Partner ORACLE UNIVERSITY

Certifica TIC Beyond Certification

INICIO

Desarrollo de Aplicaciones Empresariales con Spring Framework Core 4

ISC. Ivan Venor García Baños

ATC Advanced Training Consultants

SCRUMstudy Authorized Training Partner

MEX1CO FIRST

SE Secretaría de Economía

CANIETI www.canieti.org

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 4
vi. Fundamentos Spring MVC y Spring Security
vii. Fundamentos Spring REST

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

ii.vi Definición heredada de Beans (Bean Template)

Práctica 9. Bean Templates

ii. Spring Core (d)

ii.vii Bean post processors

Práctica 10. Bean post processors

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. Spring Core (d)

ii.x Namespace p, c y util

Tarea 2. Ejemplo namespace p, c y util

ii.xi Autowiring

a. ByName

b. ByType

Práctica 13. Autowiring

Trabajo de Integración 1. Convertidor número letra configuración XML.

Práctica 14. Convertidor número letra configuración XML

ii. Spring Core (e)

ii.xii Configuración con @Anotaciones

a. Namespae 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.xiv Component-scan

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

Práctica 17. Component-scan y estereotipos

ii. Spring Core (f)

ii.xv Anotaciones JSR 330

a. @Inject y @Named

Práctica 18. @Inject y @Named

ii.xvi Spring Java Config

a. @Configuration, @Bean e @Import

Práctica 19. @Configuration, @Bean e @Import

Tarea 3. Migración Convertidor número letra configuración por @Anotaciones

ii. Spring Core (g)

ii.xvii Resources

a. Conceptos

b. Tipos de Resources

Práctica 20. Resources

ii.xviii Spring Expression Language (SpEL)

a. Conceptos

Práctica 21. API SpEL

b. Evaluación de expresiones

Práctica 22. SpEL configuración XML

Práctica 23. SpEL configuración @Anotaciones

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

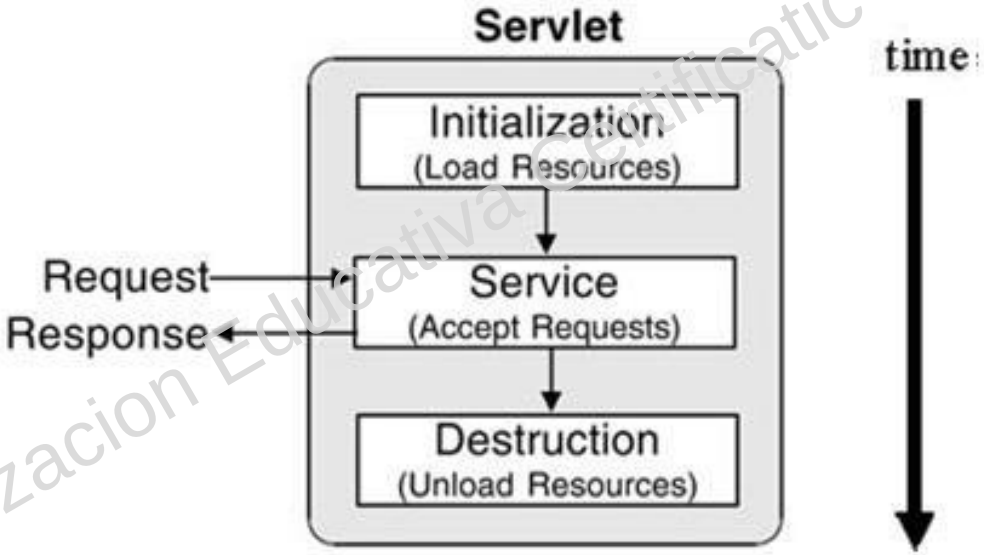
ii.i Inversión de Control (a)

- 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:



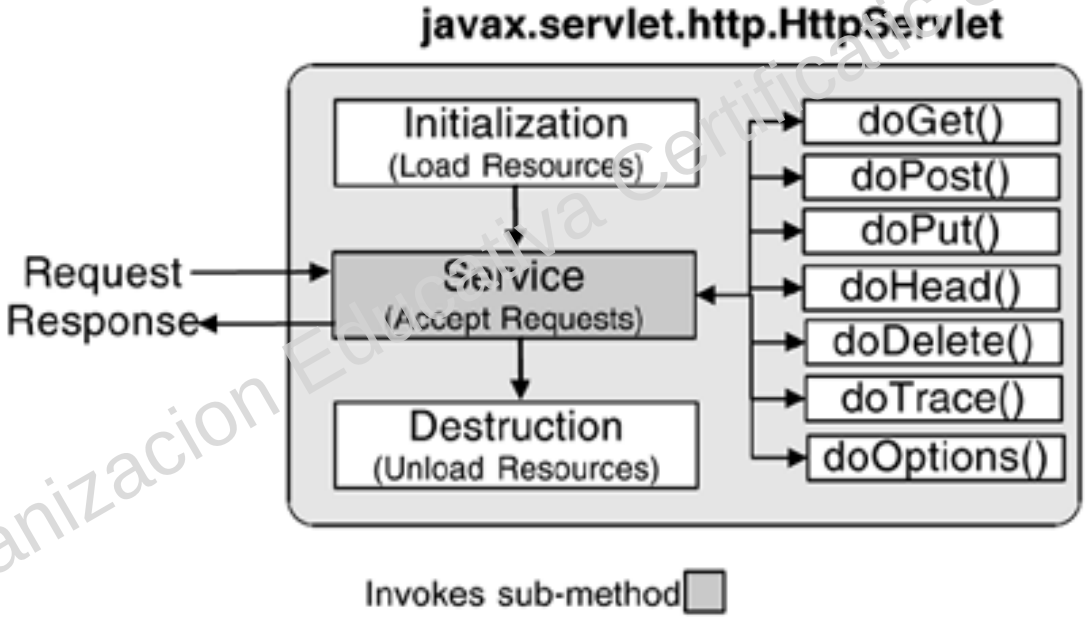
ii.i Inversión de Control (c)

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

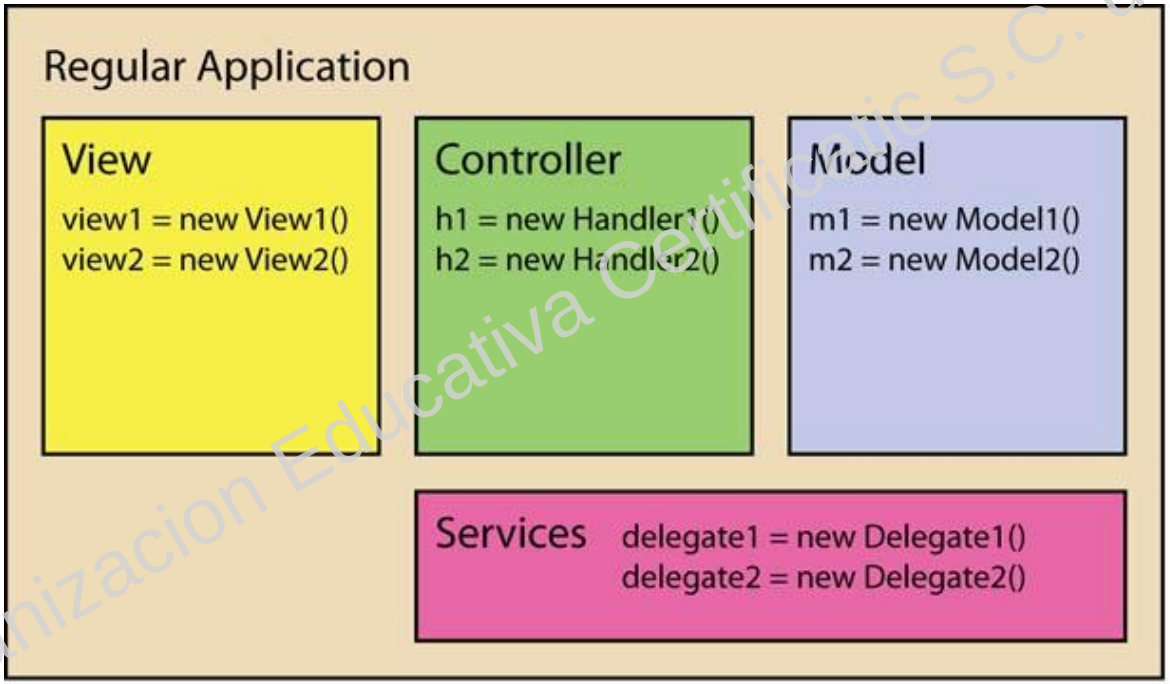
- La Inversión de Control especifica respuestas deseadas a sucesos o solicitudes concretas, dejando que algún framework, librería, entidad o arquitectura externa lleve a cabo las acciones de control que tengan que ejecutarse, en el orden necesario y para el conjunto de sucesos que tengan que ocurrir.

ii.i Inversión de Control (d)

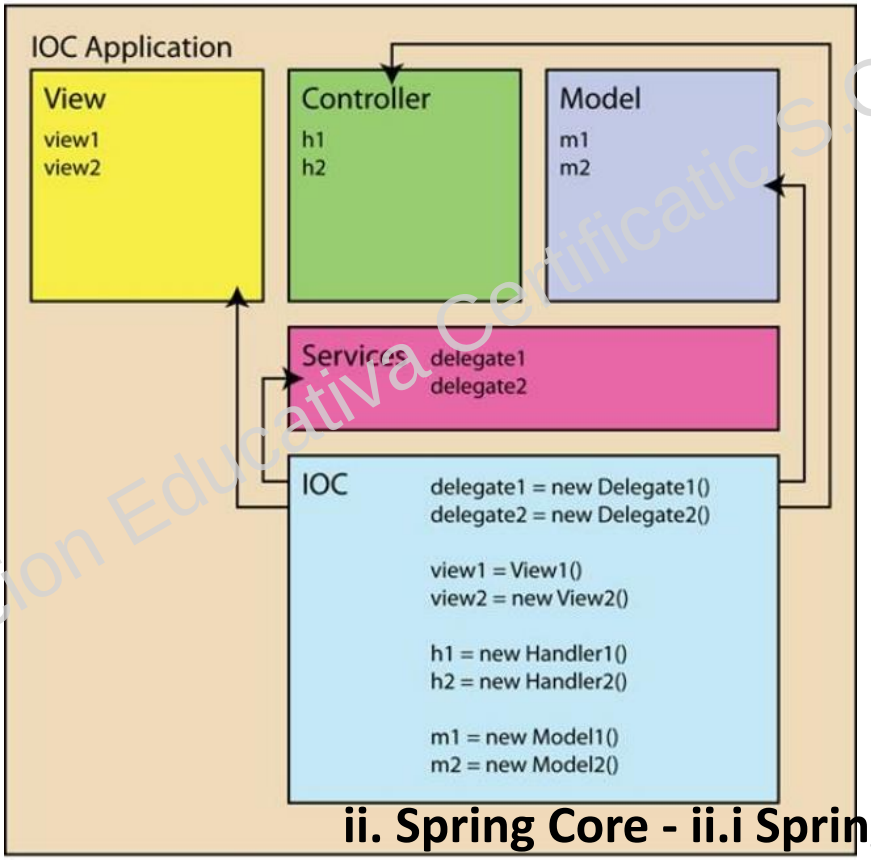
- IoC ejemplo:



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.

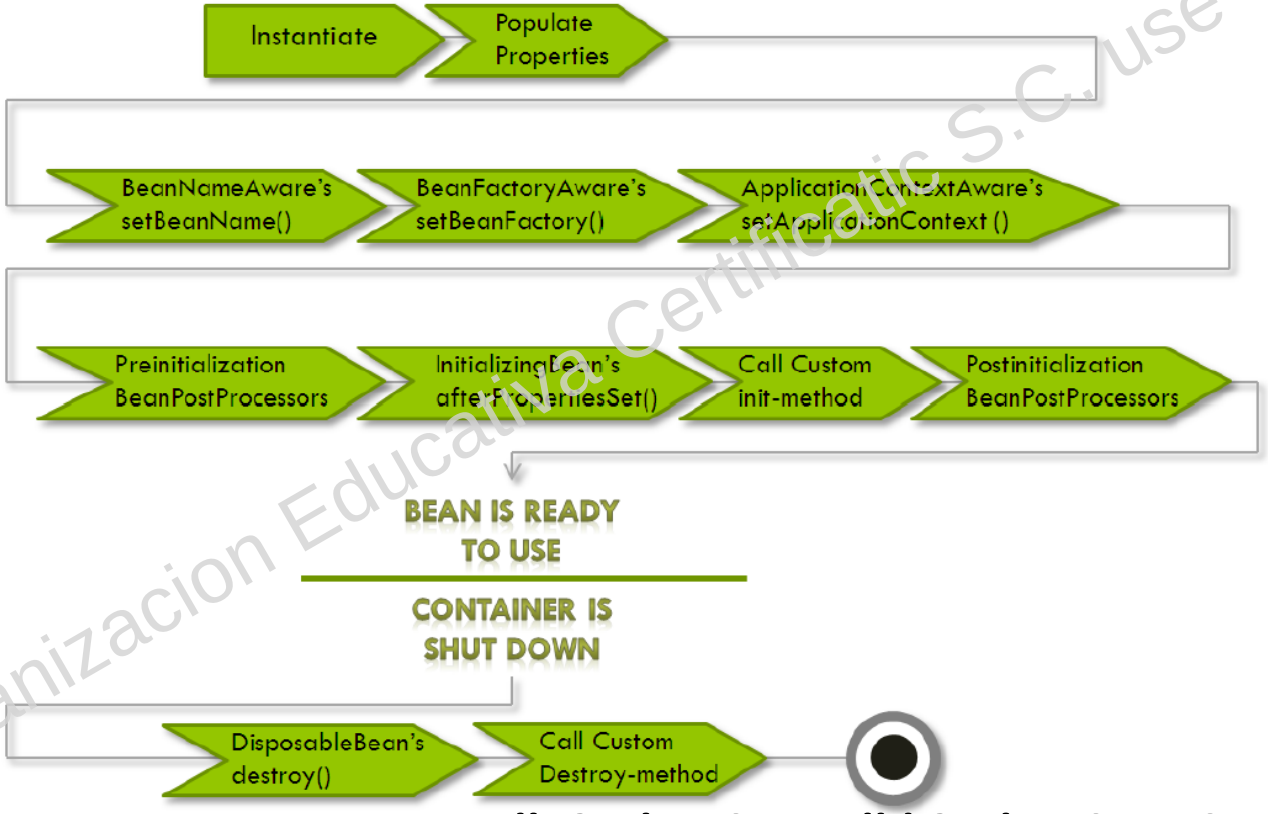
ii.i Inversión de Control (h)

- 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)



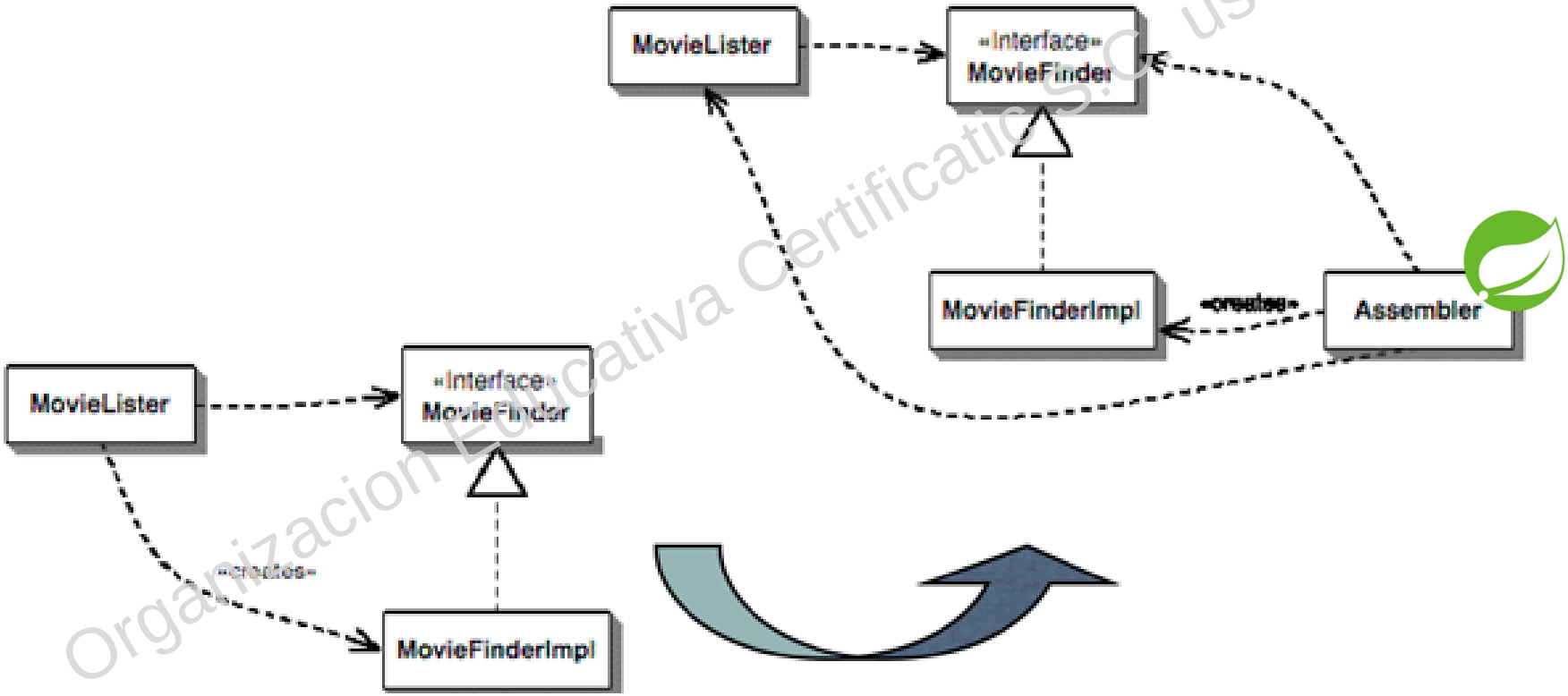
ii.i Inversión de Control (j)

- 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)

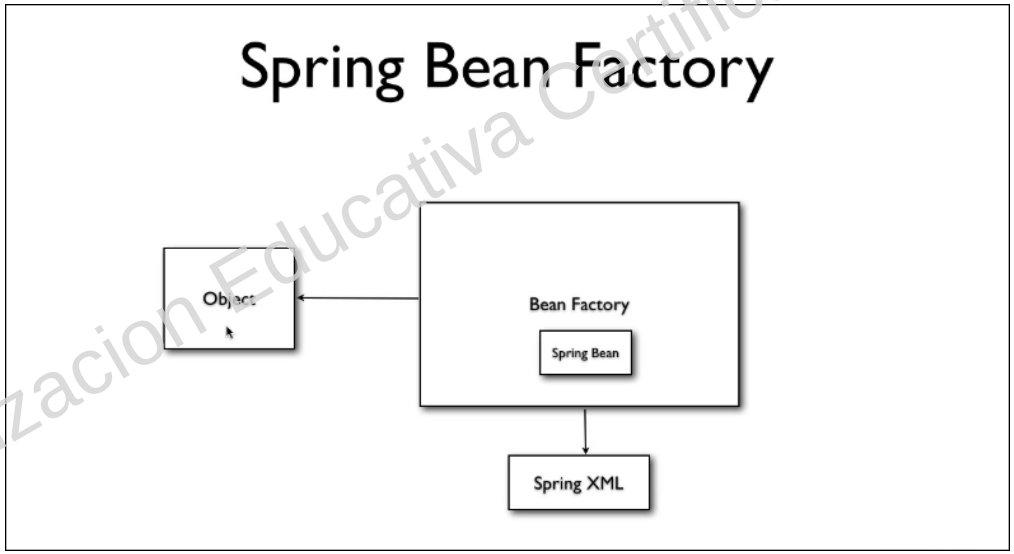


ii.i Inversión de Control (I)

- 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.

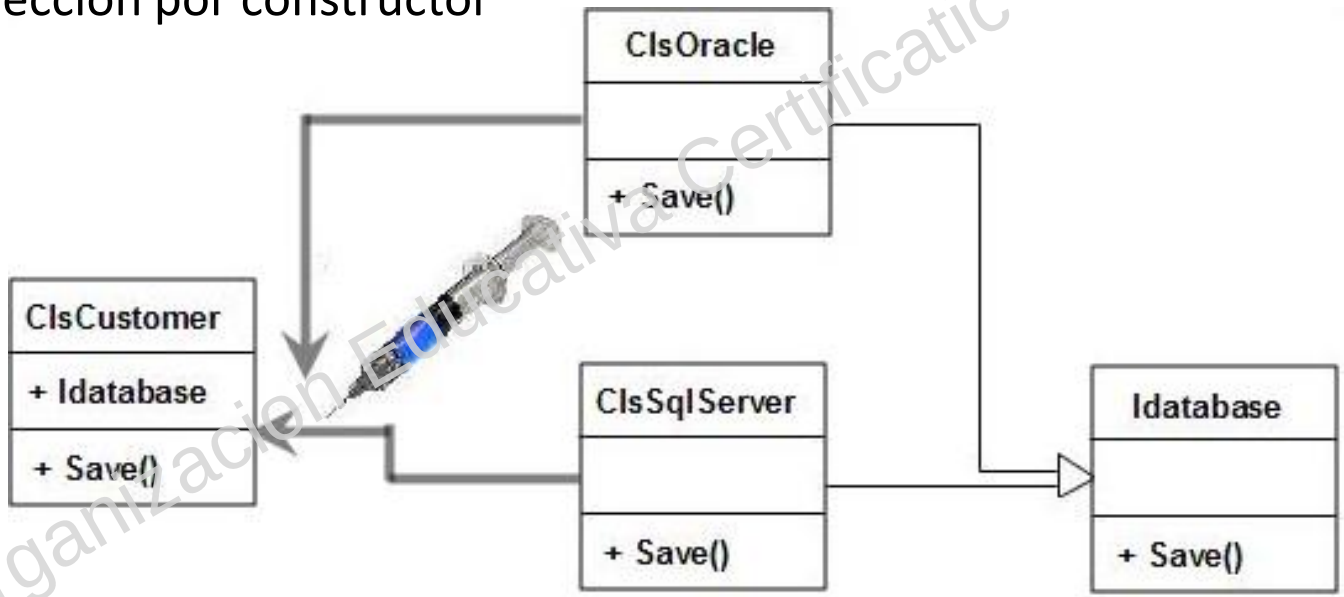
ii.i Inyección de Dependencias (b)

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


```
public class Customer
{
public Logger Log;
public Customer(Logger obj)
{
Log = obj;
}
```

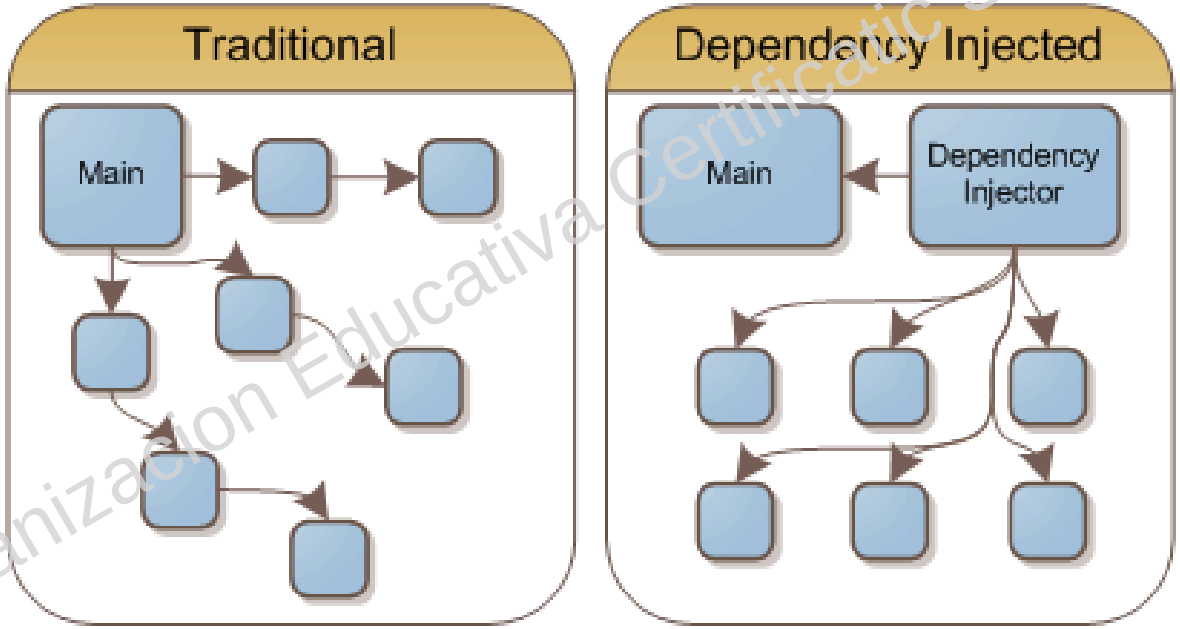
ii.i Inyección de Dependencias (c)

- Inyección por setter
- Inyección por constructor



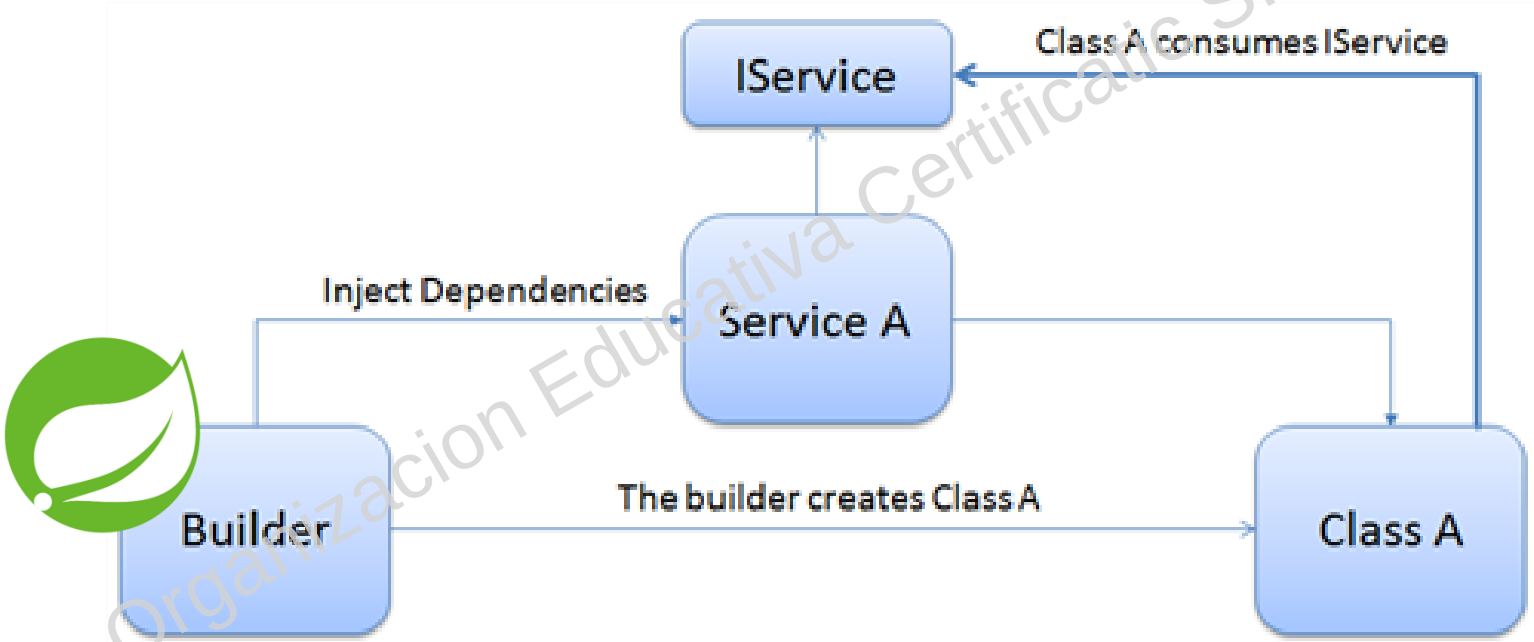
ii.i Inyección de Dependencias (d)

- Tradicional (Sin IoC) vs IoC + DI



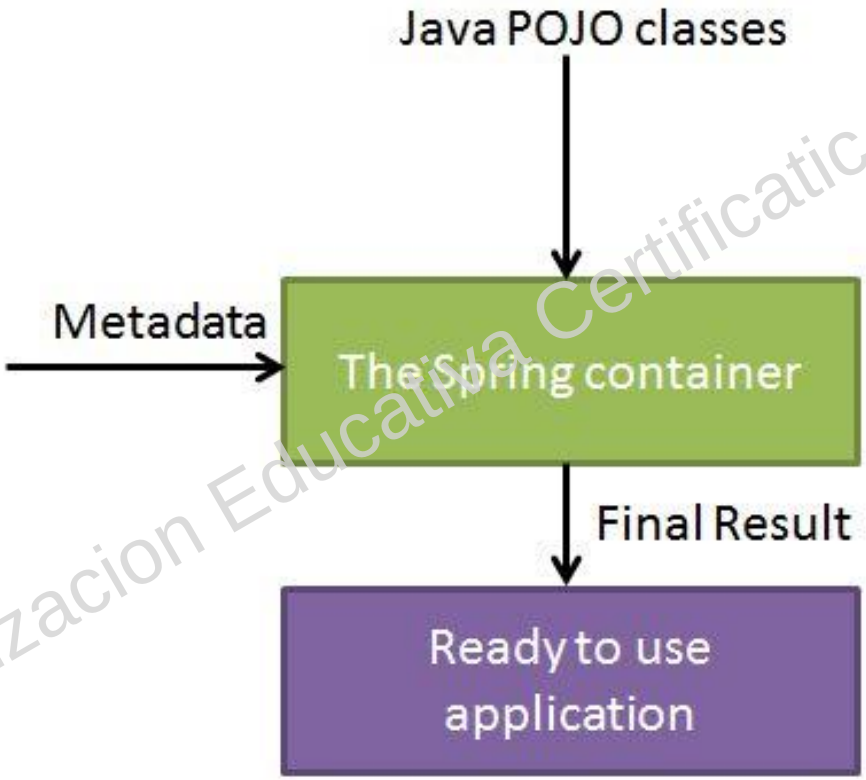
ii.i Inyección de Dependencias (e)

- IoC + DI



ii.i Inyección de Dependencias (f)

- Spring DI

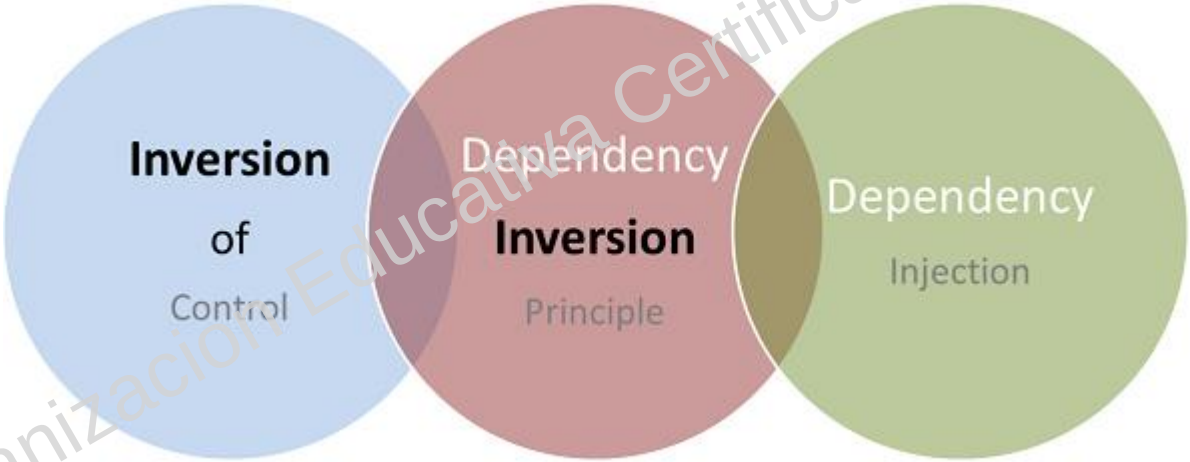


ii.i Spring Core Conceptos

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

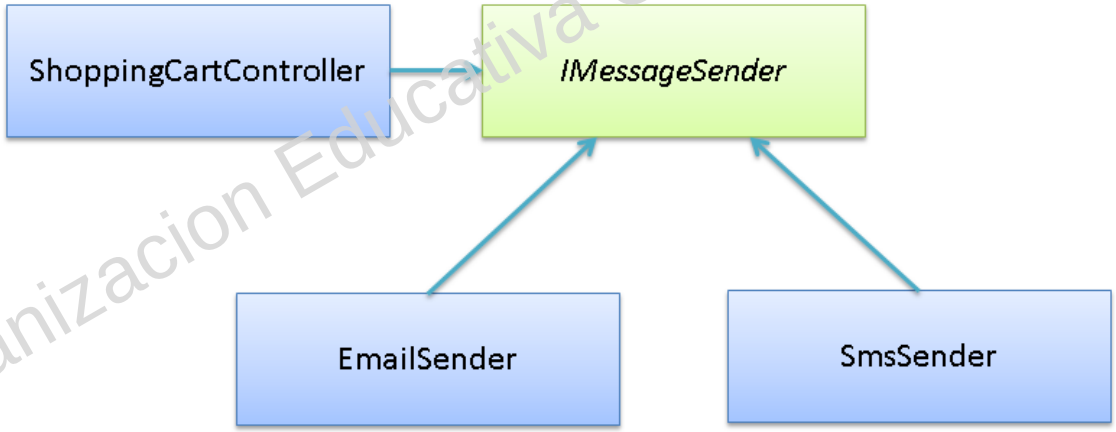
ii.i Inversión de Dependencias (a)

- Inversión de Control no es Inyección de Dependencias



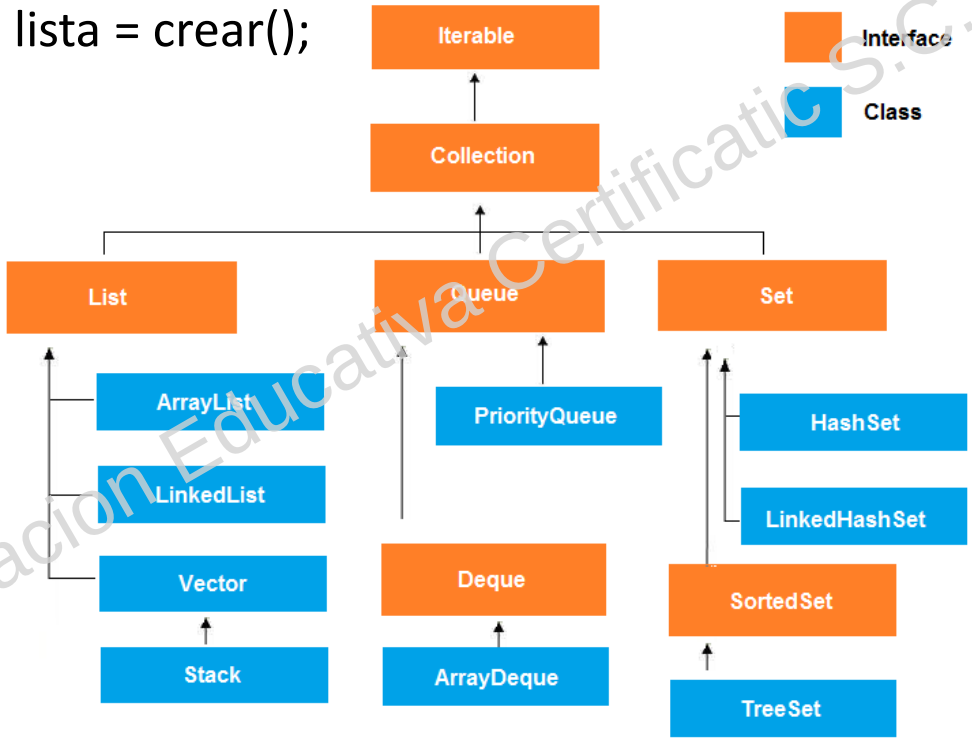
ii.i Inversión de Dependencias (b)

- 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.



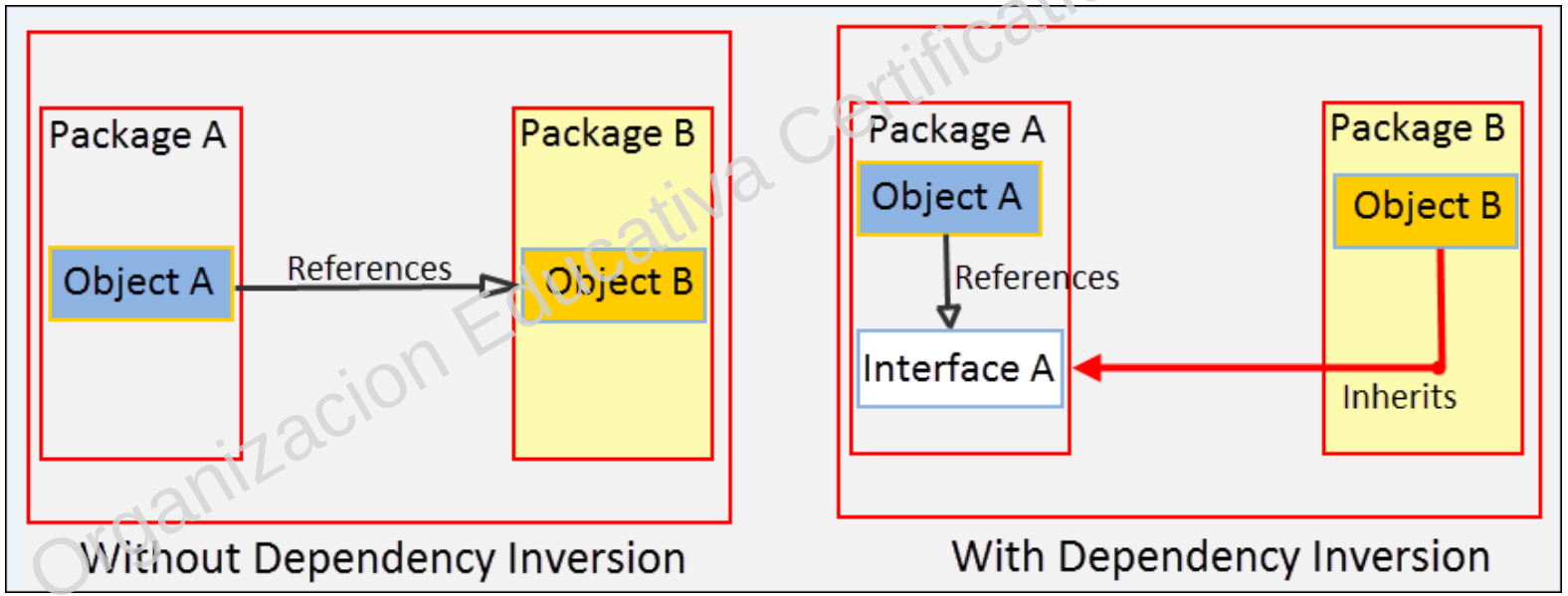
ii.i Inversión de Dependencias (c)

- List<String> lista = crear();



ii.i Inversión de Dependencias (d)

- 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)

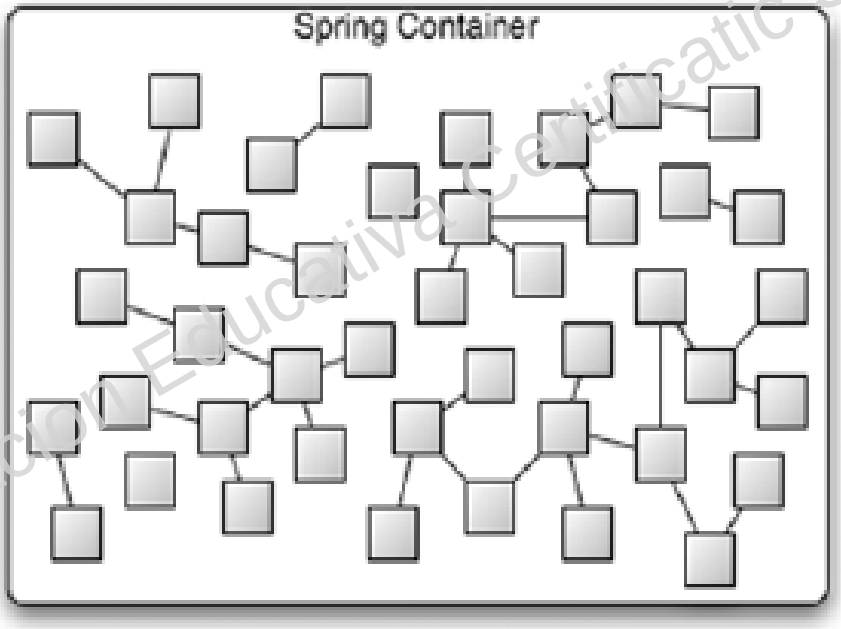
- Spring Core:
  - Provee contenedor de IoC e DI (contenedor de beans)
  - Implementa patrones de diseño
  - Objeto BeanFactory (Factory pattern)
  - IoC nos evita gestionar el ciclo de vida de los objetos
  - Genera Singletons por default (no thread safe)

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)

- Spring IoC Container:



ii.ii Contenedor de IoC (d)

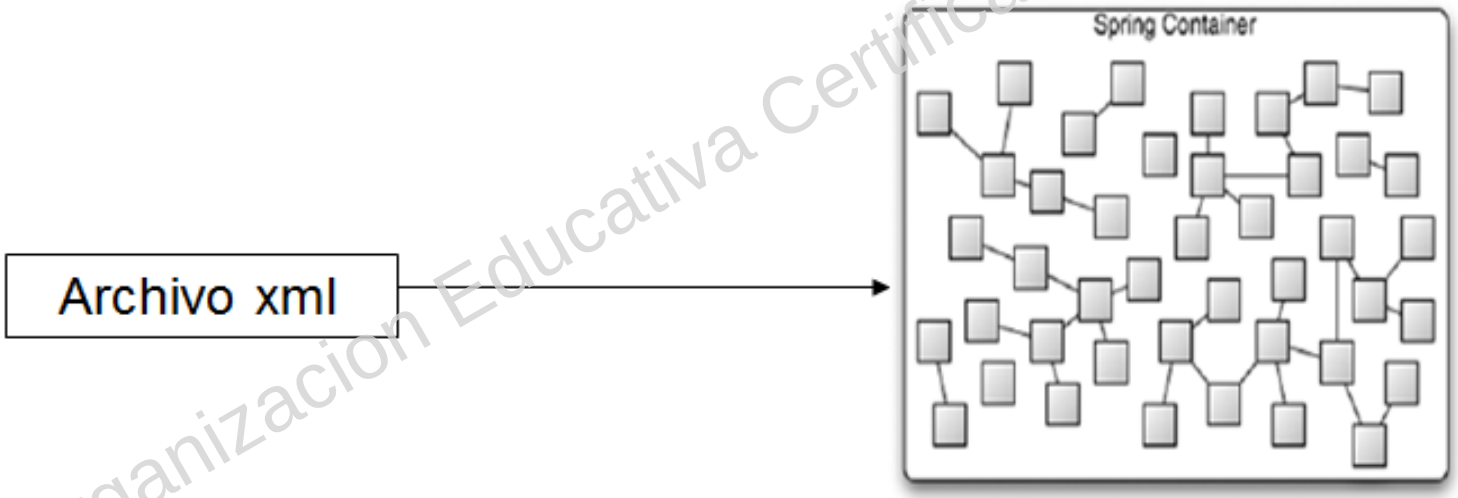
- El contenedor ejecutará las instrucciones configuradas para saber sobre qué Clases deberá crear instancias, configurar y preparar para su uso.

- 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

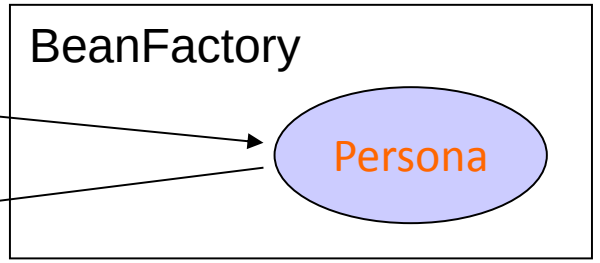
c. Tipos de configuración de Beans

ii.ii Bean Factory (a)

```
BeanFactory 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 (b)

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 Bean Factory (c)

- Práctica 2. Hola Mundo Spring Framework

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

ii.ii Bean Factory (d)

- Se recomienda utilizar el contenedor BeanFactory para aplicaciones de peso ligero tales como dispositivos móviles o aplicaciones APPLET ya que el peso y volumen de la información en este tipo de aplicaciones es crítico.

ii.ii Contenedor de IoC

a. BeanFactory

Práctica 2. Hola Mundo Spring Framework

b. ApplicationContext

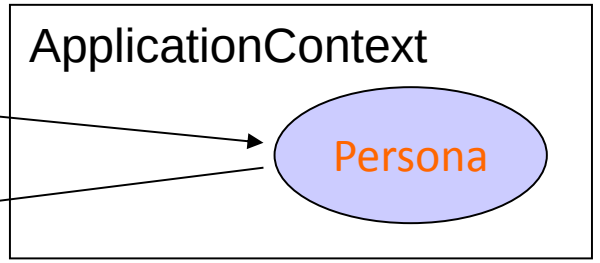
c. Tipos de configuración de Beans

ii.ii ApplicationContext (a)

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

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

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



ii.ii ApplicationContext (b)

- 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).

ii.ii ApplicationContext (c)

- Práctica 2. Hola Mundo Spring Framework

- Implementar Main utilizando ApplicationContext

ii.ii ApplicationContext (d)

- Contenedor de IoC de Spring

ii.ii Contenedor de IoC

a. BeanFactory

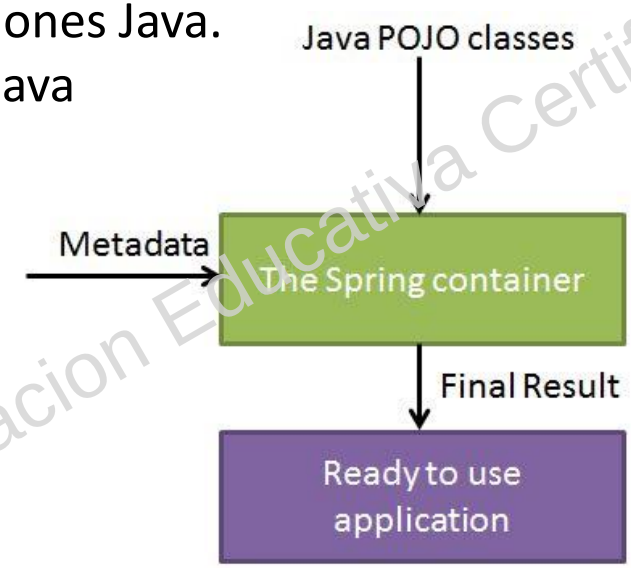
Práctica 2. Hola Mundo Spring Framework

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.
  - Clases Java



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

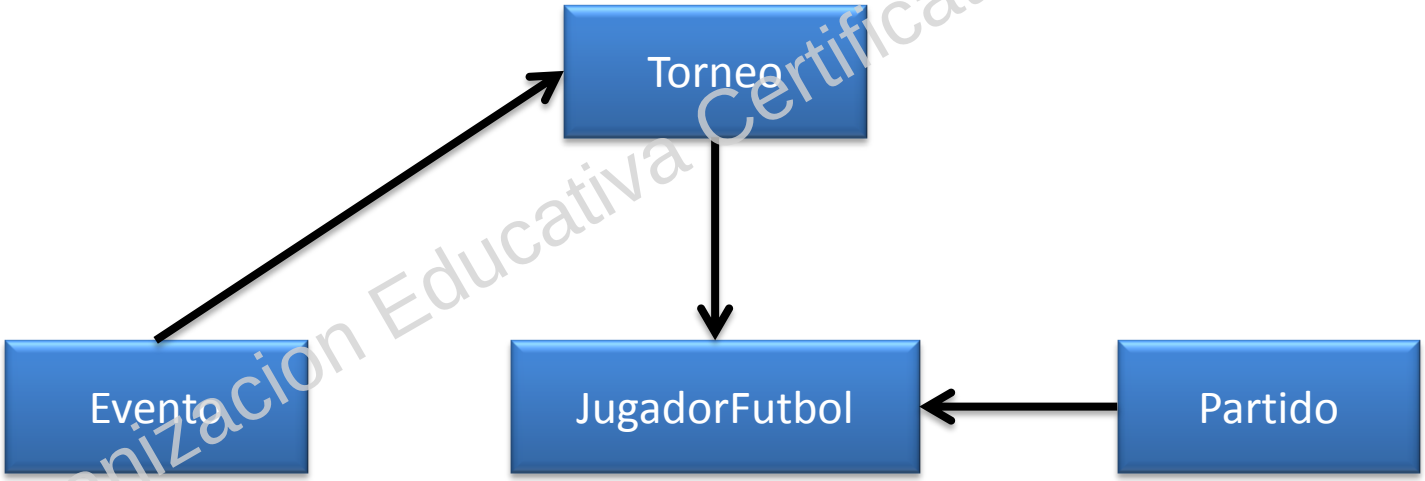
Práctica 3. Inyección de Dependencias Jugadores

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

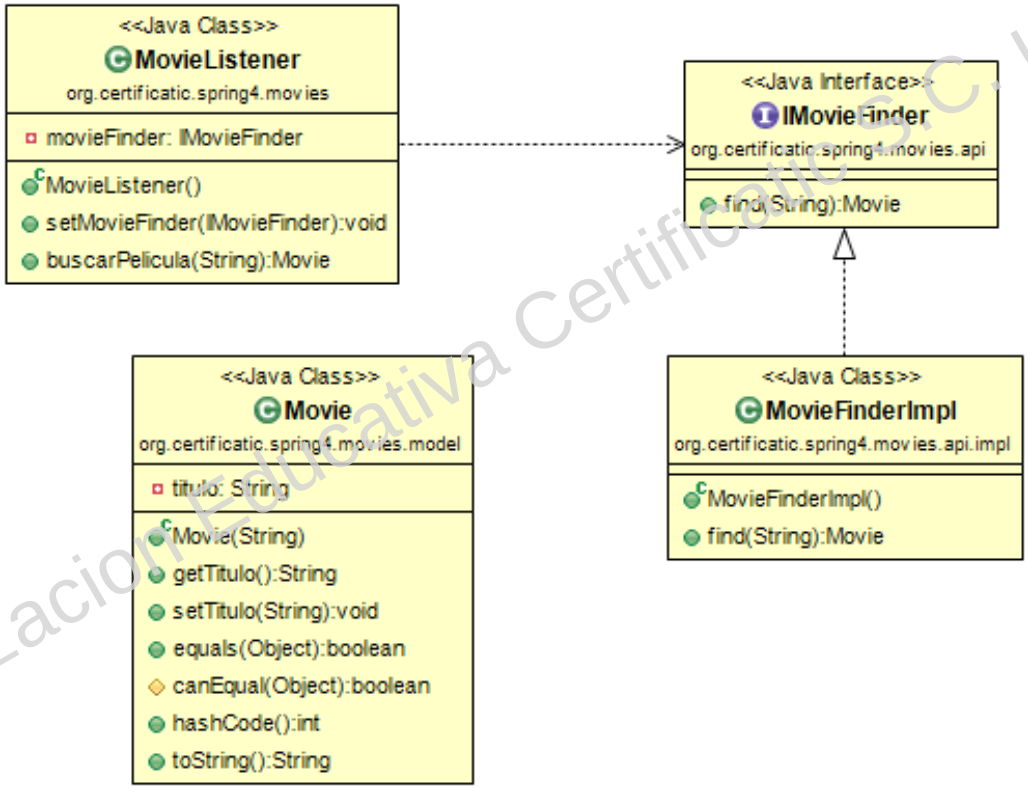
Tarea 1. Implementación Notification Service

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

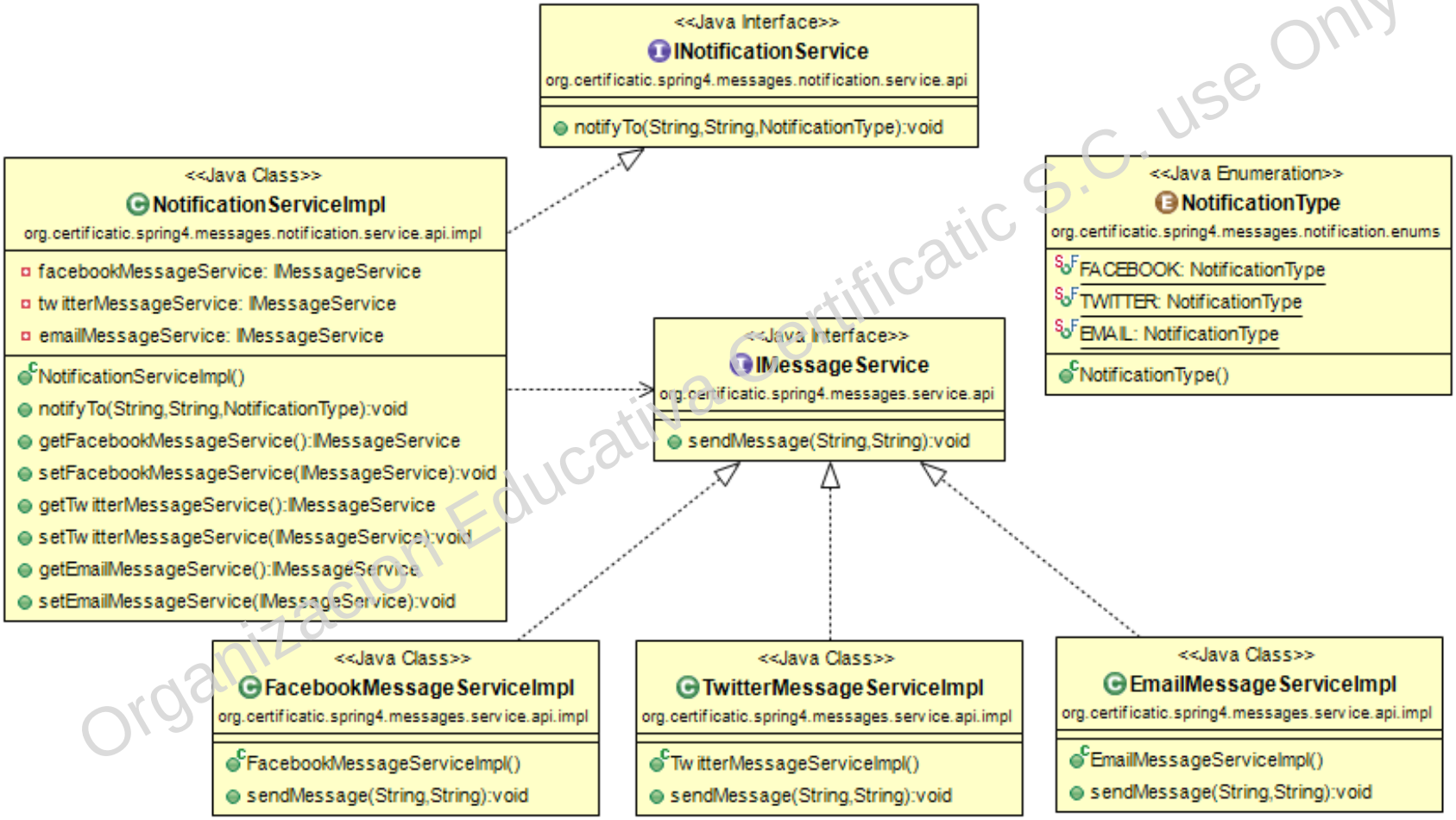
Práctica 4. Inyección de Dependencias Movie Finder

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.



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 5 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.

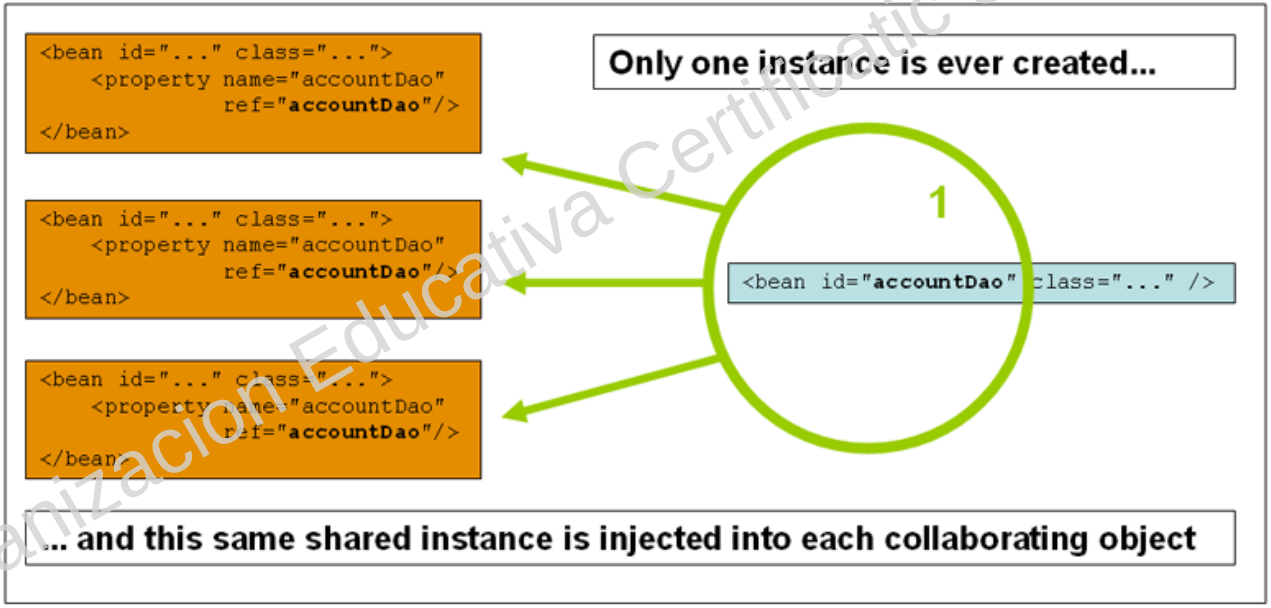
ii.iv Bean Scopes (d)

- 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 (e)

- Singleton



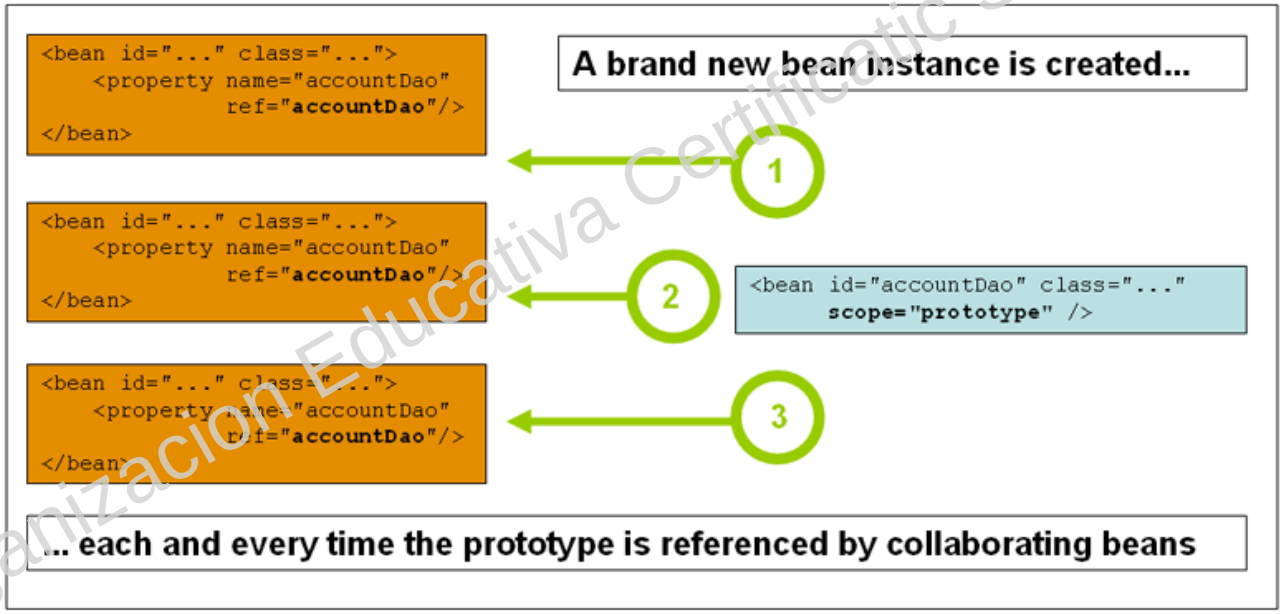
ii.iv Bean Scopes (f)

- 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 (g)

- Prototype



ii.iv Bean Scopes

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

ii.iv Bean Scopes (h)

- 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 (i)

- 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

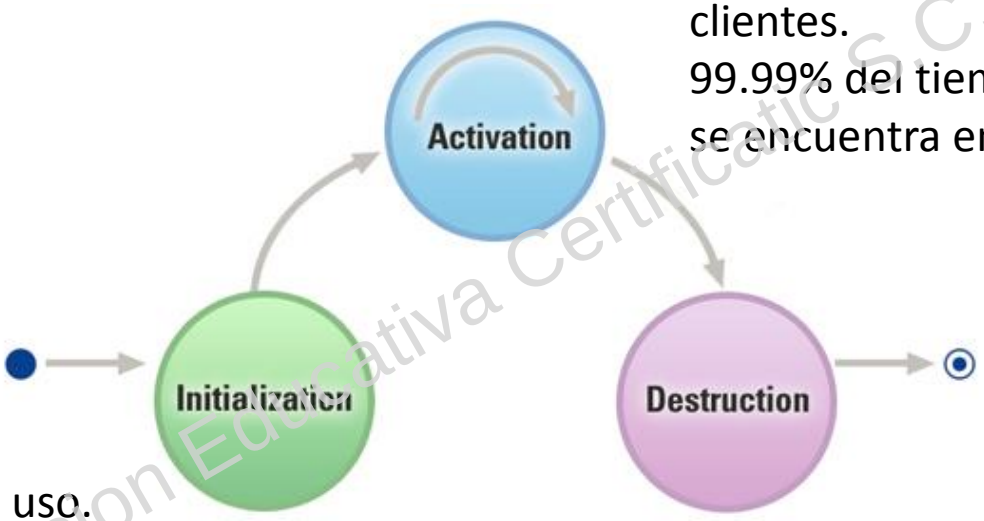
- Conocer de manera general el ciclo de vida de los Beans.
- Configurar la inicialización y destrucción de beans.
- Conocer la inicialización lazy
- Implementar la personalización durante la inicialización y destrucción de beans.
- Implementar inicialización lazy de beans.

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

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.

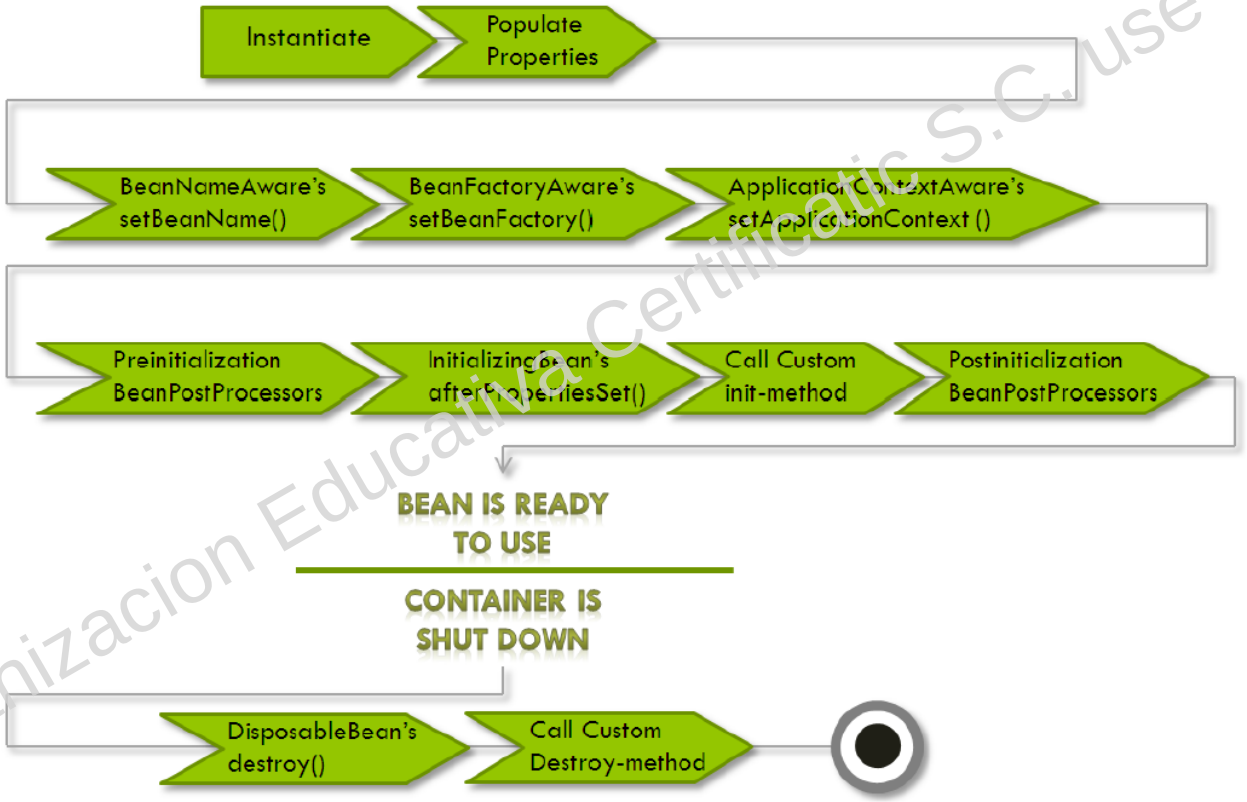
ii.v Ciclo de vida de Beans (b)

- 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-3.0.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

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-3.0.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

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

a. Inicialización y Destrucción
b. Inicialización Lazy
c. Factory Method
Práctica 6. Init – Destroy
Práctica 7. Lazy Beans
Práctica 8. Factory Method

ii.v Ciclo de vida de Beans (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

ii.v Ciclo de vida de Beans (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

ii.v Ciclo de vida de Beans (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.

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.

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 Bean Post Processors

Objetivos de la lección

ii.vii Bean Post Processors

- Implementar Bean Post Processors para configurar beans.
- Conocer la forma de ordenamiento de procesamiento de Bean Post Processors.

ii.vii Bean Post Processors

Práctica 10. Bean Post Processors

ii.vii Bean Post Processors (a)

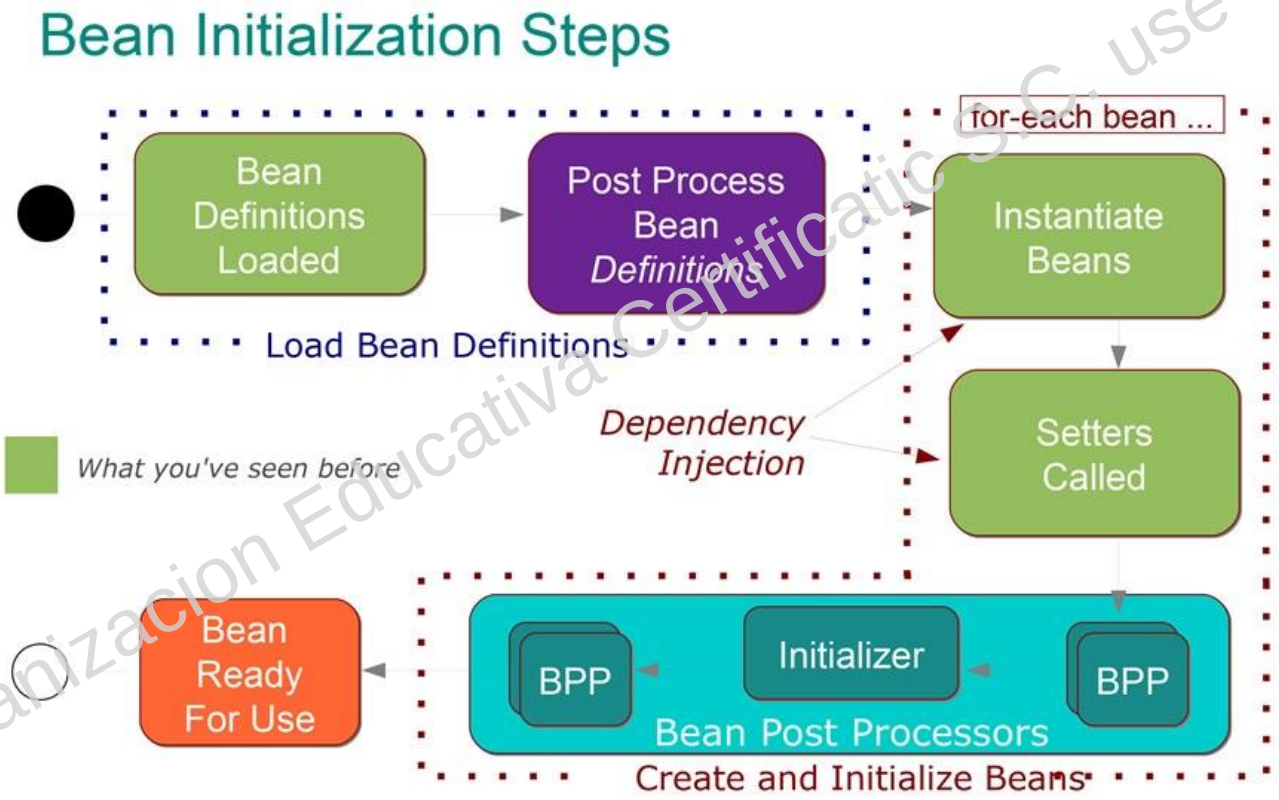
- Los Bean Post Processors son una interfaz que define métodos callback que permite personalizar la inicialización de Beans.

- Los Bean Post Processors se aplican en el ciclo de vida de todos los 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 Bean Post Processors (b)



ii.vii Bean Post Processors (c)

- Implementar interface BeanPostProcessor

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

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

ii.vii Bean Post Processors (d)

- Implementar interface Ordered

```
int getOrder();
```

ii.vii Bean Post Processors (e)

- Práctica 10. Bean Post Processors

- Implementar Bean
- Implementar múltiples Bean Post Processors.

Resumen de la lección

ii.vii Bean Post Processors

- Comprendimos más en detalle el ciclo de vida de los beans, específicamente la fase de inicialización.
- Implementamos Bean Post Processors, los cuales proveen un mecanismo de personalización en la configuración de beans.
- Aplicamos ordenamiento de ejecución a los Bean Post Processors.

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 (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 (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:
  - Propenso a error.
  - No todos los IDEs proveen funcionalidad “autocomplete”.
  - Para producción, es necesario desplegar utilizando Java Debugging Tables (debug mode opción -g).

- Utilización de Namespace c, no se recomienda.
  - Leer más: http://springinpractice.com/2012/05/07/springs-constructor-namespace-is-a-bad-idea/

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 (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 (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;
}
```

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 y arreglos.

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

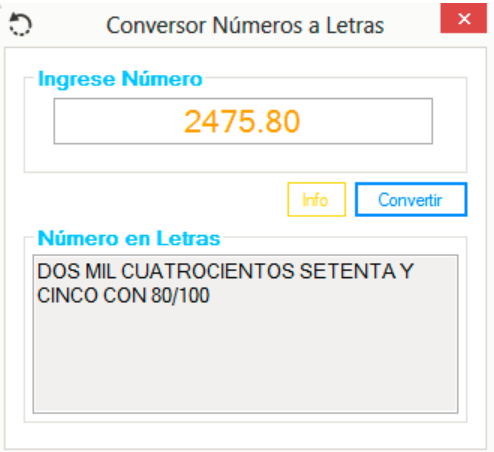
- Conocer el concepto de Autowiring (auto-hilado).
- Implementar los diferentes tipos de Autowiring para Inyectar Dependencias.

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 en dos o más idiomas diferentes, 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.

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

Esta página fue intencionalmente dejada en blanco.

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