Partner ORACLE UNIVERSITY

Certifica TIC Beyond Certification

INICIO

Desarrollo de Aplicaciones Empresariales con Spring Framework Core 5

ISC. Ivan Venor García Baños

ATC Advanced Training Consultants

SCRUMstudy Authorized Training Partner

Agenda

1. Presentación
2. Objetivos
3. Contenido
4. Despedida

3. Contenido

i. Introducción a Spring Framework
ii. Spring Core
iii. Spring AOP
iv. Spring JDBC – Transaction
v. Spring ORM – Hibernate 5
vi. Spring Data JPA
vii. Fundamentos Spring MVC y Spring REST
viii. Fundamentos Spring Security
ix. Seguridad en Servicios REST
x. Introducción Spring Boot

ii. Spring Core

ii. Spring Core (a)

ii.i Spring Core Conceptos

a. Inversión de Control

b. Inyección de Dependencias

c. Inversión de Dependencias

ii.ii Contenedor de IoC

a. BeanFactory

Práctica 2. Hola Mundo Spring Framework

b. ApplicationContext

c. Tipos de configuración de Beans

ii. Spring Core (b)

ii.iii Configuración de Beans con XML

a. Definición de Beans

b. Inyección de Dependencias

Práctica 3. Inyección de Dependencias Jugadores

Práctica 4. Inyección de Dependencias Movie Finder

Tarea 1. Implementación Notification Service

ii.iv Bean Scopes

a. Singleton

b. Prototype

c. Custom Scope

Práctica 5. Bean Scopes

ii. Spring Core (c)

ii.v Ciclo de vida de Beans

a. Inicialización y Destrucción

b. Inicialización Lazy

c. Factory Method

Práctica 6. Init – Destroy

Práctica 7. Lazy Beans

Práctica 8. Factory Method

d. Apagando el contenedor de IoC

e. Startup y Shutdown Callbacks

Práctica a. Startup y Shutdown Callbacks

f. Interfaces Aware

ii. Spring Core (d)

ii.vi Definición heredada de Beans

Práctica 9. Bean Templates

ii.vii Puntos de extensión del Contenedor

a. Puntos de extensión del Contenedor
b. BeanPostProcessor
c. BeanFactoryPostProcessor
d. FactoryBean

Práctica 10. BeanPostProcessor

Práctica b. BeanFactoryPostProcessor

ii. Spring Core (e)

ii.viii Definición de Beans internos

Práctica 11. Beans Internos

ii.ix Inyección de Colecciones y Arreglos

Práctica 12. Inyección de Colecciones y Arreglos

ii.x Namespace p, c y util

Tarea 2. Ejemplo namespace p, c y util

ii. Spring Core (f)

ii.xi Autowiring

a. byName

b. byType

c. constructor

Práctica 13. Autowiring

Trabajo de Integración 1. Convertidor número letra configuración XML.

Práctica 14. Convertidor número letra configuración XML

ii. Spring Core (g)

ii.xii Configuración con @Anotaciones

a. Namespace context

b. @Required, @Autowired y @Qualifier

Práctica 15. @Required, @Autowired y @Qualifier

ii.xiii Anotaciones JSR 250

a. @Resource, @PostConstruct y @PreDestroy

Práctica 16. @Resource, @PostConstruct y @PreDestroy

ii. Spring Core (h)

ii.xiv Component-scan

a. Estereotipos @Component, @Service, @Repository, @Controller y @RestController

Práctica 17. Component-scan y estereotipos

b. Filtrado de componentes con @ComponentScan

Práctica c. Filtrado de componentes

ii.xv Anotaciones JSR 330

a. @Inject y @Named

Práctica 18. @Inject y @Named

ii. Spring Core (i)

ii.xvi Spring Java Config

a. @Configuration, @Bean e @Import

Práctica 19. @Configuration, @Bean e @Import

Trabajo de Integración 2. Convertidor número letra configuración por @Anotaciones.

Tarea 3. Migración Convertidor número letra configuración por @Anotaciones

ii. Spring Core (j)

ii.xvii Resources

a. Tipos de Resources

b. Tipos de ApplicationContext

c. Inyección y obtención de recursos

d. Obtención de archivos de propiedades

Práctica 20. Resources

e. @PropertySource

Práctica d. @PropertySource

ii. Spring Core (k)

ii.xviii Spring Expression Language (SpEL)

a. Introducción

b. Evaluación de expresiones

c. Referencias del lenguaje

Práctica 21. API SpEL

ii.i Spring Core Conceptos

Objetivos de la lección

ii.i Spring Core Conceptos

- Comprender la Inversión de Control (IoC)
- Comprender la Inyección de Dependencias (DI)
- Comprender la Inversión de Dependencias.

ii.i Spring Core Conceptos

a. Inversión de Control

b. Inyección de Dependencias

c. Inversión de Dependencias

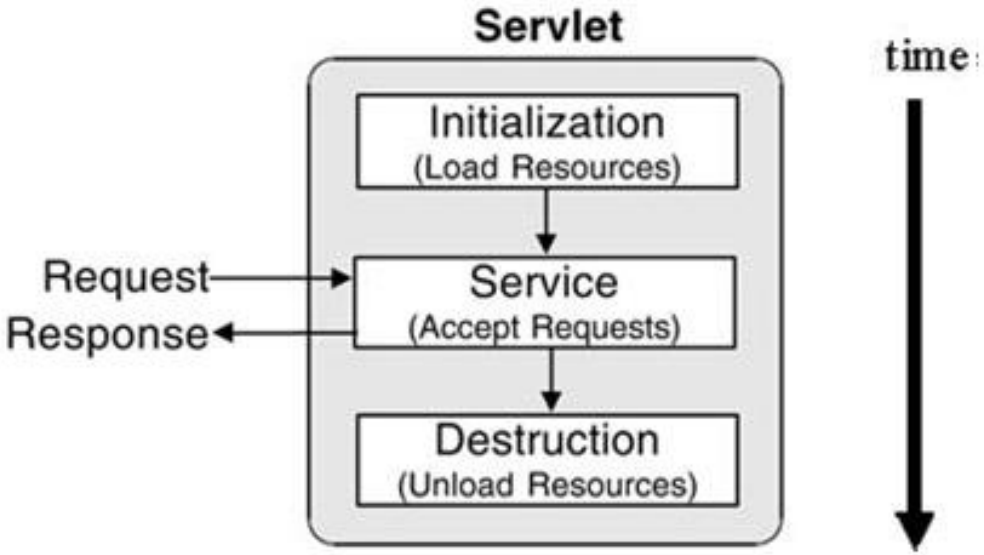
ii.i Inversión de Control (a)

- Principio de Hollywood
  - Don´t call us we´ll call you

- Inversión de Control (Inversion of Control, IoC), es un estilo de programación en el cual un framework o librería controla el flujo de ejecución de un programa. Esto es un cambio con respecto a paradigmas tradicionales donde el programador especifica todo el flujo de un programa.

ii.i Inversión de Control (b)

- IoC ejemplo:



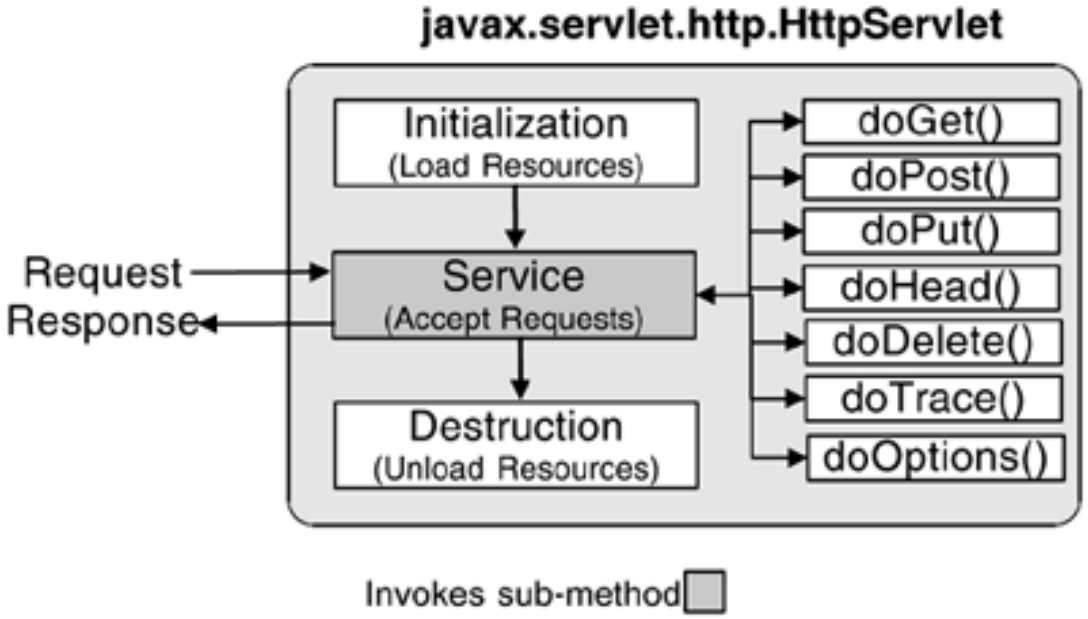
ii.i Inversión de Control (c)

- Tradicionalmente el programador especifica la secuencia de decisiones y procedimientos que deben ejecutarse durante el ciclo de vida de una aplicación.

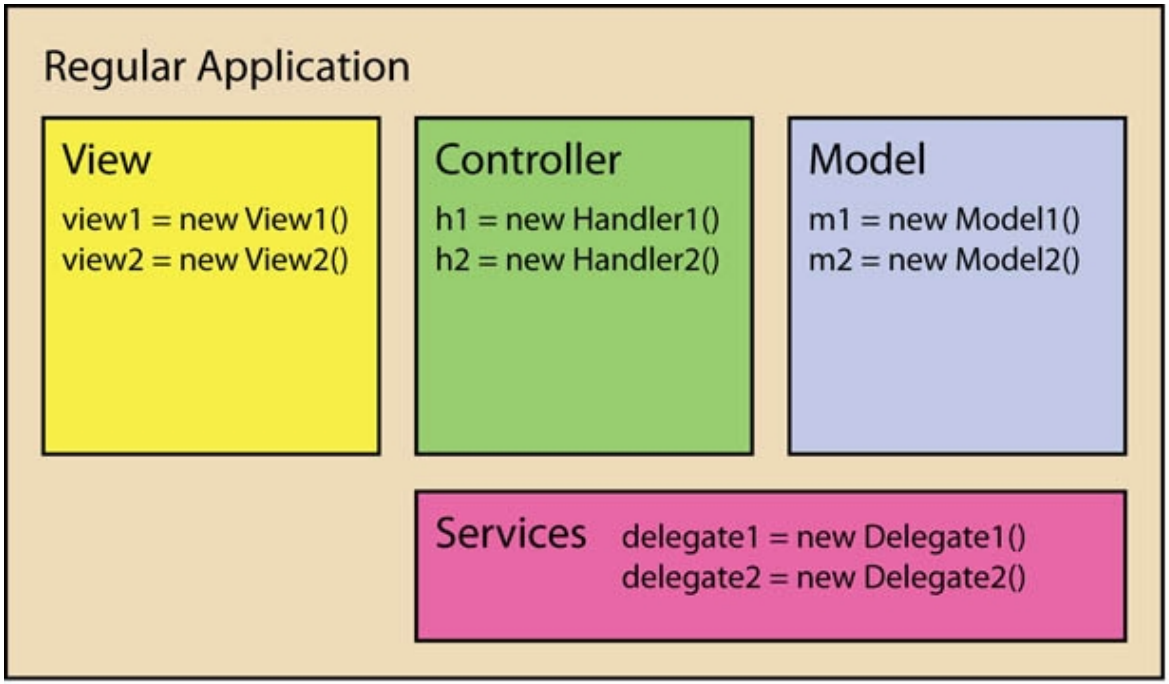
- La Inversión de Control especifica respuestas deseadas a sucesos o solicitudes concretas, dejando que algún framework, librería, entidad o arquitectura externa lleve a cabo las acciones de control que tengan que ejecutarse, en el orden necesario y para el conjunto de sucesos que tengan que ocurrir.

ii.i Inversión de Control (d)

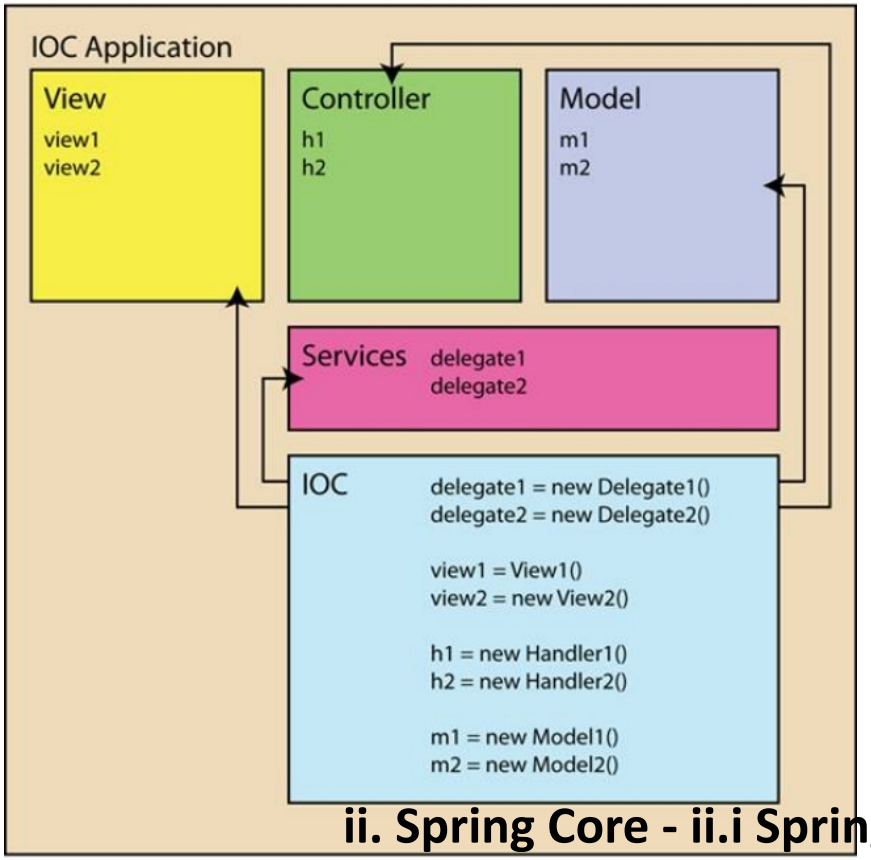
- IoC ejemplo:



ii.i Inversión de Control (e)



ii.i Inversión de Control (f)



ii.i Inversión de Control (g)

- El principio de Inversión de Control en Spring (IoC) es la habilidad de poder intercambiar el modo en el que los objetos son construidos, dando el control a Spring para gestionar la creación, inicialización, así como todo el ciclo de vida de un objeto hasta su destrucción.

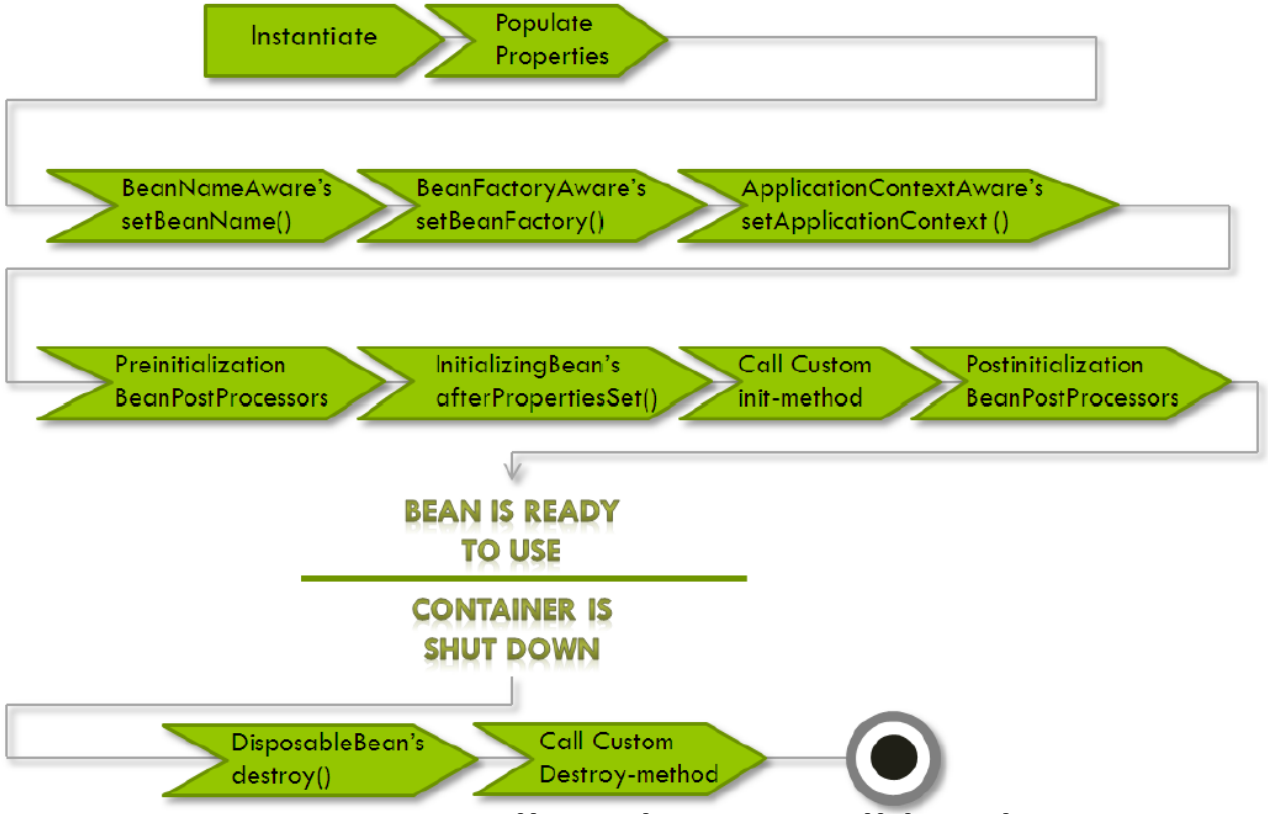
ii.i Inversión de Control (h)

- Spring bean lifecycle (introducción)

- Spring administra un ciclo de vida para la construcción y destrucción de los objetos (beans) bien definido, el cual no es suprimible, pero si configurable.

- SOLID.
  - Open/Close principle

ii.i Inversión de Control (i)



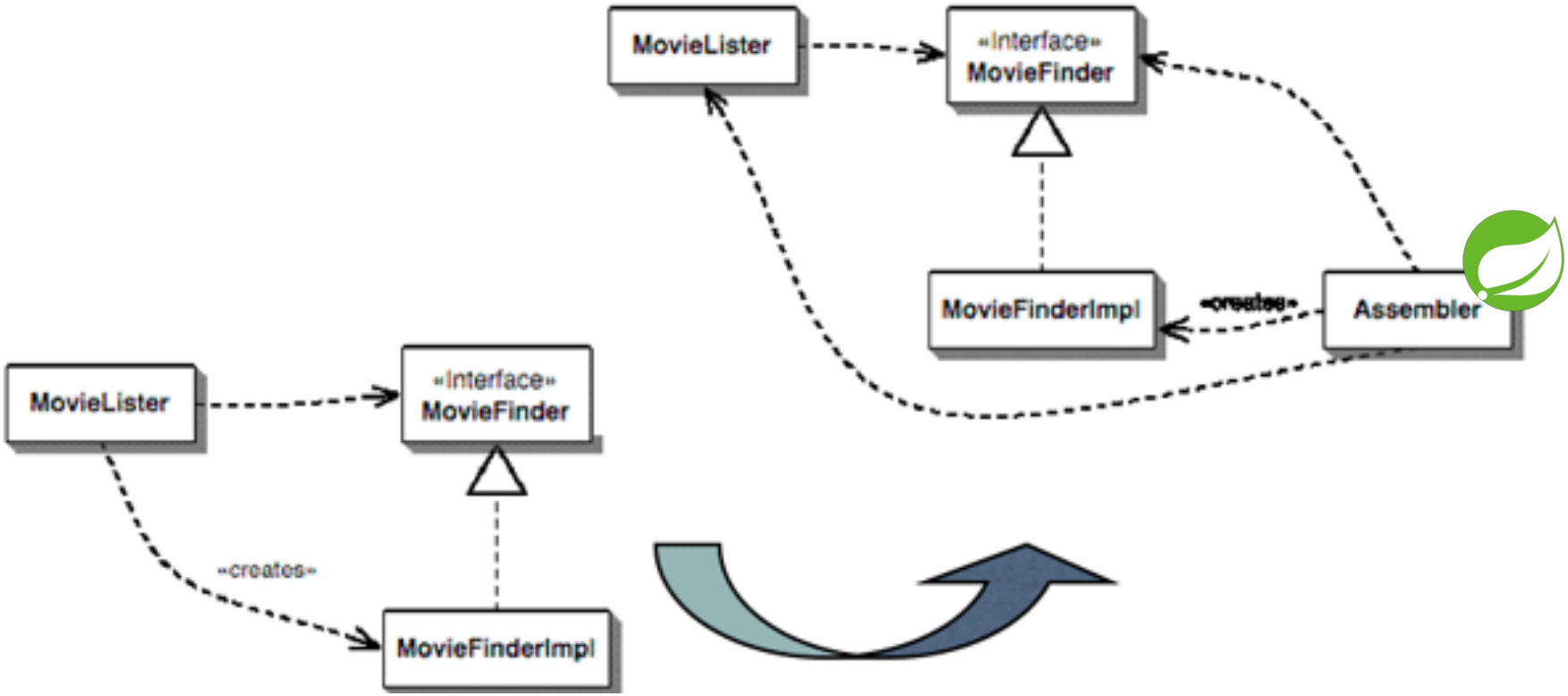
ii.i Inversión de Control (j)

- IoC es una técnica que invierte el flujo tradicional de una aplicación.

- Lo tradicional es que el código de la aplicación llame a las librerías; la inversión de control ocurre cuando son las librerías las que llaman al código de la aplicación.

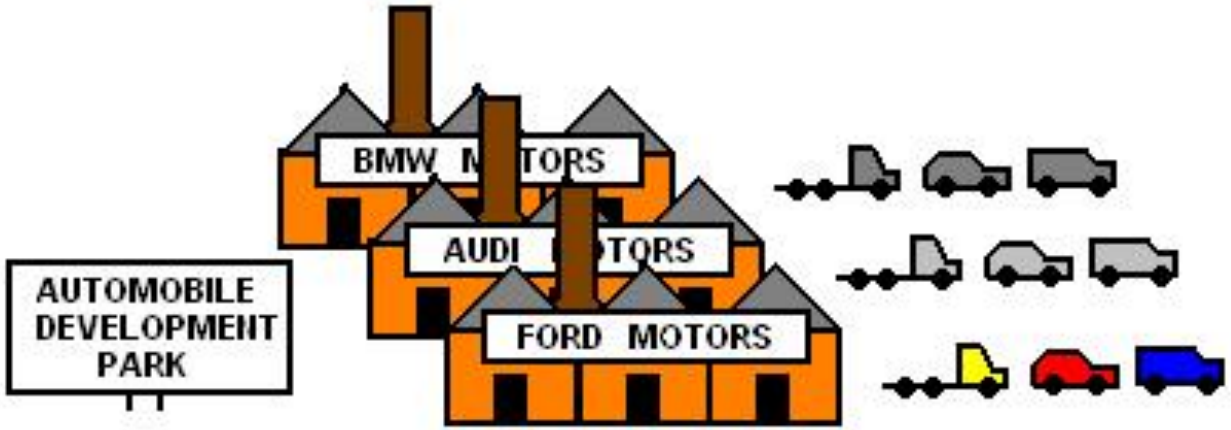
- En Spring, la inversión de control consiste en ceder el control a una entidad externa a la aplicación, llamada "Contenedor“ (Contenedor de IoC), el cual se encargará de gestionar las instancias (beans) de la aplicación (creaciones, configuraciones y destrucciones).

ii.i Inversión de Control (k)



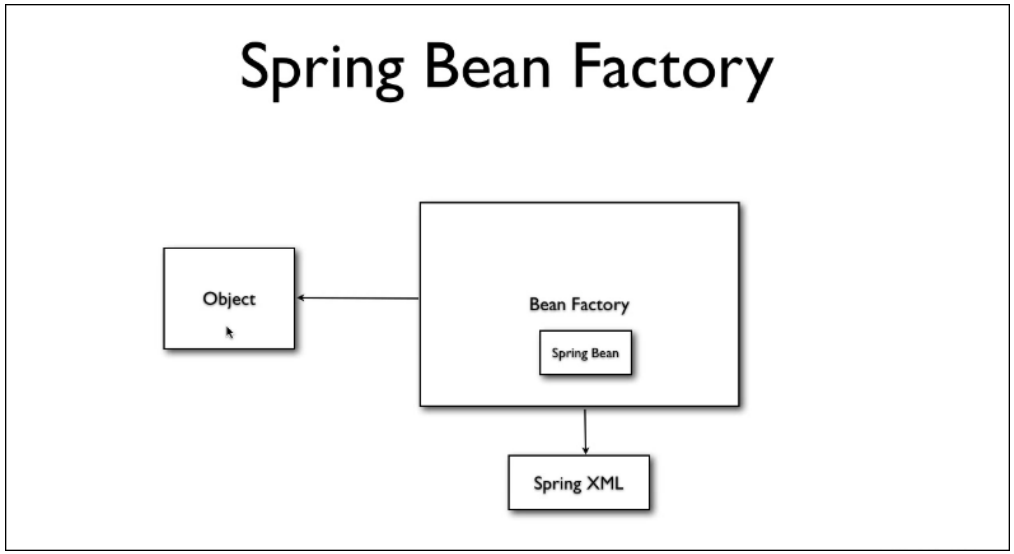
ii.i Inversión de Control (I)

- IoC centraliza la construcción de objetos.
- El contenedor de IoC es implementado por un tercero, Spring Framework.
- Utiliza el Patrón de Diseño Factory (BeanFactory)



ii.i Inversión de Control (m)

- Para que Spring pueda construir los objetos requeridos, es necesaria la aplicación de otro concepto llamado Inyección de Dependencias.



ii.i Spring Core Conceptos

a. Inversión de Control
b. Inyección de Dependencias
c. Inversión de Dependencias

ii.i Inyección de Dependencias (a)

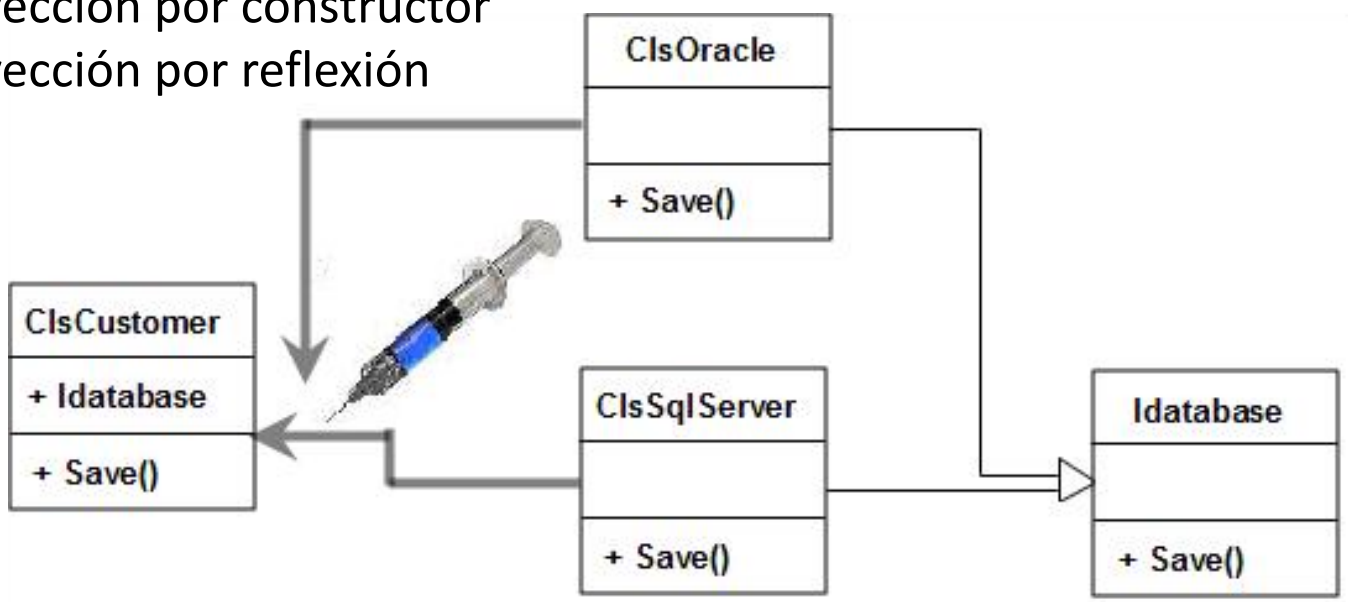
- La Inyección de Dependencias es un patrón de diseño orientado a objetos, en el que se suministran objetos a una clase en lugar de ser la propia clase quien cree los objetos que requiere.

ii.i Inyección de Dependencias (b)

- Creación de dependencias (componentes / beans)
- Asignación de beans componentes en beans dependientes

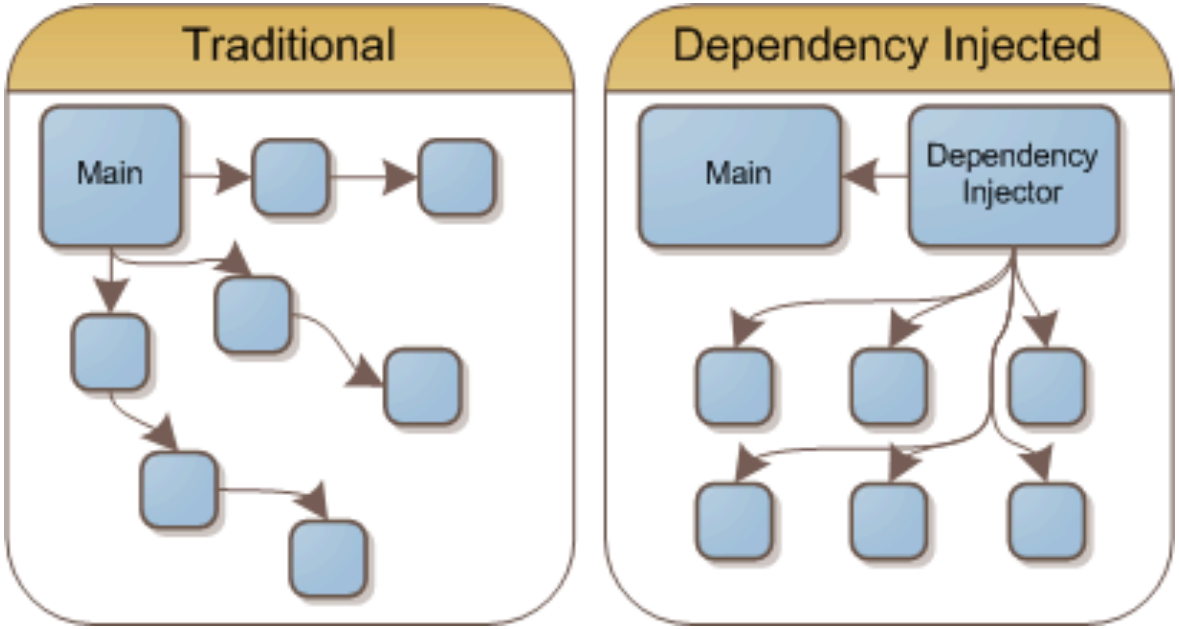
ii.i Inyección de Dependencias (c)

- Inyección por setter
- Inyección por constructor
- Inyección por reflexión



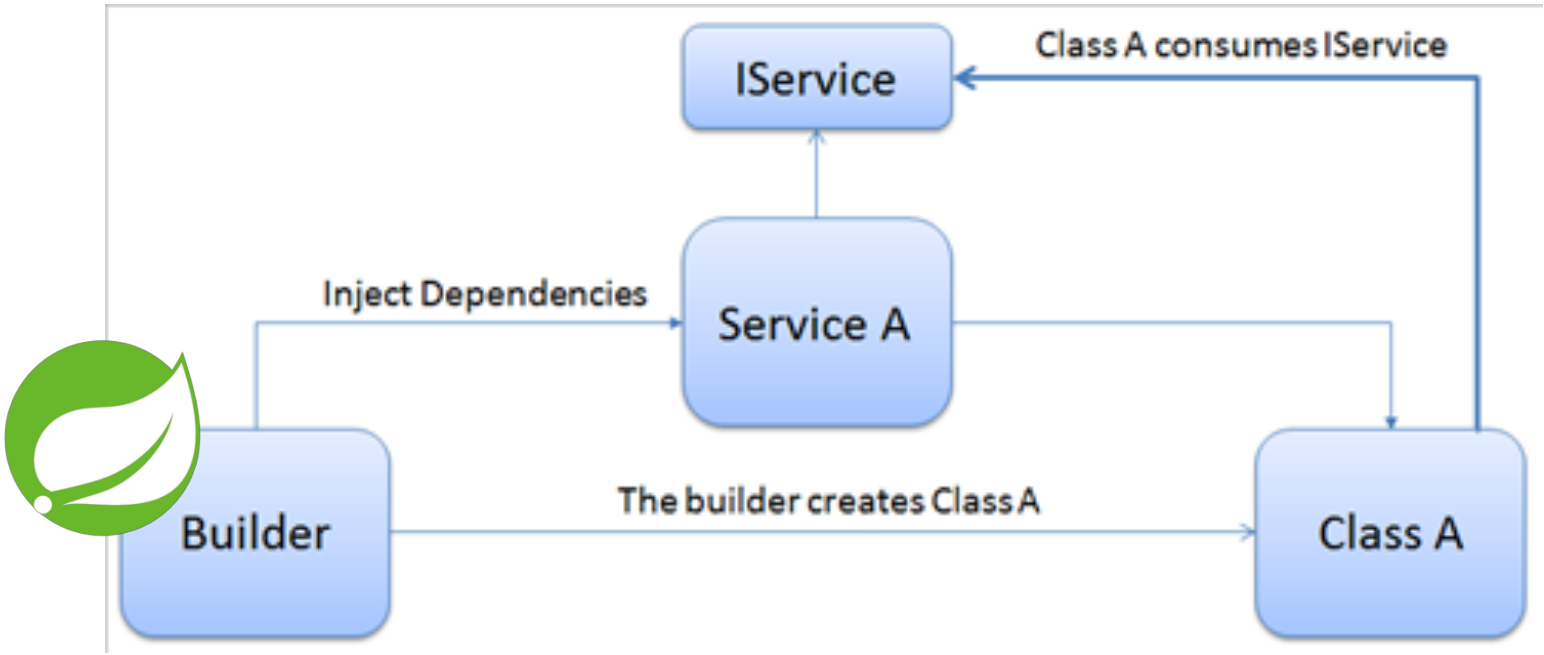
ii.i Inyección de Dependencias (d)

- Tradicional (Sin IoC) vs IoC + DI



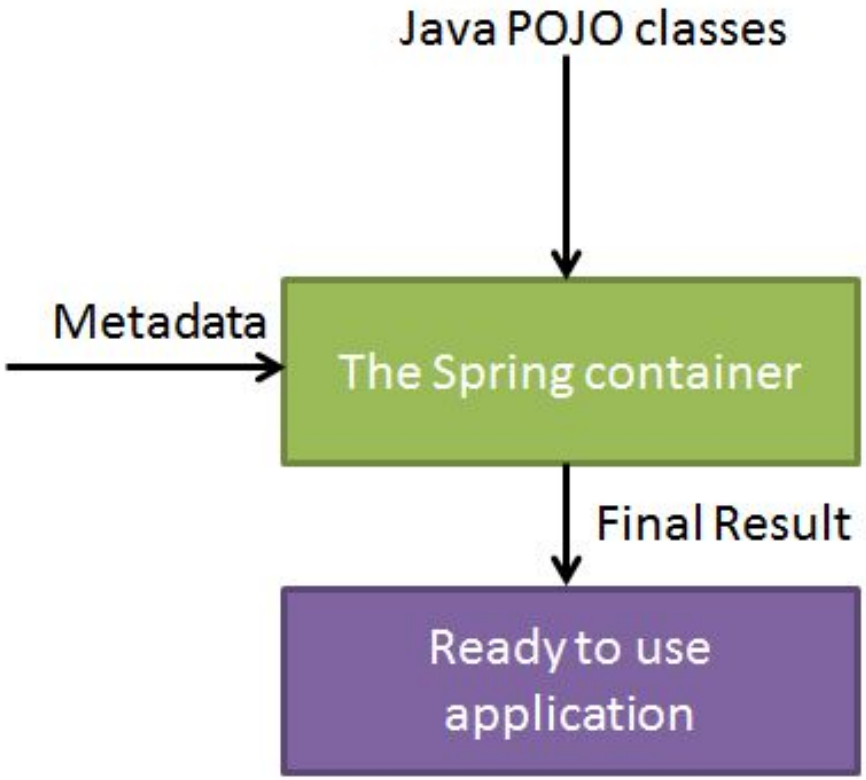
ii.i Inyección de Dependencias (e)

- IoC + DI



ii.i Inyección de Dependencias (f)

- Spring DI

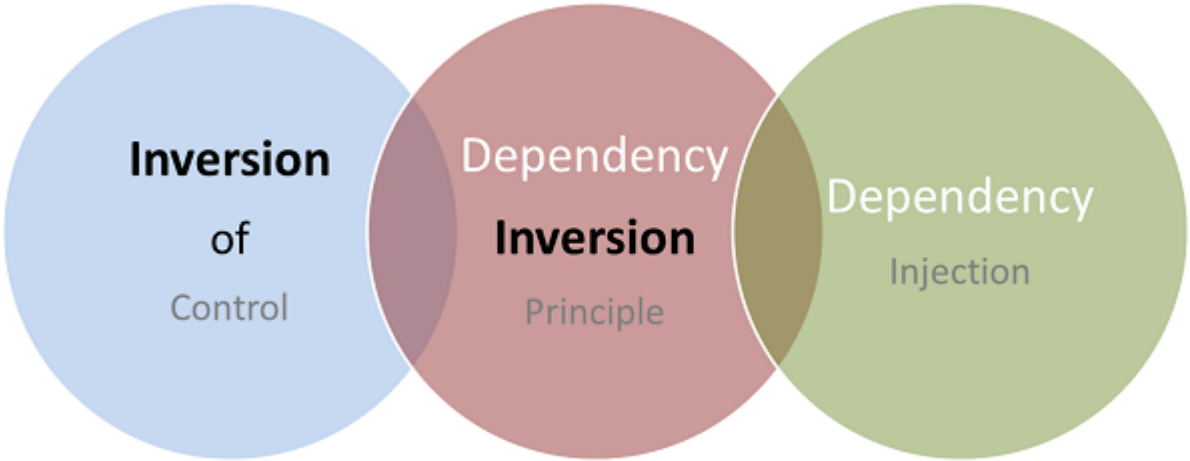


ii.i Spring Core Conceptos

a. Inversión de Control
b. Inyección de Dependencias
c. Inversión de Dependencias

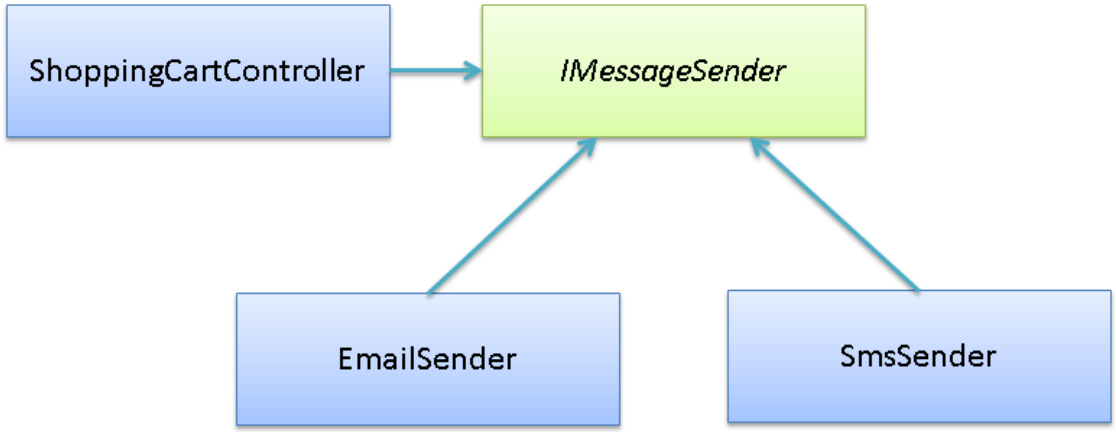
ii.i Inversión de Dependencias (a)

- Inversión de Control no es Inyección de Dependencias



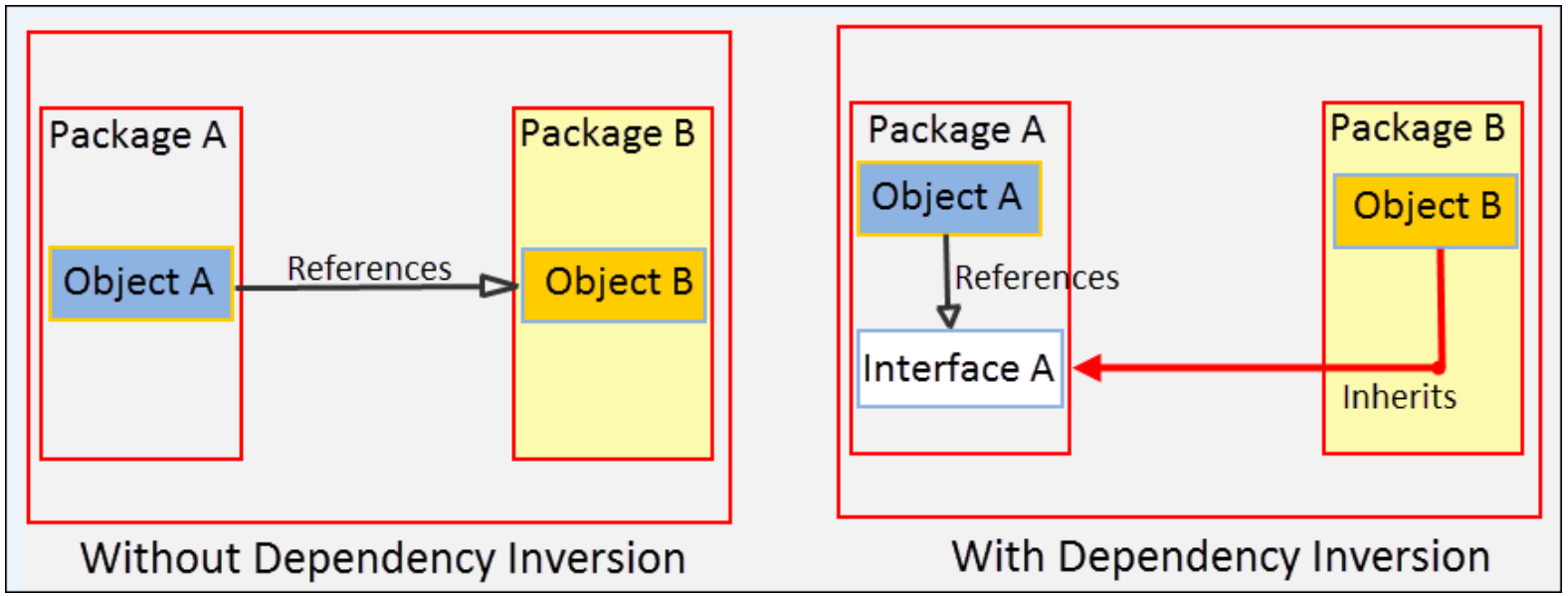
ii.i Inversión de Dependencias (b)

- Las clases de alto nivel no deberían depender de las clases de bajo nivel, ambas deberían depender de las abstracciones.
- Las abstracciones no deberían depender de los detalles, son los detalles los que deberían depender de las abstracciones.



ii.i Inversión de Dependencias (c)

- Ayuda a mantener el código totalmente desacoplado, asegurando la dependencia con abstracciones (interfaces) y no con clases concretas.



Resumen de la lección

ii.i Spring Core Conceptos

- Comprendimos lo que es la Inversión de Control y sus beneficios.
- Analizamos la diferencia entre Inversión de Control e Inyección de Dependencias.
- Conocimos maneras de implementar la Inyección de Dependencias (DI)
- Conocimos las mejores prácticas en desarrollo de software SOLID
- Comprendimos el principio de Inversión de Dependencias

Esta página fue intencionalmente dejada en blanco.

ii.ii Contenedor de IoC

Objetivos de la lección

ii.ii Contenedor de IoC

- Conocer los distintos contenedores de Inversión de Control de Spring.
- Conocer las distintas formas de configuración de Beans.
- Implementar práctica Hola Mundo Spring.

ii.ii Contenedor de IoC

a. BeanFactory

Práctica 2. Hola Mundo Spring Framework

b. ApplicationContext

c. Tipos de configuración de Beans

ii.ii Contenedor de IoC (a)

- Spring Core:
  - Provee contenedor de IoC e DI (contenedor de beans)
  - Implementa patrones de diseño
  - Objeto BeanFactory (Factory pattern)
  - IoC nos evita gestionar el ciclo de vida de los objetos
  - Genera Singletons por default (no thread safe)

ii.ii Contenedor de IoC (b)

- Spring Core:
  - El contenedor gestionará la creación, inyección y configuración de los objetos (ciclo de vida del bean).
  - El contenedor de IoC de Spring utiliza inyección de dependencias (DI) para administrar los componentes de la aplicación.
  - Estos objetos serán llamados como Spring Beans o simplemente beans.

ii.ii Contenedor de IoC (c)

- Contenedores de IoC
  - BeanFactory
  - ApplicationContext

ii.ii Contenedor de IoC (d)

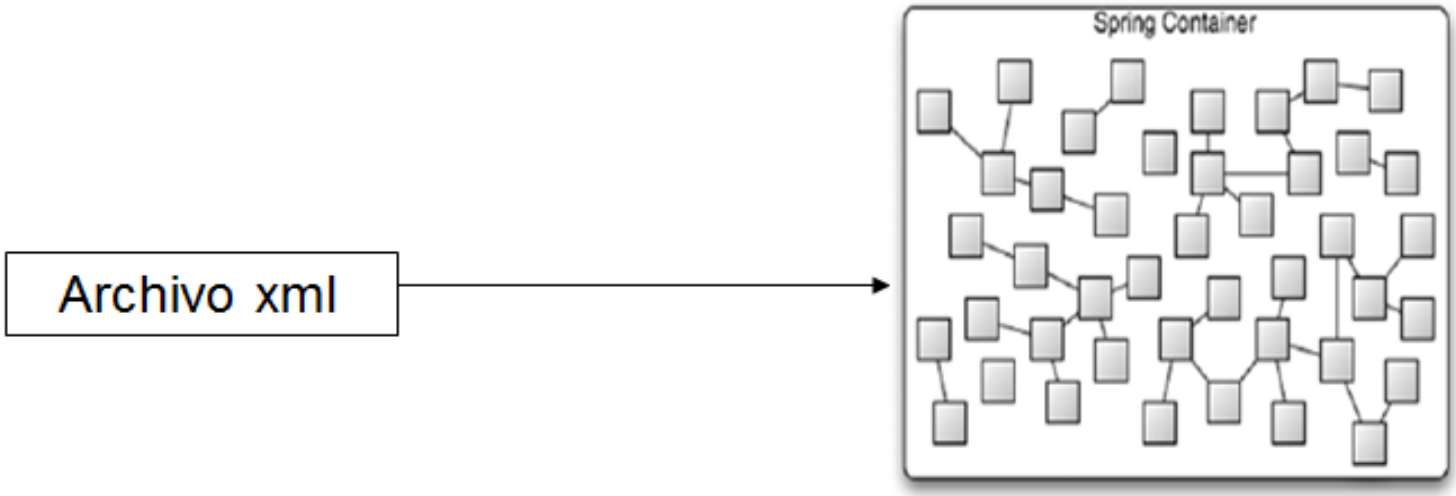
- El contenedor ejecutará las instrucciones configuradas para saber sobre qué Clases deberá crear instancias, configurar y preparar para su uso.

- La configuración de beans se genera mediante metadatos de configuración en formato XML conocido como Bean Configuration File o Bean Definition Application Context.

- El Bean Configuration File (archivo XML) contiene la definición (metadatos) de beans que el contenedor IoC de Spring debe leer para crear, inicializar y gestionar el ciclo de vida de los beans en la aplicación.

ii.ii Contenedor de IoC (e)

- Bean Configuration File ó Bean Definition Application Context.



ii.ii Contenedor de IoC

a. BeanFactory

Práctica 2. Hola Mundo Spring Framework

b. ApplicationContext

c. Tipos de configuración de Beans

ii.ii Bean Factory (a)

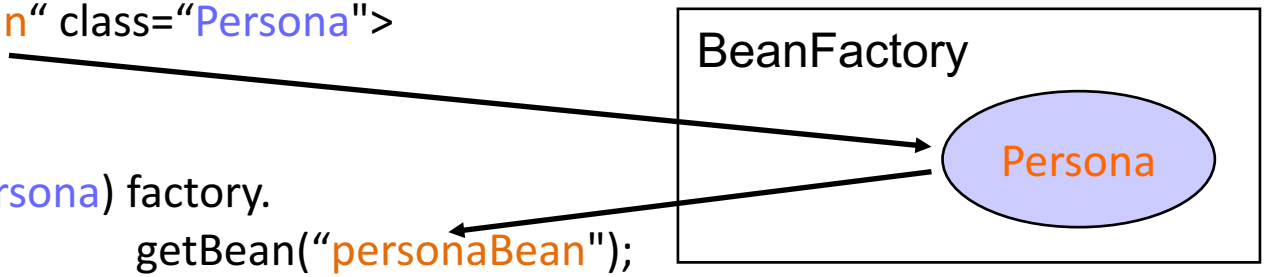
- BeanFactory:
  - Ofrece funcionalidad para instanciar objetos en memoria (beans) y aplicar Inyección de Dependencias (wiring).

ii.ii Bean Factory (b)

```
BeanFactory factory;
factory = new XmlBeanFactory(new
            ClassPathResource("applicationContext.xml"));
```

```
<bean id="personaBean" class="Persona">
</bean>

Persona persona = (Persona) factory.
                                getBean("personaBean");
persona.metodo();
```



ii.ii Bean Factory (c)

Bean 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.xsd">

    <bean id="..." class="...">
        <!– dependencias y configuraciones para este bean -->
    </bean>

</beans>
```

ii.ii Contenedor de IoC

a. BeanFactory

Práctica 2. Hola Mundo Spring Framework

b. ApplicationContext

c. Tipos de configuración de Beans

ii.ii Bean Factory. Práctica 2 (d)

- Práctica 2. Hola Mundo Spring Framework

- Implementar POJO HolaMundo.java
- Configurar <bean id="holaMundoBean">
- Configurar BeanFactory y solicitar holaMundoBean.

ii.ii Contenedor de IoC

a. BeanFactory

Práctica 2. Hola Mundo Spring Framework

b. ApplicationContext

c. Tipos de configuración de Beans

ii.ii ApplicationContext (a)

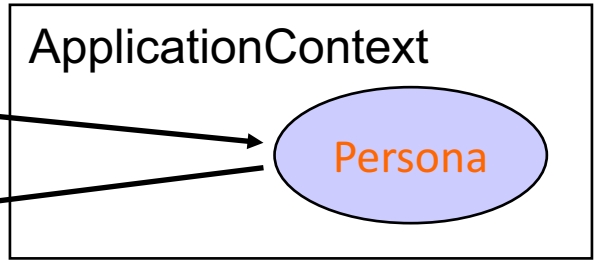
- ApplicationContext:
  - Sub-implementación de BeanFactory, provee mecanismos avanzados como integración con AOP, internacionalización, publicación de eventos y funcionalidades específicas tal como contenedor de aplicaciones web mediante WebApplicationContext.

ii.ii ApplicationContext (b)

```
ApplicationContext applicationContext;
applicationContext = new
        ClassPathXmlApplicationContext("applicationContext.xml");
```

```
<bean id="personaBean" class="Persona">
</bean>

Persona persona = (Persona) applicationContext.
                          getBean("personaBean");
persona.metodo();
```



ii.ii ApplicationContext (c)

- El contenedor ApplicationContext incluye toda la funcionalidad del contenedor BeanFactory, por lo que generalmente se recomienda más el ApplicationContext para aplicaciones empresariales Java EE.

- ApplicationContext provee de toda la funcionalidad de todos los módulos de Spring Framework a excepción del módulo Web (WebApplicationContext).

- Spring registra en automático muchos beans requeridos para la integración de alguna funcionalidad tal como Spring Data o Spring AOP, la implementación ApplicationContext registra todos los beans necesarios mediante BeanPostProcessors requeridos para habilitar estas funcionalidades.

ii.ii ApplicationContext. Práctica 2 (d)

- Práctica 2. Hola Mundo Spring Framework
- Implementar Main utilizando ApplicationContext

ii.ii ApplicationContext (e)

- Contenedor de IoC de Spring

ii.ii ApplicationContext (f)

- Se recomienda utilizar el contenedor BeanFactory para aplicaciones de peso ligero tales como dispositivos móviles o aplicaciones APPLET o, símplemente donde se requiera únicamente un motor de IoC y DI.

- Es recomendado utilizar el ApplicationContext, en lugar de BeanFactory a menos que se tenga una buena razón para no hacerlo

ii.ii Contenedor de IoC

a. BeanFactory

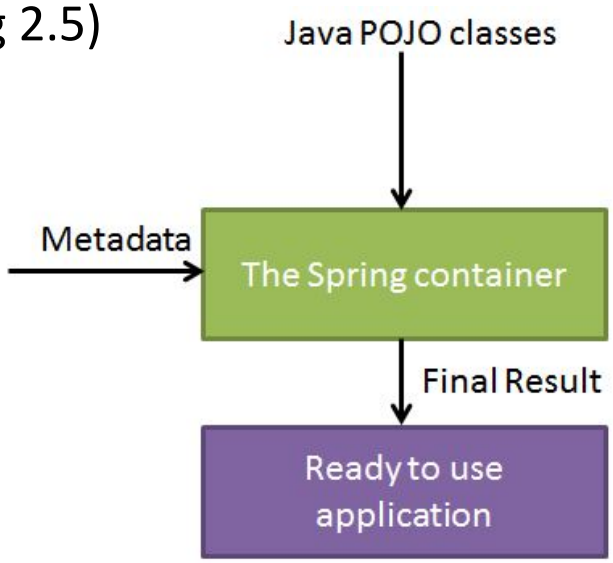
Práctica 2. Hola Mundo Spring Framework

b. ApplicationContext

c. Tipos de configuración de Beans

ii.ii Tipos de configuración de Beans (a)

- La configuración de beans puede ser mediante tres artefactos:
  - XML (Bean Configuration File).
  - Anotaciones Java. (Spring 2.5)
  - Clases Java. (Spring 3.0)



Resumen de la lección

ii.ii Contenedor de IoC

- Conocimos los distintos Contenedores de IoC de Spring Framework.
- Implementamos los contenedores BeanFactory y ApplicationContext.
- Desarrollamos práctica Hola Mundo Spring utilizando ambos contenedores de IoC de Spring Framework.
- Conocimos las distintas formas de configuración de Beans en Spring Framework.

Esta página fue intencionalmente dejada en blanco.

ii.iii Configuración de Beans con XML

Objetivos de la lección

ii.iii Configuración de Beans con XML

- Implementar definición de Beans por medio de configuración XML
- Implementar Inyección de Dependencias por constructor
- Implementar Inyección de Dependencias por setter

ii.iii Configuración de Beans con XML

a. Definición de Beans

b. Inyección de Dependencias

Práctica 3. Inyección de Dependencias Jugadores

Práctica 4. Inyección de Dependencias Movie Finder

Tarea 1. Implementación Notification Service

ii.iii Configuración de Beans con XML (a)

- El contenedor de IoC de Spring esta totalmente desacoplado del formato en que los metadatos de configuración de los beans son definidos.

- Existen tres métodos para proveer la configuración de los Beans al contenedor de IoC y estos son:

- Basado en configuración XML.
- Basado en configuración con Anotaciones.
- Basado en configuración de Clases Java.

ii.iii Configuración de Beans con XML

a. Definición de Beans

b. Inyección de Dependencias

Práctica 3. Inyección de Dependencias Jugadores

Práctica 4. Inyección de Dependencias Movie Finder

Tarea 1. Implementación Notification Service

ii.iii Definición de Beans (a)

- ¿Qué objetos debo configurar como Beans de Spring en mi aplicación?

- Los objetos que forman parte de la columna vertebral de la aplicación; en otras palabras, todos aquellos objetos que dan soporte la aplicación serán Beans de Spring.

- Ejemplo:
  - Cifrador
  - Impresor
  - EmailSender
  - DAOs
  - Validador

ii.iii Definición de Beans (b)

- Un Bean es un objeto que ha sido instanciado, ensamblado (configurado) y es manejado por el contenedor IoC de Spring.

- Una forma de definir un bean es mediante la etiqueta <bean /> en el Bean Configuration File.

- La definición del bean contiene la siguiente información:
- ¿Cómo crear el bean?
- El tipo del Bean (Clase)
- Detalles del ciclo de vida del bean
- Dependencias del bean

ii.iii Definición de Beans (c)

- La definición de beans (básica) contiene las siguientes propiedades.

- **class**: Este atributo es obligatorio y especifica la Clase Java que será utilizada para instanciar el bean.

- **id/name**: Este atributo identifica a un bean de forma única. No es posible que existan dos beans con el mismo id o nombre.

- **scope**: Este atributo especifica el ámbito o alcance de los beans creados a partir de su definición.

ii.iii Definición de Beans (d)

- Ejemplo:

```
<!-- Beans -->

<bean id="..." class="..." />
<bean name="..." class="..." scope="..." />
```

ii.iii Configuración de Beans con XML

a. Definición de Beans

b. Inyección de Dependencias

Práctica 3. Inyección de Dependencias Jugadores

Práctica 4. Inyección de Dependencias Movie Finder

Tarea 1. Implementación Notification Service

ii.iii Inyección de Dependencias (a)

- La Inyección de Dependencias consiste en proveer objetos a una clase en lugar de ser la propia clase quien cree los objetos.

- Para ello se utiliza inyección de dependencias por constructor y por setter.

- **constructor-arg**: Este atributo es utilizado para inyectar dependencias mediante el constructor de la clase definida.

- **property**: Este atributo es utilizado para inyectar dependencias mediante setters de la clase definida.

ii.iii Inyección de Dependencias (b)

- Ejemplo:

```
<bean id="hmb1" class="HolaMundo" scope="singleton" >
    <constructor-arg>
        <value>Hola Mundo Spring !!!</value>
    </constructor-arg>
</bean>

<bean id="hmb2" class="HolaMundo" scope="prototype" >
    <property name="mensaje">
        <value>Hola Mundo 2 Spring !!!</value>
    </property>
</bean>
```

ii.iii Configuración de Beans con XML

a. Definición de Beans

b. Inyección de Dependencias

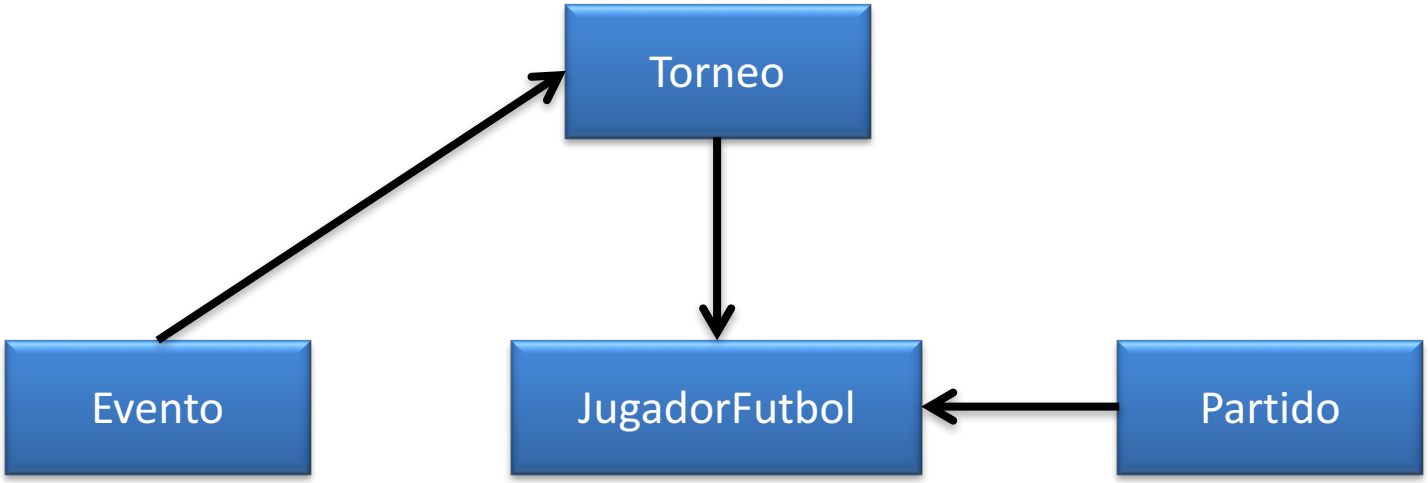
Práctica 3. Inyección de Dependencias Jugadores

Práctica 4. Inyección de Dependencias Movie Finder

Tarea 1. Implementación Notification Service

ii.iii Configuración de Beans con XML. Práctica 3 (a)

- Práctica 3. Inyección de Dependencias Jugadores (DI setter)
- Implementar IoC y DI



ii.iii Configuración de Beans con XML

a. Definición de Beans

b. Inyección de Dependencias

Práctica 3. Inyección de Dependencias Jugadores

Práctica 4. Inyección de Dependencias Movie Finder

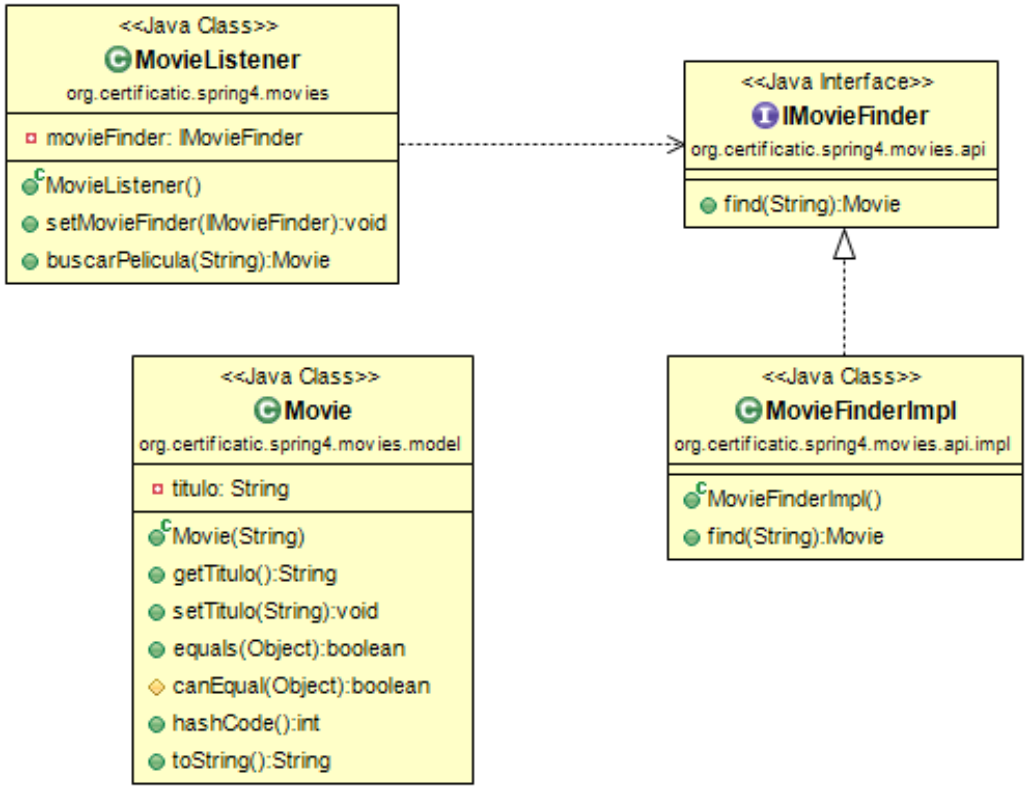
Tarea 1. Implementación Notification Service

ii.iii Configuración de Beans con XML. Práctica 4 (a)

- Práctica 4. Inyección de Dependencias Movie Finder
- Implementar IoC y DI

- Diagrama de clases en el siguiente slide.

ii.iii Configuración de Beans con XML. Práctica 4 (b)



ii.iii Configuración de Beans con XML

a. Definición de Beans

b. Inyección de Dependencias

Práctica 3. Inyección de Dependencias Jugadores

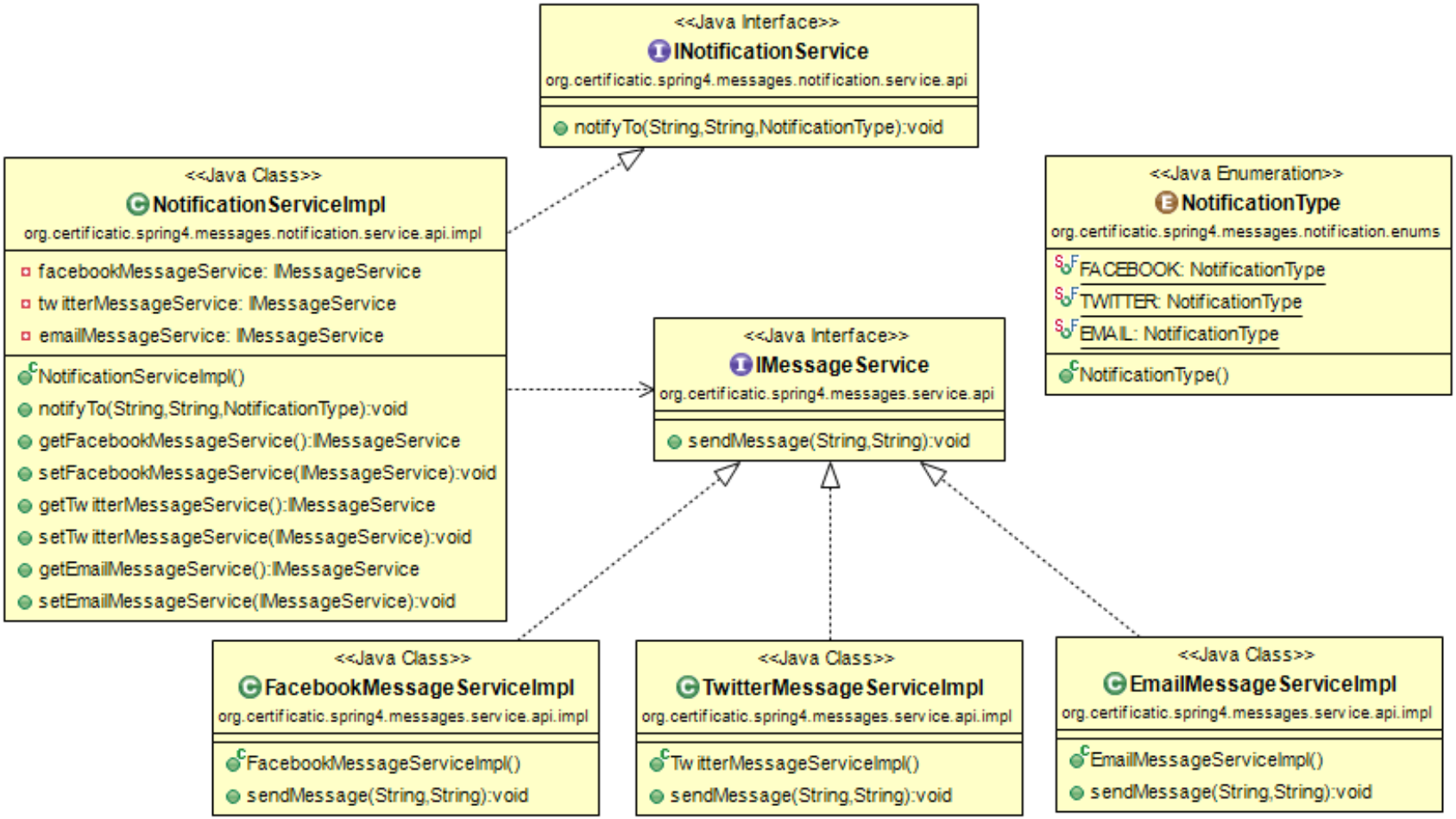
Práctica 4. Inyección de Dependencias Movie Finder

Tarea 1. Implementación Notification Service

ii.iii Configuración de Beans con XML. Tarea 1 (a)

- Tarea 1. Implementación Notification Service
- Implementar IoC y DI

- Diagrama de clases en el siguiente slide.


<<Java Interface>>
INotificationService
org.certificatic.spring4.messages.notification.service.api
notifyTo(String,String,NotificationType):void
<<Java Class>>
NotificationServiceImpl
org.certificatic.spring4.messages.notification.service.api.impl
facebookMessageService: IMessageService
twitterMessageService: IMessageService
emailMessageService: IMessageService
NotificationServiceImpl()
notifyTo(String,String,NotificationType):void
getFacebookMessageService():IMessageService
setFacebookMessageService(IMessageService):void
getTwitterMessageService():IMessageService
setTwitterMessageService(IMessageService):void
getEmailMessageService():IMessageService
setEmailMessageService(IMessageService):void
<<Java Enumeration>>
NotificationType
org.certificatic.spring4.messages.notification.enums
FACEBOOK: NotificationType
TWITTER: NotificationType
EMAIL: NotificationType
NotificationType()
<<Java Interface>>
IMessageService
org.certificatic.spring4.messages.service.api
sendMessage(String,String):void
<<Java Class>>
FacebookMessageServiceImpl
org.certificatic.spring4.messages.service.api.impl
FacebookMessageServiceImpl()
sendMessage(String,String):void
<<Java Class>>
TwitterMessageServiceImpl
org.certificatic.spring4.messages.service.api.impl
TwitterMessageServiceImpl()
sendMessage(String,String):void
<<Java Class>>
EmailMessageServiceImpl
org.certificatic.spring4.messages.service.api.impl
EmailMessageServiceImpl()
sendMessage(String,String):void

Resumen de la lección

ii.iii Configuración de Beans con XML

- Comprendimos que tipo de objetos deberán ser manejados por Spring Framework.
- Conocimos la configuración mínima de Beans de Spring Framework.
- Implementamos Inyección de Dependencias mediante setters.
- Implementamos Inyección de Dependencias mediante constructor.

Esta página fue intencionalmente dejada en blanco.

ii.iv Bean Scopes

Objetivos de la lección

ii.iv Bean Scopes

- Conocer los diferentes tipos de scopes que aplican en el ciclo de vida de Beans.
- Implementar un custom scope.

ii.iv Bean Scopes

a. Singleton
b. Prototype
c. Custom Scope
Práctica 5. Bean Scopes

ii.iv Bean Scopes (a)

- El scope define el ciclo de vida (construcción – destrucción) de un Bean en Spring Framework.

- El scope define “el ámbito” donde será útil un bean.

- Existen principalmente 6 scopes aplicables a beans.

- Por default Spring provee beans singleton.

ii.iv Bean Scopes (b)

- Scopes fundamentales.

- **singleton** (default): Este scope define a un bean como única instancia en el contenedor de IoC.

- **prototype**: Este scope retorna un nuevo bean cada que es solicitado al contenedor de IoC (mediante getBean()).

ii.iv Bean Scopes (c)

- Scopes ambiente web.

- **request**: Este scope retorna un nuevo bean cada que es recibida una petición HTTP. Este scope sólo es válido para aplicaciones web.

- **session**: Este scope retorna un nuevo bean para cada sesión HTTP. Este scope sólo es válido para aplicaciones web.

- ~~**global session**~~: Este scope retorna un nuevo bean para cada sesion global HTTP. Este scope sólo es válido para aplicaciones web portlet.

ii.iv Bean Scopes (d)

- Scopes ambiente web.

- **application**: Este scope crea un nuevo bean asociado al ciclo de vida de un ServletContext. Este scope sólo es válido para aplicaciones web.

- **websocket**: Este scope crea un nuevo bean asociado al ciclo de vida de un WebSocket. Este scope sólo es válido para aplicaciones web.

- **thread**: Este scope crea un nuevo bean asociado a un hilo en ejecución en particular. No se recomienda su uso, Spring lo utiliza a bajo nivel. No está registrado para su uso por default.

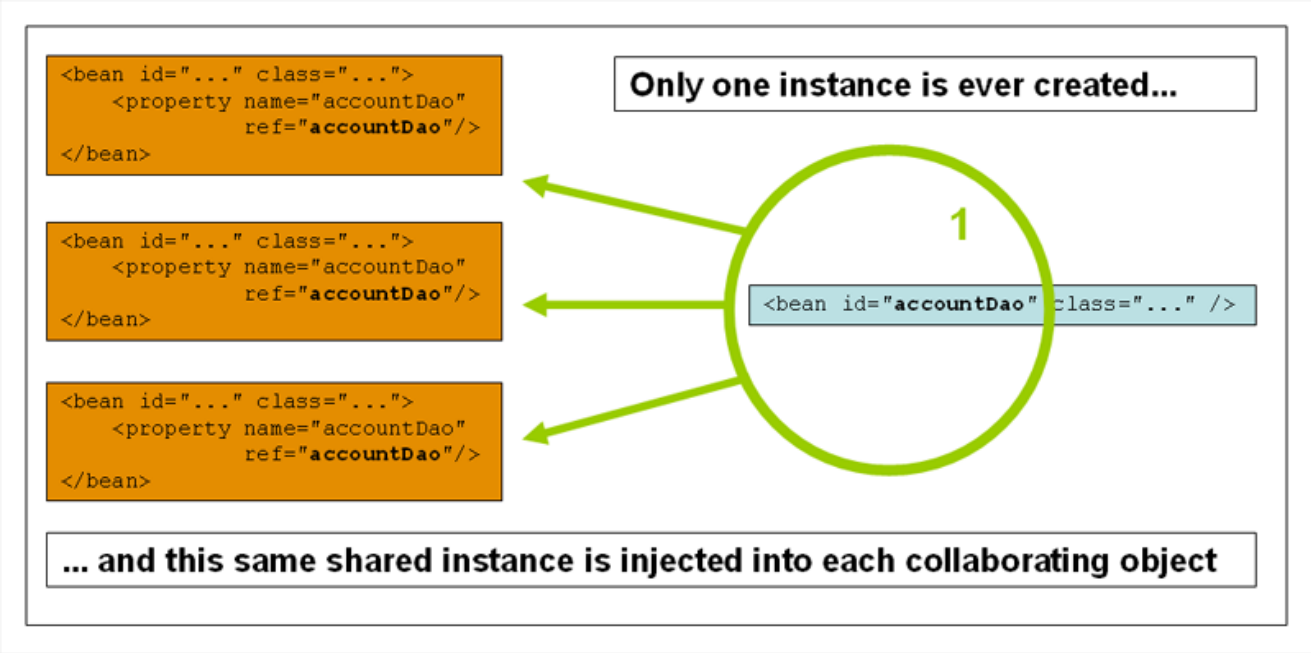
ii.iv Bean Scopes (e)

- Singleton

```
<?xml version="1.0"?>
<!– definicion de bean singleton -->
<bean id="..." class="..." scope="singleton">
    <!– dependencias y configuraciones para este bean -->
</bean>
```

ii.iv Bean Scopes (f)

- Singleton



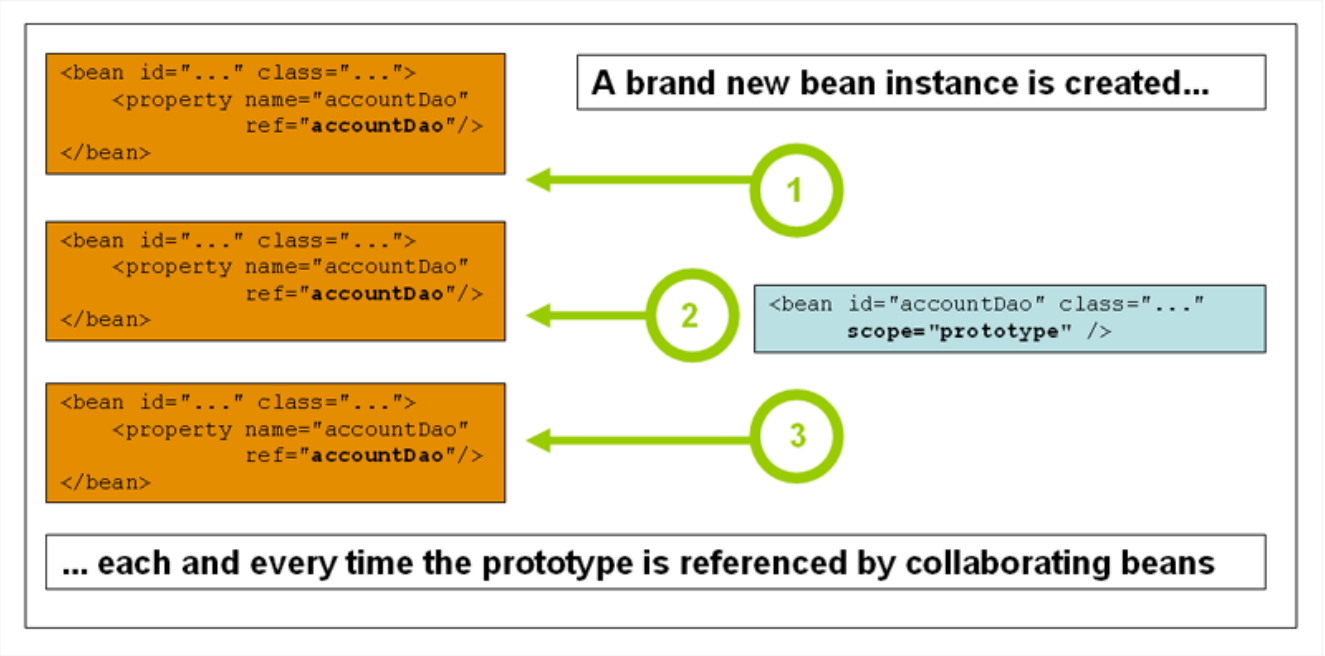
ii.iv Bean Scopes (g)

- Prototype

```
<?xml version="1.0"?>
<!– definicion de bean prototype -->
<bean id="..." class="..." scope=“prototype">
    <!– dependencias y configuraciones para este bean -->
</bean>
```

ii.iv Bean Scopes (h)

- Prototype



ii.iv Bean Scopes

a. Singleton
b. Prototype
c. Custom Scope
Práctica 5. Bean Scopes

ii.iv Bean Scopes (i)

- Custom Scope

- Es posible definir scopes personalizados e incluso sobre-escribir los scopes existentes (no recomendable).

- Un scope personalizado típicamente corresponde a un sistema de gestión y almacenamiento de beans.

- Spring provee un API para habilitar Inyección de Dependencias y búsqueda (look-up) de beans.

ii.iv Bean Scopes (j)

- Custom Scope

- No es común utilizar custom scopes, dependerá de el problema a resolver.

- Por ejemplo:
  - Balanceo de carga de hilos sincronizados.

ii.iv Bean Scopes

a. Singleton
b. Prototype
c. Custom Scope
Práctica 5. Bean Scopes

ii.iv Bean Scopes. Práctica 5 (a)

- Práctica 5. Bean Scopes
- Desarrollar y poner en práctica los scopes singleton y prototype.
- Implementar un custom scope.

Resumen de la lección

ii.iv Bean Scopes

- Conocimos los 5 distintos scopes de Spring Framework.
- Implementamos bean singleton scope.
- Implementamos bean prototype scope.
- Conocimos como implementar custom scopes.
- Desarrollamos un custom scope implementando lógica personalizada para la creación, almacenamiento y gestión de beans.

Esta página fue intencionalmente dejada en blanco.

ii.v Ciclo de vida de Beans

Objetivos de la lección

ii.v Ciclo de vida de Beans

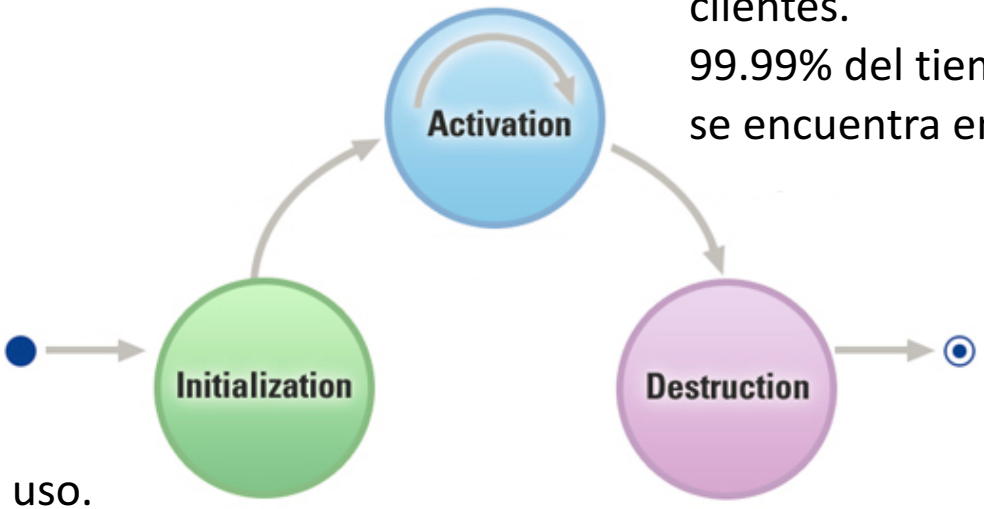
- Conocer de manera general el ciclo de vida de los Beans.
- Configurar la inicialización y destrucción de beans.
- Conocer la inicialización lazy
- Implementar la personalización durante la inicialización y destrucción de beans.
- Implementar inicialización lazy de beans.

ii.v Ciclo de vida de Beans

a. Inicialización y Destrucción
b. Inicialización Lazy
c. Factory Method
Práctica 6. Init – Destroy
Práctica 7. Lazy Beans
Práctica 8. Factory Method
d. Apagando el contenedor de IoC
e. Startup y Shutdown Callbacks
Práctica a. Startup y Shutdown Callbacks
f. Interfaces Aware

ii.v Ciclo de vida de Beans (a)

La aplicación es usada por los clientes.
99.99% del tiempo, la aplicación se encuentra en esta fase.



Se prepara para su uso.
La aplicación configura servicios y recursos.
En este momento la aplicación No es usable.

La aplicación se apaga, o repliega.
Se liberan recursos.
Los objetos son elegibles por el GC.

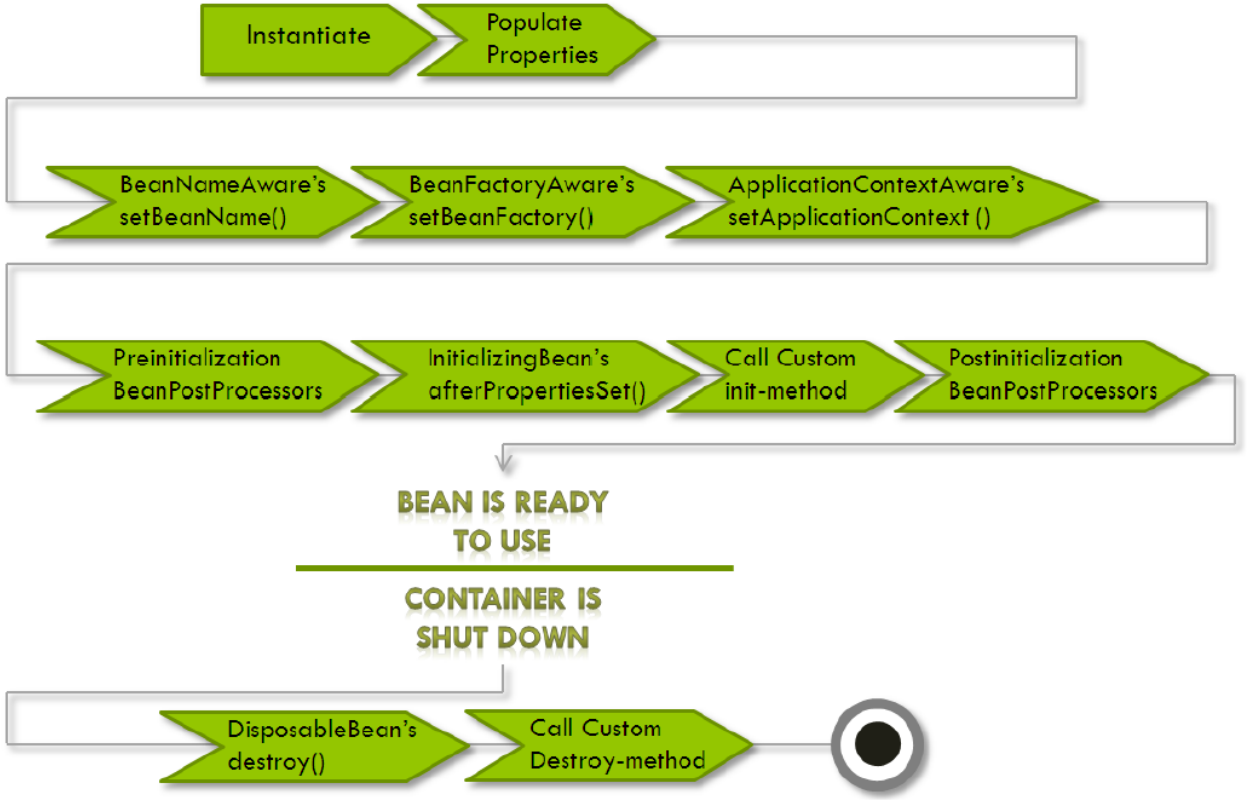
ii.v Ciclo de vida de Beans (b)

- Spring Framework se encarga de gestionar el ciclo de vida de Beans

- Spring Framework permite configurar ciertos eventos que ocurren durante la construcción y destrucción de un bean.

- Técnicas para la personalización de inicialización y destrucción:
  - Implementando InitializingBean y DisposableBean
  - @PostConstruct, @PreDestroy (se verán más adelante)
  - Utilizando init-method y destroy-method en configuración XML

ii.v Ciclo de vida de Beans (c)



ii. Spring Core - ii.v Ciclo de vida de Beans

ii.v Ciclo de vida de Beans (d)

- **init-method.**
- El atributo init-method especifica el nombre del callback a ejecutar sobre el bean inmediatamente después de que sus propiedades fueron 'setteadas'.

- **destroy-method.**
- El atributo destroy-method especifica el nombre del callback a ejecutar sobre el bean inmediatamente antes de que éste sea removido del contenedor de IoC. (sólo singletons)

ii.v Ciclo de vida de Beans (e)

- **init-method y destroy-method.**

```
<?xml version="1.0"?>
<!– definicion de bean init y destroy method-->
<bean id="..." class="..." init-method=“init“ destroy-method=“destroy”>
    <!– dependencias y configuraciones para este bean -->
</bean>
```

ii.v Ciclo de vida de Beans (f)

- **Default init-method y destroy-method.**

```
<?xml version="1.0"?>
<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"
    default-init-method="init" default-destroy-method="destroy">

   <bean id="..." class="...">
       <!– dependencias y configuraciones para este bean -->
   </bean>
</beans>
```

ii.v Ciclo de vida de Beans (g)

- **Implementando InitializingBean.**
- La interfaz **InitializingBean** define:
  - public void afterPropertiesSet() throws Exception();

- **Implementando DisposableBean.**
- La interfaz **DisposableBean** define:
  - public void destroy() throws Exception();

ii.v Ciclo de vida de Beans

a. Inicialización y Destrucción
b. Inicialización Lazy
c. Factory Method
Práctica 6. Init – Destroy
Práctica 7. Lazy Beans
Práctica 8. Factory Method
d. Apagando el contenedor de IoC
e. Startup y Shutdown Callbacks
Práctica a. Startup y Shutdown Callbacks
f. Interfaces Aware

ii.v Ciclo de vida de Beans (h)

- Inicialización Lazy

- **lazy-init.**
- El modo de inicialización perezosa (lazy) de un bean indica al contenedor de IoC crear el bean al momento en que éste es solicitado por primera vez y no al cargar la aplicación.

ii.v Ciclo de vida de Beans (i)

- **lazy-init.**

```
<?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.xsd">

    <!– lazy-initialization -->
    <bean id="..." class="..." lazy-init="true">
        <!– dependencias y configuraciones para este bean -->
    </bean>
</beans>
```

ii.v Ciclo de vida de Beans

a. Inicialización y Destrucción
b. Inicialización Lazy
c. Factory Method
Práctica 6. Init – Destroy
Práctica 7. Lazy Beans
Práctica 8. Factory Method
d. Apagando el contenedor de IoC
e. Startup y Shutdown Callbacks
Práctica a. Startup y Shutdown Callbacks
f. Interfaces Aware

ii.v Ciclo de vida de Beans (j)

- Factory Method

- Define el método callback a utilizar como constructor del bean.
- Puede utilizar **<constructor-arg />** para proveer DI al factory method.
- El método debe ser public static y debe existir en la clase del bean.

ii.v Ciclo de vida de Beans (k)

- Factory Method

- Ejemplo:

```
<bean class=“Auto“ factory-method="constructAuto">
    <constructor-arg ref=“engineBean" />
    <property …  />
</bean>
```

ii.v Ciclo de vida de Beans (I)

- Factory Method
- Ejemplo:

```
@Data
public class Auto {
    private Motor motor;
    private …;

    private Auto(){}

    public static Auto constructAuto(Motor m){
        Auto a = new Auto();
        a.setMotor(m);
        return a;
    }
}
```

ii.v Ciclo de vida de Beans (m)

- Factory Method y Factory Bean

- Ejemplo:

```
<bean class="Auto" factory-bean="autoBuilderBean"
                   factory-method="buildAuto">
    <constructor-arg ref="engineBean" />
    <property …  />
</bean>

<bean id="autoBuilderBean" class="AutoBuilder" />
```

ii.v Ciclo de vida de Beans (n)

- Factory Method y Factory Bean

- Ejemplo:

```
@Data
public class AutoBuilder {

    public Auto buildAuto(Motor m){
        return Auto.constructAuto(m);
    }
}
```

```
@Data
public class Auto {
    private Motor motor;
    private …;

    private Auto(){}

    public static Auto constructAuto(Motor m){
        Auto a = new Auto();
        a.setMotor(m);
        return a;
    }
}
```

ii.v Ciclo de vida de Beans

a. Inicialización y Destrucción
b. Inicialización Lazy
c. Factory Method
Práctica 6. Init – Destroy
Práctica 7. Lazy Beans
Práctica 8. Factory Method
d. Apagando el contenedor de IoC
e. Startup y Shutdown Callbacks
Práctica a. Startup y Shutdown Callbacks
f. Interfaces Aware

ii.v Ciclo de vida de Beans. Práctica 6 (m)

- Práctica 6. Init - Destroy

- Desarrollar y poner en práctica los callback init-method, destroy-method.

- Implementar default-init-method y default-destroy-method.

ii.v Ciclo de vida de Beans

a. Inicialización y Destrucción
b. Inicialización Lazy
c. Factory Method
Práctica 6. Init – Destroy
Práctica 7. Lazy Beans
Práctica 8. Factory Method
d. Apagando el contenedor de IoC
e. Startup y Shutdown Callbacks
Práctica a. Startup y Shutdown Callbacks
f. Interfaces Aware

ii.v Ciclo de vida de Beans. Práctica 7 (n)

- Práctica 7. Lazy Beans

- Desarrollar y poner en práctica la definición de lazy initialization utilizando el callback lazy-init.

- Implementar Lazy initialization utilizando DI.

ii.v Ciclo de vida de Beans

a. Inicialización y Destrucción
b. Inicialización Lazy
c. Factory Method
Práctica 6. Init – Destroy
Práctica 7. Lazy Beans
Práctica 8. Factory Method
d. Apagando el contenedor de IoC
e. Startup y Shutdown Callbacks
Práctica a. Startup y Shutdown Callbacks
f. Interfaces Aware

ii.v Ciclo de vida de Beans. Práctica 8 (o)

- Práctica 8. Factory-method

- Desarrollar y poner en práctica la definición de fabrica abstracta de beans utilizando el callback factory-method.

- Implementar factory-method utilizando DI por constructor.

ii.v Ciclo de vida de Beans

a. Inicialización y Destrucción
b. Inicialización Lazy
c. Factory Method
Práctica 6. Init – Destroy
Práctica 7. Lazy Beans
Práctica 8. Factory Method
d. Apagando el contenedor de IoC
e. Startup y Shutdown Callbacks
Práctica a. Startup y Shutdown Callbacks
f. Interfaces Aware

ii.v Ciclo de vida de Beans (p)

- Apagando el contenedor de IoC

- Es necesario llamar al método registerShutdownHook() del ApplicationContext (ConfigurableApplicationContext) para apagar el contenedor correctamente.

- Únicamente aplicable a entornos de aplicaciones NO “web aware” de Spring.

- También es posible apagar el contenedor mandando a llamar al método close() del ApplicationContext.

ii.v Ciclo de vida de Beans (q)

- Apagando el contenedor de IoC

```
applicationContext = new ClassPathXmlApplicationContext(
                                    "spring/practicaA/lifecycle-application-context.xml");

…

((AbstractApplicationContext) applicationContext).registerShutdownHook();
ó
((AbstractApplicationContext) applicationContext).close();
```

ii.v Ciclo de vida de Beans

a. Inicialización y Destrucción
b. Inicialización Lazy
c. Factory Method
Práctica 6. Init – Destroy
Práctica 7. Lazy Beans
Práctica 8. Factory Method
d. Apagando el contenedor de IoC
e. Startup y Shutdown Callbacks
Práctica a. Startup y Shutdown Callbacks
f. Interfaces Aware

ii.v Ciclo de vida de Beans (r)

- Startup y Shutdown Callbacks

- Es posible implementar la interface **Lifecycle** sobre un bean que desee poder implementar mayor control sobre su ciclo de vida.

```
public interface Lifecycle {
        void start();
        void stop();
        boolean isRunning();
}
```

ii.v Ciclo de vida de Beans (s)

- Startup y Shutdown Callbacks

- La interface **Lifecycle** no ejecuta el método start() cuando el ApplicationContext es construido (listo para usar), debido a que no se ejecuta el evento “start” del ApplicationContext.

- El ApplicationContext ejecuta el evento “refresh” al momento de ejecutar el “auto-startup” del contexto es decir, no ejecuta el evento “start”.

ii.v Ciclo de vida de Beans (t)

- Startup y Shutdown Callbacks

- El evento ”refresh” siempre es ejecutado implícitamente al momento de la creación del ApplicationContext.

- El evento “start” debe de mandarse a llamar explícitamente sobre el ApplicationContext (ConfigurableApplicationContext).

ii.v Ciclo de vida de Beans (u)

- Startup y Shutdown Callbacks

- Para tener un mejor control sobre el arranque y detención del ApplicationContext se sugiere implementar la interface **SmartLifecycle** sub-interface de **Lifecycle**, la cuál ejecuta el método “start” del bean al momento de que el ApplicationContext “auto-inicia” (auto-startup).

ii.v Ciclo de vida de Beans (v)

- SmartLifecycle

```
public interface SmartLifecycle extends Lifecycle, Phased {
        int DEFAULT_PHASE = Integer.MAX_VALUE;
        default boolean isAutoStartup() {
                return true;
        }
        default void stop(Runnable callback) {
                stop();
                callback.run();
        }
        default int getPhase() {
                return DEFAULT_PHASE;
        }
}
```

ii.v Ciclo de vida de Beans

a. Inicialización y Destrucción
b. Inicialización Lazy
c. Factory Method
Práctica 6. Init – Destroy
Práctica 7. Lazy Beans
Práctica 8. Factory Method
d. Apagando el contenedor de IoC
e. Startup y Shutdown Callbacks
Práctica a. Startup y Shutdown Callbacks
f. Interfaces Aware

ii.v Ciclo de vida de Beans. Práctica a (w)

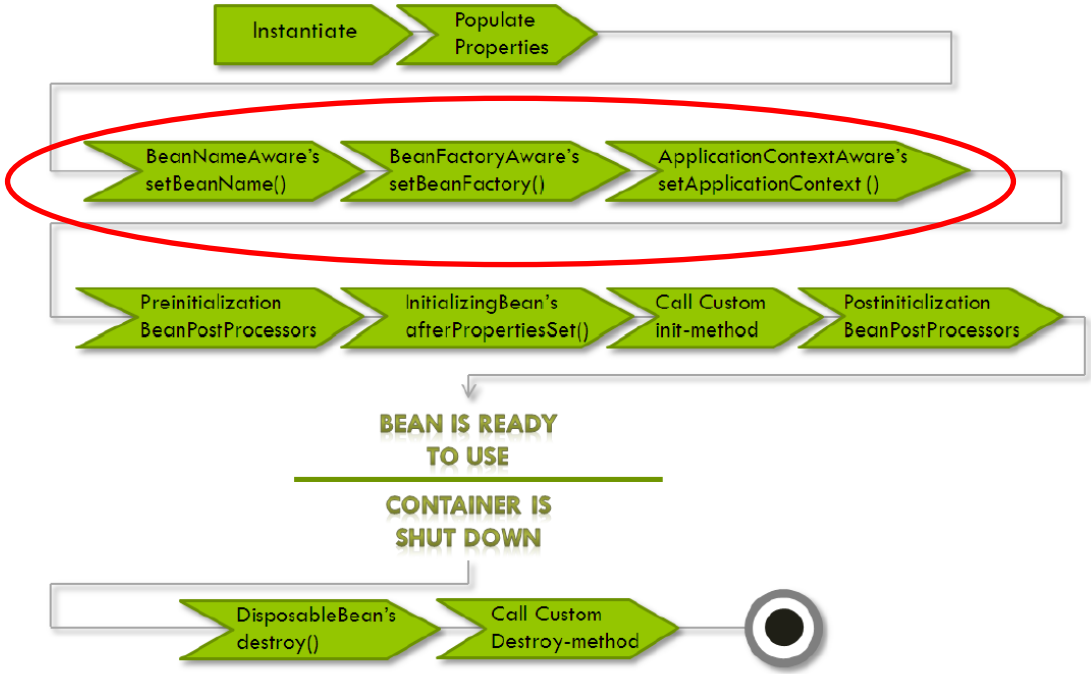
- Práctica a. Startup y Shutdown Callbacks
- Implementar la creación y cierre del ApplicationContext.
- Implementar las interfaces Lifecycle y SmartLifecycle.
- Comprender la diferencia entre ambas interfaces.

ii.v Ciclo de vida de Beans

a. Inicialización y Destrucción
b. Inicialización Lazy
c. Factory Method
Práctica 6. Init – Destroy
Práctica 7. Lazy Beans
Práctica 8. Factory Method
d. Apagando el contenedor de IoC
e. Startup y Shutdown Callbacks
Práctica a. Startup y Shutdown Callbacks
f. Interfaces Aware

ii.v Ciclo de vida de Beans (x)

- Interfaces Aware.



ii. Spring Core - ii.v Ciclo de vida de Beans

ii.v Ciclo de vida de Beans (y)

- Interfaces Aware.

- Existen múltiples interfaces “aware” (tener el conocimiento o percepción de una situación o de un hecho, ser consiente de algo en particular).

- ApplicationContextAware
- BeanNameAware
- ApplicationEventPublisherAware
- ResourceLoaderAware
- entre otras

https://docs.spring.io/spring/docs/5.1.4.RELEASE/spring-framework-reference/core.html#aware-list

ii.v Ciclo de vida de Beans (z)

- ApplicationContextAware

- Interface utilizada para inyectar el ApplicationContext a un bean específico. Posiblemente tenga un requerimiento particular para ejecutar el método getBean() explícitamente.

```
public interface ApplicationContextAware extends Aware {

    void setApplicationContext(ApplicationContext applicationContext)
                                                    throws BeansException;

}
```

Resumen de la lección (a)

ii.v Ciclo de vida de Beans

- Comprendimos las principales fases de construcción e inicialización de beans .
- Implementamos inicialización de beans utilizando configuración por XML init-method así como implementando la interface InitializingBean.
- Implementamos destrucción de beans utilizando configuración por XML destroy-method así como implementando la interface DisposableBean.

Resumen de la lección (b)

ii.v Ciclo de vida de Beans

- Realizamos configuración de beans bajo demanda (lazy-init).
- Implementamos DI de beans utilizando patrón abstract factory mediante configuración XML utilizando factory-method callback.
- Implementamos las interfaces Lifecycle y SmartLifecycle.
- Conocimos que son y para qué sirven las interfaces “Aware”.

Esta página fue intencionalmente dejada en blanco.

ii.vi Definición heredada de Beans

Objetivos de la lección

ii.vi Definición heredada de Beans

- Implementar herencia de configuración de beans.
- Desarrollar plantillas (Templates) de configuración de beans reusables para configurar beans distintos.

ii.vi Definición heredada de Beans

Práctica 9. Bean Templates

ii.vi Definición heredada de Beans (a)

- Es posible heredar la configuración de un bean padre a un bean hijo, mediante el atributo <bean **parent="…"** />

- La herencia de los metadatos de configuración de beans no implica herencia de clases, aunque el concepto es el mismo.

- Es posible re-definir la configuración heredada de un bean.

- La definición heredada de Beans permite reutilizar la configuración de un bean común.

ii.vi Definición heredada de Beans (b)

- Ejemplo:

```
<bean id="conexionProduccionBean" class="...">
   <property name="baseDatos" value="sistemaBD"/>
   <property name="usuario" value="root"/>
   <property name="contrasenia" value="123abc"/>
</bean>

<bean id="conexionPruebasBean" class="..."
                                parent="conexionProduccionBean">
   <property name="baseDatos " value="sistemaTestBD"/>
   <property name="modoDebug " value="true"/>
</bean>
```

ii.vi Definición heredada de Beans (c)

- También es posible utilizar una configuración de bean abstracta, es decir, definir una configuración de bean sin especificar su tipo (clase).

```
<bean id="conexionBeanTemplate" abstract="true">
   <property name="usuario" value="root"/>
   <property name="contrasenia" value="123abc"/>
   <property name="modoDebug" value="false"/>
</bean>

<bean id="conexionProduccionBean" class="..."
                                  parent="conexionBeanTemplate">
   <property name="baseDatos" value="sistemaBD"/>
</bean>
```

ii.vi Definición heredada de Beans (d)

- Práctica 9. Bean Templates

- Implementar herencia de configuración de beans mediante:
  - Herencia de configuración de beans de un tipo de bean especifico
  - Herencia de configuración de beans mediante definición de bean abstracto.

Resumen de la lección

ii.vi Definición heredada de Beans

- Implementamos herencia de configuración de beans utilizando la configuración de un bean especifico.
- Implementamos herencia de configuración de beans mediante la configuración de un bean abstracto.
- Realizamos distintas implementaciones de un bean a partir de la aplicación de una configuración de bean template.

Esta página fue intencionalmente dejada en blanco.

ii.vii Puntos de extensión del Contenedor

Objetivos de la lección

ii.vii Puntos de extensión del Contenedor

- Implementar Bean Post Processors para personalizar la inicialización de beans.
- Implementar Bean Factory Post Processor para modificar la definición de beans.
- Conocer la forma de ordenamiento de procesamiento de Bean Post Processors y Bean Factory Post Processors.

ii.vii Puntos de extensión del Contenedor

a. Puntos de extensión del Contenedor
b. BeanPostProcessor
c. BeanFactoryPostProcessor
d. FactoryBean
Práctica 10. BeanPostProcessor
Práctica b. BeanFactoryPostProcessor

ii.vii Puntos de extensión del Contenedor (a)

- Spring implementa puntos de extensión del contenedor para facilitar la integración de nuevas funcionalidades al framework. Principio Open-Close.

- BeanPostProcessor

- BeanFactoryPostProcessor

- FactoryBean

ii.vii Puntos de extensión del Contenedor

a. Puntos de extensión del Contenedor
b. BeanPostProcessor
c. BeanFactoryPostProcessor
d. FactoryBean
Práctica 10. BeanPostProcessor
Práctica b. BeanFactoryPostProcessor

ii.vii BeanPostProcessor (a)

- Típicamente un desarrollador no requiere de heredar y/o re-implementar el ApplicationContext para extenderlo, especializarlo y/o agregar nuevas funcionalidades requeridas para el aplicativo desarrollado.

- Spring provee de un mecanismo tipo “plug-and-play” para extender el Framework mediante la interface BeanPostProcessor.

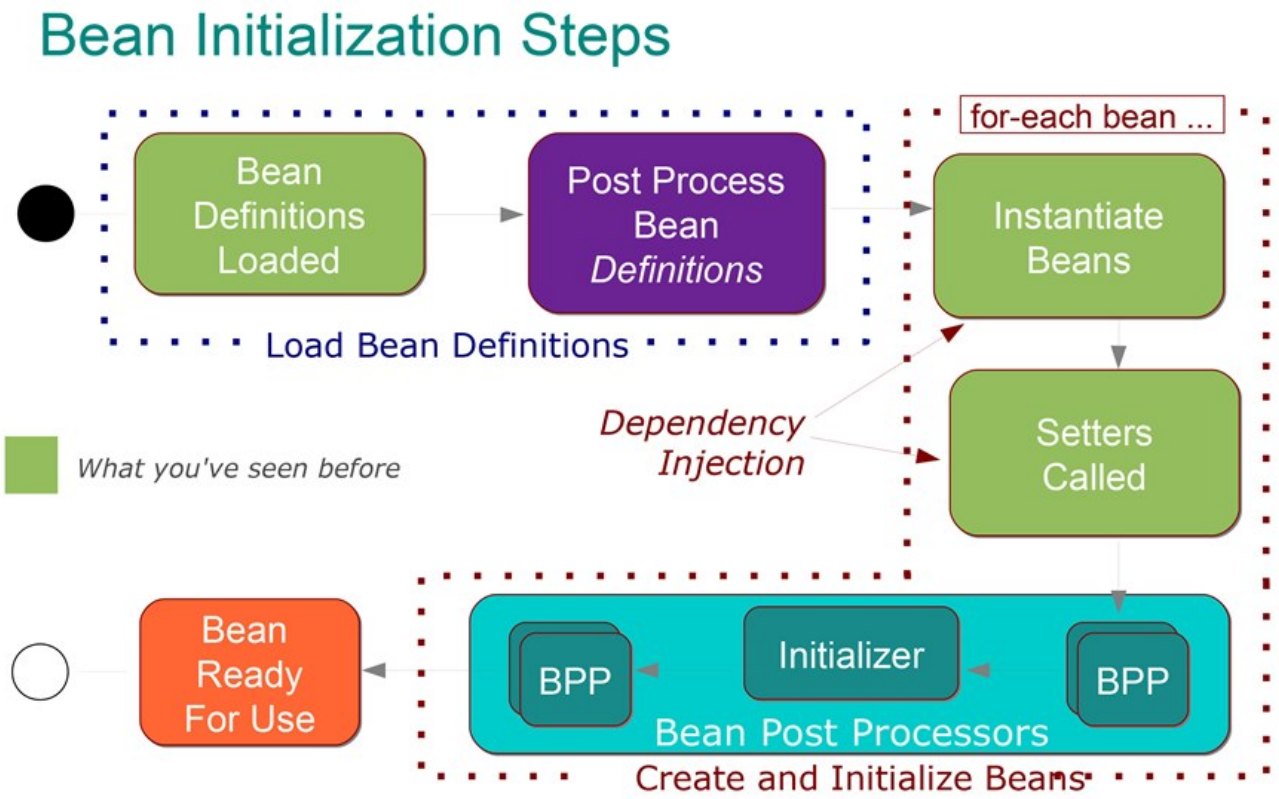
ii.vii BeanPostProcessor (b)

- Los Bean Post Processors son una interfaz que define métodos callback que permite personalizar la inicialización y resolución de Beans.

- Es posible implementar uno o mas Bean Post Processors.

- Se puede configurar el orden de ejecución de los Bean Post Processors implementando la interface Ordered.

ii.vii BeanPostProcessor (c)



ii.vii BeanPostProcessor (d)

- Implementar interface BeanPostProcessor

Object **postProcessBeforeInitialization** (Object bean, String beanName)
throws BeansException;

Object **postProcessAfterInitialization** (Object bean, String beanName)
throws BeansException;

ii.vii BeanPostProcessor (e)

- Implementar interface Ordered

```
int getOrder();
```

ii.vii Puntos de extensión del Contenedor

a. Puntos de extensión del Contenedor
b. BeanPostProcessor
c. BeanFactoryPostProcessor
d. FactoryBean
Práctica 10. BeanPostProcessor
Práctica b. BeanFactoryPostProcessor

ii.vii BeanFactoryPostProcessor (a)

- Los Bean Factory Post Processor son una interfaz que define métodos callback que permite personalizar la definición/configuración de Beans al momento de que dicha definición es cargada.

- Los Bean Factory Post Processor actuan sobre la configuración, metadatos, de la definición de beans ya sea que mediante configuración por XML, @Anotaciónes o Java Config.

- Se puede configurar el orden de ejecución de los Bean Factory Post Processors implementando la interface Ordered.

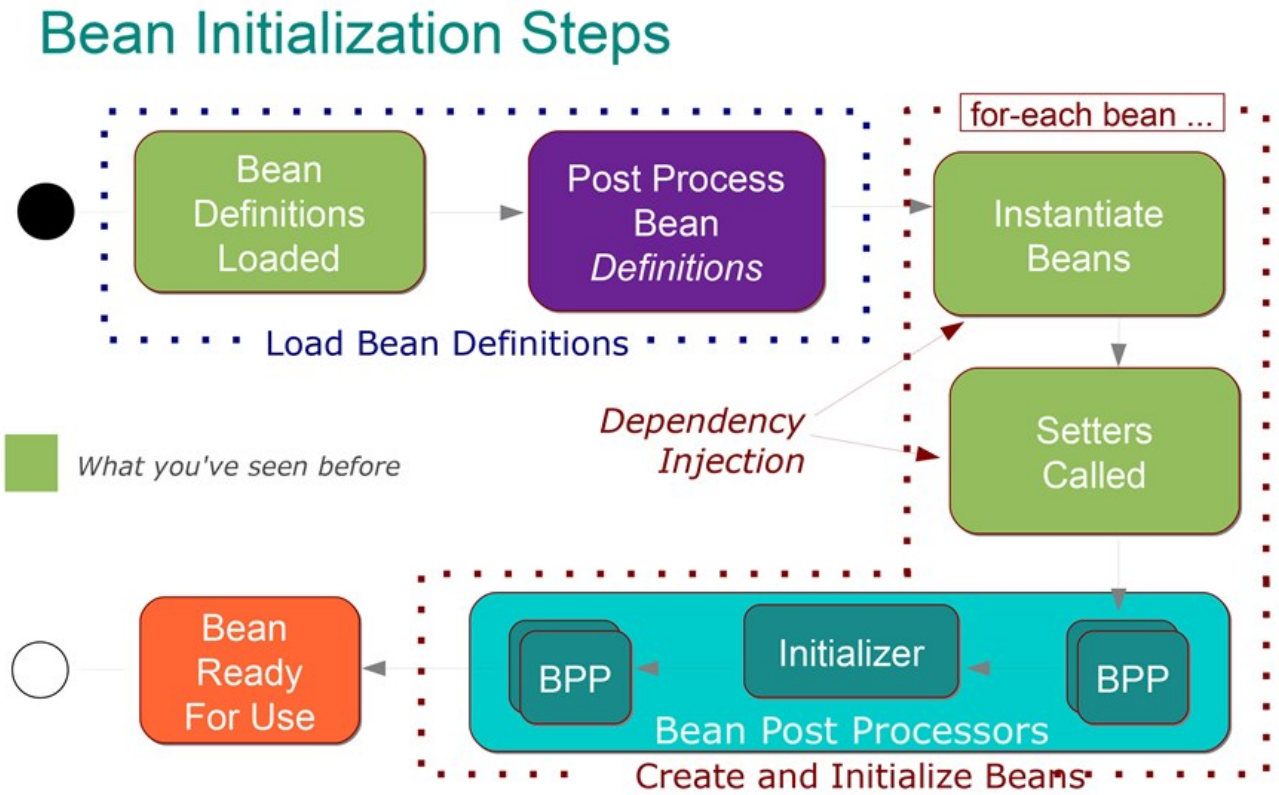
ii.vii BeanFactoryPostProcessor (b)

- Básicamente los Bean Factory Post Processor tienen la posibilidad de leer la configuración / definición de beans, cambiarla, suprimirla o simplemente definir nuevos beans no definidos previamente en los archivos de configuración de beans (XML, @Anotaciones o Java Config).

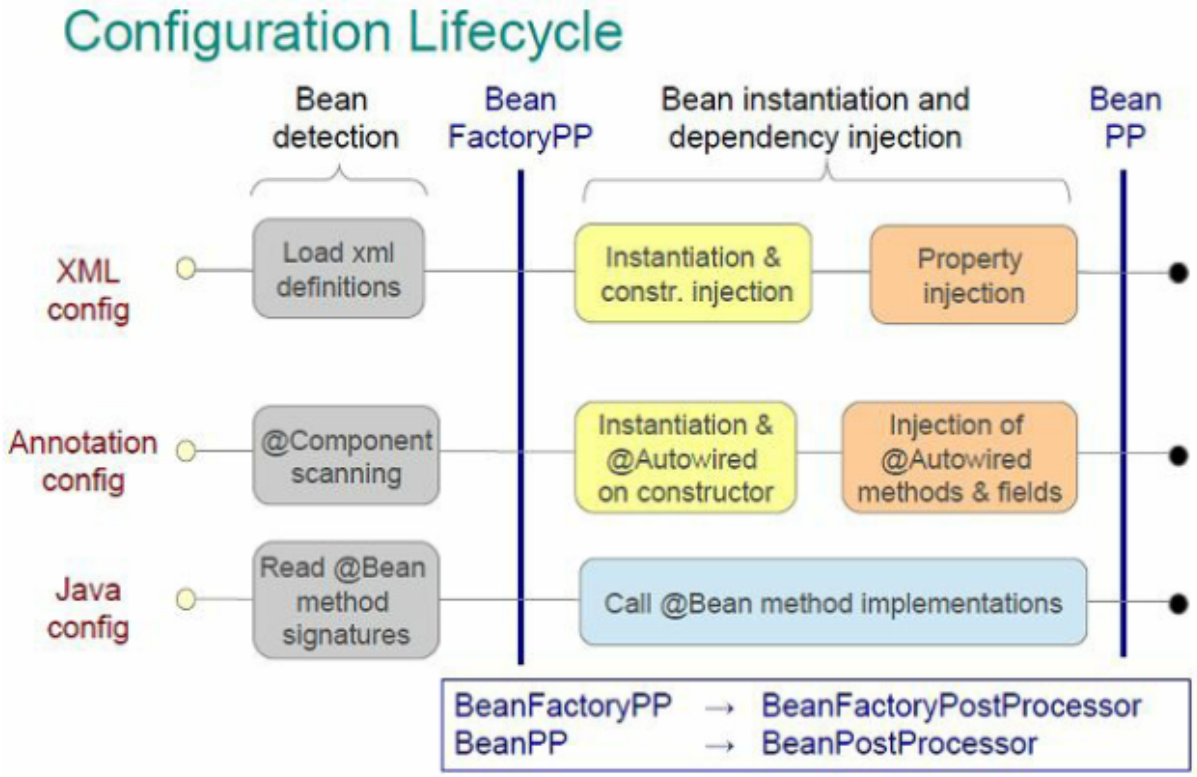
- No se recomienda su implementación para aplicativos de usuario final, es peligroso.

- Ampliamente utilizado para extender el framework de Spring por otros proyectos: Spring Data, Spring Security, Spring MVC, etc.

ii.vii BeanFactoryPostProcessor (c)



ii.vii BeanFactoryPostProcessor (d)



ii.vii BeanFactoryPostProcessor (e)

- Implementar interface BeanFactoryPostProcessor

void **postProcessBeanFactory**(
ConfigurableListableBeanFactory beanFactory)
throws BeansException;

ii.vii Puntos de extensión del Contenedor

a. Puntos de extensión del Contenedor
b. BeanPostProcessor
c. BeanFactoryPostProcessor
d. FactoryBean
Práctica 10. BeanPostProcessor
Práctica b. BeanFactoryPostProcessor

ii.vii FactoryBean (a)

- La interface FactoryBean es otro mecanismo de extensión del framework útil para los casos donde la lógica de instanciación del objeto es compleja y es la propia clase del bean la encargada de ejecutar dicha construcción.

- FactoryBean es ampliamente utilizada por el framework mismo y sus subproyectos.

- Para aplicativos de usuario final, se recomienda utilizar factory-bean y factory-method por configuración XML, menos verboso y no acoplado a infraestructura de Spring.

ii.vii FactoryBean (b)

- Interface FactoryBean

```
public interface FactoryBean<T> {

    T getObject() throws Exception;
    Class<?> getObjectType();
    boolean isSingleton();

}
```

ii.vii Puntos de extensión del Contenedor

a. Puntos de extensión del Contenedor
b. BeanPostProcessor
c. BeanFactoryPostProcessor
d. FactoryBean
Práctica 10. BeanPostProcessor
Práctica b. BeanFactoryPostProcessor

ii.vii Puntos de extensión del Contenedor. Práctica 10 (a)

- Práctica 10. BeanPostProcessor

- Implementar inicialización de bean personalizado mediante BeanPostProcessor
- Implementar múltiples BeanPostProcessor.

ii.vii Puntos de extensión del Contenedor

a. Puntos de extensión del Contenedor
b. BeanPostProcessor
c. BeanFactoryPostProcessor
d. FactoryBean
Práctica 10. BeanPostProcessor
Práctica b. BeanFactoryPostProcessor

ii.vii Puntos de extensión del Contenedor. Práctica b (a)

- Práctica b. BeanFactoryPostProcessor

- Implementar definición / configuración de bean personalizado mediante BeanFactoryPostProcessor
- Implementar múltiples BeanFactoryPostProcessor.

Resumen de la lección

ii.vii Puntos de extensión del Contenedor

- Comprendimos más en detalle el ciclo de vida de los beans, específicamente la fase de inicialización y definición.
- Implementamos BeanPostProcessor, los cuales proveen un mecanismo de personalización en la inicialización de beans.
- Aplicamos ordenamiento de ejecución a los BeanPostProcessor.
- Implementamos BeanFactoryPostProcessor, los cuales proveen un mecanismo de personalización en la definición / configuración de beans.

Esta página fue intencionalmente dejada en blanco.

ii.viii Definición de Beans internos

Objetivos de la lección

ii.viii Definición de Beans internos

- Implementar definición de Beans internos.
- Conocer el beneficio de definir Beans internos.

ii.viii Definición de Beans internos

Práctica 11. Beans Internos

ii.viii Definición de Beans internos (a)

- Los Beans internos son particulares del Bean donde son definidos.
- No pueden ser referenciados de forma externa (ctx.getBean(nombre))
- No requieren definir atributo id o name.

ii.viii Definición de Beans internos (b)

- Ejemplo:

```
<bean id="beanExterno" class="...">
    <property name="atributo">
        <bean class="java.lang.String">
            <constructor-arg value="unString" />
        </bean>
    </property>
</bean>
```

ii.viii Definición de Beans internos. Práctica 11 (c)

- Práctica 11. Beans Internos

- Implementar Bean Persona
- Inyectar Bean Interno de tipo String (atributo nombre)

Resumen de la lección

ii.viii Definición de Beans internos

- Aprendimos a definir Beans internos.
- Comprendimos el beneficio de utilizar Beans internos.
- Conocimos la forma de crear Beans propios de objetos del API Java, específicamente clase String.

Esta página fue intencionalmente dejada en blanco.

ii.ix Inyección de Colecciones y Arreglos

Objetivos de la lección

ii.ix Inyección de Colecciones y Arreglos

- Implementar inyección de colecciones del tipo:
  - List
  - Set
  - Map
  - Properties
- Implementar inyección de arreglos de objetos.

ii.ix Inyección de Colecciones y Arreglos

Práctica 12. Inyección de Colecciones y Arreglos

ii.ix Inyección de Colecciones y Arreglos (a)

- Inyección de dependencias mediante **value** permite inyectar valores escalares.
- Inyección de dependencias mediante **ref** permite inyectar beans definidos por id o nombre.
- Es posible inyectar colecciones como:
  - List<E>
  - Set<E>
  - Map<K, V>
  - Properties

ii.ix Inyección de Colecciones y Arreglos (b)

- Inyección de List mediante **<list>**, nos permite inyectar una lista de valores que permite duplicados.

- Inyección de Set mediante **<set>**, nos permite inyectar un conjunto de valores que no permite duplicados.

- Es posible inyectar cualquier implementación de java.util.Collection mediante <list> y <set>.

ii.ix Inyección de Colecciones y Arreglos (c)

- Inyección de Map mediante **<map>**, nos permite inyectar una colección (mapa) de objetos del tipo llave – valor, donde la llave y el valor pueden ser objetos de cualquier tipo.

- Inyección de Properties mediante **<props>**, nos permite inyectar una colección (properties) de objetos del tipo llave – valor, donde la llave y el valor ser objetos de tipo String.

ii.ix Inyección de Colecciones y Arreglos (d)

- Inyección de Arreglos mediante **<array>**, aunque es posible usar **<list>** y **<set>** indistintamente.

ii.ix Inyección de Colecciones y Arreglos (e)

- Ejemplo, bean Directorio:

```
public class Directorio {
    private List<Direccion> direcciones;
    private Set<Telefono> telefonos;
    private Map<Integer, String> numerosDeEmergencia;
    private Properties familiares;

    private Integer[] numeros;
    private String[] textos;
    private Persona[] personas;
}
```

ii.ix Inyección de Colecciones y Arreglos (f)

- Ejemplo List:

```
<property name="direcciones">
    <list>
        <ref bean="ivanDireccionBean" />
        <ref bean="lauraDireccionBean" />
        <bean class=“..” … />
    </list>
</property>
```

ii.ix Inyección de Colecciones y Arreglos (g)

- Ejemplo Set:

```
<property name="telefonos">
    <set>
        <ref bean="ivanDireccionBean" />
        <ref bean="lauraDireccionBean" />
        <bean class=".." ... />
    </set>
</property>
```

ii.ix Inyección de Colecciones y Arreglos (h)

- Ejemplo Map:

```
<property name="numerosDeEmergencia">
    <map>
        <entry key="2" value="Policia: 040" />
        <entry key="3">
            <value>Protección civil: 040</value>
        </entry>
    </map>
</property>
```

ii.ix Inyección de Colecciones y Arreglos (i)

- Ejemplo Properties:

```
<property name="familiares">
    <props>
        <prop key="papa">Julio Regalado</prop>
        <prop key="mama">Karla Amezcua</prop>
    </props>
</property>
```

ii.ix Inyección de Colecciones y Arreglos (j)

- Ejemplo Arrays:

```
<property name="numeros">
    <array>
        <value>1</value>
        <ref bean="numero2" />
        <bean class="java.lang.Integer">
            <constructor-arg value="3" />
        </bean>
    </array>
</property>
```

ii.ix Inyección de Colecciones y Arreglos. Práctica 12 (k)

- Práctica 12. Inyección de Colecciones y Arreglos

- Implementar Bean Directorio
- Inyectar Colecciones List, Set, Map, Properties
- Inyectar Arrays de Integer, String y Persona

Resumen de la lección

ii.ix Inyección de Colecciones y Arreglos

- Reforzamos la aplicación de inyección de dependencias utilizando value y ref.
- Implementamos inyección de dependencias de colecciones usando <list>, <set>, <map>, <props>
- Realizamos inyección de arreglos mediante <array>
- Realizamos inyección de dependencias de colecciones utilizando Beans internos.

Esta página fue intencionalmente dejada en blanco.

ii.x Namespace p, c y util

Objetivos de la lección

ii.x Namespace p, c y util

- Implementar los namespaces p y c para agilizar la definición de Beans.
- Conocer las implicaciones de utilizar los namespaces p y c.
- Conocer las principales características del namespace util.
- Simplificar la creación de Beans de tipo List, Set, Map y Properties.

ii.x Namespace p, c y util

Tarea 2. Ejemplo namespace p, c y util

ii.x Namespace p, c y util (a)

- Namespace p

- Provee de una alternativa para definir inyección de dependencias por setter sin utilizar el tag **<property>** mediante la inyección directamente sobre el tag **<bean>** utilizando un atributo especial.

- Reduce la cantidad de código XML en el Bean Configuration file.

ii.x Namespace p, c y util (b)

- Ejemplo Namespace p:

```
<bean class="Persona">
    <property name="nombre" value="Ivan García" />
    <property name="auto" ref="miAutoBean" />
</bean>
```

- Utilizando namespace p:

```
<bean class="Persona"
        p:nombre="Ivan García" p:auto-ref="miAutoBean" />
```

ii.x Namespace p, c y util (c)

- Namespace p

- Implicaciones:
  - Propenso a error.
  - No todos los IDEs proveen funcionalidad “autocomplete”.

ii.x Namespace p, c y util (d)

- Namespace c

- Provee de una alternativa para definir inyección de dependencias por constructor sin utilizar el tag **<constructor-arg>** mediante la inyección directamente sobre el tag **<bean>** utilizando un atributo especial.

- Reduce la cantidad de código XML en el Bean Configuration file.

ii.x Namespace p, c y util (e)

- Ejemplo Namespace c:

```
<bean class="Persona">
    <constructor-arg value="Ivan García" />
    <constructor-arg ref="miAutoBean" />
</bean>
```

- Utilizando namespace c:

```
<bean class="Persona"
        c:nombre="Ivan García" c:auto-ref="miAutoBean" />
```

ii.x Namespace p, c y util (f)

- Namespace c

- Implicaciones:
  - Para el raro caso donde la compilación del código se realice sin información de "*debbug*" (sin opción –g), namespace c no podrá resolver por nombre de argumento.
  - Es posible utilizar el namespace c basado en índice del argumento en lugar de nombre del argumento.

```
<bean id="beanOne" class="x.y.ThingOne"
      c:_0-ref="beanTwo" c:_1-ref="beanThree" c:_2="something@somewhere.com" />
```

ii.x Namespace p, c y util (g)

- Namespace util

- Provee de utilerías para el manejo de colecciones y constantes.

- Permite crear una colección, como un bean; recordar que <list> define un conjunto de elementos a inyectar, no define un Bean de tipo List.

ii.x Namespace p, c y util (h)

- Namespace util:constant

- Permite inyectar constantes a atributos de un Bean especifico.

```
<bean class="Circulo" p:radio="5.55">
    <property name="pi">
        <util:constant static-field="java.lang.Math.PI"/>
    </property>
</bean>
```

ii.x Namespace p, c y util (i)

- Namespace util:list

- Permite crear Beans de tipo List.
- Facilita la creación de listas para ser inyectadas en distintos Beans.

```
<bean class="Agenda">
    <property name="notas" ref="misNotasBean" />
</bean>

<util:list id="misNotasBean" list-class="java.util.ArrayList">
    <value>Una Nota</value>
    <value>Otra Nota</value>
</util:list>
```

ii.x Namespace p, c y util (j)

- Namespace util:set

- Permite crear Beans de tipo Set.
- Facilita la creación de conjuntos para ser inyectados en distintos Beans.

```
<bean class="Agenda">
    <property name="autosFamilia" ref="misAutosBean" />
</bean>

<util:set id=" misAutosBean" set-class="java.util.HashSet">
    <ref bean="autoBean" />
    <ref bean="autoBean" />
</util:set>
```

ii.x Namespace p, c y util (k)

- Namespace util:map

- Permite crear Beans de tipo Map.
- Facilita la creación de mapas para ser inyectadas en distintos Beans.

```
<bean class="Agenda">
    <property name="numeros" ref="numerosBean" />
</bean>

<util:map id="numerosBean" map-class="java.util.HashMap">
    <entry key="uno" value="1" />
    <entry key="dos" value="2" />
</util:map>
```

ii.x Namespace p, c y util (I)

- Namespace util:properties

- Permite cargar en el contexto un archivo de propiedades.

```
<bean class="Agenda">
    <property name="properties" ref="misPropertiesBean" />
</bean>

<util:properties id="misPropertiesBean"
    location="classpath:/spring/tarea2/properties/*.properties">
    <prop key="programmer.name">Paula Sofía</prop>
</util:properties>
```

ii.x Namespace p, c y util. Tarea 2 (m)

- Tarea 2. Ejemplo namespace p, c y util (a)

- Implementar Beans de tipo <util:list>, <util:set>, <util:map>, <util:properties>
- Inyectarlos en un Bean Agenda.
- Imprimir los valores.

ii.x Namespace p, c y util. Tarea 2 (n)

- Tarea 2. Ejemplo namespace p, c y util (b)

- Propuesta, Bean Agenda:

```
@Data
public class Agenda {
    private List<String> notas;
    private Set<Auto> autosFamilia;
    private Map<String, Integer> numeros;
    private Properties properties;
}
```

Bonus. Valores vacíos, nulos y depends-on (a)

- Al momento de aplicar inyección de dependencias, Spring tratará de inyectar un valor específico. Si se requiere inyectar un “null”, utilizar la etiqueta <null />

```
<bean class="ExampleBean">
    <property name="email" value=""/>
</bean>
```

```
exampleBean.setEmail("");
```

```
<bean class="ExampleBean">
    <property name="email">
        <null/>
    </property>
</bean>
```

```
exampleBean.setEmail(null);
```

Bonus. Valores vacíos, nulos y depends-on (b)

- Típicamente la forma de resolución de beans de Spring se genera mediante inyección de dependencias explicita, de esta forma el contenedor ”sabrá” que beans construir primero.

- Existen momentos en que la dependencia entre beans es menos directa, por ejemplo cuando es necesario ejecutar un inicializador estático (factory-method estatico).

Bonus. Valores vacíos, nulos y depends-on (c)

- Para definir que un bean, requiere o tiene dependencia de otro bean de forma indirecta o menos directa es posible utilizar el atributo **depends-on** del elemento <beans> o la anotación **@DependsOn**.

```
<bean id="beanOne" class="ExampleBean" depends-on="manager,accountDao">
    <property name="manager" ref="manager" />
</bean>

<bean id="manager" class="ManagerBean" />
<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />
```

Resumen de la lección

ii.x Namespace p, c y util

- Conocimos la utilidad de los namespaces p, c y util.
- Aprendimos como crear Beans de tipo List, Set, Map y Properties.
- Comprendimos como realizar inyección de constantes en propiedades de Beans.

Esta página fue intencionalmente dejada en blanco.

ii.xi Autowiring

Objetivos de la lección

ii.xi Autowiring

- Conocer el concepto de Autowiring (auto-hilado).
- Implementar los diferentes tipos de Autowiring para Inyectar Dependencias.

ii.xi Autowiring

a. byName
b. byType
c. constructor
Práctica 13. Autowiring

ii.xi Autowiring (a)

- Hasta el momento se han **definido** beans utilizando el tag **<bean>.**

- La **inyección de dependencias** se ha realizado utilizando setter o constructor mediante el tag **<property>** o **<constructor-arg>** correspondientemente.

- El contenedor de IoC de Spring permite auto-hilar (auto inyectar) las dependencias entre beans colaboradores sin utilizar inyección de dependencias por setter o constructor explícitamente.

ii.xi Autowiring (b)

- El autowiring (auto-hilado) disminuye significativamente la cantidad de código XML en los Bean Configuration File.

- Autowiring es una buena práctica debido a que por lo regular los beans de Spring son singletons (únicas instancias) y Spring puede inyectar dicho bean en otros beans donde sea colaborador o dependencia.

ii.xi Autowiring (c)

- Existen distintos modos de autowiring que le especifican al contenedor IoC de Spring como inyectar las dependencias de los beans.

- Es necesario utilizar el atributo **autowire** del tag **<bean>** para especificar la definición de autowiring para un bean en especifico.

ii.xi Autowiring (d)

- Tipos de Autowiring

  - **no (default):** No autowiring, inyección mediante <property> o <constructor-arg> utilizando los atributos value o ref.

  - **byName:** Autowiring por **nombre de la propiedad**. El contenedor de IoC de Spring tratará de auto hilar las dependencias de un bean comparando el **nombre de la propiedad** contra el **nombre** de algún bean que califique para ser inyectado.

ii.xi Autowiring (e)

- Tipos de Autowiring

  - **byType:** Autowiring por **tipo (clase) de la propiedad**. El contenedor de IoC de Spring tratará de auto hilar las dependencias de un bean comparando el **tipo (clase) de la propiedad** contra el **tipo (clase)** de algún bean que califique para ser inyectado.

  - **constructor:** Similar al autowiring **byType** pero aplicado a los argumentos del constructor del bean.

ii.xi Autowiring (f)

- byName

```
<bean id=“pedroBean" class="com.Persona" autowire="byName">
    <property name=“nombre" value=“Pedro Hernández"/>
    <!– La propiedad direccion se auto hilará byName -->
</bean>

<bean id=“direccion" class="com.Direccion“ />
```

```
public class Persona {
    private String nombre;
    private Direccion direccion;
}

public class Direccion {
    private String calle;
    private String numero;
}
```


ii.xi Autowiring (g)

- byType

```
<bean id=“pedroBean" class="com.Persona" autowire="byType">
    <property name=“nombre" value=“Pedro Hernández"/>
    <!– La propiedad direccion se auto hilará byName -->
</bean>

<bean id=“direccionBean" class="com.Direccion“ />
```

```
public class Persona {
    private String nombre;
    private Direccion direccion;
}

public class Direccion {
    private String calle;
    private String numero;
}
```

ii.xi Autowiring (h)

- constructor

```
<bean id=“pedroBean" class="com.Persona" autowire=“constructor">
    <property name=“nombre" value=“Pedro Hernández"/>
    <!– La propiedad direccion se auto hilará byName -->
</bean>

<bean class="com.Direccion“ />
```

```
public class Persona {
    private String nombre;
    private Direccion direccion;
    public Persona(Direccion d){
        this.direccion=d;
    }
}
public class Direccion {
    …
}
```

ii.xi Autowiring (i)

- Limitaciones

- El autowiring funciona mejor cuando es aplicado consistentemente a través de toda la aplicación. Si el autowiring no se utiliza de forma general para todos los beans, resultará confuso para algunos desarrolladores utilizar autowiring sólo para la definición de algunos beans.

- Utilizar <property> o <constructor-arg> sobre-escribe el autowiring.

- No es posible auto-hilar primitivos, Strings, Classes y arreglos (de los tipos anteriores).

ii.xi Autowiring (j)

- Limitaciones

- El autowiring es menos exacto que la inyección de dependencias explicita.

- Es posible caer en ambigüedades cuando existen más de un único bean en el contenedor de IoC de Spring.

- Cambiar la implementación a inyectar por autowiring requiere re-compilación.

ii.xi Autowiring

a. byName
b. byType
c. constructor

Práctica 13. Autowiring

ii.xi Autowiring. Práctica 13 (a)

- Práctica 13. Autowiring

- Implementar Inyección de Dependencias por autowiring byType, byName y por constructor.

- Comprender más a fondo la diferencia entre beans singleton y prototype.

Resumen de la lección

ii.xi Autowiring

- Conocimos los distintos tipos de autowiring que provee Spring.
- Comprobamos que el autowiring es un mecanismo muy útil para inyectar dependencias a beans colaboradores.
- Analizamos las limitantes e implicaciones de implementar autowiring.

Esta página fue intencionalmente dejada en blanco.

Trabajo de Integración 1. Convertidor número letra configuración XML

Objetivos de la lección

Trabajo de Integración 1. Convertidor número letra configuración XML

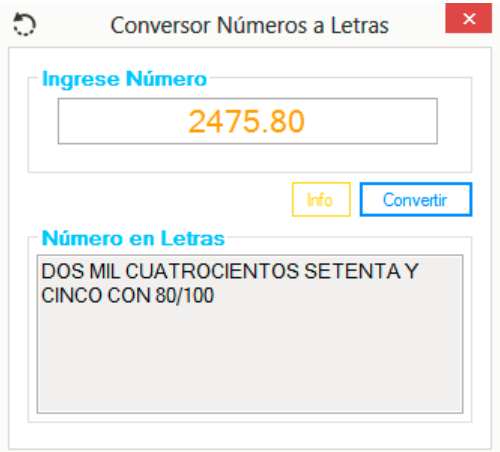
- Conocer el concepto de Autowiring (auto-hilado).
- Implementar los diferentes tipos de Autowiring para Inyectar Dependencias.

Trabajo de Integración 1. Convertidor número letra configuración XML

Práctica 14. Convertidor número letra configuración XML

Trabajo de Integración 1. Convertidor número letra configuración XML (a)

- Desarrollar un componente convertidor número a texto utilizando Inyección de Dependencias.



ii. Spring Core – T.I. 1. Convertidor número letra configuración XML

Trabajo de Integración 1. Convertidor número letra configuración XML

Práctica 14. Convertidor número letra configuración XML

Trabajo de Integración 1. Práctica 14. (a)

- Práctica 14. Convertidor número letra configuración XML

- Analizar el código referido al componente NumericalConverter.
- Realizar la configuración de beans por XML que se adecue más a las dependencias de cada componente.

```
abr 08, 2016 10:56:09 PM org.springframework.context.support.ClassPathXmlApplicationContext
INFORMACIÓN: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@2
abr 08, 2016 10:56:09 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadB
INFORMACIÓN: Loading XML bean definitions from class path resource [spring/practica13/conver
numero: 5465158.32
cinco millones cuatrocientos sesenta y cinco mil ciento cincuenta y ocho pesos 32/100
abr 08, 2016 10:56:10 PM org.springframework.context.support.ClassPathXmlApplicationContext
INFORMACIÓN: Closing org.springframework.context.support.ClassPathXmlApplicationContext@21b8
```

ii. Spring Core – T.I. 1. Convertidor número letra configuración XML

Trabajo de Integración 1. Práctica 14. (b)

- Práctica 14. Convertidor número letra configuración XML

- Analizar el diseño y debatir las capacidades o estrategias para implementar el componente de forma multi-idioma.

```
abr 08, 2016 11:48:56 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadB
INFORMACIÓN: Loading XML bean definitions from class path resource [beansConvertidorNumeroLe
abr 08, 2016 11:48:56 PM org.springframework.beans.factory.support.DefaultListableBeanFactor
INFORMACIÓN: Pre-instantiating singletons in org.springframework.beans.factory.support.Defau
numero: 730424
SEVEN-HUNDRED  THIRTY  THOUSAND FOUR-HUNDRED  TWENTY   FOUR  DOLLARS 00/100
abr 08, 2016 11:48:56 PM org.springframework.context.support.AbstractApplicationContext doCl
INFORMACIÓN: Closing org.springframework.context.support.ClassPathXmlApplicationContext@d5f6
abr 08, 2016 11:48:56 PM org.springframework.beans.factory.support.DefaultSingletonBeanRegis
INFORMACIÓN: Destroying singletons in org.springframework.beans.factory.support.DefaultListA
```

ii. Spring Core – T.I. 1. Convertidor número letra configuración XML

Resumen de la lección

Trabajo de Integración 1. Convertidor número letra configuración XML (a)

- Explotamos las características aprendidas hasta el momento referidas a la configuración de beans con Spring framework.
- Realizamos la configuración de un bean complejo mediante configuración por XML.
- Analizamos las oportunidades que presenta el diseño del componente para poder extenderlo multi-idioma.

Esta página fue intencionalmente dejada en blanco.

ii.xii Configuración con @Anotaciones

Objetivos de la lección

ii.xii Configuración con @Anotaciones

- Conocer las principales funcionalidad del namespace context.
- Implementar configuración de beans mediante anotaciones de Spring Framework.

ii.xii Configuración con @Anotaciones

a. Namespace context

b. @Required, @Autowired y @Qualifier

Práctica 15. @Required, @Autowired y @Qualifier

ii.xii Configuración con @Anotaciones (a)

- Namespace context

- El namespace context se encarga de crear configuraciones relacionadas a beans de soporte que utiliza Spring tras bambalinas (código de tubería).

ii.xii Configuración con @Anotaciones (b)

- <context:property-placeholder/>
- Activa el reemplazo de placehlders (${*nombreProperty*}) en configuración de beans.
- Similar a configurar un bean PropertyPlaceholderConfigurer.
- (Código de tubería)

ii.xii Configuración con @Anotaciones (c)

- <context:annotation-config/>

- Activa la configuración de beans por medio de anotaciones.

- ~~@Required~~, @Autowired y @Qualifier así como las referidas al JSR 250 y JSR 330.

- Similar a configurar **AutowiredAnnotationBeanPostProcessor**, **CommonAnnotationBeanPostProcessor**, **PersistenceAnnotationBeanPostProcessor** y **RequiredAnnotationBeanPostProcessor**. (Código de tubería)

ii.xii Configuración con @Anotaciones (d)

- <context:component-scan/>
- Activa el escaneo de definiciones de beans por medio de anotaciones.
- @Component, @Service, @Repository, @Controller, @RestController y @Scope
- Evita definir beans mediante configuración XML.

ii.xii Configuración con @Anotaciones (e)

- <context:load-time-weaver/>
- Activa el hilado (weaving) de aspectos en tiempo de carga de las clases.
- No cubierto en este curso.

ii.xii Configuración con @Anotaciones (f)

- <context:spring-configured/>
- Activa inyección de dependencias a objetos cuyo ciclo de vida no está gestionado por Spring.
- Se aplica mediante aspectos.
- No cubierto en este curso.

ii.xii Configuración con @Anotaciones (g)

- <context:mbean-export/>

- Exprota beans como Managed Beans útiles para la gestión de beans en tiempo real mediante JMX.

- No cubierto en este curso.

ii.xii Configuración con @Anotaciones

a. Namespace context

b. @Required, @Autowired y @Qualifier

Práctica 15. @Required, @Autowired y @Qualifier

ii.xii Configuración con @Anotaciones (a)

- @Required, @Autowired, @Qualifier
- Habilitados por la instrucción <context:annotation-config />
- Permiten inyectar dependencias a través de @Anotaciones
- Es posible combinar configuración de beans por medio de XML y por @Anotaciones (XML sobreescribe @Anotaciones).

ii.xii Configuración con @Anotaciones (b)

~~@Required~~

- Aplica a métodos setter de propiedades de beans.

- Utiliza inyección por setter e indica que la dependencia es requerida y debe ser inyectada en tiempo de configuración. Se lanza una excepción si la propiedad no es satisfecha.

- Deprecada a partir de Spring Framework 5.1 a favor de utilizar **@Autowired** por **constructor** para dependencias requeridas.

ii.xii Configuración con @Anotaciones (c)

~~@Required~~

```
<beans>
    <context:annotation-config/>
    <bean id="auto" class="Auto">
        <property name="modelo" value="2011" />
    </bean>
</beans>
```

```
public class Auto{
    private String modelo;

    @Required
    public void setModelo(String modelo){
        this.modelo = modelo;
    }
}
```

ii.xii Configuración con @Anotaciones (d)

- @Autowired

- Provee mayor control para inyectar dependencias sobre beans e indica que cierta dependencia debe ser satisfecha mediante inyección de dependencias.

- Aplica a:
  - Métodos setter
  - Constructores
  - Propiedades
  - Otros métodos

ii.xii Configuración con @Anotaciones (e)

- @Autowired sobre métodos setter
- Similar a utilizar <property>, permite prescindir de esta instrucción.
- Utiliza inyección de dependencias por tipo (byType).

ii.xii Configuración con @Anotaciones (f)

- @Autowired sobre métodos setter

```
<beans>
    <context:annotation-config/>
    <bean class="Auto">
        <!– Sin Inyección de Dependencias -->
    </bean>
    <bean class="Motor">
        <!– Configuración del Bean -->
    </bean>
</beans>
```

```
public class Auto{
    private Motor motor;

    @Autowired
    public void setMotor(Motor motor){
        this.motor = motor;
    }
}
```

ii.xii Configuración con @Anotaciones (g)

- @Autowired sobre propiedades

- Permite inmutabilidad pues elimina métodos setter; a su vez es similar a utilizar <property>, permite prescindir de esta instrucción.

- Utiliza inyección de dependencias por tipo (byType).

ii.xii Configuración con @Anotaciones (h)

- @Autowired sobre propiedades

```
<beans>
    <context:annotation-config/>
    <bean class="Auto">
        <!– Sin Inyección de Dependencias -->
    </bean>
    <bean class="Motor">
        <!– Configuración del Bean -->
    </bean>
</beans>
```

```
public class Auto{
    @Autowired
    private Motor motor;
}
```

ii.xii Configuración con @Anotaciones (i)

- @Autowired sobre constructores
- Similar a utilizar <constructor-arg>, permite prescindir de esta instrucción.
- Aplica sobre todos los argumentos del constructor.
- Utiliza inyección de dependencias por tipo (byType).

ii.xii Configuración con @Anotaciones (j)

- @Autowired sobre constructores

```
<beans>
    <context:annotation-config/>
    <bean class="Auto">
        <!– Sin Inyección de Dependencias -->
    </bean>
    <bean class="Motor">
        <!– Configuración del Bean -->
    </bean>
</beans>
```

```
public class Auto{
    private Motor motor;

    @Autowired
    public Auto(Motor m){
        this.motor = m;
    }
}
```

ii.xii Configuración con @Anotaciones (k)

- @Autowired sobre otros métodos
- Similar a implementar un init-method.
- Aplica sobre todos los argumentos del método.
- Utiliza inyección de dependencias por tipo (byType).
- A partir de Spring 4.3 **@Autowired** sobre el constructor no es necesario sólo si la clase tiene un único constructor. Si la clase tiene más de un constructor **@Autowired** es requerido en alguno de ellos.

ii.xii Configuración con @Anotaciones (I)

- @Autowired sobre otros métodos

```
<beans>
    <context:annotation-config/>
    <bean class="Auto">
        <!– Sin Inyección de Dependencias -->
    </bean>
    <bean class="Motor">
        <!– Configuración del Bean -->
    </bean>
</beans>
```

```
public class Auto{
    private Motor motor;

    @Autowired
    public init(Motor m){
        this.motor = m;
    }
}
```

ii.xii Configuración con @Anotaciones (m)

- @Autowired(required=false)
- Por default @Autowired es requerido (required=true)
- Es posible permitir inyección de dependencias insatisfechas.
- Por default cuando @Autowired no satisface una dependencia, se lanza una excepción.

ii.xii Configuración con @Anotaciones (n)

- @Autowired(required=false)

```
<beans>
    <context:annotation-config/>
    <bean class="Auto">
        <!– Sin Inyección de Dependencias -->
    </bean>
</beans>
```

```
public class Auto{
    @Autowired(required=false)
    private Motor motor;
}
```

ii.xii Configuración con @Anotaciones (ñ)

- @Autowired(required=false)

- Alternativamente es posible utilizar la clase Optional (Java 8) para definir que una dependencia es no requerida.

- A partir de Spring 5.x es posible utilizar la anotación @Nullable (JSR 305) en contra parte de definir el atributo @Autowired(required=false).

ii.xii Configuración con @Anotaciones (o)

- @Autowired sobre dependencias de interfaces reconocidas.

- Es posible inyectar dependencias de interfaces (y sub-interfaces) bien reconocidas por el contenedor de IoC de Spring, directamente sobre propiedades, tales como:
  - BeanFactory
  - ApplicationContext
  - Environment
  - ResourceLoader
  - ApplicationEventPublisher y,
  - MessageSource

ii.xii Configuración con @Anotaciones (p)

- @Autowired sobre dependencias de interfaces reconocidas.

```
public class MovieRecommender {

    @Autowired
    private ApplicationContext context;

    public MovieRecommender() { … }
}
```

ii.xii Configuración con @Anotaciones (q)

- @Qualifier

- Por default @Autowired implementa inyección por tipo (byType).

- ¿Qué sucederá si existe más de un bean definido del mismo tipo?
  - Ambigüedad

- @Qualifier se utiliza en conjunto con @Autowired y se usa para evitar ambigüedad mediante la especificación del nombre o id del bean requerido a inyectar.

ii.xii Configuración con @Anotaciones (r)

- @Qualifier

```
<beans>
    <context:annotation-config/>
    <bean class="Auto">
        <!– Sin Inyección de Dependencias -->
    </bean>
    <bean id="motorV6" class="Motor">
        <!– Configuración del Bean -->
    </bean>
    <bean id="motorV8" class="Motor">
        <!– Configuración del Bean -->
    </bean>
</beans>
```

```
public class Auto{
    @Autowired
    @Qualifier("motorV8")
    private Motor motor;
}
```

ii.xii Configuración con @Anotaciones (s)

- @Qualifier es una meta-anotación, es decir, permite generar anotaciones ”qualifiers” ”type-safe”. Se revisará más adelante.

```
public class MovieRecommender {
        @Autowired
        @Genre("Action")
        private MovieCatalog actionCatalog;

        private MovieCatalog comedyCatalog;

        @Autowired
        public void setComedyCatalog(@Genre("Comedy") MovieCatalog comedyCatalog) {
                this.comedyCatalog = comedyCatalog;
        }
}
```

```
@Target({ElementType.FIELD,
ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Genre {
        String value();
}
```

ii.xii Configuración con @Anotaciones

a. Namespace context

b. @Required, @Autowired y @Qualifier

Práctica 15. @Required, @Autowired y @Qualifier

ii.xii Configuración con @Anotaciones. Práctica 15 (a)

- Práctica 15. @Required, @Autowired y @Qualifier

- Implementar Inyección de Dependencias por medio configuración mediante anotaciones Spring.

Resumen de la lección

ii.xii Configuración con @Anotaciones

- Conocimos a grandes rasgos la utilidad del namespace context.
- Comprendimos cómo habilitar configuración de beans por medio de anotaciones.
- Utilizamos configuración de beans mediante @Required, @Autowired y @Qualifier.

Esta página fue intencionalmente dejada en blanco.

ii.xiii Anotaciones JSR 250

Objetivos de la lección

ii.xiii Anotaciones JSR 250

- Conocer las principales anotaciones del JSR 250 referentes a Inyección de Dependencias.
- Implementar Inyección de Dependencias mediante @Anotaciones de Java EE con Spring Framework.

ii.xiii Anotaciones JSR 250

a. @Resource, @PostConstruct y @PreDestroy

Práctica 16. @Resource, @PostConstruct y @PreDestroy

ii.xiii Anotaciones JSR 250 (a)

- Para personalizar la configuración de beans es posible implementar las interfaces:

- InitializingBean
  - afterPropertiesSet()

- DisposableBean
  - destroy()

- Configuración acoplada a Interfaces de Spring.

ii.xiii Anotaciones JSR 250 (b)

- Anotaciones JSR 250 (Common Annotations).
- Se recomienda utilizar las anotaciones JSR 250 para gestionar las fases de construcción y destrucción de beans.
- @PostConstruct
- @PreDestroy
- Soportadas a partir de Spring 2.5

ii.xiii Anotaciones JSR 250 (c)

- @PostConstruct

- Aplicable a métodos.
- Define el método callback a ejecutar después de construir el bean.

- @PreDestroy

- Aplicable a métodos.
- Define el método callback a ejecutar inmediatamente antes de que el bean sea removido del contenedor y éste sea replegado.

ii.xiii Anotaciones JSR 250 (d)

- @PostConstruct y @PreDestroy

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

   <bean class="Auto">
   </bean>

   <bean id="motorV6" class="Motor">
   </bean>

   <bean id="motorV8" class="Motor">
   </bean>

</beans>
```

```
public class Auto{
   @Autowired
   @Qualifier("motorV8")
   private Motor motor;

   @PostConstruct
   private void init(){
      …
   }

   @PreDestroy
   private void destroy(){
      …
   }
}
```


ii.xiii Anotaciones JSR 250 (e)

- Orden de inicialización de beans:
  1. @PostConstruct
  2. afterPropertiesSet() -> InitializingBean callback
  3. init-method.

- Orden de destrucción:
  1. @PreDestroy
  2. destroy() -> DisposableBean callback
  3. destroy-method.

- Sólo si los nombres de los métodos son diferentes, si el nombre es el mismo, sólo se ejecuta 1 vez.

ii.xiii Anotaciones JSR 250 (f)

- @Resource
- Aplicable a atributos y métodos con un único argumento.
- Anotación estándar para auto-hilar dependencias, similar a @Autowired.
- Implementa Inyección de Dependencias “byName” por default. Si ninguna dependencia es satisfecha por nombre, resolverá por tipo.

ii.xiii Anotaciones JSR 250 (g)

- @Resource

- @Resource recibe atributo “name” el cual especifica el nombre del bean a inyectar.

- De no recibir el atributo “name” se asume que el nombre del bean es igual al nombre de la propiedad o del método setter anotado.

ii.xiii Anotaciones JSR 250 (h)

- @Resource

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

   <bean class="Auto">
   </bean>

   <bean id="motorV6" class="Motor">
   </bean>

   <bean id="motorV8" class="Motor">
   </bean>

</beans>
```

```
public class Auto{
   @Resource(name="motorV8")
   private Motor motor;
}
```

ii.xiii Anotaciones JSR 250

a. @Resource, @PostConstruct y @PreDestroy

Práctica 16. @Resource, @PostConstruct y @PreDestroy

ii.xiii Anotaciones JSR 250. Práctica 16 (a)

- Práctica 16. @Resource, @PostConstruct, @PreDestroy

- Implementar Inyección de Dependencias mediante las anotaciones estándar JSR 250.

Resumen de la lección

ii.xiii Anotaciones JSR 250

- Conocimos las anotaciones correspondientes al JSR 250 (Common Annotations) referentes a Inyección de Dependencias de Java EE.

- Implementamos Inyección de Dependencias con Spring mediante anotaciones JSR 250.

Esta página fue intencionalmente dejada en blanco.

ii.xiv Component-scan

Objetivos de la lección

ii.xiv Component-scan

- Conocer la utilidad del <context:component-scan />
- Implementar el escaneo de clases para definir y configurar beans mediante anotaciones.
- Agilizar la definición de beans mediante configuración mínima requerida por medio de Bean Configuration File (XML).

ii.xiv Component-scan

a. Esteretipos @Component, @Service, @Repository, @Controller y @RestController

Práctica 17. Component-scan y estereotipos

b. Filtrado de componentes con @ComponentScan

Práctica c. Filtrado de componentes

ii.xiv Component-scan (a)

- <context:component-scan base-package="*base.packages.to.scan*" />
- Habilita y activa el escaneo de clases con definiciones y configuraciones de beans por medio de anotaciones.
- @Component, @Service, @Repository, @Controller, @RestController y @Scope.
- Habilita implícitamente <context:annotation-config />
- Evita definir beans mediante configuración XML.

ii.xiv Component-scan

a. Esteretipos @Component, @Service, @Repository, @Controller y @RestController

Práctica 17. Component-scan y estereotipos

b. Filtrado de componentes con @ComponentScan

Práctica c. Filtrado de componentes

ii.xiv Component-scan (a)

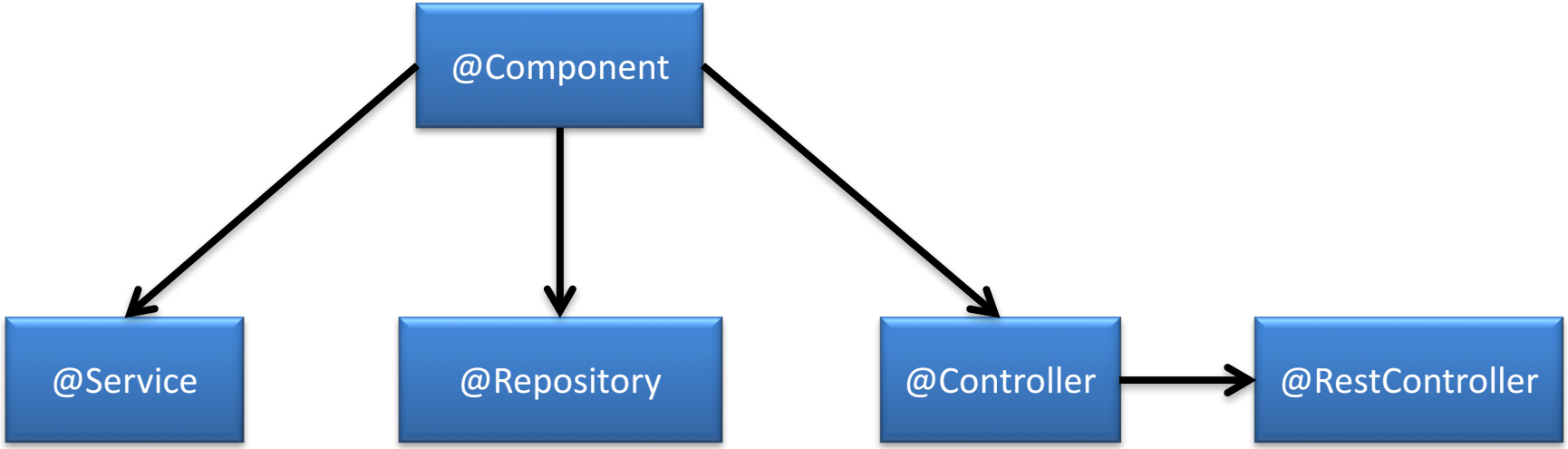
- Esteretipos @Component, @Service, @Repository, @Controller y @RestController

- Permiten anotar beans (clases) mediante anotaciones.

- @Service, @Repository, @Controller y @RestController son estereotipos de @Component.

ii.xiv Component-scan (b)

- Esteretipos @Component, @Service, @Repository, @Controller y @RestController



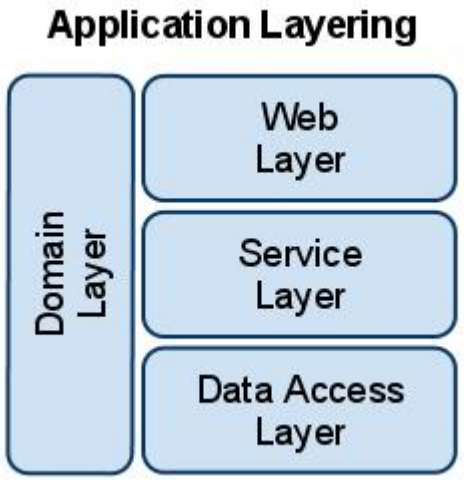
ii.xiv Component-scan (c)

- Esteretipos @Component, @Service, @Repository, @Controller y @RestController

- Permiten clasificar los beans en cada capa de la aplicación.

- Técnicamente es posible implementar los estereotipos de forma indistinta.

- Permiten discriminar clases durante <context:component –scan />

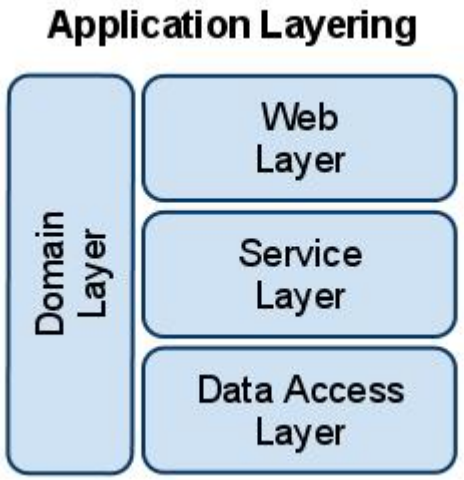


ii.xiv Component-scan (d)

- Esteretipos @Component, @Service, @Repository, @Controller y @RestController

- Permiten aplicar BeanPostProcessors a beans específicos de la aplicación (por capa).

- @Repository convierte unchecked exceptions en DataAccessException.

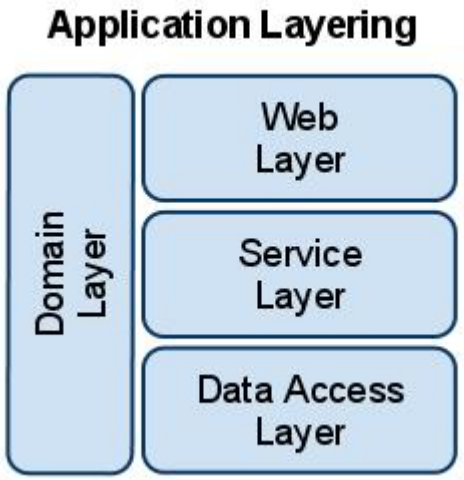


ii.xiv Component-scan (e)

- Esteretipos @Component, @Service, @Repository, @Controller y @RestController

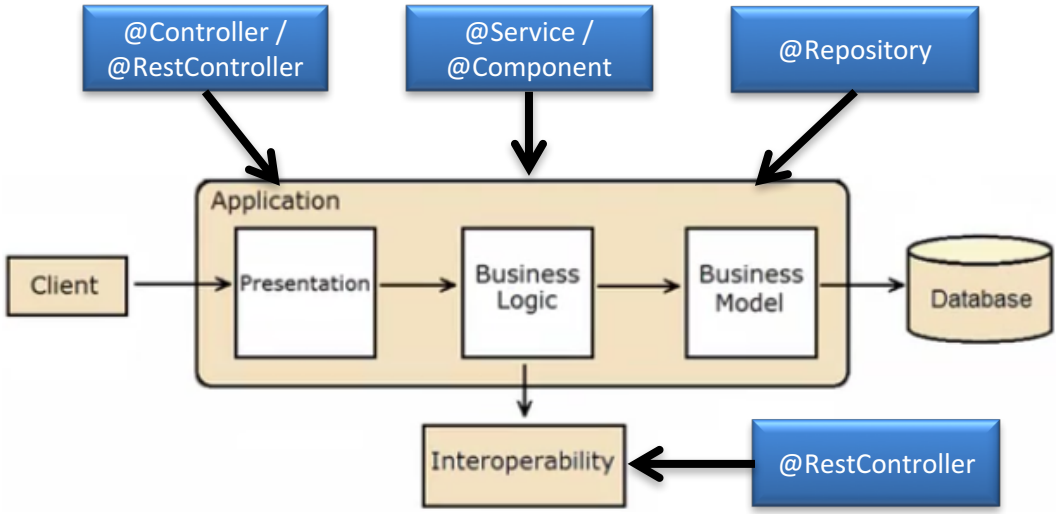
- @Controller habilita el uso de @RequestMapping para mapear URLs a métodos del controlador (Spring MVC).

- @RestController estereotipo de @Controller habilita el uso de @ResponseBody en controladores Spring MVC para permitir respuestas REST (JSON, XML).



ii.xiv Component-scan (f)

- Esteretipos @Component, @Service, @Repository, @Controller y @RestController



ii.xiv Component-scan (g)

- Esteretipos @Component, @Service, @Repository, @Controller y @RestController

- Técnicamente, los estereotipos de @Component fueron diseñados, también, para aplicar “puntos de corte objetivo” (AOP). “target-pointcuts”. Se revisará más adelante durante Spring AOP.

ii.xiv Component-scan (h)

- @Scope
- Aplica en conjunto con @Component y sus estereotipos.
- Define el ámbito (scope) del bean.
- Recordar: Por default los beans son singleton

ii.xiv Component-scan (i)

- @Scope

```
<beans>
    <context:component-scan
        base-package="com.package"/>
</beans>
```

```
@Service("motorV8")
public class Motor{
    private Integer valvulas;
    ...
}
```

```
@Component("autoBean")
@Scope("prototype")
public class Auto{
    @Autowired
    @Qualifier("motorV8")
    private Motor motor;

    public Auto(Motor m){
        this.motor = m;
    }
}
```

ii.xiv Component-scan (j)

- @Primary

- Anotación utilizada para de finir un bean como principal candidato para inyectar (autowiring) en caso de existir más de un bean definido del mismo tipo (ambigüedad). Evita ambigüedad.

ii.xiv Component-scan (k)

- @Primary

```
@Component("motorV8")
@Primary
public class MotorV8 extends Motor{
    private Integer valvulas;
    …
}

@Component("motorV6")
public class MotorV6 extends Motor{
    private Integer valvulas;
    …
}
```

```
@Service
public class Auto{
    @Autowired
    private Motor motor;

    public Auto(Motor m){
        this.motor = m;
    }
}
```

ii.xiv Component-scan

a. Esteretipos @Component, @Service, @Repository, @Controller y @RestController

Práctica 17. Component-scan y estereotipos

b. Filtrado de componentes con @ComponentScan

Práctica c. Filtrado de componentes

Bonus. Spring Test (a)

- @RunWith: Permite definir una implementación especifica de Spring como ejecutor de unit-test de JUnit que habilite inyección de dependencias en la clase Test.
  - SpringJUnit4ClassRunner.class
  - SpringRunner.class (alias de SpringJUnit4ClassRunner.class, a partir de Spring Framework 4.3+)

- @ContextConfiguration: Permite definir los application-context.xml a utilizar para la clase Test.

- Nota: Agregar dependencia spring-test al pom.xml de maven.

Bonus. Spring Test (b)

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(
            "classpath:/spring/practica17/component-scan-stereotypes-application-context.xml")
public class StereotypesBetterTest {
    @Autowired
    private IRestControllerClass restController;
    @Before
    public void beforeClass() {
        Assert.assertNotNull(restController);
    }
    @Test
    public void restControllerTest() {
        String name = "My REST Controller Implementation";
        Assert.assertEquals(name, restController.getRestControllerClassName());
    }
}
```

ii.xiv Component-scan. Práctica 17 (a)

- Práctica 17. Component-scan y estereotipos

- Implementar definición y configuración de beans mediante escaneo de clases.
- Implementar aplicación y uso de @Scope.
- Mejorar el desarrollo de test cases mediante el módulo spring-test.

ii.xiv Component-scan

a. Esteretipos @Component, @Service, @Repository, @Controller y @RestController

Práctica 17. Component-scan y estereotipos

b. Filtrado de componentes con @ComponentScan

Práctica c. Filtrado de componentes

ii.xiv Component-scan (a)

- Filtrado de componentes con @ComponentScan

- Por default todas las clases anotadas con los estereotipos derivados de @Component o meta-anotaciones de @Component, son los únicos tipos detectados candidatos para ser registrados como beans.

- Mediante @ComponentScan podemos incluir o excluir filtros para habilitar o deshabilitar el registro de beans anotados con @Component o alguno de sus estereotipos.

https://docs.spring.io/spring/docs/5.1.4.RELEASE/spring-framework-reference/core.html#beans-scanning-filters

ii.xiv Component-scan (b)

- Configuración de filtrado de componentes:

```
<context:component-scan base-package="org.certificatic.spring.core.practicaC">

    <context:exclude-filter type="annotation"
            expression="org.springframework.stereotype.Repository"/>
    <context:exclude-filter type="assignable"
            expression="org.certificatic.spring.core.practicaC.
                        filteringcomponents.bean.Biker"/>
</context:component-scan>
```

ii.xiv Component-scan (c)

- Configuración de filtrado de componentes:

```
@Configuration
@ComponentScan(basePackages = "org.certificatic.spring.core.practicaC",
        excludeFilters = {
                @Filter(type = FilterType.ANNOTATION, classes = Repository.class),
                @Filter(type = FilterType.ASSIGNABLE_TYPE, classes = Biker.class),
        })
public class FilteringComponentsConfig {

}
```

ii.xiv Component-scan

a. Esteretipos @Component, @Service, @Repository, @Controller y @RestController

Práctica 17. Component-scan y estereotipos

b. Filtrado de componentes con @ComponentScan

Práctica c. Filtrado de componentes

ii.xiv Component-scan. Práctica c (a)

- Práctica c. Filtrado de componentes

- Implementar component-scan para escanear un paquete base de beans.
- Excluir componentes @Repository y tipos asignables a un tipo en particular de clase, del escaneo de componentes.

Resumen de la lección

ii.xiv Component-scan

- Comprendimos la utilización y aplicación del escaneo de paquetes para detectar beans de Spring.
- Implementamos los distintos estereotipos @Component para definir beans de Spring.
- Realizamos definición e inyección de dependencias únicamente por anotaciones.
- Revisamos brevemente el módulo spring-test para mejorar la implementación de pruebas unitarias de beans de Spring.

Esta página fue intencionalmente dejada en blanco.

ii.xv Anotaciones JSR 330

Objetivos de la lección

ii.xv Anotaciones JSR 330

- Conocer las principales anotaciones del JSR 330 Dependency Injection for Java.
- Implementar Inyección de Dependencias mediante @Anotaciones del JSR 330.
- Verificar las similitudes entre anotaciones Spring vs anotaciones JSR 330

ii.xv Anotaciones JSR 330

a. @Inject y @Named

Práctica 18. @Inject y @Named

ii.xv Anotaciones JSR 330 (a)

- JCP define el JSR 330 como un conjunto de anotaciones para implementación de Inyección de Dependencias estándar.

- JSR 330 Incluye las anotaciones @Inject, @Named, @ManagedBean, @Qualifier, @Singleton entre otras, sólo estas se pueden aplicar a Spring.

- JSR 330 incluye @Scope sin embargo, no se implementan en Spring, no se recomienda su uso, ya que sólo sirve como meta-anotación.

- Provider<T> (JSR 330) es similar a ObjectFactory<T>

ii.xv Anotaciones JSR 330 (b)

- Comparativa Anotaciones Spring vs JSR 330

Spring	JSR 330	Comentarios
@Autowired	@Inject	@Inject no contiene atributo *required*, *es posible utilizar* Optional (Java 8) o @Nullable.
@Component	@Named / @ManagedBean	JSR 330 no contiene estereotipos sólo componentes y no pueden usarse como meta-anotación.
@Scope("singleton")	@Singleton	El scope default de JSR 330 es prototype
@Qualifier	@Qualifier / @Named	JSR 330 provee la meta-anotacion @Qualifier para crear "custom" Qualifiers. @Named es similar a @Qualifier.
ObjectFactory<T>	Provider<T>	Provider es una alternativa a ObjectFactory (Bean Factory y ApplicationContext) , sin embargo se requiere inyectar en cada clase donde se requiera utilizar.

ii.xv Anotaciones JSR 330 (c)

- @Inject y @Qualifier
- Similar a @Autowired y más parecido a @Resource.
- Si no se especifica un @javax.inject.Qualifier intentará inyectar "byName" si no, intentará inyectar "byType".
- Si se especifica un @javax.inject.Qualifier tratará de inyectar el bean especificado, si no lanzará una excepción.

ii.xv Anotaciones JSR 330 (d)

- @Inject y @Qualifier

- No soporta atributo *required*, implica que toda dependencia será forzosamente requerida. Es posible omitir este comportamiento mediante la clase Optional (Java 8) o @Nullable.

- Spring soporta conjugar @javax.inject.Inject con @org.springframework.beans.factory.annotation.Qualifier sin embargo es mala práctica.

ii.xv Anotaciones JSR 330 (e)

- @Inject y @Qualifier

```
@Inject
@Qualifier("generalDirectorBean")
private IDirector generalDirector;

@Inject
private IDirector itDirector;

@Inject
@Qualifier("generalSecretaryBean")
private Optional<Secretary> generalSecretary;

@Inject
@SecretaryAssistantEmployeeQualifier
private Secretary secretaryAssistant;
```

```
@javax.inject.Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.METHOD, ElementType.FIELD,
ElementType.PARAMETER, ElementType.TYPE })
public @interface SecretaryAssistantEmployeeQualifier {

}
```

ii.xv Anotaciones JSR 330 (f)

- @Named / @ManagedBean

- Similar a @Component; define un atributo *value* que especifica el id o nombre del bean.

- No contiene estereotipos.

- Puede emplearse con o sin especificar el atributo *value*, en otras palabras al igual que @Component, puede o no definir el nombre del bean.

ii.xv Anotaciones JSR 330 (g)

- @Named / @ManagedBean

```
@Data
@Named
public class Manager {
    @Inject
    @ManagerEmployeeQualifier
    private Employee employee;
}
```

```
@Named
@ManagerEmployeeQualifier
public class EmployeeManager extends Employee {
    public EmployeeManager() {
        this.name = "Ilse Hernández";
        this.dni = "11-44-77-55";
    }
}
```

```
@javax.inject.Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.METHOD,
ElementType.FIELD, ElementType.PARAMETER,
ElementType.TYPE })
public @interface ManagerEmployeeQualifier {

}
```

ii.xv Anotaciones JSR 330 (h)

- @Named / @ManagedBean

```
@Data
@Named
public class Manager {
    @Inject
    @Qualifier(“employeeManager”)
    private Employee employee;
}
```

```
@Named(“employeeManager”)
public class EmployeeManager extends Employee {
    public EmployeeManager() {
        this.name = "Ilse Hernández";
        this.dni = "11-44-77-55";
    }
}
```

ii.xv Anotaciones JSR 330 (i)

- @Singleton

- Por default @Named / @ManagedBean especifica beans prototype, en implementaciones CDI con estándar JSR-330.

- Spring define como singleton los beans anotados con @Named o @ManagedBean, siempre y cuando el motor de IoC y DI sea Spring.

- El JSR 330 establece una anotación @javax.inject.Scope sin embargo al igual que @javax.inject.Qualifier, es necesario implementar tu propio Scope mediante BeansPostProcessor. Mala idea.

ii.xv Anotaciones JSR 330 (j)

- @Singleton

- Si se requiere utilizar otro scope diferente de singleton y prototype con @Named se recomienda utilizar @Scope de Spring framework.

- No se recomienda implementar @javax.inject.Scope.

ii.xv Anotaciones JSR 330 (k)

- @Singleton

```
@Data
@Named("ACMECorporationBean")
@Singleton
public class Corporation {
    …
}
```

```
@Data
@Named("ACMECorporationBean")
@Scope(“singleton”)
public class Corporation {
    …
}
```

ii.xv Anotaciones JSR 330 (I)

- Limitaciones de @Autowired, @Inject, @Resource y @Value

- Las anotaciones **@Autowired**, **@Inject**, **@Resource** y **@Value** son resueltas por una implementación **BeanPostProcessor**, es decir, resuelven e inyectan beans mediante **BeanPostProcessor**, por lo tanto las anotaciones mencionadas no pueden ser utilizadas en una propia implementación de **BeanPostProcessor** o **BeanFactoryPostProcessor**, de ser requerido, sus dependencias deben ser hiladas explícitamente mediante XML o Java Config @Bean.

ii.xv Anotaciones JSR 330. Práctica 18. (a)

- Práctica 18. @Inject y @Named

- Implementar definición y configuración de beans mediante anotaciones estándar JSR 330 y Spring.

- Utilizar distintas combinaciones de anotaciones para obtener mayor agilidad al momento de implementar Inyección de Dependencias con anotaciones estándar.

Resumen de la lección

ii.xv Anotaciones JSR 330

- Implementamos las distintas anotaciones del JSR 330 para definir y configurar beans.
- Realizamos definición e inyección de dependencias únicamente por anotaciones.
- Conjugamos anotaciones estándar JSR 330 y JSR 250 con anotaciones de Spring.

Esta página fue intencionalmente dejada en blanco.

ii.xvi Spring Java Config

Objetivos de la lección

ii.xvi Spring Java Config

- Conocer el mecanismo de configuración de beans que provee Spring de forma programática (Java Config).
- Implementar las @Anotaciones correspondientes a la configuración de beans de forma programática (Java Config).

ii.xvi Spring Java Config

a. @Configuration, @Bean e @Import

Práctica 19. @Configuration, @Bean e @Import

ii.xvi Spring Java Config (a)

- Spring provee de mecanismos para centralizar la configuración de beans en clases Java sin necesidad de Bean Configuration File (XML).

- Definición de beans con anotaciones @Component, @Service, @Repository, @Controller, @RestController, @Named, @Scope, etc.

- Inyección de Dependencias con anotaciones @Autowire, @Inject, @Resource, @Qualifier, etc.

- Definición de beans programáticamente mediante Java Config @Configuration, @ComponentScan, @Bean, @Import, @Scope, @Qualifier, etc.

ii.xvi Spring Java Config

a. @Configuration, @Bean e @Import

Práctica 19. @Configuration, @Bean e @Import

ii.xvi Spring Java Config (a)

- @Configuration

- Indica que la clase anotada, es una clase de configuración de beans, similar a especificar un archivo **application-context.xml**.

- @ComponentScan(basePackages = “com.packages”)

- Habilita el escaneo de clases en el paquete determinado.

- Clases de configuración @Configuration no pueden ser finales.

ii.xvi Spring Java Config (b)

- @Configuration y @ComponentScan

```
@Configuration
@ComponentScan(basePackages = “com.packages")
public class AppConfig {
    ...
}
```

```
<beans>
    <context:component-scan
        base-package=“com.packages"/>
</beans>
```

ii.xvi Spring Java Config (c)

- @Bean
- Aplicable a nivel de método de clase Java Config (@Configuration)
- Define como un bean, de Spring, el objeto retornado por el método.
- Por default el nombre del método será el nombre o id del bean.
- Es posible definir un qualifier (sólo para Spring) que provea un nombre especifico al bean mediante @Qualifier.

ii.xvi Spring Java Config (d)

- @Bean

```
@Configuration
public class AppConfig {
        @Bean
        public TransferServiceImpl transferService() {
                return new TransferServiceImpl();
        }
}
```

```
<beans>
        <bean id="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

ii.xvi Spring Java Config (e)

- @Bean y @Qualifier

```
@Configuration
@ComponentScan(basePackages = "com.packages")
public class AppConfig {
    @Bean
    @Qualifier("quadraticEquationServiceBean")
    public IQuadraticEquationService quadraticService2() {
        return new QuadraticEquationServiceImpl();
    }
}
```

```
@Inject
private IQuadraticEquationService
                        quadraticService2;

@Inject
@Qualifier("quadraticService2")
private IQuadraticEquationService
                        quadraticService2x;

@Inject
@Qualifier("quadraticEquationServiceBean")
private IQuadraticEquationService
                        quadraticService2xx;
```

ii.xvi Spring Java Config (f)

- @Bean y @Scope

- Es posible definir el scope de un bean definido mediante Java Config con @Bean y @Scope.

- Mediante @Bean es posible definir init-method y destroy-method callbacks.

- Por default, Java Config, define el método público “close” o “shutdown” como destroy-method. Para deshabilitar dicha configuración es necesario definir: @Bean(destroyMethod="")

ii.xvi Spring Java Config (g)

- @Bean y @Scope

```
@Bean(initMethod = "init", destroyMethod = "destroy")
@Qualifier("quadraticEquationServiceBean")
public IQuadraticEquationService quadraticService2() {
    return new QuadraticEquationServiceImpl();
}

@Bean
@Scope("prototype")
public IQuadraticEquationService quadraticService3() {
    return new QuadraticEquationServiceImpl();
}
```

ii.xvi Spring Java Config (h)

- Inyectando dependencias internas de beans con Java Config.

```
@Configuration
public class AppConfig {
        @Bean
        public BeanOne beanOne() {
                return new BeanOne(beanTwo());
        }
        @Bean
        public BeanTwo beanTwo() {
                return new BeanTwo();
        }
}
```

ii.xvi Spring Java Config (i)

- @Import/<import>

- ¿Que pasa si el Bean Definition File se vuelve muy grande?
  - Difícil de mantener.

- Divide y vencerás

- Segrega el Bean Definition File en múltiples XMLs o múltiples clases de configuración @Configuration.

ii.xvi Spring Java Config (j)

- <import>
  - xml-context-config.xml

```
<beans>
   <import resource="classpath:xml-repository-config.xml"/>
   <import resource="classpath:xml-service-config.xml"/>
</beans>
```

  - xml-service-config.xml

```
<beans>
   <bean id="lister" class="mypackage.MovieLister">
      <constructor-arg ref="finder"/>
   </bean>
</beans>
```

  - xml-repository-config.xml

```
<beans>
   <bean id="finder" class="mypackage.MovieFinder"/>
</beans>
```

ii.xvi Spring Java Config (k)

- @Import
- @Import aplica sobre clases de configuración @Configuration Java Config.
- Importa clases de configuración Java Config (no Bean Configuration Files en XML).

ii.xvi Spring Java Config (I)

- @ImportResource
- Aplicable sobre clases de configuración @Configuration Java Config.
- Permite importar un recurso XML, es decir, un Bean Configuration File (application-context.xml).
- Es posible combinar las distintas configuraciones.

ii.xvi Spring Java Config (m)

- @Import e @ImportResource

```
@Configuration
@ComponentScan(basePackages = { "org.certificatic.spring.core.practica19.javaconfig" })
@Import({ RepositoryConfig.class, ServiceConfig.class })
public class ApplicationConfig {
    ...
}
```

```
@Configuration
public class ServiceConfig {
}
```

```
@Configuration
@ImportResource(locations = { "classpath:/spring/practica19/datasource-application-context.xml" })
public class RepositoryConfig {
    ...
}
```

ii.xvi Spring Java Config (n)

- Inyectando dependencias externas de beans con Java Config.
- Los métodos @Bean pueden tener cualquier número de argumentos, los cuales serán aplicados como dependencias del bean que define.
- Siendo los argumentos del método @Bean, las dependencias del bean, éstas aplican como dependencias a “auto-inyectar” (autowiring) por constructor (byType).
- Para hacer referencia a las dependencias del Bean no es necesario especificarlos explícitamente mediante “ref=beanName” como en XML, debido a que Java Config es fuertemente tipado.

ii.xvi Spring Java Config (ñ)

- Inyectando dependencias externas de beans con Java Config.

```
@Configuration
public class ServiceConfig {
    @Bean
    public TransferService transferService(
            AccountRepository accountRepo) {
        return new TransferServiceImpl(accountRepo);
    }
}

@Configuration
public class RepositoryConfig {
    @Bean
    public AccountRepository accountRepo(DataSource dataSource) {
        return new JdbcAccountRepository(dataSource);
    }
}
```

```
@Configuration
@Import({ServiceConfig.class,
        RepositoryConfig.class})
public class SystemTestConfig {
    @Bean
    public DataSource dataSource() {
        return new DataSource(…);
    }
}
```

ii.xvi Spring Java Config (o)

- Inyectando dependencias externas de beans con Java Config.

- Recordar que una clase de configuración @Configuration no es más que otro bean definido en el ApplicationContext.

- También es posible inyectar dependencias externas de beans a través de @Autowired sobre las clases de configuración @Configuration.

ii.xvi Spring Java Config (p)

- Inyectando dependencias externas de beans con Java Config (a).

```
@Configuration
public class RepositoryConfig {

    private final DataSource dataSource;

    @Autowired
    public RepositoryConfig(DataSource dataSource) {
        this.dataSource = dataSource;
    }
    @Bean
    public AccountRepository accountRepo() {
        return new JdbcAccountRepository(
                                        dataSource);
    }
}
```

```
@Configuration
public class ServiceConfig {

    @Autowired
    private AccountRepository accountRepo;

    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl(
                                    accountRepo);
    }
}
```

ii.xvi Spring Java Config (q)

- Inyectando dependencias externas de beans con Java Config (b).

```
@Configuration
@Import({ServiceConfig.class, RepositoryConfig.class})
public class SystemTestConfig {

        @Bean
        public DataSource dataSource() {
                return new DataSource(…);
        }
}
```

ii.xvi Spring Java Config (r)

- Instanciando ApplicationContext desde clases @Configuration.

- A partir de Spring 3.0+ la clase AnnotationConfigApplicationContext permite inicializar el ApplicationContext a partir de clases de configuración Java Config.

- AnnotationConfigApplicationContext también soporta como entrada clases @Component (y sus estereotipos) así como clases anotadas con el JSR 330 para generar un ApplicationContext con únicamente dichos componentes.

ii.xvi Spring Java Config (s)

- Instanciando ApplicationContext desde clases @Configuration.

```
public static void main(String[] args) {

        ApplicationContext ctx = new AnnotationConfigApplicationContext(
                                                        AppConfig.class);
        MyService myService = ctx.getBean(MyService.class);

        myService.doStuff();
}
```

ii.xvi Spring Java Config (t)

- Instanciando ApplicationContext desde clases @Component.

```
public static void main(String[] args) {

        ApplicationContext ctx = new AnnotationConfigApplicationContext(
                                                MyServiceImpl.class, Dependency1.class,
                                                Dependency2.class);
        MyService myService = ctx.getBean(MyService.class);

        myService.doStuff();
}
```

ii.xvi Spring Java Config (u)

- Instanciando ApplicationContext programaticamente.

```
public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx = new
                                                AnnotationConfigApplicationContext();
        ctx.scan("com.acme");
        ctx.register(OtherConfig.class);
        ctx.register(AdditionalConfig.class);

        ctx.refresh();

        MyService myService = ctx.getBean(MyService.class);
        myService.doStuff();
}
```

```
package com.acme;

@Configuration
public class AppConfig { ... }
```

ii.xvi Spring Java Config

a. @Configuration, @Bean e @Import

Práctica 19. @Configuration, @Bean e @Import

ii.xvi Spring Java Config. Práctica 19 (a)

- Práctica 19. @Configuration, @Bean e @Import

- Implementar una configuración básica de beans de Spring mediante Java Config.

- Utilizar las distintas anotaciones para definir beans por medio de Java Config así como importar otras clases Java Config a la clase Java Config principal e importar recursos XML.

- Analizar el alcance de mezclar los 3 tipos de configuración de beans de Spring.

Resumen de la lección

ii.xvi Spring Java Config

- Implementamos definición de beans mediante Java Config.
- Comprendimos como realizar configuración de beans programáticamente.
- Conocimos como aplicar Inyección de Dependencias a beans configurados con @Bean.
- Analizamos la posibilidad de utilizar los 3 diferentes tipos de configuración de beans.

Esta página fue intencionalmente dejada en blanco.

Trabajo de Integración 2. Convertidor número letra configuración por @Anotaciones

Objetivos de la lección

Trabajo de Integración 2. Convertidor número letra configuración por @Anotaciones

- Desacoplar la el diseño del Convertidor número a letra que permita configurar ágilmente la traducción de los textos numéricos en distintos idiomas.

- El objetivo es aplicar lo aprendido hasta el momento para desarrollar un componente convertidor número a letra multi-idioma.

ii. Spring Core – T.I. 2. Convertidor número letra configuración por @Anotaciones

Trabajo de Integración 2. Convertidor número letra configuración por @Anotaciones

Tarea 3. Migración Convertidor número letra configuración por @Anotaciones

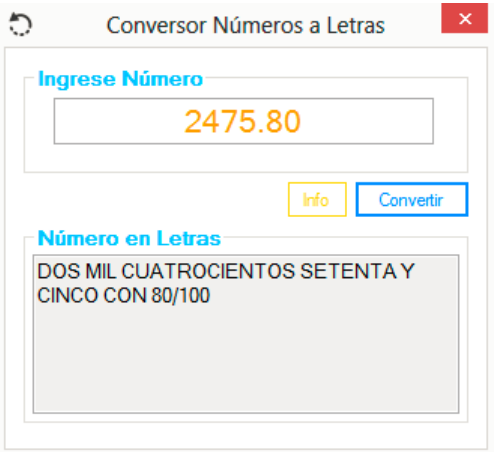
Trabajo de Integración 2. Convertidor número letra configuración por @Anotaciones (a)

- Desarrollar un componente convertidor número a texto en dos o más idiomas diferentes, utilizando Inyección de Dependencias con @Anotaciones, Java Config o cualquier estrategia que considere relevante para su aprendizaje.

ii. Spring Core – T.I. 2. Convertidor número letra configuración por @Anotaciones

Trabajo de Integración 2. Convertidor número letra configuración por @Anotaciones (b)

- Puede re-ajustar el diseño de clases como considere, sin embargo se recomienda no sobre-escribir el componente funcional de parseo entre números a letras.



ii. Spring Core – T.I. 2. Convertidor número letra configuración por @Anotaciones

Trabajo de Integración 2. Convertidor número letra configuración por @Anotaciones

Tarea 3. Migración Convertidor número letra configuración por @Anotaciones

Trabajo de Integración 2. Tarea 3 (a)

- Tarea 3. Migración Convertidor número letra configuración por @Anotaciones

- Analizar el código referido al componente NumericalConverter.
- Realizar la configuración de beans por @Anotaciones, Java Config o la configuración que considere para que se adecue más a las dependencias de cada componente.

ii. Spring Core – T.I. 2. Convertidor número letra configuración por @Anotaciones

Trabajo de Integración 2. Tarea 3 (b)

- Tarea 3. Migración Convertidor número letra configuración por @Anotaciones

- La salida del Componente debe ser aproximada a la siguiente.

```
abr 08, 2016 11:48:56 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadB
INFORMACIÓN: Loading XML bean definitions from class path resource [beansConvertidorNumeroLe
abr 08, 2016 11:48:56 PM org.springframework.beans.factory.support.DefaultListableBeanFactor
INFORMACIÓN: Pre-instantiating singletons in org.springframework.beans.factory.support.Defau
numero: 730424
SEVEN-HUNDRED  THIRTY  THOUSAND FOUR-HUNDRED  TWENTY   FOUR  DOLLARS 00/100
abr 08, 2016 11:48:56 PM org.springframework.context.support.AbstractApplicationContext doCl
INFORMACIÓN: Closing org.springframework.context.support.ClassPathXmlApplicationContext@d5f6
abr 08, 2016 11:48:56 PM org.springframework.beans.factory.support.DefaultSingletonBeanRegis
INFORMACIÓN: Destroying singletons in org.springframework.beans.factory.support.DefaultListA
```

ii. Spring Core – T.I. 2. Convertidor número letra configuración por @Anotaciones

Trabajo de Integración 2. Tarea 3 (b)

- Tarea 3. Migración Convertidor número letra configuración por @Anotaciones

- Considerar un diseño multi-idioma y que para el cambio de éste, sólo sea necesario cambiar textos en un Bean Configuration File (application-context.xml).

- No se recomienda implementar distintas clases concretas con las traducciones necesarias.

ii. Spring Core – T.I. 2. Convertidor número letra configuración por @Anotaciones

Resumen de la lección

Trabajo de Integración 2. Convertidor número letra configuración por @Anotaciones (a)

- Explotamos las características aprendidas hasta el momento referidas a la configuración de beans con Spring framework mediante el uso de configuración por Bean Configuration File (XML), @Anotaciones y Java Config.
- Analizamos y comprendimos la importancia del diseño de clases para nuestros componentes (alta cohesión y bajo acoplamiento).

ii. Spring Core – T.I. 2. Convertidor número letra configuración por @Anotaciones

Esta página fue intencionalmente dejada en blanco.

ii.xvii Resources

Objetivos de la lección

ii.xvii Resources

- Conocer las distintas implementaciones del API Resources de Spring.
- Verificar las diferentes formas de obtener un recurso.
- Implementar Inyección de Dependencias de Recursos mediante XML.
- Inyectar Recursos en beans de Spring mediante ResourceLoaderAware.
- Inyectar propiedades mediante placeholders y anotacion @Value.
- Conocer como cargar archivos de propiedades mediante @PropertySource.

ii.xvii Resources

a. Tipos de Resources
b. Tipos de ApplicationContext
c. Inyección y obtención de recursos
d. Obtención de archivos de propiedades
Práctica 20. Resources
e. @PropertySource
Práctica d. @PropertySource

ii.xvii Resources (a)

- Resources

- Spring provee un mecanismo estandarizado para obtener un recurso mediante la interface Resource.

- Recurso se define como cualquier archivo, imagen, texto, URL, etc.

- Los recursos pueden ubicarse en cualquier ubicación tal como:
  - Classpath
  - Sistema de archivos
  - Dirección URL.

ii.xvii Resources

a. Tipos de Resources
b. Tipos de ApplicationContext
c. Inyección y obtención de recursos
d. Obtención de archivos de propiedades
Práctica 20. Resources
e. @PropertySource
Práctica d. @PropertySource

ii.xvii Resources (a)

- Tipos de Resources

- Existen múltiples implementaciones (a):
  - **UrlResource**: representa un recurso mediante una URL (http: ó https:)
  - **ClasspathResource**: representa un recurso en el classpath (classpath:)
  - **FileSystemResource**: representa un recurso en el sistema de archivos (file:)

ii.xvii Resources (b)

- Existen múltiples implementaciones (b):
  - **ServletContextResource**: representa un recurso desplegado en un contexto web.
  - **InputStreamResource**: representa un recurso mediante un flujo InputStream cuando no es posible representar su implementación con alguna de las anteriores.
  - **ByteArrayResource**: Representa un recurso mediante un arreglo de bytes mediante un ByteArrayInputStream.

- No debemos preocuparnos por elegir una implementación, Spring determina la implementación Resource adecuada.

ii.xvii Resources (c)

- Spring utiliza ampliamente la abstracción Resource internamente, por ejemplo en las implementaciones de ApplicationContext.

- **ResourceLoader**: Interface que permite la obtención de recursos.

- Todos los ApplicationContext implementan la interface ResourceLoader, por tanto cualquier implementación ApplicationContext permite la obtención de recursos.

ii.xvii Resources (d)

- Obtención de recursos mediante ResourceLoader.

- Resource file =
  resourceLoader.getResource("file:///path/file/name.txt");

- Resource file =
  resourceLoader.getResource("classpath:path/file/name.txt");

- Resource file =
  resourceLoader.getResource("http://myserver/name.txt");

ii.xvii Resources (e)

- ResourceLoader y ApplicationContext

- La relación entre ApplicationContext y ResourceLoader es muy estrecha.

- Es posible obtener recursos a partir del ApplicationContext debido a que sus implementaciones implementan ResourceLoader sin embargo, en una aplicación Spring empresarial es mala idea inyectar un ApplicationContext en un bean de Spring.

ii.xvii Resources

a. Tipos de Resources

b. Tipos de ApplicationContext

c. Inyección y obtención de recursos

d. Obtención de archivos de propiedades

Práctica 20. Resources

e. @PropertySource

Práctica d. @PropertySource

ii.xvii Resources (a)

- Tipos de ApplicationContext

- Existen principalmente tres implementaciones de ApplicationContext
  - ClassPathXmlApplicationContext
  - FileSystemXmlApplicationContext
  - XmlWebApplicationContext

ii.xvii Resources (b)

- Tipos de ApplicationContext

- **ClassPathXmlApplicationContext**: Por default esta implementación localiza los recursos en el classpath (prefijo classpath: por default).

- **FileSystemXmlApplicationContext**: Por default esta implemetnación localiza los recursos en el sistema de archivos (prefijo file: por default).

- **XmlWebApplicationContext**: Contexto utilizado en ambiente web para Spring web o Spring MVC. Por default localiza los recursos en la carpeta webapp del proyecto.

ii.xvii Resources

a. Tipos de Resources
b. Tipos de ApplicationContext
c. Inyección y obtención de recursos
d. Obtención de archivos de propiedades
Práctica 20. Resources
e. @PropertySource
Práctica d. @PropertySource

ii.xvii Resources (a)

- Inyección y obtención de recursos.

- Es posible inyectar recursos a un bean si la propiedad a inyectar es de tipo Resource.

```
<bean class="BeanResource">
    <property name="template"
        value="some/resource/path/myTemplate.txt"/>
</bean>
```

```
public class BeanResource{
    private Resource template;
    public void setTemplate(Resource file){
        this.template = file;
    }
}
```

ii.xvii Resources (b)

- Inyección y obtención de recursos.

- Es posible obtener recursos en un ApplicationContext utilizando los prefijos classpath:, file:, http:, https:.

- Si no se especifica un prefijo, la implementación Resource que el ResourceLoader utilizará será dependiente de la implementación del ApplicationContext.

ii.xvii Resources (c)

- Inyección de recursos.

- Ejemplo.

```
<bean class="BeanResources">
    <property name="txtFile"
        value="file:path/to/my-text-file.txt" />
    <property name="imageFile"
        value="file:src/main/resources/path/to/logo.png" />
    <property name="propertiesFile"
        value="classpath:path/to/my-properties.properties" />
    <property name="urlFile" value="https://spring.io/" />
</bean>
```

```
public class BeanResource{
    private Resource txtFile;
    private Resource imageFile;
    private Resource propertiesFile;
    private Resource urlFile;
    //getters and setters
}
```

ii.xvii Resources (d)

- ¿Como poder inyectar recursos Resource?
  - Inyectando Resources mediante XML.
  - Implementando interface ResourceLoaderAware.

- ¿Inyectando Application Context?
  - Mala idea

ii.xvii Resources (e)

- Implementación ResourceLoaderAware

```
@Component
public class BeanResourceLoaderAware implements ResourceLoaderAware {

    private ResourceLoader resourceLoader;

    @Override
    public void setResourceLoader(ResourceLoader resourceLoader) {
        this.resourceLoader = resourceLoader;
    }
}
```

ii.xvii Resources

a. Tipos de Resources
b. Tipos de ApplicationContext
c. Inyección y obtención de recursos
d. Obtención de archivos de propiedades
Práctica 20. Resources
e. @PropertySource
Práctica d. @PropertySource

ii.xvii Resources (a)

- Obtención de archivos de propiedades

- Como buena práctica, se recomienda externalizar los valores escalares que los beans requieran mediante un archivo de propiedades.

- Es posible cargar archivos de propiedades configurando un bean del tipo PropertyPlaceholderConfigurer en ves de trabajarlo manualmente vía Resource.

- El bean PropertyPlaceholderConfigurer mantiene una referencia a las propiedades y éstas pueden ser accesibles a través de toda la aplicación.

ii.xvii Resources (b)

- Configuración PropertyPlaceholderConfigurer

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations" value="classpath:com/foo/jdbc.properties"/>
</bean>

<bean id="dataSource" destroy-method="close"
                                        class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

ii.xvii Resources (c)

- Obtención de archivos de propiedades con <context:property-placeholder>

- El namespace <context: /> permite la configuración de un PropertyPlaceHolderConfigurer con una simple instrucción.

```
<context:property-placeholder location="classpath:com/foo/jdbc.properties"/>
```

ii.xvii Resources

a. Tipos de Resources
b. Tipos de ApplicationContext
c. Inyección y obtención de recursos
d. Obtención de archivos de propiedades
Práctica 20. Resources
e. @PropertySource
Práctica d. @PropertySource

ii.xvii Resources. Práctica 20. (a)

- Práctica 20. Resources
- Implementar inyección de dependencias de tipo Resource.
- Implementar diferentes formas de obtener recursos en classpath, sistema de archivos y URLs web.
- Implementar inyección de dependencias de ResourceLoader mediante ResourceLoaderAware.

ii.xvii Resources

a. Tipos de Resources
b. Tipos de ApplicationContext
c. Inyección y obtención de recursos
d. Obtención de archivos de propiedades
Práctica 20. Resources
e. @PropertySource
Práctica d. @PropertySource

ii.xvii Resources (a)

- @PropertySource

- A partir de Spring Framework 3.0+, se permite la carga de propiedades mediante anotación @PropertySource utilizando Java Config.

- @PropertySource es similar a utilizar <context:property-placeholder />

- La anotación @PropertySource provee un mecanismo declarativo para agregar propiedades al objeto Environment de Spring que, a su vez, dichas propiedades estarán disponibles como property-placeholders para usar sobre la anotación @Value.

ii.xvii Resources (b)

- @PropertySource

```
@Configuration
@PropertySource("classpath:/com/myco/app.properties")
public class AppConfig {
    @Autowired
    Environment env;

    @Bean
    public TestBean testBean() {
        TestBean testBean = new TestBean();
        testBean.setName(env.getProperty("testbean.name"));
        return testBean;
    }
}
```

ii.xvii Resources

a. Tipos de Resources
b. Tipos de ApplicationContext
c. Inyección y obtención de recursos
d. Obtención de archivos de propiedades
Práctica 20. Resources
e. @PropertySource
Práctica d. @PropertySource

ii.xvii Resources. Práctica d (a)

- Práctica d. @PropertySource

- Implementar carga de archivo de propiedades mediante @PropertySource.

- Implementar inyección de dependencias de property-placeholders mediante Environment y @Value.

Resumen de la lección

ii.xvii Resources

- Comprendimos la paridad entre ApplicationContext y ResourceLoader.
- Aplicamos inyección de dependencias a beans cuyas dependencias son del tipo Resource.
- Implementamos inyección de dependencias de ResourceLoader como buena práctica para obtener recursos cuando la aplicación lo solicite.
- Conocimos los diferentes prefijos aplicables para la obtención de recursos en Spring.
- Aprendimos como integrar archivos de propiedades al objeto Environment mediante anotaciones con @PropertySource.

Esta página fue intencionalmente dejada en blanco.

ii.xviii Spring Expression Language (SpEL)

Objetivos de la lección

ii.xviii Spring Expression Language (SpEL)

- Conocer a profundidad el API SpEL.
- Realizar un análisis del poder del API de SpEL para trabajar con objetos o beans de Spring.
- Identificar los distintos operadores que soporta SpEL, muchos de ellos se implementan en Groovy.
- Realizar inyección de dependencias mediante expresiones SpEL tanto en configuración XML como por @Anotaciones.

ii.xviii Spring Expression Language (SpEL)

a. Introducción

b. Evaluación de expresiones

c. Referencias del lenguaje

Práctica 21. API SpEL

ii.xviii Spring Expression Language (SpEL) (a)

- SpEL

- Spring Expression Language es un lenguaje de expresiones muy poderoso que provee de un mecanismo ágil para el acceso y manipulación de objetos en tiempo de ejecución.

- Utiliza una sintaxis similar a JSF EL, pero ofrece funcionalidades como invocación de métodos y definición de plantillas mediante simples Strings.

ii.xviii Spring Expression Language (SpEL) (b)

- Spring Expression Language permite evaluar expresiones complejas.
- Es posible utilizar SpEL sin utilizar IoC de Spring.
- SpEL es un API.
- SpEL no soporta expresiones que contengan bucles (while, for).

ii.xviii Spring Expression Language (SpEL) (c)

- Spring Expression Language soporta (a):
  - Expresiones literales
  - Operadores relacionales y lógicos
  - Expresiones regulares
  - Expresiones de clases
  - Acceso a propiedades, arreglos, listas y mapas.
  - Invocación de métodos
  - Asignación de variables
  - Invocación de constructores
  - Referenciar a beans directamente

ii.xviii Spring Expression Language (SpEL) (d)

- Spring Expression Language soporta (b):
  - Construcción de arreglos
  - Definición en línea de listas y mapas
  - Expresiones con operador ternario
  - entre otros.

ii.xviii Spring Expression Language (SpEL)

a. Introducción

b. Evaluación de expresiones

c. Referencias del lenguaje

Práctica 21. API SpEL

ii.xviii Spring Expression Language (SpEL) (a)

- Evaluación de expresiones.
- SpEL provee un API rica para crear expresiones.
- **ExpressionParser**: Es la interface responsable del parseo de una expresión representada por medio de un String.
- **Expression**: Es responsable de evaluar la expresión definida en el ExpressionParser.

ii.xviii Spring Expression Language (SpEL) (b)

- Uso de evaluación de expresiones.

- #{ <expresión> }

- Inyección de dependencias vía setter o constructor mediante configuración XML.

```
<bean class=“GuessNumber">
    <property name="randomNumber" value="#{ T(java.lang.Math).random() * 100.0 }"/>
</bean>
```

ii.xviii Spring Expression Language (SpEL) (c)

- Referencia a beans.

```
<bean id="gessNumberBean" class="GuessNumber">
    <property name="randomNumber" value="#{ T(java.lang.Math).random() * 100.0 }" />
</bean>

<bean class="Magician">
    <property name="initialNumber" value="#{gessNumberBean.randomNumber }" />
</bean>
```

ii.xviii Spring Expression Language (SpEL) (d)

- Inyección de Dependencias mediante SpEL con @Anotaciones
- @Value
- Aplicable a campos, métodos y argumentos para especificar un valor por defecto.

```
@Value("#{systemProperties}")
private Properties properties;
```

ii.xviii Spring Expression Language (SpEL) (e)

- Es posible mezclar inyección de dependencias con @Autowired y @Value.

```
public class MovieLister {
    private MovieFinder movieFinder;
    private String defaultLocale;

    @Autowired
    public void configure(MovieFinder movieFinder,
                @Value("#{ systemProperties['user.region'] }") String defaultLocale) {
        this.movieFinder = movieFinder;
        this.defaultLocale = defaultLocale;
    }
}
```

ii.xviii Spring Expression Language (SpEL)

a. Introducción

b. Evaluación de expresiones

c. Referencias del lenguaje

Práctica 21. API SpEL

ii.xviii Spring Expression Language (SpEL) (a)

- Referencias del Lenguaje: Expresiones literales

- La expresión de literales soporta:
  - Strings (delimitados por comilla simple)
  - Date
  - Números (int, float, double, hexadecimal)
  - Boleanos
  - Null

- Las expresiones literales se definen mediante comilla simple.

ii.xviii Spring Expression Language (SpEL) (b)

- Referencias del Lenguaje: Expresiones literales

- Ejemplo

```
ExpressionParser spelParser = new SpelExpressionParser();

Expression spelExpression = spelParser.parseExpression("'Ivan García'");
String stringExpression = (String) spelExpression.getValue();

Assert.assertEquals("Ivan García", stringExpression);
```

ii.xviii Spring Expression Language (SpEL) (c)

- Referencias del Lenguaje: Properties, Arrays, Lists y Maps

- Para acceder con SpEL a objetos es necesario crear un contexto de evaluación de la expresión.

```
ExpressionParser spelParser = new SpelExpressionParser();
Inventor tesla = createTesla();
EvaluationContext teslaContext = new StandardEvaluationContext(tesla);

String teslaName = (String)
spelParser.parseExpression("name").getValue(teslaContext);
```

ii.xviii Spring Expression Language (SpEL) (d)

- Referencias del Lenguaje: Properties, Arrays, Lists y Maps

- Para acceder a propiedades del objeto contexto de la expresión a evaluar no es necesario utilizar comilla simple, pues no es una literal.

```
EvaluationContext teslaContext = new
        StandardEvaluationContext(tesla);

String teslaNationality = (String) spelParser.
        parseExpression("nationality").
        getValue(teslaContext);
```

```
public class Inventor {
    private String name;
    private String nationality;
    private String[] inventions;
    …
    //getters y setters
}
```

ii.xviii Spring Expression Language (SpEL) (e)

- Acceso a Arreglos

```
String invention = (String) spelParser.parseExpression("inventions[2]").getValue(teslaContext);
```

- Acceso a listas

```
String teslaName = (String)
spelParser.parseExpression(“members[2].name").getValue(societyContext);

String invention = (String) spelParser.parseExpression("getMembers()[2].getInventions()[1]").
                                                        getValue(societyContext);
```

ii.xviii Spring Expression Language (SpEL) (f)

- Acceso a Mapas

- Para acceder a la llave del mapa es necesario utilizar corchetes.

```
Inventor president = (Inventor) spelParser.parseExpression("officers['president']").
                                                        getValue(societyContext);

public class Society {
    private String name;
    private List<Inventor> members;
    private Map<String, Object> officers;
}
```

ii.xviii Spring Expression Language (SpEL) (g)

- Acceso a Listas en línea

```
List numbers = (List) spelParser.parseExpression("{1,2,3,4}").getValue(context);

List listOfLists = (List) spelParser.parseExpression("{{'a','b'},{'x','y'}}"). getValue(context);


Integer sum = numbers.stream().reduce(0, (i, j) -> i + j);
```

ii.xviii Spring Expression Language (SpEL) (h)

- Acceso a Mapas en línea

```
Map inventorInfo = (Map) spelParser.parseExpression(
                                "{name:'Nikola', dob:'10-July-1856'}").getValue(context);

Map mapOfMaps = (Map) spelParser.parseExpression(
            "{name:{first:'Nikola',last:'Tesla'},dob:{day:10,month:'July',year:1856}}").
                                                                    getValue(context);


String nikolaName = (String) spelParser.parseExpression("['name']").getValue(inventorInfo);
```

ii.xviii Spring Expression Language (SpEL) (i)

- Construcción de Arreglos.

- Es posible construir arreglos en línea con la misma sintaxis de Java.

```
int[] numbers1 = (int[]) spelParser.parseExpression("new int[4]").getValue(context);

int[] numbers2 = (int[]) spelParser.parseExpression("new int[]{1,2,3}").getValue(context);

int[][] numbers3 = (int[][]) spelParser.parseExpression("new int[4][5]").getValue(context);
```

ii.xviii Spring Expression Language (SpEL) (j)

- Referencias del Lenguaje: Métodos

- Soporta la sintaxis típica de Java, es posible pasar argumentos como literales y varargs.

```
EvaluationContext teslaContext teslaContext = new StandardEvaluationContext(tesla);

String teslaFirstName = (String) spelParser.parseExpression("name.substring(0,6)").
                                                    getValue(teslaContext);
```

ii.xviii Spring Expression Language (SpEL) (k)

- Referencias del Lenguaje: Operadores

- Soporta diferentes tipos de operadores:
  - Relacionales: lt (<), gt (>), le (<=), ge (>=), eq (==), ne (!=), div (/), mod (%), not (!).
  - Lógicos: and, or, not (!).
  - Matemáticos

ii.xviii Spring Expression Language (SpEL) (I)

- Operadores Relacionales.

```
boolean booleanTrueExpression = (Boolean) spelParser.parseExpression("2 == 2").getValue();

boolean booleanTrueExpression = (Boolean) spelParser.parseExpression("2 eq 2").getValue();


boolean booleanFalseExpression = (Boolean) spelParser.parseExpression("2 < 2").getValue();

boolean booleanFalseExpression = (Boolean) spelParser.parseExpression("2 lt 2").getValue();
```

ii.xviii Spring Expression Language (SpEL) (m)

- Operadores Lógicos.

```
boolean booleanExpression = (Boolean) spelParser.parseExpression("true and false").
                                                                        getValue();

boolean booleanExpression = (Boolean) spelParser.parseExpression(
                        “isMember('Anatoly Alexandrov') and isMember('Bruce Ames')").
                                                        getValue(societyContext);
```

ii.xviii Spring Expression Language (SpEL) (n)

- Operadores Matemáticos.

```
Integer integerValue = (Integer) spelParser.parseExpression("1 + 2 + 3").getValue();

integerValue = (Integer) spelParser.parseExpression("5 - -5").getValue();

Double doubleValue = (Double) spelParser.parseExpression("3.5 * 2.5 / -5").getValue();
```

ii.xviii Spring Expression Language (SpEL) (o)

- Referencias del Lenguaje: Asignación de valores

- Es posible asignar valores a objetos mediante SpEL, mediante los métodos setValue y getValue del objeto Expression.

```
EvaluationContext teslaContext teslaContext = new StandardEvaluationContext(tesla);

spelParser.parseExpression(“name").setValue(teslaContext, “Nikolai");

spelParser.parseExpression( “name = ‘Nikolai'").getValue(teslaContext, String.class);
```

ii.xviii Spring Expression Language (SpEL) (p)

- Referencias del Lenguaje: Tipos (Clases Java)

- Mediante el operador especial T, es posible especificar la instancia de una clase (tipo).

- Permite acceder a constantes o métodos estáticos de una clase.

```
Class dateClass = spelParser.parseExpression("T(java.util.Date)").getValue(Class.class);

boolean trueValue = parser.parseExpression(
                    "T(java.math.RoundingMode).CEILING < T(java.math.RoundingMode).FLOOR") .
                                                                    getValue(Boolean.class);
```

ii.xviii Spring Expression Language (SpEL) (q)

- Referencias del Lenguaje: Constructores (Clases Java)

- Es posible invocar constructores mediante new; únicamente es necesario utilizar el nombre calificado de la clase a instanciar.

```
Inventor tesla = spelParser.parseExpression(
        "new org.certificatic.Inventor('Nikola Tesla', 'Serbian' )").getValue(Inventor.class);

spelParser.parseExpression(
        "members.add(new org.certificatic.Inventor('Albert Einstein', 'German'))").
                getValue(societyContext);
```

ii.xviii Spring Expression Language (SpEL) (r)

- Referencias del Lenguaje: Variables

- Es posible usar variables (literales u objetos) en contexto de la expresión a evaluar.
- Para usar variables, es necesario agregarlas al contexto.

```
Inventor tesla = new Inventor("Nikola Tesla", "Serbian");
StandardEvaluationContext context = new StandardEvaluationContext(tesla);
context.setVariable("newName", "Mike Tesla");

String teslaName = (String) parser.parseExpression("name = #newName").getValue(context);
```

ii.xviii Spring Expression Language (SpEL) (s)

- Referencias del Lenguaje: Funciones

- Es posible extender SpEL mediante el registro de funciones/métodos estáticos que pueden ser llamados en una expresión.
- Para usar los métodos, es necesario registrarlos en el contexto.

```
StandardEvaluationContext context = new StandardEvaluationContext();
context.registerFunction("reverseString",
        StringUtils.class.getDeclaredMethod("reverseString", new Class[] { String.class }));

String helloWorldReversed = parser.parseExpression("#reverseString('hello')").
                                    getValue(context, String.class);
```

ii.xviii Spring Expression Language (SpEL) (t)

- Referencias del Lenguaje: Referencias a Beans

- Es posible hacer referencia a beans de Spring mediante expresiones.
- Para hacer referencia a un bean es necesario hacerlo por su nombre o identificador con el prefijo **@**.
- Es necesario configurar en el contexto de expresiones un bean encargado de proveer los beans mediante la implementación de **BeanResolver**.

ii.xviii Spring Expression Language (SpEL) (u)

- Referencias del Lenguaje: Referencias a Beans

- **BeanResolver**: Interface que define un método para resolver un bean especifico.

- Es posible ubicar un bean de Spring por su nombre mediante el método getBean(beanName) del ApplicationContext.

ii.xviii Spring Expression Language (SpEL) (v)

- Referencias del Lenguaje: Referencias a Beans

- Para inyectar el ApplicationContext en un bean es necesario implementar la interface **ApplicationContextAware**.

- **ApplicationContextAware**: Similar a la interface ResourceLoaderAware que provee un método para inyectar el ResourceLoader, la interface ApplicationContextAware provee de un método para inyectar el ApplicationContext.

ii.xviii Spring Expression Language (SpEL) (w)

- Referencias del Lenguaje: Referencias a Beans

```
public class MyBeanResolver implements BeanResolver, ApplicationContextAware {
    private ApplicationContext applicationContext;

    public Object resolve(EvaluationContext context, String beanName) throws AccessException {
        return applicationContext.getBean(beanName);
    }

    public void setApplicationContext(ApplicationContext applicationContext) throws
                                                                    BeansException {
        this.applicationContext = applicationContext;
    }
}
```

ii.xviii Spring Expression Language (SpEL) (x)

- Referencias del Lenguaje: Referencias a Beans

```
StandardEvaluationContext springContext;
ApplicationContext applicationContext;

applicationContext = new AnnotationConfigApplicationContext(ApplicationConfig.class);

springContext = new StandardEvaluationContext();
springContext.setBeanResolver( applicationContext.getBean(MyBeanResolver.class) );

Inventor tesla = spelParser.parseExpression("@teslaBean").getValue(springContext,
Inventor.class);
```

ii.xviii Spring Expression Language (SpEL) (y)

- Referencias del Lenguaje: Operador Ternario
- Operador Ternario (if –then – else).
- Definido de la siguiente forma <expresion> ? <then> : <else>

```
String falseString = parser.parseExpression( "false?'trueExp' :'falseExp'").
                                                    getValue(String.class);
```

ii.xviii Spring Expression Language (SpEL) (z)

- Referencias del Lenguaje: Operador Elvis

- Operador Elvis (?:).

- Definido por la expresión: <objeto>?:<if-null-valor-default>

- Es una definición corta del operador ternario de la forma:
  name != null ? name : “Unknown”

```
String name = parser.parseExpression("name?:'Unknown'").getValue(String.class);
```

ii.xviii Spring Expression Language (SpEL) (a')

- Referencias del Lenguaje: Operador Safe Navigation

- Operador Safe Navigation (?.).

- Definido por la expresión: <objeto>?.<método>

- Provee de un mecanismo para navegación segura evitando accesos a atributos o métodos de objetos nulos. Evita NullPointerException.

- Es una definición corta del operador ternario de la forma:
  name != null ? name : "Unknown"

ii.xviii Spring Expression Language (SpEL) (b')

- Referencias del Lenguaje: Operador Safe Navigation
- Operador Safe Navigation (?.).
- Definido por la expresión: <objeto>?.<método>
- Es una definición corta del operador ternario de la forma:
  objeto != null ? objeto.metodo() : null

```
String city = parser.parseExpression("placeOfBirth?.city").
                                                getValue(context, String.class);
```

ii.xviii Spring Expression Language (SpEL) (c')

- Referencias del Lenguaje: Template Expressions

- En configuración XML definir una expresión se realiza mediante la forma:
  #{ expresión }

- Para definir expresiones mediante Expression no es requerido, pues por default se espera una expresión.

```
Expression spelExpression = spelParser.parseExpression("'Ivan García'");
String stringExpression = (String) spelExpression.getValue();
```

ii.xviii Spring Expression Language (SpEL) (d')

- Referencias del Lenguaje: Template Expressions

- El API SpEL puede recibir no sólo una expresión, sino varias con un formato específico.

- La definición explicita de una expresión en el API SpEL se define mediante la implementación de la interface ParserContext.

- **ParserContext**: La interface ParserContext define el prefijo y sufijo que SpEL parseará y lo que exista dentro de este prefijo y sufijo será la expresión.

ii.xviii Spring Expression Language (SpEL) (e')

- Referencias del Lenguaje: Template Expressions

- La implementación por default de ParserContext es la clase **TemplateParserContext**, la cual define que por default el prefijo y sufijo de una expresión en SpEL es de la forma:

  #{ expresión }

- Con la definición explicita de una expresión podemos definir expresiones template, con más de una expresión SpEL con un formato específico.

ii.xviii Spring Expression Language (SpEL) (f')

- Referencias del Lenguaje: Template Expressions

- Ejemplo:

```
springContext.setVariable("name", "Ivan García");

String greeting = spelParser.parseExpression("Hi #{ #name +' '+'you''re' } awesome!",
                        new TemplateParserContext()).getValue(springContext, String.class);

Assert.assertEquals("Hi Ivan García you're awesome!", greeting);
```

ii.xviii Spring Expression Language (SpEL)

a. Introducción
b. Evaluación de expresiones
c. Referencias del lenguaje
Práctica 21. API SpEL

ii.xviii Spring Expression Language (SpEL). Práctica 21. (a)

- Práctica 21. API SpEL

- Implementar inyección de dependencias por medio de configuración XML.
- Implementar inyección de dependencias por medio de @Anotaciones.
- Implementar los diferentes operadores soportados por SpEL.
- Manipular objetos directamente sobre SpEL.
- Evaluar distintas expresiones mediante la agregación de variables y funciones que extiendan la funcionalidad de SpEL.

Resumen de la lección

ii.xviii Spring Expression Language (SpEL)

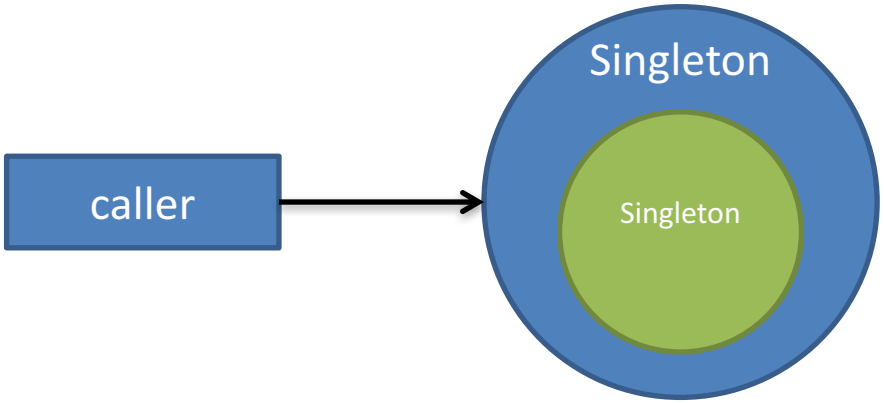
- Conocimos a profundidad las características de SpEL.
- Realizamos inyección de dependencias mediante la aplicación de expresiones SpEL tanto por configuración XML como por @Anotaciones.
- Manipulamos objetos y beans mediante expresiones SpEL.

Esta página fue intencionalmente dejada en blanco.

Bonus. Inyección de Métodos (a)

- Inyección de métodos

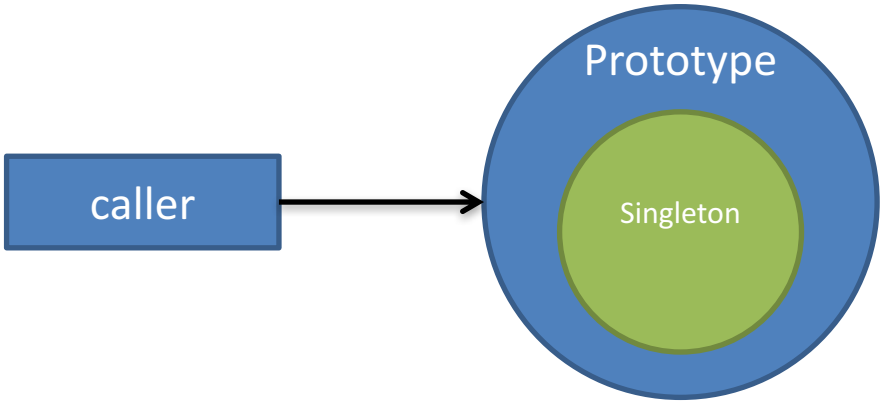
- ¿Qué sucede cuando un bean singleton se inyecta en otro bean singleton?



Bonus. Inyección de Métodos (b)

- Inyección de métodos

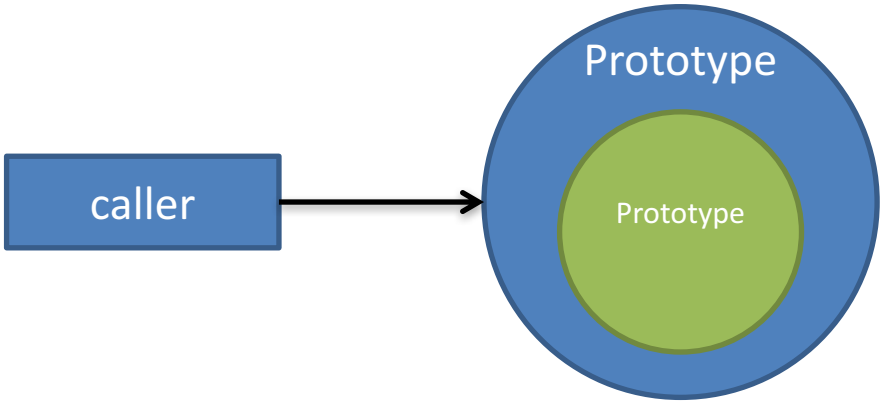
- ¿Qué sucede cuando un bean singleton se inyecta en otro bean protoype?



Bonus. Inyección de Métodos (c)

- Inyección de métodos

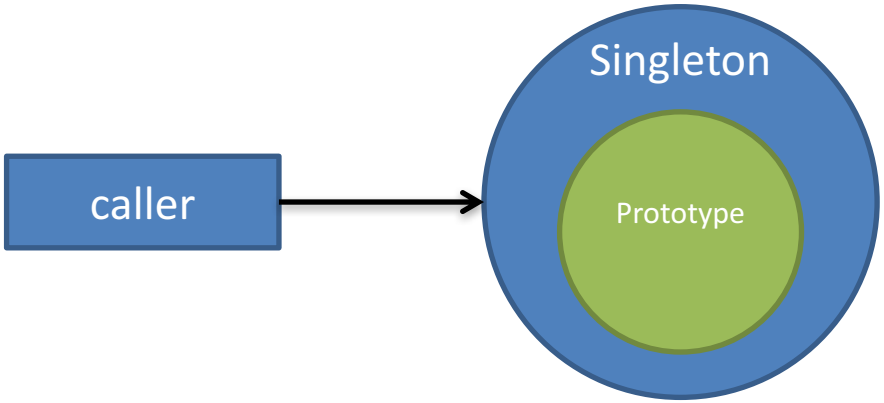
- ¿Qué sucede cuando un bean prototype se inyecta en otro bean protoype?



Bonus. Inyección de Métodos (d)

- Inyección de métodos

- ¿Qué sucede cuando un bean prototype se inyecta en otro bean singleton?



Bonus. Inyección de Métodos (e)

- Inyección de métodos

- Una vez que el singleton se construye, sus dependencias (prototypes) se construyen e inyectan, no se vuelven a instanciar sus dependencias pues el bean que las contiene es singleton y éste se instancia una única vez.

- ¿Solución?
  - ApplicationContextAware
  - Inyección de Métodos

Bonus. Inyección de Métodos (f)

- Solución: ApplicationContextAware

```
public class CommandManager implements ApplicationContextAware {
    private ApplicationContext applicationContext;
    public Object process(Map commandState) {
        Command command = createCommand();
        Command instance command.setState(commandState);
        return command.execute();
    }
    protected Command createCommand() {
        return this.applicationContext.getBean("command", Command.class);
    }
    public void setApplicationContext( ApplicationContext applicationContext) throws BeansException {
        this.applicationContext = applicationContext;
    }
}
```

Bonus. Inyección de Métodos (g)

- Solución: Inyección de Métodos, configuración XML.

```
public abstract CommandManager {
    public Object process(Map commandState) {
        Command command = createCommand();
        command.setState(commandState);
        return command.execute();
    }
    protected abstract Command createCommand();
}
```

```
<bean id="command“
        class="AsyncCommand"
        scope="prototype“ />

 <bean id="commandManager"
        class="CommandManager">
   <lookup-method
        name="createCommand"
        bean="command"/>
</bean>
```

Bonus. Inyección de Métodos (h)

- Solución: Inyección de Métodos, configuración por anotaciones.

```
public abstract CommandManager {
    public Object process(Map commandState) {
        Command command = createCommand();
        command.setState(commandState);
        return command.execute();
    }

    @Lookup("command")
    protected abstract Command createCommand();
}
```

```
<bean id="command"
        class="AsyncCommand"
        scope="prototype" />

<bean id="commandManager"
        class="CommandManager" />
```

Bonus. Inyección de Métodos (i)

- Práctica 21. Method Injection.

- Implementar inyección de métodos para simular la inyección de beans prototype sobre bean singleton.

- Complender el uso de <lookup-method> y de la anotación @Lookup.

Esta página fue intencionalmente dejada en blanco.

Bonus. Reemplazo de Métodos (a)

- Reemplazo de métodos

- ¿Será posible reemplazar la implementación de un método de un bean en tiempo de ejecución?
  - Implementando MethodReplacer

Bonus. Reemplazo de Métodos (b)

- Reemplazo de métodos

- MethodReplacer

```
public interface MethodReplacer {
    Object reimplement(Object obj, Method method, Object[] args)
                                                    throws Throwable;
}
```

Bonus. Reemplazo de Métodos (c)

- Implementación método a reemplazar (a)

```
public class SportCar {
        private String model;

        public int run() {
                int mills = 120;
                log.info("Sport car model: {} is running at: {}...", model, mills);
                return mills;
        }
        …
}
```

Bonus. Reemplazo de Métodos (d)

- Implementación MethodReplacer (b)

```
public class RunReplacer implements MethodReplacer {

    @Override
    public Object reimplement(
            Object obj, Method method, Object[] args) throws Throwable {
            int mills = 200;

            SportCar sportCar = (SportCar) obj;
            log.info("{} is running at: {} mph...", sportCar.getModel(), mills);

            return mills;
    }
}
```


Bonus. Reemplazo de Métodos (e)

- Configuración MethodReplacer (c)

```
<!-- Method replacement -->
<bean
        class="org.certificatic.spring.core.practica21.methodreplacement.bean.SportCar">
        <property name="model" value="CLA 250 sport" />
        <replaced-method name="run" replacer="runReplacer" />
</bean>

<bean id="runReplacer"
        class="org.certificatic.spring.core.practica21.methodreplacement.replacer.
                                                                RunReplacer" />
```

Bonus. Reemplazo de Métodos (f)

- Práctica 21. Method Replacement.

- Implementar reemplazo de métodos de beans en tiempo de ejecución. Útil cuando requerimos cambiar la implementación de un objeto que no es posible vía código.

- Comprender el uso de <replaced-method> y de la interface MethodReplacer.

Esta página fue intencionalmente dejada en blanco.

Bonus. Property Editors (a)

- Property Editors

- Spring Framework utiliza PropertyEditor extensivamente para convertir texto a objetos y viceversa, donde sea requerido.

- El concepto de PropertyEditor es parte de la especificación JavaBeans.

https://docs.oracle.com/javase/8/docs/technotes/guides/beans/

Bonus. Property Editors (b)

- Property Editors

- En específico, existen principalmente tres casos de uso donde es necesario utilizar PropertyEditor.
  - Definir beans a través de texto.
  - Inyectar dependencias de beans a través de texto.
  - Obtener a partir de solicitudes HTTP representaciones de objetos a través de texto.

Bonus. Property Editors (c)

- Property Editors

- Para definir un PropertyEditor es necesario extender **PropertyEditorSupport** y registrarlo en **CustomEditorConfigurer**.

- Al extender **PropertyEditorSupport** es necesario sobre-escribir los métodos:
  - String getAsText()
  - void setAsText(String text)

Bonus. Property Editors (d)

- Property Editors.

- Implementando: String getAsText()
  - Apoyándose del método heredado **this.getValue()**, éste método debe devolver la representación en texto del Objeto determinado.

- Implementando: void setAsText(String text)
  - Éste método debe convertir el parámetro ”**string**” de entrada del método setAsText(text), al Objeto en cuestión y “**setearlo**” al PropertyEditor mediante el método heredado **this.setValue(object)**.

Bonus. Property Editors (e)

- Property Editors

```
@Data
@Builder
public class CreditCard {
        private String rawCardNumber;
        private Integer bankIdNo;
        private Integer accountNo;
        private Integer checkCode;
}
```

Bonus. Property Editors (f)

- Property Editors

```
public class CreditCardEditor extends PropertyEditorSupport {

    @Override
    public String getAsText() {

        CreditCard creditCard = (CreditCard) this.getValue();

        return creditCard == null ? "" : creditCard.getRawCardNumber();

    }

    …
```

ii. Spring Core - ii.xviii Spring Expression Language (SpEL)

Bonus. Property Editors (g)

- Property Editors

```
...
@Override
public void setAsText(String text) throws IllegalArgumentException {
        if (StringUtils.isEmpty(text)) {
                this.setValue(null);
        } else {
                CreditCard.CreditCardBuilder creditCardBuilder =
                                                        CreditCard.builder();
                // Extraer, parsear el string y construir el objeto en cuestión.
                ...
                this.setValue(creditCardBuilder.build());
        }
}
}
```

Bonus. Property Editors (h)

- Práctica e. Property Editors.

- Implementar definición de beans e inyección de dependencias de tipos complejos, Clases, a base de texto.

- Extender la clase PropertyEditorSupport para dar soporte a PropertyEditors personalizados.

- Registrar los PropertyEditors en el contenedor de IoC de Spring.

Esta página fue intencionalmente dejada en blanco.