Spring Core – Introduction

Presented by

Sagar
Java Consultant

Spring Framework

A software framework is a re-usable design for a software system.

Application Framework : A framework that is fit for any type of application

Ex : Stand Alone, Web, Enterprise, Persistence applications etc.

Spring is an Application Framework Developed by

Spring Framework …

•A modular framework, i.e., spring framework is a layered architecture

•Allows selection of components based on the need in a real time application

Ex:
- Modelled component – POJO, JDBC,
- MVC components – Spring MVC
- ORM Components – Hibernate integration

- Spring is non-invasive – Spring it doesn't force developers to inherit classes or implement interfaces during development.

- Spring provides predefined templates for JDBC, Hibernate, etc.

•Non - Invasive – Does not force to implement the frameworks classes

- Ex : Struts is invasive –Need to use the built-in classes and overriding their methods (Action class methods) - Use everything (MVC) of Struts

Light Weight Framework

•**Light Weight Framework :**

- Heavy weight frameworks **depend on the classes to hook components into them**

  Ex : Struts / EJB – Code is dependent on built in classes Action, (Forces to use them)

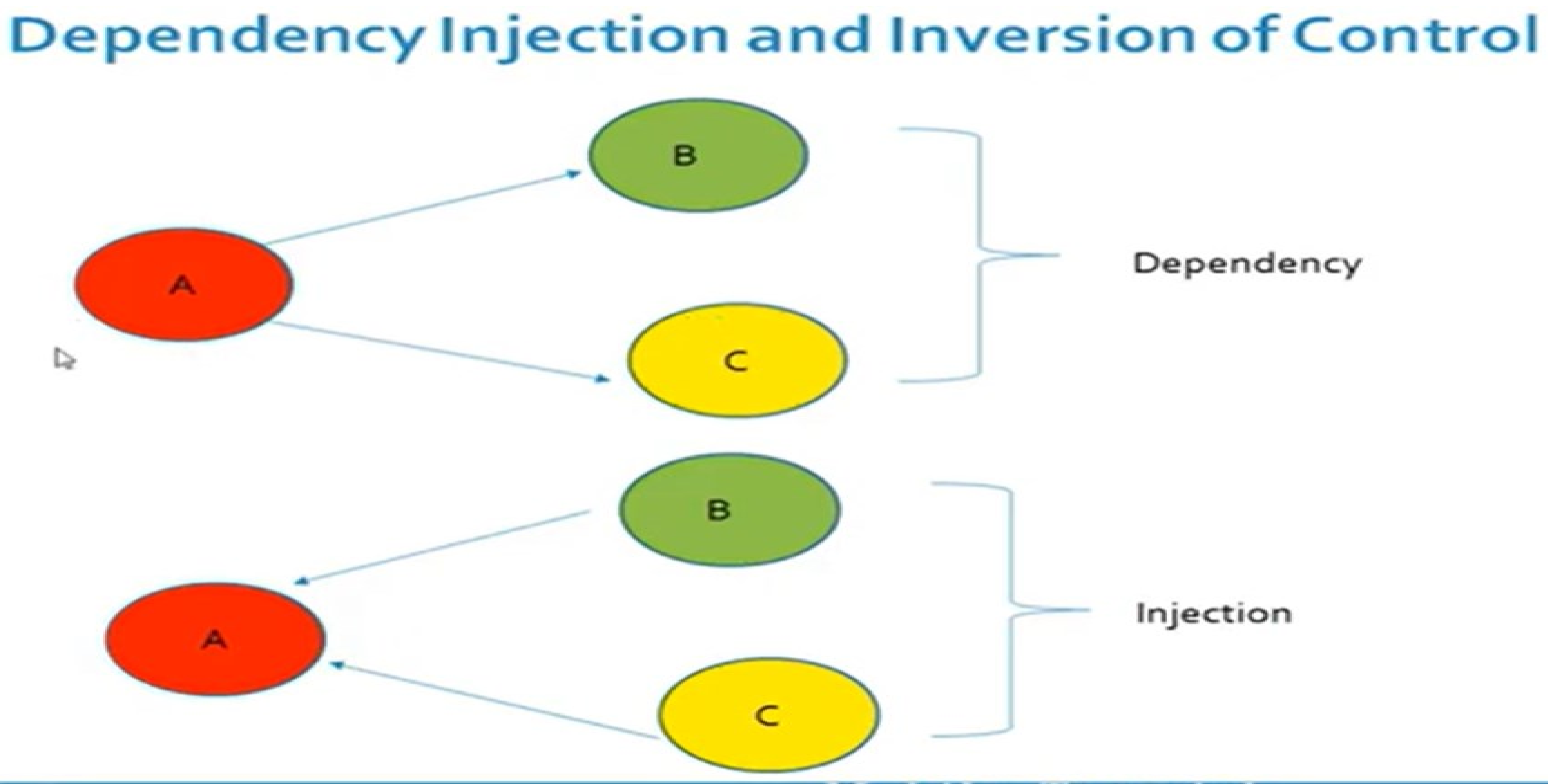
- Light weight **-** Lightweight framework **does not depend on classes to hook components into them!**

  - Uses user defined classes and methods.

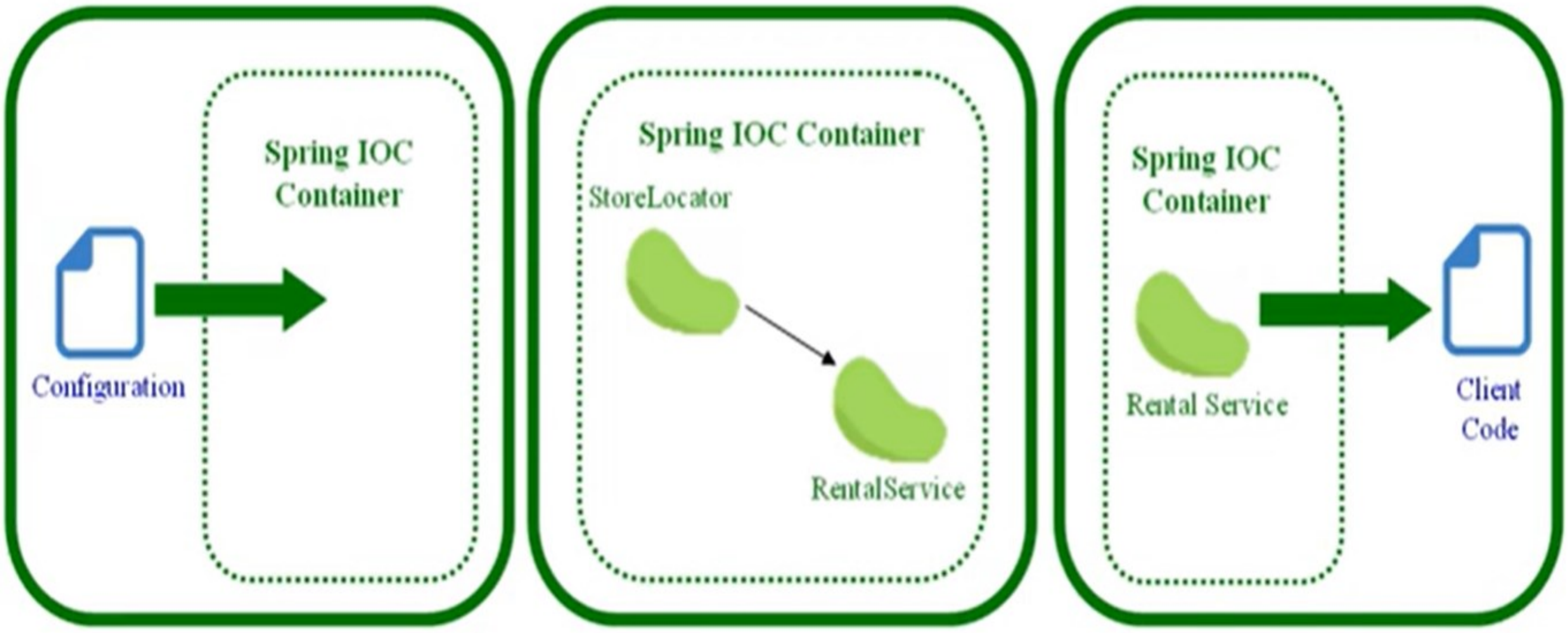
Core Principles of Spring

- **Inversion of Control (IoC):**
  - Objects provide their dependencies to the framework, which injects them as needed.
- **Dependency Injection (DI):**
  - The process of providing dependencies to objects without requiring them to create or locate their dependencies.
- **Aspect-Oriented Programming (AOP):**
  - Separates application concerns – Logger info, Exception logs etc (cross-cutting concerns) from the core business logic. A background process api to know the status of the application.
  - Does'nt clutter with main business flow.

DI and IoC


Dependency Injection and Inversion of Control
B
A
C
Dependency
B
A
C
Injection

Spring DI and IoC Container …



*The act of Dependency Injection is known as **'wiring'.***

Benefits of Spring

Loose coupling:

- Promotes modularity and reusability.

Testability:

- Easier to write unit tests due to loose coupling and DI.

Productivity:

- Reduces boilerplate code and simplifies development.

Community and ecosystem:

- Large community and extensive ecosystem of libraries and tools.

Benefits of Spring …

- **An alternative / replacement for the Enterprise Java Bean (EJB) model**

- **Flexible** - Programmers decide how to program.
-
    - Example : User will define methods in a **single conroller** instead of having lifecycle methods (like servlets)

- **Solutions to typical coding busywork** – Need not not use everything of Spring

Benefits of DI

- **Flexibility**:
    - Easier to change dependencies and configurations – avoid **new** keyword.
- **Testability:**
    - Simplifies unit testing by allowing easy mocking of dependencies.
- **Reusability:**
    - Promotes the creation of reusable components.

Bean Factory

BeanFactory:

•**Basic Functionality:** Provides the fundamental infrastructure for creating and managing beans.

•**Lazy Initialization:** Beans are instantiated only when they are explicitly requested, promoting efficient memory usage.

•**Configuration:** Can be configured using XML configuration files.

•**Dependency Injection:** Supports dependency injection.

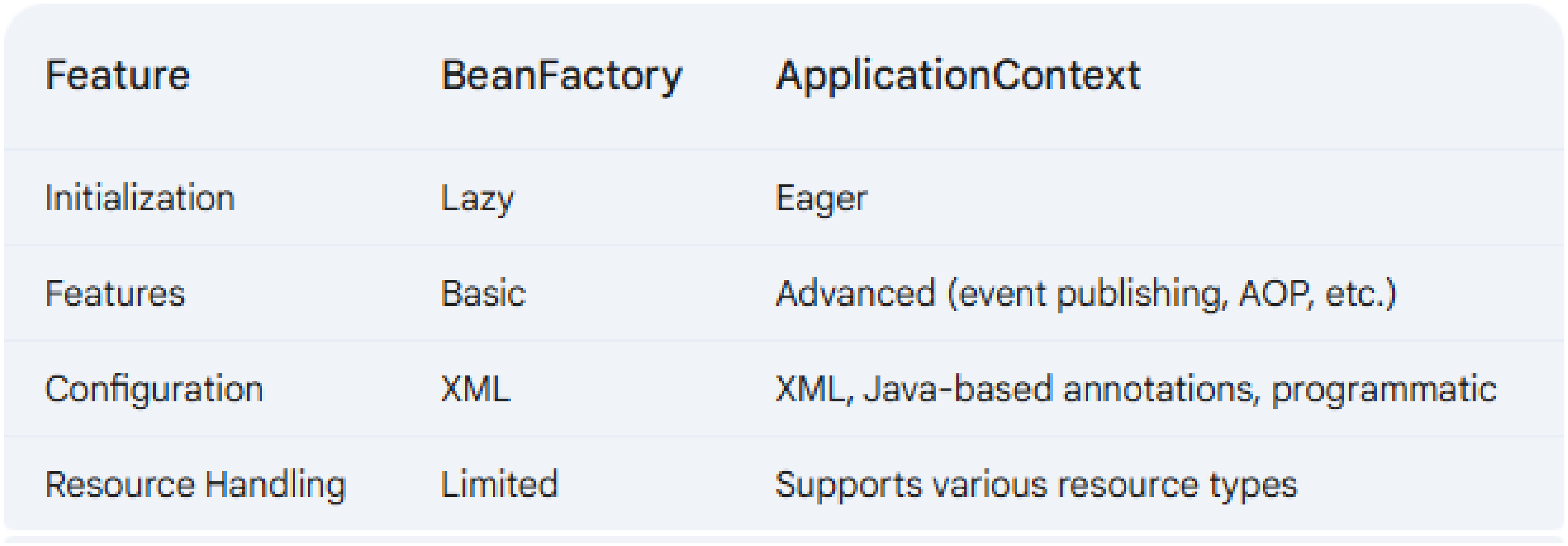
Application Context

ApplicationContext:

•**Extends BeanFactory:** Inherits all the capabilities of BeanFactory and adds additional features.

•**Eager Initialization:** By default, all beans are instantiated upon application startup, ensuring they are ready for use.

Bean Context vs Appication Context



Feature	BeanFactory	ApplicationContext
Initialization	Lazy	Eager
Features	Basic	Advanced (event publishing, AOP, etc.)
Configuration	XML	XML, Java-based annotations, programmatic
Resource Handling	Limited	Supports various resource types

Spring Core Container

Spring's Container uses IoC to manage components of the application.

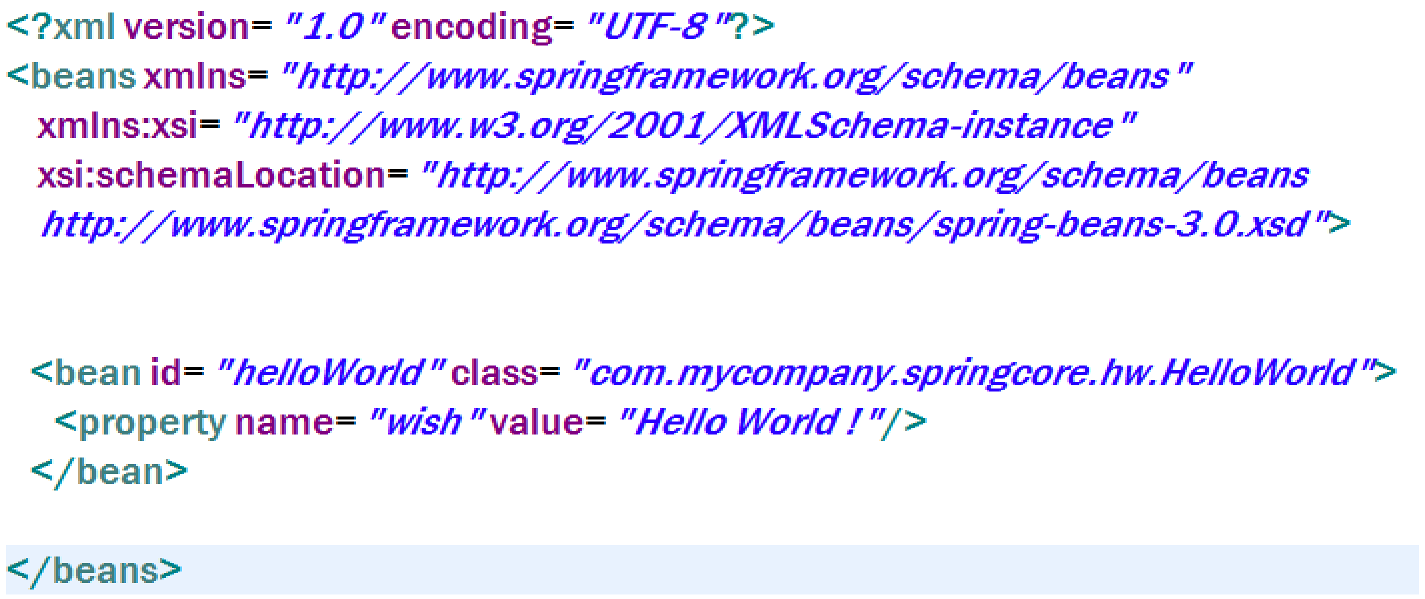
- **Application context** (org.springframework.context.ApplicationContext) provides application framework services

Spring way of DI

- Java classes should be as independent as possible from each other
- Spring Framework injects these dependencies via the container.
- Piecing together all beans in the Spring Container is called **wiring**.
- Wiring can be done through xml or thru @nnotations.

Bean Configuration

Beans are listed in the configuration file



```
<?xml version= "1.0" encoding= "UTF-8"?>
<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-3.0.xsd">


 <bean id= "helloWorld" class= "com.mycompany.springcore.hw.HelloWorld">
   <property name= "wish" value= "Hello World !"/>
 </bean>

</beans>
```

**HelloWorld application*

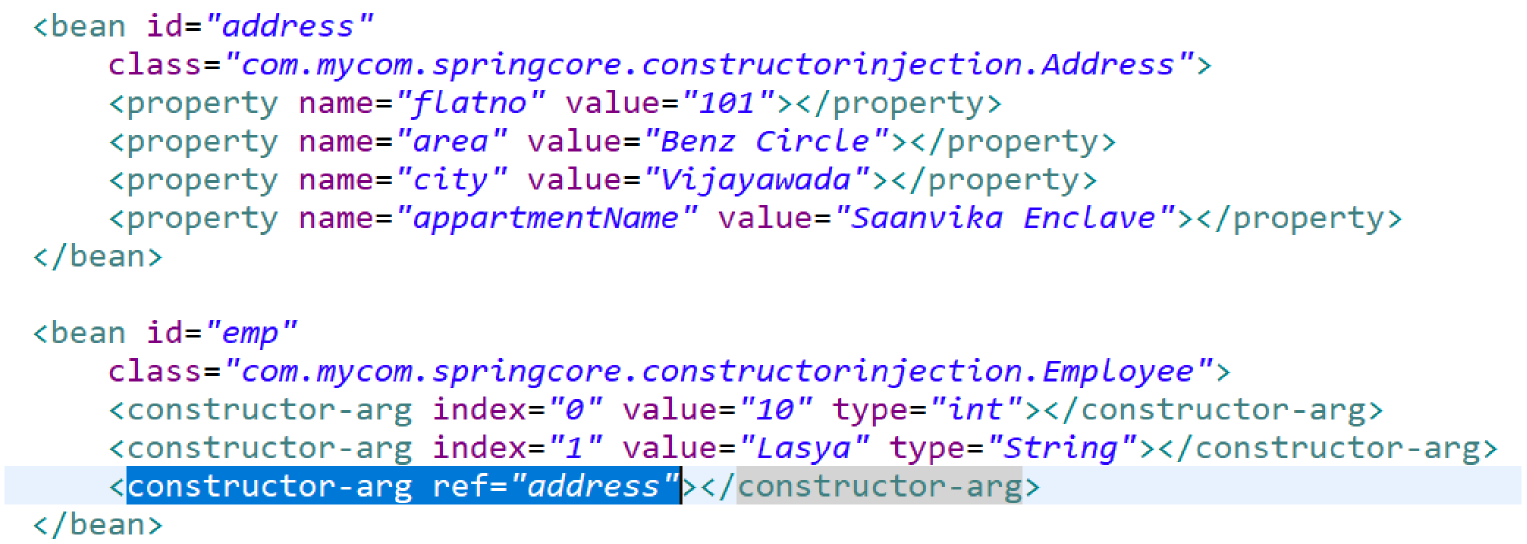
Spring Constructor Injection

Scenario : **Employee** **HAS-A** **Address**.

We can inject the dependency(Address) by constructor.

When using constructor injection, **setters are not used !**

The **<constructor-arg>** sub element of **<bean>** tag is used for constructor injection.



```xml
<bean id="address"
    class="com.mycom.springcore.constructorinjection.Address">
    <property name="flatno" value="101"></property>
    <property name="area" value="Benz Circle"></property>
    <property name="city" value="Vijayawada"></property>
    <property name="appartmentName" value="Saanvika Enclave"></property>
</bean>

<bean id="emp"
    class="com.mycom.springcore.constructorinjection.Employee">
    <constructor-arg index="0" value="10" type="int"></constructor-arg>
    <constructor-arg index="1" value="Lasya" type="String"></constructor-arg>
    <constructor-arg ref="address"></constructor-arg>
</bean>
```

**Demo

Setter Injection

Scenario : **Employee HAS-A Address**.

The **Address** class object will be termed as the **Dependent** object.

The **Employee** class object will be termed as the **Dependency**

```
 <bean id="address" class="com.mycom.springcore.setterinjection.beans.Address">
     <property name="flatno" value="1001"></property>
     <property name="appartmentName" value="Star Castle" />
     <property name="area" value="Wakad"></property>
     <property name="city" value="Pune"></property>
</bean>

<bean id="emp" class="com.mycom.springcore.setterinjection.beans.Employee">
   <property name="id" value="1020"/>
   <property name="name" value="Nesha"/>
   <property name="address" ref="address"/>
</bean>
```

****Demo**

Annotations

Annotations are instructions to the compiler

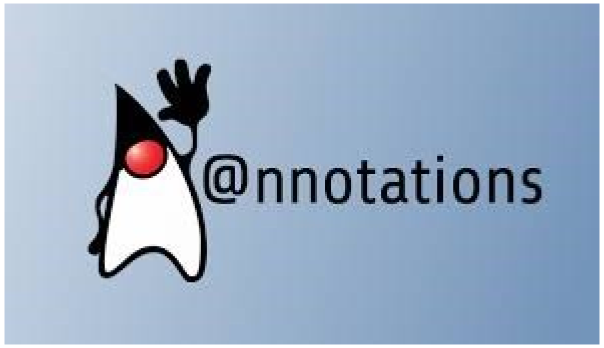
Meta data / data about code

@Override

Checks that the method is an override.

Ex:

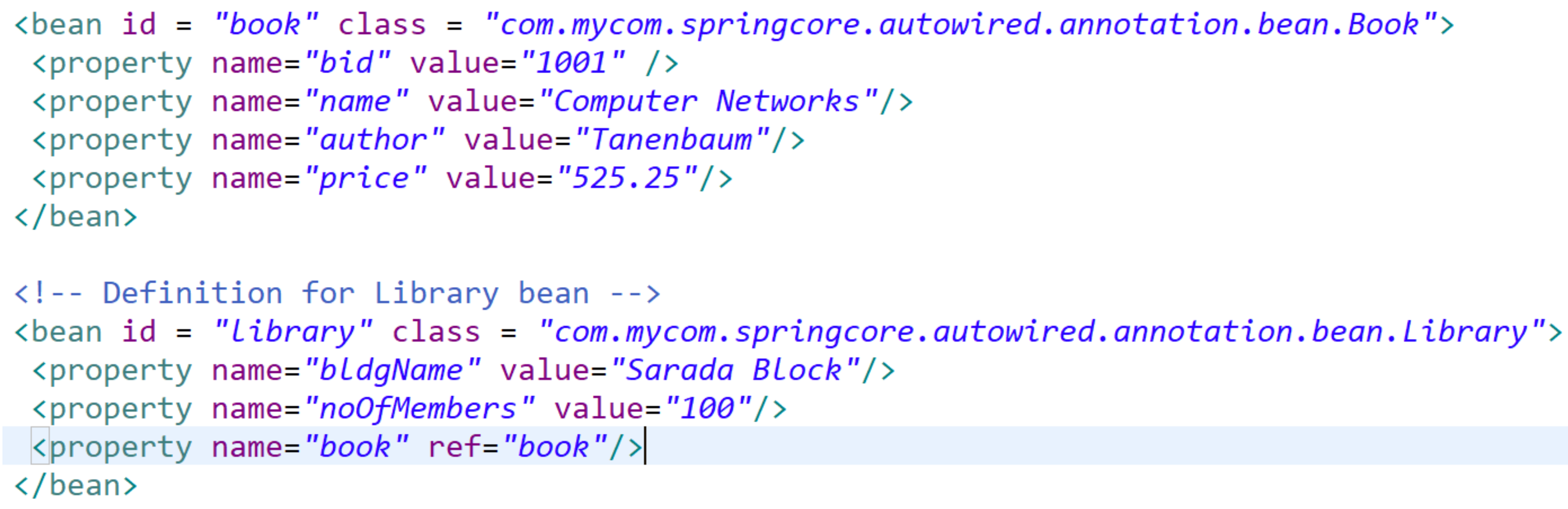
```
@Override
public String toString() {
		return “Welcome !";
}
```



Without @Autowired

@Autowired annotation is used to inject a bean automatically with out using **ref** attribute

Normal configuration : without @Autowired



```
<bean id = "book" class = "com.mycom.springcore.autowired.annotation.bean.Book">
 <property name="bid" value="1001" />
 <property name="name" value="Computer Networks"/>
 <property name="author" value="Tanenbaum"/>
 <property name="price" value="525.25"/>
</bean>

<!-- Definition for Library bean -->
<bean id = "library" class = "com.mycom.springcore.autowired.annotation.bean.Library">
 <property name="bldgName" value="Sarada Block"/>
 <property name="noOfMembers" value="100"/>
 <property name="book" ref="book"/>
</bean>
```

With @Autowired

In xml file

```xml
<context:annotation-config/>

<bean id = "book" class = "com.mycom.springcore.autowired.annotation.bean.Book">
 <property name="bid" value="1001" />
 <property name="name" value="Computer Networks"/>
 <property name="author" value="Tanenbaum"/>
 <property name="price" value="525.25"/>
</bean>

<!-- Definition for Library bean -->
<bean id = "library" class = "com.mycom.springcore.autowired.annotation.bean.Library">
 <property name="bldgName" value="Sarada Block"/>
 <property name="noOfMembers" value="100"/>
</bean>
```

In Java Bean

```java
public class Library {
String bldgName;
int noOfMembers;
@Autowired
Book book;
```

**** Demo**

@Component

@Component annotation marks a java class as a bean

So spring container can pick it up and push it into the application

Called as component-scanning .

No need to configure it!

Ex :

```
@Component
public class Employee {
}
```

Annotation Based Configuration

- Spring dependency injection uses @nnotations instead configuring the beans in XML file with <bean id=“….”>

- Move the bean configuration into the component class itself by using annotations on the relevant class, method, or field declaration using **@Autowired**

Java based config@Configurtion & @Bean

- **@Configuration & @Bean Annotations:**

- Annotating a class with the **@Configuration** indicates that the class can be used by the Spring IoC container as a source of bean definitions.

- The **@Bean** annotation tells Spring that a method annotated with will return an object that should be registered **@Bean** as a bean in the Spring application context.

** Demo

Example

XML Configuration

```
<beans> <bean id = "helloWorld" class = "com.mycom.HelloWorld" /> </beans>
```

Java Based Configuration

```
package com.mycom;
import org.springframework.context.annotation.*;
@Configuration
public class HelloWorldConfig {
          @Bean public HelloWorld helloWorld(){
          return new HelloWorld();
 }
}
```

** Demo

Bean Scopes

Singleton:

•Default scope.

•Creates a single instance of the bean and shares it across the entire application context.

•Suitable for stateless beans that don't maintain any state specific to individual requests or users.

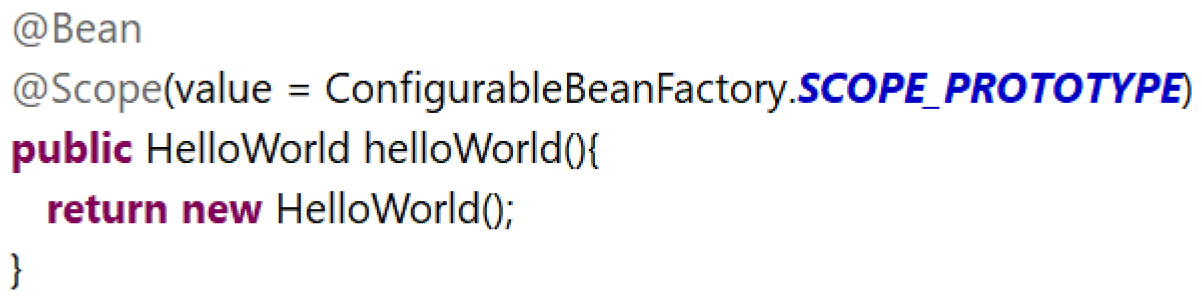
```
@Bean
@Scope(value = ConfigurableBeanFactory.SCOPE_SINGLETON)
public HelloWorld helloWorld(){
   return new HelloWorld();
}
```

** Demo

Bean Scopes …

•**Prototype:** Creates a new instance of the bean every time it's requested.

•Used for stateful beans that need to maintain unique state for each client or request.



```java
@Bean
@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public HelloWorld helloWorld(){
   return new HelloWorld();
}
```

** Demo

Bean Scopes …

•**Request:**

•Web-specific scope.

•Creates a new instance of the bean for each HTTP request.

•Useful for request-scoped data that needs to be isolated for each request.

•Example: Request parameters, model attributes.

Bean Scopes …

- **Session:**
- Web-specific scope.
- Creates a new instance of the bean for each HTTP session.
- Used for session-scoped data that needs to be shared within a single session.
- Example: User preferences, shopping cart items.

Bean Scopes …

Application:

•Web-specific scope.

•Creates a single instance of the bean that is shared across all HTTP sessions within the same web application.

•Suitable for application-wide data that needs to be accessible from any session.

•Example: Application-level configuration, global resources.

Thank You