

# **Spring: IOC**

Antonio Espín Herranz

# Contenidos

- Conceptos básicos.
- Patrón IOC.
- Dependencias.
- Alcance (Scope).
- Personalización de beans.
- Herencia.
- Extensión del contenedor.
- ApplicationContext.
- Anotaciones.
- Escaneo classpath.
- JavaConfig.
- SpEL.
- Conectar colecciones.

# Conceptos básicos

# Conexión básica de beans

- En una aplicación basada en Spring, los objetos de la aplicación vivirán dentro del contenedor de Spring.
- Spring, controla su ciclo de vida: los instancia, los utiliza y los destruye.
- Hay dos contenedores de beans dentro de Spring (representados por las clases):
  - **BeanFactory**: (cuando hay **pocos recursos**).
  - **ApplicationContext**

# BeanFactory

- Representa una fábrica de objetos, no sólo despacha objetos de un tipo, si no, de muchos.
- También crea asociaciones entre los beans que ha creado.
- Hay varias implementaciones de BeanFactory dentro de Spring.
  - La mas utilizada **XmlBeanFactory**, las definiciones de beans viene en un fichero XML.

# BeanFactory

- Ejemplo:

```
BeanFactory factoria = new XmlBeanFactory(new  
    FileSystemResource("C:\beans.xml"));
```

- Para recuperar un bean de la factoría le damos el ID del bean que queremos recuperar.

```
MyBean miBean = (MyBean)factoria.getBean("myBean");
```

- Este bean estaría definido dentro del fichero.

# ApplicationContext

- Nos ofrece mas posibilidades que la otra clase.
  - Soporte para **i18n**.
  - **Abrir recursos de archivos** de una forma genérica, como imágenes.
  - Puede **publicar eventos** en beans que estén registrados como receptores.
- Dentro de las **implementaciones de ApplicationContext**, las más utilizadas:
  - **ClassPathXmlApplicationContext**: abre desde un fichero XML. El fichero está situado en una ruta de clase.
  - **FileSystemXmlApplicationContext**: abre una definición de contexto desde un archivo XML.
  - **XmlWebApplicationContext**: Abre definiciones de contexto desde un archivo XML contenido en una app Web.

# ApplicationContext

- Ejemplos:
  - `ApplicationContext contexto = new FileSystemXmlApplicationContext("C:\foo.xml");`
  - `ApplicationContext contexto = new ClassPathXmlApplicationContext("foo.xml");`
  - // Varios ficheros:**
  - `ApplicationContext context = new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});`
- Para recuperar el bean, igual que con BeanFactory, con el método **getBean()**.

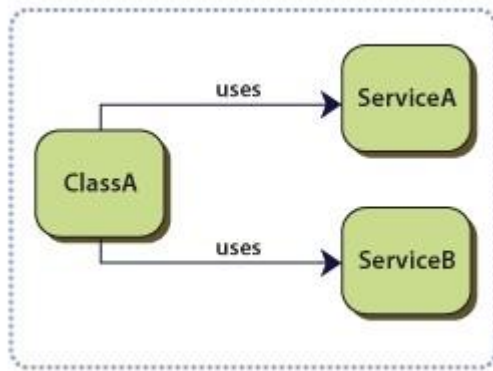


# Patrón IOC

Inversion of Control

# IoC : Inversion Of Control

- La Inversión de Control:
  - Es un patrón de diseño pensado para permitir un menor acoplamiento entre componentes de una aplicación y fomentar así el reuso de los mismos y facilitar las pruebas.



**La claseA depende de las clases ServiceA, ServiceB.**  
Los problemas que esto plantea son:

- Al reemplazar o actualizar las dependencias, se necesita cambiar el código fuente de la clase A.
- Las implementaciones concretas de las dependencias tienen que estar disponibles en tiempo de compilación.
- Las clases son difíciles de testear aisladamente porque tienen directas definiciones a sus dependencias
- Las clases tienen código repetido para crear, localizar y gestionar sus dependencias.

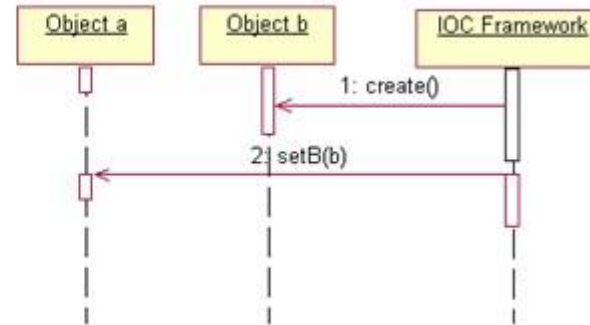
# IoC : Inversion Of Control

- El control de cómo un objeto A obtiene la referencia de un objeto B es invertido. **El objeto A no es responsable de obtener una referencia al objeto B** sino que es el Componente Externo el responsable de esto. Esta es la base del patrón IoC.
- El patrón IOC aplica un principio de diseño denominado **principio de Hollywood** (*No nos llames, nosotros te llamaremos*).

# IoC : Inversion Of Control

Ejemplo:

```
public class A{  
    private B b;  
  
    public A(){  
    }  
  
    public setB(B b){  
        this.b=b;  
    }  
}
```



- A necesita una referencia a B, pero no necesita saber como debe crear la instancia de B, solo quiere una referencia a ella.
- B puede ser una interface, clase abstracta o clase concreta.
- Antes de que una instancia de A sea usada, necesitara una referencia a una instancia de B.
- Aquí no hay un alto acoplamiento entre A y B, ambas pueden cambiar internamente sin afectar a la otra. Los detalles de cómo se crea y gestiona un objeto B, no es decidido en la implementación de A. Es un framework IoC quien usa un método como setB() de A para inyectar luego a un objeto B.
- **ESTA TÉCNICA SE LLAMA INYECCIÓN DE DEPENDENCIAS.**

# Dependencias

# Inyección de Dependencias (DI)

- La inyección de dependencias favorece el **bajo acoplamiento** entre la clases.
- ***El objetivo es crear el menor acoplamiento posible, esto facilita: las pruebas, la integración y la reutilización.***
- La inyección de dependencias: hace que la clase que necesita algo no lo cree, ni la busque en algún lugar, lo obtiene por inyección: mediante un **constructor** un método **set**.

# Inyección de Dependencias (DI)

- La inyección de dependencias se puede definir como un término utilizado para describir el desacoplamiento entre la implementación de un objeto y la construcción de otro objeto del cual depende.



# Inyección de Dependencias (DI)

- **Tenemos que evitar cosas del estilo:**

```
public class Clase1{  
    private Clase2 atributo1;  
  
    public Clase1(){  
        this.atributo1 = new Clase2();  
    }  
}  
  
public class Clase2 { ... }
```

- Con esto estamos creando un **acoplamiento** entre la clase (Clase y Clase2).
- La Clase1 es la encargada de instanciar la Clase2.



# Inyección de Dependencias (DI)

- Esto se puede evitar, disponiendo de un **contenedor de Beans** (en nuestro caso Spring).
  - **Spring** es un contenedor o factoría de beans que **gestiona el ciclo de vida** de estos: (los crea, los destruye, etc.)
- Los objetos se crean 1 vez y ya están disponibles y se pueden compartir.
- Cuando los objetos se han creado (**se inyectan** a los objetos que los necesitan) mediante métodos **set** o **constructores**.

# Inyección de Dependencias (DI)

```
public class Clase1 {  
    private Clase2 atributo1;  
  
    public Clase1(){ }  
  
    public void setAtributo1(Clase2 atributo1){  
        this.atributo1 = atributo1;  
    }  
}  
  
public class Clase2 { // IMPLEMENTACIÓN }
```

# Inyección de Dependencias (DI)

- Declaramos nuestros beans en el contexto de la aplicación. Utilizamos programación declarativa.

```
< bean id="atributo1" class="org.mipaquete.bean.Clase2">
```

```
< /bean>
```

```
<!-- Definimos un bean de la Clase2 -->
```

```
< bean id="atributo" class="org.mipaquete.bean.Clase1">
```

```
< property name=" atributo1" ref="atributo1" />
```

```
< /bean>
```

```
<!-- El bean de la Clase1 tiene una propiedad que referencia al objeto de la Clase 2 -->
```

# Ejemplo I

- **Con dependencias:**

```
public class Coche {  
    private Radio radio;  
    private Motor motor;  
  
    public Coche(){  
        this.radio = new Radio();  
        this.motor = new Motor();  
    }  
}  
  
public class Radio { ... }  
public class Motor { ... }
```

- *La construcción de un Coche implica la construcción de una Radio y un Motor.*
- **Evitar ESTA FORMA.**

- **Inyección de Dependencias:**

```
public class Coche {  
    private Radio radio;  
    private Motor motor;  
  
    public Coche(){ }  
    public setRadio(Radio r){  
        this.radio = r;  
    }  
    public setMotor(Motor m){  
        this.motor = m;  
    }  
}  
  
public class Radio { ... }  
public class Motor { ... }
```

- *Cada objeto se creará por separado y luego se inyectan al coche.*

# Ejemplo II

- En el contexto de la aplicación se definirían:

```
< bean id="radio" class="miclases.coche.Radio">< /bean>
```

```
< bean id="motor" class="miclases.coche.Motor">< /bean>
```

```
< bean id="coche" class="misclases.coche.Coche">
```

```
< property name="radio" ref="radio" / >
```

```
< property name="motor" ref="motor" / >
```

```
< /bean>
```

# Inyección de Dependencias

- La inyección de dependencias se puede realizar de dos formas:
  - Mediante **setters**: Disponemos de una clase con las convenciones de los JavaBeans → Constructor por defecto, setters y getters.
  - Mediante **constructores**: Disponemos de un constructor que recibe todos los parámetros.
- Ambos métodos, así como los objetos se declaran, de forma declarativa en el fichero xml.

# Inyección de dependencias mediante **setters**

- Partimos de una clase User (nombre, edad, pais) → JavaBeans.

```
public class User {  
    private String nombre;  
    private int edad;  
    private String pais;  
  
    public User() { }  
    // Método set / get para las 3 propiedades:  
    // Método toString  
}
```


- En un fichero XML, describimos el bean con los valores que va a tener en sus propiedades y le asignamos un ID.
- A partir de este fichero cargaremos el contexto de la aplicación y podremos recuperar el objeto.

# El fichero beans.xml

- El nombre es libre. Se suele llamar también context.xml.

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

```
  <bean id="user" class="ejemplo.inyeccion.dependencias.User" >
    <property name="nombre" value="Eswar" />
    <property name="edad" value="24"/>
    <property name="pais" value="India"/>
  </bean>
```



Las propiedades  
inyectan los valores  
mediante métodos **set**.

```
</beans>
```



# Clase Principal

```
package ejemplo.inyeccion.dependencias;
```

```
import org.springframework.context.ApplicationContext;
```

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
public class Main {
```

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

```
        ApplicationContext contexto = new ClassPathXmlApplicationContext("beans.xml");
```

```
        User usuario = (User)contexto.getBean("user");
```

```
        System.out.println(usuario);
```

```
        usuario.setPais("España");
```

```
        System.out.println(usuario);
```

```
    }
```

```
}
```

# Otra forma de inyectar propiedades en el XML

- Utilizando **p schema**, podemos utilizar los nombres de los atributos de la clase.
- **Incluir el namespace:**  
`xmlns:p="http://www.springframework.org/schema/p"`

```
<bean id="config" class="com.ejemplo.spring.Configuracion"  
      p:bd="empresa3"  
      p:url="jdbc:mysql"  
      p:user="admin"  
      p:pwd="1234" />
```

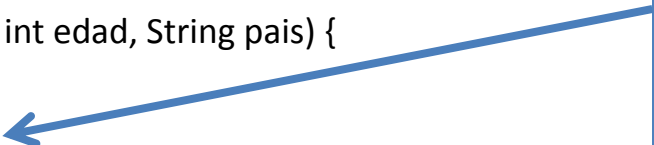
```
public class Configuracion {  
  
    private String url;  
    private String bd;  
    private String user;  
    private String pwd;  
}
```

# Inyección de dependencias mediante constructor

```
package ejemplo.inyeccion.dependencias;
```

```
public class User {  
    private String nombre;  
    private int edad;  
    private String pais;  
  
    public User(String nombre, int edad, String pais) {  
        super();  
        this.nombre = nombre;  
        this.edad = edad;  
        this.pais = pais;  
    }  
  
    @Override  
    public String toString() {  
        return "User [nombre=" + nombre + ", edad=" + edad + ", pais=" + pais + "];"  
    }  
}
```

En este caso no disponemos de métodos get/set. Tenemos un **constructor** que recibe todos los valores.



# El fichero beans.xml

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

    <bean id="user" class="ejemplo.inyeccion.dependencias.User" >
        <constructor-arg name="nombre" value="Eswar" />
        <constructor-arg name="edad" value="24"/>
        <constructor-arg name="pais" value="India"/>
    </bean>

</beans>
```

# Clase Principal

```
package ejemplo.inyeccion.dependencias;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext contexto = new ClassPathXmlApplicationContext("beans.xml");
        User usuario = (User)contexto.getBean("user");

        System.out.println(usuario);

        usuario.setPais("España");
        System.out.println(usuario);
    }
}
```

# Consideraciones

- Tener en cuenta que si queremos dotar a la clase User de algún otro constructor, podemos hacerlo pero teniendo en cuenta lo siguiente:
- Si en la clase tuviéramos estos dos constructores y el fichero xml:

```
User( int age, String country)
{
    this.age=age;
    this.country=country;
}

User(String name, String country)
{
    this.name=name;
    this.country=country;
}
```

```
<bean id="user" class="com.vaannila.User" >
    <constructor-arg value="24"/>
    <constructor-arg value="India"/>
</bean>
```

# Consideraciones II

- Esto puede dar lugar a ambigüedad porque **Spring** considera los dos argumentos de tipo String.
- Para evitar este tipo de problemas podemos indicar el tipo dentro del fichero de configuración. Utilizando el atributo **type**.

```
<bean id="user" class="com.vaannila.User" >  
    <constructor-arg type="int" value="24"/>  
    <constructor-arg type="java.lang.String" value="India"/>  
</bean>
```

# Consideraciones III

- También se pueden dar este tipo de ambigüedades:

```
User(String name, int age)
{
    this.name=name;
    this.age=age;
}

User( int age, String country)
{
    this.age=age;
    this.country=country;
}
```

```
<bean id="user" class="com.vaannila.User" >
    <constructor-arg type="int" value="24"/>
    <constructor-arg type="java.lang.String" value="India"/>
</bean>
```

**OJO**, el orden en que aparecen los parámetros en los constructores no tiene porque ser el mismo a la hora de llamar al **constructor**. Para ello tenemos el atributo **index**.



# Consideraciones IV

- Se resolvería de esta forma:
  - Con index indicamos el orden de los parámetros.

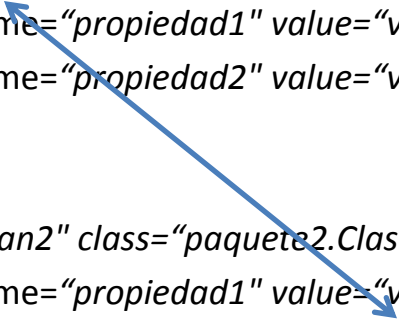
```
<bean id="user" class="com.vaannila.User" >  
  <constructor-arg index="0" type="int" value="24"/>  
  <constructor-arg index="1" type="java.lang.String" value="India"/>  
</bean>
```

# Referencias a otros Beans (depends on)

- Las propiedades de un bean pueden hacer referencias a otros bean.
- Para ello tenemos que utilizar el atributo **ref**, que se corresponderá con el **id** del otro bean.
- Ejemplo:

```
<bean id="bean1" class="paquete1.Clase1">  
<property name="propiedad1" value="valor1"></property>  
<property name="propiedad2" value="valor2"></property>  
</bean>
```

```
<bean id="bean2" class="paquete2.Clase2">  
<property name="propiedad1" value="valor1"></property>  
<property name="propiedad2" ref="bean1"></property>  
</bean>
```



La dependencia más normal es que un bean dependa de otro porque este tiene una propiedad del mismo tipo que el otro bean.

# Referencias a otros Beans (depends on)

- Disponemos del atributo “**depends on**” que se puede utilizar en otras situaciones.
- Hay veces que no tiene porque darse una situación como la página anterior... y podríamos necesitar que ***unos beans fueran inicializados*** antes que otros (se fuerza con el atributo “**depends on**”).

```
<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" />
```

# Inicialización perezosa (**lazy-init**)

- Como normal general todos los beans se instancian cuando se carga el contexto.
- Este comportamiento se puede cambiar y podemos hacer que un bean se instancie cuando reciba la primera petición.
- Para activamos el atributo lazy-init en la declaración del bean.

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>  
<bean name="not.lazy" class="com.foo.AnotherBean"/>
```

# Autowiring (autocableado)

- Spring proporciona una conexión automática de beans, sin necesidad de declararlos en el contexto.
- Tiene **4 tipos de conexión automática**:
  - **byName**: intenta conectar propiedades de un bean con otros que tengan el mismo nombre.
  - **byType**: intenta conectar propiedades de un bean con otros bean que sean del mismo tipo.
  - **constructor**: intenta conectar un constructor con aquellos beans que coincidan con los parámetros del constructor.
  - **autodetect**: intenta primero conectar por constructor y si no por tipo: byType.

# Ejemplos

- Partiendo de las siguientes clases:

```
package com.mkyong.common;

public class Customer
{
    private Person person;

    public Customer(Person person) {
        this.person = person;
    }

    public void setPerson(Person person) {
        this.person = person;
    }
    //...
}
```

```
package com.mkyong.common;

public class Person
{
    //...
}
```

# AutoWiring byName

- En el fichero XML, declaramos:

```
<bean id="customer" class="com.mkyong.common.Customer" autowire="byName" />  
  
<bean id="person" class="com.mkyong.common.Person" />
```

- Coincide el nombre del bean person, con el nombre de la propiedad del bean Customer.
- Utiliza inyección por set...

# AutoWiring byType

- En este caso el tipo del bean person es el mismo que la propiedad del Customer.

```
<bean id="customer" class="com.mkyong.common.Customer" autowire="byType" />  
  
<bean id="person" class="com.mkyong.common.Person" />
```



# AutoWiring constructor

- En este caso el constructor de la clase Customer tiene un parámetro del tipo Person.
- 

```
<bean id="customer" class="com.mkyong.common.Customer" autowire="constructor" />  
  
<bean id="person" class="com.mkyong.common.Person" />
```

# Cuando utilizar IoC

## Inyección de Dependencias

- La inyección de dependencias no debería usarse siempre que una clase dependa de otra, sino más bien es efectiva en situaciones específicas como las siguientes:
  - Inyectar datos de configuración en un componente.
  - Inyectar la misma dependencia en varios componentes.
  - Inyectar diferentes implementaciones de la misma dependencia.
  - Inyectar la misma implementación en varias configuraciones
  - Se necesitan alguno de los servicios provistos por un contenedor.
  - La IoC no es necesaria si uno va a utilizar siempre la misma implementación de una dependencia o la misma configuración, o al menos, no reportará grandes beneficios en estos casos.

Alcance de los beans

# Control del ciclo de vida del Bean

- Hasta ahora hemos usado la creación básica de beans, asumiendo que Spring crea una única instancia de cada uno.
- Existen más opciones para gestionar la instanciación de beans:
  - Control del número de instancias creadas
    - Singleton
    - Una por request
    - Una por petición
  - Creación mediante factoría
  - Inicialización y destrucción controlada del bean

# Control del ciclo de vida del Bean

## Control del ámbito (scope) del bean

- Por defecto, todos son **singleton**
- **Problema:** en determinados contextos, no podemos/queremos usar singletons.
- Utilizando el atributo scope del elemento bean determinamos su ciclo de vida:

Scope	Descripción
singleton	Valor por defecto. <b>Única instancia</b> por contenedor.
prototype	Se crea una <b>instancia por petición</b>
request	El ámbito se reduce a la <b>request HTTP</b> (Spring <b>MVC</b> )
session	El ámbito se reduce a la <b>sesión HTTP</b> (Spring <b>MVC</b> )
global-session	El ámbito se reduce al <b>contexto HTTP (Portlets)</b>

# Uso del atributo scope

- Si queremos definir un bean como prototype u otro que no sea singleton tendremos que modificar el **atributo scope**.

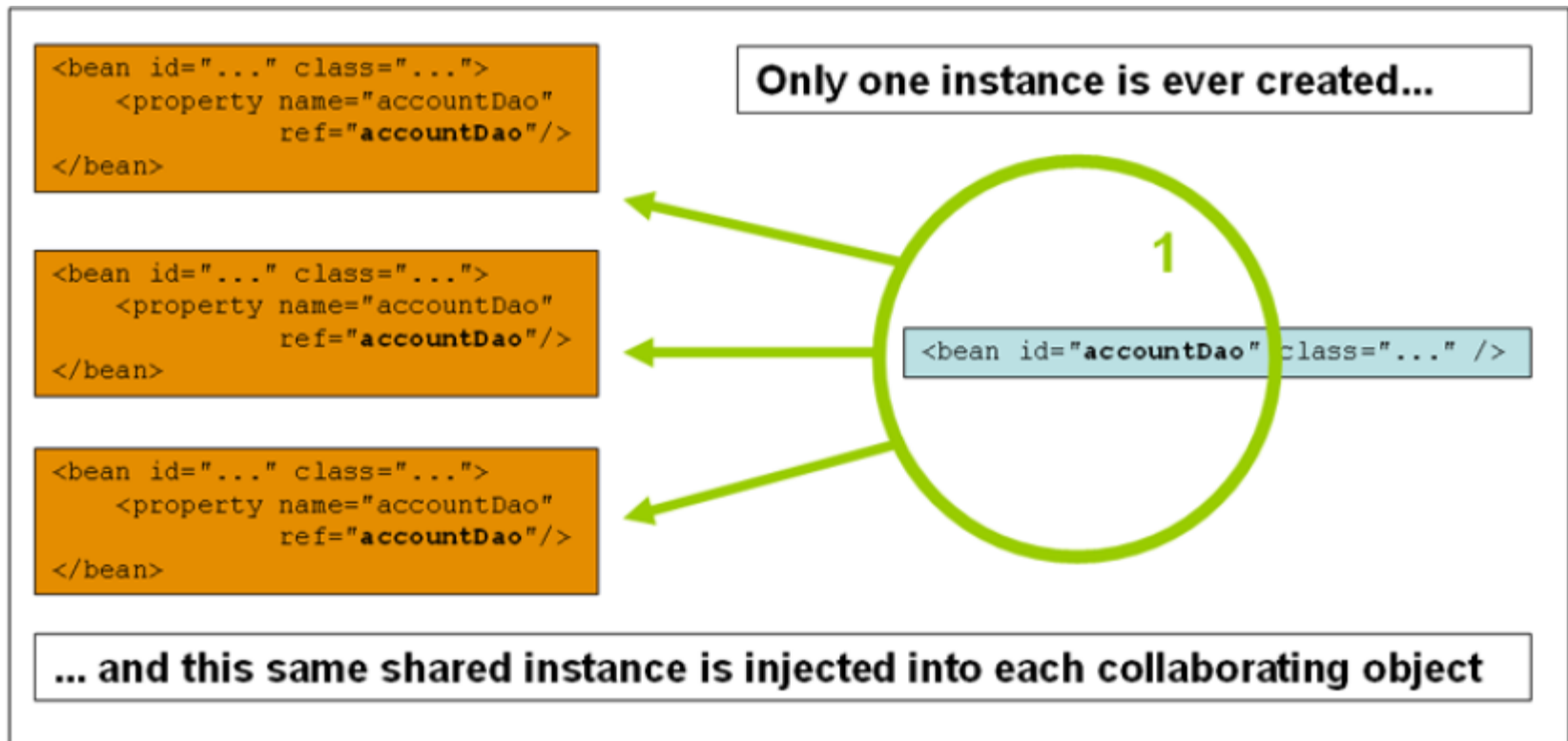
```
<bean id=<identificador>  
    class=<nombre de la clase>  
    scope={prototype  
        |singleton  
        |request  
        |session  
        |global-session/>
```

# Ciclo de Vida de los beans

- Por ejemplo, si queremos que cada vez que se utilice un bean se cree uno nuevo. El atributo scope le daremos el valor “**prototype**”.
- Este caso se dará cuando conectemos Struts2 con Spring. Todos los Actions de Struts2 que se encargue de gestionar Spring, deberá de crearse nuevos por cada petición.

# Singleton

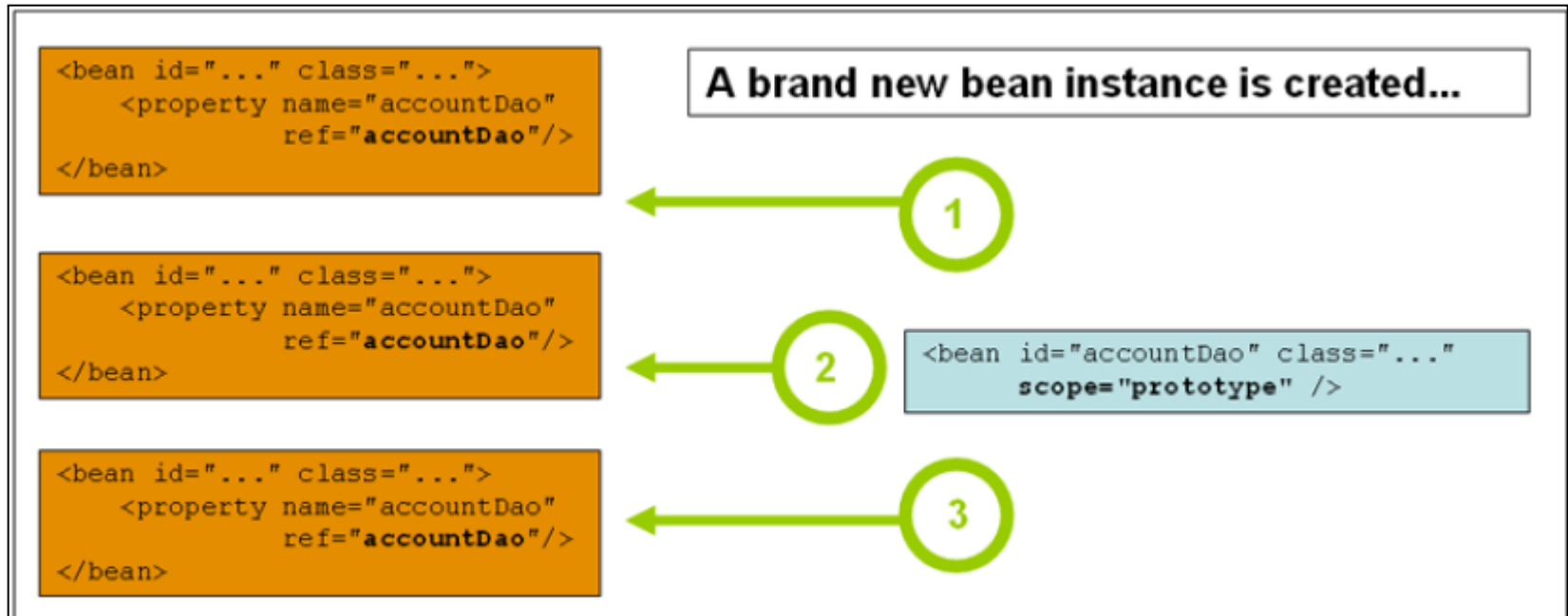
- Una única instancia: (los 3 beans referencian al mismo)





# Prototype

- Son distintas instancias: Cada vez que se referencia un bean se crea una nueva instancia.



# request, session, global-context

- Estos **scopes** sólo están disponibles en un entorno Web para las implementaciones de `ApplicationContext` como `XmlWebApplicationContext`.
- En nuestras aplicaciones Web tendremos que hacer una declaración en el **web.xml**, de este estilo (*lo vemos en MVC*):

```
<web-app>
...
<listener>
  <listener-class>
    org.springframework.web.context.request.RequestContextListener
  </listener-class>
</listener>
...
</web-app>
```

# Ejemplos

- En nuestro fichero: applicationContext.xml

```
<bean id="loginAction" class="com.foo.LoginAction"  
      scope="request"/>
```

- Crea una nueva instancia por cada petición HTTP.

```
<bean id="userPreferences" class="com.foo.UserPreferences"  
      scope="session"/>
```

- El bean userPreferences estará disponible a nivel de session.

```
<bean id="userPreferences" class="com.foo.UserPreferences"  
      scope="globalSession"/>
```

- El bean será compartido por todos los portlets.

# Personalización de Scope

- Spring permite que nos diseñemos nuestros propios Scopes.
- Necesitamos una clase que implemente el interface Scope.
- Registrar el nuevo Scope y un bean que represente el Scope.

# Métodos del interface Scope

- **Object get(String name, ObjectFactory<?> objectFactory)**
  - Recupera una instancia del bean, si no existe en el contexto (lo añade) y lo devuelve.
- **String getConversationId()**
  - Devolver el identificador de la conversión. Puede ser el nombre de la clase del Scope. Método opcional.
- **void registerDestructionCallback(String name, Runnable callback)**
  - Registra las llamadas a callbacks cuando es destruido. Método opcional.
- **Object remove(String name)**
  - Elimina la instancia del bean del ámbito de la aplicación. Método opcional.
- **Object resolveContextualObject(String key)**
  - Resolver el objeto contextual (si lo hay) para la palabra clave dada. Sería para peticiones HttpServletRequest (entorno web).
- Y se suele añadir un método **clear()** para limpiar el Scope.

# Fichero de configuración

```
<bean class="com.javabeat.MyScope" id="myScope"/>
- <bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
  - <property name="scopes">
    - <map>
      - <entry key="customscope">
        <ref bean="myScope"/>
      </entry>
    </map>
  </property>
</bean>
<bean class="com.javabeat.Person" name="p1" scope="customscope"/>
</beans>
```

p1 es un bean de tipo **Person** que asociamos al nuevo **Scope**.

Registra el scope dentro de un bean de tipo: **CustomScopeConfigurer**

# MyScope

```
public class MyScope implements Scope {  
    // Spring interactúa con esta clase según vayamos pidiendo objetos ...  
    private Map<String, Object> objectMap = Collections.synchronizedMap(new HashMap<String,  
        Object>());  
  
    @Override  
    public Object get(String name, ObjectFactory<?> objectFactory) {  
        if (!objectMap.containsKey(name)) {  
            objectMap.put(name, objectFactory.getObject());  
        }  
        return objectMap.get(name);  
    }  
  
    @Override  
    public String getConversationId() {  
        return "MyScope";  
    }  
  
    @Override  
    public Object remove(String name) {  
        return objectMap.remove(name);  
    }  
}
```

# MainApp

- `// Cargamos el contexto:`
- `ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");`
- `// Pedimos dos instancias al contexto: (será la misma Singleton)`
- `Person p1 = ctx.getBean("p1", Person.class);`
- `Person p2 = ctx.getBean("p1", Person.class);`
- `// Al imprimir obtenemos la misma referencia (toString() de Object): p1 y p2`
- `System.out.println(p1 + " es la misma que " + p2);`
- `// Limpiamos el Scope:`
- `MyScope myScope = ctx.getBean("myScope", MyScope.class);`
- `myScope.clearBean();`
- `// Después de limpiar el Scope volvemos a pedir instancias:`
- `Person p3 = ctx.getBean("p1", Person.class);`
- `Person p4 = ctx.getBean("p1", Person.class);`
- `// Al imprimir obtenemos la misma referencia (toString() de Object): p3 y p4`
- `System.out.println(p3 + " es la misma que " + p4);`
- ***Por un lado p1 y p2 coinciden y por otro p3 y p4, pero p1, p2 NO coinciden con p3,p4 porque se había eliminado del Scope y Spring ha creado una nueva instancia.***

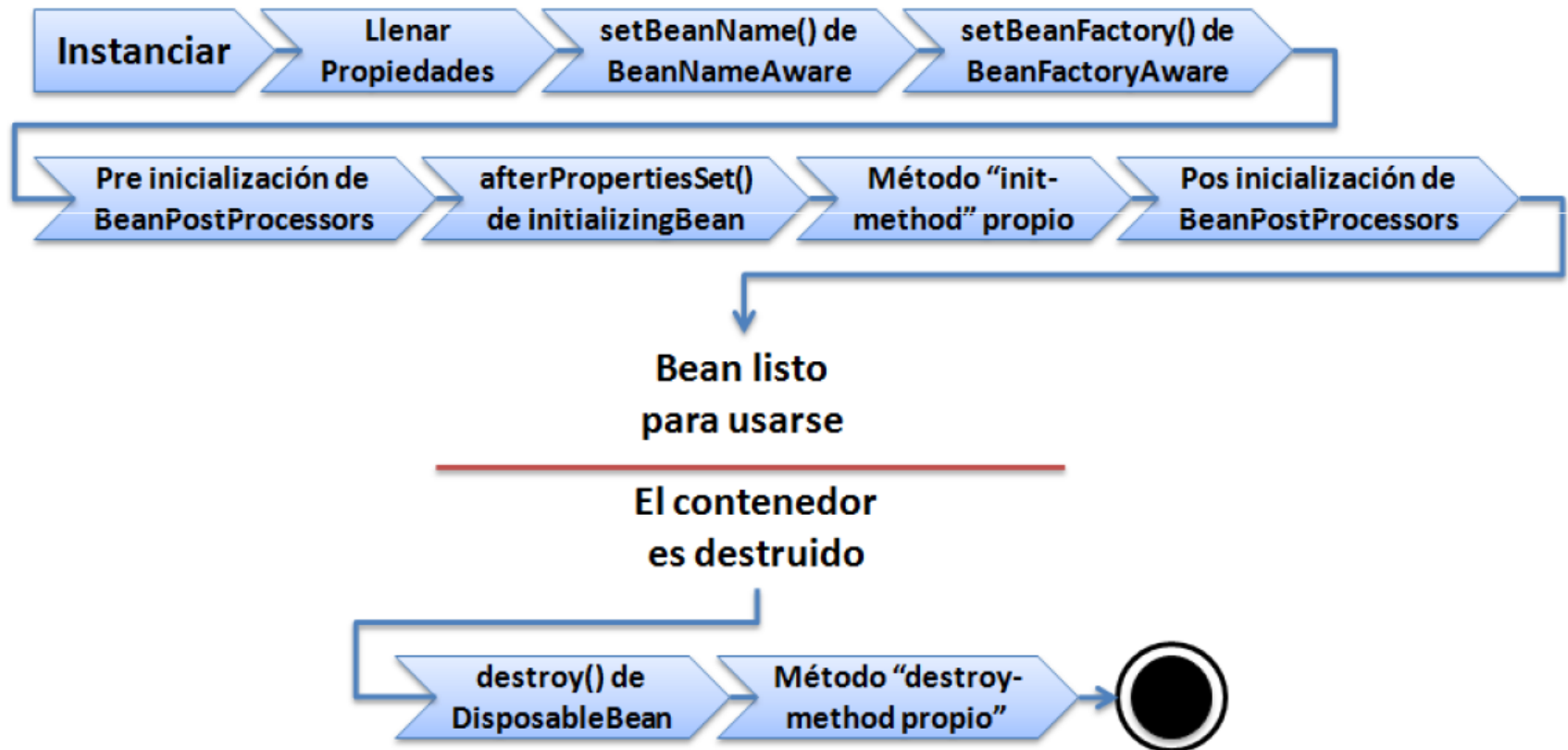


Personalización de la naturaleza  
del un bean

# Ciclo de vida de un Bean

- Como hemos comentado antes **Spring** se puede ver como **un contenedor de Beans NO INVASIVO**. Se basa en **POJO** (Plain Old Java Object).
- **Los beans no necesitan incorporar clases de Spring.**
- No tenemos que modificar la definición de nuestras clases.
- **El proceso de creación de un bean pasa por una serie de etapas → CICLO DE VIDA.**

# Ciclo de Vida de un Bean



# Ciclo de Vida de un Bean

Paso	Descripción
1. Instanciar	<code>Spring</code> instancia el bean
2. Llenar Propiedades	<code>Spring</code> inyecta las propiedades del bean
3. Establecer el nombre del bean	Si el bean implementa <code>"BeanNameAware"</code> , <code>Spring</code> pasa el <b>id</b> del bean a <code>"setBeanName ()"</code>
4. Establecer el bean factory	Si el bean implementa <code>"BeanFactoryAware"</code> , <code>Spring</code> pasa el bean factory a <code>"setBeanFactory ()"</code>
5. Post procesar (antes de la inicialización)	Si hay algún <code>"BeanPostProcessors"</code> , <code>Spring</code> llama a sus métodos <code>"postProcessBeforeInitialization ()"</code>
6. Inicializar beans	Si el bean implementa <code>"InitializingBean"</code> , se llamará a su método <code>"afterPropertiesSet ()"</code> . Si el bean tiene un método <code>"init-method"</code> propio (que veremos en la siguiente sección), el método será llamado.
7. Post procesar (después de la inicialización)	Si hay algún <code>"BeanPostProcessors"</code> , <code>Spring</code> llama a sus métodos <code>"postProcessAfterInitialization ()"</code>
8. El bean está listo para usarse	En este punto el bean está listo para ser usado por la aplicación y permanecerá en el bean factory hasta que deje de ser ocupado.
9. Destruir el bean	Si el bean implementa <code>"DisposableBean"</code> , se llama a su método <code>"destroy ()"</code> . Si el bean tiene un método <code>"destroy-bean"</code> propio, el método especificado será llamado.

# Ciclo de Vida de un Bean

- **Spring nos proporciona varios mecanismos para interactuar** con la administración del ciclo de vida, de los beans, de nuestro contenedor; más específicamente para el momento en el que los beans son creados y en el momento en el que serán destruidos.
  - Estos mecanismos se conocen como métodos de retrollamada, o **callback en inglés**.

# Inicialización

- El interface:  
`org.springframework.beans.factory.InitializingBean`  
permite ejecutar tareas de inicialización una vez el contenedor de Spring ha rellenado las propiedades del bean.  
**`void afterPropertiesSet() throws Exception;`**
- Aunque se aconseja mejor utilizar el atributo **`init-method`** haciendo referencia a un método del propio bean.

# Ejemplo

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

```
public class AnotherExampleBean implements InitializingBean {  
  
    public void afterPropertiesSet() {  
        // do some initialization work  
    }  
}
```

```
<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>
```

```
public class ExampleBean {  
  
    public void init() {  
        // do some initialization work  
    }  
}
```

# Destrucción

- De forma similar a la inicialización disponemos del interface:  
`org.springframework.beans.factory.DisposableBean` para ejecutar tareas cuando se produce la destrucción del bean.  
**`void destroy() throws Exception;`**
- Al igual que ocurre con la inicialización se aconseja implementar el método en el bean y utilizar el atributo **`destroy-method`**.



# Ejemplo

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>
```

```
public class AnotherExampleBean implements DisposableBean {  
    public void destroy() {  
        // do some destruction work (like releasing pooled connections)  
    }  
}
```

```
<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup"/>
```

```
public class ExampleBean {  
    public void cleanup() {  
        // do some destruction work (like releasing pooled connections)  
    }  
}
```

# Inicialización por defecto

```
public class DefaultBlogService implements BlogService {  
  
    private BlogDao blogDao;  
  
    public void setBlogDao(BlogDao blogDao) {  
        this.blogDao = blogDao;  
    }  
  
    // this is (unsurprisingly) the initialization callback method  
    public void init() {  
        if (this.blogDao == null) {  
            throw new IllegalStateException("The [blogDao] property must be set.");  
        }  
    }  
}
```

```
<beans default-init-method="init">  
  
    <bean id="blogService" class="com.foo.DefaultBlogService">  
        <property name="blogDao" ref="blogDao" />  
    </bean>  
  
</beans>
```

Herencia

# Herencia

- Hasta ahora hemos declarado los beans de forma individual, estableciendo las propiedades de cada uno una a una de forma específica.
- Problema: Puede degenerar en ficheros de configuración **muy extensos y poco tratables**.
- Ejemplo:
  - Tenemos muchos beans de un determinado tipo que comparten características → Podemos definir la misma característica en todos ellos.

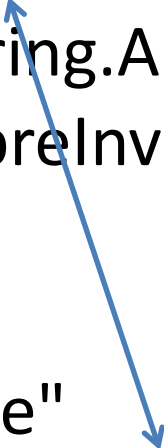
# Herencia

- Al igual que en la POO, es posible definir **relaciones padre-hijo** entre los beans declarados en un contenedor.
- Un bean que extiende la declaración de otro bean se define como **sub-bean** del segundo.
- Dos atributos específicos para esto:
  - **Parent**: Declara de qué bean hereda el que estamos declarando
  - **Abstract**: Declara el bean como abstracto, y por lo tanto, **no instanciable**.

# Ejemplo I

- `<bean id="abstractInvasorService" abstract="true" class="com.dosideas.spring.AbstractInvasorService">  
<property name="nombreInvasor" value="Zim"/>  
</bean>`

`<bean id="InvasorService"  
parent="abstractInvasorService"  
class="com.dosideas.spring.InvasorService">  
<property name="robotAsignado" value="Gir"/>  
</bean>`



El bean InvasorService tendrá dos propiedades.

# Ejemplo II

```
<bean id="<nombre padre>" class="<clase del padre>"  
  abstract="true">  
  <property name="<propiedad común>" ref="<ref bean a  
    heredar>"/> <!-- También puede ser value - - >  
</bean>
```

```
<bean id="<hombre hijo 1>" class="<clase del hijo1>"  
  parent="<nombre padre>">  
<property name="<otra propiedad>" value="<valor 1>"/>  
</bean>
```

```
<bean id="<hombre hijo 2>" class="<clase del hijo2>"  
  parent="<nombre padre>">  
<property name="<otra propiedad>" value="<valor 2>"/>  
</bean>
```

# Puntos de extensión del contenedor



# BeanPostProcessor

- Nos proporciona un interface para interactuar con el proceso de instanciación del bean.
- Estos métodos son llamados por el contenedor de Spring de forma automática.

```
public class InitHelloWorld implements BeanPostProcessor {  
    public Object postProcessBeforeInitialization(Object bean, String beanName)  
        throws BeansException {  
        System.out.println("BeforeInitialization : " + beanName);  
        return bean; // you can return any other object as well  
    }  
  
    public Object postProcessAfterInitialization(Object bean, String beanName)  
        throws BeansException {  
        System.out.println("AfterInitialization : " + beanName);  
        return bean; // you can return any other object as well  
    }  
}
```

# Ficheros de Propiedades

- En Spring podemos utilizar ficheros de configuración .properties y luego recuperar estos valores para inyectarlos en propiedades del bean.
- La clase ***PropertyPlaceholderConfigurer*** nos permite hacer referencia a propiedades de archivos de texto en la configuración de Spring.
  - Ubicada en el paquete:
    - ***org.springframework.beans.factory.config***.
- De esta manera podemos **externalizar propiedades** a un archivo .properties de texto.

# Externalizar Propiedades

- En el fichero de configuración tenemos que incluir el namespace del contexto y donde se encuentra ubicado el fichero de propiedades.

```
<context:property-placeholder location="classpath:config.properties"/>

<bean id="config" class="com.ejemplo.spring.Configuracion">
  <property name="url" value="${url}"></property>
  <property name="bd" value="${bd}"></property>
  <property name="user" value="${user}"></property>
  <property name="pwd" value="${pwd}"></property>
</bean>
```

```
url=http://www.prueba.com
bd=empresa3
user=admin
pwd=admin
```

# Externalizar propiedades

- También se puede indicar la ubicación configurando un bean que haga referencia a la clase:

```
<bean  
  class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  
  <property name="location">  
    <value>database.properties</value>  
  </property>  
</bean>
```

# FactoryBean

- Spring proporciona el interface `FactoryBean` para personalizar la creación de beans.
- Necesitamos una clase que implemente el interface **`FactoryBean`**.
- Hay 3 métodos:
  - **`Object getObject()`**
    - Retorna un bean que puede ser compartido o no, dependiendo si la factoría devuelve `Singleton` o `Prototype`.
  - **`boolean isSingleton()`**
    - Retorna `true` si devuelve un singleton, falso en caso contrario.
  - **`Class getObjectType()`**
    - Retorna el tipo del objeto.

# factory-bean / factory-method

- Otra forma de personalizar las factorías es haciendo referencia desde el fichero de configuración que factoría queremos y que método para crear una instancia declarada en dicho fichero.
- Por ejemplo:

```
<bean id="userService" factory-bean="serviceFactory"  
factory-method="createUserService" />  
  
<bean id="loginService" factory-bean="serviceFactory"  
factory-method="createLoginService" />
```

# ApplicationContext

# i18n

- **ApplicationContext** proporciona la posibilidad de internacionalizar los mensajes de respuesta al usuario.
- Para ello utiliza ficheros **.properties** y la clase **ResourceBundleMessageSource**.
- Nos creamos un fichero properties por cada idioma que queremos.
  - mensajes\_**fr\_FR**.properties
  - mensajes\_**en\_US**.properties



# i18n

- El contenido de los ficheros de propiedades puede tener parámetros:
- clave=mensaje
- Ejemplo:
  - customer.name=Mi nombre es {0}, tengo {1} años.
  - Se pueden pasar parámetros.

# i18n

- En el contexto registramos un objeto de tipo ***org.springframework.context.support.ResourceBundleMessageSource***.
- Y rellenamos su propiedades basename, indicando la carpeta donde se encuentran los ficheros de mensajes y su prefijo, el resto lo monta Spring según el **Locale seleccionado**.

```
<bean id="messageSource"  
      class="org.springframework.context.support.ResourceBundleMessageSource">  
  <property name="basename">  
    <value>es\curso\ejemplo\mensajes</value>  
  </property>  
</bean>
```

# i18n

- **ApplicationContext** dispone del método:  
`String getMessage(String clave, Object[] param, Locale locale)`
  - **Clave:** es una clave del mensaje (que va dentro de los ficheros).
  - **Param:** un array de objetos con los parámetros a rellenar dentro del mensaje.
  - **Locale:** Indicamos el idioma que queremos:
    - `Locale.US`, `Locale.FRANCE`, etc.

# Eventos

- ApplicationContext maneja todo el ciclo de creación de beans, durante este proceso lanza una serie de eventos:
- **ContextRefreshedEvent**
  - Este evento se publica cuando el ApplicationContext es ya sea inicializado o se actualiza.
- **ContextStartedEvent**
  - Este evento se publica cuando se inicia el ApplicationContext utilizando el método start () en la interfaz theConfigurableApplicationContext
- **ContextStoppedEvent**
  - Este evento se publica cuando el ApplicationContext se detiene utilizando el método stop() en la interfaz theConfigurableApplicationContext
- **ContextClosedEvent**
  - Este evento se publica cuando el ApplicationContext se cierra mediante el método close () en la interfaz theConfigurableApplicationContext.
- **RequestHandledEvent**
  - Se trata de un evento específico en la web diciendo que todos los beans de una petición HTTP se han servido.

# ApplicationListener

- Disponemos del interface **ApplicationListener** y lo parametrizamos con la clase del evento que queremos capturar.
- Ejemplo:
  - public class CStartEventHandler implements ApplicationListener<**ContextStartedEvent**>
  - public class CStopEventHandler implements ApplicationListener<**ContextStoppedEvent**>

# Lanzar Eventos

- Desde nuestra aplicación una vez que hemos cargado el contexto podemos llamar a los métodos `start()` y `stop()` para provocar los eventos.

```
ApplicationContext contexto = new ...  
contexto.start();  
// trabajar con los beans ...  
contexto.stop();
```

# Configuración con Anotaciones

# Configuración con Anotaciones

- La configuración con anotaciones es una alternativa a la configuración en XML.
- Necesitamos indicar una configuración especial para trabajar con anotaciones (en el fichero de configuración):
  - El **namespace context**.
  - Y declarar este tag: **<context:annotation-config />**
  - Para que reconozca las anotaciones.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <context:annotation-config/>

</beans>
```



# Tipos de anotaciones

- **@Autowired**
  - (Anotación de Spring). Inyecta una dependencia **por tipo**.
- **@Qualifier**
  - Para **distinguir** entre varios **objetos** del **mismo tipo**.
- **@Inject**
  - (Anotación de java JSR-330), equivalente a @Autowired.
- **@Required**
  - Para forzar a que se rellene una propiedad.
- **@Resource**
  - Inyecta la dependencia **por nombre**.

# Anotación: @Autowired

- Esta anotación la podemos utilizar en:
  - **Declaración de un atributo.**
  - En un **método set.**
  - En un **constructor.**
- **Conecta por tipo.**
- De esta forma nos evitamos utilizar el tag: property en el fichero de XML.
- La **anotación la tenemos** en:
  - org.springframework.beans.factory.annotation.Autowired;
- ***Con esta anotación nos podemos ahorrar property y constructor-arg. Pero no la declaración del bean.***

# Ejemplos

```
package com.mkyong.common;

import org.springframework.beans.factory.annotation.Autowired;

public class Customer
{
    private Person person;
    private int type;
    private String action;
    //getter and setter methods

    @Autowired
    public void setPerson(Person person) {
        this.person = person;
    }
}
```

- De forma similar se puede colocar por encima de la **declaración del atributo**, o en un **constructor**.

# Ejemplos

- Autowired en un **constructor**

```
package com.mkyong.common;

import org.springframework.beans.factory.annotation.Autowired;

public class Customer
{
    private Person person;
    private int type;
    private String action;
    //getter and setter methods

    @Autowired
    public Customer(Person person) {
        this.person = person;
    }
}
```

# Ejemplos

- Si Spring no consigue conectar lanzará una excepción.
- Podemos hacer que no sea obligatorio conectarlo, con el atributo **required**.

```
package com.mkyong.common;

import org.springframework.beans.factory.annotation.Autowired;

public class Customer
{
    @Autowired(required=false)
    private Person person;
    private int type;
    private String action;
    //getter and setter methods
}
```

# @Required

- Se aplica a los métodos **setters** de un bean.

```
public class SimpleMovieLister {  
  
    private MovieFinder movieFinder;  
  
    @Required  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
  
    // ...  
}
```

- Esta anotación indica que esa propiedad tiene que ser poblada en tiempo de configuración, con un valor explícito o con autowiring.
- Si no es así se lanza una excepción.

# @Qualifier: Evitar Ambigüedades

- Puede ocurrir que el fichero de beans tengamos declarados dos objetos del mismo tipo. *¿cómo distinguir el que queremos conectar con @Autowired?*

```
<context:annotation-config />

<bean id="CustomerBean" class="com.mkyong.common.Customer">
    <property name="action" value="buy" />
    <property name="type" value="1" />
</bean>

<bean id="PersonBean1" class="com.mkyong.common.Person">
    <property name="name" value="mkyong1" />
    <property name="address" value="address 1" />
    <property name="age" value="28" />
</bean>

<bean id="PersonBean2" class="com.mkyong.common.Person">
    <property name="name" value="mkyong2" />
    <property name="address" value="address 2" />
    <property name="age" value="28" />
</bean>
```

# Ejemplo

```
package com.mkyong.common;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;

public class Customer
{
    @Autowired
    @Qualifier("PersonBean1")
    private Person person;
    private int type;
    private String action;
    //getter and setter methods
}
```



# @Qualifier

- También la podemos encontrar delante de un argumento de un constructor o un método.

```
public class MovieRecommender {  
  
    private MovieCatalog movieCatalog;  
  
    private CustomerPreferenceDao customerPreferenceDao;  
  
    @Autowired  
    public void prepare(@Qualifier("main") MovieCatalog movieCatalog,  
                        CustomerPreferenceDao customerPreferenceDao) {  
        this.movieCatalog = movieCatalog;  
        this.customerPreferenceDao = customerPreferenceDao;  
    }  
  
    // ...  
}
```

# @Resource

- **Permite la inyección de beans mediante el nombre.**
- Los beans que queremos inyectar tendrán que estar definidos en el contexto de Spring (en XML) o mediante la anotación **@Component**.
- **@Resource** se indicará por encima de un método set o por encima de la declaración de una propiedad.
- Tiene un atributo **name** donde se puede indicar el nombre del bean a inyectar y si no se utilizará convención de nombres.

# Ejemplos

```
public class SimpleMovieLister {  
  
    private MovieFinder movieFinder;  
  
    @Resource(name="myMovieFinder")  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
}
```

En este caso el bean a buscar se llamará **movieFinder!!**

```
public class SimpleMovieLister {  
  
    private MovieFinder movieFinder;  
  
    @Resource  
    public void setMovieFinder(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
}
```

# @Component

- También podemos evitar la definición de beans en el contexto de Spring.
- **@Component**
  - Sustituye a las declaraciones de los beans. Componente de uso general.
  - Se indica por encima de la clase, si no indicamos el id, será el nombre de la clase en minúsculas.

```
@Component("mi_Id")  
public class MiClase { .. }
```

# Escaneo de beans

- Cuando utilizamos **@Component** no declaramos los beans en el contexto.
- Pero tenemos que indicar a Spring que localice los beans anotados.
- Incluimos estas declaraciones dentro del contexto:

```
<context:annotation-config />  
<context:component-scan base-package="es.curso.ejemplo" />
```

En **base-package** indicamos el paquete a escanear ...

# @Value

- Podemos dar valores a las propiedades de la clase.
- Esto puede resultar muy estático pero soporta expresiones en lenguaje **SpEL** (*más adelante lo vemos*).

```
@Value("España")  
private String pais;
```

# Ejemplo: @Resource, @Component y @Value

```
@Component
public class Motor {

    @Value("Honda")
    private String marca;

    @Value("Diesel")
    private String tipo;
}
```

```
public class Coche {

    @Autowired
    @Qualifier("radio1")
    private Radio radio;

    private String marca;
    private String modelo;

    |
    @Resource
    private Motor motor;
}
```

En el fichero de contexto no  
Es necesario definir el bean  
Motor.

En la clase Coche, también  
Podemos utilizar @Autowired para  
Conectar la propiedad motor.

OJO!!

```
<context:annotation-config />
<context:component-scan base-package="es.curso.ejemplo" />
```

# @PostConstruct / @PreDestroy

- En Spring podemos implementar los interfaces InitializingBean y DisposableBean o especificar los métodos “**init-method**” y “**destroy-method**” en el fichero de configuración para hacer llamadas a funciones durante el proceso de creación y destrucción de los beans.
- Esto es lo que nos permiten las anotaciones @PostConstruct y @PreDestroy.



# @PostConstruct / @PreDestroy

- En el fichero de configuración es necesario declarar:

**<context:annotation-config />**

```
public class CachingMovieLister {  
  
    @PostConstruct  
    public void populateMovieCache() {  
        // populates the movie cache upon initialization...  
    }  
  
    @PreDestroy  
    public void clearMovieCache() {  
        // clears the movie cache upon destruction...  
    }  
}
```

# javax.annotation

- Las **anotaciones**:
  - @PostConstruct,
  - @PreDestroy y
  - @Resource
- No son de spring framework son de javax.annotation.
- Para proyectos con Maven añadir la dependencia a **pom.xml**

# Otras anotaciones

- **@Controller:**
  - Representa un controlador dentro de Spring MVC.
  - `org.springframework.stereotype.Controller;`
- **@Repository:**
  - Repositorio de datos. Capa de persistencia.
  - `org.springframework.stereotype.Repository;`
- **@Service:**
  - la clase define un servicio. Lógica de negocio.
  - `org.springframework.stereotype.Service;`

# Ejemplo

- Nombre por defecto y nombre concreto:

```
CustomerService cust = (CustomerService)context.getBean("customerService");
```

```
@Service("AAA")  
public class CustomerService  
...
```

```
CustomerService cust = (CustomerService)context.getBean("AAA");
```

# JavaConfig

- JavaConfig es un proyecto que se ha añadido al core del framework de Spring.
- Tiene soporte para las siguientes anotaciones:
  - @Configuration**: Permite definir una clase que representa la configuración del contexto.
  - @Bean**: Define un bean dentro de la clase de configuración.
  - @Value**: Para dar valor a una propiedad de un bean.
  - @Lazy**: Carga perezosa. No se instancia hasta la primera petición.
  - @DependsOn**: Para indicar que un bean tiene que ser instanciado antes que otro. Relación “depende de”.

# JavaConfig

- JavaConfig es una alternativa de configuración XML
- Se definen instancias, lógica de inicialización, etc.

```
@Configuration
public class AppConfig {
    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }
}
```

Equivalente



```
<beans>
  <bean id="myService" class="com.acme.services.MyServiceImpl"/>
</beans>
```

Cuando queremos cargar la configuración disponemos de una implementación

De **ApplicationContext** exclusiva para las **anotaciones**:  
**AnnotationConfigApplicationContext.**

# Ejemplo

```
package com.mkyong.hello;

public interface HelloWorld {

    void printHelloWorld(String msg);

}
```

```
package com.mkyong.hello.impl;

import com.mkyong.hello.HelloWorld;

public class HelloWorldImpl implements HelloWorld {

    @Override
    public void printHelloWorld(String msg) {

        System.out.println("Hello : " + msg);

    }

}
```

```
package com.mkyong.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import com.mkyong.hello.HelloWorld;
import com.mkyong.hello.impl.HelloWorldImpl;

@Configuration
public class AppConfig {

    @Bean(name="helloBean")
    public HelloWorld helloWorld() {
        return new HelloWorldImpl();
    }

}
```

# Ejemplo 2

```
package org.example.config;

@Configuration
public class AppConfig {
    private @Value("#{jdbcProperties.url}") String jdbcUrl;
    private @Value("#{jdbcProperties.username}") String username;
    private @Value("#{jdbcProperties.password}") String password;

    @Bean
    public FooService fooService() {
        return new FooServiceImpl(fooRepository());
    }

    @Bean
    public FooRepository fooRepository() {
        return new HibernateFooRepository(sessionFactory());
    }

    @Bean
    public SessionFactory sessionFactory() {
        // wire up a session factory
        AnnotationSessionFactoryBean asFactoryBean =
            new AnnotationSessionFactoryBean();
        asFactoryBean.setDataSource(dataSource());
        // additional config
        return asFactoryBean.getObject();
    }

    @Bean
    public DataSource dataSource() {
        return new DriverManagerDataSource(jdbcUrl, username, password);
    }
}
```



# Ejemplo 2

```
<context:component-scan base-package="org.example.config"/>  
<util:properties id="jdbcProperties" location="classpath:org/example/config/jdbc.properties"/>
```

```
public static void main(String[] args) {  
    ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);  
    FooService fooService = ctx.getBean(FooService.class);  
    fooService.doStuff();  
}
```

En la definición del contexto:

Indica la ubicación de donde se encuentra el fichero **.properties** con la Configuración de la Base de datos.

Carga el contexto y recupera un bean mediante su class.

# Expresiones SpEL

- Se incorpora en Spring 3.
- Es un lenguaje de expresiones similar a otros como EL, o los que utilizan Struts2 / JSF para acceder a las propiedades de los Beans.
- Principales funcionales del Lenguaje:
  - Referenciar a los bean mediante su id.
  - Invocar métodos y acceder a propiedades de beans.
  - Operaciones lógicas, relacionales y matemáticas.
  - Expresiones regulares.
  - Manipulación de colecciones.

# Requisitos

- Si utilizamos Maven en los proyectos, la única dependencia necesaria para usar empezar a usar **SpEL** es:
- Por ejemplo, con la versión 3.0.3

<dependency>

<groupid>org.springframework</groupid>

<artifactid>**spring-context**</artifactid>

<version>3.0.3.RELEASE</version>

</dependency>

- Si deseas gestionar tus librerías sin Maven, además de todos los JAR's necesarios para levantar una aplicación Spring 3.0 necesitas incluir en tu classpath el siguiente JAR: **org.springframework.expression-3.0.X.RELEASE.jar**

# Sintaxis

- La sintaxis de SpEL toma la siguiente forma cuando es usada de forma nativa:  
**#{EXPRESIÓN}**
- Cuando la expresión es usada por un parser, debemos omitir el carácter de almohadilla y la pareja de llaves:  
**EXPRESIÓN**

# Literales con SpEL

- Los valores literales irían encerrados entre los símbolos: `#{}`, se especifican en el atributo `value` del elemento `property`.
- Ejemplos:  

```
<property name="count" value="#{5}" />
```

```
<property name="mensaje" value="El valor es #{5}" />
```
- Otros valores válidos son:
  - `#{56.9}`, `#{'cadena'}`, `{false}`

# Referencias a bean, propiedades y métodos

- Podemos acceder a propiedades de otros bean.

- Ejemplo:

```
<bean id="carl"  
      class="ejemplo.Instrumentalist">  
  <property name="song" value="#{kenny.song}"/>  
</bean>
```

// Captura la propiedad song de otro bean cuyo id  
// es kenny. **Equivalente a:**

```
Instrumentalist carl = new Instrumentalist();  
carl.setSong(kenny.getSong());
```

# Ejemplos

```
class Login {  
    @Value("#{usuario.nombre}")  
    private String nombreUsuario;  
    // ...  
}
```

- La anotación **@Value** puede ser aplicada tanto en variables como métodos setter, así como en parámetros de un método o constructor.
- Para usar la misma expresión en un archivo de configuración de Spring 3.0:  

```
<bean class="Login" id="login">  
    <property name="nombreUsuario" value="#{usuario.nombre}">  
</property></bean>
```

# SpEL a través de un Parser

- **SpEL** puede ser usado fuera de un archivo XML o anotación **@Value** mediante un parser. Este parser requiere el ensamblaje de cierta infraestructura:

```
ExpressionParser parser = new SpelExpressionParser();  
Expression expresion = parser.parseExpression("'Hola SpEL'");  
String nombreUsuario = expresion.getValue(String.class);
```

- Primero crea un parser de la clase SpelExpressionParser.
- A continuación crea una expresión llamando al método parseExpression(EXPRESSION) del recién creado parser.
- Por último, obtenemos el resultado de la evaluación de la expresión a través del objeto expresion.



# SpEL a través de un Parser

- Cuando la expresión hace referencia a un objeto, debemos crear un contexto de ejecución para dicho objeto:

```
Usuario usuario = new Usuario("David Marco");  
EvaluationContext contexto = new StandardEvaluationContext(usuario);  
ExpressionParser parser = new SpelExpressionParser();  
Expression expresion = parser.parseExpression("nombre");  
String nombreUsuario = expresion.getValue(contexto, String.class);
```

# Referencias a bean, propiedades y métodos

- Mediante al lenguaje SpEL también se puede llamar a métodos:

`#{id_objeto.metodo()}`

— Por ejemplo, si método devuelve un String podríamos hacer:

`#{id_objeto.metodo().toUpperCase()}`

— Se puede proteger por si acaso metodo() devuelve null:

`#{id_objeto.metodo()}.toUpperCase()`

La ?. nos asegura que lo de la izquierda no es null.

# Trabajar con tipos

- Por ejemplo, podemos acceder a la clase Math y a métodos static y constantes de la clase.
- Acceso a una cte:  
`<property name="..." value="#{T(java.lang.Math).PI}" />`
- Acceso a un método static:  
`<property name="num" value="#{T(java.lang.Math).random()}" />`

# Operadores

- Aritméticos                    +, -, \*, /, %, ^ (potencia).
  - Relacionales                 <, >, ==, <=, >=, lt, gt, eq, le, ge
  - Lógicos                      and, or, not
  - Condicionales                ?:
  - Exp. Regulares               matches
- 
- En el caso de los relacionales para evitar problemas con el XML utilizar: lt, gt, eq, le, ge.

# Ejemplos

- `<property name="area" value="#{T(java.lang.Math).PI * circulo.radio ^ 2}" />`
- Se pueden comparar cadenas con `==`.
- `<property name="equal" value="#{contador.total eq 100}" />`
- `<property name="instrumento"`  
`value="#{songSelector.selectSong()=='Jingle Bells' ?`  
`piano : saxofon}"`

# Filtrado de colecciones

- Dada la clase:

```
package ejemplos.spring  
public class City {  
    private String name;  
    private String state;  
    private int population;  
}
```

```
<bean ... >  
<property name="lista">  
    <list id="cities">  
        <bean  
            class="ejemplos.spring.City"  
            p:name="Chicago" p:state="IL"  
            p:population="438348"/>  
        <bean  
            class="ejemplos.spring.City"  
            p:name="Dallas" p:state="TX"  
            p:population="112348"/>  
    </list>  
</property>  
</bean>
```

# Acceso a miembros de la colección

- Acceso a elementos de la colección por posición:
  - En caso de las **listas**:
    - `<property name="chosenCity" value="#{cities[2]}" />`
  - En caso de que la colección sea un **mapa**:
    - `<property name="chosenCity" value="#{cities['Dallas']}" />`
  - Propiedades disponible en **SpEL**, son las colecciones **systemEnvironment** y **systemProperties**.
    - `<property name="homePath" value="#{systemEnviroment['HOME']}" />`
    - `<property name="homePath" value="#{systemProperties['application.home']}" />`

# Seleccionar miembros de la colección

- Con SpEL se pueden realizar filtrado de colecciones utilizando el operador `.?[]`
  - `<property name="bigCities" value="#{cities.?[population gt 100000]}" />`
    - Lista de objetos City cuya población supera los 100000.
- También se puede obtener el primero y el último elemento que cumple la condición con los operadores: `.^[]` y `.$[]`
  - **El primero:**
    - `<property name="aBigCity" value="#{cities.^[population gt 100000]}" />`
  - **El último:**
    - `<property name="aBigCity" value="#{cities.$[population gt 100000]}" />`



# Proyectar colecciones

- Proyectar una colección es recopilar una propiedad concreta de cada uno de los miembros para incluirlas en una nueva.
- Operador de proyección (.![])
  - `<property name="cityNames" value="#{cities.![name]}" />`
  - Podemos incluir mas de una propiedad, pero en este caso irían concatenadas.
    - `<property name="cityNames" value="#{cities.![name + ' ' + estado]}" />`

# Ejemplo

- La clase que representa las colecciones:

```
public class ListaUsers {  
    private List<User> usuarios;  
    private List<User> lista2;  
    private User usuario;  
  
    // Con los métodos set / get de cada  
    propiedad.  
}
```

// La clase User:

```
public class User {  
    private int edad;  
    private String name;  
    private String country;  
  
    // Con sus métodos get / set.  
}
```

```
- <bean class="org.ejemplo.ListaUsers" id="listausuarios">  
  - <property name="usuarios">  
    - <list>  
      <ref bean="user1"/>  
      <ref bean="user2"/>  
      <ref bean="user3"/>  
      <ref bean="user4"/>  
      <ref bean="user5"/>  
    </list>  
  </property>  
  - <property name="lista2">  
    - <list>  
      - <bean class="org.ejemplo.User">  
        <property value="user1" name="name"/>  
        <property value="11" name="edad"/>  
        <property value="pais1" name="country"/>  
      </bean>  
      + <bean class="org.ejemplo.User">  
      + <bean class="org.ejemplo.User">  
    </list>  
  </property>  
</bean>
```

# Continua Ejemplo

- En otra clase recuperamos, filtramos, proyectamos colecciones ...

```
public class Entidad {
```

```
    private User usuario;
```

```
    private List<String> nombreUsuarios;
```

```
    private List<User> mayores18;
```

```
    private User mayor;
```

```
    // Métodos set / get de cada propiedad ...
```

```
}
```

```
- <bean class="org.ejemplo.Entidad" id="entidad">
    <property value="#{listausuarios.lista2[1]}" name="usuario"/>
    <property value="#{listausuarios.lista2.![name]}" name="nombreUsuarios"/>
    <property value="#{listausuarios.lista2.?[edad gt 18]}" name="mayores18"/>
    <property value="#{listausuarios.lista2.$[edad gt 18]}" name="mayor"/>
</bean>
```

# Conectar colecciones

- Spring a parte de configurar propiedades simples, también permite trabajar con cuatro tipos de colecciones.

Colección	Descripción
<b>List</b>	Lista de elementos con duplicados permitidos
<b>Set</b>	Conjunto (sin elementos repetidos)
<b>Map</b>	Colección de pares nombre-valor donde ambos elementos pueden ser de cualquier tipo
<b>Props</b>	Colección de pares nombre-valor donde ambos elementos son de tipo <b>String</b>

# Conectar colecciones

- Las colecciones pueden ser de tipos primitivos o pueden referenciar a otros beans previamente declarados en el contexto.

- Por ejemplo, una lista de String podría ser:

```
<bean id="coleccion1" class="colecciones.ejemplo.clases.Colecciones">
<property name="lista"> <!-- En este caso, se define una propiedad que es una lista. -->
    <list>
        <value>Lista1</value>
        <value>Lista2</value>
        <value>Lista3</value>
    </list>
</property>
```

- Si queremos indicar elementos que a su vez sean objetos: `<ref bean="ref_a_otro Bean"/>`
- Incluso podemos tener listas de listas.

# Conectar colecciones

- Para el caso de los mapas se indicará con la etiqueta **<map>**, y en este caso habrá que indicar clave – valor.

**<entry** key="clave1" value="valor1"></entry>

– Si los elementos fueran objetos:

**<entry** key-ref="referencia a clave" value-ref="valor ref" />

# Conectar colecciones

- Para conjuntos:

```
<set>  
  <value>Conjunto1</value>  
  <value>Conjunto2</value>  
</set>
```

- Para propiedades:

```
<props>  
  <prop key="c1">valor1</prop>  
  <prop key="c2">valor2</prop>  
  <prop key="c3">valor3</prop>  
</props>
```