Lightweight J2EE Framework

Struts, spring, hibernate

Software System Design
Zhu Hongjun

Session 6: Spring IoC

- Spring Foundations
- Interface-oriented development
- Spring bean
- Dependency injection
- Spring AOP
- Data access with spring

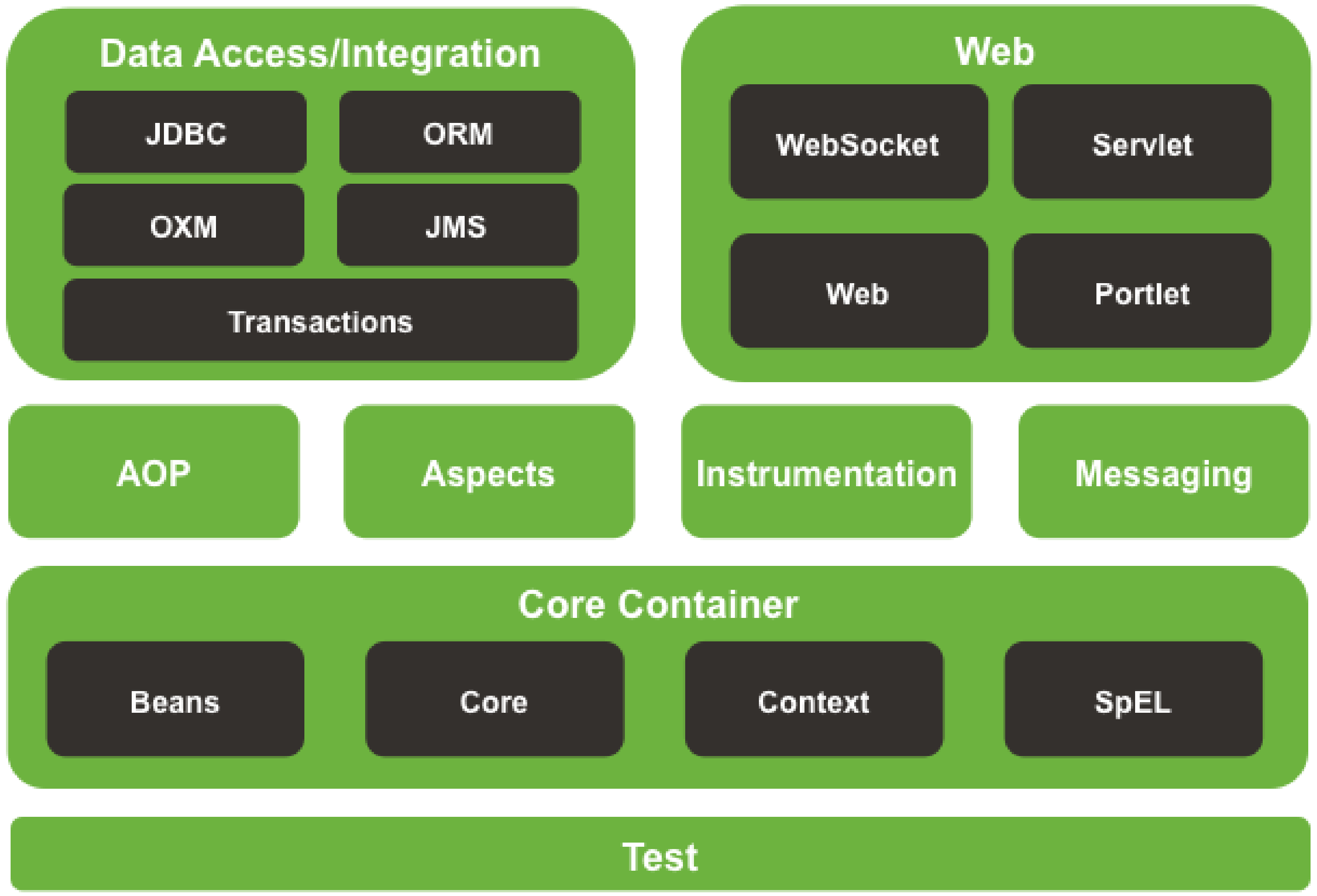
Spring Foundations

- The Spring Framework is a lightweight solution and a potential one-stop-shop for building your enterprise-ready applications
- Spring is modular. You can use the IoC container, with Struts on top, but you can also use only the Hibernate integration code or the JDBC abstraction layer

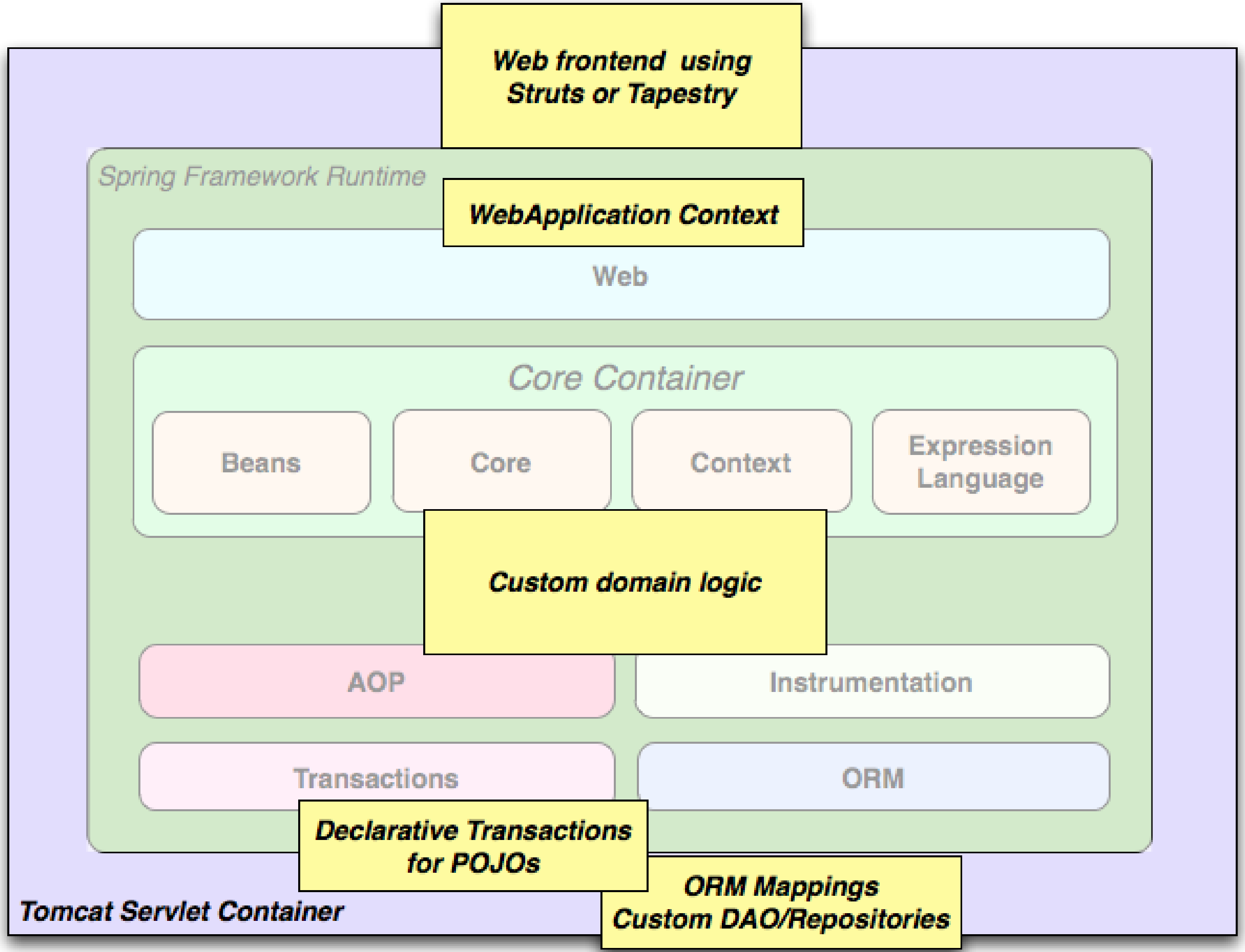
Spring Foundations

- Spring enables you to build applications from "plain old Java objects" (POJOs) and to apply enterprise services non-invasively to POJOs
  - This capability applies to the Java SE programming model and to full and partial Java EE
- The Spring Framework consists of features organized into about 20 modules

Overview of the spring framework



Spring middle-tier using a third-party web framework

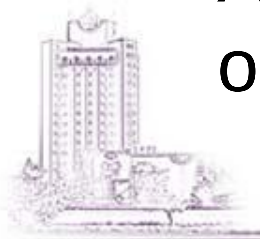


Spring Foundations

- IoC
  - Is a process whereby objects define their dependencies only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method
  - The container then *injects* those dependencies when it creates the bean

Spring Foundations

- IoC
  - The org.springframework.beans and org.springframework.context packages are the basis for Spring Framework's IoC container
  - The BeanFactory interface provides an advanced configuration mechanism capable of managing any type of object
  - ApplicationContext is a sub-interface of BeanFactory



Spring Foundations

- IoC
  - In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC *container* are called *beans*
  - A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container
  - Beans, and the *dependencies* among them, are reflected in the *configuration metadata* used by a container

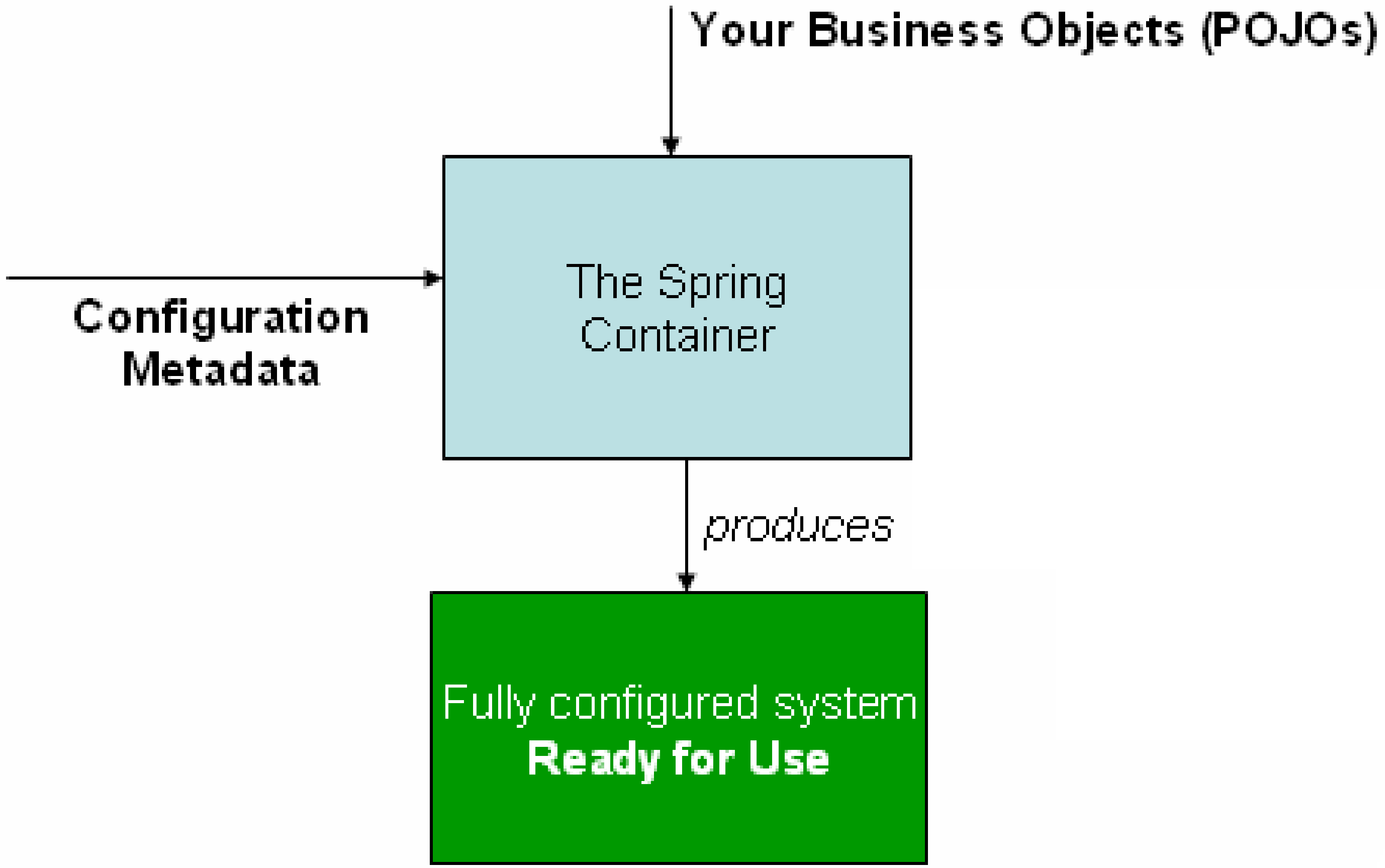
Spring Foundations

- DI
  - *Dependency injection, is aka IoC*
- AOP
  - *Aspect-Oriented Programming*
  - In AOP the key unit of modularity is the *aspect*
  - Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects

Spring Foundations

- IoC Container
  - The interface org.springframework.context. ApplicationContext represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the aforementioned beans
  - The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata

The Spring IoC Container



Spring Foundations

- IoC Container (cont.)
  - Configuration meta data
    - represents how you as an application developer tell the Spring container to instantiate, configure, and assemble the objects in your application
    - supplied in a simple and intuitive XML format
    - shows these beans configured as <bean/> elements inside a top-level<beans/> element

The Basic Structure of XML-based Configuration Metadata

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

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

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

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

</beans>
```

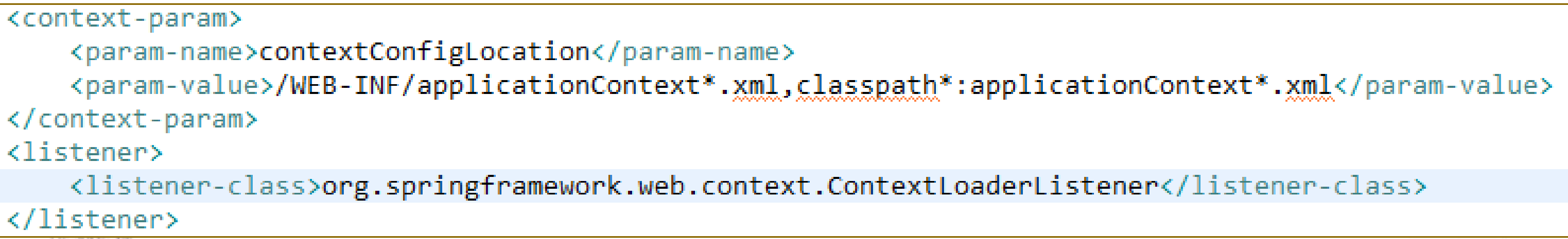
Spring Foundations

- IoC Container (cont.)
  - Instantiating a container
    - The location path or paths supplied to an ApplicationContext constructor are actually resource strings that allow the container to load configuration metadata from a variety of external resources such as the local file system, from the Java CLASSPATH, and so on

```
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});
```

Spring Foundations

- IoC Container (cont.)
  - Instantiating a container (cont.)
    - The ContextLoader mechanism comes in two flavors: the ContextLoaderListener and the ContextLoaderServlet
    - You should probably prefer ContextLoaderListener



```xml
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/applicationContext*.xml,classpath*:applicationContext*.xml</param-value>
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

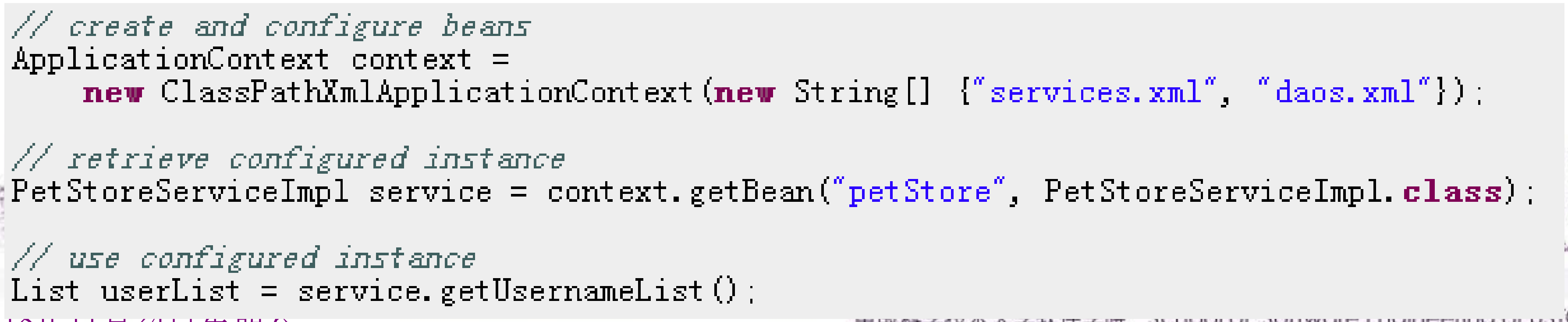
Spring Foundations

- IoC Container (cont.)
  - Composing XML-based configuration metadata
    - use one or more occurrences of the <import/> element to load bean definitions from another file or files

```
<beans>

    <import resource="services.xml"/>
    <import resource="resources/messageSource.xml"/>
    <import resource="/resources/themeSource.xml"/>

    <bean id="bean1" class="..."/>
    <bean id="bean2" class="..."/>

</beans>
```

Spring Foundations

- IoC Container (cont.)
  - Using the container
    - The ApplicationContext is the interface for an advanced factory capable of maintaining a registry of different beans and their dependencies
    - Using the method T getBean(String name, Class<T> requiredType) you can retrieve instances of your beans

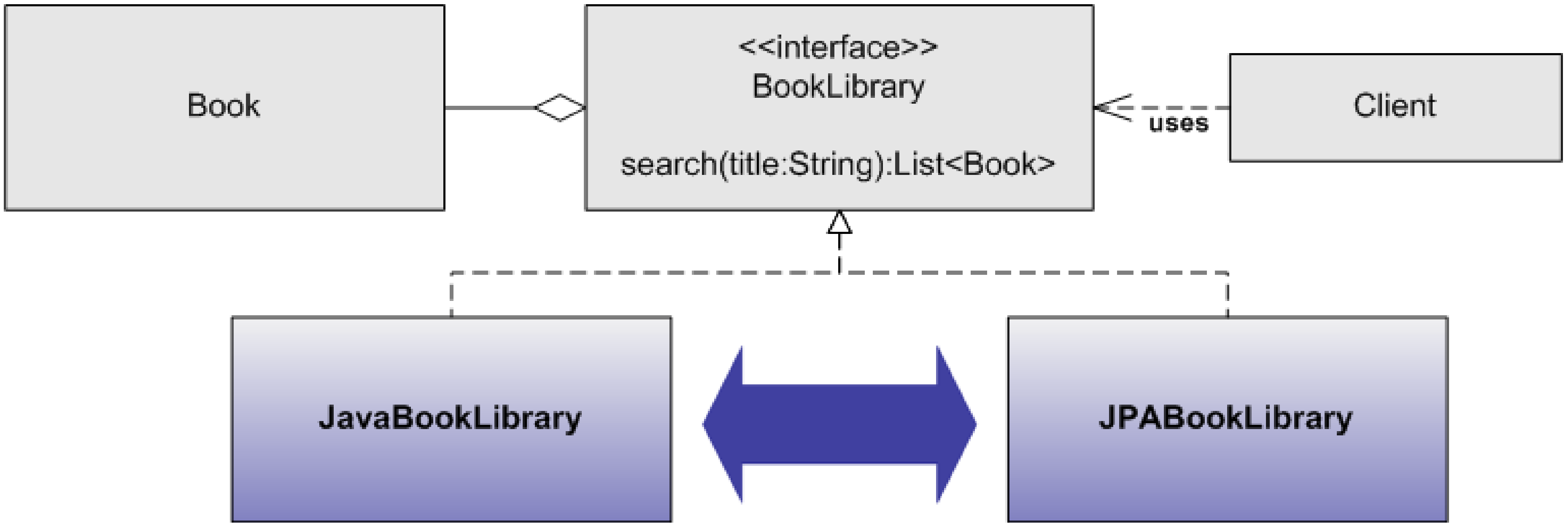


```
// create and configure beans
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});

// retrieve configured instance
PetStoreServiceImpl service = context.getBean("petStore", PetStoreServiceImpl.class);

// use configured instance
List userList = service.getUsernameList();
```

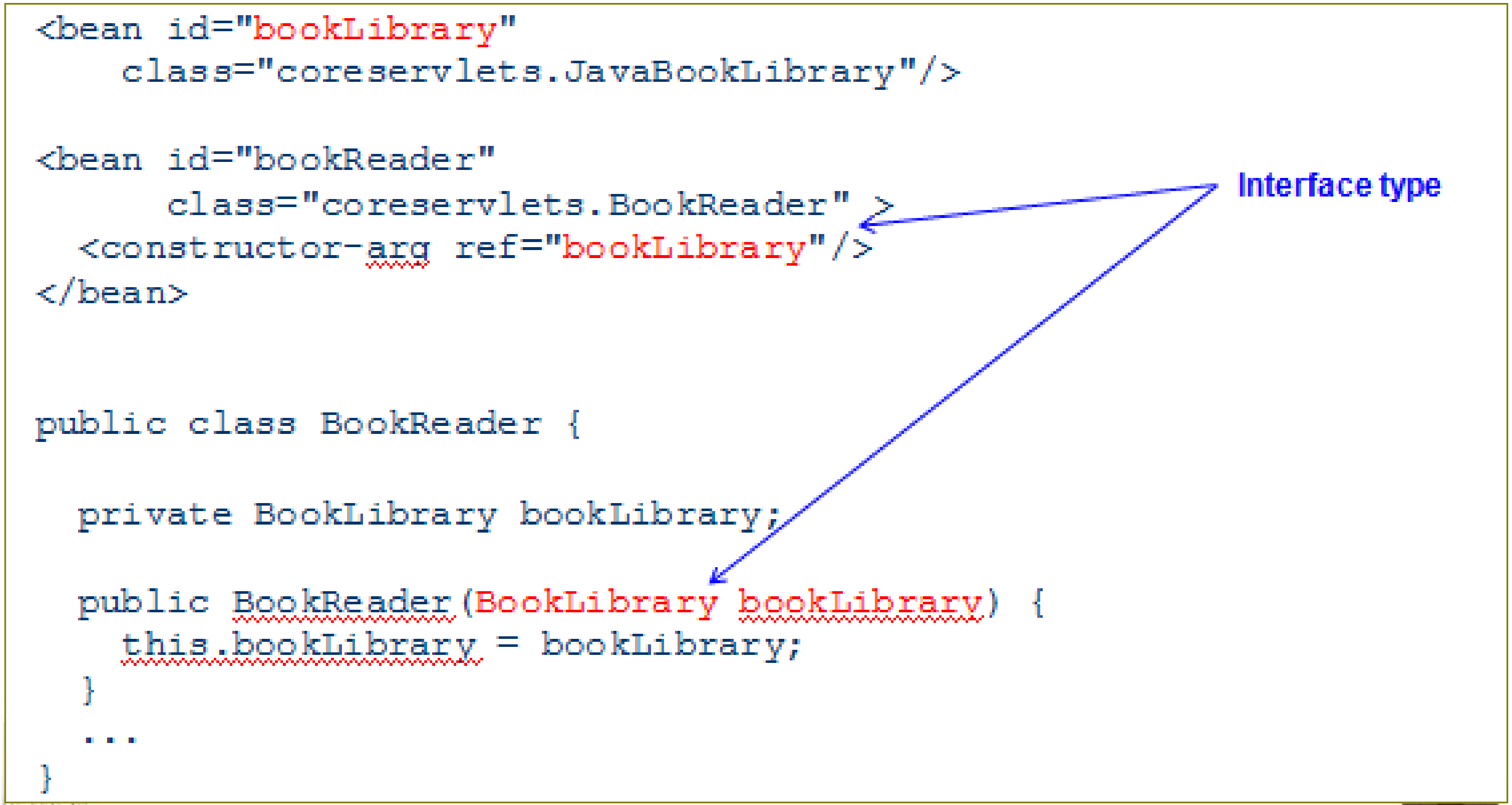
Interface-oriented development

- Take advantage of type-polymorphism
  - Flexible architecture
  - Tolerant to changes
  - Enables new capabilities with minimal effort
- Commit to interfaces, not implementations
  - Included, but not limited to, compiled interactions
  - Declarative interfaces

Scenario of Interface-oriented Development 1

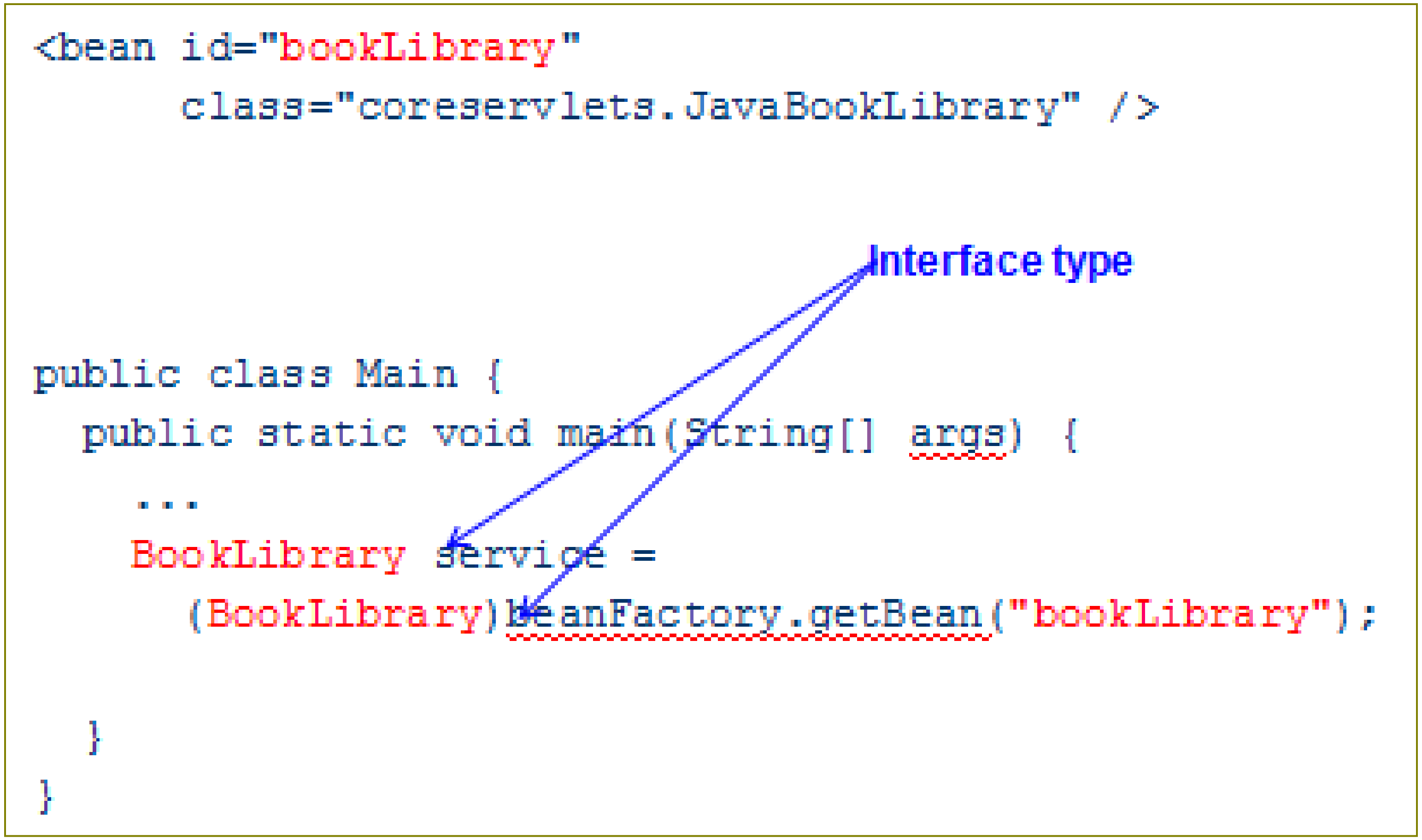


Scenario of Interface-oriented Development 2



```
<bean id="bookLibrary"
    class="coreservlets.JavaBookLibrary"/>

<bean id="bookReader"
      class="coreservlets.BookReader" >
  <constructor-arg ref="bookLibrary"/>
</bean>


public class BookReader {

  private BookLibrary bookLibrary;

  public BookReader(BookLibrary bookLibrary) {
    this.bookLibrary = bookLibrary;
  }
  ...
}
```

Scenario of Interface-oriented Development 3


```
<bean id="bookLibrary"
      class="coreservlets.JavaBookLibrary" />



public class Main {
  public static void main(String[] args) {
    ...
    BookLibrary service =
      (BookLibrary)beanFactory.getBean("bookLibrary");

  }
}
```

Spring bean

- A Spring IoC container manages one or more *beans*
- These beans definitions contain (among other information) the following metadata
  - *A package-qualified class name*
  - Bean behavioral configuration elements, which state how the bean should behave in the container
  - References to other beans that are needed for the bean to do its work
  - Other configuration settings to set in the newly created object

Spring bean

- Naming bean
  - Every bean has one or more identifiers. These identifiers must be unique within the container that hosts the bean
  - Use the id and/or name attributes to specify the bean identifier(s)
  - You can Alias a bean outside the bean definition

```
<alias name="fromName" alias="toName"/>
```

Spring bean

- Instantiating bean
  - A bean definition essentially is a recipe for creating one or more objects.
  - The container looks at the recipe for a named bean when asked, and uses the configuration metadata encapsulated by that bean definition to create (or acquire) an actual object

Spring bean

- Instantiating bean (cont.)
  - If you use XML-based configuration metadata, you specify the type (or class) of object that is to be instantiated in the class attribute of the <bean/> element
  - You use the Class property in one of two ways
    - Instantiation with a constructor
    - Instantiation with a factory method

Instantiating bean

Instantiation with constructor

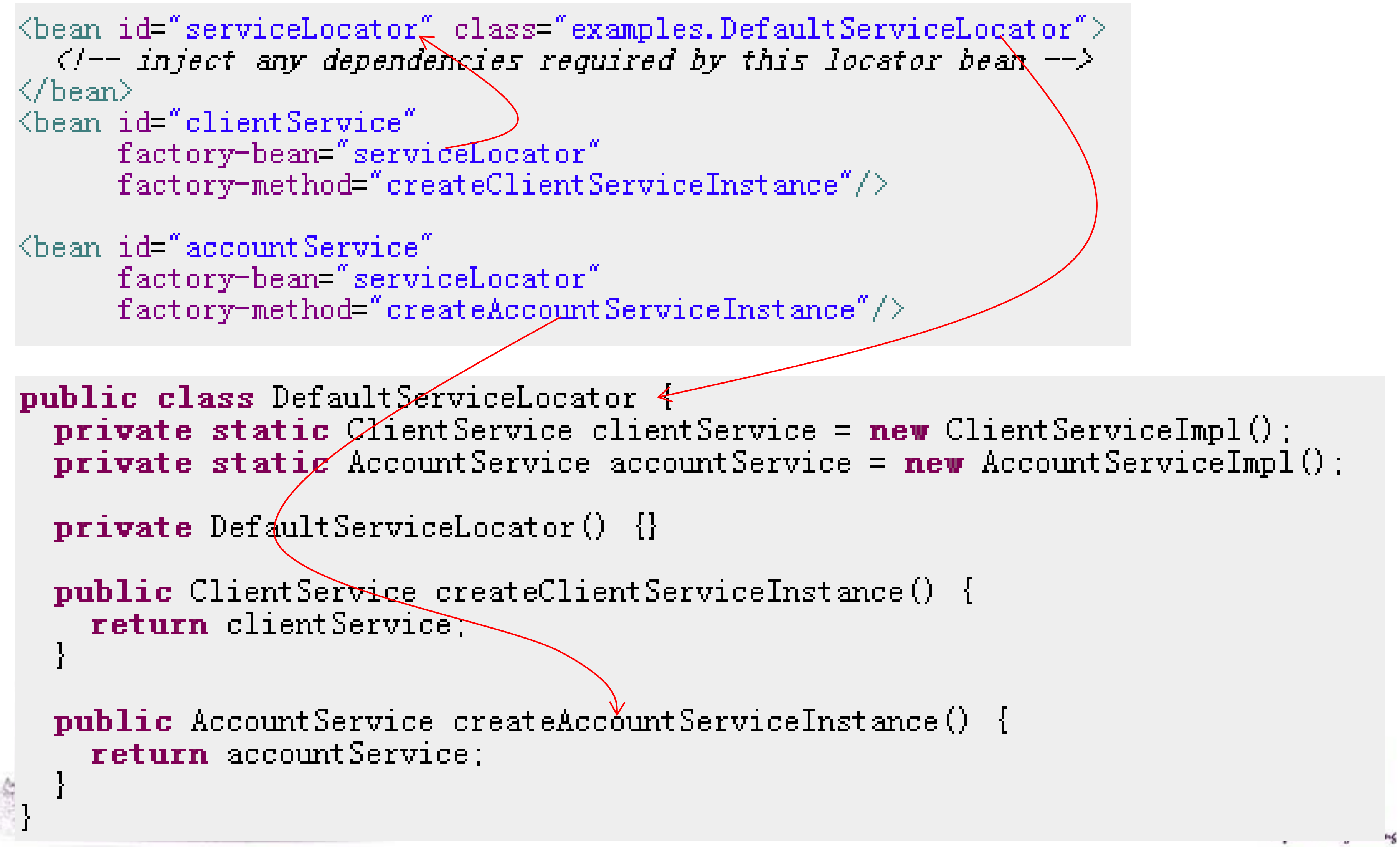
```
<bean id="exampleBean" class="examples.ExampleBean"/>

<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

Instantiation with static factory method

```
<bean id="clientService"
      class="examples.ClientService"
      factory-method="createInstance"/>
```

```
public class ClientService {
  private static ClientService clientService = new ClientService();
  private ClientService() {}

  public static ClientService createInstance() {
    return clientService;
  }
}
```

Instantiating bean with factory method



```
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
  <!-- inject any dependencies required by this locator bean -->
</bean>
<bean id="clientService"
      factory-bean="serviceLocator"
      factory-method="createClientServiceInstance"/>

<bean id="accountService"
      factory-bean="serviceLocator"
      factory-method="createAccountServiceInstance"/>
```

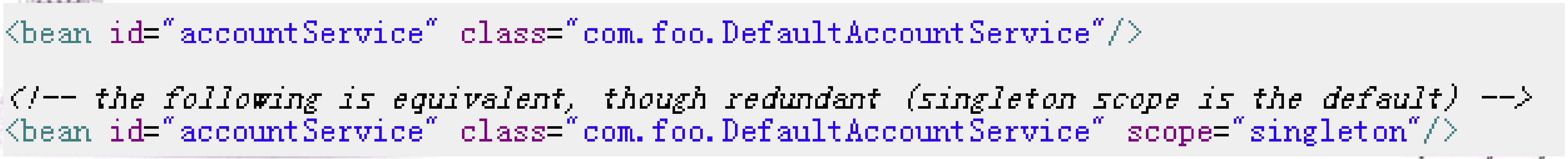
```
public class DefaultServiceLocator {
  private static ClientService clientService = new ClientServiceImpl();
  private static AccountService accountService = new AccountServiceImpl();

  private DefaultServiceLocator() {}

  public ClientService createClientServiceInstance() {
    return clientService;
  }

  public AccountService createAccountServiceInstance() {
    return accountService;
  }
}
```

Spring bean

- Bean scopes
  - You can control the *scope* of the objects created from a particular bean definition
    - Singleton scope
    - Prototype scope
    - Request scope
    - Session scope
    - Global session scope（valid in portlet context）
    - Application scope

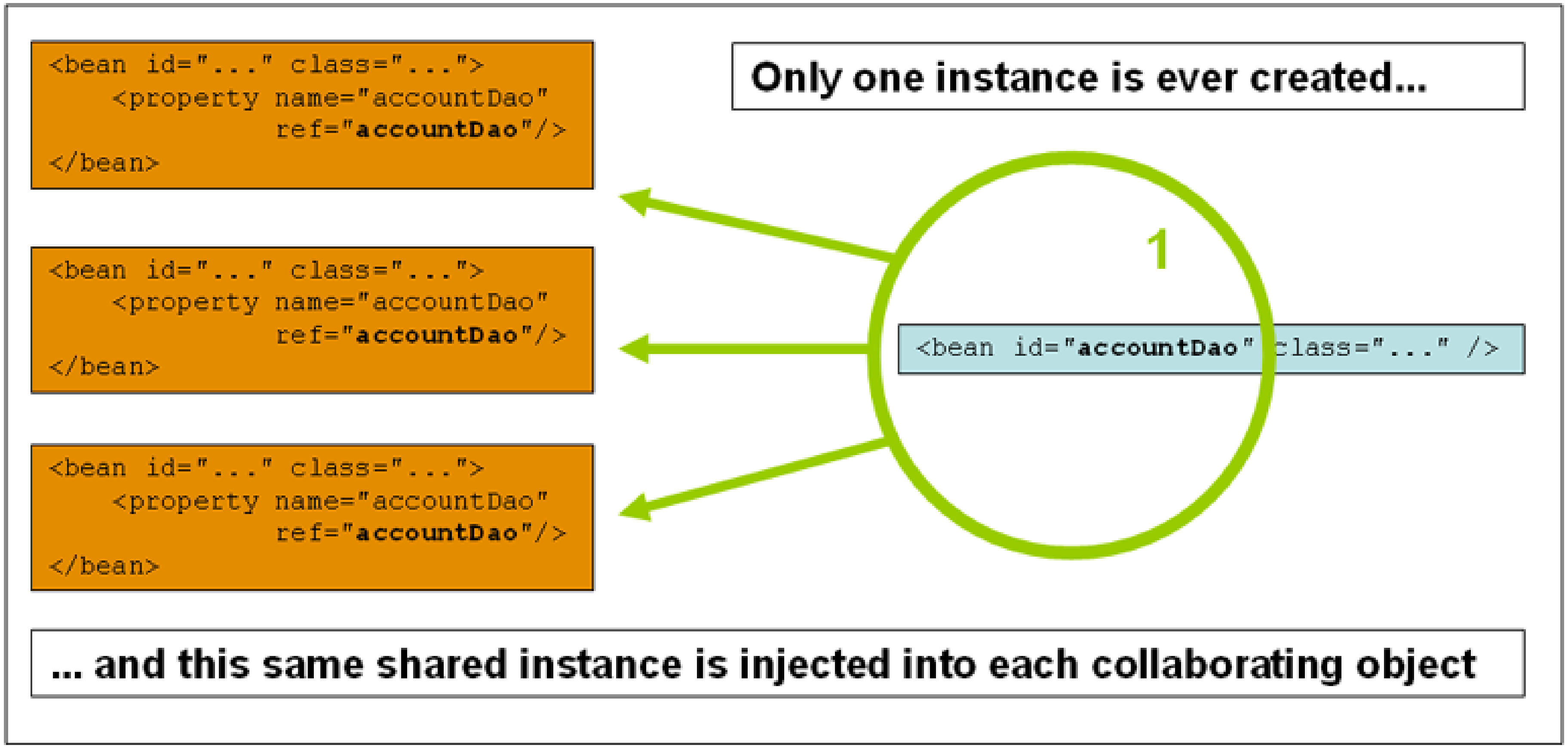
Spring bean

- Bean scopes (cont.)
  - The singleton scope
    - Only one *shared* instance of a singleton bean is managed
    - And all requests for beans with an id or ids matching that bean definition result in that one specific bean instance being returned by the Spring container
    - *The singleton scope is the default scope in Spring*



```
<bean id="accountService" class="com.foo.DefaultAccountService"/>

<!-- the following is equivalent, though redundant (singleton scope is the default) -->
<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>
```

The singleton scope


Only one instance is ever 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="..." />
```

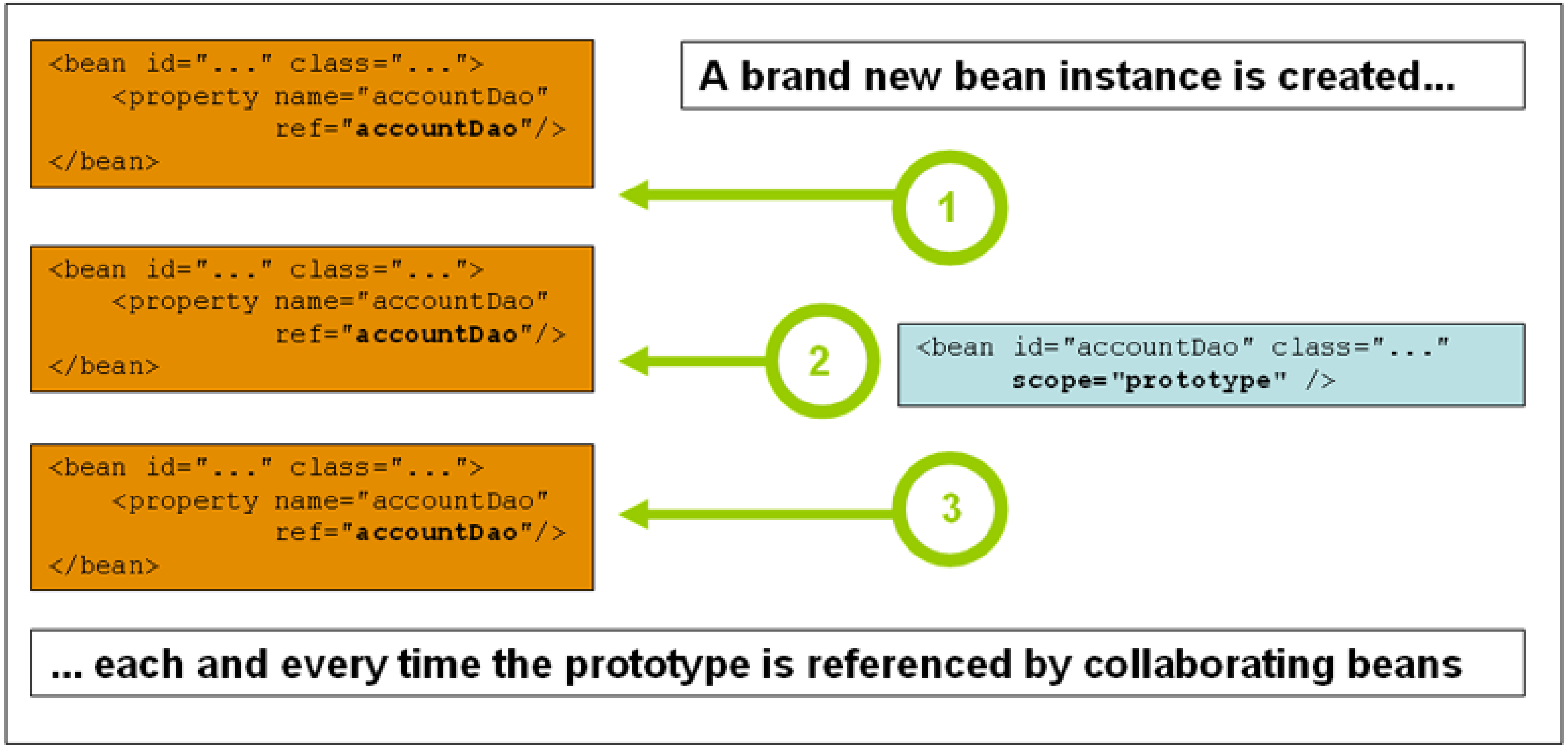
... and this same shared instance is injected into each collaborating object

Spring bean

- Bean scopes (cont.)
  - The prototype scope
    - The non-singleton, prototype scope of bean deployment results in the *creation of a new bean instance* every time a request for that specific bean is made
    - As a rule, use the prototype scope for all stateful beans and the singleton scope for stateless beans

```
<!-- using spring-beans-2.0.dtd -->
<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>
```

The prototype scope



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

Spring bean

- Bean scopes (cont.)
  - Request, session, global session and application scopes
    - Those scopes are *only* available if you use a web-aware Spring ApplicationContext implementation
    - To support the scoping of beans at the request, session, global session and application levels (web-scoped beans), some minor initial configuration is required before you define your beans
      - Initialize web configuration

Initialize web configuration

Servlet 2.4+ web container

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

Servlet 2.3 web container

```
<web-app>
..
<filter>
  <filter-name>requestContextFilter</filter-name>
  <filter-class>org.springframework.web.filter.RequestContextFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>requestContextFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
...
</web-app>
```

Spring bean

- Bean scopes (cont.)
  - The request scope
    - The Spring container creates a new instance of the bean by using the bean definition for each and every HTTP request
    - When the request completes processing, the bean that is scoped to the request is discarded

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

Spring bean

- Bean scopes (cont.)
  - The session scope
    - The Spring container creates a new instance of the bean by using the bean definition for the lifetime of a single HTTP Session
    - When the HTTP Session is eventually discarded, the bean that is scoped to that particular HTTP Session is also discarded

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

Spring bean

- Bean scopes (cont.)
  - The global session scope
    - The global session scope is similar to the standard HTTP Session scope, and applies only in the context of beans defined at the global session scope are scoped (or bound) to the lifetime of the global portlet Session

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

Spring bean

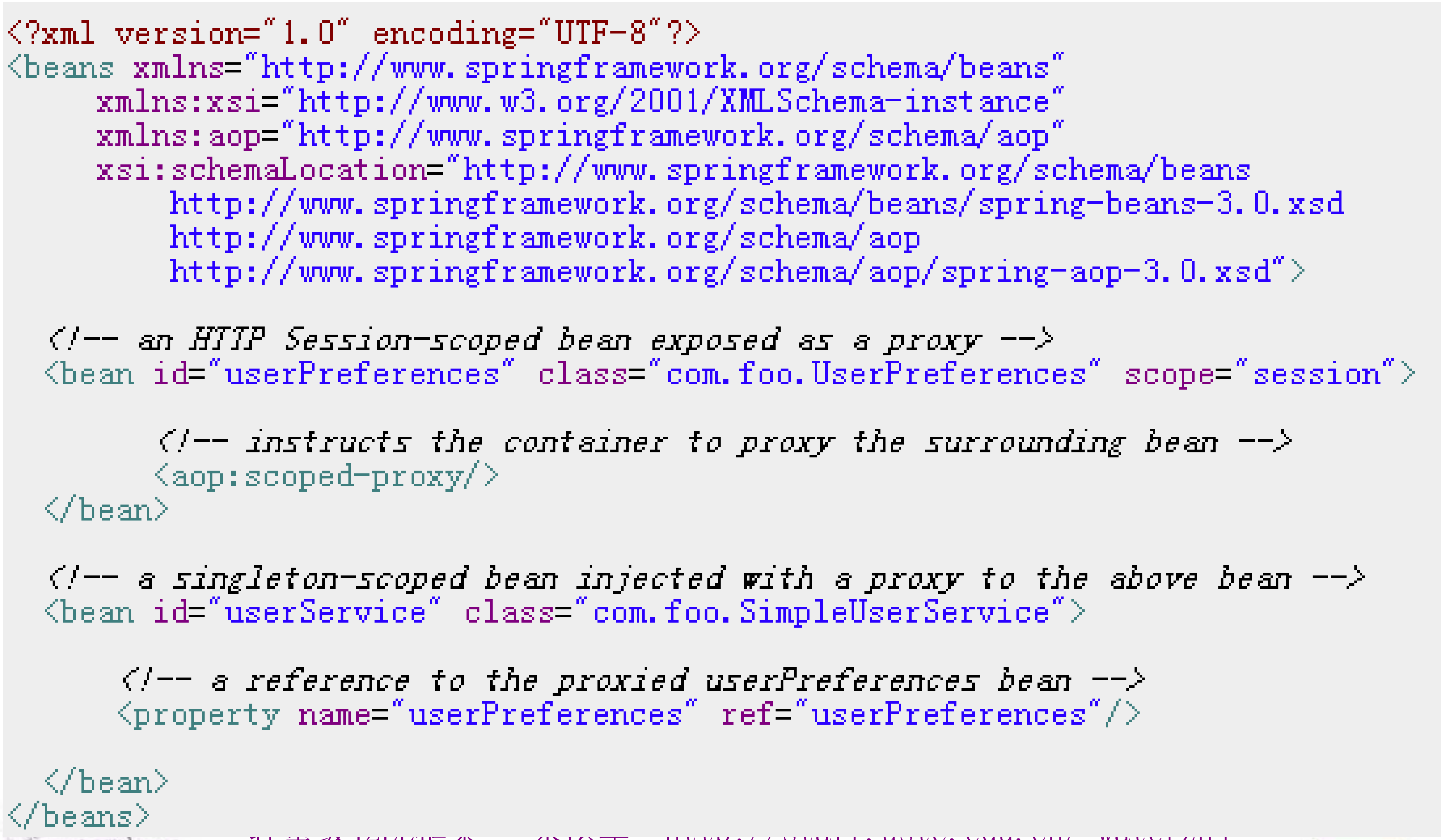
- Bean scopes (cont.)
  - The application scope
    - scopes a single bean definition to the lifecycle of a ServletContext
    - similar to a Spring singleton bean but differs in two important ways:
      - It is a singleton per ServletContext, not per Spring 'ApplicationContext'
      - It is actually exposed and therefore visible as a ServletContext attribute

```
<bean id="appPreferences" class="com.foo.AppPreferences" scope="application"/>
```

Spring bean

- Bean scopes (cont.)
  - Scoped bean as dependencies
    - The Spring IoC container manages not only the instantiation of your objects (beans), but also the wiring up of collaborators (or dependencies)
    - If you want to inject (for example) an HTTP request scoped bean into another singleton bean, you must inject an AOP proxy in place of the scoped bean

Scoped bean as dependencies



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

  <!-- an HTTP Session-scoped bean exposed as a proxy -->
  <bean id="userPreferences" class="com.foo.UserPreferences" scope="session">

        <!-- instructs the container to proxy the surrounding bean -->
        <aop:scoped-proxy/>
  </bean>

  <!-- a singleton-scoped bean injected with a proxy to the above bean -->
  <bean id="userService" class="com.foo.SimpleUserService">

      <!-- a reference to the proxied userPreferences bean -->
      <property name="userPreferences" ref="userPreferences"/>

  </bean>
</beans>
```

Spring bean

- Customizing the nature of a bean
  - Lifecycle callback
    - To interact with the container's management of the bean lifecycle, you can implement the Spring InitializingBean and DisposableBean interfaces
    - You can also achieve the same integration with the container without coupling your classes to Spring interfaces through the use of init-method and destroy method object definition metadata

Customizing lifecycle callbacks of bean

```
public class ExampleBean {

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

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

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

```
public class ExampleBean {

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

Spring bean

- Bean definition inheritance
  - A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed
  - When you use XML-based configuration metadata, you indicate a child bean definition by using the parent attribute

Bean definition inheritance

```
<bean id="inheritedTestBean" abstract="true"
    class="org.springframework.beans.TestBean">
  <property name="name" value="parent"/>
  <property name="age" value="1"/>
</bean>

<bean id="inheritsWithDifferentClass"
      class="org.springframework.beans.DerivedTestBean"
      parent="inheritedTestBean" init-method="initialize">

  <property name="name" value="override"/>
  <!-- the age property value of 1 will be inherited from  parent -->

</bean>
```

Dependency Injection

- DI
  - Code is cleaner with the DI principle and decoupling is more effective when objects are provided with their dependencies
    - The object does not look up its dependencies, and does not know the location or class of the dependencies
    - Your classes become easier to test, in particular when the dependencies are on interfaces or abstract base classes

Dependency Injection

- DI (cont.)
  - Dependency resolution process
    - The ApplicationContext is created and initialized with configuration metadata that describes all the beans
    - For each bean, its dependencies are expressed in the form of properties, constructor arguments, or arguments to the static-factory method if you are using that instead of a normal constructor

Dependency Injection

- DI (cont.)
  - Dependency resolution process (cont.)
    - Each property or constructor argument is an actual definition of the value to set, or a reference to another bean in the container
    - Each property or constructor argument which is a value is converted from its specified format to the actual type of that property or constructor argument
  - Two types DI(constructor-based and setter-based DI)

Dependency Injection

- Constructor-based DI
  - *Constructor-based* DI is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency
    - Reference beans already exist
      - you do not need to specify the constructor argument indexes and/or types explicitly in the <constructor-arg/> element
    - Use simple types

Constructor-based DI by Referencing an bean already exists

```
package x.y;

public class Foo {

  public Foo(Bar bar, Baz baz) {
      // ...
  }
}
```

```
<beans>
  <bean id="foo" class="x.y.Foo">
      <constructor-arg ref="bar"/>
      <constructor-arg ref="baz"/>
  </bean>

  <bean id="bar" class="x.y.Bar"/>
  <bean id="baz" class="x.y.Baz"/>

</beans>
```

Constructor-based DI by referencing simple types

```java
package examples;

public class ExampleBean {

  // No. of years to the calculate the Ultimate Answer
  private int years;

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

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

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

Dependency Injection

- Setter-based DI
  - Is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean
  - The ApplicationContext supports constructor- and setter-based DI for the beans it manages

Setter-based DI

```java
public class ExampleBean {

  private AnotherBean beanOne;
  private YetAnotherBean beanTwo;
  private int i;

  public void setBeanOne(AnotherBean beanOne) {
      this.beanOne = beanOne;
  }

  public void setBeanTwo(YetAnotherBean beanTwo)
      this.beanTwo = beanTwo;
  }

  public void setIntegerProperty(int i) {
      this.i = i;
  }
}
```

```xml
<bean id="exampleBean" class="examples.ExampleBean">

<!-- setter injection using the nested <ref/> element -->
<property name="beanOne"><ref bean="anotherExampleBean"/></property>

<!-- setter injection using the neater 'ref' attribute -->
<property name="beanTwo" ref="yetAnotherBean"/>
<property name="integerProperty" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

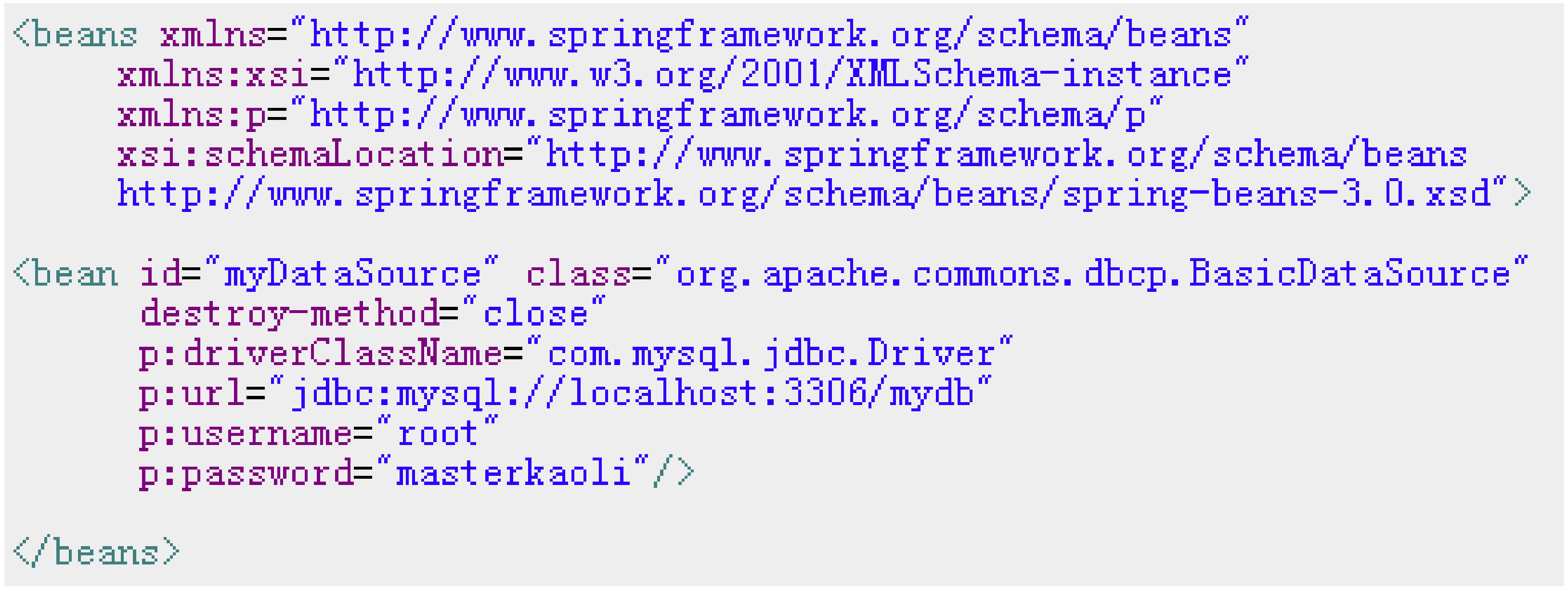
Dependency Injection

- Circular dependencies
  - When the Spring IoC container detects the circular reference at runtime, it will throw a BeanCurrentlyInCreationException
  - One possible solution is to edit the source code of some classes to be configured by setters DI rather than constructors DI

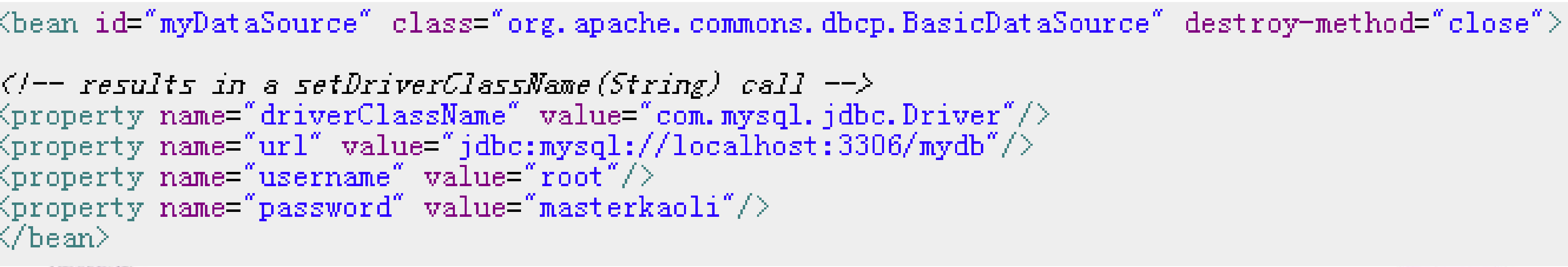
Dependency Injection

- Dependencies and configuration
  - You can define bean properties and constructor arguments as references to other managed beans (collaborators), or as values defined inline
  - Straight values
    - The value attribute of the <property/> element specifies a property or constructor argument as a human-readable string representation

Straight values Demo



```
<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-3.0.xsd">

<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"/>

</beans>
```



```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<!-- results in a setDriverClassName(String) call -->
<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>
```

Dependency Injection

- Dependencies and configuration (cont.)
  - The idref element
    - The idref element is simply an error-proof way to pass the *id* (string value - not a reference) of another bean in the container to a <constructor-arg/> or <property/> element
    - Using the idref tag allows the container to validate *at deployment time* that the referenced, named bean actually exists

The idref element in DI demo

```xml
<bean id="theTargetBean" class="..."/>

<bean id="theClientBean" class="...">
  <property name="targetName">
      <idref bean="theTargetBean" />
  </property>
</bean>
```

```xml
<property name="targetName">
 <!-- a bean with id 'theTargetBean' must exist; otherwise an exception will be thrown -->
 <idref local="theTargetBean"/>
</property>
```

Additionally, if the referenced bean is in the same XML unit, and the bean name is the bean *id*, you can use the local attribute, which allows the XML parser itself to validate the bean id earlier, at XML document parse time

Dependency Injection

- Dependencies and configuration (cont.)
  - Reference to other beans (collaborator)
    - The ref element is the final element inside a <constructor-arg/> or <property/> definition element
    - Here 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 depend on whether you specify the id/name of the other object through the bean, local, or parent attributes

Reference to other beans demo

```
<ref bean="someBean"/>
```

```
<ref local="someBean"/>
```

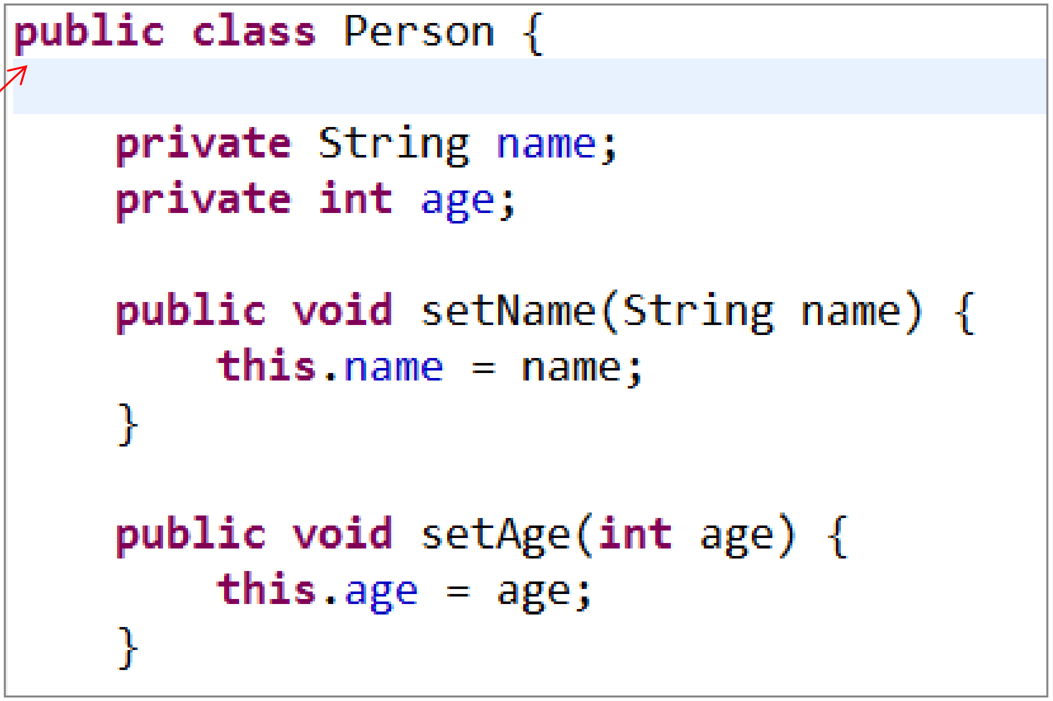
```
<!-- in the parent context -->
<bean id="accountService" class="com.foo.SimpleAccountService">
  <!-- insert dependencies as required as here -->
</bean>
```

```
<!-- in the child (descendant) context -->
<bean id="accountService"  <-- bean name is the same as the parent bean -->
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
        <ref parent="accountService"/>  <!-- notice how we refer to the parent bean -->
    </property>
  <!-- insert other configuration and dependencies as required here -->
</bean>
```

Dependency Injection

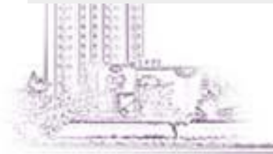
- Dependencies and configuration (cont.)
  - Inner bean
    - A <bean/> element inside the <property/> or <constructor-arg/> elements defines a so-called *inner bean*
    - An inner bean definition does not require a defined id or name; the container ignores these values
    - It also ignores the scope flag
      - Inner beans are *always* anonymous and they are *always* scoped as prototypes

Inner bean demo



```java
public class Person {

    private String name;
    private int age;

    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }
```

```xml
<bean id="outer" class="...">
<!-- instead of using a reference to a target bean, simply define the target bean inline -->
<property name="target">
  <bean class="com.example.Person"> <!-- this is the inner bean -->
    <property name="name" value="Fiona Apple"/>
    <property name="age" value="25"/>
  </bean>
</property>
</bean>
```

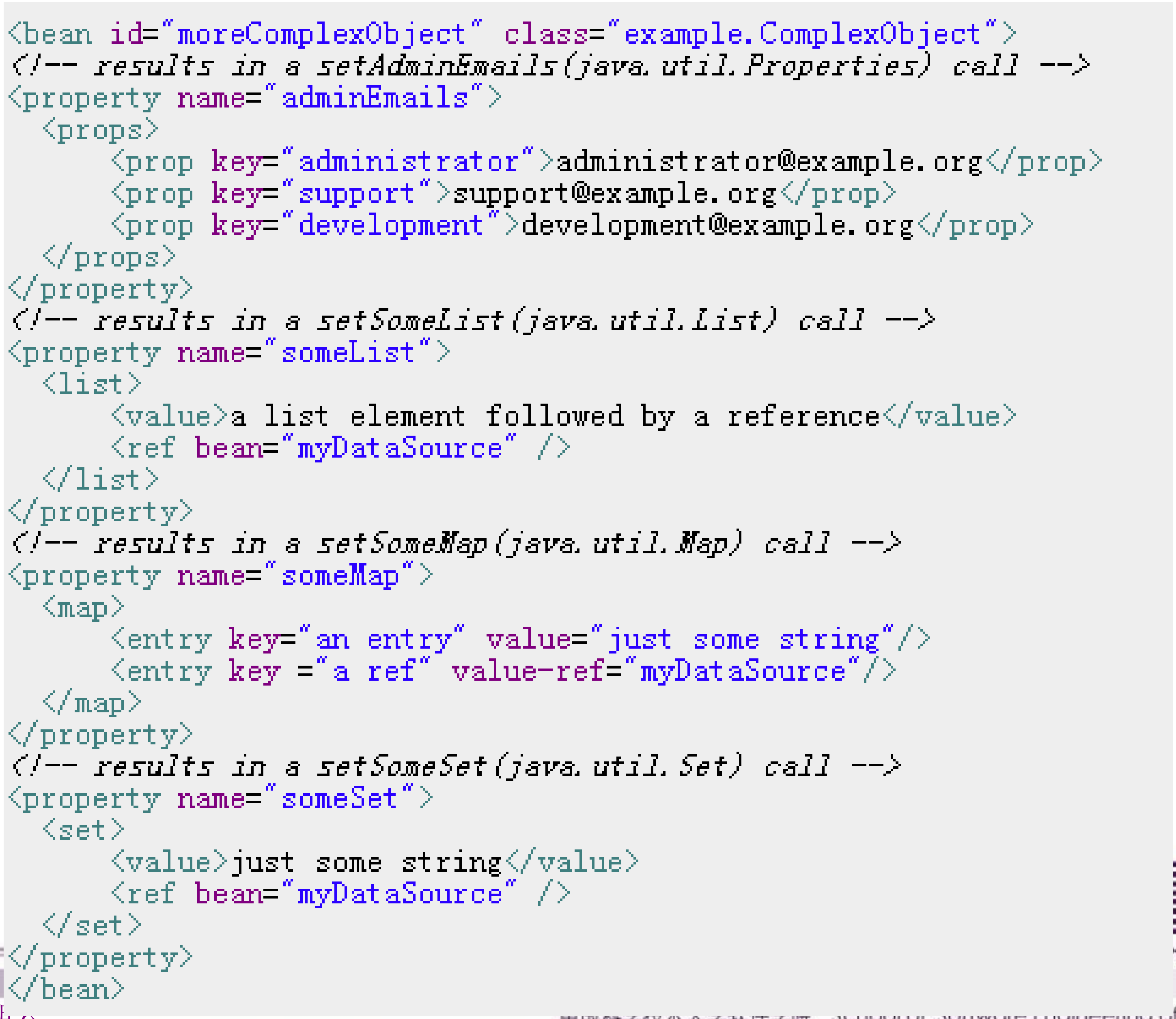


Dependency Injection

- Dependencies and configuration (cont.)
  - Collections
    - In the <list/>, <set/>, <map/>, and <props/> elements, you set the properties and arguments of the Java Collection types List, Set, Map, and Properties, respectively
    - *The value of a map key or value, or a set value, can also again be any of the following elements*

```
bean | ref | idref | list | set | map | props | value | null
```

Collections demo



```
<bean id="moreComplexObject" class="example.ComplexObject">
<!-- results in a setAdminEmails(java.util.Properties) call -->
<property name="adminEmails">
  <props>
      <prop key="administrator">administrator@example.org</prop>
      <prop key="support">support@example.org</prop>
      <prop key="development">development@example.org</prop>
  </props>
</property>
<!-- results in a setSomeList(java.util.List) call -->
<property name="someList">
  <list>
      <value>a list element followed by a reference</value>
      <ref bean="myDataSource" />
  </list>
</property>
<!-- results in a setSomeMap(java.util.Map) call -->
<property name="someMap">
  <map>
      <entry key="an entry" value="just some string"/>
      <entry key ="a ref" value-ref="myDataSource"/>
  </map>
</property>
<!-- results in a setSomeSet(java.util.Set) call -->
<property name="someSet">
  <set>
      <value>just some string</value>
      <ref bean="myDataSource" />
  </set>
</property>
</bean>
```

Dependency Injection

- Dependencies and configuration (cont.)
  - Null and empty string values
    - Spring treats empty arguments for properties and the like as empty Strings
    - The <null/> element handles null values

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

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

Dependency Injection

- Dependencies and configuration (cont.)
  - XML shortcut
    - With p-namespace
      - The p-namespace enables you to use the bean element's attributes, instead of nested <property/> elements, to describe your property values and/or collaborating beans
    - With c-namespace
      - the *c-namespace*, newly introduced in Spring 3.1, allows usage of inlined attributes for configuring the constructor arguments rather than nested constructor-arg elements

C-namespace and p-namespace demo

```
<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-3.0.xsd">

  <bean name="classic" class="com.example.ExampleBean">
      <property name="email" value="foo@bar.com"/>
  </bean>

  <bean name="p-namespace" class="com.example.ExampleBean"
        p:email="foo@bar.com"/>
</beans>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:c="http://www.springframework.org/schema/c"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="bar" class="x.y.Bar"/>
  <bean id="baz" class="x.y.Baz"/>

  <-- 'traditional' declaration -->
  <bean id="foo" class="x.y.Foo">
      <constructor-arg ref="bar"/>
      <constructor-arg ref="baz"/>
      <constructor-arg value="foo@bar.com"/>
  </bean>

  <-- 'c-namespace' declaration -->
  <bean id="foo" class="x.y.Foo" c:bar-ref="bar" c:baz-ref="baz" c:email="foo@bar.com">

</beans>
```

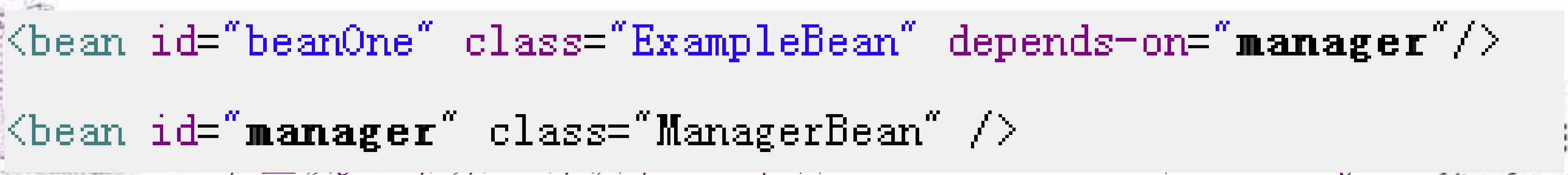
Dependency Injection

- Dependencies and configuration (cont.)
  - 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

```
<bean id="foo" class="foo.Bar">
<property name="fred.bob.sammy" value="123" />
</bean>
```

Dependency Injection

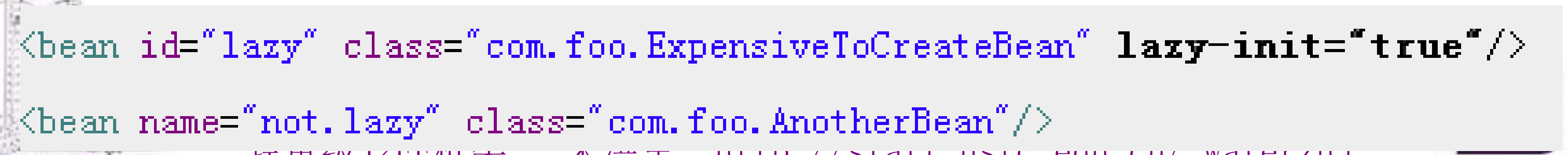
- Using dependents-on
  - The depends-on attribute can explicitly force one or more beans to be initialized before the bean using this element is initialized
  - Dependent beans that define a depends-on relationship with a given bean are destroyed first, prior to the given bean itself being destroyed



```
<bean id="beanOne" class="ExampleBean" depends-on="manager"/>

<bean id="manager" class="ManagerBean" />
```

Dependency Injection

- Lazy-initialized beans
  - By default, ApplicationContext implementations eagerly create and configure all singleton beans as part of the initialization process
  - you can prevent pre-instantiation of a singleton bean by marking the bean definition as lazy-initialized



```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>

<bean name="not.lazy" class="com.foo.AnotherBean"/>
```

Dependency Injection

- Autowiring collaborators
  - The Spring container can *autowire* relationships between collaborating beans
    - Autowiring can significantly reduce the need to specify properties or constructor arguments
    - Autowiring can update a configuration as your objects evolve
  - You can specify autowire mode for a bean definition with the autowire attribute of the <bean/> element

Dependency Injection

- Autowiring collaborators (cont.)
  - Autowiring modes
    - no (default)
      - No autowiring. Bean references must be defined via a ref element
    - byName
      - Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired
    - byType
    - constructor

Autowiring collaborators demo

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

<bean id="person" class="com.mkyong.common.Person" />
```

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

<bean id="person" class="com.mkyong.common.Person" />
```

```
package com.mkyong.common;

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

```
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;
        }
        //...
}
```

Dependency Injection

- Method Injection
  - When a singleton bean needs to collaborate with another singleton bean, or a non-singleton bean needs to collaborate with another non-singleton bean, you typically handle the dependency by defining one bean as a property of the other
  - The container only creates the singleton bean A once, and thus only gets one opportunity to set the properties

Dependency Injection

- Method Injection (cont.)
  - Method Injection, a somewhat advanced feature of the Spring IoC container, allows that problem to be handled
    - Lookup method injection
      - override methods on *container managed beans*, to return the lookup result for another named bean in the container
    - Arbitrary method replacement
      - replace arbitrary methods in a managed bean with another method implementation

Lookup method injection demo

```java
package fiona.apple;

// no more Spring imports!

public abstract class CommandManager {

 public Object process(Object commandState) {
    // grab a new instance of the appropriate Command interface
    Command command = createCommand();
    // set the state on the (hopefully brand new) Command instance
    command.setState(commandState);
    return command.execute();
 }

  // okay... but where is the implementation of this method?
 protected abstract Command createCommand();
}
```

```xml
<!-- a stateful bean deployed as a prototype (non-singleton) -->
<bean id="command" class="fiona.apple.AsyncCommand" scope="prototype">
<!-- inject dependencies here as required -->
</bean>

<!-- commandProcessor uses statefulCommandHelper -->
<bean id="commandManager" class="fiona.apple.CommandManager">
<lookup-method name="createCommand" bean="command"/>
</bean>
```

Arbitrary method replacement demo

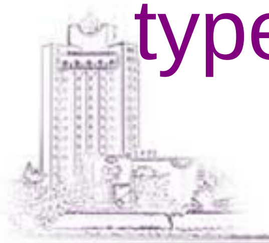
```java
public class MyValueCalculator {

public String computeValue(String input) {
  // some real code...
}

// some other methods...

}
```

```xml
<bean id="myValueCalculator" class="x.y.z.MyValueCalculator">

<!-- arbitrary method replacement -->
<replaced-method name="computeValue" replacer="replacementComputeValue">
  <arg-type>String</arg-type>
</replaced-method>
</bean>

<bean id="replacementComputeValue" class="a.b.c.ReplacementComputeValue"/>
```

```java
/** meant to be used to override the existing computeValue(String)
  implementation in MyValueCalculator
*/
public class ReplacementComputeValue implements MethodReplacer {

  public Object reimplement(Object o, Method m, Object[] args) throws Throwable {
      // get the input value, work with it, and return a computed result
      String input = (String) args[0];
      ...
      return ...;
  }
}
```

Spring AOP

- *Aspect-Oriented Programming* (AOP) complements Object-Oriented Programming (OOP) by providing another way of thinking about program structure
- Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects



Data access with spring

- The Spring Framework provides a consistent abstraction for transaction management
- The Data Access Object (DAO) support in Spring
- The Spring Framework supports integration with Hibernate, Java Persistence API (JPA), etc.

Conclusions

- Spring Foundations
- Interface-oriented development
- Spring bean
- Dependency injection
- Spring AOP
- Data access with spring