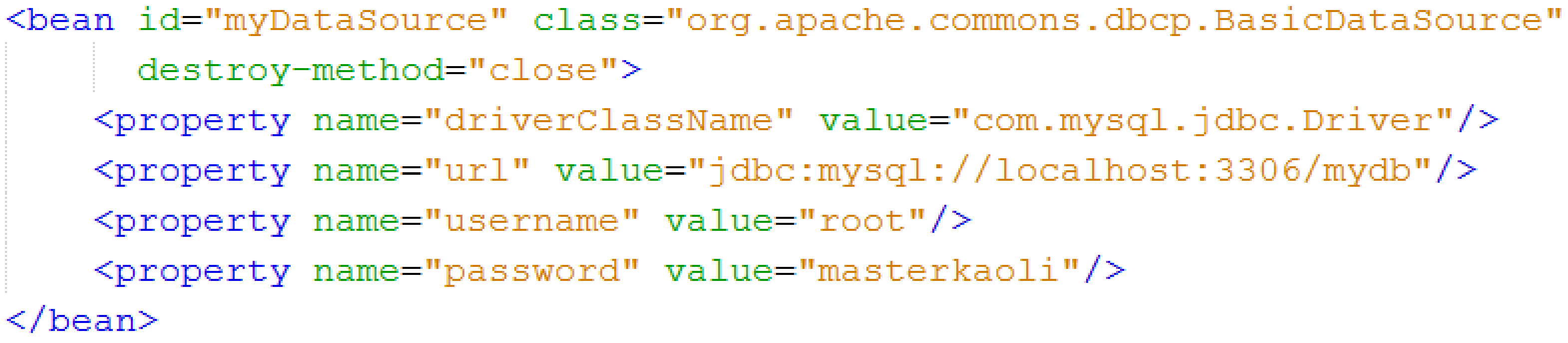
Circular dependencies

- If you use predominantly constructor injection, it is possible to create an unresolvable circular dependency scenario.
    - For example:
        - **Class A** requires an instance of **Class B** through constructor injection.
        - And **Class B** requires an instance of **Class A** through constructor injection.
        - If you configure beans for classes A and B to be injected into each other, the Spring IoC container detects this circular reference at runtime, and throws a BeanCurrentlyInCreationException.
- One possible solution is to edit the source code of some classes to be configured by setters rather than constructors.
- Alternatively, avoid constructor injection and use setter injection only. In other words, although it is not recommended.

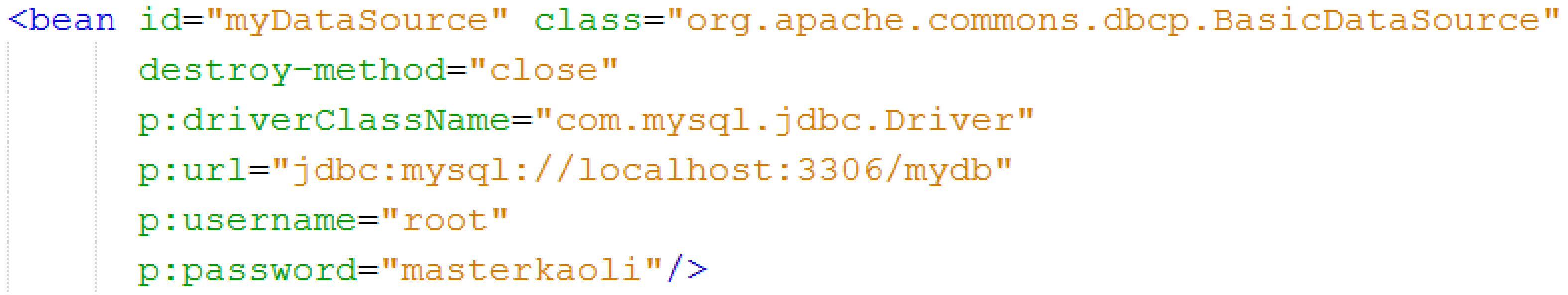
Straight Values vs. p-namespace

- **Straight Values (Primitives, Strings, and so on)**
  - Spring’s conversion service is used to convert property values from a String to the type of the property.



```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
    <property name="username" value="root"/>
    <property name="password" value="masterkaoli"/>
</bean>
```

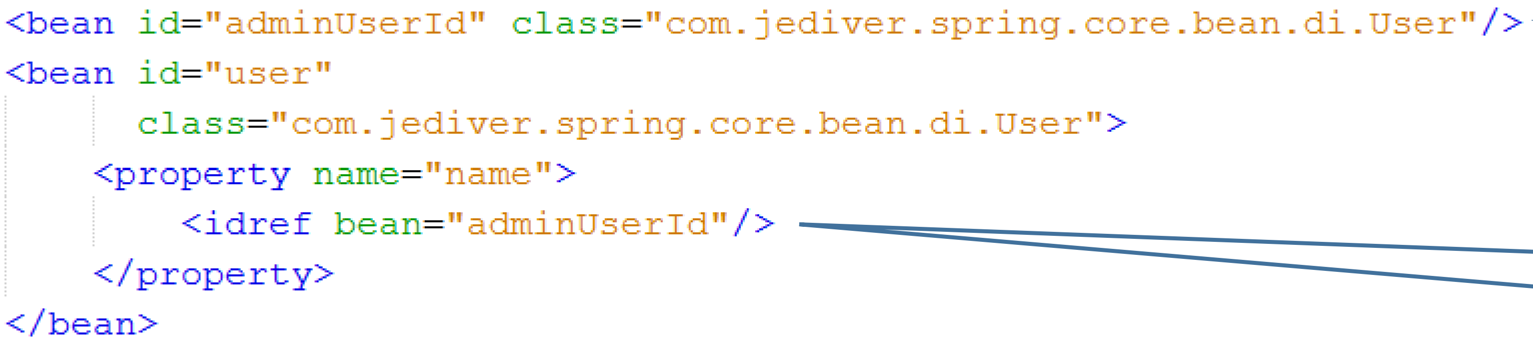
OR Using p-namespace



```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close"
      p:driverClassName="com.mysql.jdbc.Driver"
      p:url="jdbc:mysql://localhost:3306/mydb"
      p:username="root"
      p:password="masterkaoli"/>
```

The idref element

- If you want to pass a bean id to another bean.

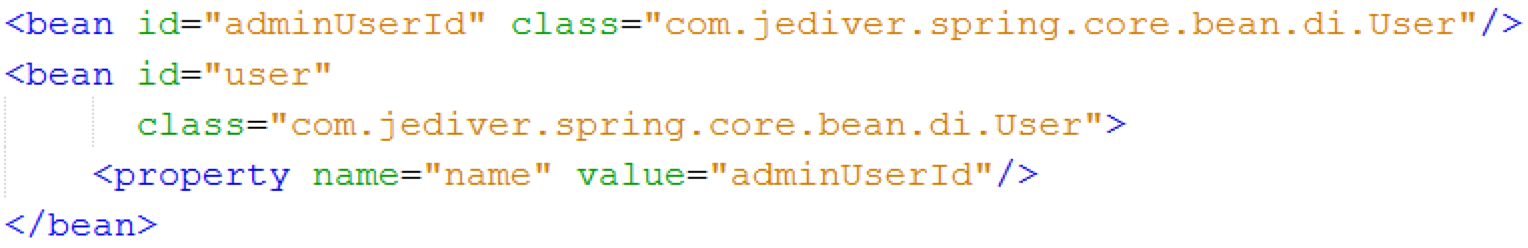


```
<bean id="adminUserId" class="com.jediver.spring.core.bean.di.User"/>
<bean id="user"
      class="com.jediver.spring.core.bean.di.User">
    <property name="name">
        <idref bean="adminUserId"/>
    </property>
</bean>
```

Any bean from any class

Injected the id Value not the bean itself

OR



```
<bean id="adminUserId" class="com.jediver.spring.core.bean.di.User"/>
<bean id="user"
      class="com.jediver.spring.core.bean.di.User">
    <property name="name" value="adminUserId"/>
</bean>
```

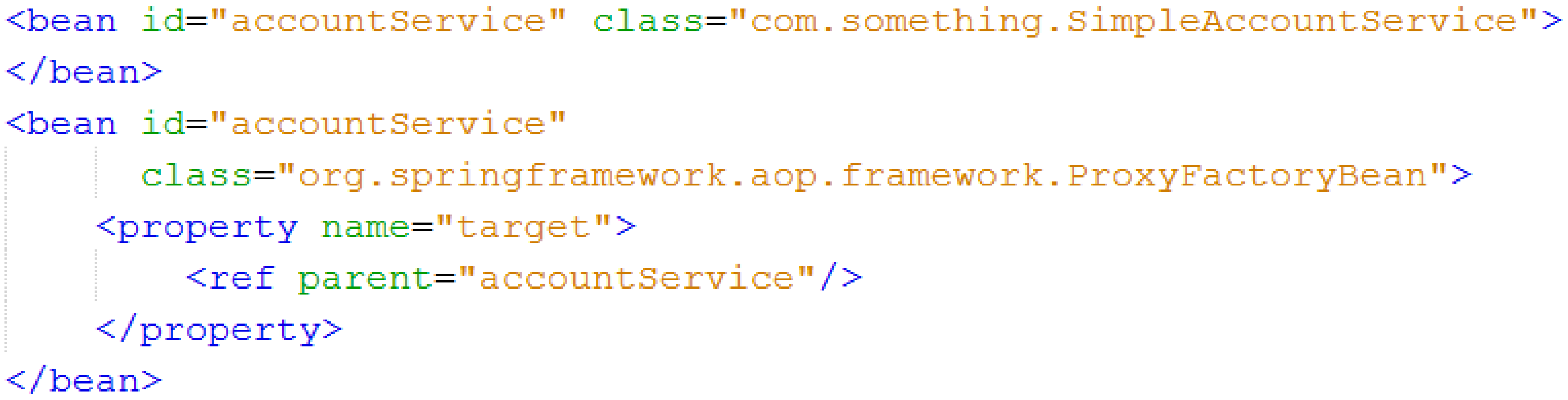
- Using idref is preferable, because it lets the container validate at deployment time that the referenced, named bean actually exists.

References to Other Beans (Collaborators)

- The final element inside a <constructor-arg/> or <property/> definition element.
- You set the value of the specified property of a bean to be a reference to another bean (a collaborator) managed by the container.
- Scoping and validation of collaborators as follows:
- bean (most general form) <ref bean="collaboratorRef"/>
  - Any bean in the same container or parent container
  - Regardless of whether it is in the same XML file
  - Could be **bean id** or with this **bean name**
- local <ref local="collaboratorRef"/>
  - In the same XML file
  - is no longer supported in the 4.0 beans XSD, since it does not provide value over a regular bean reference any more.

References to Other Beans (Collaborators)(EX.)

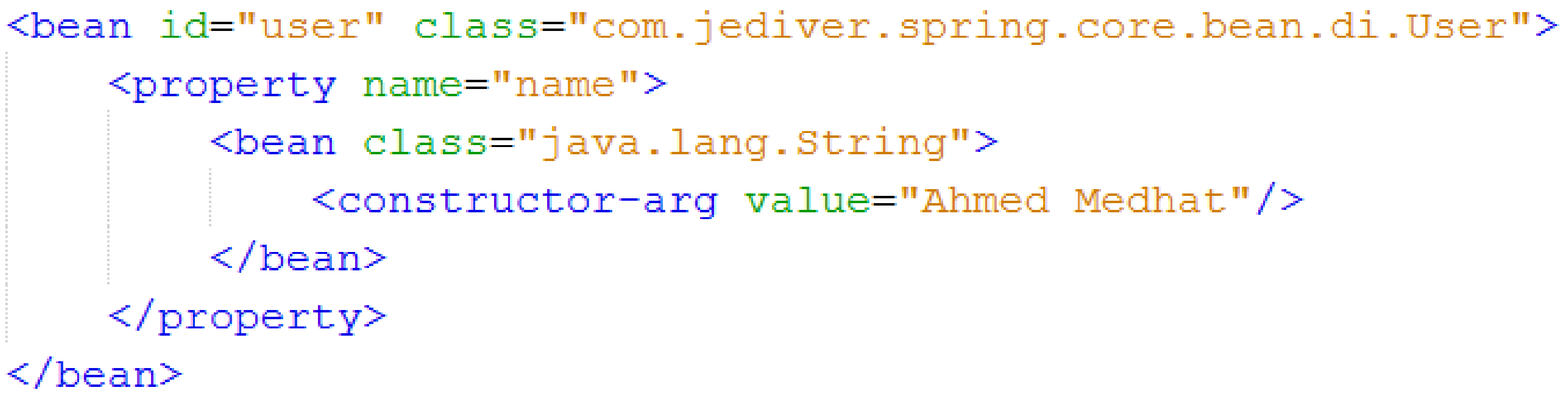
- parent
  - Any bean in a parent container
  - Regardless of whether it is in the same XML file
  - Could be **bean id** or with this **bean name**
  - Most common used in AOP (Context for beans without aspects and another with aspects)



```
<bean id="accountService" class="com.something.SimpleAccountService">
</bean>
<bean id="accountService"
      class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
        <ref parent="accountService"/>
    </property>
</bean>
```

Inner beans

- A <bean/> element inside the <property/> or <constructor-arg/> elements.
- An inner bean definition does not need to have any id or name defined.
  - **Note:** in case of inner beans,
    - The 'scope' flag is ignored.
    - Inner beans are always scoped as prototypes.
    - It is not possible to inject inner beans into beans other than the enclosing bean.

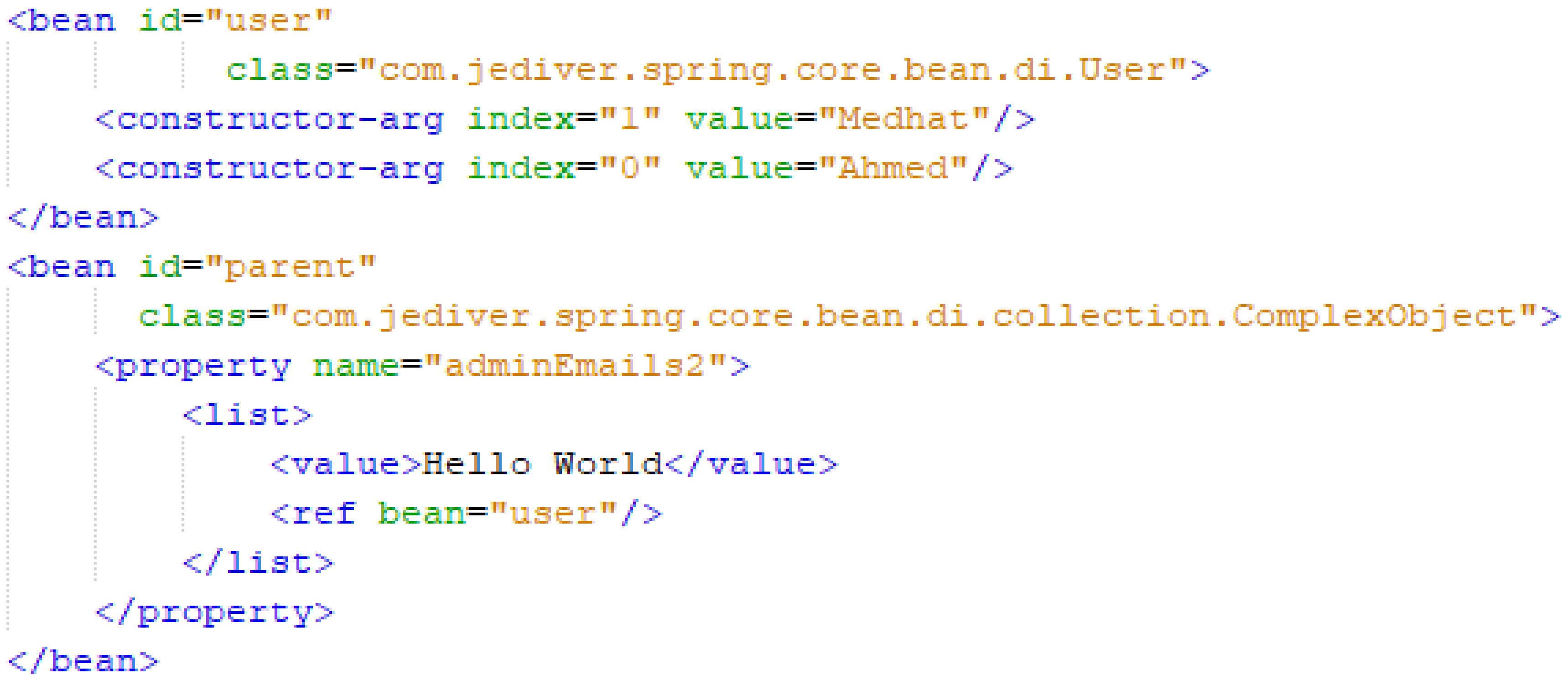


```
<bean id="user" class="com.jediver.spring.core.bean.di.User">
    <property name="name">
        <bean class="java.lang.String">
            <constructor-arg value="Ahmed Medhat"/>
        </bean>
    </property>
</bean>
```

Collections

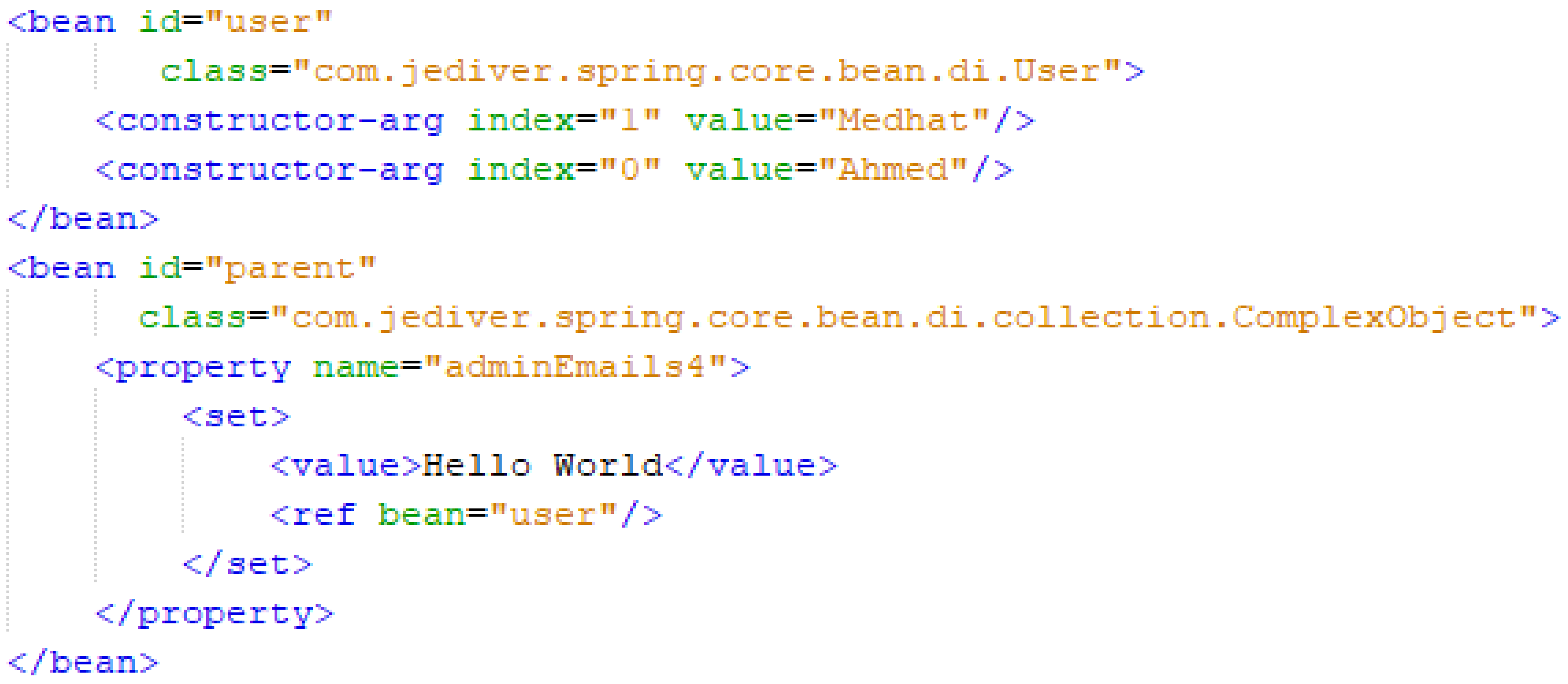
- The Collections are:
  - <list/> element.
  - <set/> element.
  - <map/> element.
  - <props/> element.

Collections <list/> element.



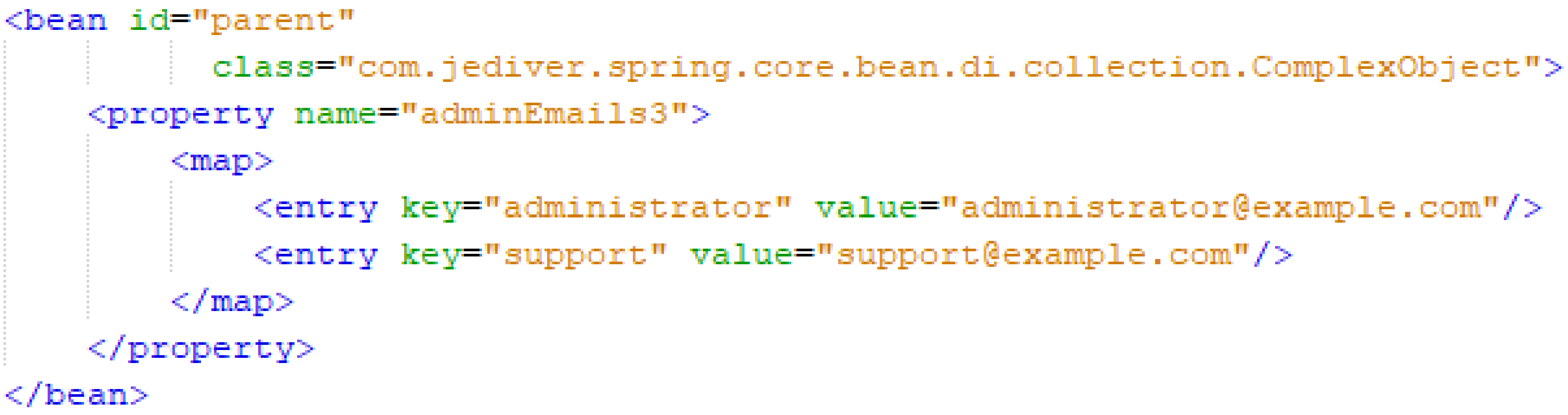
```
<bean id="user"
          class="com.jediver.spring.core.bean.di.User">
    <constructor-arg index="1" value="Medhat"/>
    <constructor-arg index="0" value="Ahmed"/>
</bean>
<bean id="parent"
      class="com.jediver.spring.core.bean.di.collection.ComplexObject">
    <property name="adminEmails2">
        <list>
            <value>Hello World</value>
            <ref bean="user"/>
        </list>
    </property>
</bean>
```

Collections <set/> element.



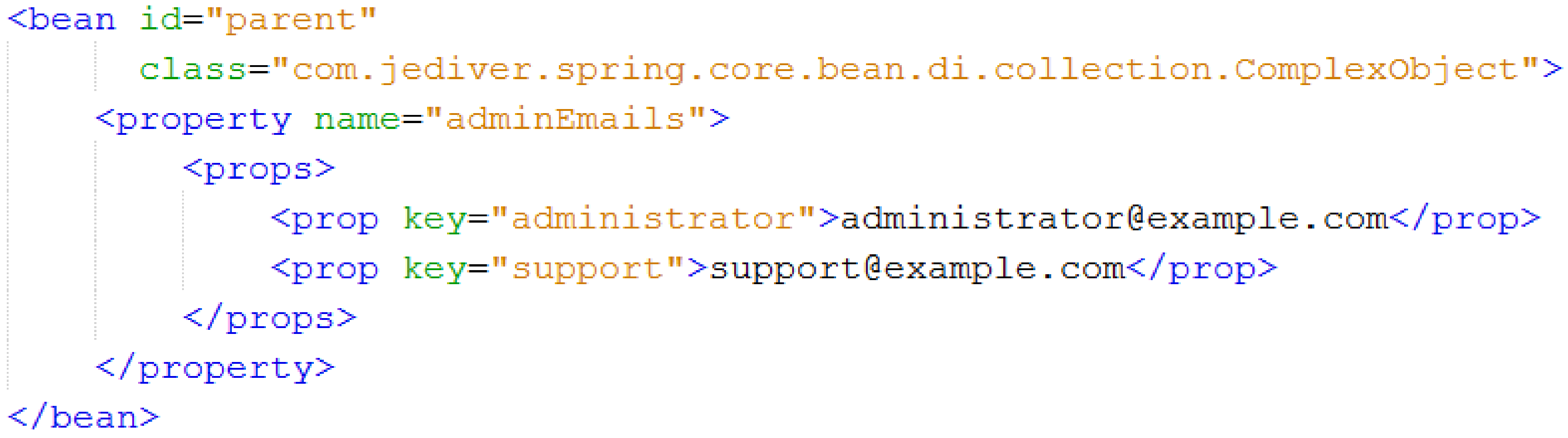
```
<bean id="user"
        class="com.jediver.spring.core.bean.di.User">
    <constructor-arg index="1" value="Medhat"/>
    <constructor-arg index="0" value="Ahmed"/>
</bean>
<bean id="parent"
      class="com.jediver.spring.core.bean.di.collection.ComplexObject">
    <property name="adminEmails4">
        <set>
            <value>Hello World</value>
            <ref bean="user"/>
        </set>
    </property>
</bean>
```

Collections <map/> element.



```
<bean id="parent"
          class="com.jediver.spring.core.bean.di.collection.ComplexObject">
    <property name="adminEmails3">
        <map>
            <entry key="administrator" value="administrator@example.com"/>
            <entry key="support" value="support@example.com"/>
        </map>
    </property>
</bean>
```

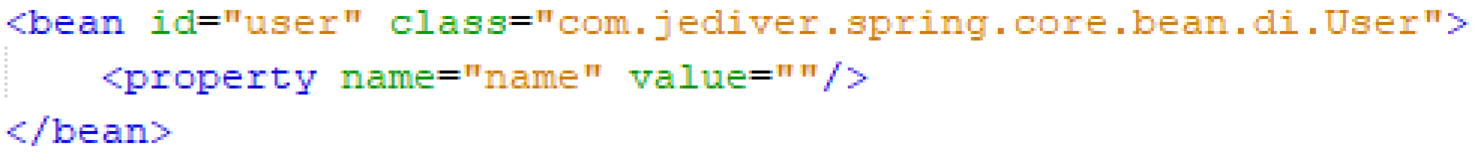
Collections <props/> element.



```
<bean id="parent"
        class="com.jediver.spring.core.bean.di.collection.ComplexObject">
    <property name="adminEmails">
        <props>
            <prop key="administrator">administrator@example.com</prop>
            <prop key="support">support@example.com</prop>
        </props>
    </property>
</bean>
```

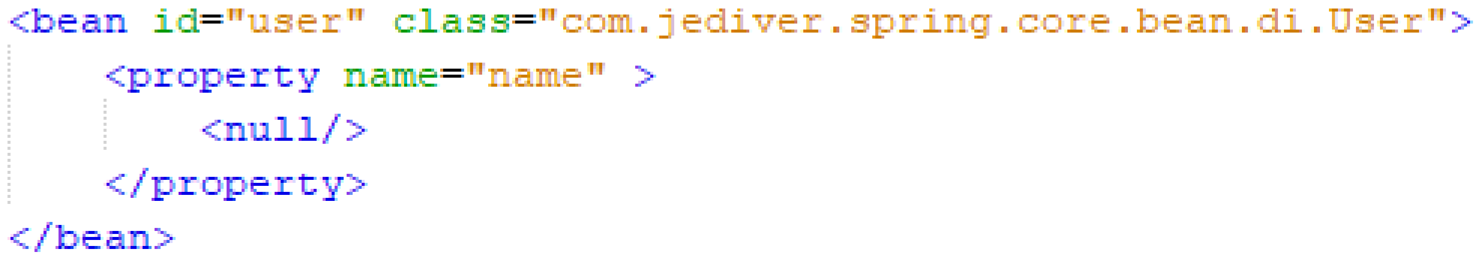
Null Element

- Empty arguments for properties like as empty Strings.
- The following configuration sets the name property to the empty String value ("").



```
<bean id="user" class="com.jediver.spring.core.bean.di.User">
    <property name="name" value=""/>
</bean>
```

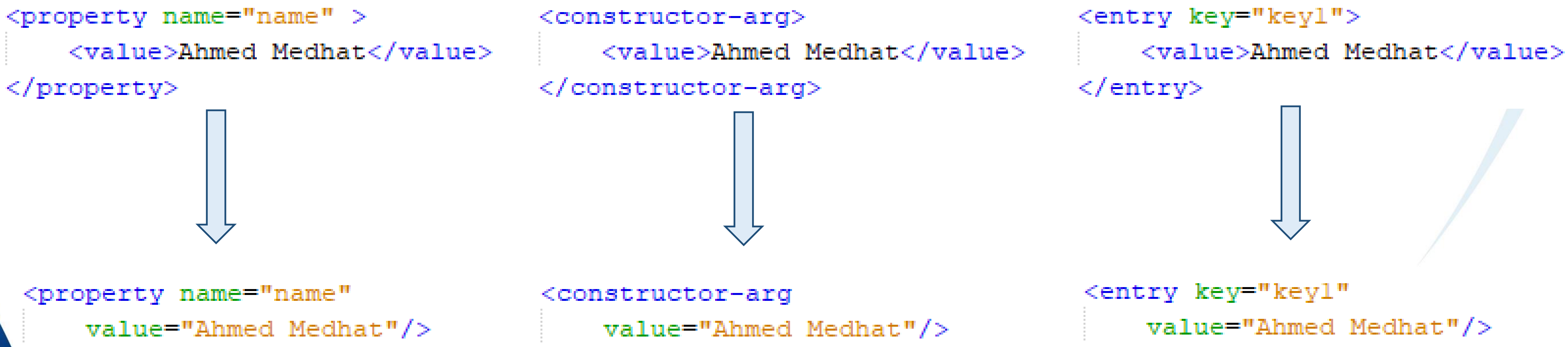
- The <null/> element is used to handle null values.



```
<bean id="user" class="com.jediver.spring.core.bean.di.User">
    <property name="name" >
        <null/>
    </property>
</bean>
```

XML Shortcuts

- The <property/>, <constructor-arg/>, and <entry/> elements
- All support a 'value' attribute which may be used instead of embedding a full <value/> element.
- Examples:



```
<property name="name" >
    <value>Ahmed Medhat</value>
</property>
```

```
<property name="name"
    value="Ahmed Medhat"/>
```

```
<constructor-arg>
    <value>Ahmed Medhat</value>
</constructor-arg>
```

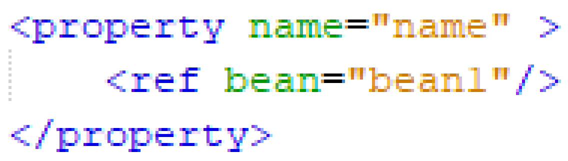
```
<constructor-arg
    value="Ahmed Medhat"/>
```

```
<entry key="key1">
    <value>Ahmed Medhat</value>
</entry>
```

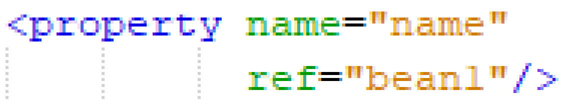
```
<entry key="key1"
    value="Ahmed Medhat"/>
```

XML Shortcuts (Ex.)

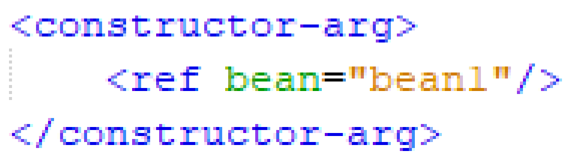
- The <property/>, <constructor-arg/> support a 'ref' attribute which may be used instead of embedding a full <ref/> element.
- The <entry/> elements allows a shortcut form to specify the key and/or value of the map, in the form of the 'key' / 'key-ref' and 'value' / 'value-ref' attributes
- Examples:



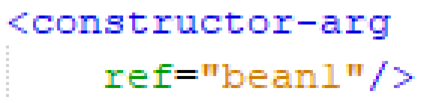
```
<property name="name" >
    <ref bean="bean1"/>
</property>
```



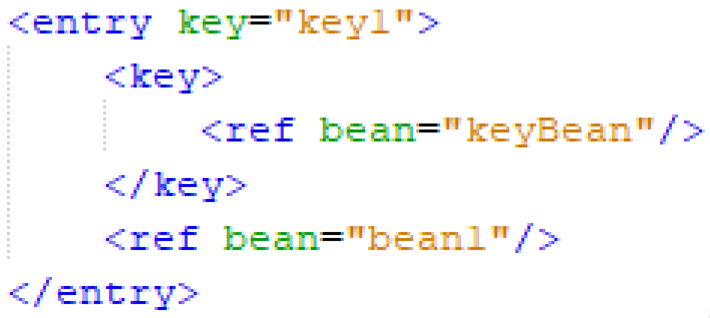
```
<property name="name"
          ref="bean1"/>
```



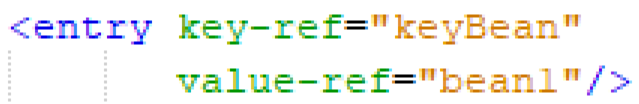
```
<constructor-arg>
    <ref bean="bean1"/>
</constructor-arg>
```



```
<constructor-arg
    ref="bean1"/>
```



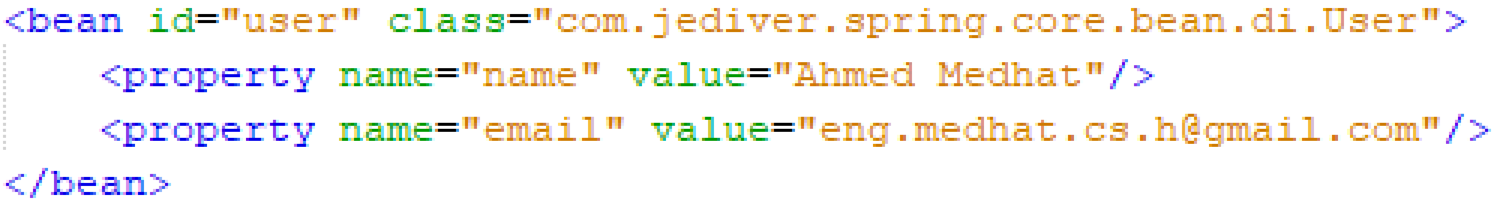
```
<entry key="key1">
    <key>
        <ref bean="keyBean"/>
    </key>
    <ref bean="bean1"/>
</entry>
```



```
<entry key-ref="keyBean"
       value-ref="bean1"/>
```

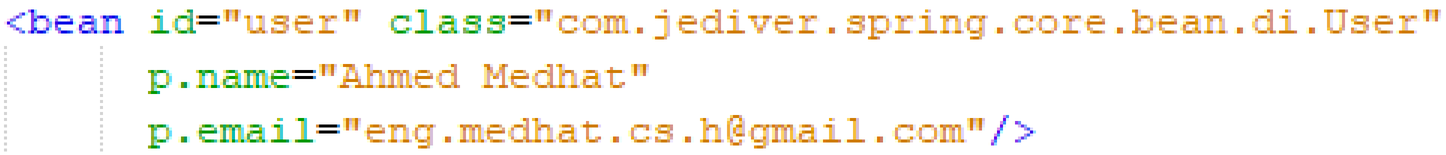
XML Shortcut with the p-namespace

- The p-namespace lets you use the bean element’s attributes (instead of nested <property/> elements) to describe your property values collaborating beans, or both.
- Import namespace (xmlns:p="http://www.springframework.org/schema/p")
- The following Example shows the standard XML format



```xml
<bean id="user" class="com.jediver.spring.core.bean.di.User">
    <property name="name" value="Ahmed Medhat"/>
    <property name="email" value="eng.medhat.cs.h@gmail.com"/>
</bean>
```

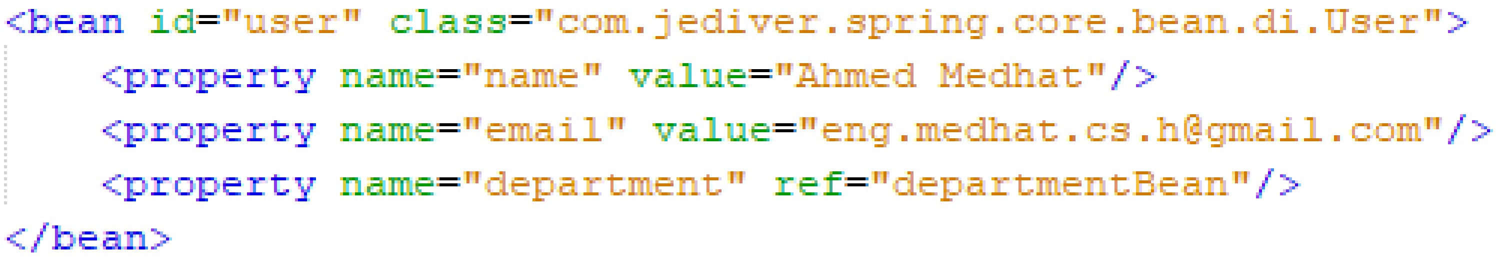
- The following Example shows the uses of p-namespace



```xml
<bean id="user" class="com.jediver.spring.core.bean.di.User"
      p.name="Ahmed Medhat"
      p.email="eng.medhat.cs.h@gmail.com"/>
```

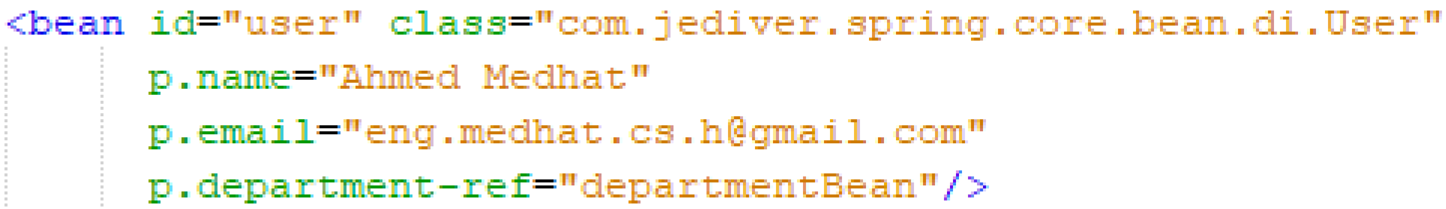
XML Shortcut with the p-namespace (Ex.)

- This next example includes two more bean definitions that have a reference to another bean:
- The following Example shows the standard XML format



```
<bean id="user" class="com.jediver.spring.core.bean.di.User">
    <property name="name" value="Ahmed Medhat"/>
    <property name="email" value="eng.medhat.cs.h@gmail.com"/>
    <property name="department" ref="departmentBean"/>
</bean>
```

- The following Example shows the uses of p-namespace

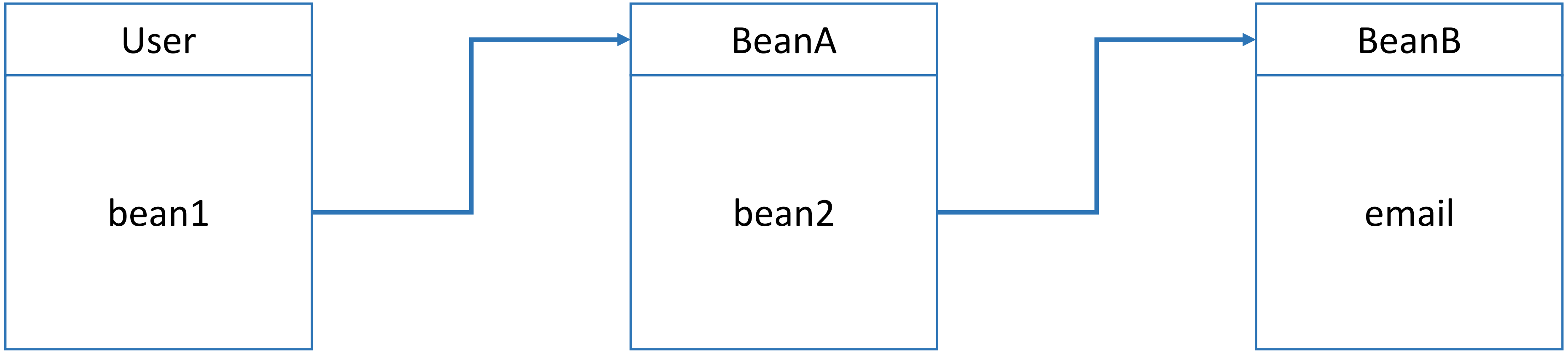


```
<bean id="user" class="com.jediver.spring.core.bean.di.User"
      p.name="Ahmed Medhat"
      p.email="eng.medhat.cs.h@gmail.com"
      p.department-ref="departmentBean"/>
```

- In this case, department is the property name, whereas the -ref part indicates that this is not a straight value but rather a reference to another bean.

Compound property names

- You can use compound or nested property names when you set bean properties.
- As long as all components of the path except the final property name are not null.
- The following Example:



User

bean1

BeanA

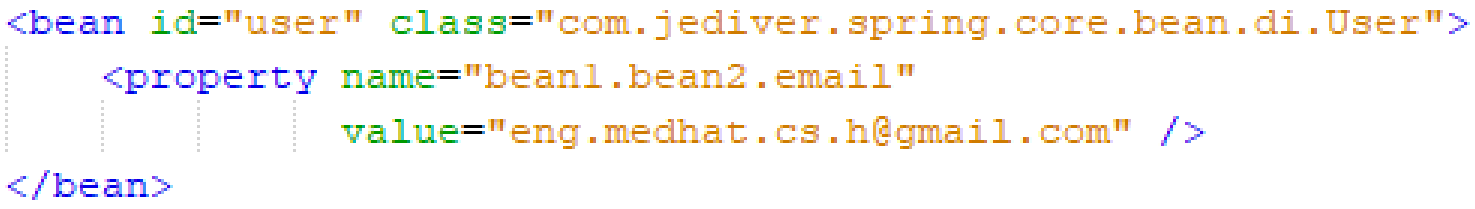
bean2

BeanB

email

Compound property names

- The following Example:



```
<bean id="user" class="com.jediver.spring.core.bean.di.User">
    <property name="bean1.bean2.email"
              value="eng.medhat.cs.h@gmail.com" />
</bean>
```

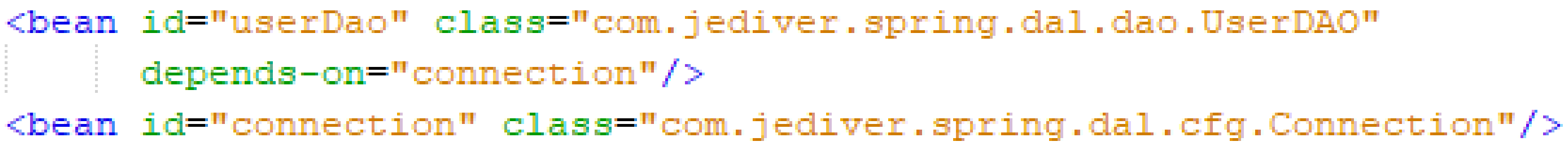
- The user bean has a bean1 property, which has a bean2 property, which has a email property.
- And that final email property is being set to a value of "eng.medhat.cs.h@gmail.com".
- In order for this to work, the bean1 property of user and the bean2 property of bean1 must not be null after the bean is constructed. Otherwise, a NullPointerException is thrown.

Using depends-on

- If a bean is a dependency of another bean, that usually means that one bean is set as a property of another. Typically you accomplish this with the <ref/> element in XML-based configuration metadata.
- However, sometimes dependencies between beans are less direct.
- An example is when a static initializer in a class needs to be triggered, such as for database driver registration.
- The "depends-on" attribute can explicitly force one or more beans to be initialized before the bean using this element is initialized.

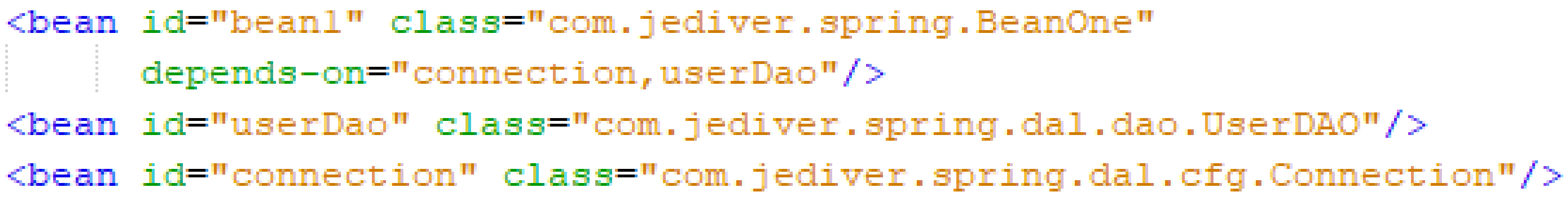
Using depends-on (Ex.)

- The following example uses the depends-on attribute to express a dependency on a single bean:



```
<bean id="userDao" class="com.jediver.spring.dal.dao.UserDAO"
        depends-on="connection"/>
<bean id="connection" class="com.jediver.spring.dal.cfg.Connection"/>
```

- We can use (commas, whitespace, and semicolons) as delimiters to express a dependency on multiple beans, Another Example:



```
<bean id="bean1" class="com.jediver.spring.BeanOne"
        depends-on="connection,userDao"/>
<bean id="userDao" class="com.jediver.spring.dal.dao.UserDAO"/>
<bean id="connection" class="com.jediver.spring.dal.cfg.Connection"/>
```

Autowiring collaborators

- The Spring container can autowire relationships between collaborating beans.
- You can let Spring resolve collaborators (other beans) automatically for your bean by ApplicationContext.
- Autowiring has the following advantages:
  - Reduce the need to specify properties or constructor arguments.
  - Update a configuration as your objects evolve.
    - For example, if you need to add a dependency to a class, that dependency can be satisfied automatically without you needing to modify the configuration.

Autowiring collaborators (Ex.)

- You can specify the autowire mode for a bean definition with the "autowire" attribute of the <bean/> element.

- The autowiring functionality has four modes:

  1. **no (default)**

     - (Default) No autowiring.
     - Bean references must be defined by ref elements.
     - Changing the default setting is not recommended for larger deployments, because specifying collaborators explicitly gives greater control and clarity.

Autowiring collaborators (Ex.)

- The autowiring functionality has four modes:

2. byName

- Autowiring by property name.
- Container Wiring a bean with the same name as the property that needs to be autowired.
- For example, if a bean definition is set to autowire by name and it contains a userDao property (that is, it has a setUserDao(..) method), Spring looks for a bean definition named userDao and uses it to set the property.

Autowiring collaborators (Ex.)

- The autowiring functionality has four modes:

  3. byType

  - Autowiring by property type
  - Container Wiring if exactly one bean of the property type exists in the container.
  - If more than one exists, a fatal exception is thrown, which indicates that you may not use byType autowiring for that bean.
  - If there are no matching beans, nothing happens (the property is not set).

Autowiring collaborators (Ex.)

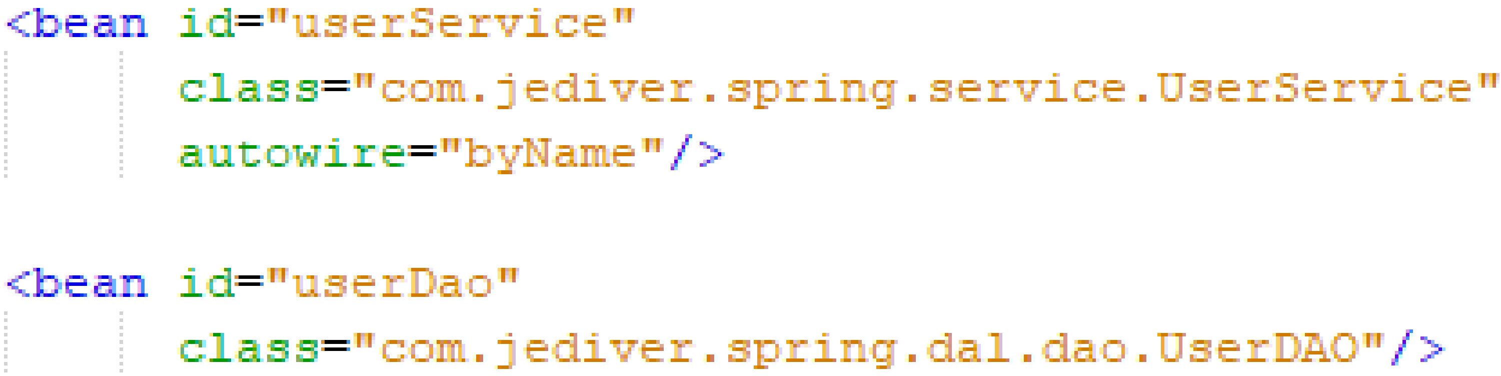
- The autowiring functionality has four modes:

4. **constructor**

   - Autowiring by property type in constructor arguments
   - Container Wiring byType but applies to constructor arguments.
   - If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.

Autowiring collaborators (Ex.)

- The following Example using autowiring "byName":
- UserService Class contains property called "userDao".



```
<bean id="userService"
        class="com.jediver.spring.service.UserService"
        autowire="byName"/>

<bean id="userDao"
        class="com.jediver.spring.dal.dao.UserDAO"/>
```

- Container will look for bean named as "userDao" and wire this instance with the property "userDao" by calling userService.setUserDao(userDao);

Autowiring collaborators (Ex.)

- With byType or constructor autowiring mode, you can wire arrays and typed collections.
- In such cases, all autowire candidates within the container that match the expected type are provided to satisfy the dependency.
  - For Example: List<UserDao> userDaos;
  - Container will look for any bean with type UserDao and wire it into this list.
- You can autowire instances in Map if the expected key type is String.
  - For Example: Map<String, UserDao> userDaos;
  - Container will look for any bean with type UserDao and wire it into this map with name as key.
- An autowired instances based on:
  - Map instance's values consist of all bean instances that match the expected type.
  - And the Map instance's keys contain the corresponding bean names.

Autowiring collaborators (Ex.)

- **Advantages of Autowiring**
  - Autowiring can reduce the amount of configuration required:
  - You just write ‘autowire’ attribute only, and You don’t need to manually inject beans into each other by write <Property> or <constructor-arg> elements.
  - Autowiring can cause configuration to keep itself up to date as your objects evolve:
    - Fully Integrated with Observer Design Pattern.
    - Like add new definition for BeanA, it autowired in List<BeanA>

Autowiring collaborators (Ex.)

- **Limitations and Disadvantages of Autowiring**
  - Explicit dependencies in property and constructor-arg settings always override autowiring.
  - You cannot autowire simple properties such as primitives, Strings, and Classes (and arrays of such simple properties).
  - Autowiring is less exact than explicit wiring. Spring is careful to avoid guessing in case of ambiguity that might have unexpected results.
    - The relationships between your Spring-managed objects are no longer documented explicitly.

Autowiring collaborators (Ex.)

- **Limitations and Disadvantages of Autowiring (Ex.)**
  - Wiring information may not be available to tools that may generate documentation from a Spring container.
  - Multiple bean definitions within the container may match the type specified by the setter method or constructor argument to be autowired.
    - For arrays, collections, or Map instances, this is not necessarily a problem. However, for dependencies that expect a single value, this ambiguity is not arbitrarily resolved. If no unique bean definition is available, an exception is thrown.

Autowiring collaborators (Ex.)

- **Resolve The Autowiring disadvantage**
  - Don’t use autowiring in favor of explicit wiring.
  - Avoid autowiring for a bean definition by setting its autowire-candidate attributes to false.
  - Designate a single bean definition as the primary candidate by setting the primary attribute of its <bean/> element to true.
  - Implement the more fine-grained control available with annotation-based configuration.
  - Limit autowire candidates based on pattern-matching against bean names, by define top-level <beans/> element accepts one or more patterns within its default-autowire-candidates attribute.

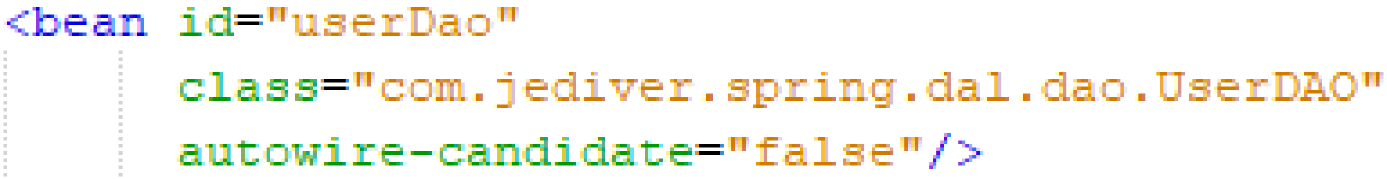
Manage a Bean in Autowiring

- You can exclude a bean from autowiring.
- By the autowire-candidate attribute of the <bean/> element to false.
- The container makes that specific bean definition unavailable to the autowiring infrastructure (including annotation style configurations such as @Autowired).
- The autowire-candidate attribute is designed to only affect type-based autowiring.
- It does not affect explicit references by name, which get resolved even if the specified bean is not marked as an autowire candidate.

Manage a Bean in Autowiring (Ex.)

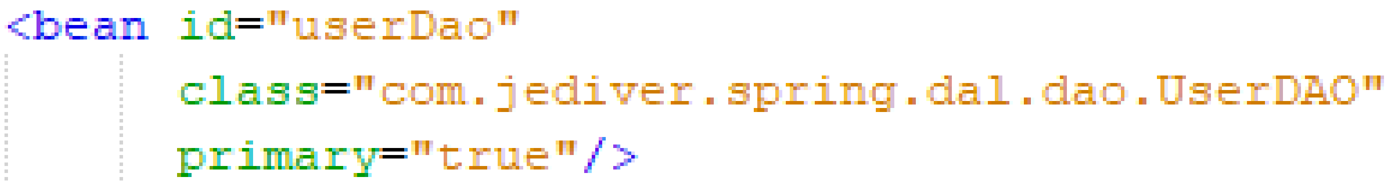
- You can manage bean in wiring by:

1. Excluding a bean from being an autowire candidate by



```
<bean id="userDao"
        class="com.jediver.spring.dal.dao.UserDAO"
        autowire-candidate="false"/>
```

2. Making a bean the only candidate for autowiring by



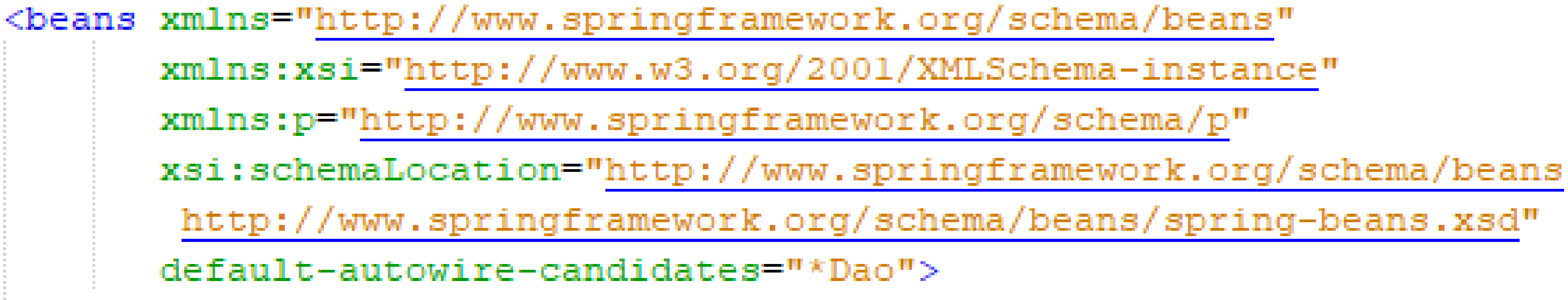
```
<bean id="userDao"
        class="com.jediver.spring.dal.dao.UserDAO"
        primary="true"/>
```

Manage a Bean in Autowiring (Ex.)

- You can manage bean in wiring by:

3. Limit autowire candidates based on pattern-matching against bean names, by define top-level

   <beans/> element accepts one or more patterns within its default-autowire-candidates attribute

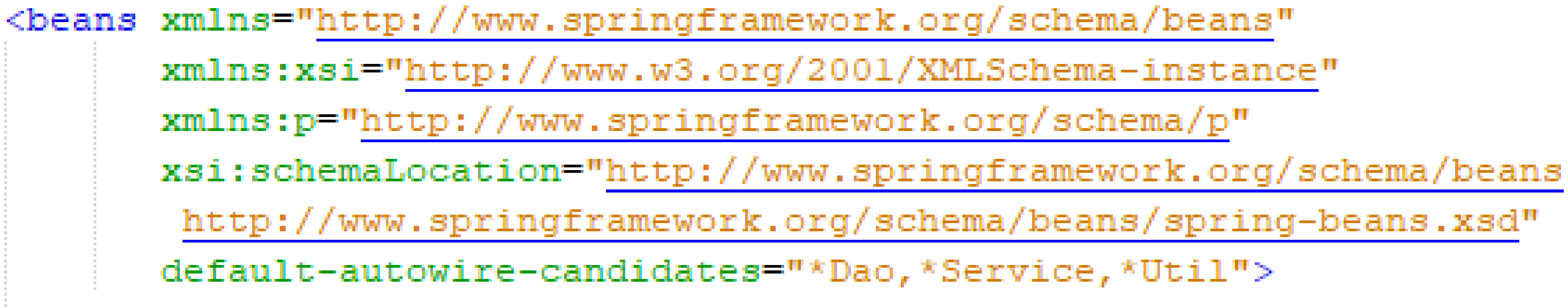


```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd"
       default-autowire-candidates="*Dao">
```

- Limit the autowiring of beans in which its names ends with "Dao".

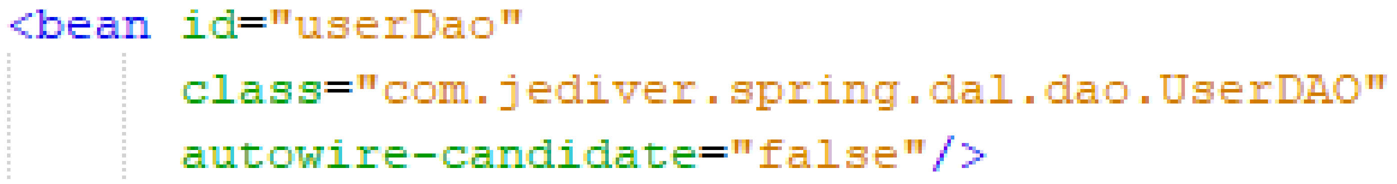
Manage a Bean in Autowiring (Ex.)

- You can mix one or more management.
- Limit the autowiring of beans in which its names ends with "Dao" or "Service" or "Util" .



```xml
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:p="http://www.springframework.org/schema/p"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd"
        default-autowire-candidates="*Dao,*Service,*Util">
```

- Exclude this bean "userDao" from autowire candidates.



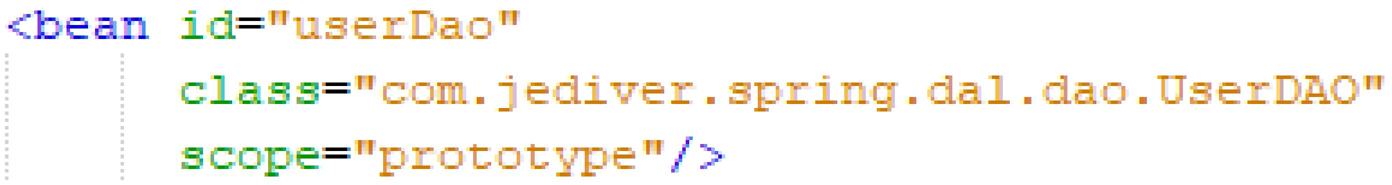
```xml
<bean id="userDao"
        class="com.jediver.spring.dal.dao.UserDAO"
        autowire-candidate="false"/>
```

Lesson 7

Core Container (Bean Scopes)

Bean Scopes

- When you create a bean definition, you create a recipe for creating actual instances of the class defined by that bean definition.
- The idea that a bean definition is a recipe is important, because it means that, as with a class, you can create many object instances from a single recipe.
- You can control the scope of the objects created from a particular bean definition.
- It gives you the flexibility to choose the scope of the objects through configuration
- You can define scope by "scope" attribute in <bean />



```
<bean id="userDao"
      class="com.jediver.spring.dal.dao.UserDAO"
      scope="prototype"/>
```

Bean Scopes

- The Spring Framework supports six scopes:
  - You can also create a custom scope.

1. **singleton**
   - (Default) Scopes a single bean definition to a single object instance for each Spring container.
2. **prototype**
   - Scopes a single bean definition to any number of object instances.
3. **request**
   - Scopes a single bean definition to the lifecycle of a single HTTP request.
   - That is, each HTTP request has its own instance of a bean created off the back of a single bean definition.
   - Only valid in the context of a web-aware Spring ApplicationContext.

Bean Scopes

- The Spring Framework supports six scopes:

4. **session**
   - Scopes a single bean definition to the lifecycle of an HTTP Session.
   - Only valid in the context of a web-aware Spring ApplicationContext.
5. **application**
   - Scopes a single bean definition to the lifecycle of a ServletContext.
   - Only valid in the context of a web-aware Spring ApplicationContext.
6. **websocket**
   - Scopes a single bean definition to the lifecycle of a WebSocket.
   - Only valid in the context of a web-aware Spring ApplicationContext.

Bean Scopes

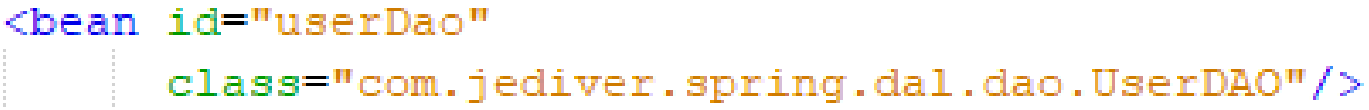
1. **singleton**

    - The singleton scope is the default scope in Spring.
    - Only one shared instance of a singleton bean is managed.
    - When you define a bean definition and it is scoped as a singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition.
    - This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.
    - A singleton from Spring Container VS. A singleton as defined in the Gang of Four (GoF) patterns.
        - The GoF singleton hard-codes the scope of an object such that one and only one instance of a particular class is created per ClassLoader.
        - The scope of the Spring singleton is best described as being per-container and per-bean.

Bean Scopes

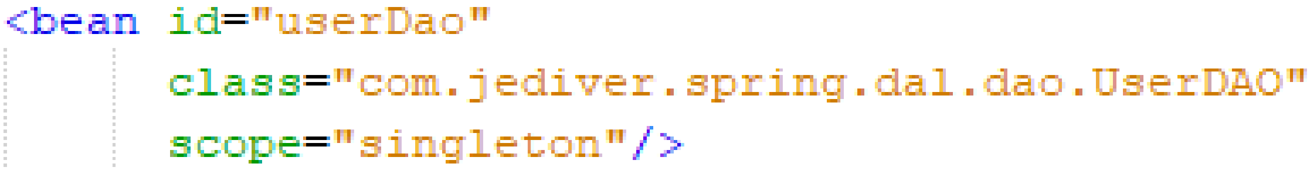
1. **singleton**

- The singleton scope is the default scope in Spring.



```
<bean id="userDao"
        class="com.jediver.spring.dal.dao.UserDAO"/>
```

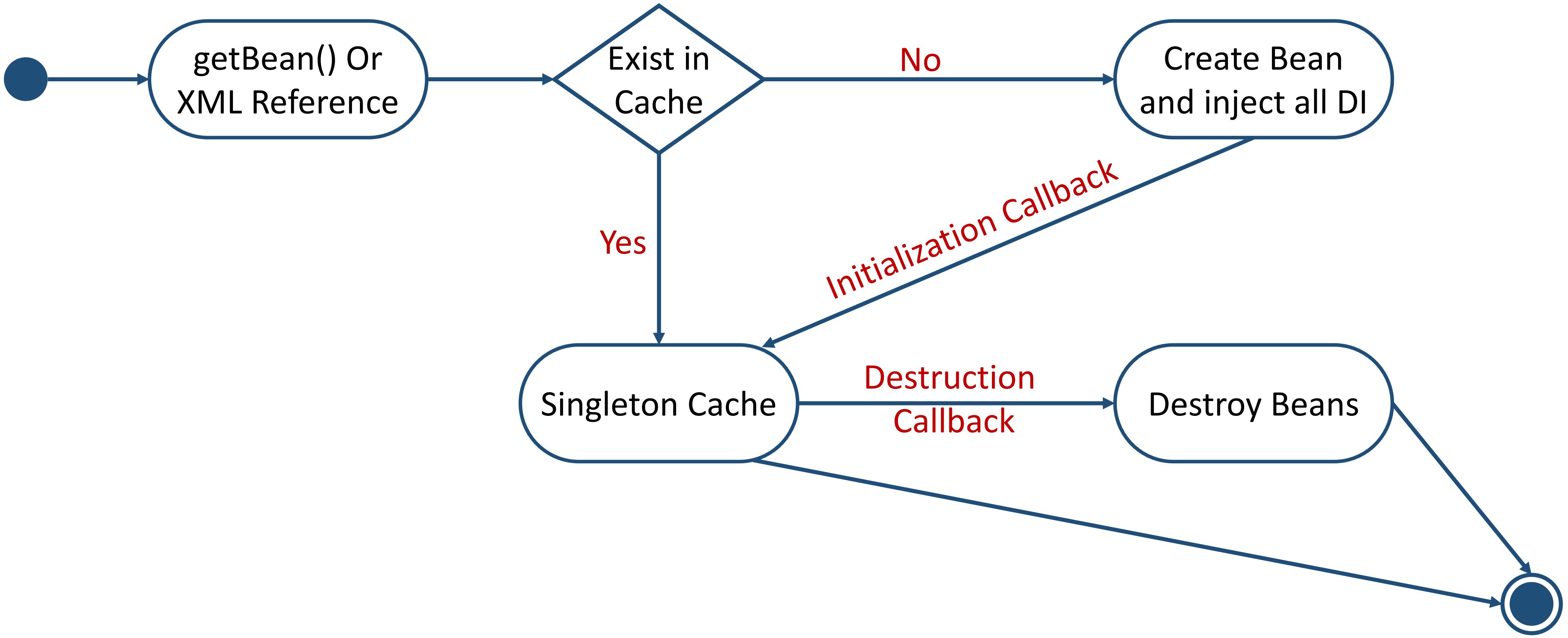
- To define a bean as a singleton in XML, you can define a bean as shown in the following example:



```
<bean id="userDao"
        class="com.jediver.spring.dal.dao.UserDAO"
        scope="singleton"/>
```

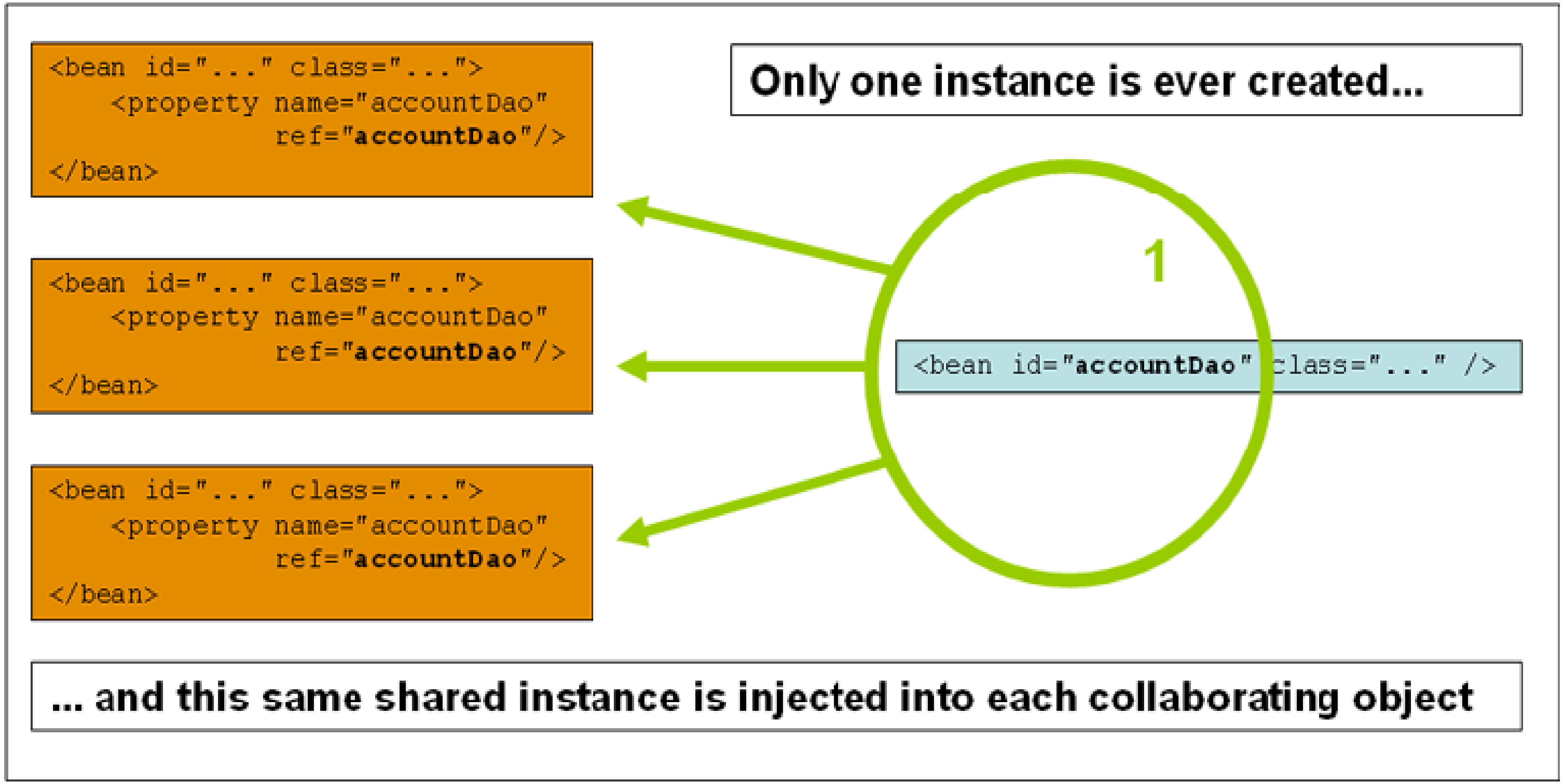
Bean Scopes

1. **singleton (Lifecycle)**



Bean Scopes

1. **singleton**



```
<bean id="..." class="...">
    <property name="accountDao"
              ref="accountDao"/>
</bean>
```

```
<bean id="..." class="...">
    <property name="accountDao"
              ref="accountDao"/>
</bean>
```

```
<bean id="..." class="...">
    <property name="accountDao"
              ref="accountDao"/>
</bean>
```

Only one instance is ever created...

```
<bean id="accountDao" class="..." />
```

... and this same shared instance is injected into each collaborating object

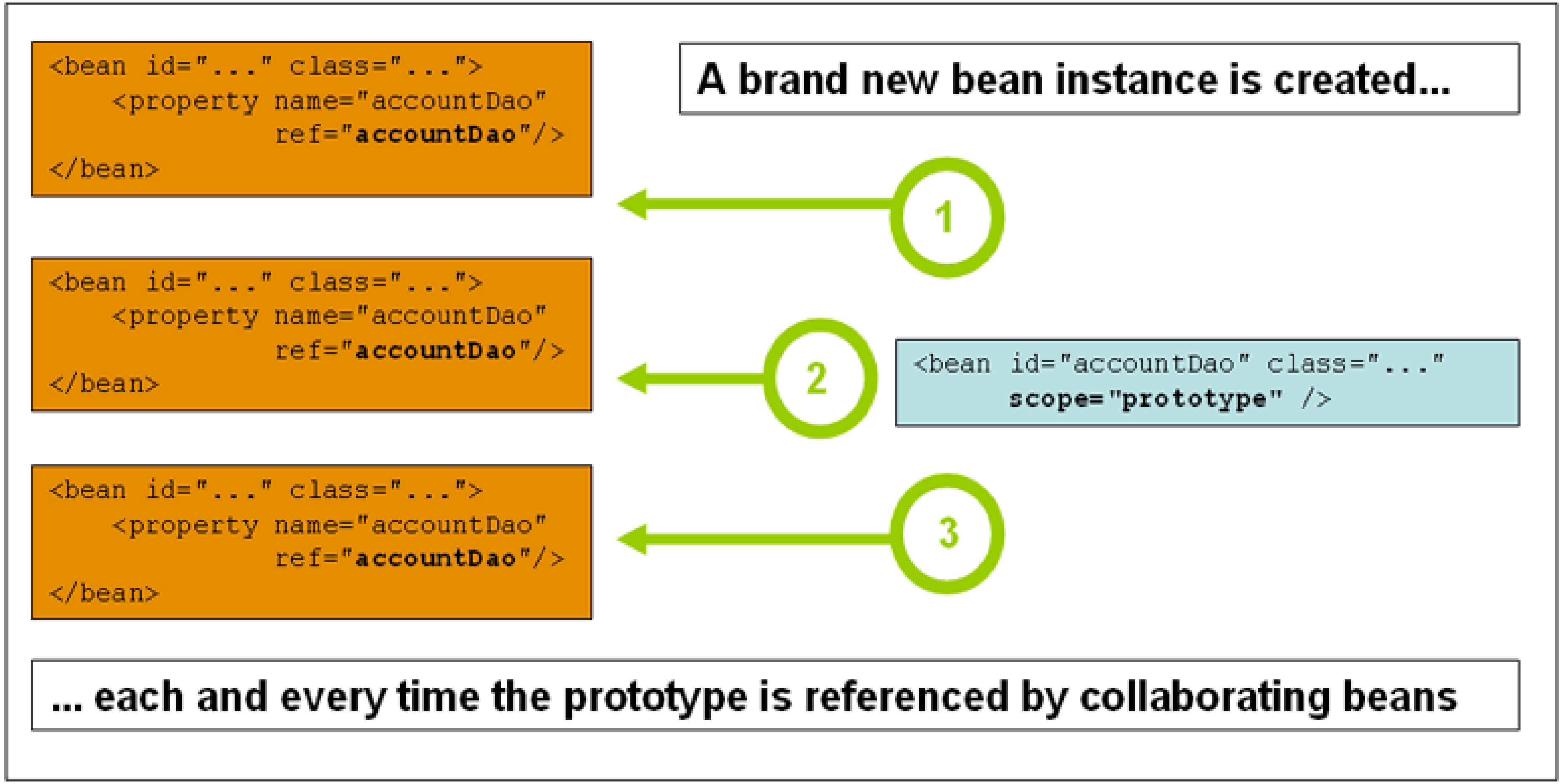
Bean Scopes

2. **prototype**

- The non-singleton prototype scope of bean deployment results in the creation of a new bean instance every time a request for that specific bean is made.
- That is, the bean is injected into another bean or you request it through a getBean() method call on the container.
- As a rule:
  - You should use the prototype scope for stateful beans.
  - You Should use the singleton scope for stateless beans.

Bean Scopes

2. **prototype**



A brand new bean instance is created...

```
<bean id="..." class="...">
    <property name="accountDao"
              ref="accountDao"/>
</bean>
```

```
<bean id="..." class="...">
    <property name="accountDao"
              ref="accountDao"/>
</bean>
```

```
<bean id="..." class="...">
    <property name="accountDao"
              ref="accountDao"/>
</bean>
```

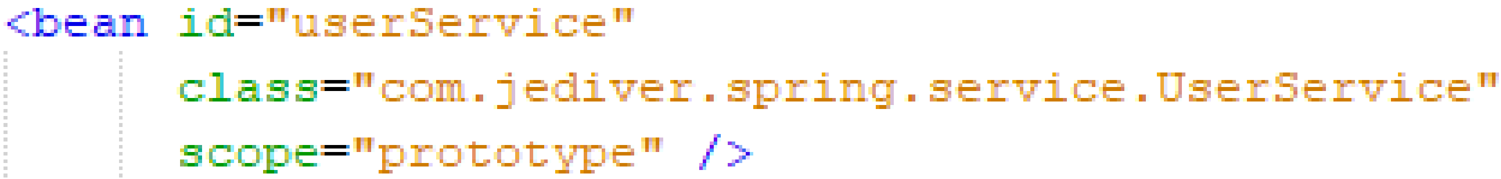
```
<bean id="accountDao" class="..."
      scope="prototype" />
```

... each and every time the prototype is referenced by collaborating beans

Bean Scopes

2. **prototype**

- The following example defines a bean as a prototype in XML:



```
<bean id="userService"
        class="com.jediver.spring.service.UserService"
        scope="prototype" />
```

- For Example:
  - A data access object (DAO) is not typically configured as a prototype, because a typical DAO does not hold any conversational state.

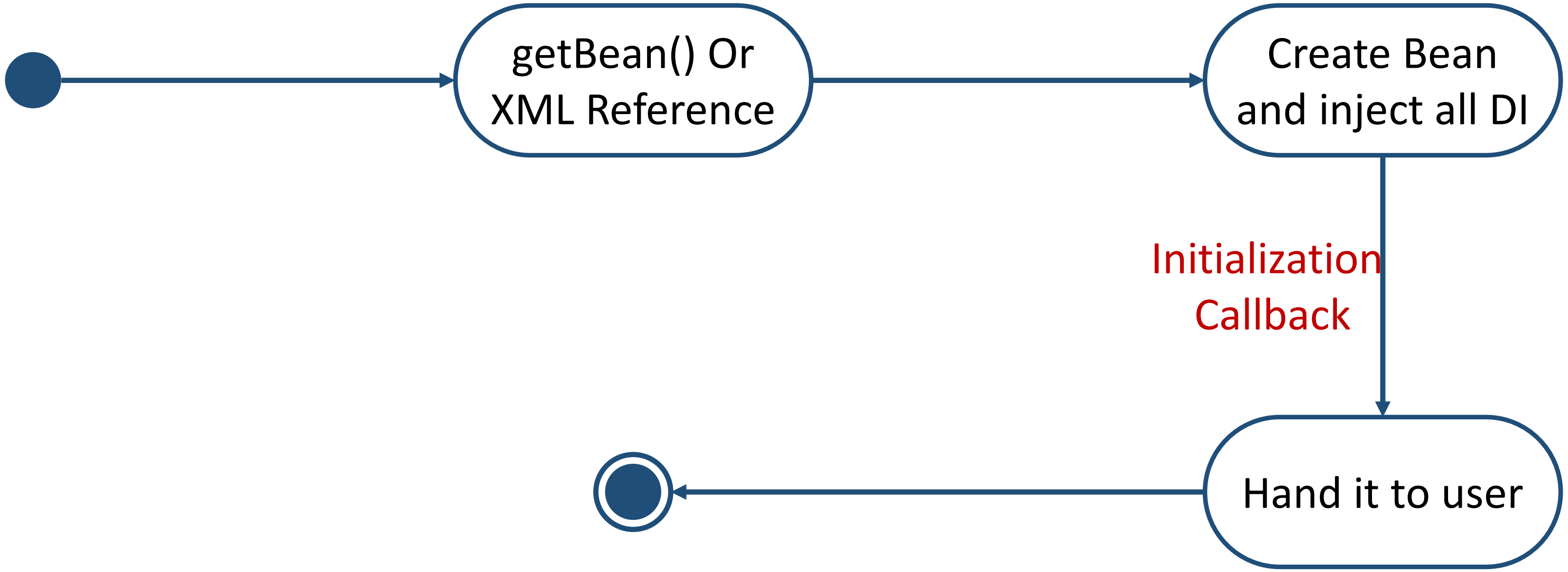
Bean Scopes

2. **prototype**

- In contrast to the other scopes, Spring does not manage the complete lifecycle of a prototype bean.
- The container instantiates, configures, and otherwise assembles a prototype object and hands it to the client, with no further record of that prototype instance.
- Initialization lifecycle callback methods are called on all objects regardless of scope.
- In the case of prototypes, Configured destruction lifecycle callbacks are not called.
- The client code must clean up prototype-scoped objects and release expensive resources that the prototype beans hold.
- To get the Spring container to release resources held by prototype-scoped beans, try using a custom bean post-processor, which holds a reference to beans that need to be cleaned up.

Bean Scopes

2. **prototype (Lifecycle)**



Bean Scopes

- **Singleton Beans with Prototype-bean Dependencies**
  - When you use singleton-scoped beans with dependencies on prototype beans, be aware that dependencies are resolved at instantiation time.
  - If you dependency-inject a prototype-scoped bean into a singleton-scoped bean, a new prototype bean is instantiated and then dependency-injected into the singleton bean.
  - However, suppose you want the singleton-scoped bean to acquire a new instance of the prototype-scoped bean repeatedly at runtime.
    - You cannot dependency-inject a prototype-scoped bean into your singleton bean, because that injection occurs only once, when the Spring container instantiates the singleton bean and resolves and injects its dependencies.
    - If you need a new instance of a prototype bean at runtime more than once, use **Method Injection**.

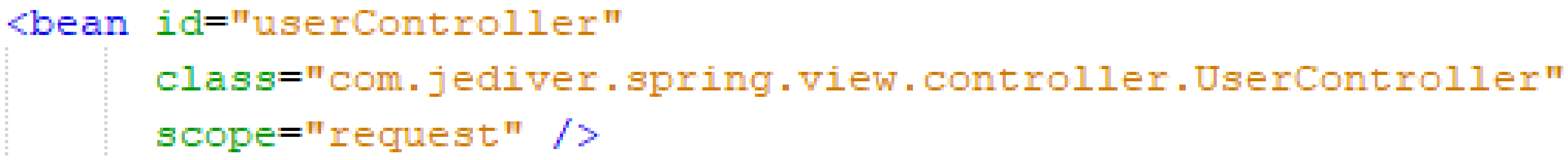
Bean Scopes

- The **request**, **session**, **application**, and **websocket** scopes are available only if you use a web-aware Spring ApplicationContext implementation.
- If you use these scopes with regular Spring IoC containers, such as the ClassPathXmlApplicationContext, an IllegalStateException that complains about an unknown bean scope is thrown.

Bean Scopes

3. **request**

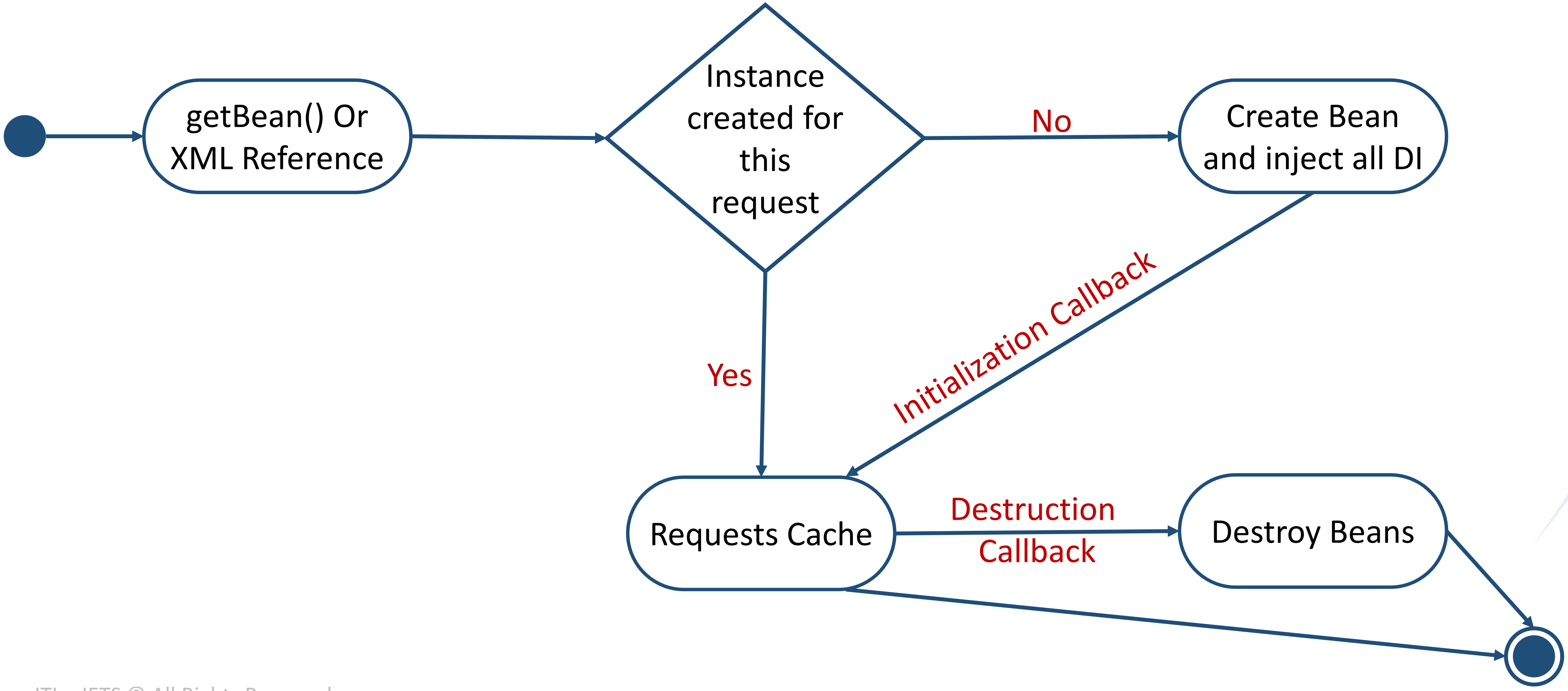
- The Spring container creates a new instance of the UserController bean by using the userController bean definition for each and every HTTP request **(Stateful Per Request)**.
- You can change the internal state of the instance that is created as much as you want, because other instances created from the same userController bean definition do not see these changes in state.
- They are particular to an individual request. When the request completes processing, the bean that is scoped to the request is discarded.
- Consider the following XML configuration for a bean definition:



```xml
<bean id="userController"
      class="com.jediver.spring.view.controller.UserController"
      scope="request" />
```

Bean Scopes

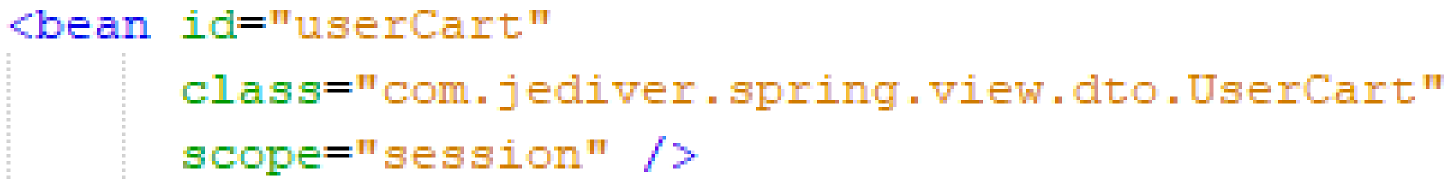
3. **request (Lifecycle)**


getBean() Or XML Reference
Instance created for this request
No
Create Bean and inject all DI
Yes
Initialization Callback
Requests Cache
Destruction Callback
Destroy Beans

Bean Scopes

4. **session**

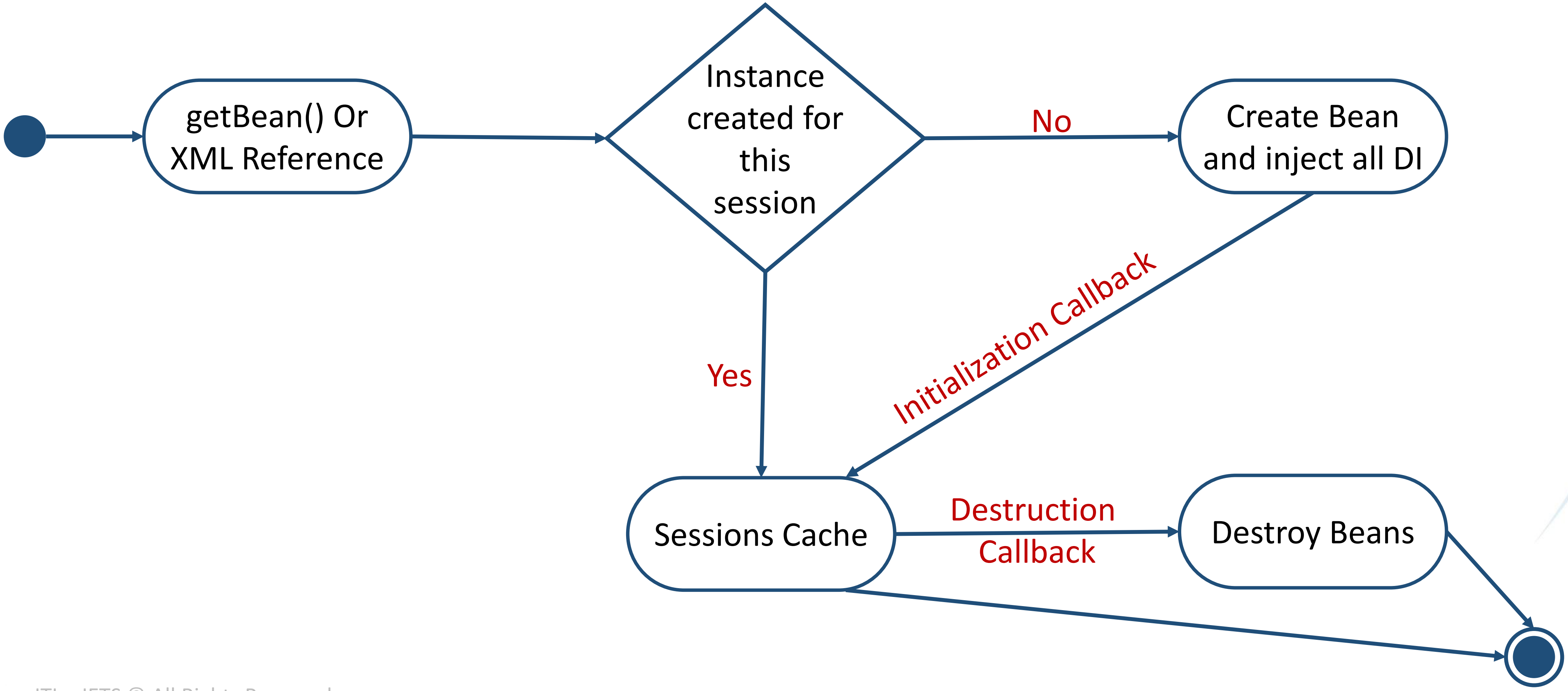
- The Spring container creates a new instance of the UserCart bean by using the userCart bean definition for the lifetime of a single HTTP Session **(Stateful per Session)**.
- When the HTTP Session is eventually discarded, the bean that is scoped to that particular HTTP Session is also discarded.
- Consider the following XML configuration for a bean definition:



```
<bean id="userCart"
      class="com.jediver.spring.view.dto.UserCart"
      scope="session" />
```

Bean Scopes

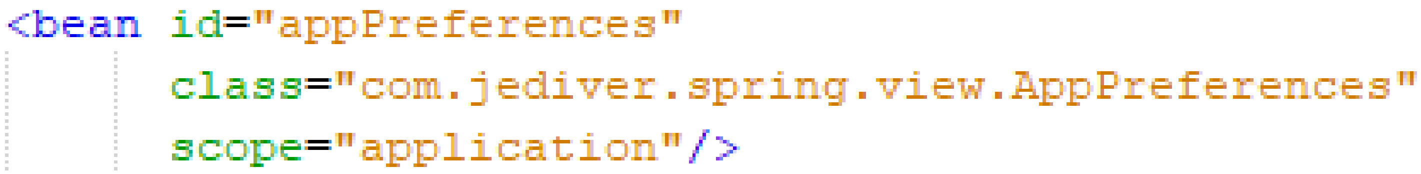
4. **session (Lifecycle)**


getBean() Or XML Reference
Instance created for this session
No
Create Bean and inject all DI
Yes
Initialization Callback
Sessions Cache
Destruction Callback
Destroy Beans

Bean Scopes

5. **application**

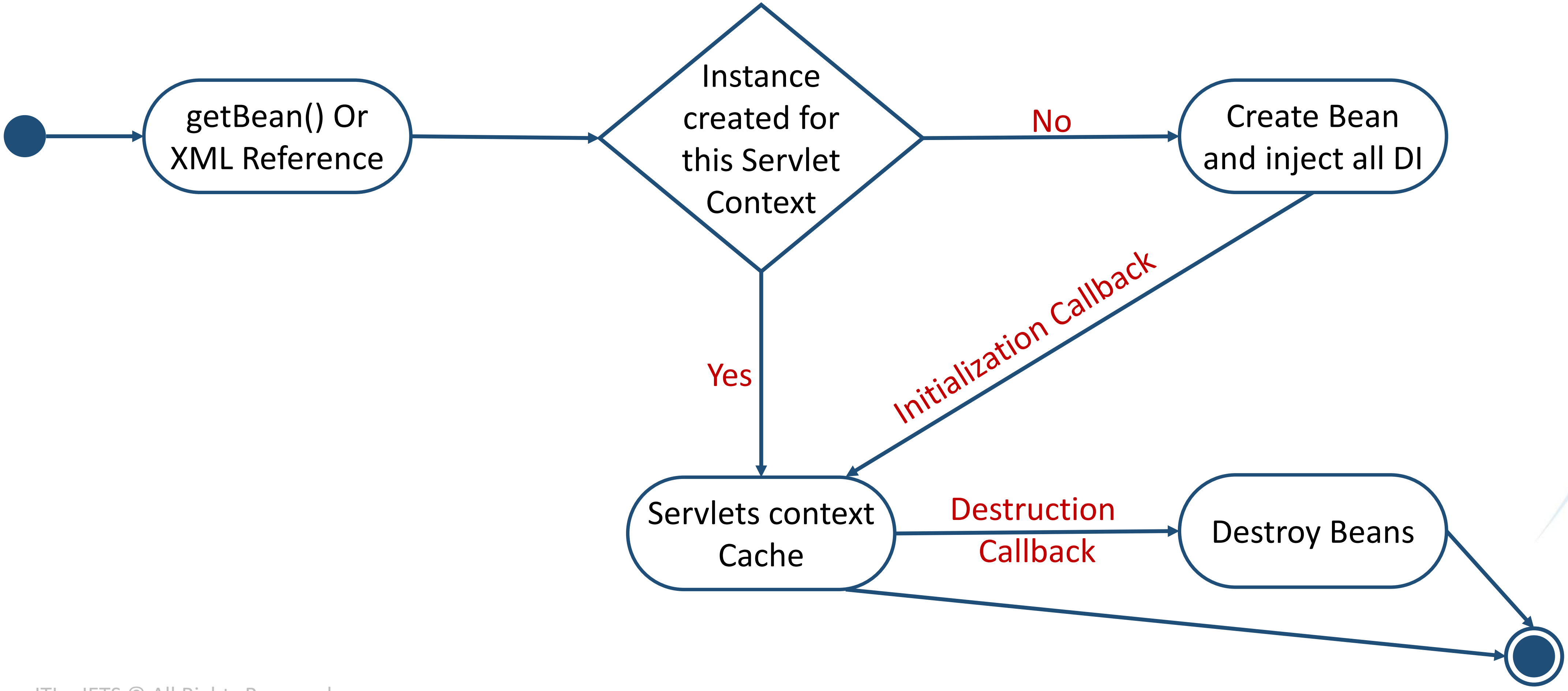
- The Spring container creates a new instance of the AppPreferences bean by using the appPreferences bean definition once for the entire web application.
- That is, the appPreferences bean is scoped at the ServletContext level.
- This is somewhat similar to a Spring singleton bean but differs in two important ways:
  - It is a singleton per ServletContext, not per Spring 'ApplicationContext' (for which there may be several in any given web application).
  - It is actually exposed and therefore visible as a ServletContext attribute.
- Consider the following XML configuration for a bean definition:



```
<bean id="appPreferences"
      class="com.jediver.spring.view.AppPreferences"
      scope="application"/>
```

Bean Scopes

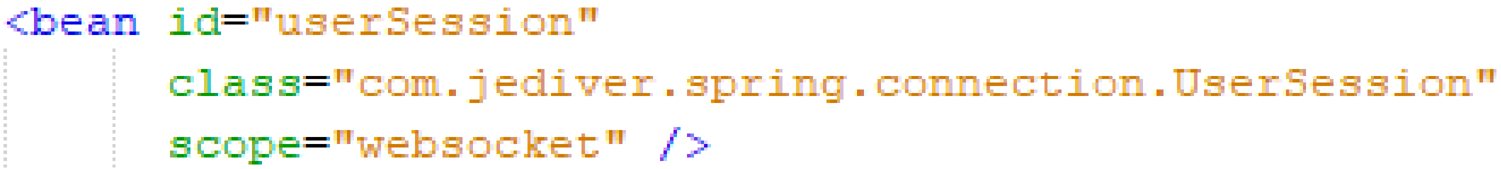
5. **application (Lifecycle)**


getBean() Or XML Reference
Instance created for this Servlet Context
No
Create Bean and inject all DI
Yes
Initialization Callback
Servlets context Cache
Destruction Callback
Destroy Beans

Bean Scopes

6. websocket

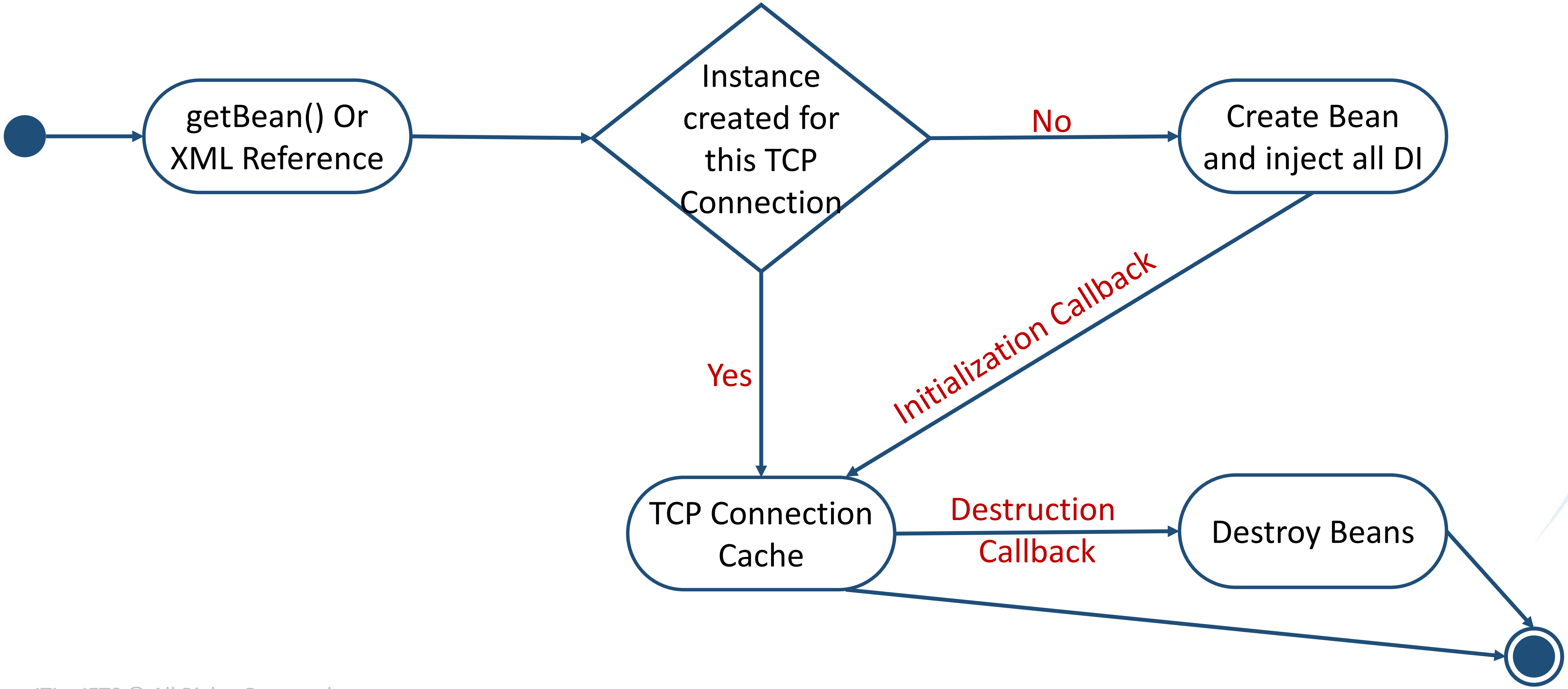
- The Spring container creates a new instance of the UserSession bean by using the userSession bean definition for the lifetime of a single TCP Connection **(Stateful per TCP Connection)**.
- When the TCP Connection is eventually discarded, the bean that is scoped to that particular websocket is also discarded.
- Consider the following XML configuration for a bean definition:



```
<bean id="userSession"
          class="com.jediver.spring.connection.UserSession"
          scope="websocket" />
```

Bean Scopes

6. **websocket (Lifecycle)**

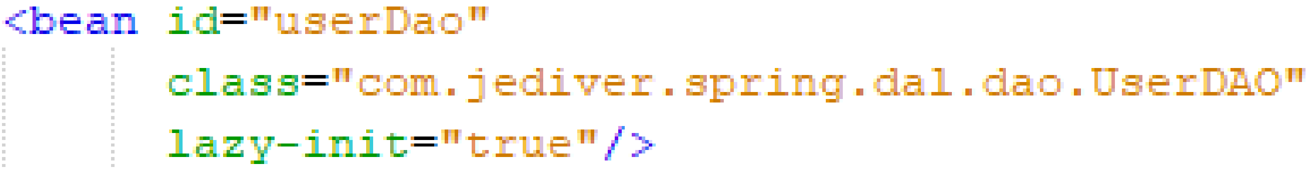

getBean() Or XML Reference
Instance created for this TCP Connection
No
Create Bean and inject all DI
Initialization Callback
Yes
TCP Connection Cache
Destruction Callback
Destroy Beans

Lazy-initialized Beans

- The default behavior for ApplicationContext
  - Implementations eagerly create and configure all singleton beans as part of the initialization process.
- When this Pre-instantiation is desirable:
  - Because errors in the configuration or surrounding environment are discovered immediately once the ApplicationContext created.
- When this Pre-instantiation is not desirable:
  - The time and memory needed to create these beans before they are actually needed.
- you can prevent pre-instantiation of a singleton bean by marking the bean definition as being lazy-initialized.

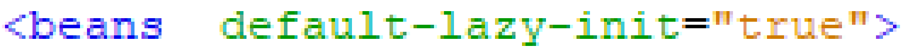
Lazy-initialized Beans (Ex.)

- A lazily-initialized bean indicates to the IoC container to create a bean instance when it is first requested, rather than at startup.
- This behavior is controlled by the lazy-init attribute on the <bean/> element, as example:



```
<bean id="userDao"
      class="com.jediver.spring.dal.dao.UserDAO"
      lazy-init="true"/>
```

- You can also control lazy-initialization at the container level by using the default-lazy-init attribute on the <beans/> element, a the following example shows:



```
<beans  default-lazy-init="true">
```

Lesson 8

Core Container

(Customizing the Nature of a Bean)

Lifecycle Callbacks

- To interact with the container's management of the bean lifecycle.
- To do this we have three ways:
  1. First use the interface
     - You can implement the Spring InitializingBean and DisposableBean interfaces.
  2. Use generic initialization and/or generic Destruction method
  3. Use Annotation
     - (@PostConstruct, @PreDestroy) annotation
- Internally, the Spring Framework uses BeanPostProcessor implementations to process any callback interfaces it can find and call the appropriate methods.

Initialization Callbacks

- We use Initialization callbacks to allow a bean to perform initialization work after all necessary properties on the bean have been set by the container.

- To do this we have three ways:

1. **First use the interface**
2. **Use generic initialization method**
3. **@PostConstruct annotation**

Initialization Callbacks

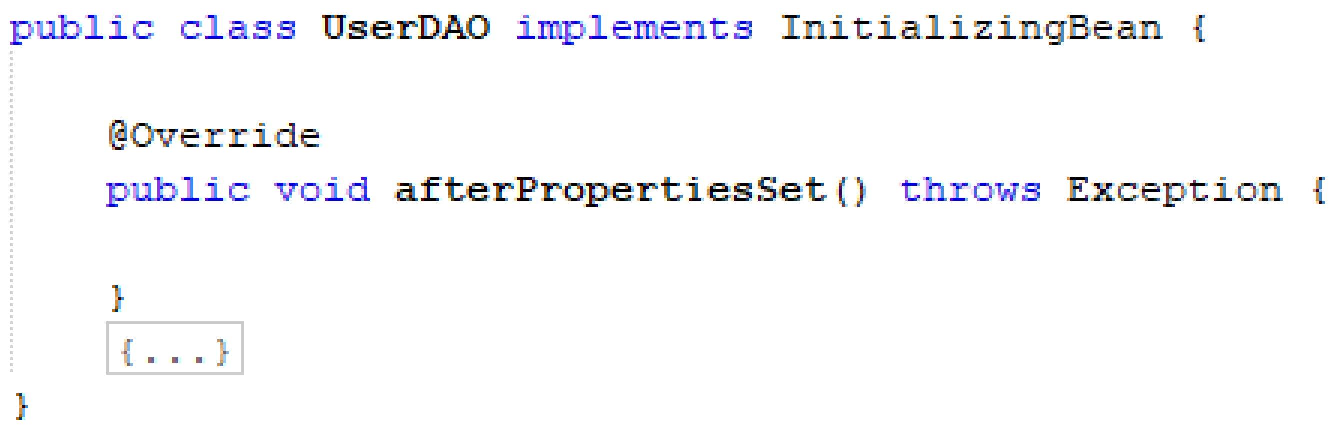
1. First use the interface

- The org.springframework.beans.factory.InitializingBean interface lets a bean perform initialization work after the container has set all necessary properties on the bean.
- The InitializingBean interface specifies a single method:
  - public void afterPropertiesSet() throws Exception;
- You make your bean implement InitializingBean interface that makes you implement afterPropertiesSet() and perform initialization work.

Initialization Callbacks

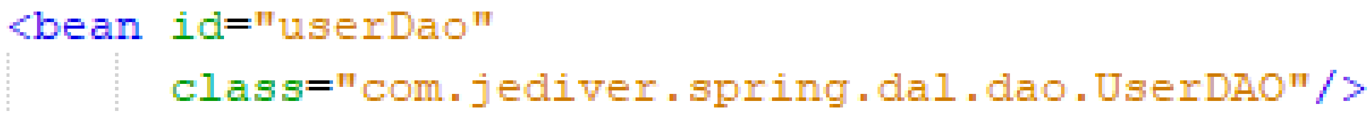
1. First use the interface

- Your Class:



```
public class UserDAO implements InitializingBean {

    @Override
    public void afterPropertiesSet() throws Exception {

    }
    {...}
}
```

- Bean Definition in XML:



```
<bean id="userDao"
      class="com.jediver.spring.dal.dao.UserDAO"/>
```

Initialization Callbacks

1. First use the interface

   - Not recommend to use the InitializingBean interface, because it unnecessarily couples the code to Spring.
   - Alternatively:
     - We suggest using the @PostConstruct annotation
     - Or specifying a POJO initialization method.

Initialization Callbacks

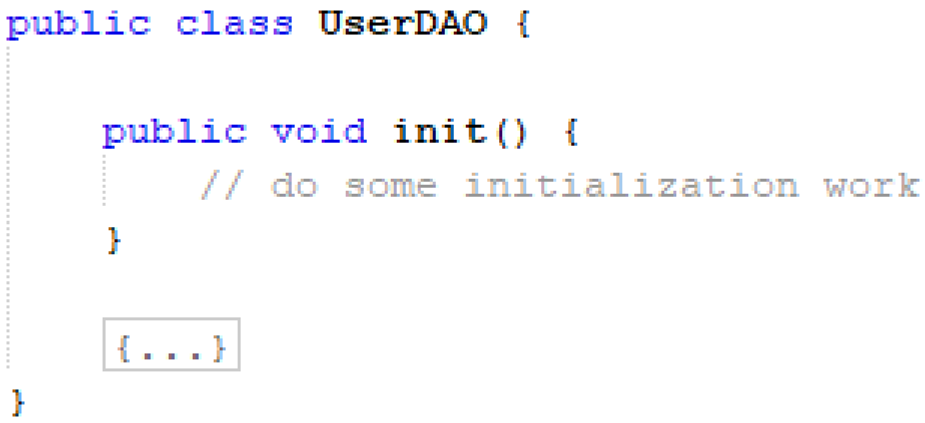
2. Use generic initialization method

- In the case of XML-based configuration metadata.
- You can use the init-method attribute to specify the name of the method that has a void no-argument signature.
- With Java configuration, you can use the initMethod attribute of @Bean.
- It's preferable to use @PostConstruct annotation instead of generic initialization method.

Initialization Callbacks

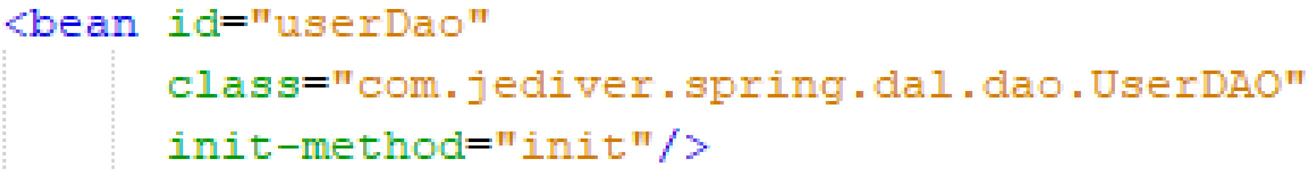
2. Use generic initialization method

- Your Class:



```
public class UserDAO {

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

    {...}
}
```

- Bean Definition in XML:



```
<bean id="userDao"
        class="com.jediver.spring.dal.dao.UserDAO"
        init-method="init"/>
```

Destruction callbacks

- We use destruction callbacks to allow a bean to perform destruction work before destroy the bean.

- To do this we have three ways:

1. **First use the interface**
2. **Use generic destruction method**
3. **@PreDestroy annotation**

Destruction callbacks

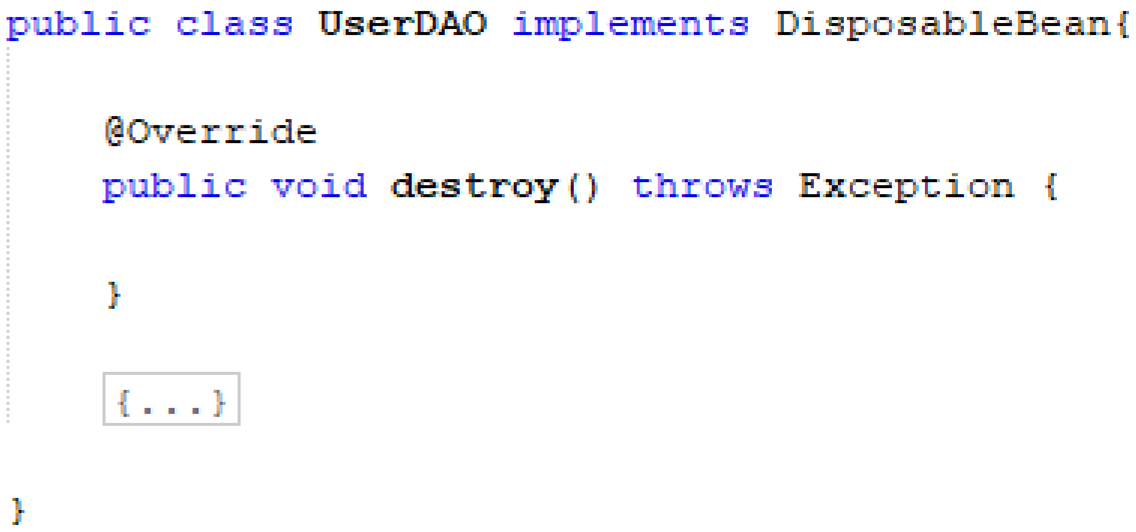
1. First use the interface

- The org.springframework.beans.factory.DisposableBean interface lets a bean perform destruction work before the container destroy the bean.
- The DisposableBean interface specifies a single method:
  - public void destroy() throws Exception;
- You make your bean implement DisposableBean interface that makes you implement destroy() and perform destruction work.

Destruction callbacks

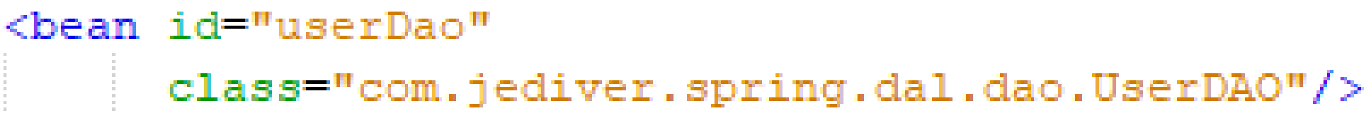
1. First use the interface

- Your Class:



```
public class UserDAO implements DisposableBean{

    @Override
    public void destroy() throws Exception {

    }

    {...}

}
```

- Bean Definition in XML:



```
<bean id="userDao"
      class="com.jediver.spring.dal.dao.UserDAO"/>
```

Destruction callbacks

1. First use the interface

    - Not recommend to use the DisposableBean interface, because it unnecessarily couples the code to Spring.
    - Alternatively:
        - We suggest using the @PreDestroy annotation
        - Or specifying a POJO destruction method.

Destruction callbacks

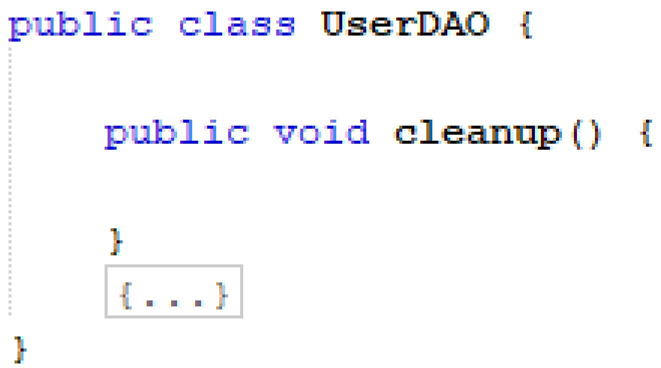
2. Use generic destruction method

- In the case of XML-based configuration metadata.
- You can use the destroy-method attribute to specify the name of the method that has a void no-argument signature.
- With Java configuration, you can use the destroyMethod attribute of @Bean.
- It’s preferable to use @PreDestroy annotation instead of generic destruction method.

Destruction callbacks

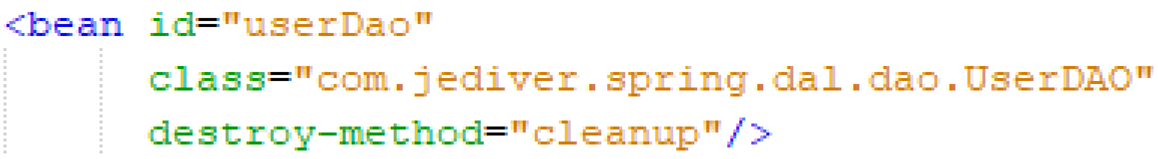
2. Use generic destruction method

- Your Class:



```
public class UserDAO {

    public void cleanup() {

    }
    {...}
}
```

- Bean Definition in XML:

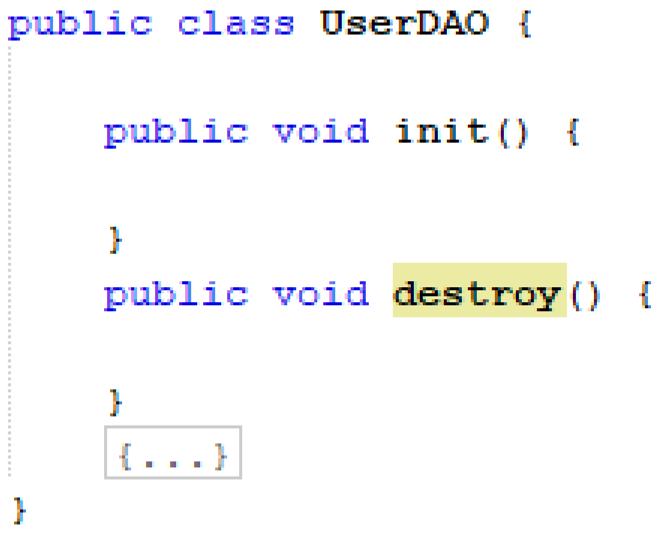


```
<bean id="userDao"
      class="com.jediver.spring.dal.dao.UserDAO"
      destroy-method="cleanup"/>
```

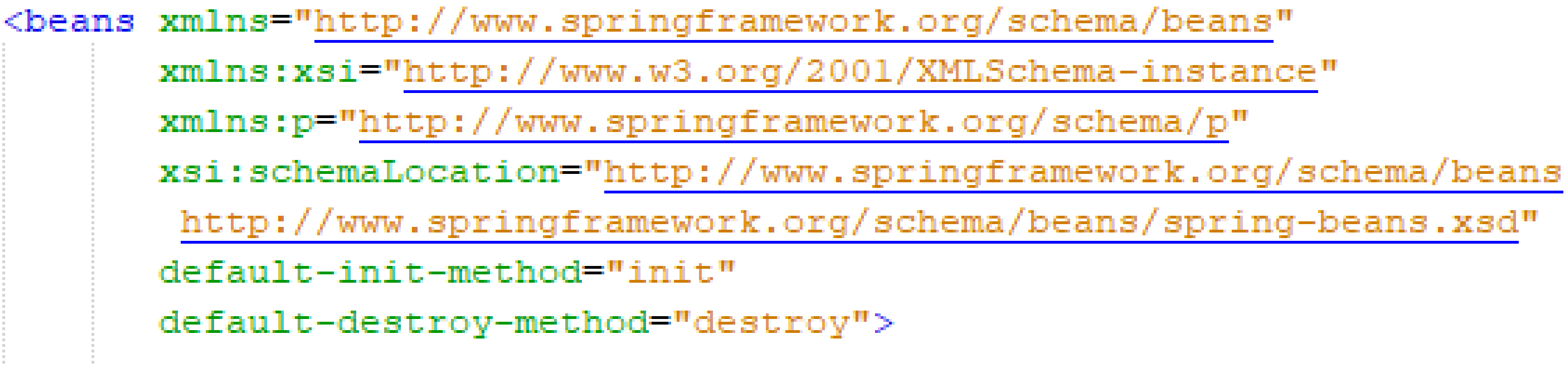
Default Initialization and Destroy Methods

- If you write methods with names such as init(), initialize(), dispose(), and so on.
- Ideally, the names of such lifecycle callback methods are standardized across a project so that all developers use the same method names and ensure consistency.
- You can configure the Spring container to “look” for named initialization and destroy callback method names on every bean.
  - This means that you, as an application developer, can write your application classes and use an initialization callback called init(), without having to configure an init-method="init" attribute with each bean definition.
  - This feature also enforces a consistent naming convention for initialization and destroy method callbacks.

Default Initialization and Destroy Methods (Ex.)



```java
public class UserDAO {

    public void init() {

    }
    public void destroy() {

    }
    {...}
}
```



```xml
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:p="http://www.springframework.org/schema/p"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd"
        default-init-method="init"
        default-destroy-method="destroy">
```

- you can override the default by specifying (in XML, that is) the method name by using the init-method and destroy-method attributes of the <bean/> itself.

Combining Lifecycle Mechanisms

- You can combine these three mechanisms to control a given bean.
- With different methods:
  - Initialization methods, are called as follows:
    1. Methods annotated with @PostConstruct
    2. afterPropertiesSet() as defined by the InitializingBean callback interface.
    3. A custom configured init() method
  - Destruction methods, are called as follows:
    1. Methods annotated with @PreDestroy
    2. destroy() as defined by the DisposableBean callback interface
    3. A custom configured destroy() method

Combining Lifecycle Mechanisms (Ex.)

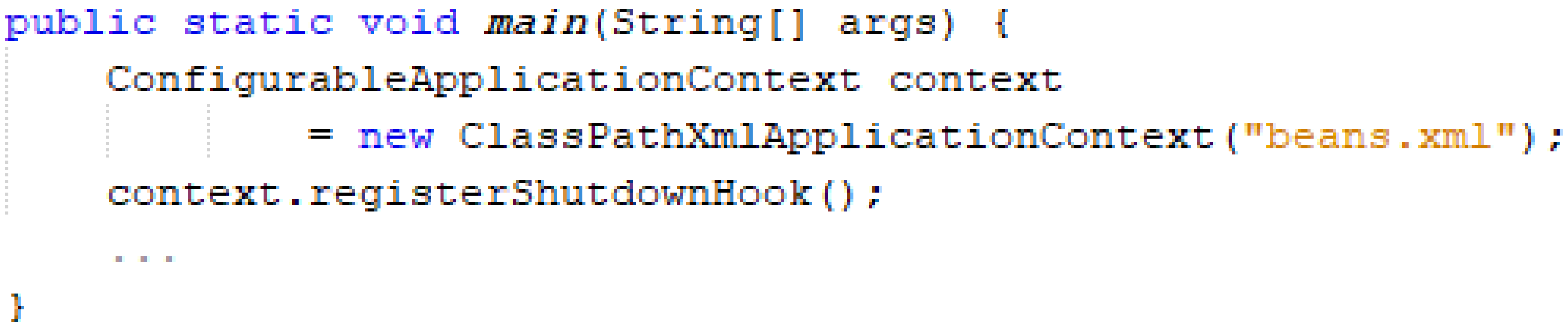
- With same methods:
    - However, if the same method name is configured for the different mechanism.
    - For example:
        - init() for an initialization method and also configured by @PostConstruct
        - that method is executed once.
    - Also Applied for the destruction.

Shutting Down the Spring IoC Container

- Shutting Down the Spring IoC Container Gracefully in Non-Web Applications
- Spring’s web-based ApplicationContext implementations already have code in place to gracefully shut down the Spring IoC container when the relevant web application is shut down.
- If you use Spring’s IoC container in a non-web application environment (ex. in a rich client desktop environment), register a shutdown hook with the JVM.
    - Doing so ensures a graceful shutdown and calls the relevant destroy methods on your singleton beans so that all resources are released. You must still configure and implement these destroy callbacks correctly.
- To register a shutdown hook, call the registerShutdownHook() method that is declared on the ConfigurableApplicationContext interface.

Shutting Down the Spring IoC Container (Ex.)

- Shutting Down the Spring IoC Container Gracefully in Non-Web Applications
- Spring’s web-based ApplicationContext implementations already have code in place to



```
public static void main(String[] args) {
    ConfigurableApplicationContext context
            = new ClassPathXmlApplicationContext("beans.xml");
    context.registerShutdownHook();
    ...
}
```