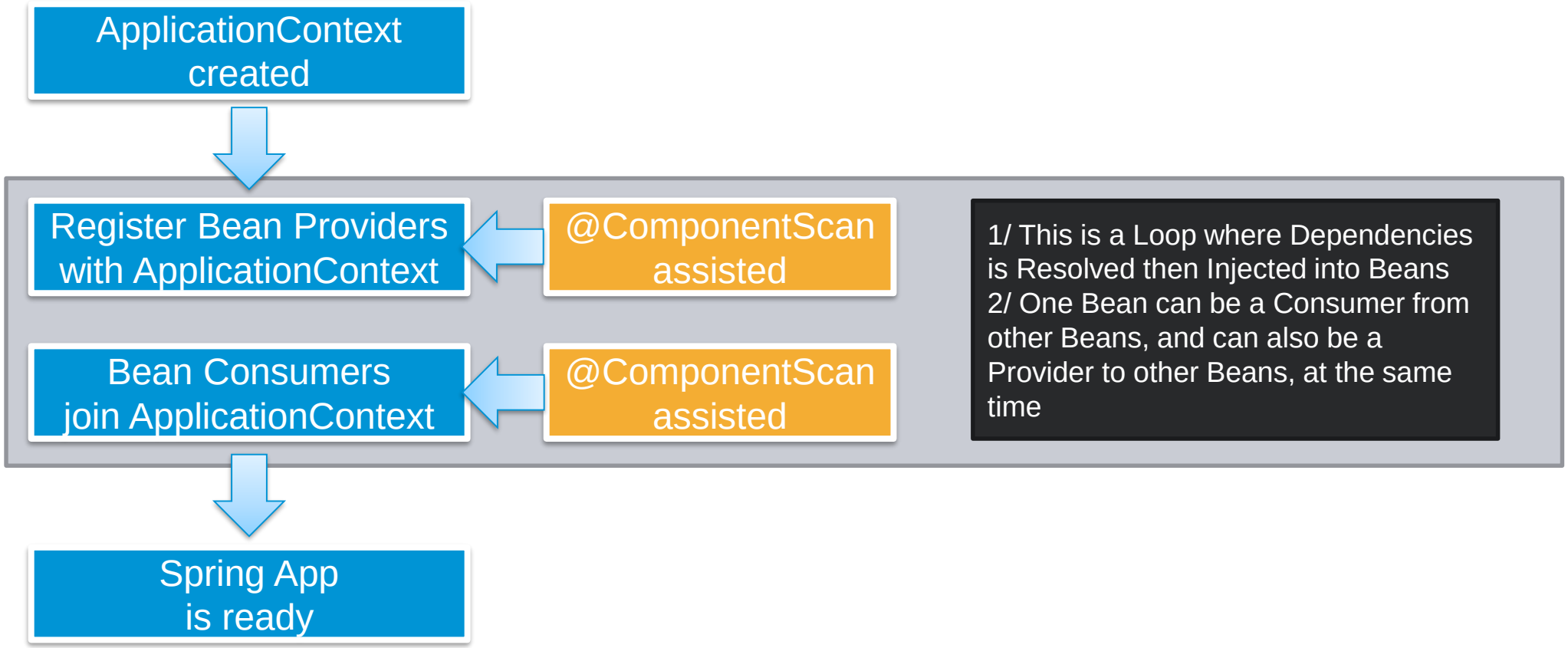
```
@Component
public class InternationalRestaurant {

    @Autowired
    @Qualifier("VietnameseChef")
    NationalChef vietnameseChef;

    @Autowired
    @Qualifier("ThaiChef")
    NationalChef thaiChef;

    public List<String> serveVietnameseDishes() {
        return vietnameseChef.cook();
    }

    public List<String> serveThaiDishes() {
        return thaiChef.cook();
    }
}
```

Class Fields Injection

- To instruct ApplicationContext to inject bean via **Class Fields**, let do:
  - Mark injecting Fields with @**Autowire**
  - Using @**Qualifier** when you have multiple **Beans of the same Type**
- When a class has **Class Constructor Injection** and **Class Fields Injection** togethers, ApplicationContext will do:
  - Inject via Class Constructor first
  - Inject via Class Fields then
  - => Class Fields Injection will override Class Constructor Injection if they are applied on the same field

05. Dependency Resolution Process

Dependency Resolution Process

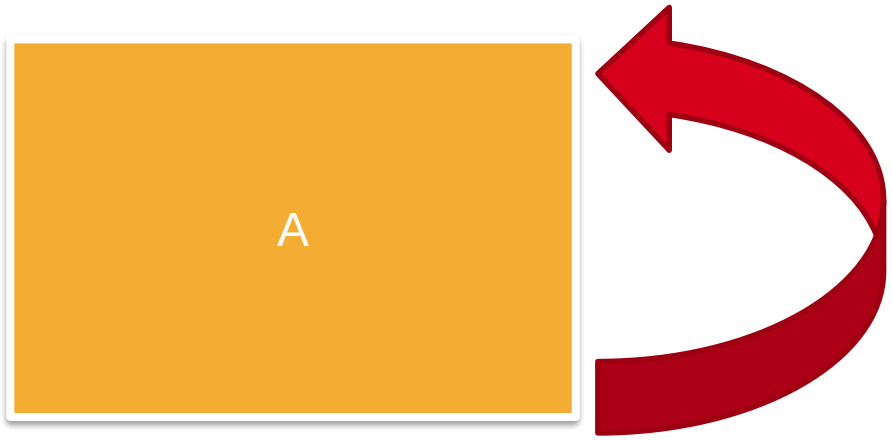


05.01 Circular Dependency

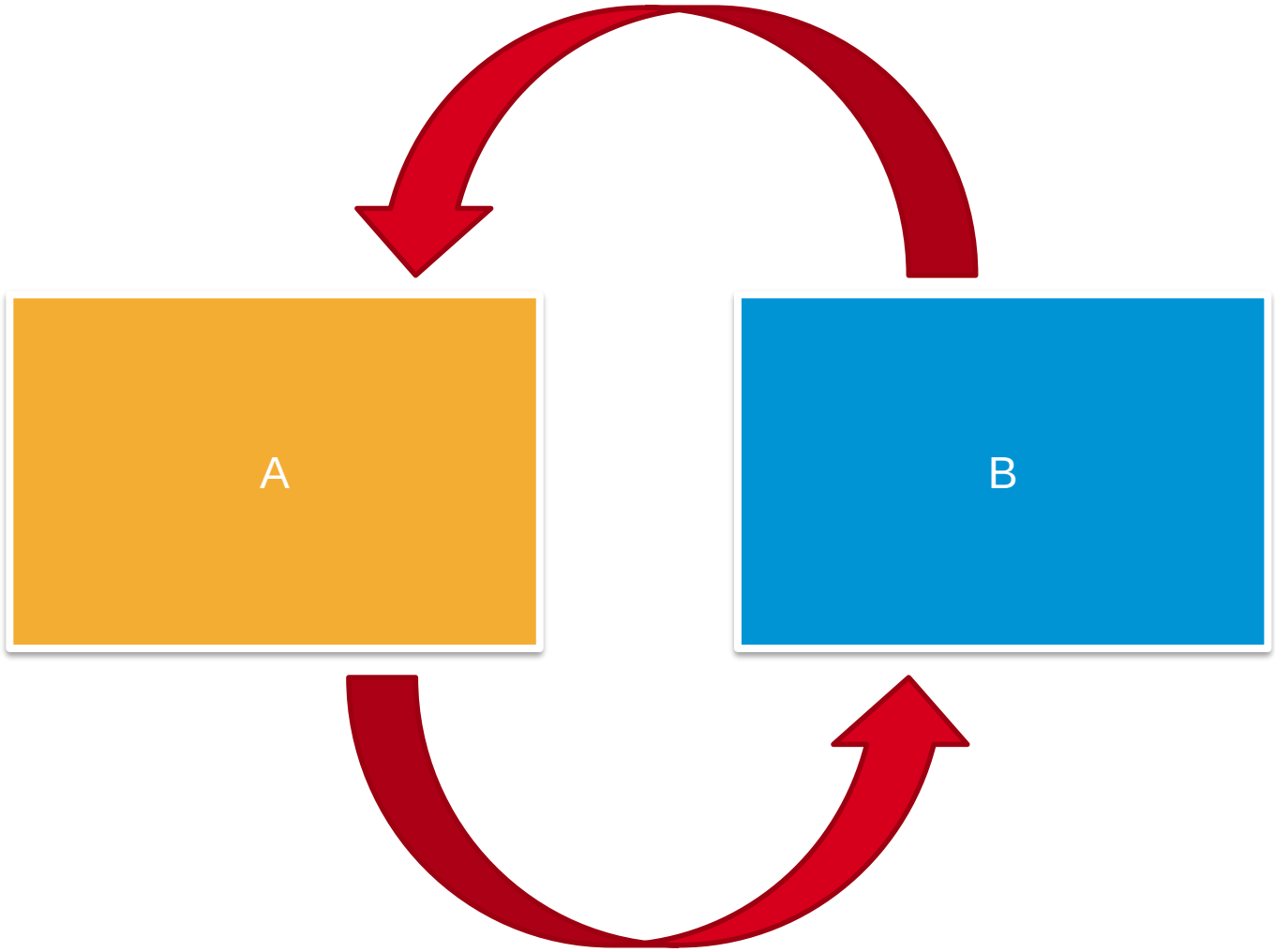
Business Upgrade

- Good news, our **International Restaurant** do it business very well
- Now we have enough money to invest **International Hotel**
- **International Restaurant** need Lobby from **International Hotel** to serve diners
- **International Hotel** need Beverages from **International Restaurant** to serve sleepers

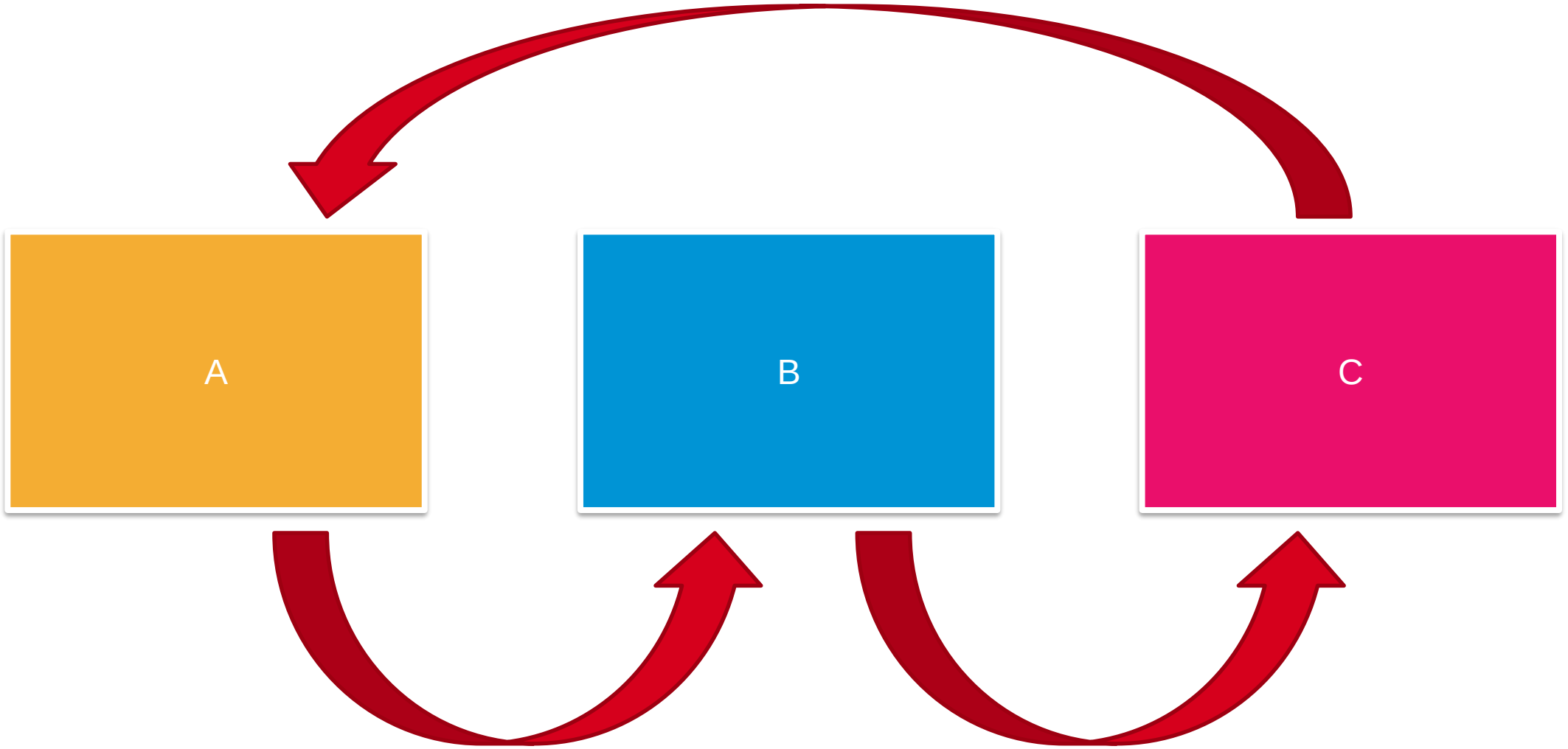
Circular Dependency Type 1



Circular Dependency Type 2



Circular Dependency Type 3



Circular Dependency

```
@Component
public class InternationalRestaurant {
   NationalChef vietnameseChef;
   NationalChef thaiChef;

   InternationalHotel internationalHotel;

   public InternationalRestaurant(
         @Autowired @Qualifier("VietnameseChef") NationalChef vietnameseChef,
         @Autowired @Qualifier("ThaiChef") NationalChef thaiChef
         , @Autowired InternationalHotel internationalHotel
   ) {
      this.vietnameseChef = vietnameseChef;
      this.thaiChef = thaiChef;
      this.internationalHotel = internationalHotel;
   }
}
```

Circular Dependency

```
@Component
public class InternationalHotel {
   InternationalRestaurant internationalRestaurant;

   public InternationalHotel(
         @Autowired InternationalRestaurant internationalRestaurant
   ) {
      this.internationalRestaurant = internationalRestaurant;
   }
}
```

Circular Dependency

- Caused by: org.springframework.beans.factory.**BeanCurrentlyInCreationException**
- Error creating bean with name 'internationalHotel':

  Requested bean is currently in creation: Is there an unresolvable circular reference?

05.02 Circular Dependency Resolution with @Lazy

Workaround - @Lazy

```
@Component
public class InternationalRestaurant {
   NationalChef vietnameseChef;
   NationalChef thaiChef;

   InternationalHotel internationalHotel;

   public InternationalRestaurant(
         @Autowired @Qualifier("VietnameseChef") NationalChef vietnameseChef,
         @Autowired @Qualifier("ThaiChef") NationalChef thaiChef
         , @Autowired @Lazy InternationalHotel internationalHotel
   ) {
      this.vietnameseChef = vietnameseChef;
      this.thaiChef = thaiChef;
      this.internationalHotel = internationalHotel;
   }
}
```

05.03 Circular Dependency Resolution with Class Fields Injection

Workaround - Class Field Injection

```
@Component
public class InternationalRestaurant {
    NationalChef vietnameseChef;
    NationalChef thaiChef;

    @Autowired
    InternationalHotel internationalHotel;

    public InternationalRestaurant(
            @Autowired @Qualifier("VietnameseChef") NationalChef vietnameseChef,
            @Autowired @Qualifier("ThaiChef") NationalChef thaiChef
//              , @Autowired InternationalHotel internationalHotel
    ) {
        System.out.println("Constructor of International-Restaurant Invoke");
        this.vietnameseChef = vietnameseChef;
        this.thaiChef = thaiChef;
//          this.internationalHotel = internationalHotel;
    }
```

Workaround - Class Field Injection

```
@Component
public class InternationalHotel {
    @Autowired
    InternationalRestaurant internationalRestaurant;

    public InternationalHotel(
//            @Autowired InternationalRestaurant internationalRestaurant
    ) {
        System.out.println("Constructor of International-Hotel Invoke");
//        this.internationalRestaurant = internationalRestaurant;
    }

    @Autowired
    public void setInternationalRestaurant(InternationalRestaurant internationalRestaurant) {
        System.out.println("Setter of International-Hotel Invoke");
        this.internationalRestaurant = internationalRestaurant;
    }
}
```

Workaround - Class Field Injection

Console output:

1/ Constructor of International-Hotel Invoke
2/ Constructor of International-Restaurant Invoke
3/ Setter of International-Restaurant Invoke
4/ Setter of International-Hotel Invoke

06. Bean Selection

Duplicate Bean Provider Problem

```
@Configuration
public class ChefConfiguration {

   @Bean
   public NationalChef getThaiChef(
         @Autowired @Qualifier("ThaiReceiptFlavor") ReceiptFlavor receiptFlavor
   ) {
      return new NationalChef(receiptFlavor);
   }

   @Bean
   public NationalChef getVietnameseChef(
         @Autowired @Qualifier("VietnameseReceiptFlavor") ReceiptFlavor receiptFlavor
   ) {
      return new NationalChef(receiptFlavor);
   }
}
```

Duplicate Bean Provider Problem

- Caused by: org.springframework.beans.factory.**NoUniqueBeanDefinitionException**
- No qualifying bean of type ‘xxx.xxx.NationalChef’ available:

  expected single matching bean **but found 2**: getThaiChef, getVietnameseChef

06.01 Bean Selection with @Primary

Bean Selection with @Primary

```
@Configuration
public class ChefConfiguration {

   @Bean
   public NationalChef getThaiChef(
         @Autowired @Qualifier("ThaiReceiptFlavor") ReceiptFlavor receiptFlavor
   ) {
      return new NationalChef(receiptFlavor);
   }

   @Bean
   @Primary // add this annotation to instruct ApplicationContext grabs this
   public NationalChef getVietnameseChef(
         @Autowired @Qualifier("VietnameseReceiptFlavor") ReceiptFlavor receiptFlavor
   ) {
      return new NationalChef(receiptFlavor);
   }
}
```

Bean Selection with @Primary

- Please don’t mark @**Primary twice on the same Bean Type**
- If you @Primary twice on the same Bean Type, what do you Primary?
- Spring also **throws exception** for this accidently configuration

06.02 Bean Selection with @Qualifier

Bean Selection with @Qualifier

- When you really need **two Beans of the same Type**, but have **difference configuration set** for each Bean, let do:
  - At the Bean Provider side: Give each Bean of the same Type the **explicit bean name**
  - At the Bean Consumer side: Select the Bean that you need, by the **explicit bean name** above, using **@Qualifier**

Bean Name at Bean Provider side

```
@Configuration
public class ChefConfiguration {

    @Bean(name = "ThaiChef")
    public NationalChef getThaiChef(
            @Autowired @Qualifier("ThaiReceiptFlavor") ReceiptFlavor receiptFlavor
    ) {
        return new NationalChef(receiptFlavor);
    }

    @Bean(name = "VietnameseChef")
    public NationalChef getVietnameseChef(
            @Autowired @Qualifier("VietnameseReceiptFlavor") ReceiptFlavor receiptFlavor
    ) {
        return new NationalChef(receiptFlavor);
    }
}
```

Bean Name at Bean Provider side

```
@Component("ThaiReceiptFlavor")
public class ThaiReceiptFlavor extends ReceiptFlavor {
    @Override
    public List<String> getReceipt() {
        return Arrays.asList("spicy", "hot", "sour");
    }
}
```

```
@Component("VietnameseReceiptFlavor")
public class VietnameseReceiptFlavor extends ReceiptFlavor {
    @Override
    public List<String> getReceipt() {
        return Arrays.asList("sweet", "warm", "delicious");
    }
}
```

Bean Selection with @Qualifier at Consumer side

```
@Component
public class InternationalRestaurant {
   NationalChef vietnameseChef;
   NationalChef thaiChef;

   private InternationalRestaurant(
         @Autowired @Qualifier("VietnameseChef") NationalChef vietnameseChef,
         @Autowired @Qualifier("ThaiChef") NationalChef thaiChef
   ) {
      this.vietnameseChef = vietnameseChef;
      this.thaiChef = thaiChef;
   }
}
```

06.03 Advanced @Qualifier

07. Extra Injection, Resource, Generics

07.01 Inject Application Configuration with @Value

Inject Application Configuration with @Value

```yaml
kvStyle:
  keyString: "9.5"
  keyInt: 10
  keyFloat: 10.5
  keyBoolean: true
  keyNull: null

arrayStyle:
  - key1: "value 1"
    key2: "value 2"
  - key1: "value 3"
    key2: "value 4"

nestedStyle:
  outer:
    outerKey1: "value 1"
    outerKey2: "value 2"
    inner:
      innerKey1: "value 3"
      innerKey2: "value 4"
```

Inject Application Configuration with @Value

```
@Component
public class YourComponent {
    @Value("${kvStyle.keyString}")
    String keyString;
    @Value("${kvStyle.keyInt}")
    Integer keyInt;
    @Value("${kvStyle.keyFloat}")
    Float keyFloat;
    @Value("${kvStyle.keyBoolean}")
    Boolean keyBoolean;
    @Value("${kvStyle.keyNull}")
    Object keyNull;


    @Value("${kvStyle.keyUndefined:default value}")
    Object keyUndefined;
```

Inject Application Configuration with @Value

```
@PostConstruct
void runShowCase() {
    System.out.println();
    System.out.println(yourComponent.getKeyString());
    System.out.println(yourComponent.getKeyInt());
    System.out.println(yourComponent.getKeyFloat());
    System.out.println(yourComponent.getKeyBoolean());
    System.out.println(yourComponent.getKeyNull());
    System.out.println(yourComponent.getKeyUndefined());
}
```

Inject Application Configuration with @Value

```
Console output:

9.5
10
10.5
true

default value
```

Inject Application Configuration with @Value

- Note that in example above, we use Spring Boot to have the capabilities of reading configuration from YAML file

07.02 Binding
Configuration Properties

Binding Configuration Properties

```yaml
kvStyle:
  keyString: "9.5"
  keyInt: 10
  keyFloat: 10.5
  keyBoolean: true
  keyNull: null

arrayStyle:
  - key1: "value 1"
    key2: "value 2"
  - key1: "value 3"
    key2: "value 4"

nestedStyle:
  outer:
    outerKey1: "value 1"
    outerKey2: "value 2"
    inner:
      innerKey1: "value 3"
      innerKey2: "value 4"
```

Binding Configuration Properties

```
@Configuration
@ConfigurationProperties(prefix = "nested-style")
public class ConfigurationBinding {
   String outerKey1;
   String outerKey2;

   InnerConfig inner;

   public static class InnerConfig {
      String innerKey1;
      String innerKey2;
```

Binding Configuration Properties

```
@PostConstruct
void showCaseConfigurationProperties() {
   System.out.println("================showCaseConfigurationProperties============");
   System.out.println(configurationBinding.getOuterKey1());
   System.out.println(configurationBinding.getOuterKey2());
   System.out.println(configurationBinding.getInner().getInnerKey1());
   System.out.println(configurationBinding.getInner().getInnerKey2());
   System.out.println("============================");
}
```

Binding Configuration Properties

```
Console output:

value 1
value 2
value 3
value 4
```

07.03 Fill up Generic Collection by @Autowire

Generics Collection fill up

```
@Component
public class InternationalRestaurant {
    NationalChef vietnameseChef;
    NationalChef thaiChef;

    @Autowired
    List<NationalChef> chefs;

    @Autowired
    Set<NationalChef> chefSet;
```

Generics Collection fill up

- When you have @**Autowire** on Field with type **Generics Collection**,
- The ApplicationContext will query **all Beans that matching the Type** you specified, then injection into your Generics Collection field
- Provide @**Order** at Bean Provider declaration side to **control the ordering** that you want ApplicationContext fill into your Generics Collection field

Generics Collection fill up, specify fill @Order

```
@Configuration
public class ChefConfiguration {

    @Bean(name = "ThaiChef")
    @Scope("singleton")
    @Order(3)
    public NationalChef getThaiChef(
            @Autowired @Qualifier("ThaiReceiptFlavor") ReceiptFlavor receiptFlavor
    ) {
        return new NationalChef(receiptFlavor);
    }

    @Bean(name = "VietnameseChef")
    @Scope("prototype")
    @Order(2)
    public NationalChef getVietnameseChef(
            @Autowired @Qualifier("VietnameseReceiptFlavor") ReceiptFlavor receiptFlavor
    ) {
        return new NationalChef(receiptFlavor);
    }
}
```

Generics Collection fill up, specify fill @Order

```
public static void main(String... args) {
    ApplicationContext context = new AnnotationConfigApplicationContext(MasterConfigurationFile.class);

    InternationalRestaurant internationalRestaurant = context.getBean(InternationalRestaurant.class);

    System.out.println("Check @Order in List:");
    for(NationalChef c : internationalRestaurant.getChefs()) {
        System.out.println(c.cook());
    }

    System.out.println("Check @Order in Set:");
    for(NationalChef c : internationalRestaurant.getChefSet()) {
        System.out.println(c.cook());
    }
}
```

Generics Collection fill up, specify fill @Order

Console output:

Check @Order in List
[sweet, warm, delicious]
[spicy, hot, sour]
Check @Order in Set
[spicy, hot, sour]
[sweet, warm, delicious]

Generics Collection fill up, specify fill @Order

- Note that Generics Set maintains its own Ordering index, based on Hash Table, this will override your specified order settings

08. Bean Scope

Bean Scope

- **AppicationContext** manage Beans with 2 primary scopes:
  - Singleton Bean scope
  - Prototype Bean scope
- **WebApplicationContext** currently 4 more scopes relate to web behavior:
  - Request Bean scope
  - Session Bean scope
  - Application Bean scope (per Servlet Application context)
  - WebSocket Bean scope
- Additional, Spring offer extension capabilities when you really need customize your scope base on your need
  - Custom Bean scope

08.01 Scope Singleton

Singleton Bean Scope

- Don’t be confused with Singleton Design Pattern (in which, Singleton is Global, and ensured by Private Static Instance within Class)
- **Singleton Bean Scope** ensure there should **only one instance** of **specific Bean** in **single ApplicationContext**
- That mean, when several consumer request for the same Bean from ApplicationContext, there’re always the same instance of that Bean returned
- **This is the default behavior of Spring Bean Scope**

08.02 Scope Prototype

Prototype Bean Scope

- If you define a Bean as **Prototype Scope**, whenever consumer request this Bean from **ApplicationContext**, there will be **freshly**, **new created instance** returned
- To instruct **ApplicationContext** that it should treat your **Bean Provider** as **Prototype Generator,** let mark **@Scope(“prototype”)** at the Bean Provider declaration side
- Remember: **Scopes are defined at Bean Provider Point, not at Bean Consumer Point**

Prototype Bean Scope

```
@Bean(name = "VietnameseChef")
@Scope("prototype")
public NationalChef getVietnameseChef(
      @Autowired @Qualifier("VietnameseReceiptFlavor") ReceiptFlavor receiptFlavor
) {
   return new NationalChef(receiptFlavor);
}
```

Prototype Bean Scope

```
@Configuration
public class ChefConfiguration {

   @Bean(name = "ThaiChef")
   @Scope("singleton")
   public NationalChef getThaiChef(
         @Autowired @Qualifier("ThaiReceiptFlavor") ReceiptFlavor receiptFlavor
   ) {
      return new NationalChef(receiptFlavor);
   }

   @Bean(name = "VietnameseChef")
   @Scope("prototype")
   public NationalChef getVietnameseChef(
         @Autowired @Qualifier("VietnameseReceiptFlavor") ReceiptFlavor receiptFlavor
   ) {
      return new NationalChef(receiptFlavor);
   }
}
```

Prototype Bean Scope

```
public class Main {
    public static void main(String... args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(MasterConfigurationFile.class);

        NationalChef thaiChef1 = context.getBean("ThaiChef", NationalChef.class);
        NationalChef thaiChef2 = context.getBean("ThaiChef", NationalChef.class);
        NationalChef vietnameseChef1 = context.getBean("VietnameseChef", NationalChef.class);
        NationalChef vietnameseChef2 = context.getBean("VietnameseChef", NationalChef.class);

        System.out.println("does Singleton equal to Singleton: " + thaiChef1.equals(thaiChef2));
        System.out.println("does Prototype equal to Prototype: " + vietnameseChef1.equals(vietnameseChef2));
    }
}
```

Prototype Bean Scope

```
Console output:

does Singleton equal to Singleton: true
does Prototype equal to Prototype: false
```

08.03 Inject Scope Prototype Bean into Scope Singleton Bean

Inject Prototype Bean into Singleton Bean

Inject Prototype Bean into Singleton Bean

- When your ApplicationContext contains Prototype Bean Scope. Sometime, there will be unexpected behavior of Prototype Bean if you don’t care about collaboration between Bean
- What is that?
- Singleton Bean Scope **only constructed one time**, remember?
- So when you have Prototype Bean **inside Singleton Bean**, these Prototype Bean, as a result, will be also the **same Instance every time you access it** (this just effects on the Prototype Bean inside Singleton Bean)
- To address this problem, Spring introduce **Method Lookup Injection**

Inject Prototype Bean into Singleton Bean

```
@Lookup("VietnameseChef")
public NationalChef lookupVietnameseChef() {
   return null;
}
```

Inject Prototype Bean into Singleton Bean

```
public static void main(String... args) {
    ApplicationContext context = new AnnotationConfigApplicationContext(MasterConfigurationFile.class);

    NationalChef thaiChef1 = context.getBean("ThaiChef", NationalChef.class);
    NationalChef thaiChef2 = context.getBean("ThaiChef", NationalChef.class);
    NationalChef vietnameseChef1 = context.getBean("VietnameseChef", NationalChef.class);
    NationalChef vietnameseChef2 = context.getBean("VietnameseChef", NationalChef.class);

    System.out.println("does Singleton equal to Singleton: " + thaiChef1.equals(thaiChef2));
    System.out.println("does Prototype equal to Prototype: " + vietnameseChef1.equals(vietnameseChef2));

    InternationalRestaurant internationalRestaurant1 = context.getBean(InternationalRestaurant.class);
    InternationalRestaurant internationalRestaurant2 = context.getBean(InternationalRestaurant.class);
    System.out.println("does Prototype in Singleton equal to Prototype in Singleton: "
            + internationalRestaurant1.getVietnameseChef()
            .equals(internationalRestaurant2.getVietnameseChef()));

    InternationalRestaurant internationalRestaurant3 = context.getBean(InternationalRestaurant.class);
    InternationalRestaurant internationalRestaurant4 = context.getBean(InternationalRestaurant.class);
    System.out.println("does Prototype Lookup Method equal to Prototype Lookup Method: "
            + internationalRestaurant3.lookupVietnameseChef()
            .equals(internationalRestaurant4.lookupVietnameseChef()));
}
```

Inject Prototype Bean into Singleton Bean

```
Console output:

does Singleton equal to Singleton: true
does Prototype equal to Prototype: false
does Prototype in Singleton equal to Prototype in Singleton: true
does Prototype Lookup Method equal to Prototype Lookup Method: false
```

09. Bean Life-Cycle Hook

09.01 Hook into Bean Life-Cycle with @PostConstruct

09.02 Hook into Bean Life-Cycle with @PreDestroy

10. Spring Profile Selection

Thank you

Nash Tech.