*@Import*

*"How do we get object of a class from a method in XML?"*

Here’s the key:

* In **Java config**, the method itself is executed by Spring, and the return object becomes the bean.
* In **XML**, there are **no methods**. You just tell Spring:  
  “This is the class. Create an object using its constructor.”

So:

* Java → method returns the object
* XML → you declare the class and arguments; Spring calls the constructor

**✅ Example with Custom Class (for better understanding)**

**Java Config:**

@Configuration

public class AppConfig {

@Bean

public MyService myService() {

return new MyService("Hello from XML");

}

}

**XML Equivalent:**

<bean id="myService" class="com.example.MyService">

<constructor-arg value="Hello from XML"/>

</bean>

Eg: ImportAnnotation

### ✅ Best Practice in Layered Architecture (Spring XML)

In a **layered approach**, keeping all the beans for Controller, Service, and DAO in **one single XML file** is **not recommended**, especially for large projects.  
This is because it **reduces readability** and **increases maintenance complexity**.

### 💡 Solution:

To improve readability and structure:

* **Separate XML files** should be created for each layer:
  + controller-config.xml
  + service-config.xml
  + dao-config.xml
* Then, use the <import resource="..."/> tag in a **main configuration file** (e.g., applicationContext.xml) to import all these.

### ✅ Example:

**applicationContext.xml**:

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

<import resource="controller-config.xml"/>

<import resource="service-config.xml"/>

<import resource="dao-config.xml"/>

</beans>

In Spring Framework, p: and c: namespaces are used in XML configuration to simplify bean property and constructor argument injection. They are shorthand notations that make XML less verbose compared to using <property> and <constructor-arg> tags.

### ✅ 1. p: Namespace – Setter Injection (Property Injection)

Instead of writing:

<bean id="student" class="com.example.Student">

<property name="name" value="Anil" />

<property name="age" value="25" />

</bean>

You can write:

<bean id="student" class="com.example.Student" p:name="Anil" p:age="25" />

**👉 Requires:**  
Add the p namespace in the <beans> tag:

xmlns:p="http://www.springframework.org/schema/p"

### ✅ 2. c: Namespace – Constructor Injection

Instead of writing:

<bean id="student" class="com.example.Student">

<constructor-arg value="Anil" />

<constructor-arg value="25" />

</bean>

You can write:

<bean id="student" class="com.example.Student" c:\_0="Anil" c:\_1="25" />

Or if the constructor has parameter names and Spring 4.3+ or parameter name discovery is enabled:

<bean id="student" class="com.example.Student" c:name="Anil" c:age="25" />

**👉 Requires:**  
Add the c namespace in the <beans> tag:

xmlns:c="http://www.springframework.org/schema/c"

## 🔴 Limitations of p: and c: Namespace in Spring XML

1. **No Support for Collection Injection**
   * p: and c: namespaces cannot inject collections like List, Set, Map, or Properties.
   * For collections, you still need verbose <property> tags with <list>, <map>, etc.
2. **Constructor Injection Is Still Ambiguous**
   * c: namespace doesn’t resolve ambiguity when constructors are overloaded.
   * It lacks precise control using type, index, or name attributes (which <constructor-arg> supports).
   * Order-sensitive and error-prone: c:\_0, c:\_1, etc. must match exact constructor order.
3. **Late Arrival & Low Adoption**
   * p: and c: namespaces were introduced **after annotations (@Autowired, @Component, etc.) became standard**.
   * Most modern Spring applications use **Java Config and annotations**, making p and c less relevant in the industry.

### ✅ ****Pure Java Configuration (No XML)****

**100% code-driven Spring App development**

#### 🔹 Advantages

1. **Avoids XML config** in most cases.
2. **Improves readability** – Java annotations are easier to read than XML.
3. **Easier debugging** – stack traces point to Java classes/methods.
4. **Builds foundation** for learning Spring Boot (which heavily uses this approach).

### 📌 Thumb Rules

1. **User-defined beans (custom classes)**:
   * Annotate with @Component, @Service, etc.
   * Use @ComponentScan in the @Configuration class to detect these beans.
   * ✅ Any class annotated with @Configuration becomes a configuration class.
2. **Pre-defined or third-party beans**:
   * Use @Bean methods inside the @Configuration class to define them.
3. **IOC Container creation**:
   * Use AnnotationConfigApplicationContext with your @Configuration class as input.

### 🔍 Notes

* @Configuration is internally a Spring bean.
* It implicitly contains @Component, which makes it component-scannable.

Eg: JavaCodeConfig

Absolutely. Let’s go **step by step** to explain how the following Spring bean:

@Bean

public LocalDateTime dateTime() {

return LocalDateTime.now();

}

gets **injected into a target class** like this one:

@Component("wmg")

public class WishMessageGenerator {

@Autowired

private LocalDateTime date;

}

## 🔁 FULL FLOW EXPLANATION OF @Autowired INJECTION

### 🔧 Step 1: Bean Registration

In the configuration class:

@Configuration

public class AppConfig {

@Bean

public LocalDateTime dateTime() {

return LocalDateTime.now();

}

}

Spring sees the @Bean method and **registers the return value** of LocalDateTime.now() into the container with:

* **Bean ID** = "dateTime" (method name)
* **Bean Type** = LocalDateTime

This is like putting an object of LocalDateTime into Spring's internal HashMap:

map.put("dateTime", LocalDateTime.now());

### 🧠 Step 2: Target Bean Creation

When Spring scans and finds this:

@Component("wmg")

public class WishMessageGenerator {

@Autowired

private LocalDateTime date;

}

Spring:

1. Detects the class WishMessageGenerator
2. Sees @Autowired on the LocalDateTime date field
3. Uses **reflection** to inject a LocalDateTime bean into it

### 🔍 Step 3: How Does @Autowired Work?

Spring internally performs the following logic:

1. **Searches the container** for a bean of type LocalDateTime
2. Finds that the @Bean method registered one (from AppConfig)
3. Injects that instance into the field using **reflection** (even if private)

It’s equivalent to:

wishMessageGenerator.setDate(context.getBean(LocalDateTime.class));

Even if setDate() doesn’t exist, Spring uses reflection to inject it.

### ⚠️ If More Than One Bean of Same Type?

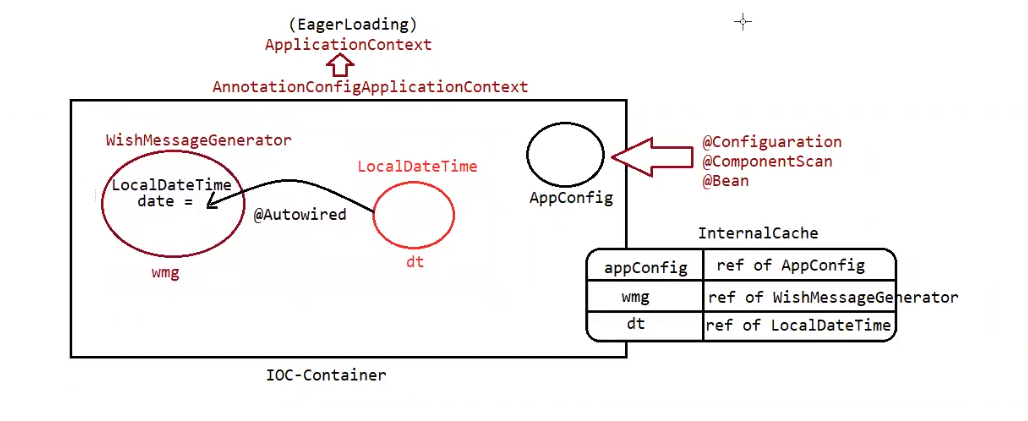
If there are multiple LocalDateTime beans, Spring will throw an error:

NoUniqueBeanDefinitionException

You can fix that with:

* @Qualifier("dateTime")
* or name matching if the field name and bean ID match

In our case, field name = date, bean ID = dateTime → Not the same.  
But since there's **only one** bean of type LocalDateTime, Spring injects it without confusion.



### 🟢 1. You Start Spring

You wrote:

ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);

#### What happens inside:

* Spring starts.
* It reads your AppConfig class.
* It checks if the class is marked with @Configuration. (Yes ✅)
* It looks inside AppConfig for:
  + @ComponentScan
  + @Bean methods

### 🟢 2. Spring Scans Your Package

From:

@ComponentScan(basePackages = "in.orcas.bean")

#### What Spring does:

* It goes into the folder/package in.orcas.bean.
* It looks for any class marked with @Component.

It finds:

@Component("wmg")

public class WishMessageGenerator { ... }

So it adds this class to its list of beans.

### 🟢 3. Spring Registers Beans

Now Spring has 2 beans:

1. From @Bean dateTime()
2. From @Component WishMessageGenerator

But it hasn’t created objects yet — it just **remembers** how to create them.

### 🟢 4. Spring Creates the Beans

Now Spring starts creating real objects:

#### 🔹 dateTime bean

* It sees your method:

@Bean

public LocalDateTime dateTime() {

return LocalDateTime.now();

}

* Spring **calls** this method.
* It gets one LocalDateTime object.
* It saves this in memory (with the name "dateTime").

#### 🔹 wmg bean (WishMessageGenerator)

* Spring **calls the constructor** of the class and makes a new object.

Now both beans are created and saved.

### 🟢 5. Spring Checks for @Autowired

Inside the WishMessageGenerator class, Spring sees:

@Autowired

private LocalDateTime date;

Spring says:

* “This class needs a LocalDateTime.”
* “Do I have one?” → Yes, it has the "dateTime" bean.
* So it **injects** that object into the date field using reflection (it sets the private field directly).

### 🟢 6. Now Your Bean is Ready

Spring finishes its work. Now:

WishMessageGenerator generator = ctx.getBean("wmg", WishMessageGenerator.class);

* You ask Spring for the "wmg" bean.
* Spring gives you the WishMessageGenerator object it already created and injected.

### 🟢 7. You Call the Method

String message = generator.greetMessage("pavan");

* Inside the method:

return "Hello " + name + "! Generated at: " + date;

* date already has the value (injected earlier).
* You get the final message:

Hello pavan! Generated at: 2025-06-09T18:40:00.123

SpringBoot:

===========

 **Your main() kicks things off**  
You call

SpringApplication.run(MyApp.class, args);

passing in:

* **MyApp.class** → tells Spring “this is my primary configuration class.”
* **args** → any command-line arguments (like --server.port=8085).

 **What @SpringBootApplication actually does**

@SpringBootApplication

public class SpringBootDependencyInjectionApplication { … }

is shorthand for:

### 1. @Configuration

* **What it is:**  
  A specialized form of @Component indicating that this class contains one or more @Bean methods.
* **What it does under the hood:**

 **Registers the class…**  
Spring puts this class on its “watch list” because it contains methods that show how to create app objects (beans).

 **Processes any @Bean-annotated methods…**  
Spring runs every method marked with @Bean, takes the object that method returns, and keeps that object ready so other parts of the app can use it.

### 2. @ComponentScan

* **What it is:**  
  Tells Spring where to look on the classpath for components—classes annotated with @Component, @Service, @Repository, @Controller, etc.
* **Defaults & customization:**
*  **Default behavior**  
  Spring automatically checks the folder where this class sits—and every folder inside it—to find other classes marked as Spring parts.
*  **Custom packages**  
  If you want Spring to look in extra folders, add something like  
  @ComponentScan(basePackages = {"com.myapp.core", "com.myapp.web"}).  
  This tells Spring, “Also search these places for parts.”
*  **Filters**  
  You can give simple rules to skip or include certain classes (for example, “ignore anything ending in Test” or “only pick classes with @Service”).
*  **Benefit**  
  Spring finds and connects your parts on its own, so you don’t have to write XML files or register each class by hand.
* **@EnableAutoConfiguration** → set up common Spring features automatically.
* **What it is**  
  Think of @EnableAutoConfiguration as Spring Boot’s **autopilot switch**.  
  When it is on, Spring looks at the libraries you added and says, “Oh, I see you brought web stuff or database stuff—let me turn that on for you.”
* **How it works (behind the scenes)**
  1. **Imports a selector**  
     Spring quietly adds a helper class (AutoConfigurationImportSelector) that will collect setup suggestions from every library on the classpath.
  2. **Reads spring.factories**  
     Each library JAR ships with a tiny file that lists its own setup suggestions. Spring reads all those files to build one big to-do list.
  3. **Runs conditional checks**  
     Before applying a suggestion, Spring asks simple yes/no questions:
     + *Is the needed class present?* (e.g., DataSource?)
     + *Did the developer already create one?*  
       If the answer fits, Spring applies the setup; otherwise, it skips it.
* **Examples of what you get “for free”**
  + **Web app JARs present?** → Spring auto-starts an embedded Tomcat/Jetty server and wires Spring MVC.
  + **JDBC or JPA JARs present?** → It sets up a DataSource, connection pool, and EntityManagerFactory.
  + **Add messaging, caching, security, or actuator JARs?** → Matching starters (RabbitMQ, Redis, Spring Security, Actuator endpoints) switch on automatically.

 **Inside SpringApplication.run(...)**  
Spring Boot simply:

* **Loads** your settings (properties + args).
* **Creates** all your beans.
* **Wires** their dependencies (@Autowired, constructors).
* **Returns** the ready-to-go application context.

 **Role of the args you pass in**

* **Override defaults** (e.g. change port with --server.port=9090).
* **Get built-in help** with --help.
* **Pass simple flags** into your app if you need them.

 **Why this matters**

* **No XML** — everything’s by annotations and conventions.
* **Predictable startup** — every app follows the same steps.
* **Still customizable** — you can add or override beans whenever you like.

Eg: SpringBootDependencyInjection

## 1. main() kicks things off

public static void main(String[] args) {

System.out.println("🔸 Starting Spring Boot application...");

ConfigurableApplicationContext ctx =

SpringApplication.run(SpringBootDependencyInjectionApplication.class, args);

// …

}

1. **Prints “🔸 Starting Spring Boot application…”**
2. Calls

SpringApplication.run(SpringBootDependencyInjectionApplication.class, args);

* + **SpringBootDependencyInjectionApplication.class** tells Spring “this is my app’s main configuration.”
  + **args** carries any command-line options (e.g. --server.port=9090) to Spring’s environment.

## 2. Spring Boot bootstraps your app

Inside SpringApplication.run(...), Spring Boot:

1. **Reads its annotations** on your main class:
   * @Configuration → “look here for @Bean methods.”
   * @ComponentScan → “search in.orcas and sub-packages for @Service, @Component, etc.”
   * @EnableAutoConfiguration → “set up common Spring features automatically.”
2. **Prepares its environment** by loading:
   * application.properties / application.yml
   * OS environment variables
   * Your args  
     All of these become available as Spring **properties**.
3. **Creates an ApplicationContext** (the Spring container that holds all beans).

## 3. Beans are created & dependencies injected

As the context starts up, Spring goes through two main bean-creation phases:

### A. Your @Bean methods

In your main class:

@Bean

public LocalDateTime date() {

System.out.println("🔹 Creating LocalDateTime bean...");

return LocalDateTime.now();

}

* Spring calls this method first.
* You see **“🔹 Creating LocalDateTime bean…”** on the console.
* It stores that LocalDateTime object in its bean registry.

### B. Your @Service bean

In in.orcas.service.WishMessageGenerator:

@Service("wmg")

public class WishMessageGenerator {

static {

System.out.println("1️⃣ WishMessageGenerator.class file is loading...");

}

@Autowired

private LocalDateTime date;

public WishMessageGenerator() {

System.out.println("2️⃣ WishMessageGenerator :: Zero-param constructor...");

}

// …

}

1. **Class loading**
   * The JVM loads WishMessageGenerator → static block runs → you see

1️⃣ WishMessageGenerator.class file is loading...

1. **Instantiate bean**
   * Spring calls its zero-arg constructor → prints

2️⃣ WishMessageGenerator :: Zero-param constructor...

1. **Inject dependencies**
   * Spring notices @Autowired private LocalDateTime date;
   * It takes the LocalDateTime bean it created earlier and sets that field.

## 4. Back in main(): getBean & business logic

WishMessageGenerator wmg =

ctx.getBean("wmg", WishMessageGenerator.class);

String msg = wmg.generateWishMessage("Sai");

System.out.println("🔹 Wish Message: " + msg);

1. **ctx.getBean("wmg", …)**
   * Returns your fully-ready WishMessageGenerator instance.
2. **generateWishMessage("Sai")**
   * Prints

3️⃣ WishMessageGenerator.generateWishMessage() called...

* + Reads date.getHour(), picks a greeting, and returns it.

1. **Prints the greeting**

🔹 Wish Message: Good Afternoon, Sai!

## 5. Shutdown: ctx.close()

ctx.close();

System.out.println("🔸 Application context closed.");

* Spring gracefully destroys all beans and releases resources.
* You see **“🔸 Application context closed.”**

Eg: SpringBootDependencyInjectionPrototypeScope

 **"prototype"** is like writing down a password by hand—quick, but a typo could lock you out.

 **SCOPE\_PROTOTYPE** is like picking the password from a dropdown menu—takes a tiny bit longer, but you’re guaranteed it’s correct.