LESSON 01

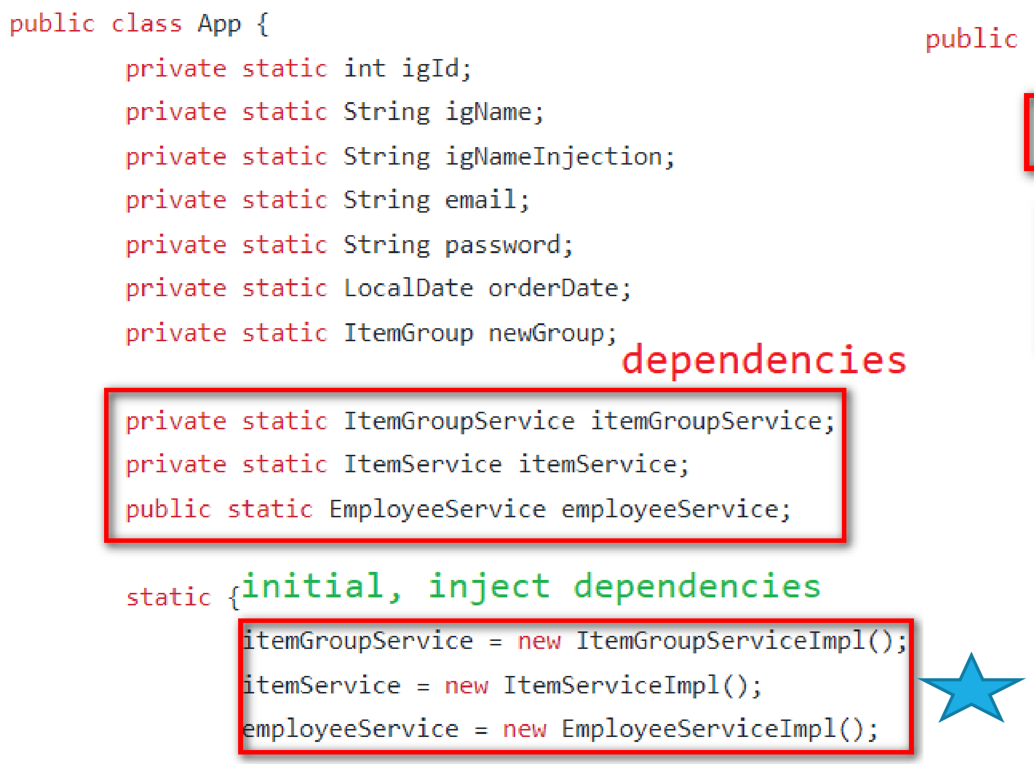
SPRING CORE/BEAN/CONTEXT

Agenda

- Understand **Inversion of control – Dependency injection** principle
- Understand definition of **dependency**
- Understand **Spring Core's IOC container**
- Understand the way Spring manage **objects** as a **beans** in core container with
  - XML
  - JAVA code, annotation
- Understand BEAN definition and life cycle

Inversion of control – dependency injection

Problem



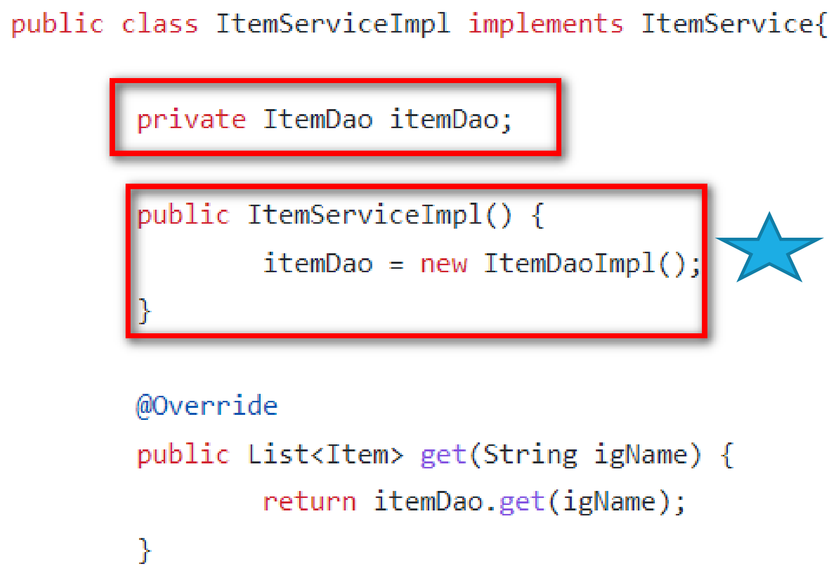
```
public class App {
        private static int igId;
        private static String igName;
        private static String igNameInjection;
        private static String email;
        private static String password;
        private static LocalDate orderDate;
        private static ItemGroup newGroup;
```

dependencies

```
        private static ItemGroupService itemGroupService;
        private static ItemService itemService;
        public static EmployeeService employeeService;
```

initial, inject dependencies

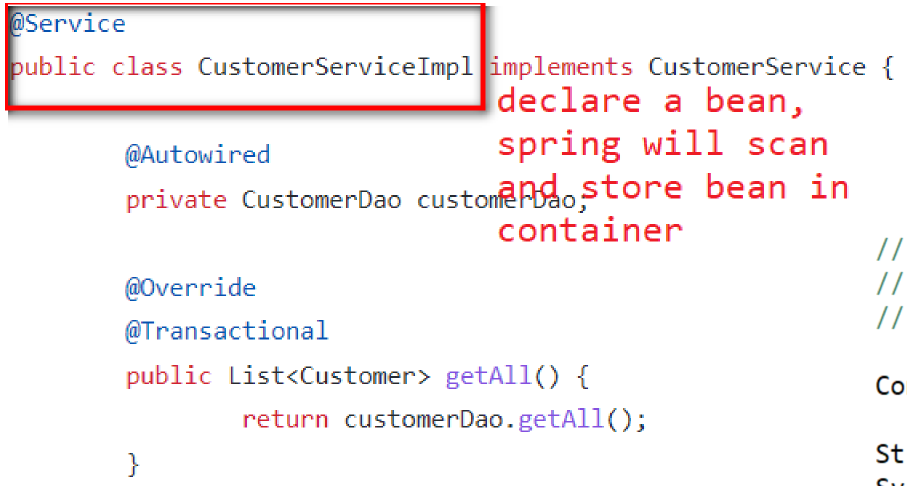
```
        static {
                itemGroupService = new ItemGroupServiceImpl();
                itemService = new ItemServiceImpl();
                employeeService = new EmployeeServiceImpl();
```



```
public class ItemServiceImpl implements ItemService{

        private ItemDao itemDao;

        public ItemServiceImpl() {
                itemDao = new ItemDaoImpl();
        }

        @Override
        public List<Item> get(String igName) {
                return itemDao.get(igName);
        }
```

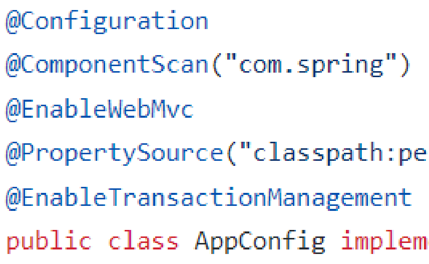
Inversion of control – dependency injection

➢ Solution



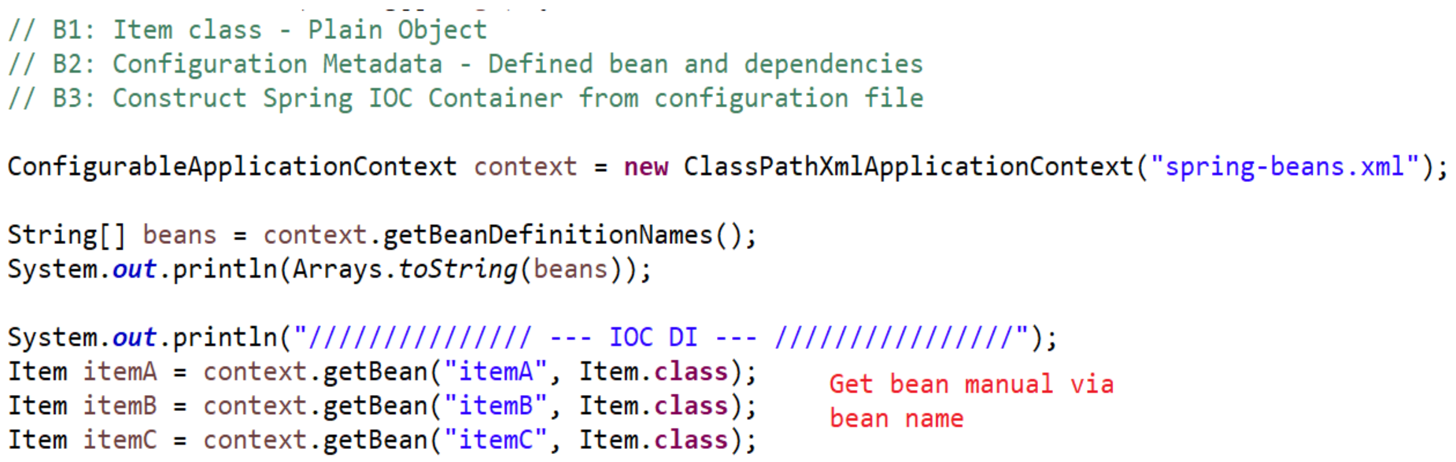
```java
@Service
public class CustomerServiceImpl implements CustomerService {

        @Autowired
        private CustomerDao customerDao;

        @Override
        @Transactional
        public List<Customer> getAll() {
                return customerDao.getAll();
        }
```

declare a bean, spring will scan and store bean in container



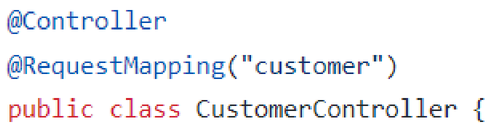
```java
@Configuration
@ComponentScan("com.spring")
@EnableWebMvc
@PropertySource("classpath:persistence-mysql.properties")
@EnableTransactionManagement
public class AppConfig implements WebMvcConfigurer {
```

Spring scan components, beans

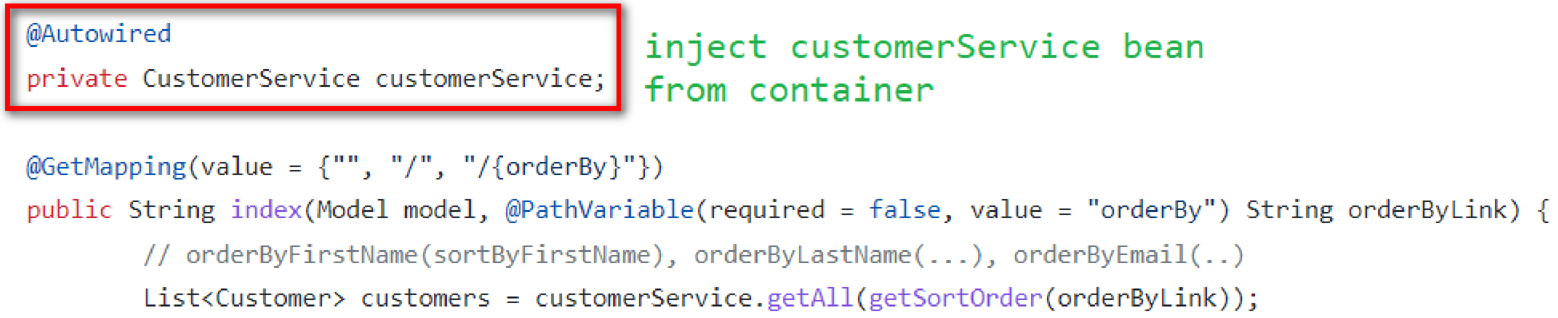


```java
// B1: Item class - Plain Object
// B2: Configuration Metadata - Defined bean and dependencies
// B3: Construct Spring IOC Container from configuration file

ConfigurableApplicationContext context = new ClassPathXmlApplicationContext("spring-beans.xml");

String[] beans = context.getBeanDefinitionNames();
System.out.println(Arrays.toString(beans));

System.out.println("/////////////// --- IOC DI --- ///////////////");
Item itemA = context.getBean("itemA", Item.class);
Item itemB = context.getBean("itemB", Item.class);
Item itemC = context.getBean("itemC", Item.class);
```

Get bean manual via bean name



```java
@Controller
@RequestMapping("customer")
public class CustomerController {
```



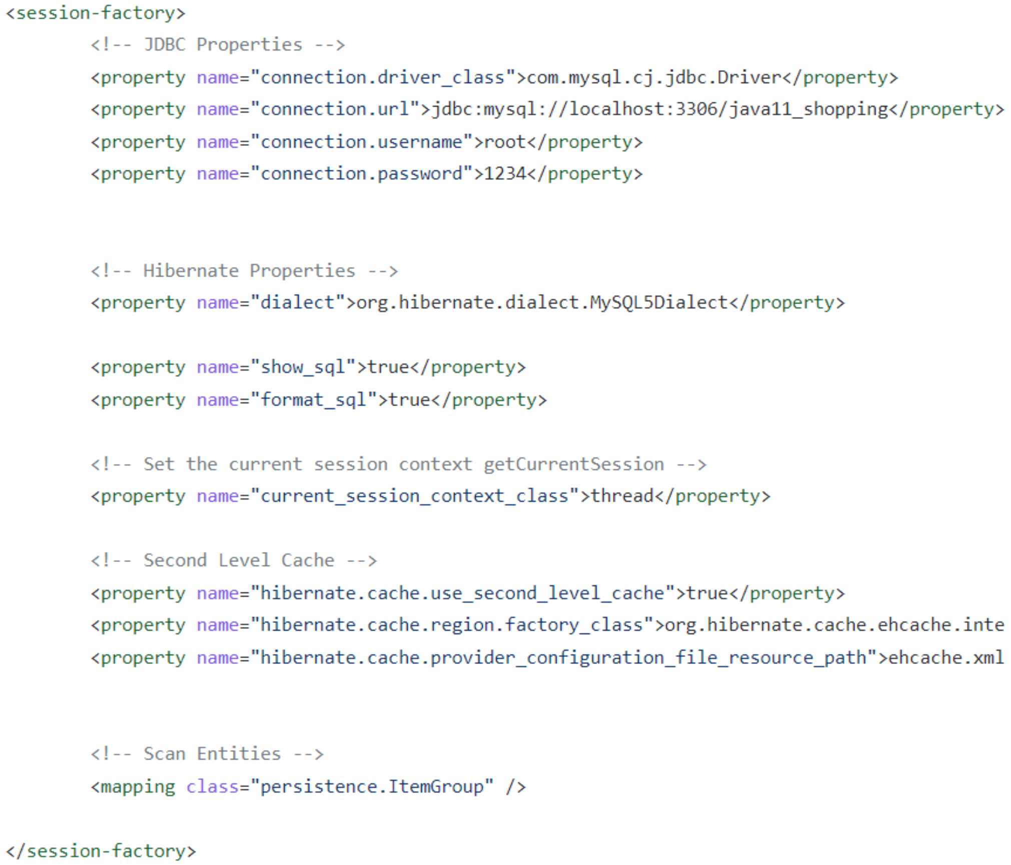
```java
        @Autowired
        private CustomerService customerService;

        @GetMapping(value = {"", "/", "/{orderBy}"})
        public String index(Model model, @PathVariable(required = false, value = "orderBy") String orderByLink) {
                // orderByFirstName(sortByFirstName), orderByLastName(...), orderByEmail(..)
                List<Customer> customers = customerService.getAll(getSortOrder(orderByLink));
```

inject customerService bean from container

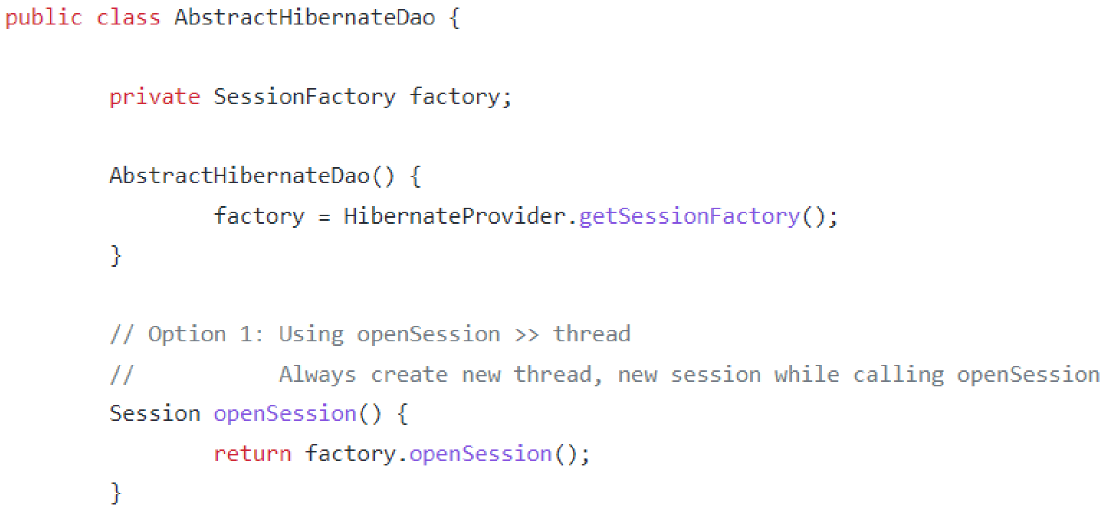
Inversion of control – dependency injection

- ➢ Why we use IOC - DI instead of create object with new keyword

- ➢ https://github.com/j4tdn/java11-repository/blob/workspace-qphan-hibernate/lesson18-hibernate/src/main/resources/hibernate.cfg.xml



```xml
<session-factory>
        <!-- JDBC Properties -->
        <property name="connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost:3306/java11_shopping</property>
        <property name="connection.username">root</property>
        <property name="connection.password">1234</property>


        <!-- Hibernate Properties -->
        <property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>

        <property name="show_sql">true</property>
        <property name="format_sql">true</property>

        <!-- Set the current session context getCurrentSession -->
        <property name="current_session_context_class">thread</property>

        <!-- Second Level Cache -->
        <property name="hibernate.cache.use_second_level_cache">true</property>
        <property name="hibernate.cache.region.factory_class">org.hibernate.cache.ehcache.inte
        <property name="hibernate.cache.provider_configuration_file_resource_path">ehcache.xml


        <!-- Scan Entities -->
        <mapping class="persistence.ItemGroup" />

</session-factory>
```

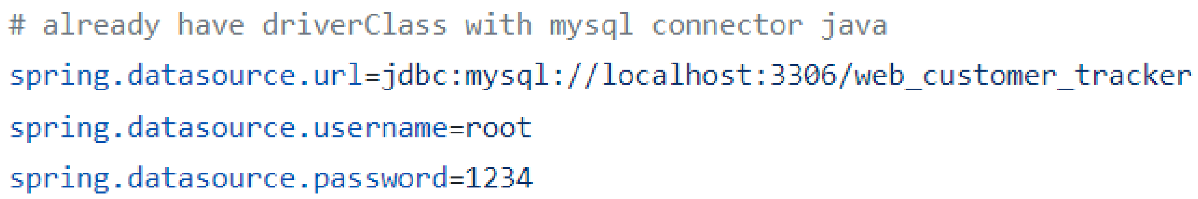


```java
public class AbstractHibernateDao {

        private SessionFactory factory;

        AbstractHibernateDao() {
                factory = HibernateProvider.getSessionFactory();
        }

        // Option 1: Using openSession >> thread
        //          Always create new thread, new session while calling openSession
        Session openSession() {
                return factory.openSession();
        }
```

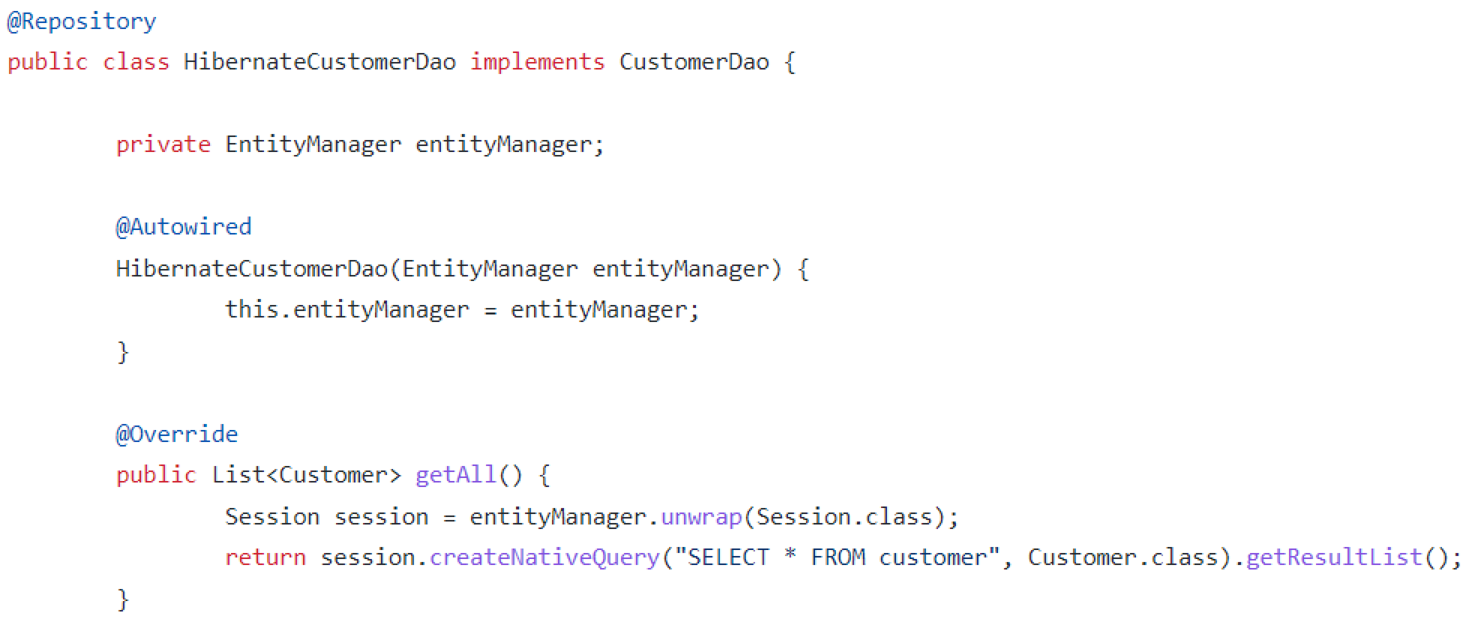
Inversion of control – dependency injection

- Why we use Inversion of control instead of create object with new keyword

- https://github.com/j4tdn/java89-repository/blob/workspace-qphan-springfw/11-spring-boot-webapp-crud/src/main/resources/application.properties



```
# already have driverClass with mysql connector java
spring.datasource.url=jdbc:mysql://localhost:3306/web_customer_tracker
spring.datasource.username=root
spring.datasource.password=1234
```



```
@Repository
public class HibernateCustomerDao implements CustomerDao {

	private EntityManager entityManager;

	@Autowired
	HibernateCustomerDao(EntityManager entityManager) {
		this.entityManager = entityManager;
	}

	@Override
	public List<Customer> getAll() {
		Session session = entityManager.unwrap(Session.class);
		return session.createNativeQuery("SELECT * FROM customer", Customer.class).getResultList();
	}
```

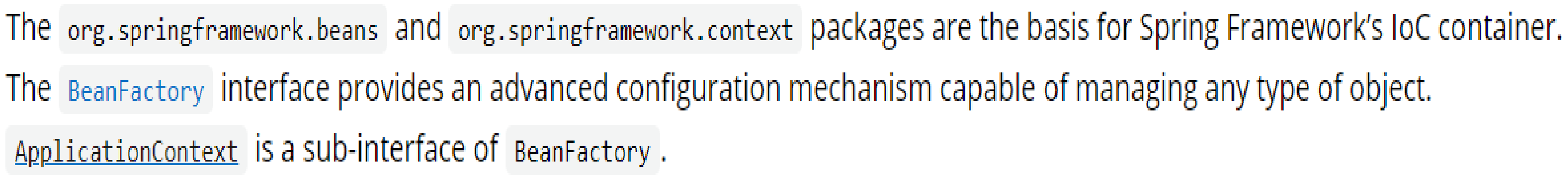
Inversion of control – IOC container

- Spring Framework implementation of the Inversion of Control (IoC) principle.

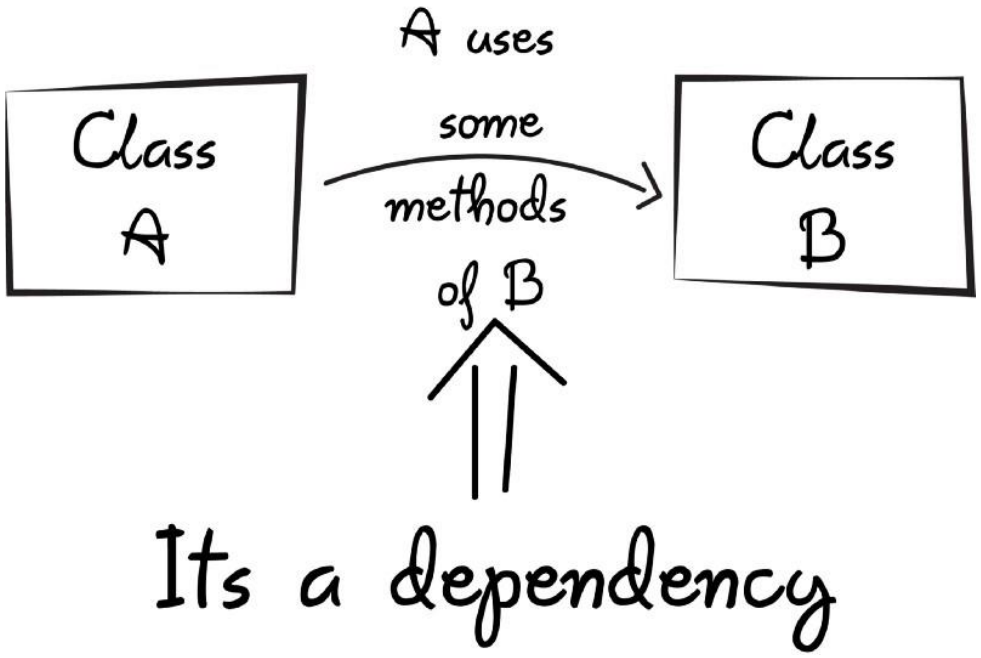
- IoC is also known as dependency injection (DI). It is a process whereby objects define their dependencies (that is, the other objects they work with) only through constructor arguments, arguments to a factory method, or properties that are set on the object instance after it is constructed or returned from a factory method.

- The container then injects those dependencies when it creates the bean. This process is fundamentally the inverse (hence the name, Inversion of Control) of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes

Dependency

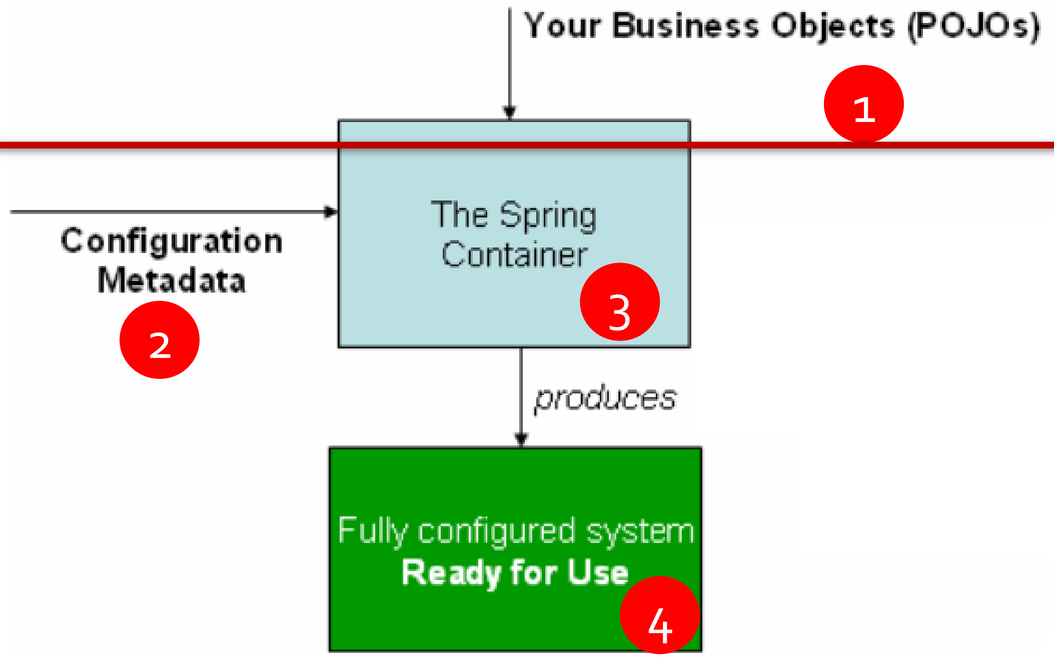


The `org.springframework.beans` and `org.springframework.context` packages are the basis for Spring Framework's IoC container. The `BeanFactory` interface provides an advanced configuration mechanism capable of managing any type of object. `ApplicationContext` is a sub-interface of `BeanFactory`.



Spring IoC container

Spring IOC container



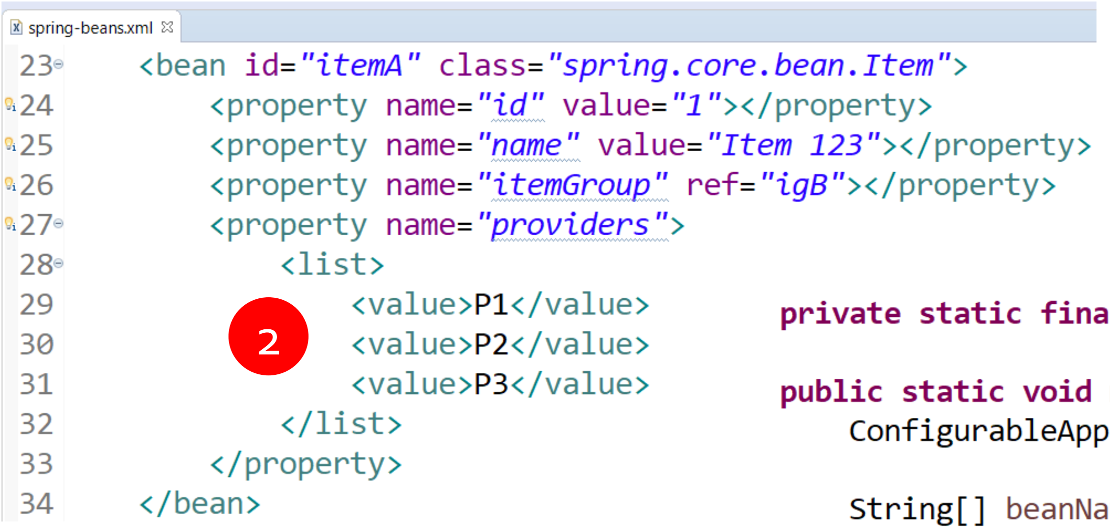
The `org.springframework.context.ApplicationContext` interface represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the beans. The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata. The configuration metadata is represented in XML, Java annotations, or Java code. It lets you express the objects that compose your application and the rich interdependencies between those objects.

Several implementations of the `ApplicationContext` interface are supplied with Spring. In stand-alone applications, it is common to create an instance of `ClassPathXmlApplicationContext` or `FileSystemXmlApplicationContext`. While XML has been the traditional format for defining configuration metadata, you can instruct the container to use Java annotations or code as the metadata format by providing a small amount of XML configuration to declaratively enable support for these additional metadata formats.

Spring IOC container

1

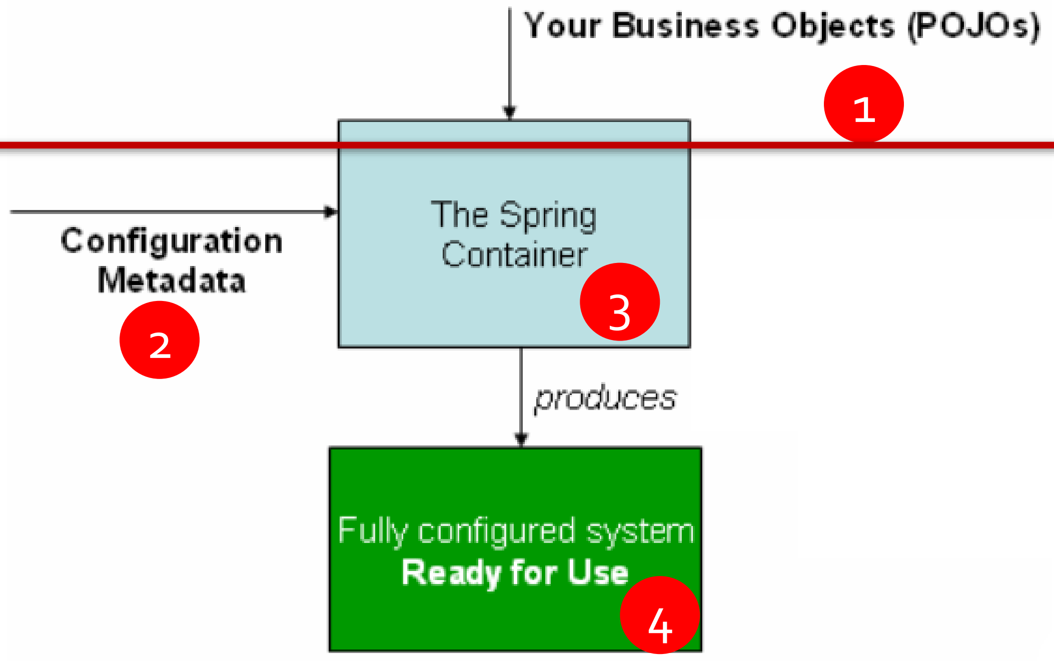
```java
public class Item {
    private Integer id;
    private String name;
    private List<String> providers;
    private ItemGroup itemGroup;
```



spring-beans.xml

2

```xml
    <bean id="itemA" class="spring.core.bean.Item">
        <property name="id" value="1"></property>
        <property name="name" value="Item 123"></property>
        <property name="itemGroup" ref="igB"></property>
        <property name="providers">
            <list>
                <value>P1</value>
                <value>P2</value>
                <value>P3</value>
            </list>
        </property>
    </bean>
```



Your Business Objects (POJOs)
1
Configuration Metadata
2
The Spring Container
3
produces
Fully configured system Ready for Use
4

3

4

```java
private static final String contextPath = "spring-beans.xml";

public static void main(String[] args) {
    ConfigurableApplicationContext context = new ClassPathXmlApplicationContext(contextPath);

    String[] beanNames = context.getBeanDefinitionNames();

    for (String beanName: beanNames) {
        System.out.println(beanName);
    }

    System.out.println("=================");
    ItemGroup igA = context.getBean("igA", ItemGroup.class);
    ItemGroup igC = context.getBean("igC", ItemGroup.class);
    Item itemA = context.getBean("itemA", Item.class);
    ClientService clienService = context.getBean("clientA", ClientService.class);
```

Configuration metadata

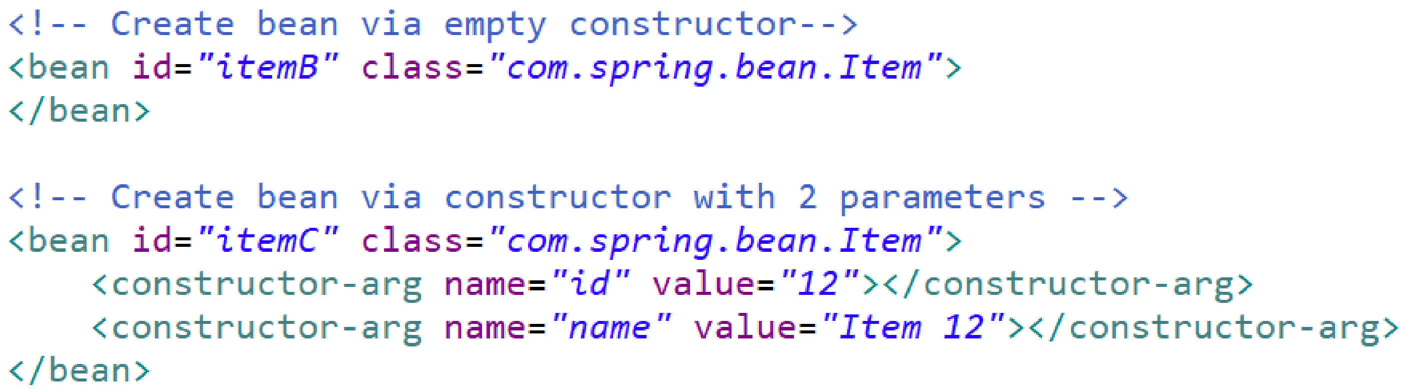
- Spring IoC container consumes a form of configuration metadata. This configuration metadata represents how you, as an application developer, tell the Spring container to instantiate, configure, and assemble the objects in your application.
- Configuration metadata is traditionally supplied in a simple and intuitive XML format, which is what most of this chapter uses to convey key concepts and features of the Spring IoC container.

Configuration metadata

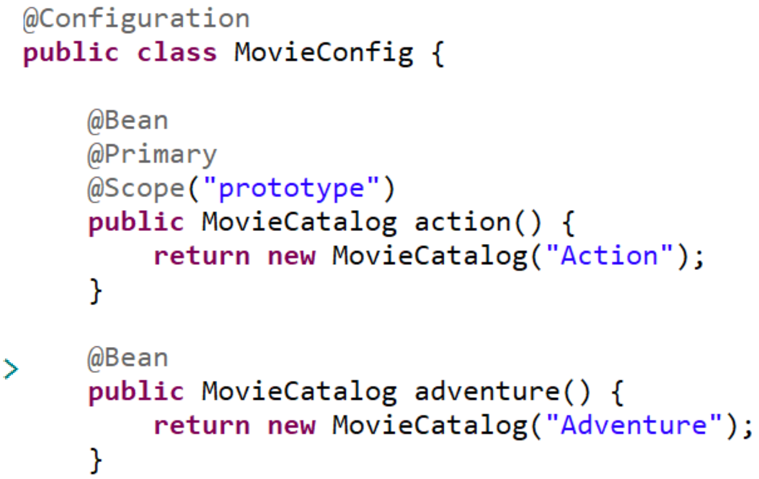
For information about using other forms of metadata with the Spring container, see:

- Annotation-based configuration: Spring 2.5 introduced support for annotation-based configuration metadata.
- Java-based configuration: Starting with Spring 3.0, many features provided by the Spring JavaConfig project became part of the core Spring Framework. Thus, you can define beans external to your application classes by using Java rather than XML files. To use these new features, see the `@Configuration`, `@Bean`, `@Import`, and `@DependsOn` annotations.

Spring configuration consists of at least one and typically more than one bean definition that the container must manage. XML-based configuration metadata configures these beans as `<bean/>` elements inside a top-level `<beans/>` element. Java configuration typically uses `@Bean`-annotated methods within a `@Configuration` class.

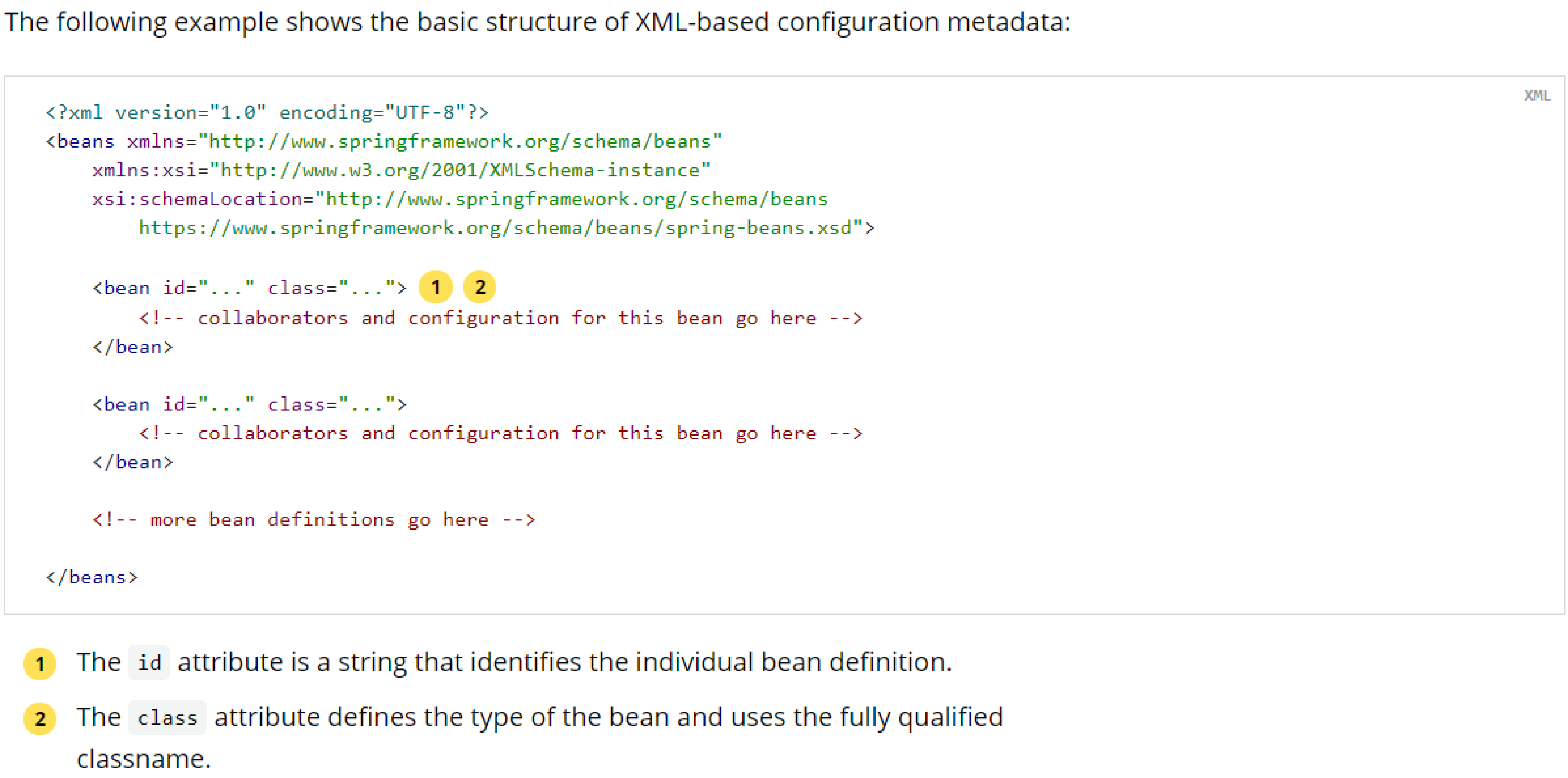


```xml
<!-- Create bean via empty constructor-->
<bean id="itemB" class="com.spring.bean.Item">
</bean>

<!-- Create bean via constructor with 2 parameters -->
<bean id="itemC" class="com.spring.bean.Item">
    <constructor-arg name="id" value="12"></constructor-arg>
    <constructor-arg name="name" value="Item 12"></constructor-arg>
</bean>
```



```java
@Configuration
public class MovieConfig {

    @Bean
    @Primary
    @Scope("prototype")
    public MovieCatalog action() {
        return new MovieCatalog("Action");
    }

    @Bean
    public MovieCatalog adventure() {
        return new MovieCatalog("Adventure");
    }
```

Configuration metadata



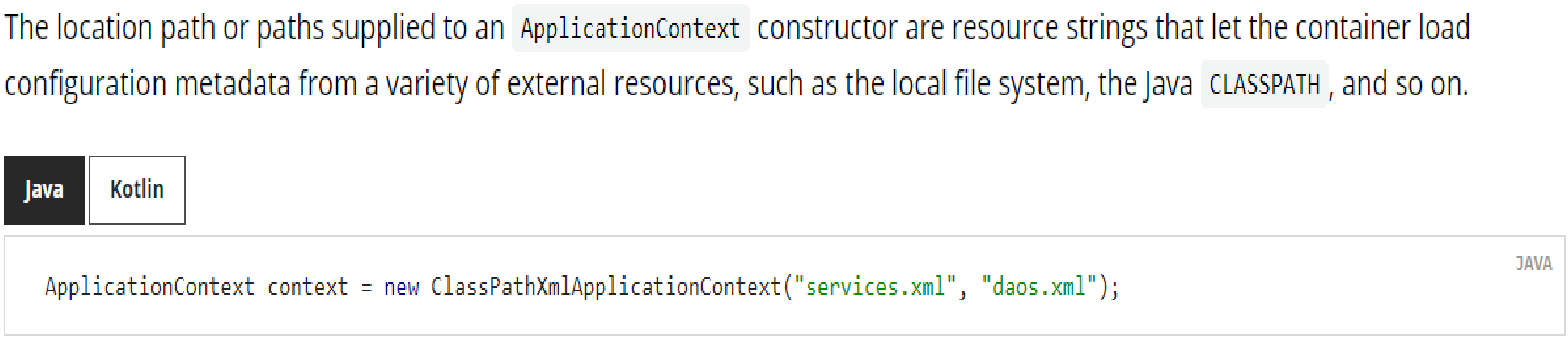
The following example shows the basic structure of XML-based configuration metadata:

XML

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

    <bean id="..." class="..."> 1 2
        <!-- 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>
```

1. The `id` attribute is a string that identifies the individual bean definition.
2. The `class` attribute defines the type of the bean and uses the fully qualified classname.

Application Context – Bean Factory

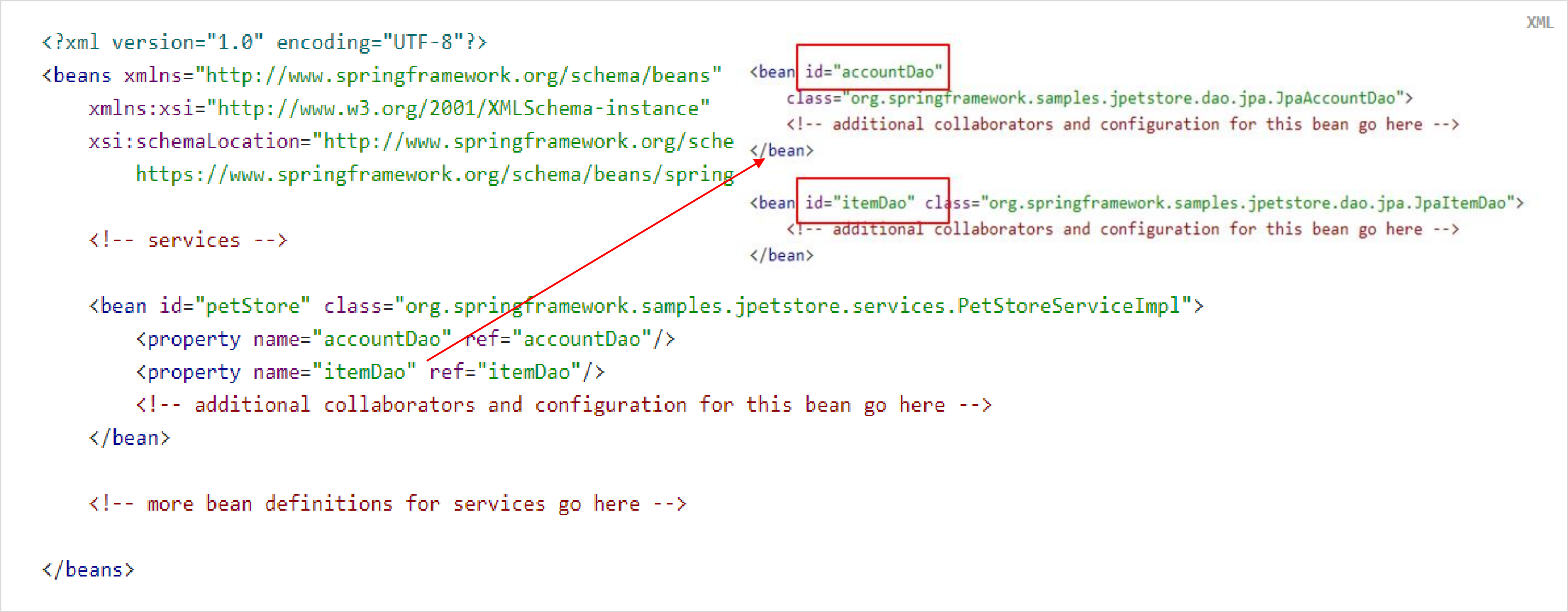


The location path or paths supplied to an `ApplicationContext` constructor are resource strings that let the container load configuration metadata from a variety of external resources, such as the local file system, the Java `CLASSPATH`, and so on.

```java
ApplicationContext context = new ClassPathXmlApplicationContext("services.xml", "daos.xml");
```

Configuration metadata

The following example shows the service layer objects `(services.xml)` configuration file:



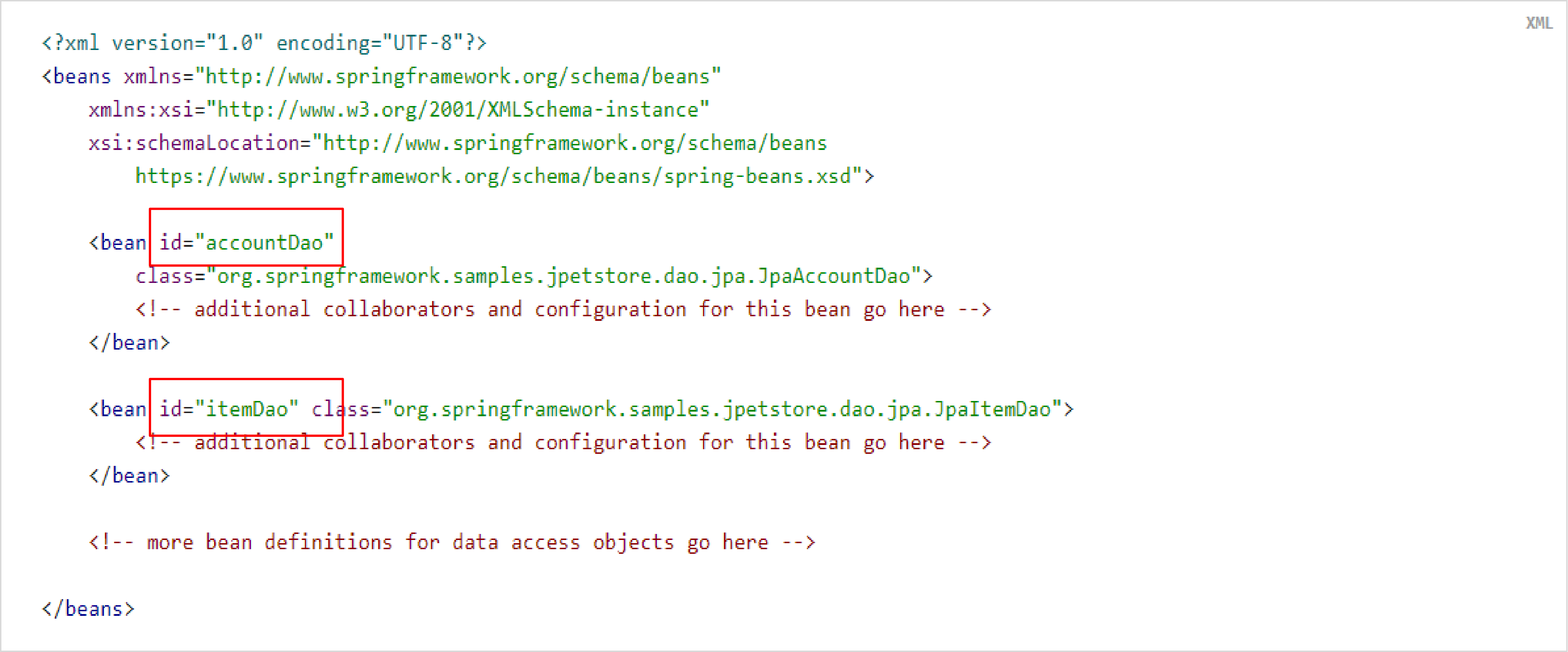
XML

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

    <!-- services -->

    <bean id="petStore" class="org.springframework.samples.jpetstore.services.PetStoreServiceImpl">
        <property name="accountDao" ref="accountDao"/>
        <property name="itemDao" ref="itemDao"/>
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

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

</beans>
```

```xml
<bean id="accountDao"
    class="org.springframework.samples.jpetstore.dao.jpa.JpaAccountDao">
    <!-- additional collaborators and configuration for this bean go here -->
</bean>

<bean id="itemDao" class="org.springframework.samples.jpetstore.dao.jpa.JpaItemDao">
    <!-- additional collaborators and configuration for this bean go here -->
</bean>
```

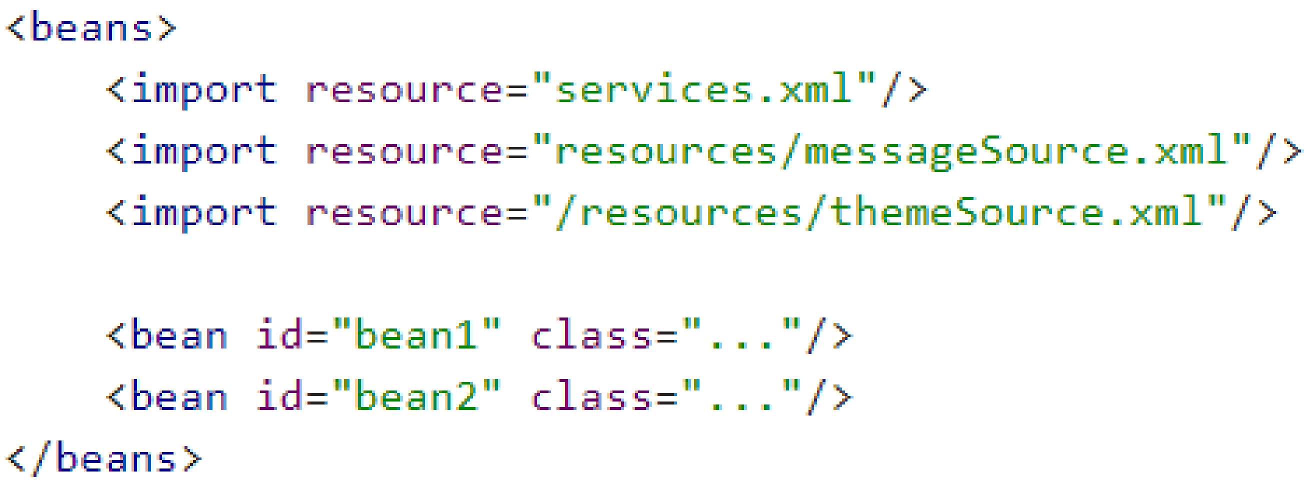
Configuration metadata

The following example shows the data access objects `daos.xml` file:



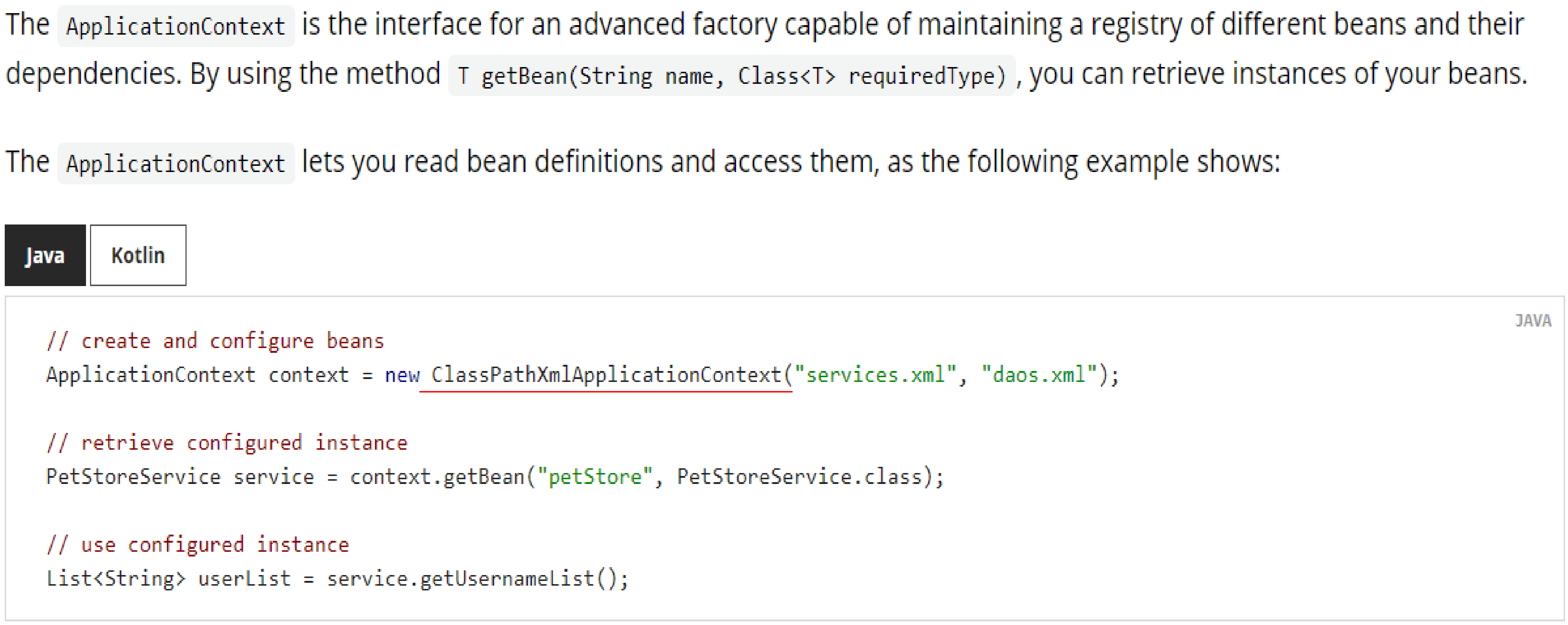
```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="accountDao"
        class="org.springframework.samples.jpetstore.dao.jpa.JpaAccountDao">
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

    <bean id="itemDao" class="org.springframework.samples.jpetstore.dao.jpa.JpaItemDao">
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>

    <!-- more bean definitions for data access objects go here -->

</beans>
```

Configuration metadata – XML composing



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

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

Application Context

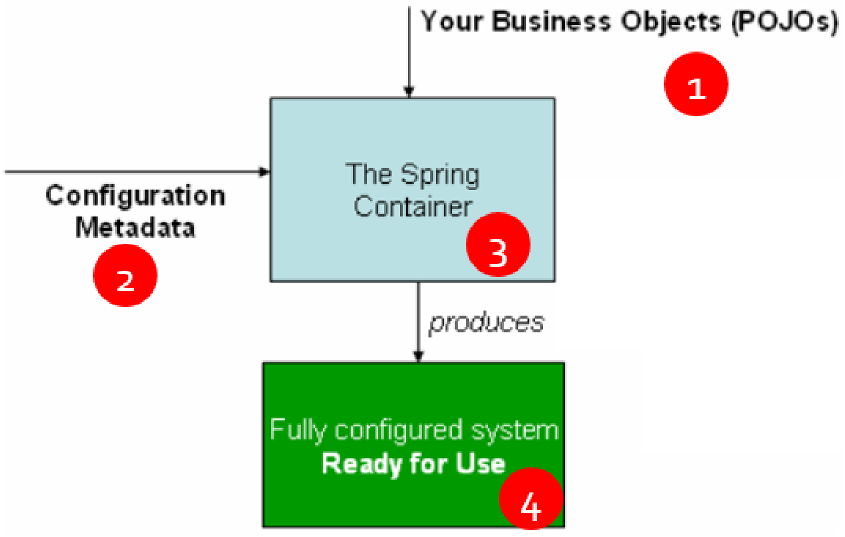


The `ApplicationContext` is the interface for an advanced factory capable of maintaining a registry of different beans and their dependencies. By using the method `T getBean(String name, Class<T> requiredType)`, you can retrieve instances of your beans.

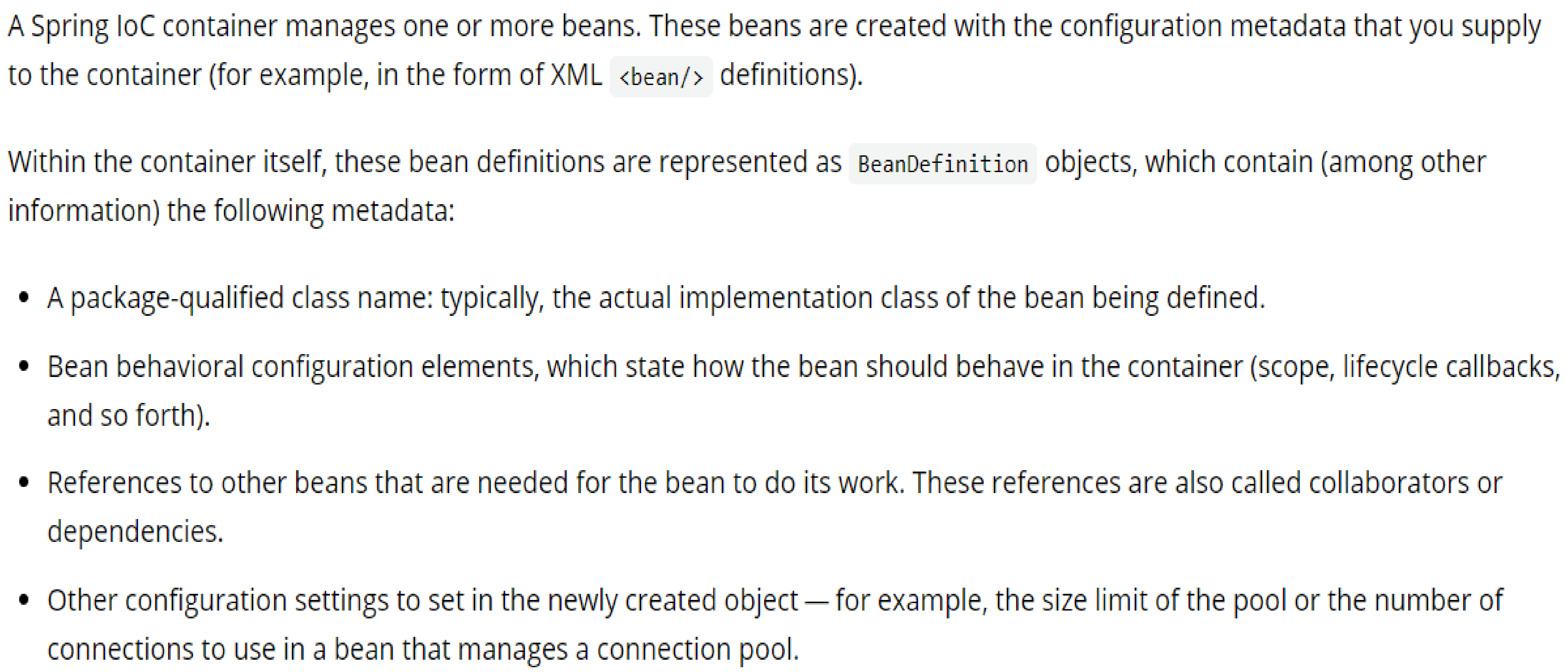
The `ApplicationContext` lets you read bean definitions and access them, as the following example shows:

```java
// create and configure beans
ApplicationContext context = new ClassPathXmlApplicationContext("services.xml", "daos.xml");

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

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

Spring Bean


Your Business Objects (POJOs)
1
Configuration Metadata
2
The Spring Container
3
produces
Fully configured system
Ready for Use
4

Spring Bean overview

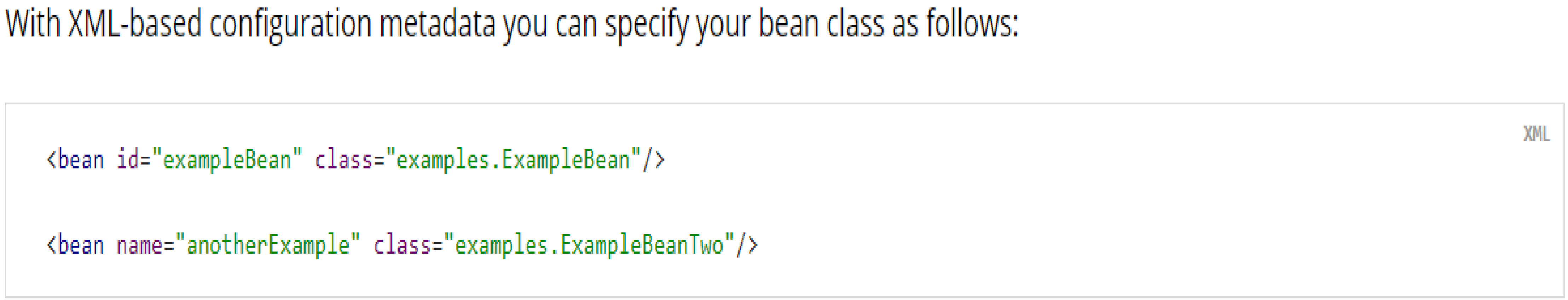


A Spring IoC container manages one or more beans. These beans are created with the configuration metadata that you supply to the container (for example, in the form of XML `<bean/>` definitions).

Within the container itself, these bean definitions are represented as `BeanDefinition` objects, which contain (among other information) the following metadata:

- A package-qualified class name: typically, the actual implementation class of the bean being defined.
- Bean behavioral configuration elements, which state how the bean should behave in the container (scope, lifecycle callbacks, and so forth).
- References to other beans that are needed for the bean to do its work. These references are also called collaborators or dependencies.
- Other configuration settings to set in the newly created object — for example, the size limit of the pool or the number of connections to use in a bean that manages a connection pool.

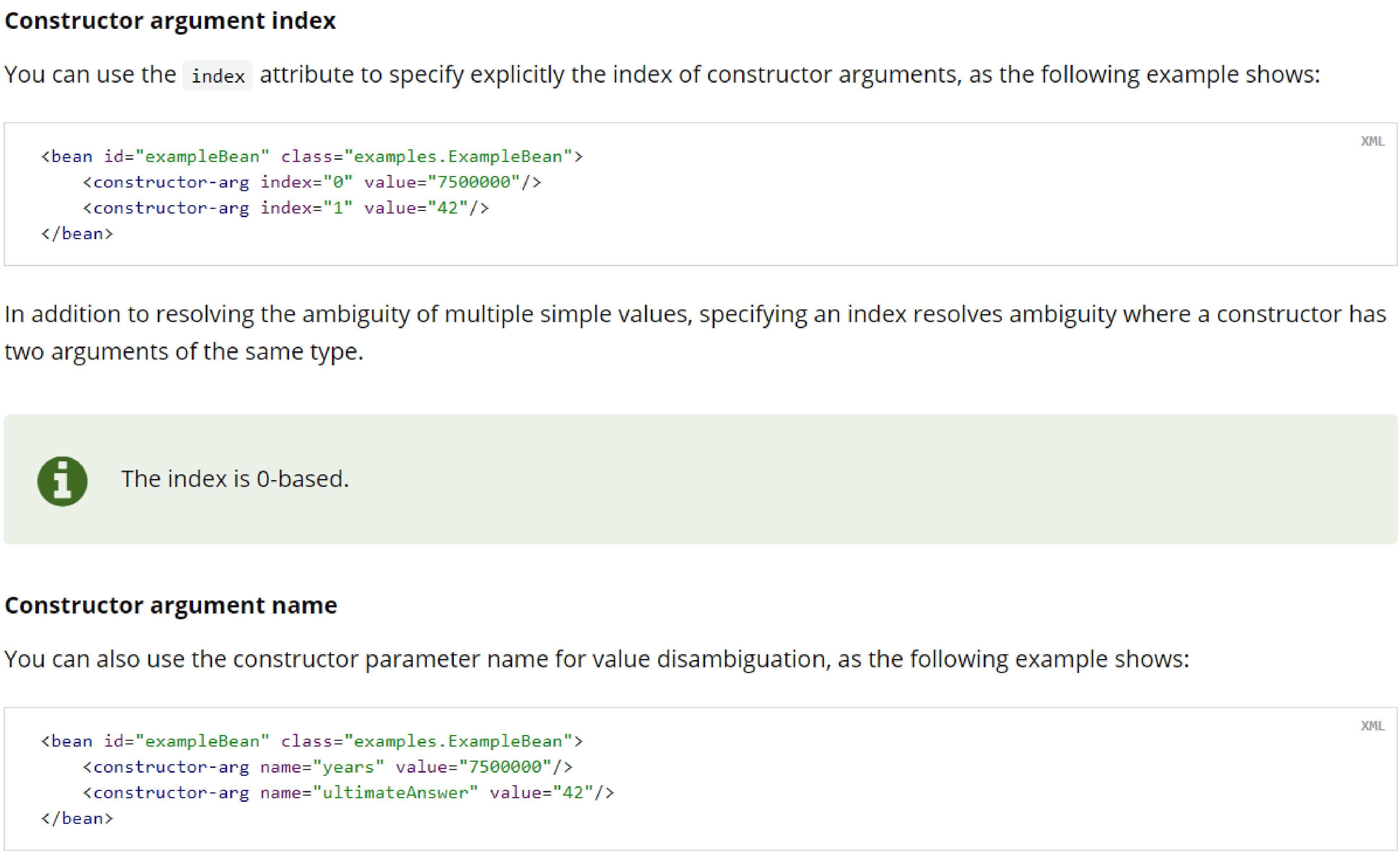
Initial a Bean via constructor



With XML-based configuration metadata you can specify your bean class as follows:

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

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

Initial a Bean via constructor



Constructor argument index

You can use the `index` attribute to specify explicitly the index of constructor arguments, as the following example shows:

XML

```xml
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg index="0" value="7500000"/>
    <constructor-arg index="1" value="42"/>
</bean>
```

In addition to resolving the ambiguity of multiple simple values, specifying an index resolves ambiguity where a constructor has two arguments of the same type.

> The index is 0-based.

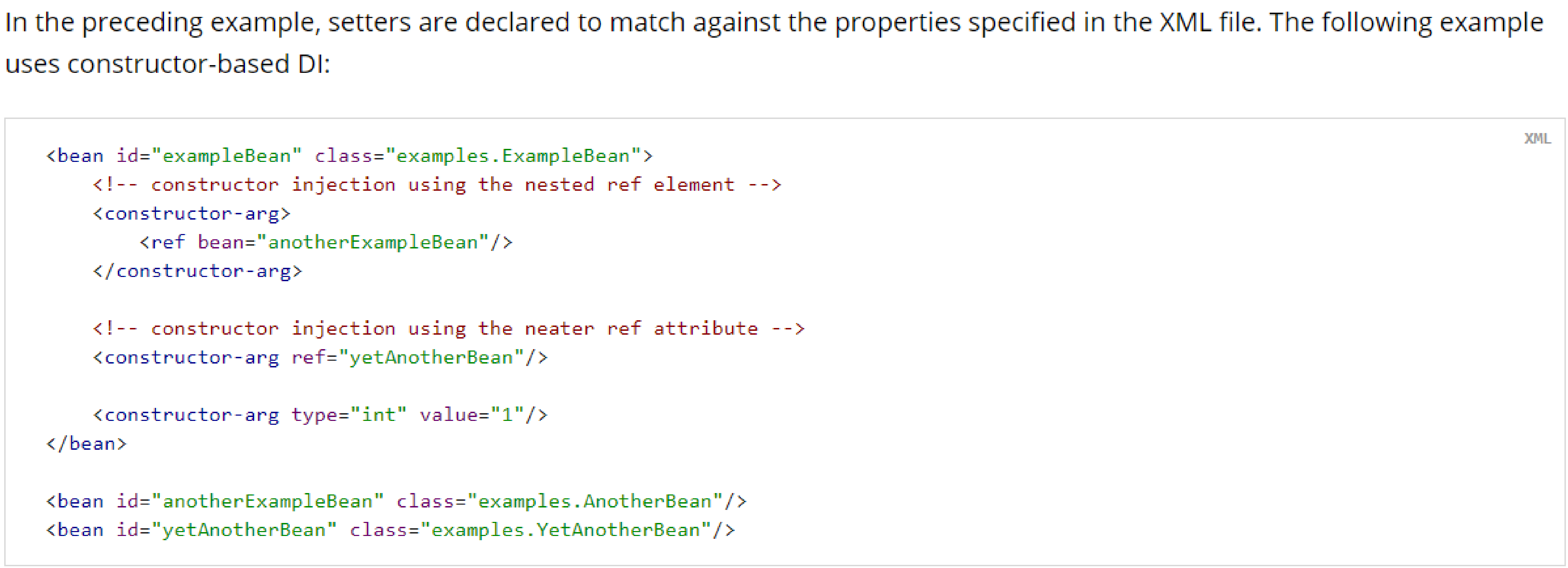
Constructor argument name

You can also use the constructor parameter name for value disambiguation, as the following example shows:

XML

```xml
<bean id="exampleBean" class="examples.ExampleBean">
    <constructor-arg name="years" value="7500000"/>
    <constructor-arg name="ultimateAnswer" value="42"/>
</bean>
```

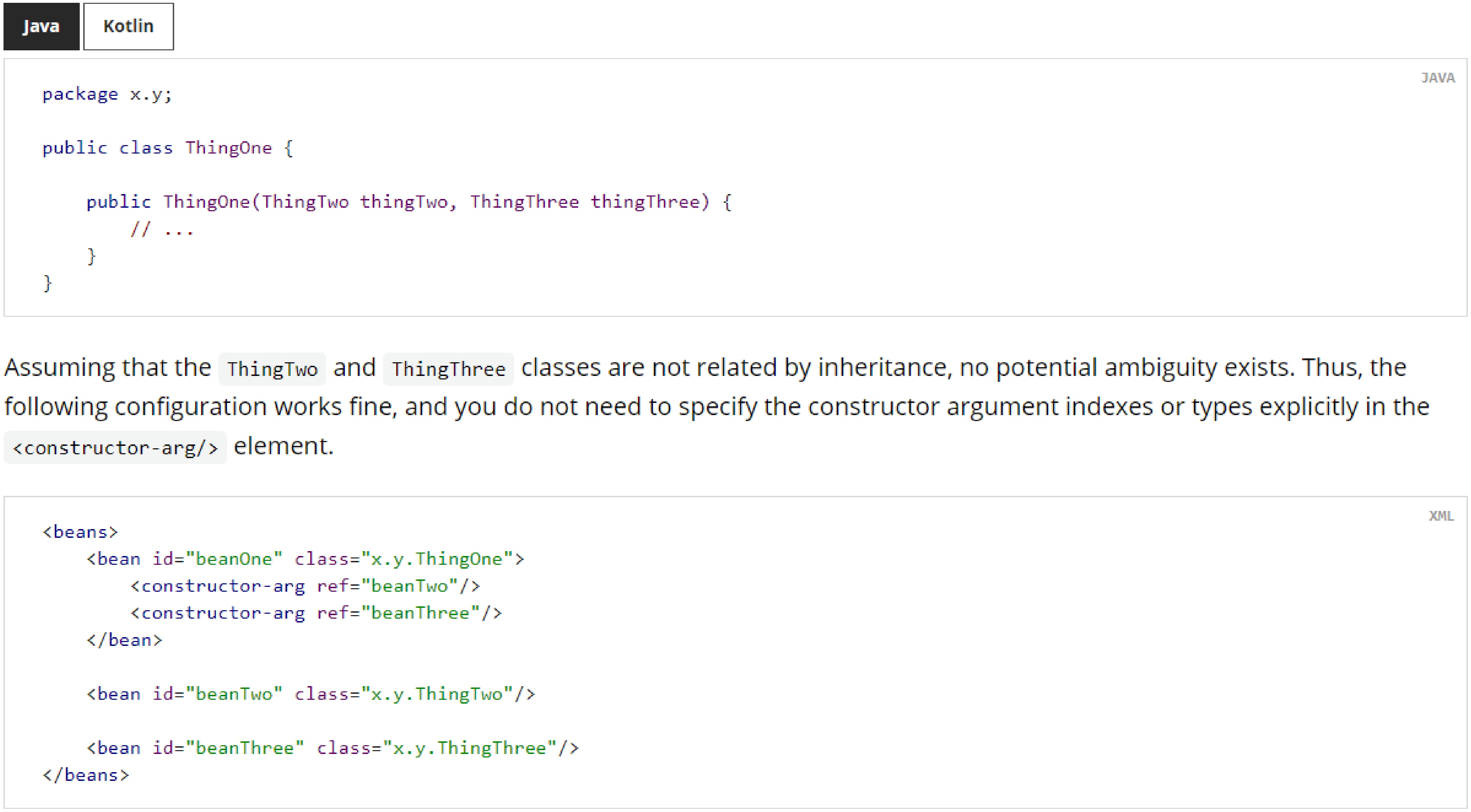
Initial a Bean via constructor



In the preceding example, setters are declared to match against the properties specified in the XML file. The following example uses constructor-based DI:

```xml
<bean id="exampleBean" class="examples.ExampleBean">
    <!-- constructor injection using the nested ref element -->
    <constructor-arg>
        <ref bean="anotherExampleBean"/>
    </constructor-arg>

    <!-- constructor injection using the neater ref attribute -->
    <constructor-arg ref="yetAnotherBean"/>

    <constructor-arg type="int" value="1"/>
</bean>

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

Initial a Bean via constructor and dependencies

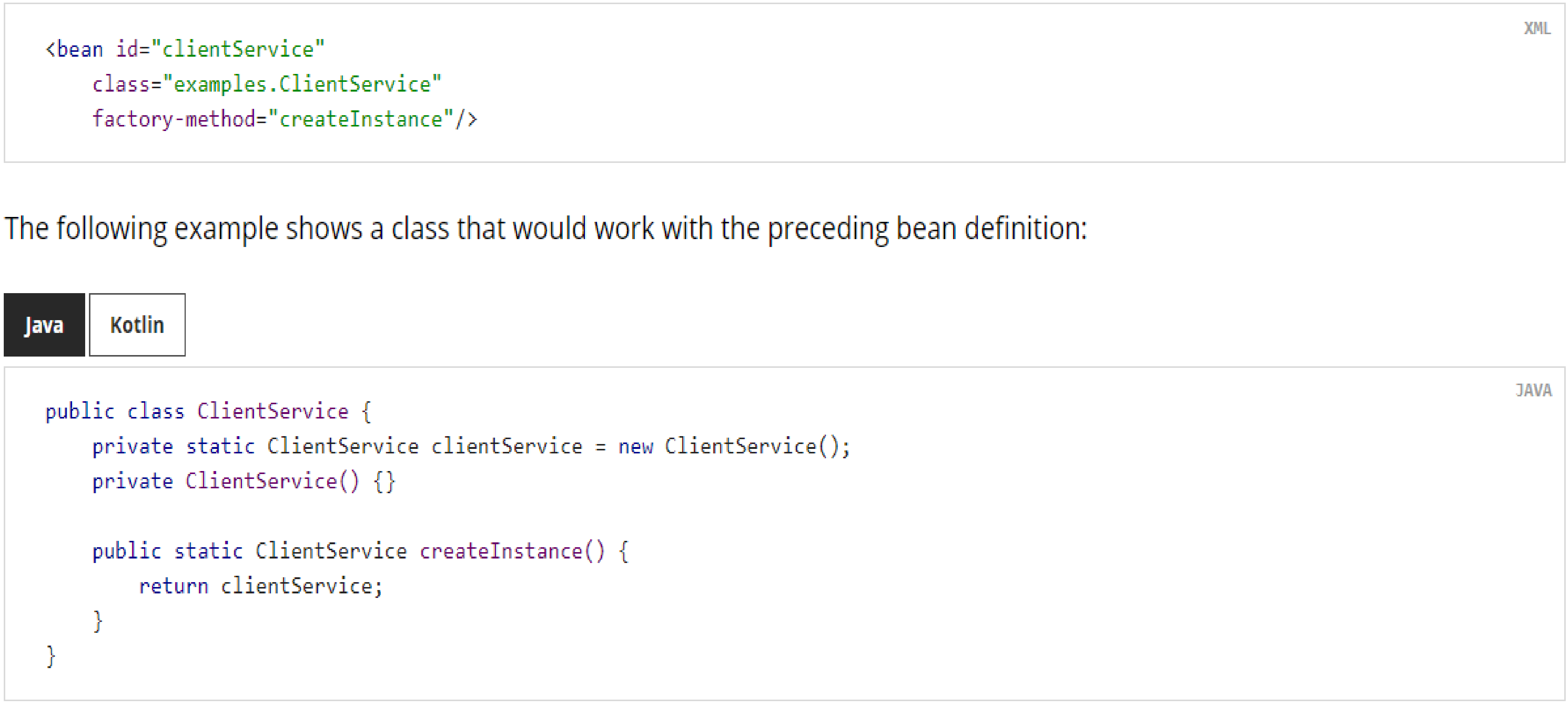


```java
package x.y;

public class ThingOne {

    public ThingOne(ThingTwo thingTwo, ThingThree thingThree) {
        // ...
    }
}
```

Assuming that the `ThingTwo` and `ThingThree` classes are not related by inheritance, no potential ambiguity exists. Thus, the following configuration works fine, and you do not need to specify the constructor argument indexes or types explicitly in the `<constructor-arg/>` element.

```xml
<beans>
    <bean id="beanOne" class="x.y.ThingOne">
        <constructor-arg ref="beanTwo"/>
        <constructor-arg ref="beanThree"/>
    </bean>

    <bean id="beanTwo" class="x.y.ThingTwo"/>

    <bean id="beanThree" class="x.y.ThingThree"/>
</beans>
```

Initial a Bean via factory method

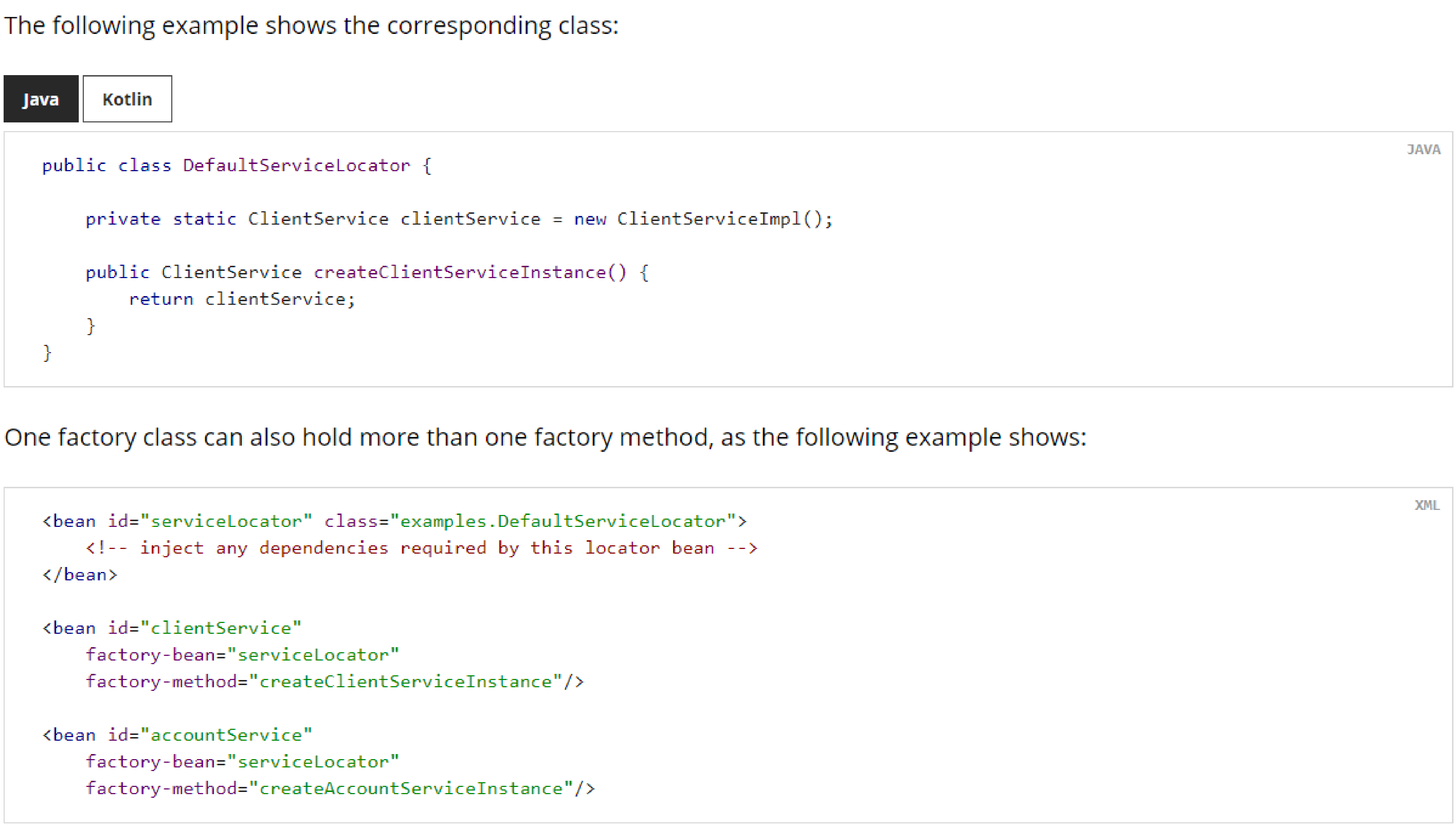


```xml
<bean id="clientService"
    class="examples.ClientService"
    factory-method="createInstance"/>
```

The following example shows a class that would work with the preceding bean definition:

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

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

Initial a Bean via factory method



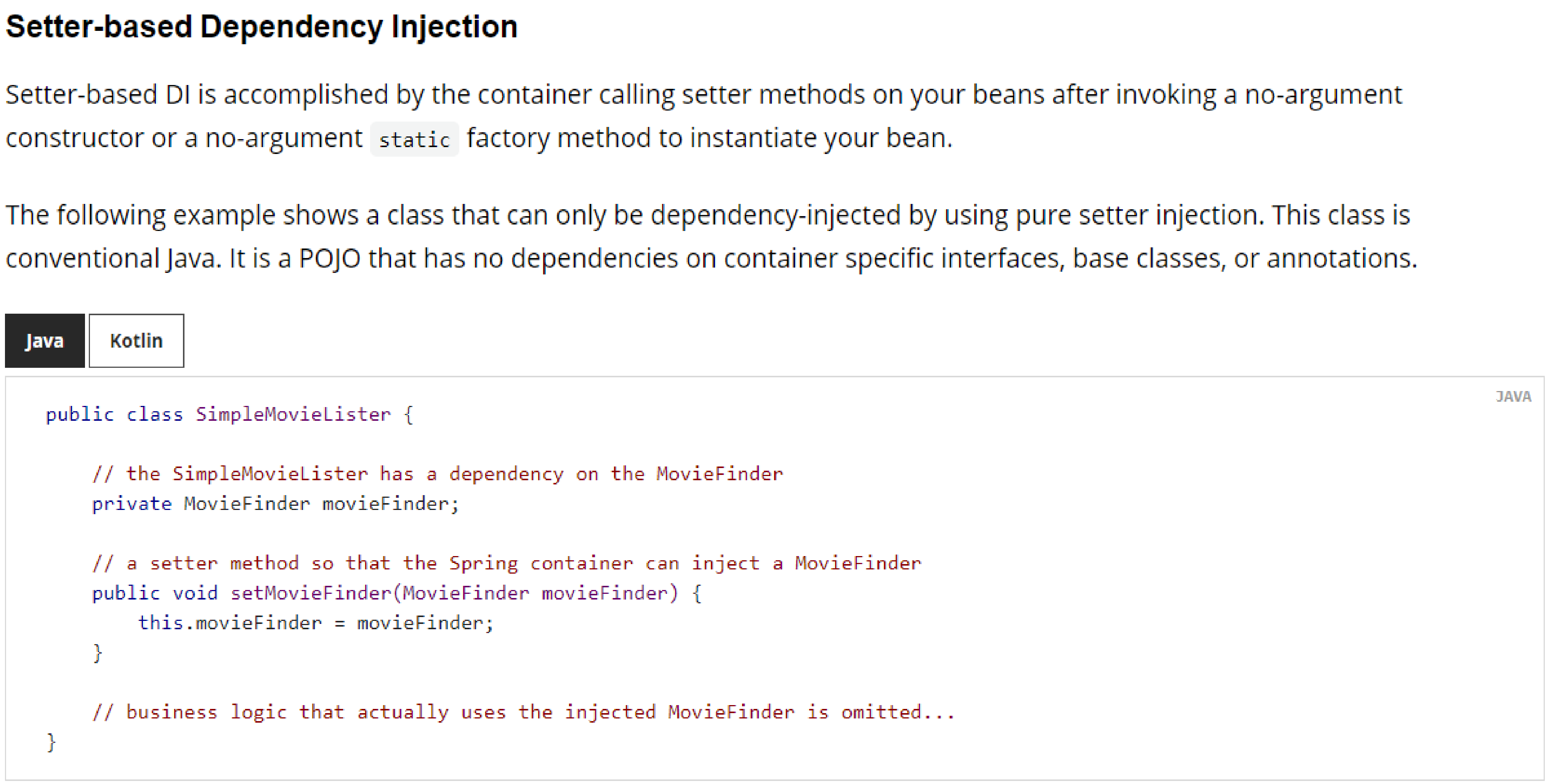
The following example shows the corresponding class:

```java
public class DefaultServiceLocator {

    private static ClientService clientService = new ClientServiceImpl();

    public ClientService createClientServiceInstance() {
        return clientService;
    }
}
```

One factory class can also hold more than one factory method, as the following example shows:

```xml
<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"/>
```

Initial a Bean via setter method



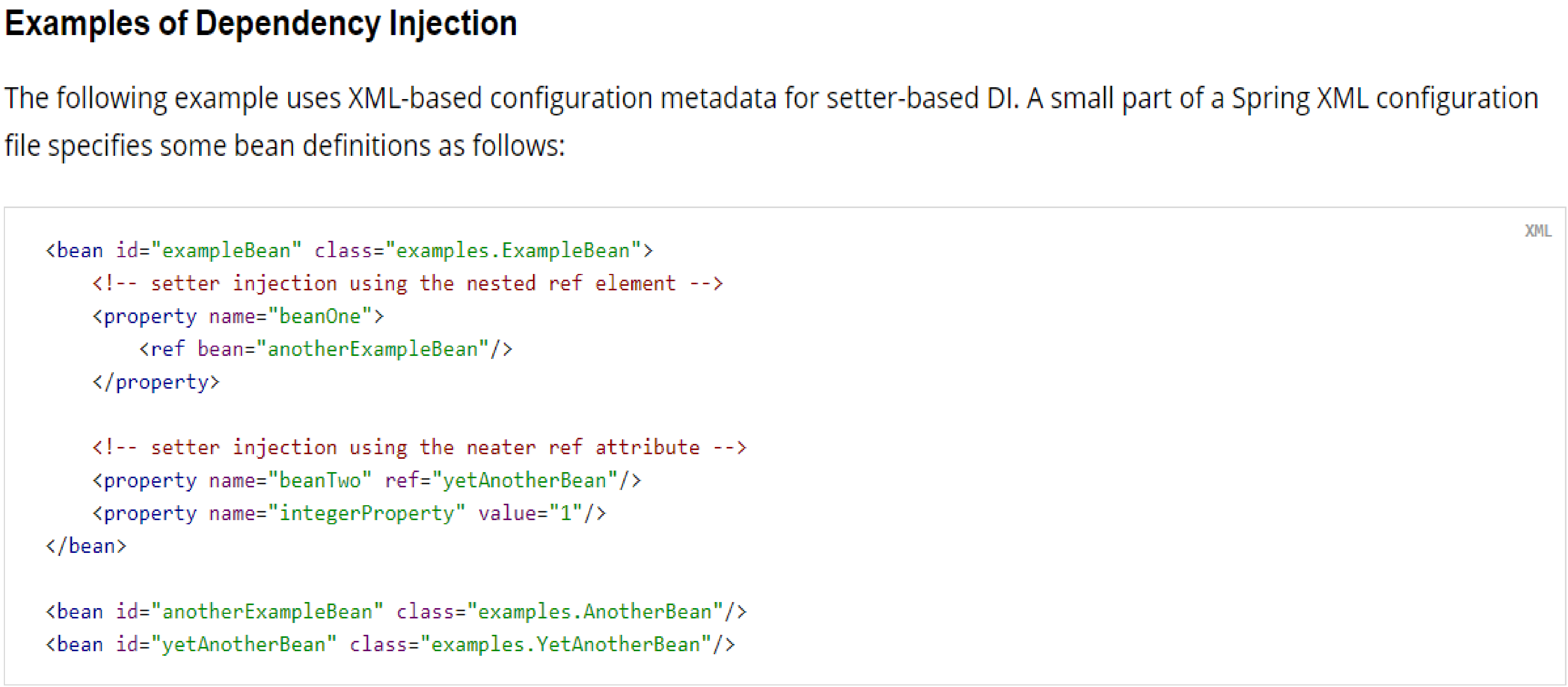
Setter-based Dependency Injection

Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or a no-argument `static` factory method to instantiate your bean.

The following example shows a class that can only be dependency-injected by using pure setter injection. This class is conventional Java. It is a POJO that has no dependencies on container specific interfaces, base classes, or annotations.

```java
public class SimpleMovieLister {

    // the SimpleMovieLister has a dependency on the MovieFinder
    private MovieFinder movieFinder;

    // a setter method so that the Spring container can inject a MovieFinder
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // business logic that actually uses the injected MovieFinder is omitted...
}
```

Initial a Bean via setter method

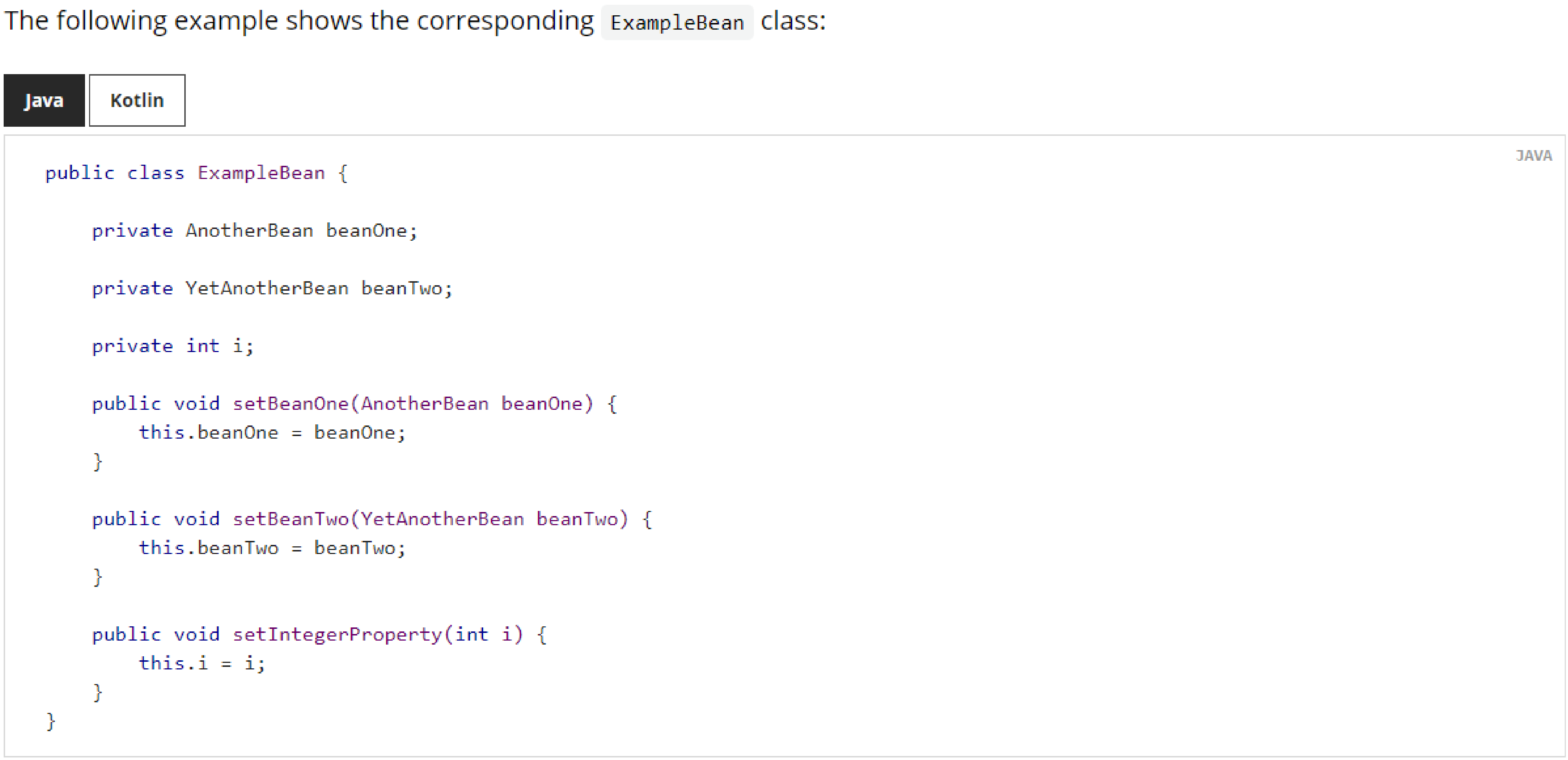


Examples of Dependency Injection

The following example uses XML-based configuration metadata for setter-based DI. A small part of a Spring XML configuration file specifies some bean definitions as follows:

```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"/>
```

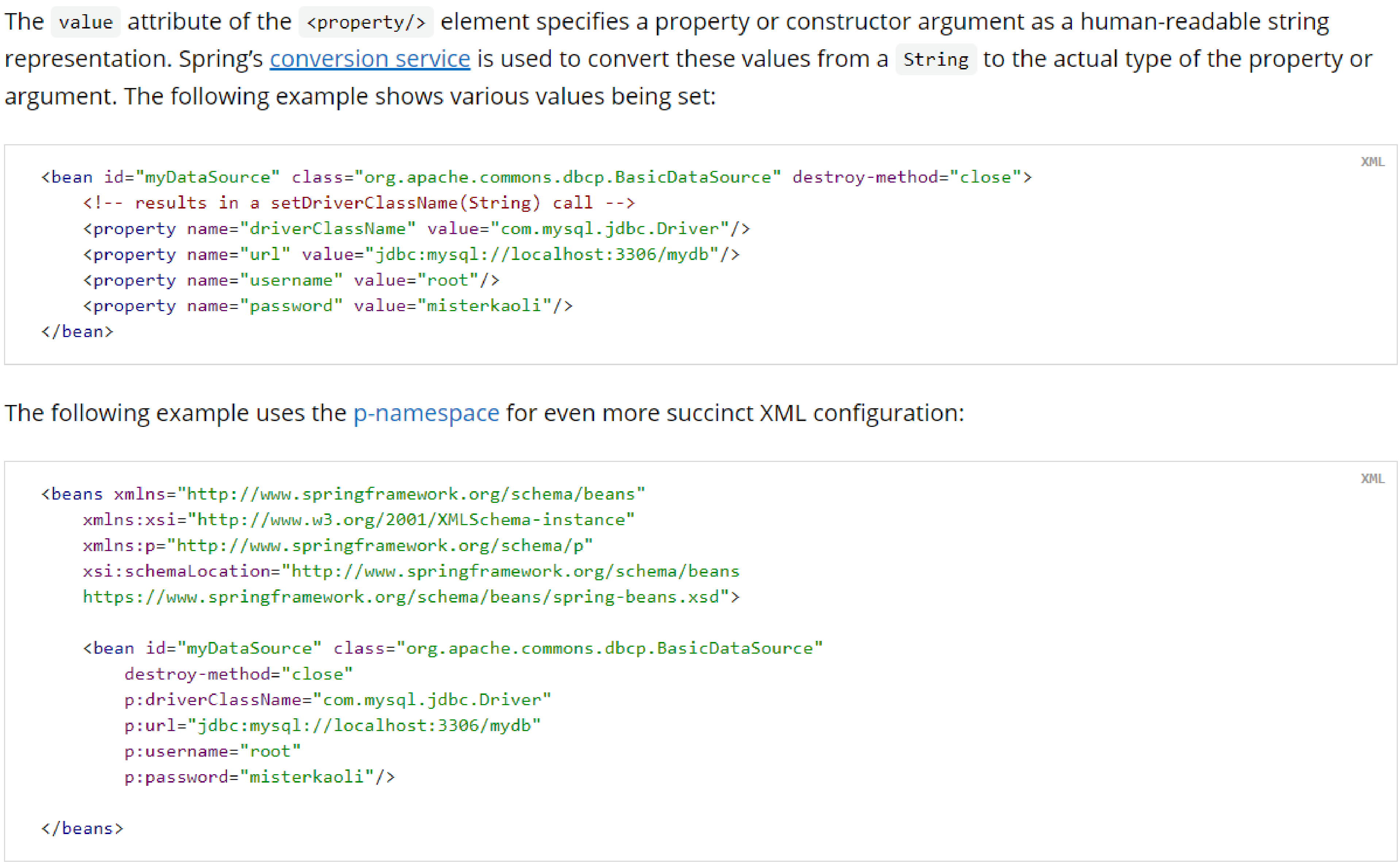
Initial a Bean via setter method



The following example shows the corresponding `ExampleBean` class:

```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;
    }
}
```

Bean with dependency and configuration



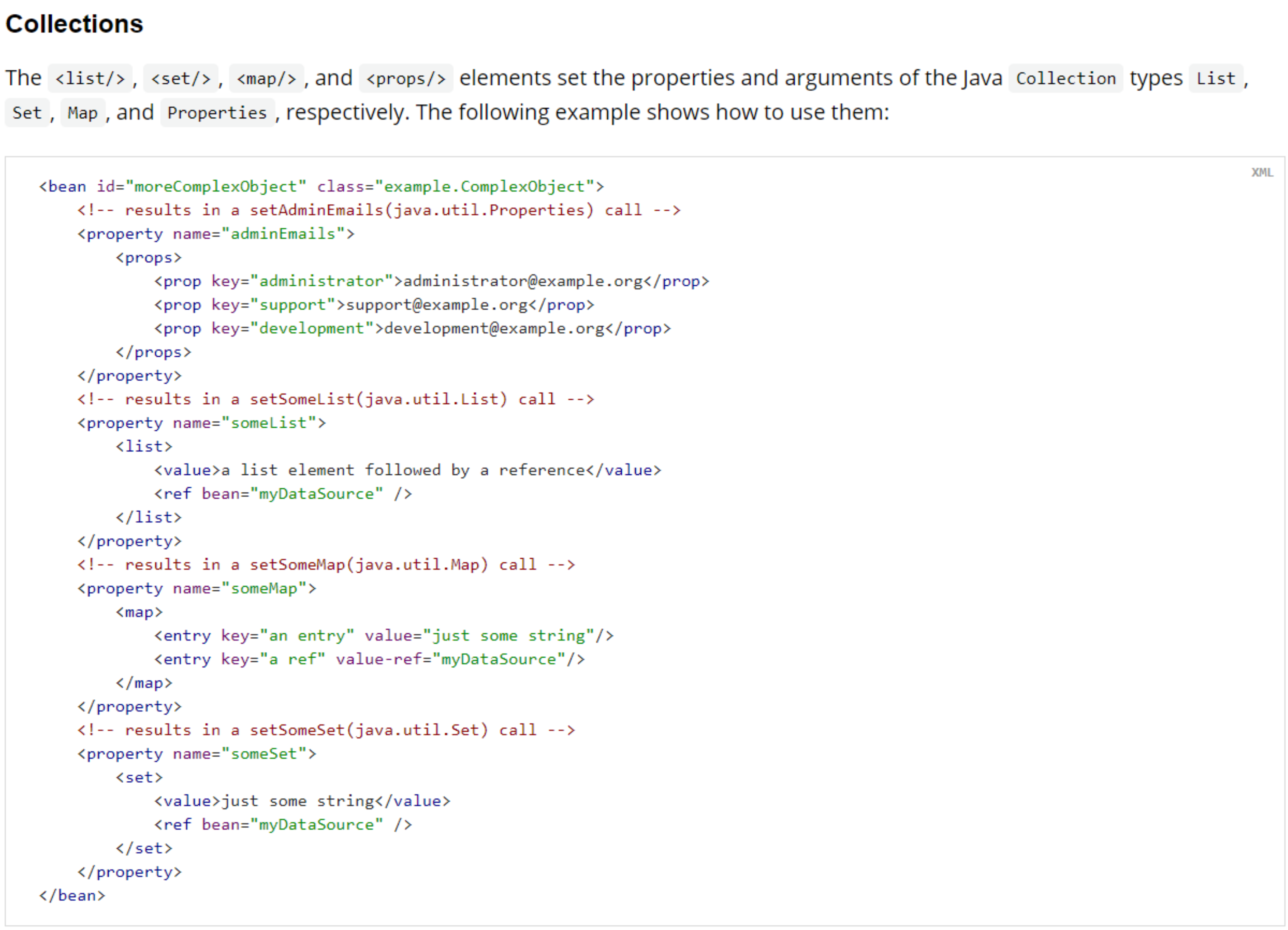
The `value` attribute of the `<property/>` element specifies a property or constructor argument as a human-readable string representation. Spring's conversion service is used to convert these values from a `String` to the actual type of the property or argument. The following example shows various values being set:

```xml
<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="misterkaoli"/>
</bean>
```

The following example uses the p-namespace for even more succinct XML configuration:

```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
    https://www.springframework.org/schema/beans/spring-beans.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="misterkaoli"/>

</beans>
```

Initial a Bean with collection dependency



Collections

The `<list/>`, `<set/>`, `<map/>`, and `<props/>` elements set the properties and arguments of the Java `Collection` types `List`, `Set`, `Map`, and `Properties`, respectively. The following example shows how to use them:

XML

```xml
<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>
```

App Demo – And - Overview

- Understand Inversion of control – Dependency injection principle

- Understand definition of dependency

- Understand Spring Core's IOC container

- Understand definition of Bean

- Understand the way to manage an object as a bean in spring core's container with
  - XML
  - JAVA code
  - JAVA annotation

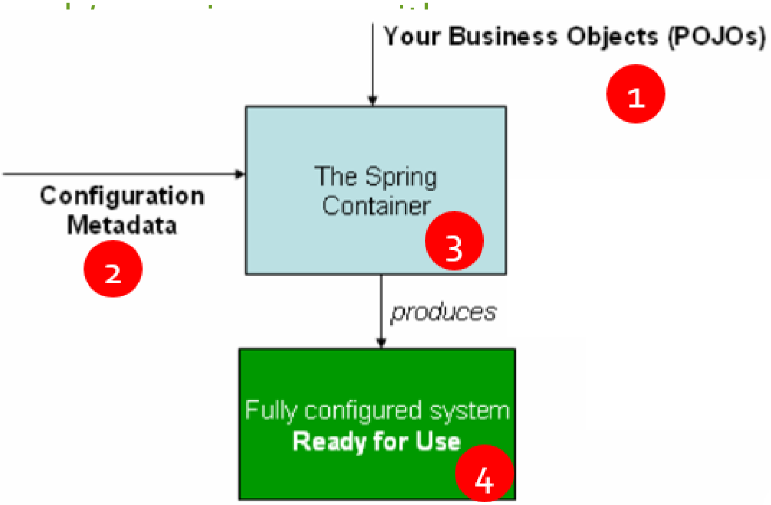
App Demo – And - Overview

- Step 1: Create a maven core project with maven-archetype-webapp or default-template
  - Name: 01-spring-core-xml

- Step 2: Import dependencies
  - https://mvnrepository.com/artifact/org.springframework/spring-core
  - https://mvnrepository.com/artifact/org.springframework/spring-context
  - https://github.com/j4tdn/java89-repository/blob/workspace-qphan-springfw/03-spring-mvc-customer-app-with-annotation/pom.xml

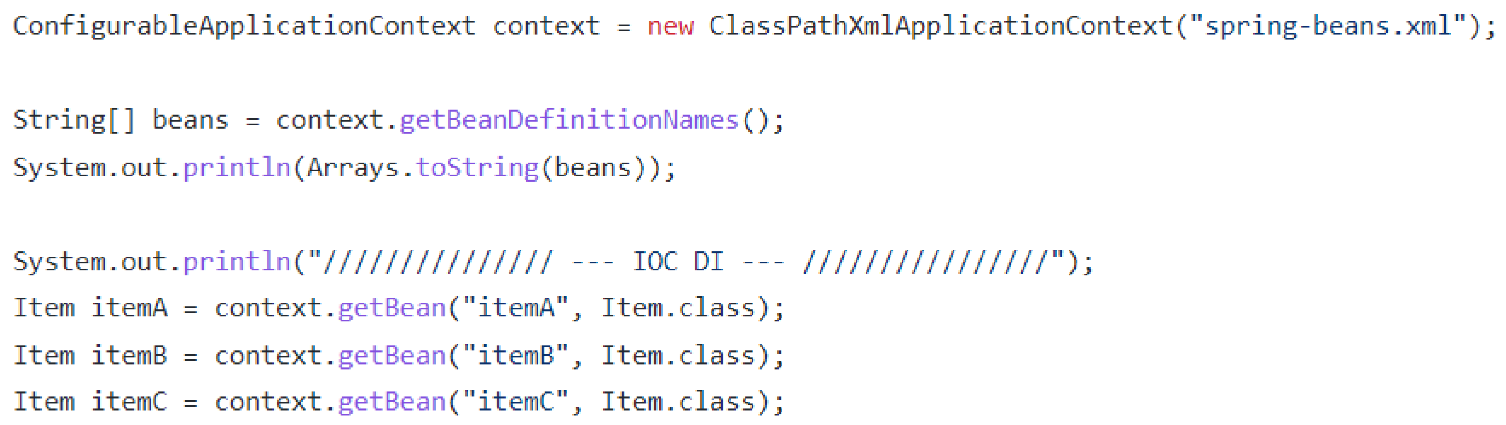
- Step 3: Create pojo classes
  - Item, ItemGroup
  - https://github.com/j4tdn/java10-repository/blob/workspace-qphan-springframe
    xml/src/main/java/com/spring/bean/Item.java
  - https://github.com/j4tdn/java10-repository/blob/workspace-qphan-springframe
    xml/src/main/java/com/spring/bean/ItemGroup.java

- Step 4: Configuration metadata
  - Configure basic Item, ItemGroup → constructor, getter, setter
  - https://github.com/j4tdn/java10-repository/blob/workspace-qphan-springframe
    xml/src/main/resources/spring-beans.xml



App Demo – And - Overview

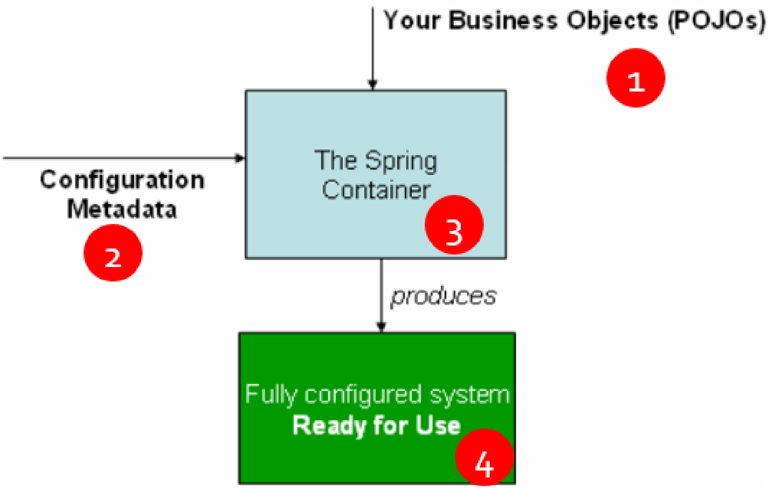
- Step 5: Construct Spring IOC container from configuration metadata and Ready for Use
  - https://github.com/j4tdn/java10-repository/blob/workspace-qphan-springframework/01-spring-core-with-xml/src/main/java/com/spring/demo/App.java



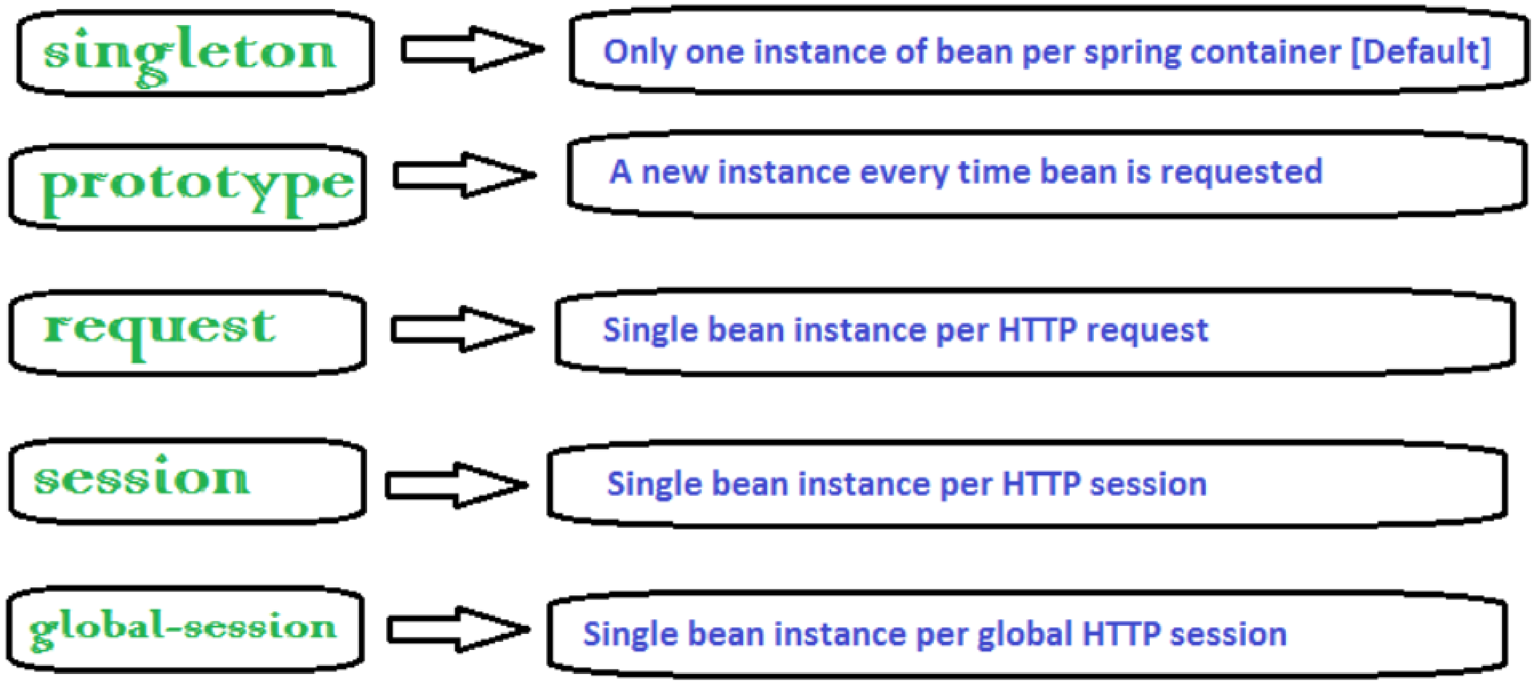
```
ConfigurableApplicationContext context = new ClassPathXmlApplicationContext("spring-beans.xml");

String[] beans = context.getBeanDefinitionNames();
System.out.println(Arrays.toString(beans));

System.out.println("/////////////// --- IOC DI --- ////////////////");
Item itemA = context.getBean("itemA", Item.class);
Item itemB = context.getBean("itemB", Item.class);
Item itemC = context.getBean("itemC", Item.class);
```

- Step 6: Configure a bean via factory-method
  - https://github.com/j4tdn/java10-repository/blob/workspace-qphan-springframe
    xml/src/main/resources/bean-overview.xml
  - ClientService#getInstance

- Step 7: Bean lifecycle and scope
  - https://github.com/j4tdn/java10-repository/blob/workspace-qphan-springframe
    xml/src/main/resources/bean-scope-lifecycle.xml
  - ItemService → ItemServiceImpl



Spring Bean Scopes



singleton → Only one instance of bean per spring container [Default]

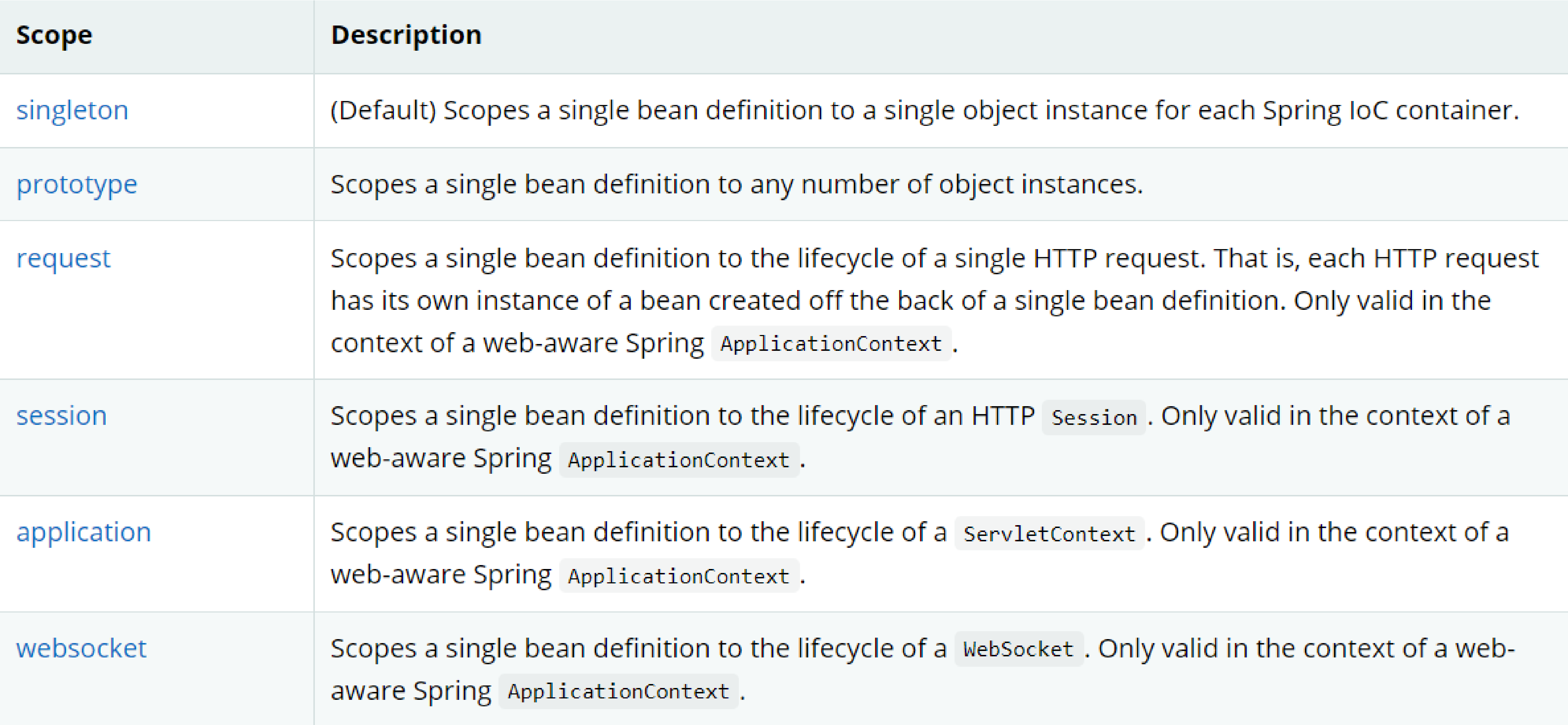
prototype → A new instance every time bean is requested

request → Single bean instance per HTTP request

session → Single bean instance per HTTP session

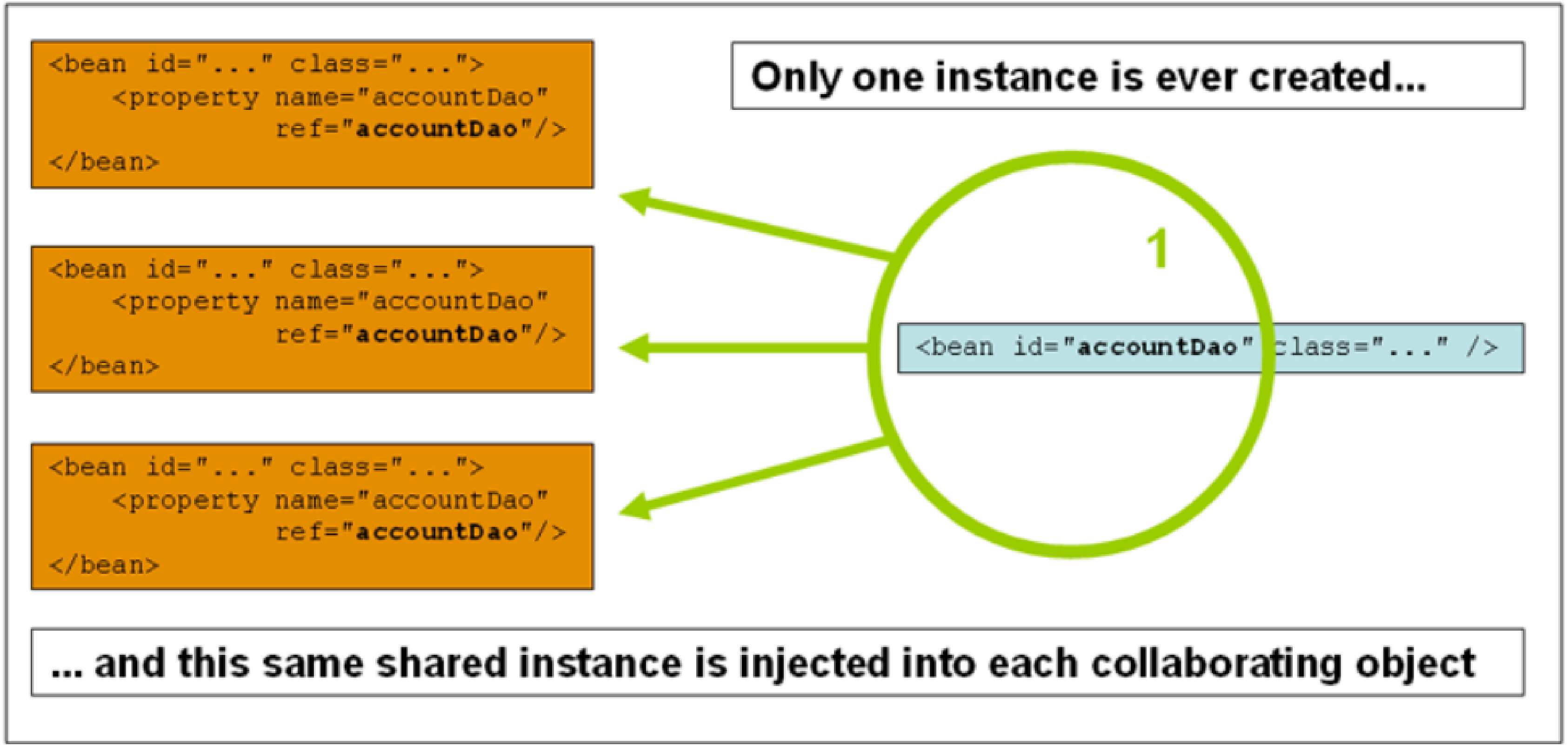
global-session → Single bean instance per global HTTP session

Beans Scope



Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance for each Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
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` .
session	Scopes a single bean definition to the lifecycle of an HTTP `Session` . Only valid in the context of a web-aware Spring `ApplicationContext` .
application	Scopes a single bean definition to the lifecycle of a `ServletContext` . Only valid in the context of a web-aware Spring `ApplicationContext` .
websocket	Scopes a single bean definition to the lifecycle of a `WebSocket` . Only valid in the context of a web-aware Spring `ApplicationContext` .

Spring bean scope - Singleton



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

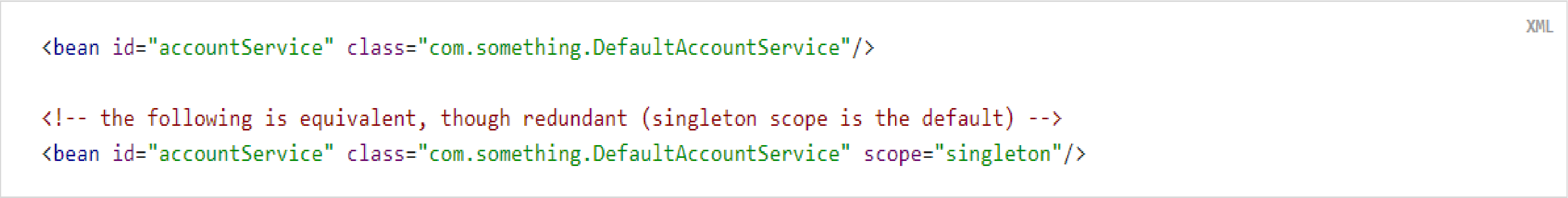
```
<bean id="..." class="...">
    <property name="accountDao"
              ref="accountDao"/>
</bean>
```

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

Only one instance is ever created...

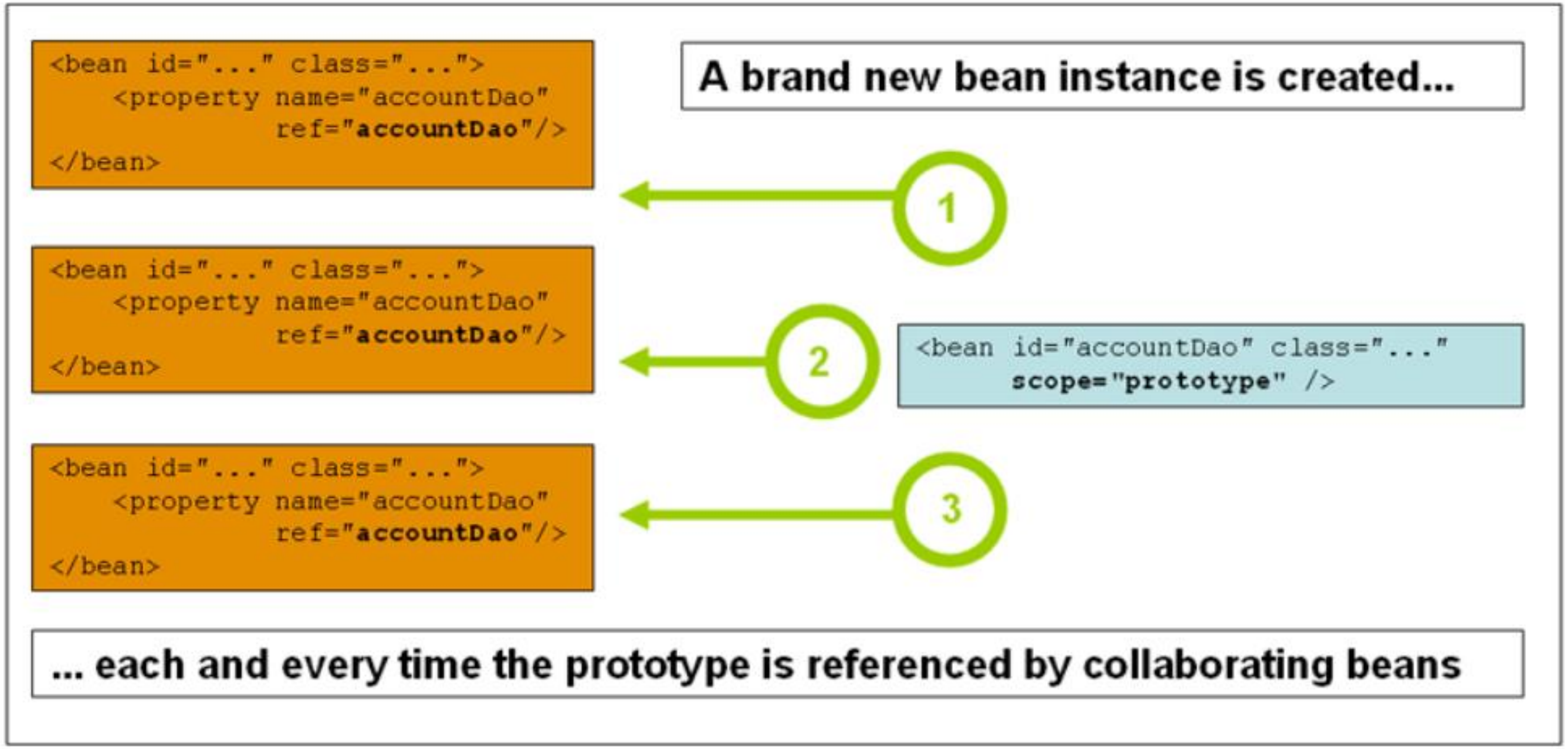
```
<bean id="accountDao" class="..." />
```

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



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

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

Spring bean scope - Prototype



A brand new bean instance is created...

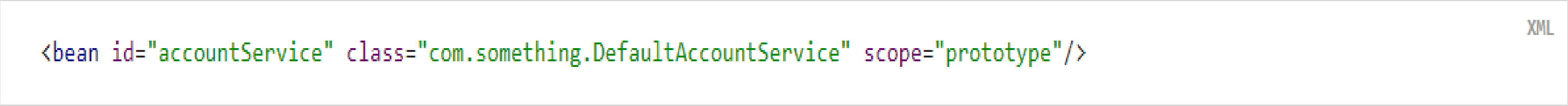
```
<bean id="..." class="...">
    <property name="accountDao"
              ref="accountDao"/>
</bean>
```

```
<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



```xml
<bean id="accountService" class="com.something.DefaultAccountService" scope="prototype"/>
```

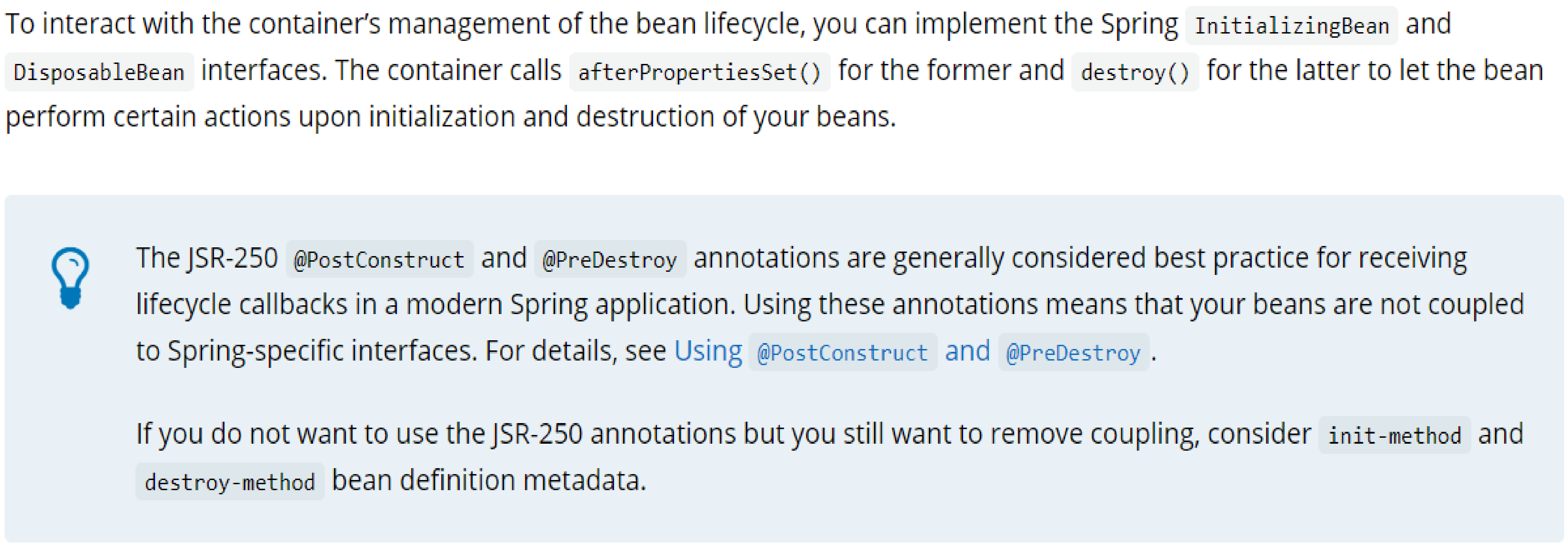
Spring bean scope - Others

- https://docs.spring.io/spring-framework/docs/3.0.0.M3/reference/html/ch04s04.html

- Initial web configuration

Customizing the Nature of a Bean

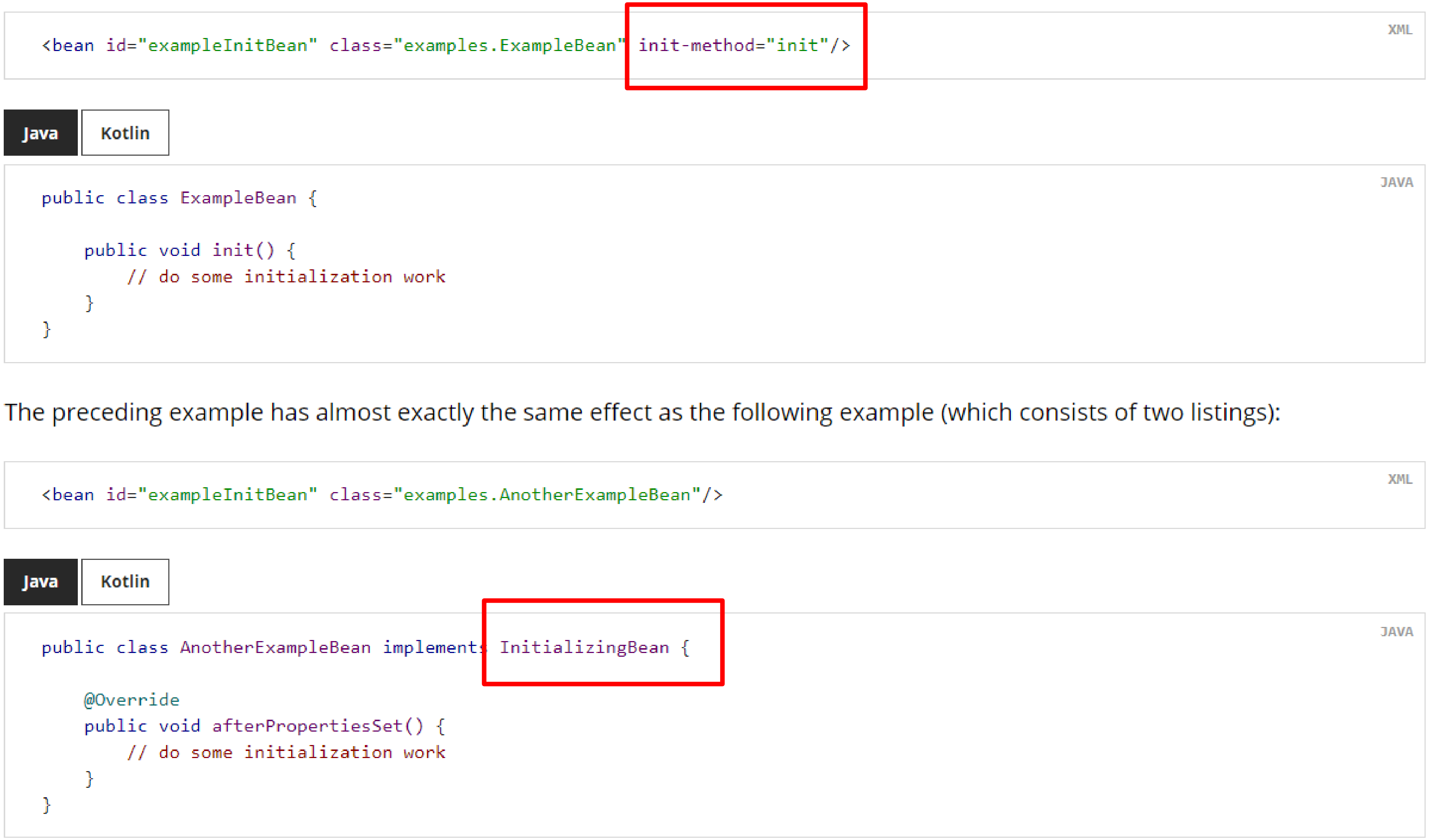
Spring bean – Life cycle and callback



To interact with the container's management of the bean lifecycle, you can implement the Spring `InitializingBean` and `DisposableBean` interfaces. The container calls `afterPropertiesSet()` for the former and `destroy()` for the latter to let the bean perform certain actions upon initialization and destruction of your beans.

> 💡 The JSR-250 `@PostConstruct` and `@PreDestroy` annotations are generally considered best practice for receiving lifecycle callbacks in a modern Spring application. Using these annotations means that your beans are not coupled to Spring-specific interfaces. For details, see Using `@PostConstruct` and `@PreDestroy`.
>
> If you do not want to use the JSR-250 annotations but you still want to remove coupling, consider `init-method` and `destroy-method` bean definition metadata.

Life cycle and callback - Initial



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

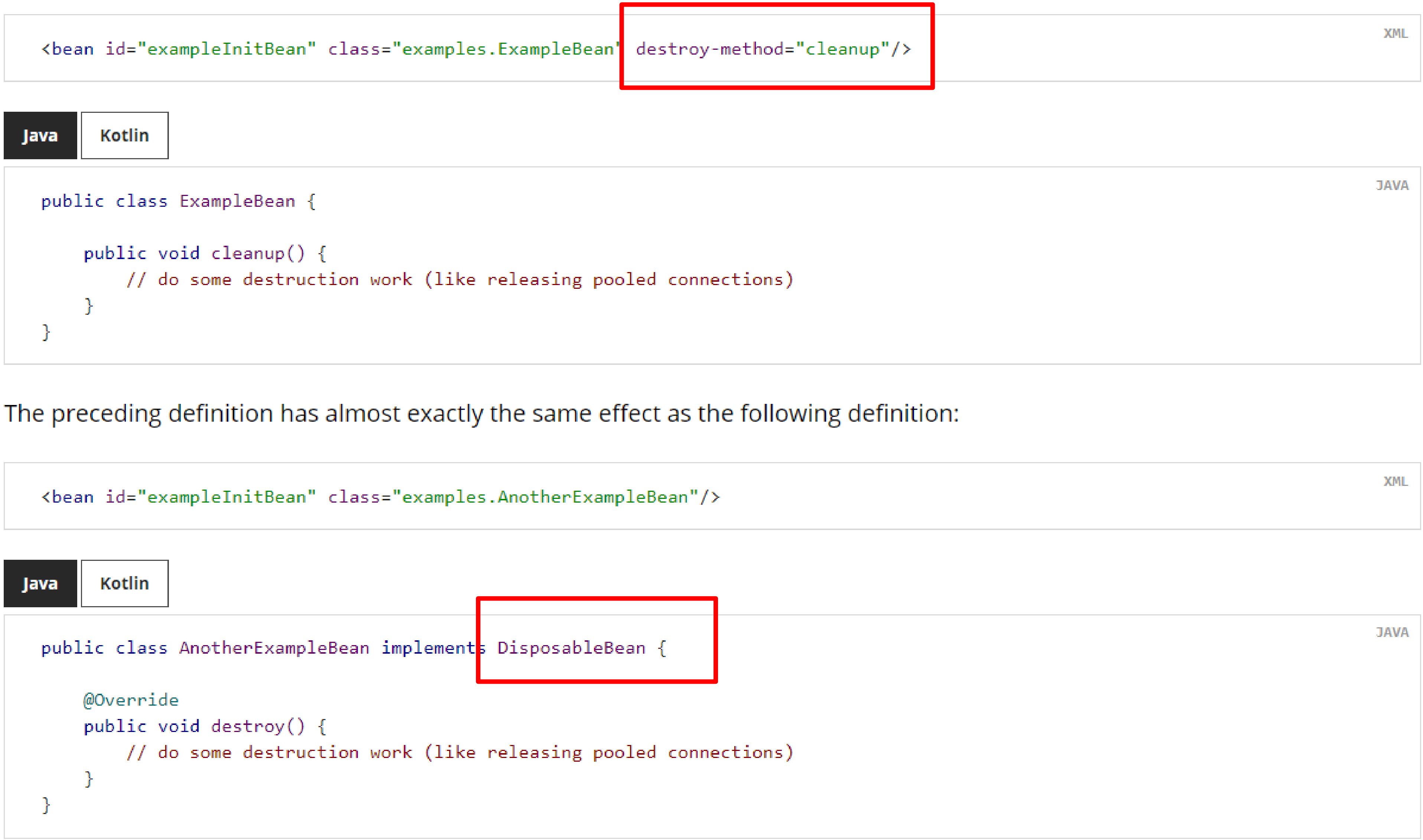
```java
public class ExampleBean {

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

The preceding example has almost exactly the same effect as the following example (which consists of two listings):

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

```java
public class AnotherExampleBean implements InitializingBean {

    @Override
    public void afterPropertiesSet() {
        // do some initialization work
    }
}
```

Life cycle and callback - Destroy



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

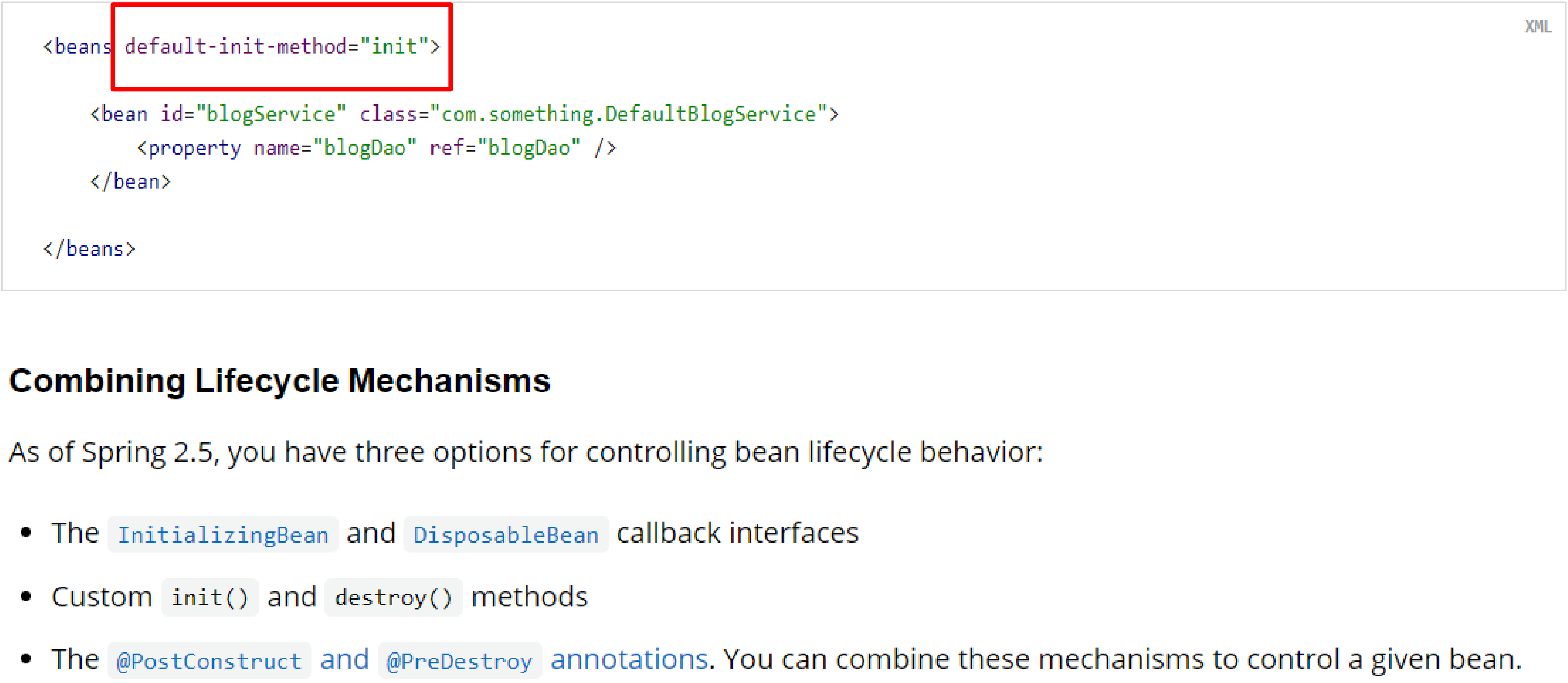
```java
public class ExampleBean {

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

The preceding definition has almost exactly the same effect as the following definition:

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

```java
public class AnotherExampleBean implements DisposableBean {

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

Life cycle and callback – Default initial destroy method



XML

```
<beans default-init-method="init">

    <bean id="blogService" class="com.something.DefaultBlogService">
        <property name="blogDao" ref="blogDao" />
    </bean>

</beans>
```

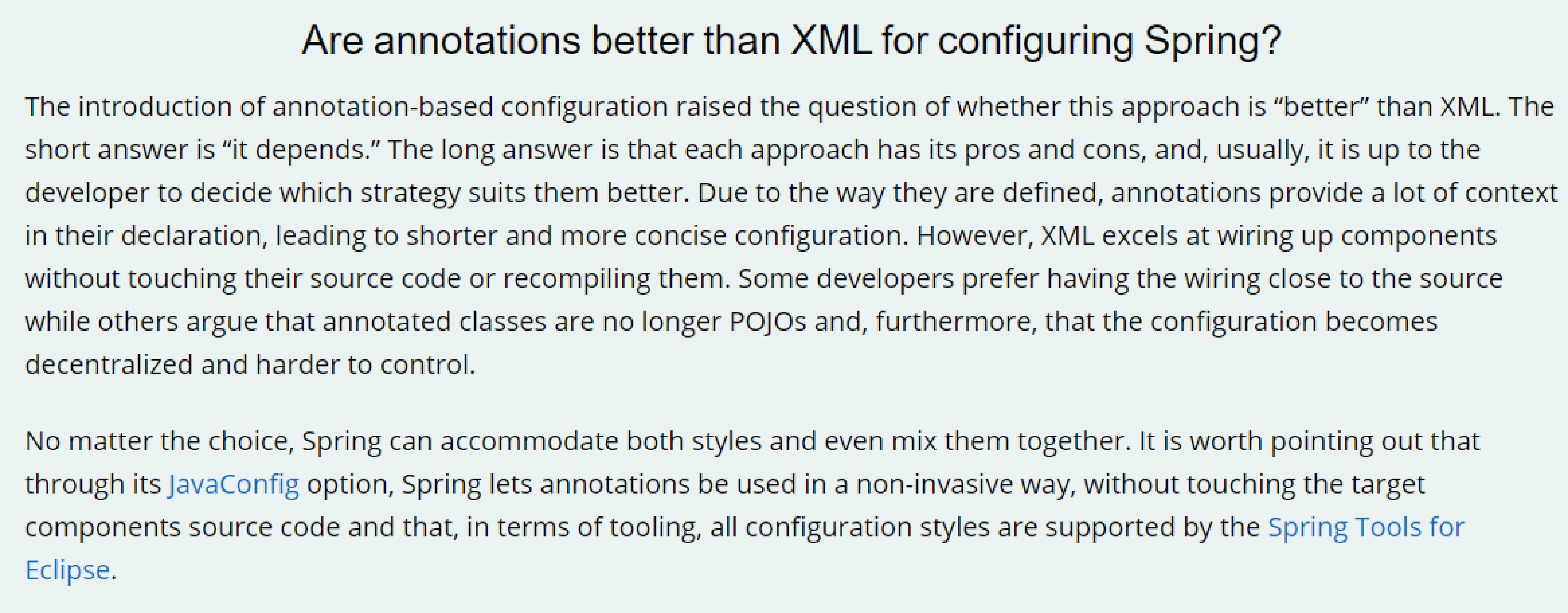
Combining Lifecycle Mechanisms

As of Spring 2.5, you have three options for controlling bean lifecycle behavior:

- The `InitializingBean` and `DisposableBean` callback interfaces
- Custom `init()` and `destroy()` methods
- The `@PostConstruct` and `@PreDestroy` annotations. You can combine these mechanisms to control a given bean.

Container Configuration With ANNOTATION

XML vs Annotation

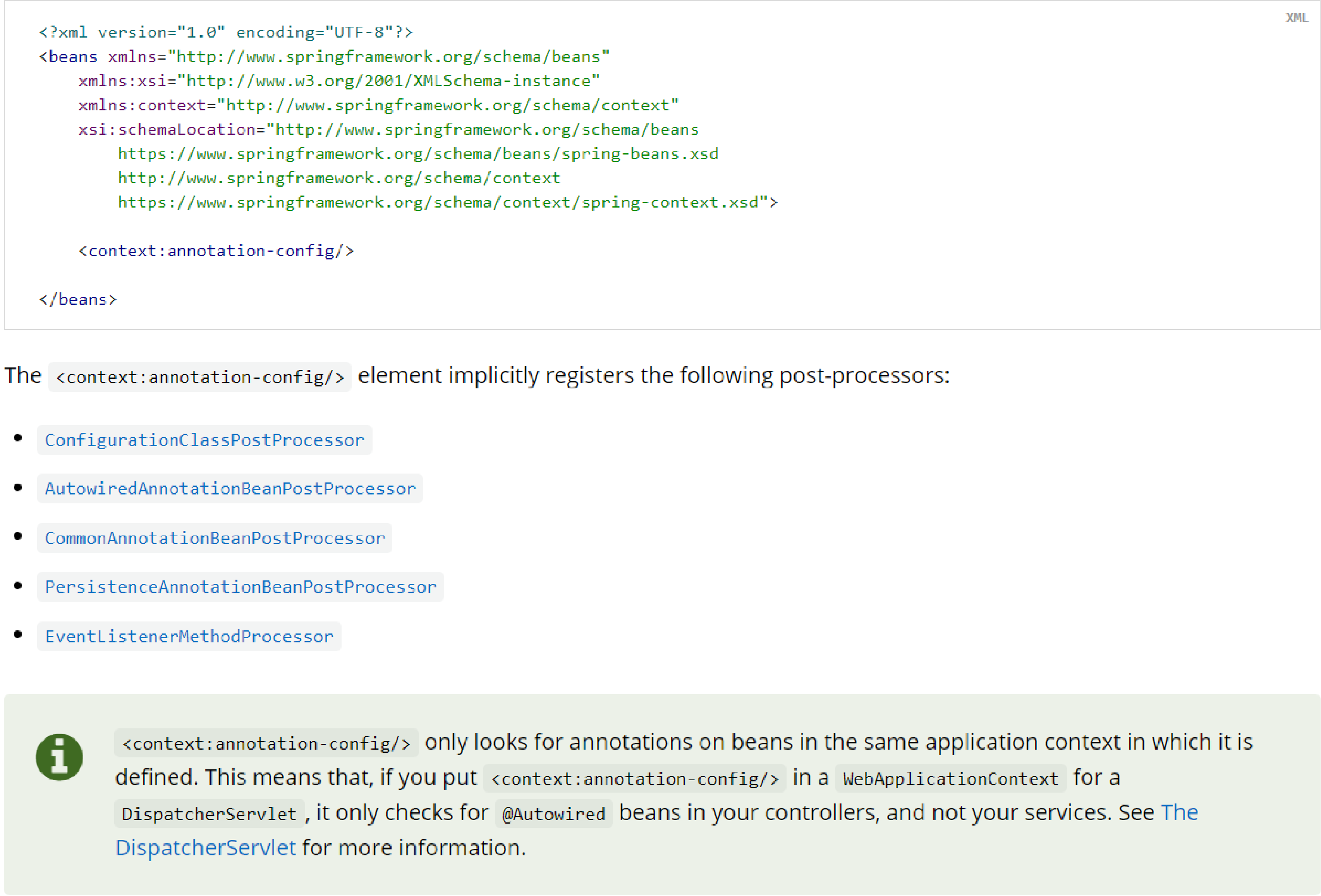


Are annotations better than XML for configuring Spring?

The introduction of annotation-based configuration raised the question of whether this approach is “better” than XML. The short answer is “it depends.” The long answer is that each approach has its pros and cons, and, usually, it is up to the developer to decide which strategy suits them better. Due to the way they are defined, annotations provide a lot of context in their declaration, leading to shorter and more concise configuration. However, XML excels at wiring up components without touching their source code or recompiling them. Some developers prefer having the wiring close to the source while others argue that annotated classes are no longer POJOs and, furthermore, that the configuration becomes decentralized and harder to control.

No matter the choice, Spring can accommodate both styles and even mix them together. It is worth pointing out that through its JavaConfig option, Spring lets annotations be used in a non-invasive way, without touching the target components source code and that, in terms of tooling, all configuration styles are supported by the Spring Tools for Eclipse.

Introduction



XML

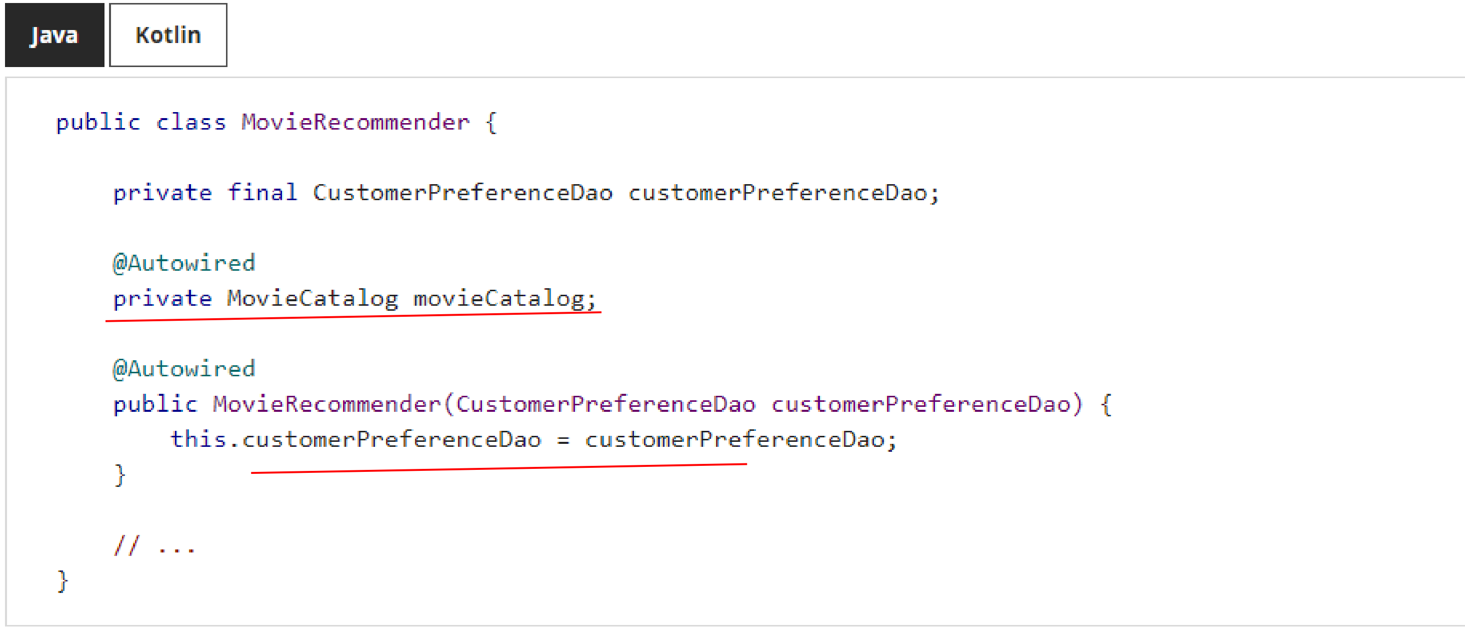
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      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

    <context:annotation-config/>

</beans>
```

The `<context:annotation-config/>` element implicitly registers the following post-processors:

- `ConfigurationClassPostProcessor`
- `AutowiredAnnotationBeanPostProcessor`
- `CommonAnnotationBeanPostProcessor`
- `PersistenceAnnotationBeanPostProcessor`
- `EventListenerMethodProcessor`

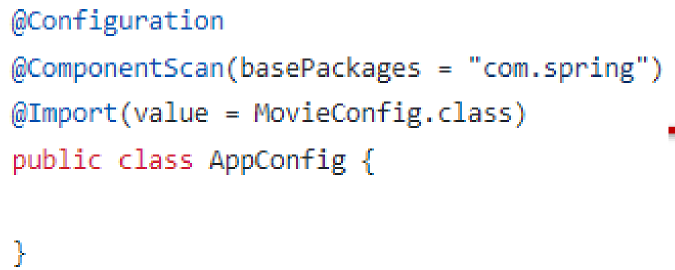
`<context:annotation-config/>` only looks for annotations on beans in the same application context in which it is defined. This means that, if you put `<context:annotation-config/>` in a `WebApplicationContext` for a `DispatcherServlet`, it only checks for `@Autowired` beans in your controllers, and not your services. See The DispatcherServlet for more information.

Autowired

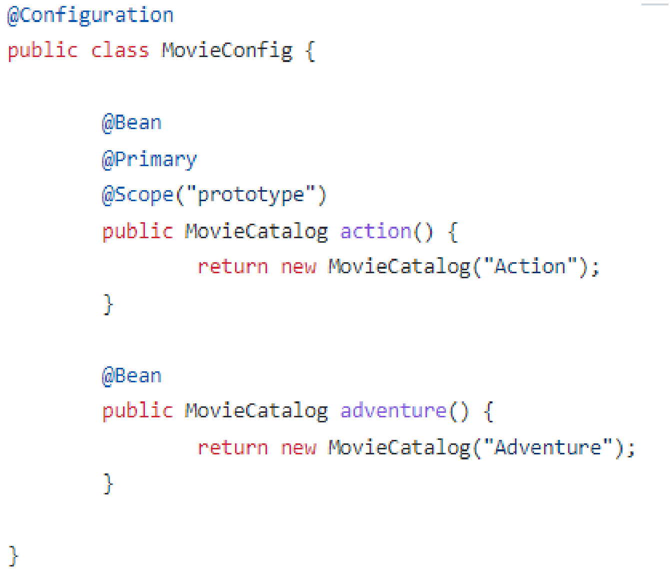


Java Kotlin

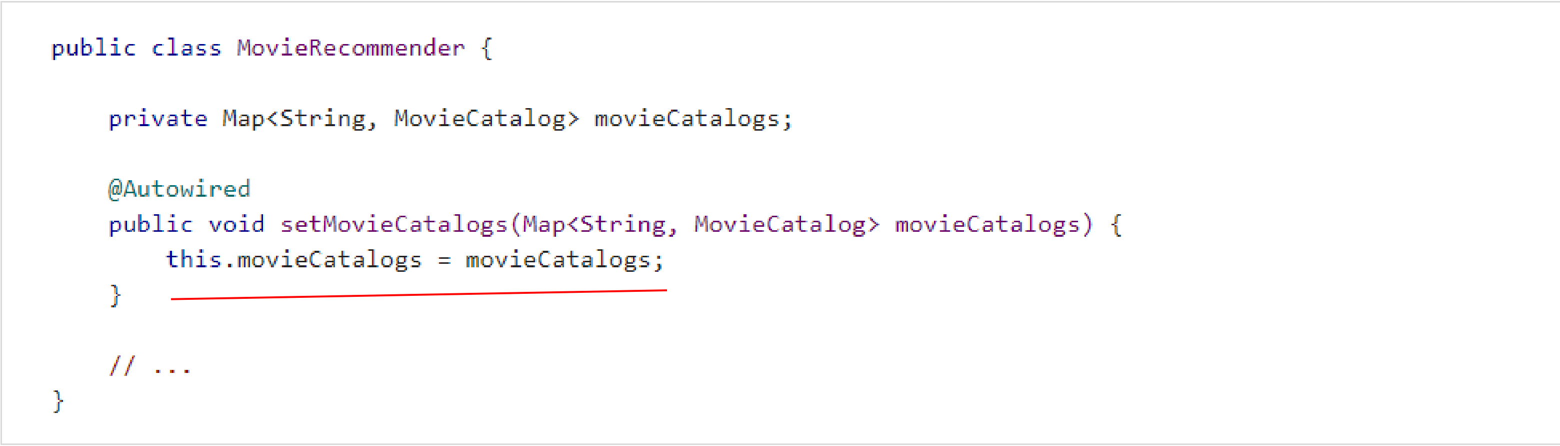
```java
public class MovieRecommender {

    private final CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    private MovieCatalog movieCatalog;

    @Autowired
    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {
        this.customerPreferenceDao = customerPreferenceDao;
    }

    // ...
}
```



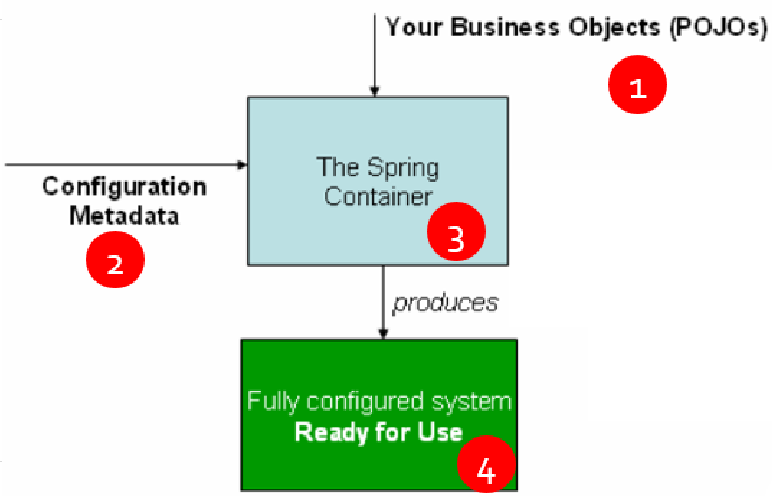
```java
@Configuration
@ComponentScan(basePackages = "com.spring")
@Import(value = MovieConfig.class)
public class AppConfig {

}
```



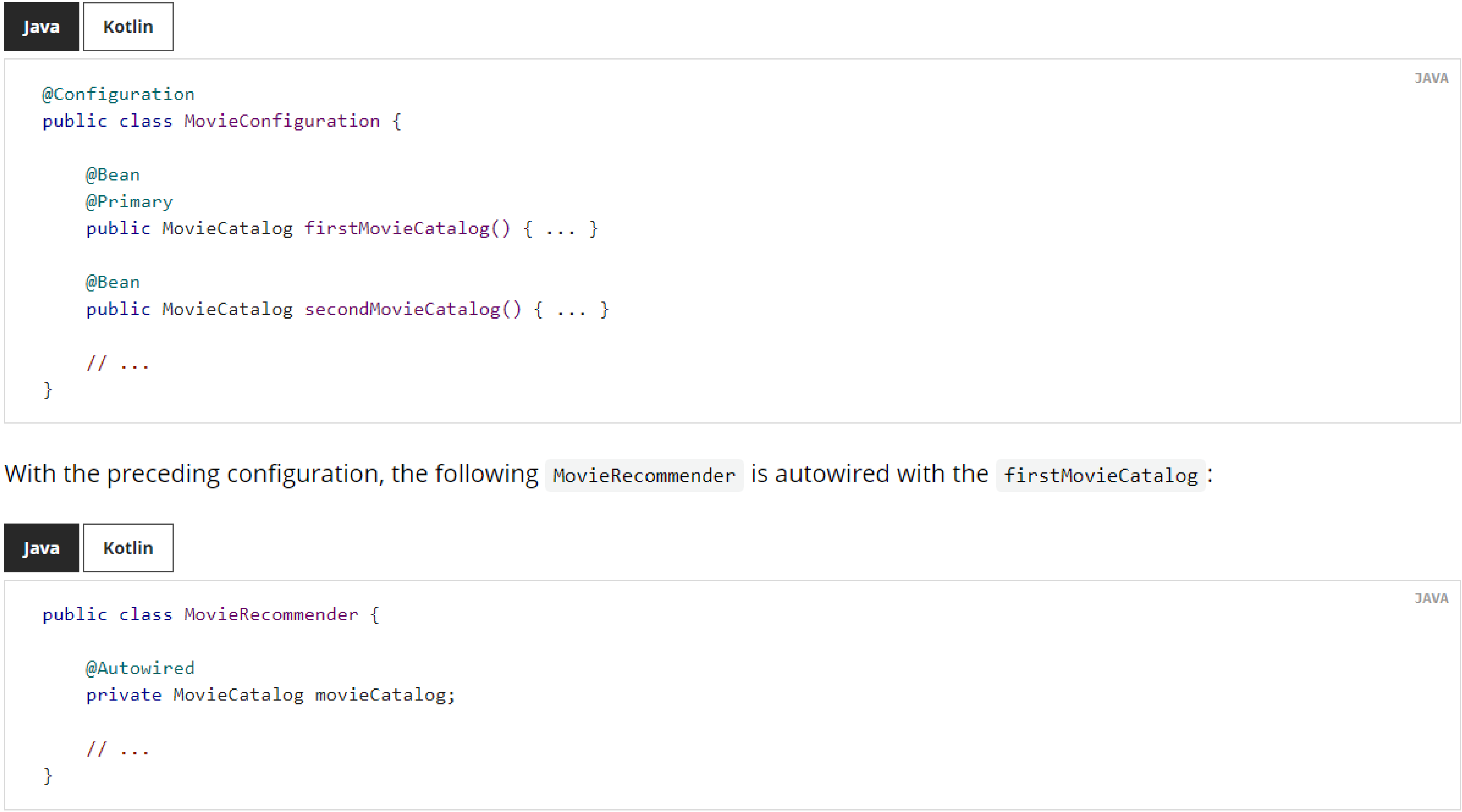
```java
@Configuration
public class MovieConfig {

        @Bean
        @Primary
        @Scope("prototype")
        public MovieCatalog action() {
                return new MovieCatalog("Action");
        }

        @Bean
        public MovieCatalog adventure() {
                return new MovieCatalog("Adventure");
        }

}
```



```java
public class MovieRecommender {

    private Map<String, MovieCatalog> movieCatalogs;

    @Autowired
    public void setMovieCatalogs(Map<String, MovieCatalog> movieCatalogs) {
        this.movieCatalogs = movieCatalogs;
    }

    // ...
}
```



Autowired & Primary

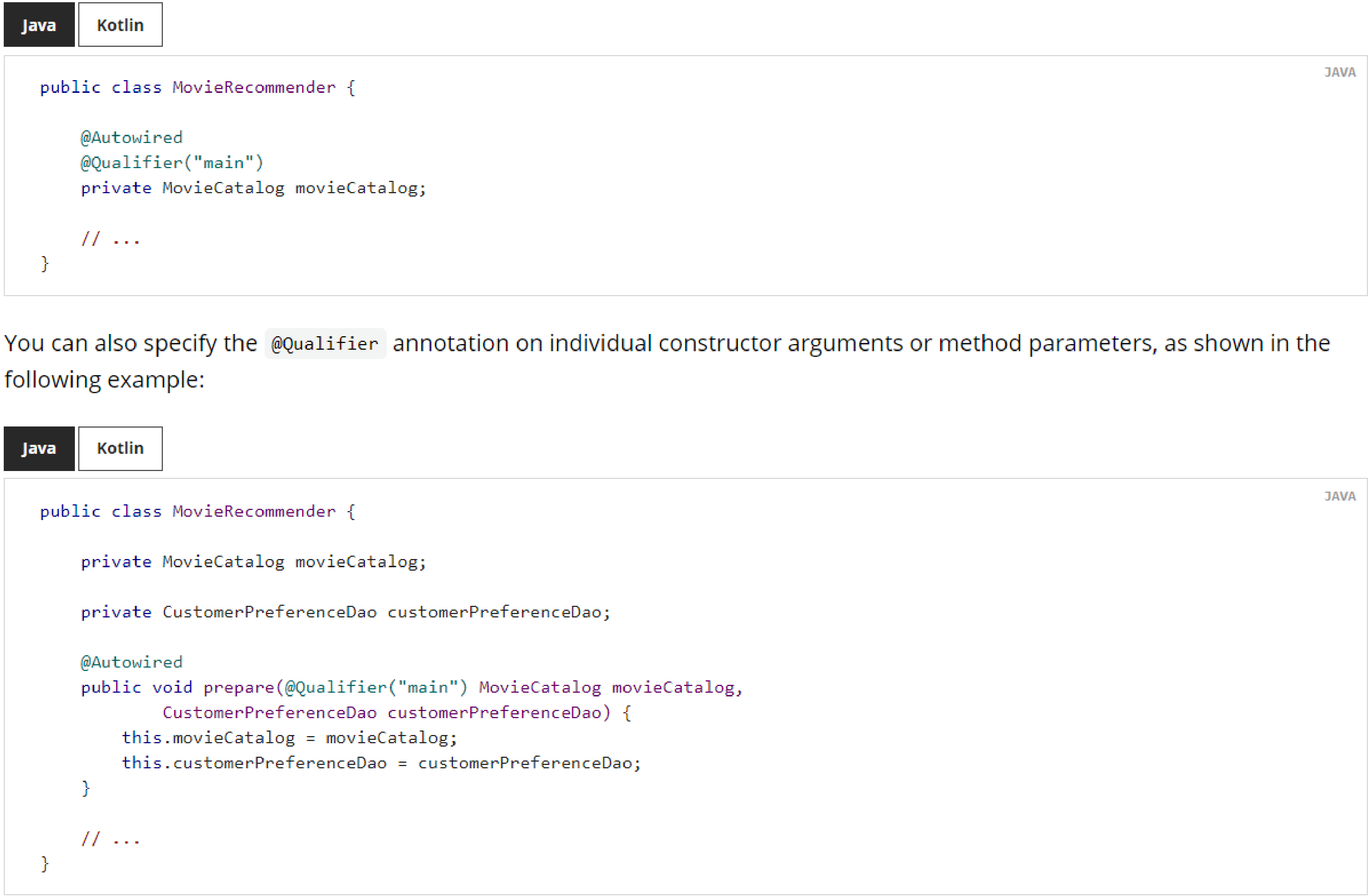


```java
@Configuration
public class MovieConfiguration {

    @Bean
    @Primary
    public MovieCatalog firstMovieCatalog() { ... }

    @Bean
    public MovieCatalog secondMovieCatalog() { ... }

    // ...
}
```

With the preceding configuration, the following `MovieRecommender` is autowired with the `firstMovieCatalog`:

```java
public class MovieRecommender {

    @Autowired
    private MovieCatalog movieCatalog;

    // ...
}
```

Autowired & Qualifier

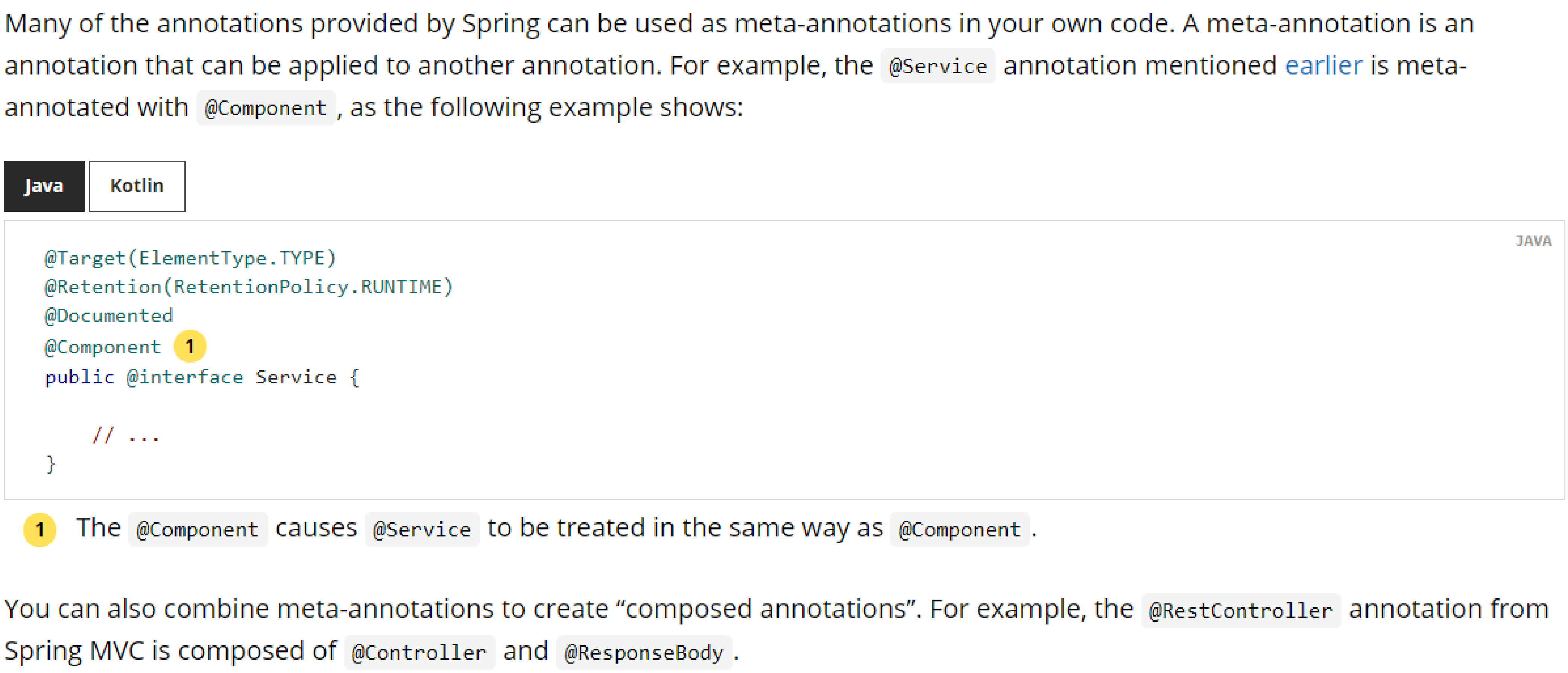


```java
public class MovieRecommender {

    @Autowired
    @Qualifier("main")
    private MovieCatalog movieCatalog;

    // ...
}
```

You can also specify the `@Qualifier` annotation on individual constructor arguments or method parameters, as shown in the following example:

```java
public class MovieRecommender {

    private MovieCatalog movieCatalog;

    private CustomerPreferenceDao customerPreferenceDao;

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

    // ...
}
```

Component and stereotype(alias) annotations



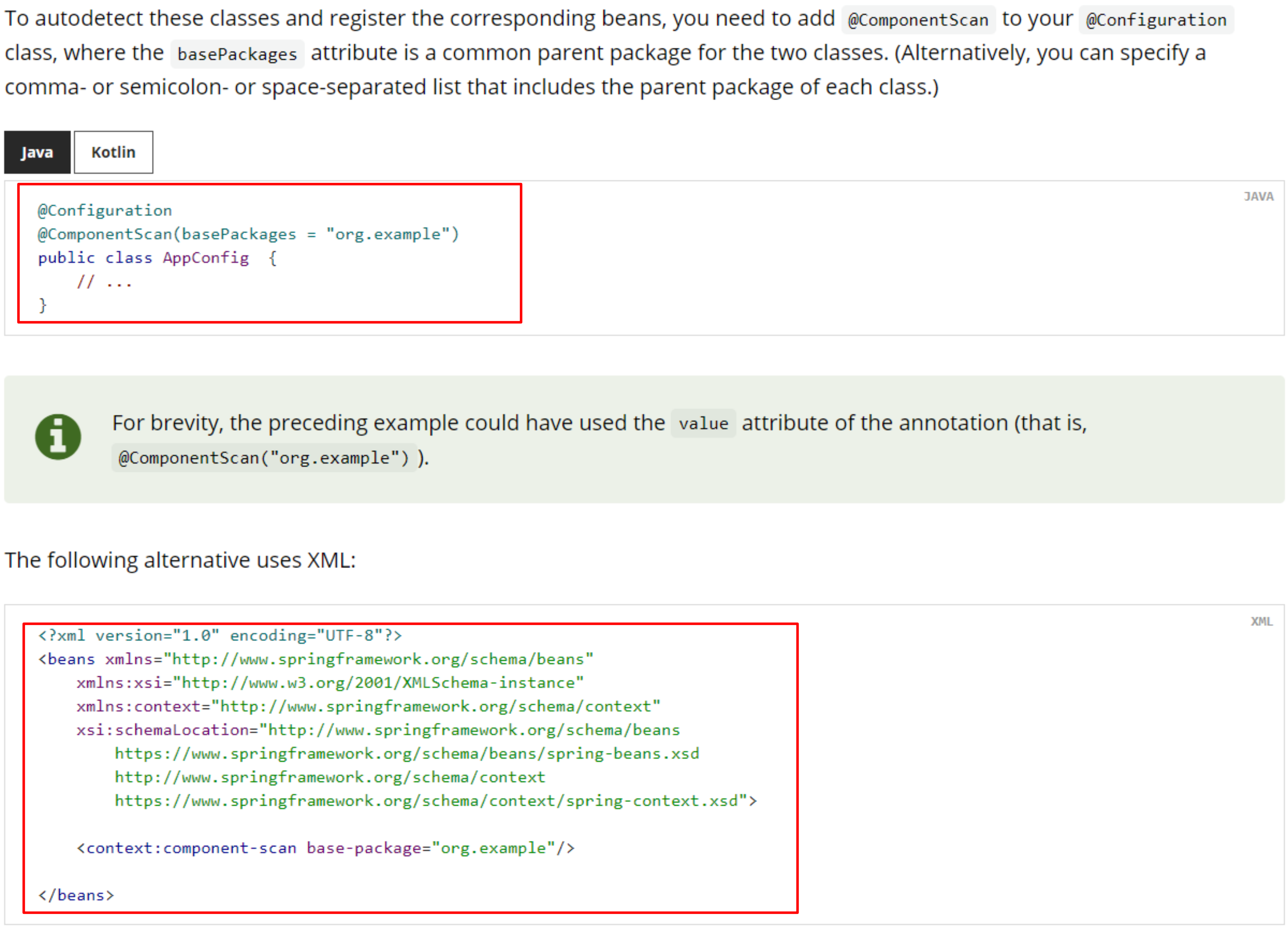
Many of the annotations provided by Spring can be used as meta-annotations in your own code. A meta-annotation is an annotation that can be applied to another annotation. For example, the `@Service` annotation mentioned earlier is meta-annotated with `@Component`, as the following example shows:

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component 1
public @interface Service {

    // ...
}
```

1 The `@Component` causes `@Service` to be treated in the same way as `@Component`.

You can also combine meta-annotations to create “composed annotations”. For example, the `@RestController` annotation from Spring MVC is composed of `@Controller` and `@ResponseBody`.

Component and stereotype(alias) annotations



To autodetect these classes and register the corresponding beans, you need to add `@ComponentScan` to your `@Configuration` class, where the `basePackages` attribute is a common parent package for the two classes. (Alternatively, you can specify a comma- or semicolon- or space-separated list that includes the parent package of each class.)

```java
@Configuration
@ComponentScan(basePackages = "org.example")
public class AppConfig  {
    // ...
}
```

For brevity, the preceding example could have used the `value` attribute of the annotation (that is, `@ComponentScan("org.example")`).

The following alternative uses XML:

```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:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        https://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="org.example"/>

</beans>
```

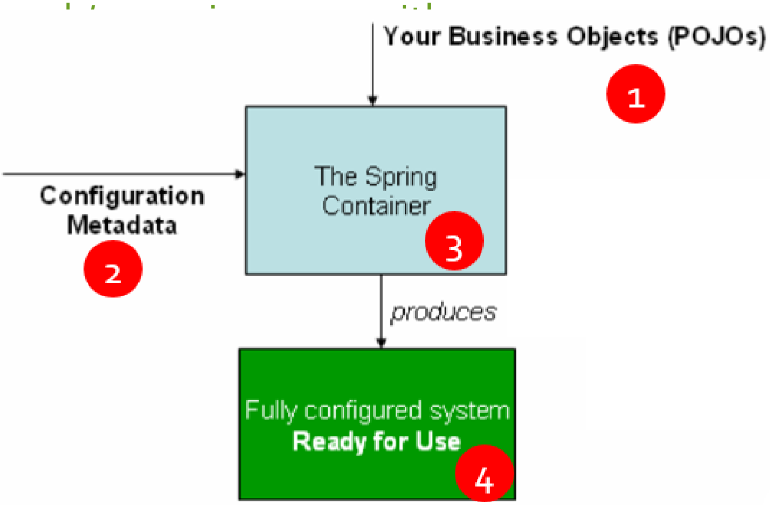
App Demo – And - Overview

- Step 1: Create a maven core project with maven-archetype-webapp
  - Name: 02-spring-core-ano

- Step 2: Import dependencies
  - https://mvnrepository.com/artifact/org.springframework/spring-core
  - https://mvnrepository.com/artifact/org.springframework/spring-context
  - https://github.com/j4tdn/java89-repository/blob/workspace-qphan-springfw/03-spring-mvc-customer-app-with-annotation/pom.xml

- Step 3: Create pojo classes
  - MovieCatalog, MovieRecomender
  - https://github.com/j4tdn/java10-repository/blob/workspace-qphan-springframe
    annotation/src/main/java/com/spring/bean/MovieCatalog.java
  - https://github.com/j4tdn/java10-repository/blob/workspace-qphan-springframe
    annotation/src/main/java/com/spring/bean/MovieRecommender.java

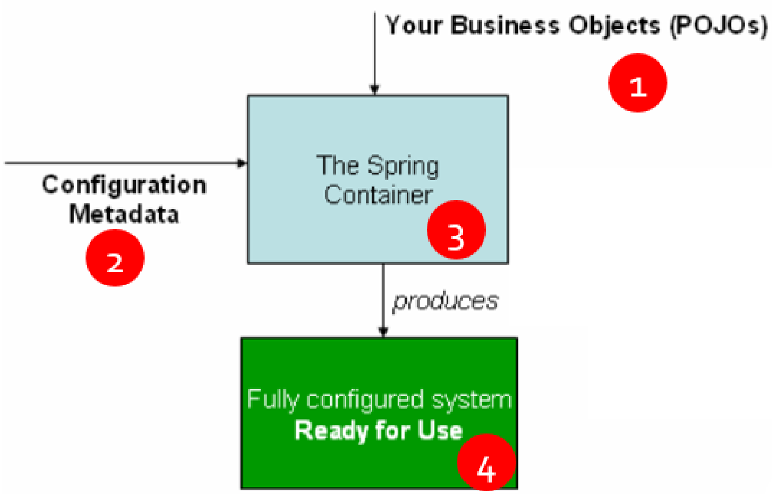
- Step 4: Configuration metadata
  - Configure basic Item, ItemGroup → constructor, getter, setter
  - https://github.com/j4tdn/java10-repository/blob/workspace-qphan-springframe
    annotation/src/main/java/com/spring/config/AppConfig.java



App Demo – And - Overview

- Step 5: Construct Spring IOC container from configuration metadata and Ready for Use

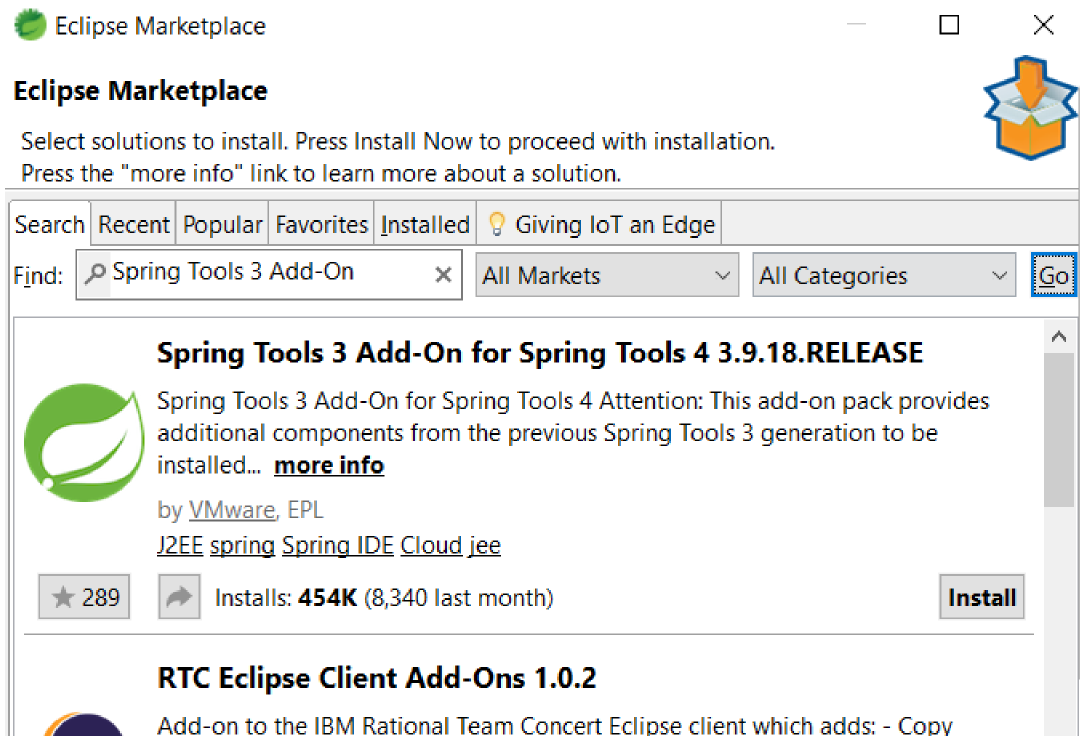
    - https://github.com/j4tdn/java10-repository/blob/workspace-qphan-springframework/02-spring-core-with-annotation/src/main/java/com/spring/demo/App.java


Your Business Objects (POJOs)
1
Configuration Metadata
2
The Spring Container
3
produces
Fully configured system Ready for Use
4

Configuration and Annotations

- @Component @Controller @Service @Repository
- @Configuration
- @Import
- @Bean
- @Autowired
- @Qualifier
- @Scope
- ApplicationContext, ConfigurableApplicationContext
- ClassPathXmlApplicationContext, AnnotationConfigApplicationContext

Plugin for managing Spring bean in Eclipse



Eclipse Marketplace

Select solutions to install. Press Install Now to proceed with installation.
Press the "more info" link to learn more about a solution.

Find: Spring Tools 3 Add-On

Spring Tools 3 Add-On for Spring Tools 4 3.9.18.RELEASE

Spring Tools 3 Add-On for Spring Tools 4 Attention: This add-on pack provides additional components from the previous Spring Tools 3 generation to be installed... **more info**

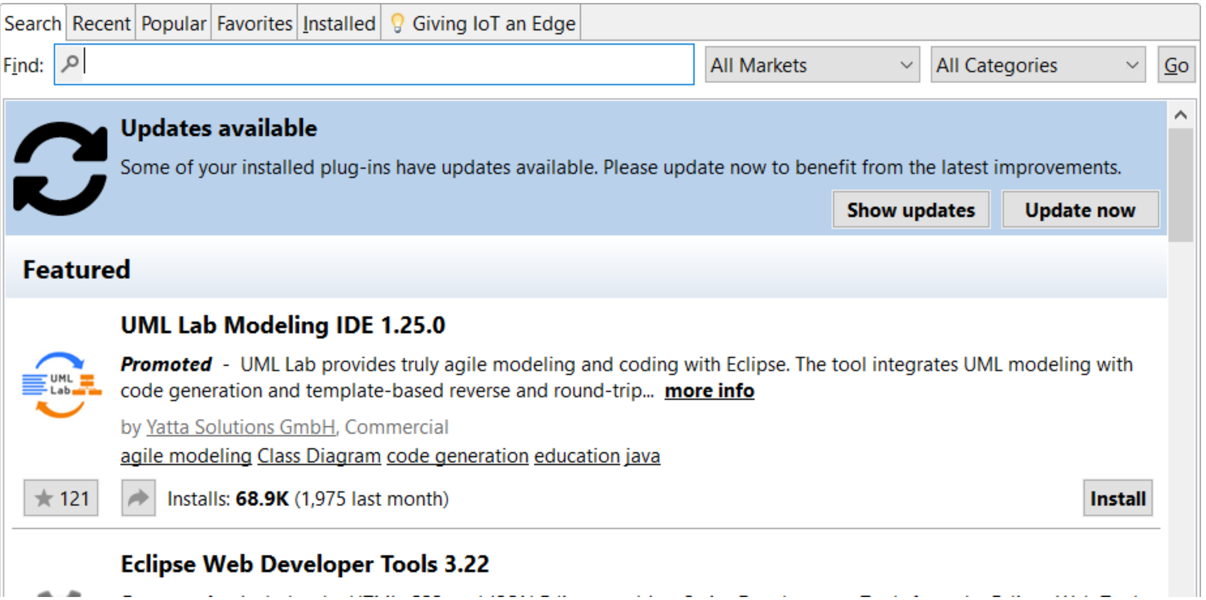
by VMware, EPL

J2EE spring Spring IDE Cloud jee

★ 289 Installs: **454K** (8,340 last month)

RTC Eclipse Client Add-Ons 1.0.2

Add-on to the IBM Rational Team Concert Eclipse client which adds: - Copy



Updates available

Some of your installed plug-ins have updates available. Please update now to benefit from the latest improvements.

Featured

UML Lab Modeling IDE 1.25.0

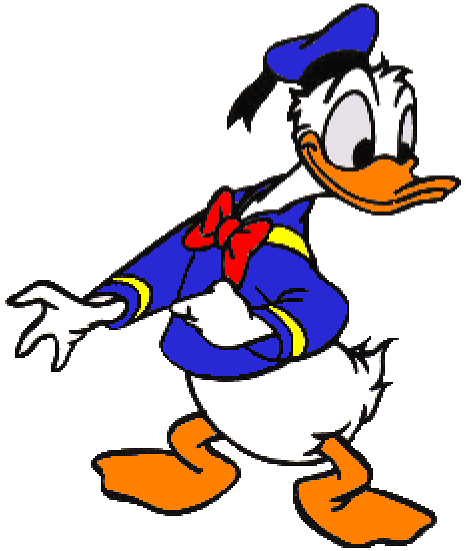
Promoted - UML Lab provides truly agile modeling and coding with Eclipse. The tool integrates UML modeling with code generation and template-based reverse and round-trip... **more info**

by Yatta Solutions GmbH, Commercial

agile modeling Class Diagram code generation education java

★ 121 Installs: **68.9K** (1,975 last month)

Eclipse Web Developer Tools 3.22



END