JAVA FULL STACK DEVELOPMENT PROGRAM

Session 14: Spring Introduction & Dependency Injection

OUTLINE

- Overview of Spring Framework
- Dependency Injection
- Inversion of Control Container

SPRING FRAMEWORK

- Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications
- Spring handles the infrastructure so you can focus on your application
- Examples of how you, as an application developer, can use the Spring platform advantage:
  - Make a Java method execute in a database transaction without having to deal with transaction APIs
  - Make a local Java method a remote procedure without having to deal with remote APIs.
  - Make a local Java method a message handler without having to deal with JMS APIs.

SPRING FRAMEWORK

- Some Advantages of Spring framework
  - Predefined Templates
    - i.e. Spring provides us HibernateTemplate, which means that we don't need to write the code for exception handling, creating connection, creating statement, committing transaction, closing connection etc. All we need to do is writing code for executing query.
- Loose Coupling — Dependency Injection
- Easy to test
- Modularity

DEPENDENCY

- You work in an organization where you and your colleagues tend to travel a lot. Your typical travel planning routine might look like the following:

  - Decide the destination, and desired arrival date and time
  - Call up the airline agency and convey the necessary information to obtain a flight booking.
  - Call up the cab agency, request for a cab to be able to catch a particular flight
  - Pickup the tickets, catch the cab and be on your way

DEPENDENCY

- Now, what if your company suddenly changed the preferred agencies and their contact mechanisms?

- You would be subject to the following relearning scenarios:

  - The new agencies, and their new contact mechanisms (say the new agencies offer internet based services and the way to do the bookings is over the internet instead of over the phone)

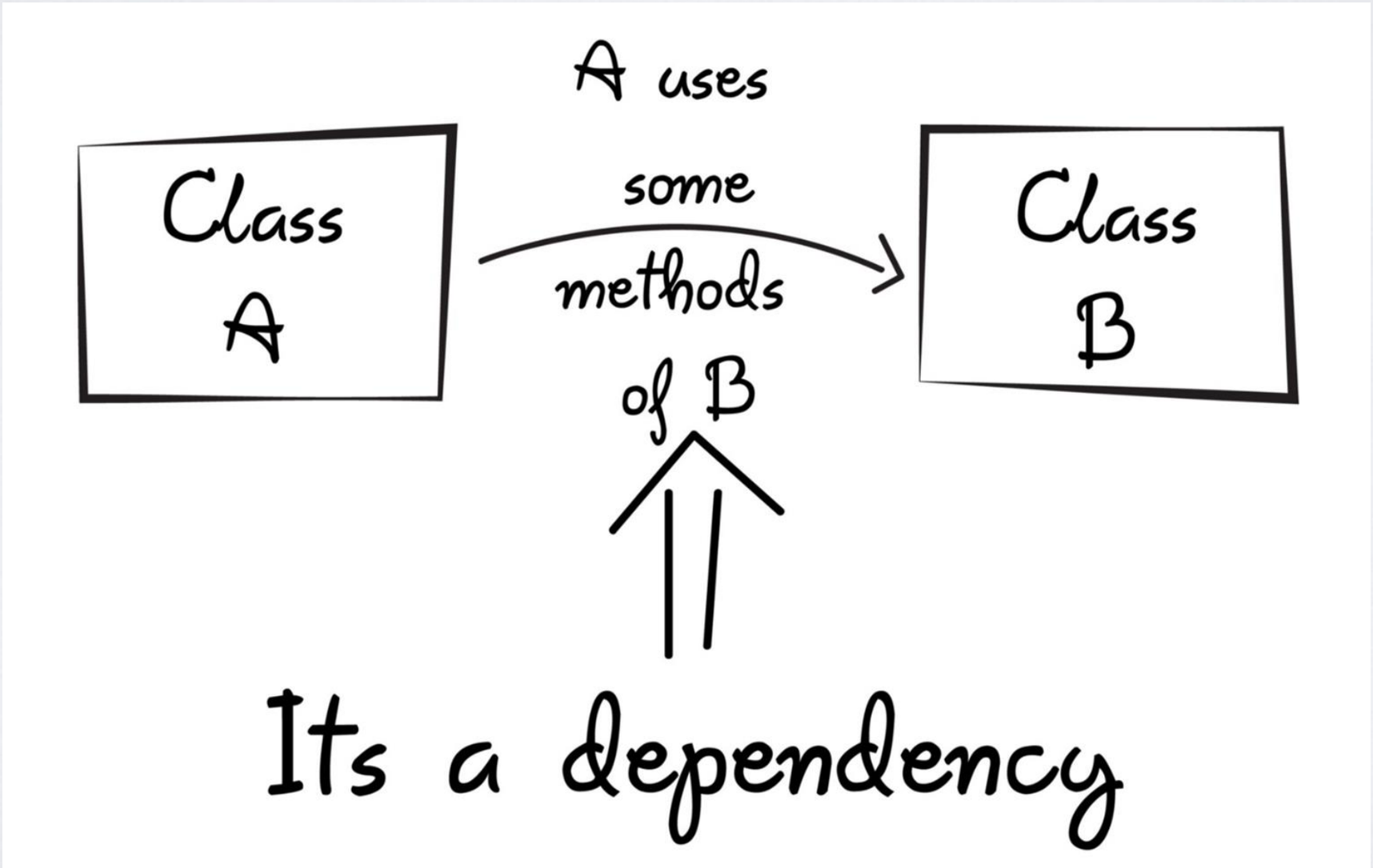
  - The typical conversational sequence through which the necessary bookings get done (Data instead of voice)

DEPENDENCY

- Now let's say the protocol is a little bit different.
  - There is an administration department in the company
  - Whenever you needed to travel an administration department interactive telephony system simply calls you up (which in turn is hooked up to the agencies)
  - Over the phone you simply state the destination, desired arrival date and time by responding to a programmed set of questions
  - The flight reservations are made for you, the cab gets scheduled for the appropriate time, and the tickets get delivered to you

DEPENDENCY

- When class A uses some functionality of class B, then its said that class A has a dependency of class B.



Coupling

- In object oriented design, Coupling refers to the degree of direct knowledge that one element has of another. In other words, how often do changes in class A force related changes in class B.

- Tightly coupling

- Loose coupling

TIGHTLY COUPLING

```
Public class Traveller {
    Car c = new Car();
    Public void startJourney() {
        c.move();
    }
}

Public class Car {
    Public void move(){
        ...
    }
}
```

```
Public class Traveller {
    Plane p = new Plane();
    Public void startJourney() {
        p.move();
    }
}

Public class Plane {
    Public void move(){
        ...
    }
}
```

LOOSE COUPLING

```
Public class Traveller {
    Vehicle v;

    Public void setV(Vehicle v) {
        this.V = V;
    }
    Public void startJourney() {
        V.move();
    }
}
```

DEPENDENCY INJECTION

- What is dependency Inject?
  - Separating the usage from the creation of the object
- Why dependency injection? (or Decoupling)
  - It improves the testability
  - It's much easier to swap other pieces of code/modules/objects/components when the pieces are not dependent on one another
  - Modularity — One module doesn't break other modules in unpredictable ways

EASY TO TEST

```
Public class Traveller {
    Car c = new Car();
    Public void startJourney() {
        c.move();
    }
}

Public class Car {
    Public void move(){
        ...
    }
}
```

```
Public class Traveller {
    Vehicle v;

    Public void setV(Vehicle v) {
        this.V = V;
    }
    Public void startJourney() {
        V.move();
    }
}
```

IOC CONTAINER

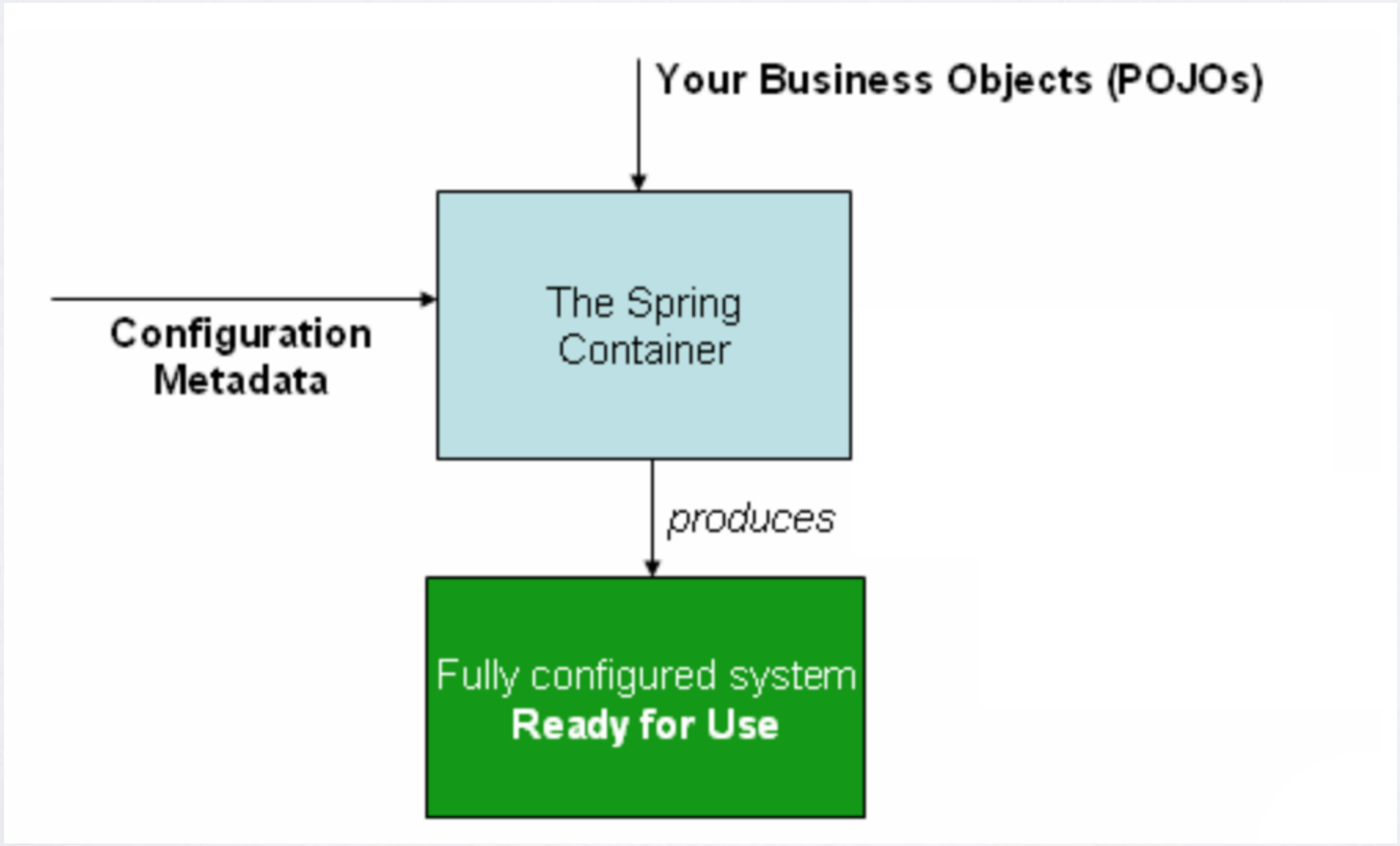
- IoC — Inversion of Control
  - It is a process whereby objects define their dependencies (that is, the other objects they work with) only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method.
  - Dependency injection is a pattern through which to implement IoC, where the control being inverted is the setting of object's dependencies.
- Spring IoC container injects those dependencies when it creates the bean.

SPRING IOC CONTAINER

- The *org.springframework.beans* and *org.springframework.context* packages are the basis for Spring Framework's IoC container.
- The *BeanFactory* interface provides an advanced configuration mechanism capable of managing any type of object
- The *ApplicationContext* is a sub-interface of *BeanFactory* with additional features:
  - Easier integration with Spring's AOP features
  - Message resource handling (for use in internationalization)
  - Application-layer specific contexts such as the *WebApplicationContext* for use in web applications

SPRING IOC CONTAINER

- The *org.springframework.context.ApplicationContext* interface represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the beans



CONFIGURATION METADATA

- As the preceding diagram shows, the Spring IoC container consumes a form of configuration metadata.

- This configuration metadata represents how you, as an application developer, tell the Spring container to instantiate, configure, and assemble the objects in your application.

- Configuration can be done in two ways:

  - XML

  - Annotation

XML CONFIGURATION

```xml
<?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
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="employee" class="com.bfs.Employee">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions go here -->

</beans>
```

- The id attribute is a string that identifies the individual bean definition.
- The class attribute defines the type of the bean and uses the fully qualified classname.

SPRING IOC CONTAINER

- Dependency Inject in Spring can be done in three ways:
  - Constructor
  - Setter
  - Field

CONSTRUCTOR INJECTION

- In the case of constructor-based dependency injection, the container will invoke a constructor with arguments each representing a dependency we want to set

```java
package x.y;

public class ThingOne {

    public ThingOne(ThingTwo thingTwo, ThingThree thingThree) {
        // ...
    }
}
```

```xml
<beans>
    <bean id="beanOne" class="x.y.ThingOne">
        <constructor-arg ref="beanTwo"/>
        <constructor-arg ref="beanThree"/>
    </bean>

    <bean id="beanTwo" class="x.y.ThingTwo"/>

    <bean id="beanThree" class="x.y.ThingThree"/>
</beans>
```

CONSTRUCTOR INJECTION

- Constructor argument resolution matching occurs by using the argument's type
  - If no potential ambiguity exists in the constructor arguments of a bean definition, the order in which the constructor arguments are defined in a bean definition is the order in which those arguments are supplied to the appropriate constructor when the bean is being instantiated
- The container can use type matching with simple types if you explicitly specify the type of the constructor argument by using the *type* or *index* or *name* attribute

CONSTRUCTOR INJECTION

```java
package examples;

public class ExampleBean {

    // Number of years to calculate the Ultimate Answer
    private int years;

    // The Answer to Life, the Universe, and Everything
    private String ultimateAnswer;

    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }
}
```

```xml
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg type="int" value="7500000"/>
    <constructor-arg type="java.lang.String" value="42"/>
</bean>
```

```xml
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg index="0" value="7500000"/>
    <constructor-arg index="1" value="42"/>
</bean>
```

```xml
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg name="years" value="7500000"/>
    <constructor-arg name="ultimateAnswer" value="42"/>
</bean>
```

SETTER INJECTION

- Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or a no-argument static factory method to instantiate your bean

```
<beans>
    <bean id="beanOne" class="x.y.ThingOne">
        <property name="thingTwo" ref="beanTwo" />
        <property name="thingTwo" ref="beanThree" />
    </bean>

    <bean id="beanTwo" class="x.y.ThingTwo"/>

    <bean id="beanThree" class="x.y.ThingThree"/>
</beans>
```

CONSTRUTOR VS SETTER INJECTION

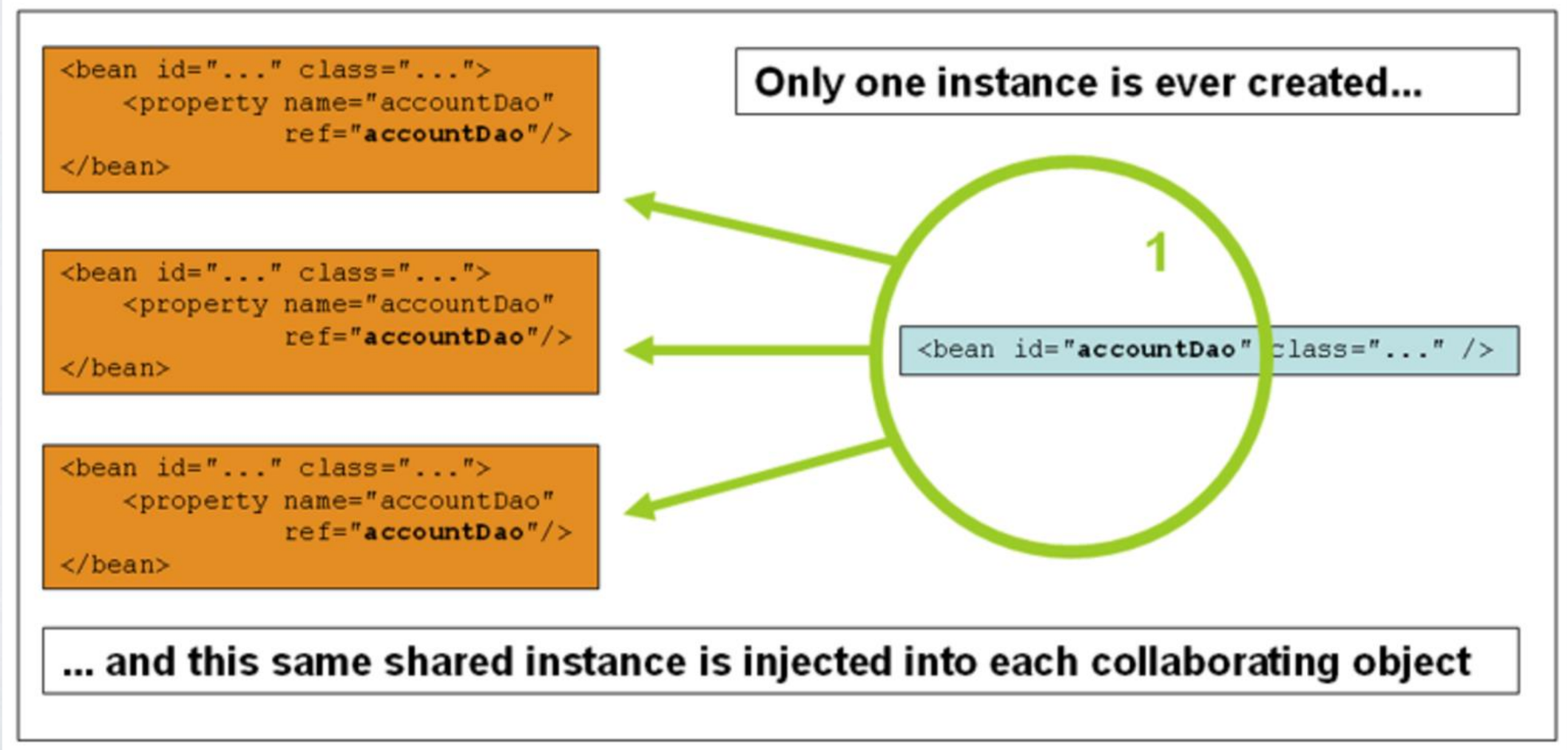
SETTER DI	CONSTRUCTOR DI
The bean must include getter and setter methods for the properties.	The bean class must declare a matching constructor with arguments. Otherwise, BeanCreationException will be thrown.
Circular dependencies or partial dependencies result with Setter DI because object creation happens before the injections.	No scope for circular or partial dependency because dependencies are resolved before object creation itself.
Preferred option when properties are less and mutable objects can be created.	Preferred option when properties on the bean are more and immutable objects (eg: financial processes) are important for application.

SPRING BEAN SCOPE

Scope	Description
Singleton	(Default) Scopes a single bean definition to a single object instance for each Spring IoC container.
Prototype	Scopes a single bean definition to any number of object instances.
Request	Scopes a single bean definition to the lifecycle of a single HTTP request. That is, each HTTP request has its own instance of a bean created off the back of a single bean definition
Session	Scopes a single bean definition to the lifecycle of an HTTP Session
Application	Scopes a single bean definition to the lifecycle of a ServletContext

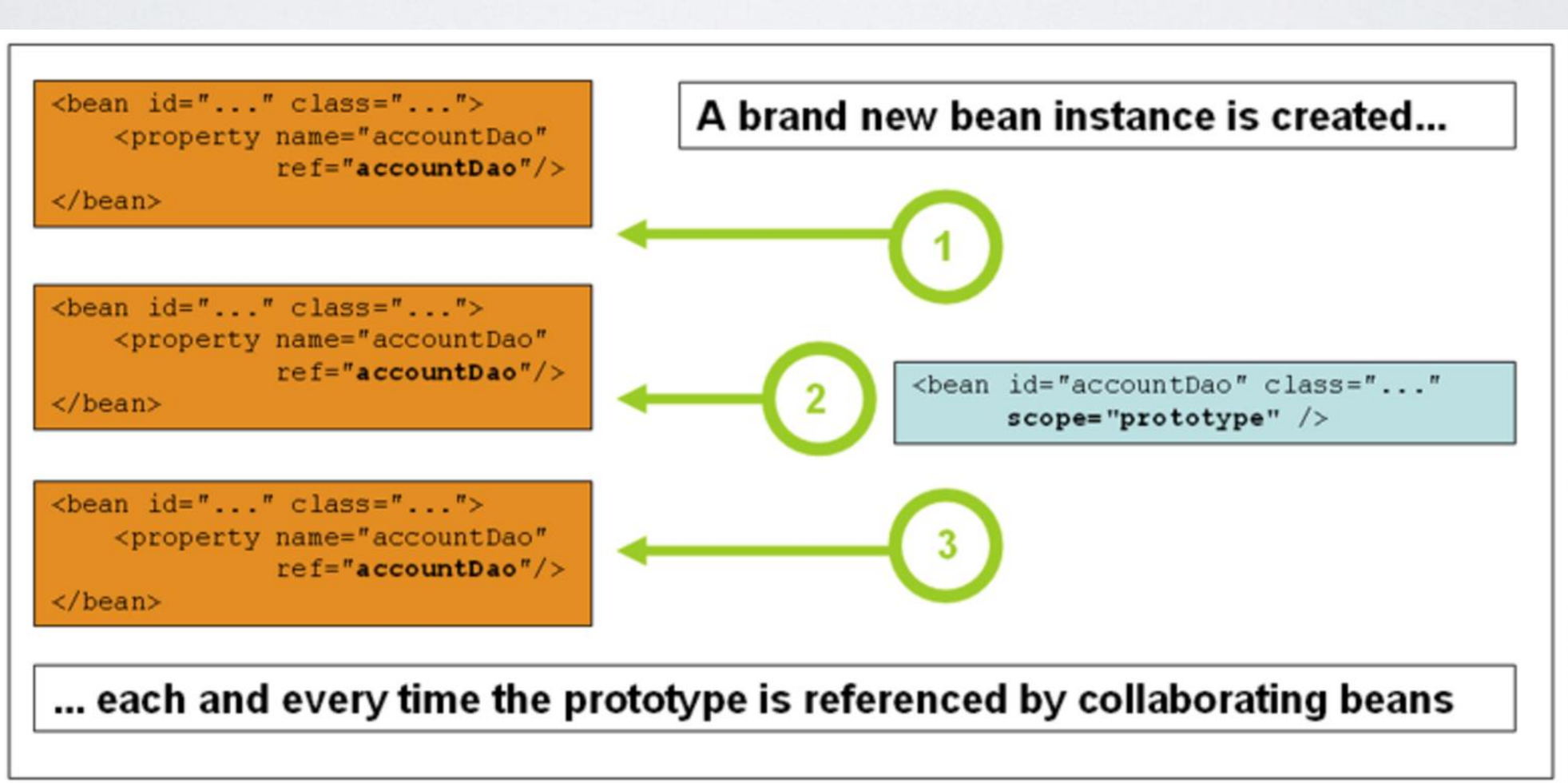
THE SINGLETON SCOPE

- Only one shared instance of a singleton bean is managed, and all requests for beans with an ID or IDs that match that bean definition result in that one specific bean instance being returned by the Spring container.



THE PROTOTYPE SCOPE

- The non-singleton prototype scope of bean deployment results in the creation of a new bean instance every time a request for that specific bean is made



ANNOTATION BASED CONFIGURATION

- How to do Annotation Based Configuration
  - Enable component scanning in Spring config file
  - Add the @Component Annotation to your Java classes
  - Retrieve bean from Spring container

ENABLE SPRING COMPONENT SCANNING

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config/>

</beans>
```

@COMPONENT ANNOTATION

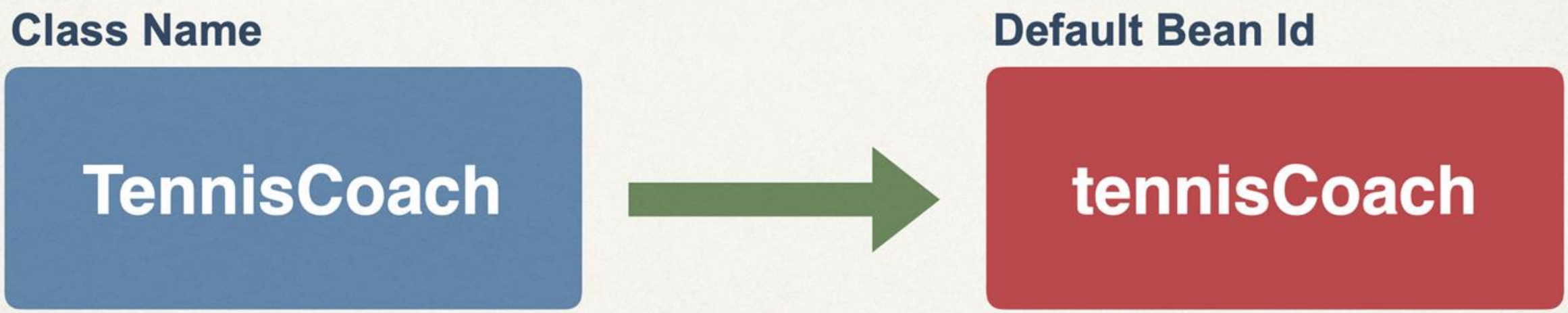
```
@Component("thatSillyCoach")
public class TennisCoach implements Coach {

  @Override
  public String getDailyWorkout() {
    return "Practice your backhand volley";
  }

}
```

@COMPONENT ANNOTATION

```
@Component
public class TennisCoach implements Coach {
```



ANNOTATION DI

- The @Autowired annotation
  - This annotation has the following execution paths, listed by precedence
    - Match by Type
    - Match by Qualifier
    - Match by Name

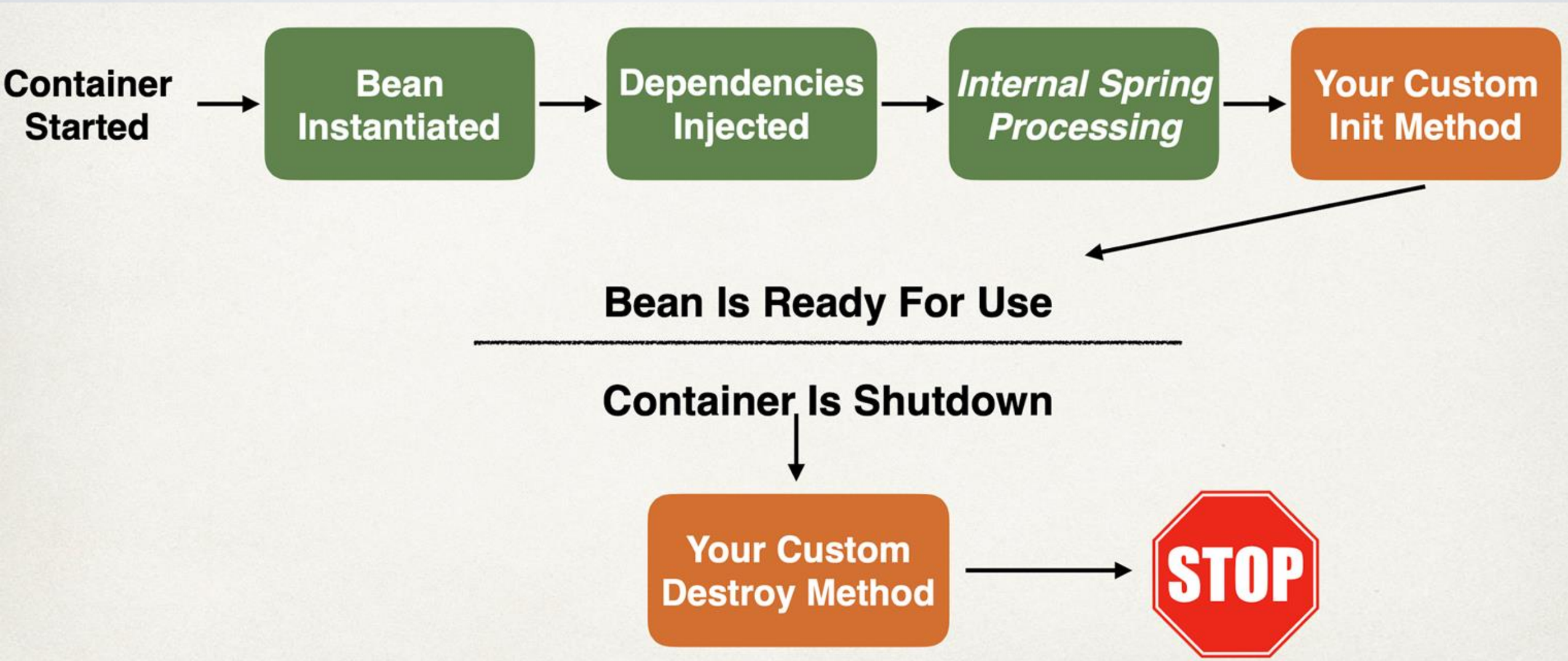
ANNOTATION DI

- Let’s take look at an example

@BEAN AND @CONFIGURATION

- Instead of using XML to do the configuration, we can use purely Java Configuration with
- @Bean
- @Configuration
- @ComponentScan

BEAN LIFE CYCLE



LIFE CYCLE CALLBACKS

- Spring bean factory controls the creation and destruction of beans. To execute some custom code, it provides the call back methods which can be categorized broadly in two groups

- Post-initialization call back methods

- Pre-destruction call back methods

LIFE CYCLE CALLBACKS

```
package com.journaldev.spring.service;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

public class MyService {

	@PostConstruct
	public void init(){
		System.out.println("MyService init method called");
	}

	public MyService(){
		System.out.println("MyService no-args constructor called");
	}

	@PreDestroy
	public void destory(){
		System.out.println("MyService destroy method called");
	}
}
```

LIFE CYCLE CALLBACKS

```xml
<?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.springframew

<!-- Not initializing employee name variable-->
<bean name="employee" class="com.journaldev.spring.bean.Employee" />

<bean name="employeeService" class="com.journaldev.spring.service.EmployeeService">
       <property name="employee" ref="employee"></property>
</bean>

<bean name="myEmployeeService" class="com.journaldev.spring.service.MyEmployeeService"
              init-method="init" destroy-method="destroy">
       <property name="employee" ref="employee"></property>
</bean>

<!-- initializing CommonAnnotationBeanPostProcessor is same as context:annotation-config -->
<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor" />
<bean name="myService" class="com.journaldev.spring.service.MyService" />
</beans>
```

ANY QUESTION?