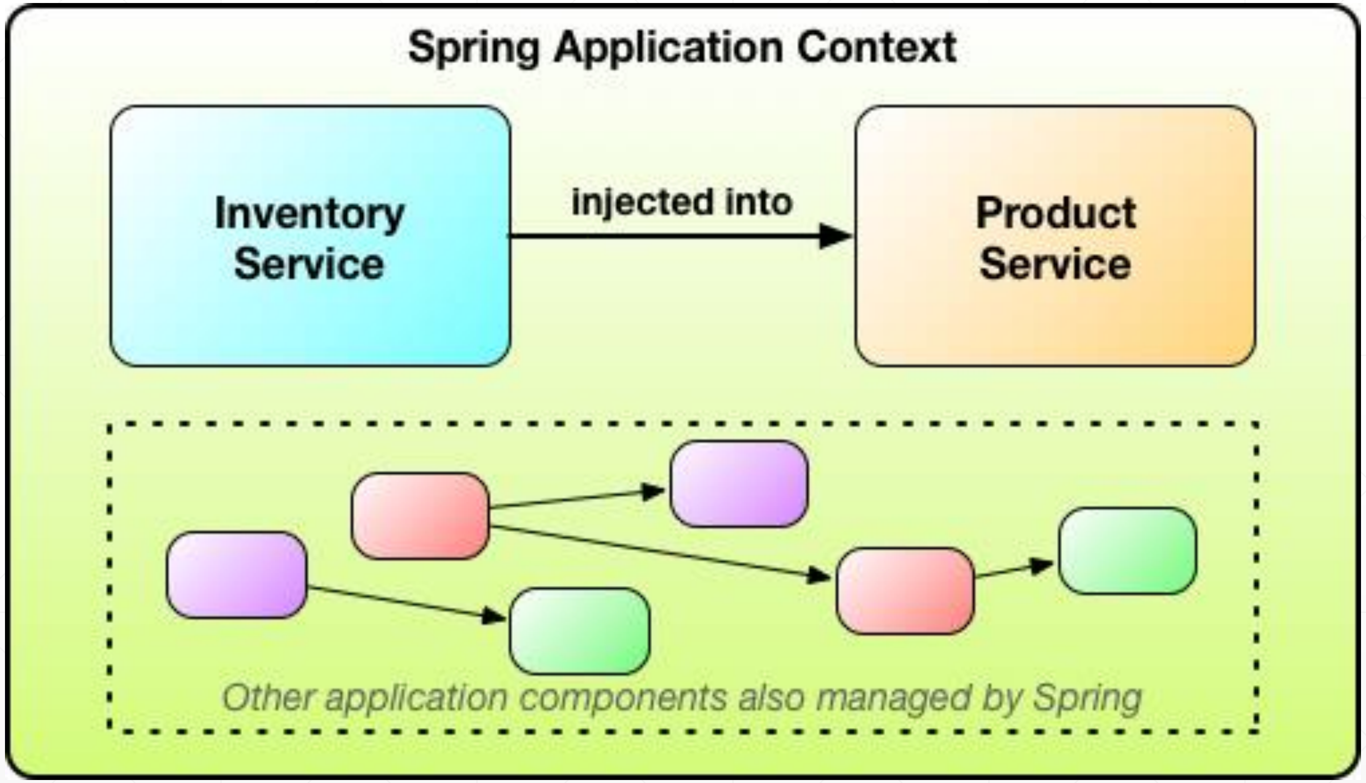
Spring Framework

1. Core Spring: Introduction


Spring Application Context
Inventory Service
injected into
Product Service
Other application components also managed by Spring

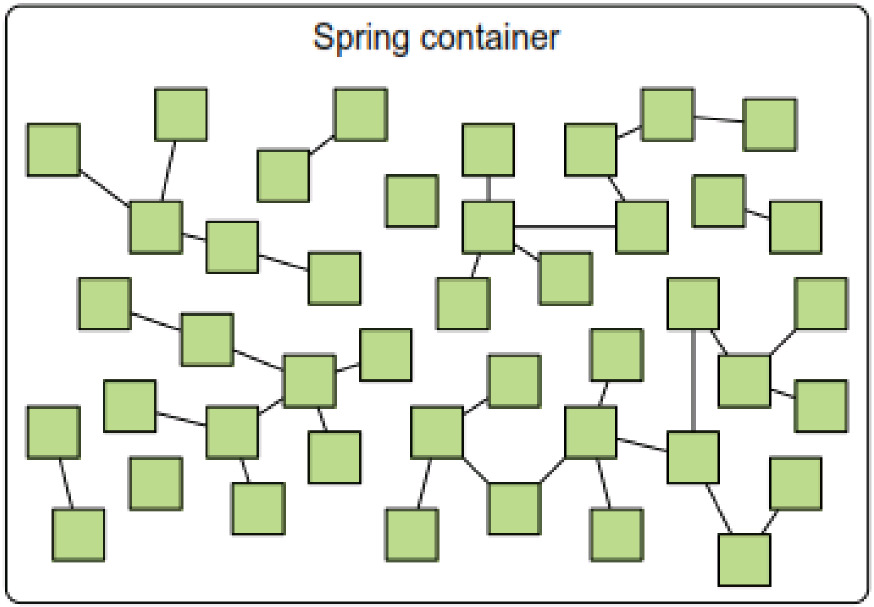
Simplifying Java Development

Spring employs four key strategies:

- Lightweight and minimally invasive development with POJOs
- Loose coupling through DI and interface orientation
- Declarative programming through aspects and common conventions
- Eliminating boilerplate code with aspects and templates

Containing your beans

- In a Spring application, objects are created, are wired together, and live in the Spring container.



Spring comes with several container implementations that can be categorized into two Distinct types:

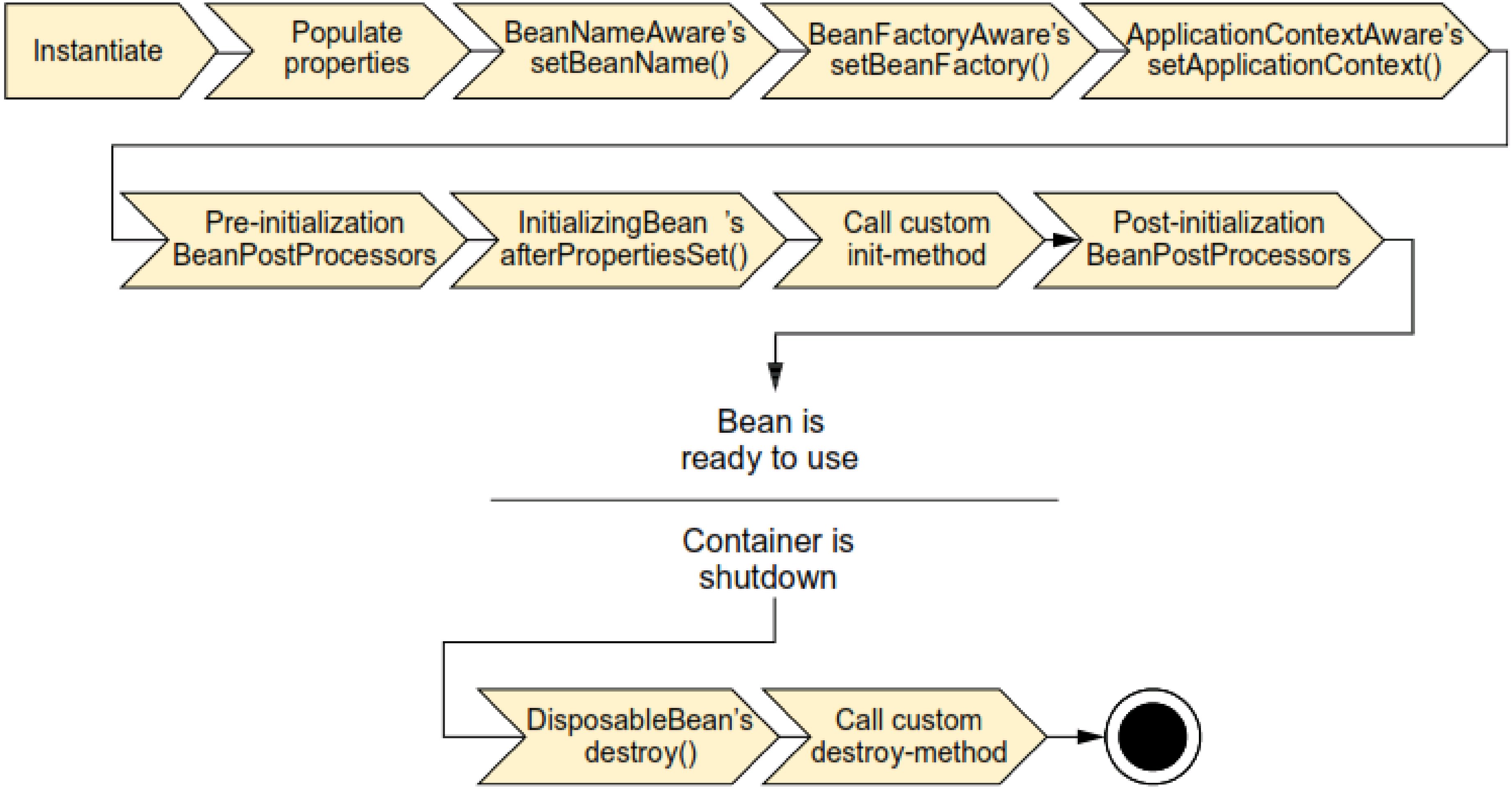
1. ***Bean factories*** (defined by the org.springframework.beans.factory.BeanFactory interface) are the simplest of containers, providing basic support for DI.
2. ***Application contexts*** *(defined by the* org.springframework.context.ApplicationContext interface) build on the notion of a bean factory by providing application-framework services

Spring Application context

- **AnnotationConfigApplicationContext**—Loads a Spring application context from one or more Java-based configuration classes
- **AnnotationConfigWebApplicationContext**—Loads a Spring web application context from one or more Java-based configuration classes
- **ClassPathXmlApplicationContex**t—Loads a context definition from one or more XML files located in the classpath, treating context-definition files as classpath resources
- **FileSystemXmlApplicationContext**—Loads a context definition from one or more XML files in the filesystem
- **XmlWebApplicationContext**—Loads context definitions from one or more XML files contained in a web application

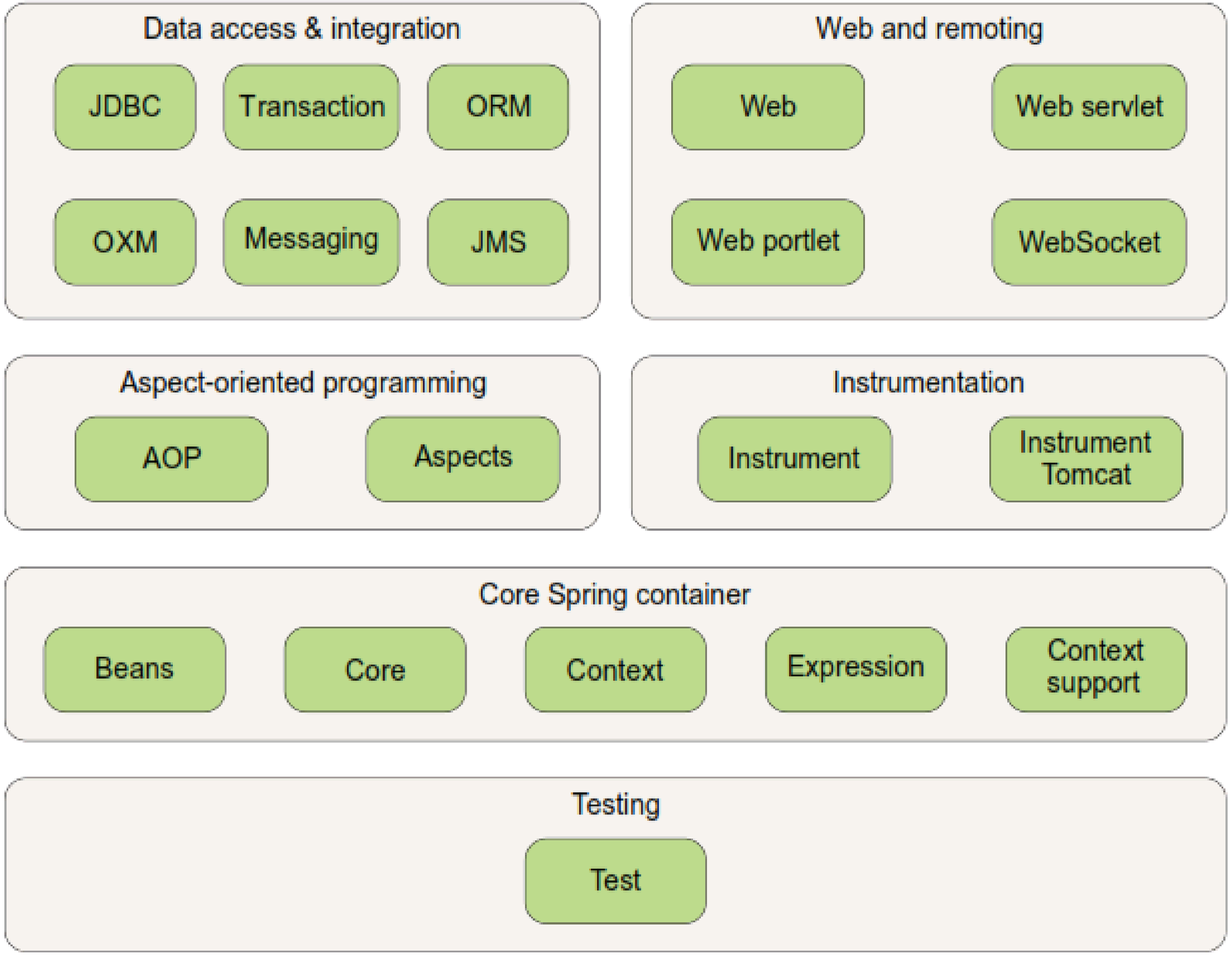
A Beans Life

- A bean goes through several steps between creation and destruction in the Spring container. Each step is an opportunity to customize how the bean is managed in Spring.


Instantiate
Populate properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
ApplicationContextAware's setApplicationContext()
Pre-initialization BeanPostProcessors
InitializingBean 's afterPropertiesSet()
Call custom init-method
Post-initialization BeanPostProcessors
Bean is ready to use
Container is shutdown
DisposableBean's destroy()
Call custom destroy-method

Spring Modules

- The Spring Framework is made up of six well-defined module categories.



Data access & integration

JDBC

Transaction

ORM

OXM

Messaging

JMS

Web and remoting

Web

Web servlet

Web portlet

WebSocket

Aspect-oriented programming

AOP

Aspects

Instrumentation

Instrument

Instrument Tomcat

Core Spring container

Beans

Core

Context

Expression

Context support

Testing

Test

The Spring portfolio

- **Spring Web Flow** - Spring‘s core MVC Framework to provide support for building conversational, flow-based web applications
- **Spring Web Services** - Spring Web Services offers a contract-first web services model where service implementations are written to satisfy the service contract (In Spring Core you can generate Webservice from Bean’s Interface – contract-last)
- **Spring Security** - Implemented using Spring AOP
- **Spring Integration** - Spring Integration offers implementations of several common integration patterns in Spring’s declarative style.
- **Spring Batch, Spring Data** (all kinds of databases, also NonSQL), **Spring Social** (Facebook, Twitter etc. via REST API), **Spring Mobile** (Extension to Spring MVC)**, Spring for Android, Spring Boot**

Wiring beans

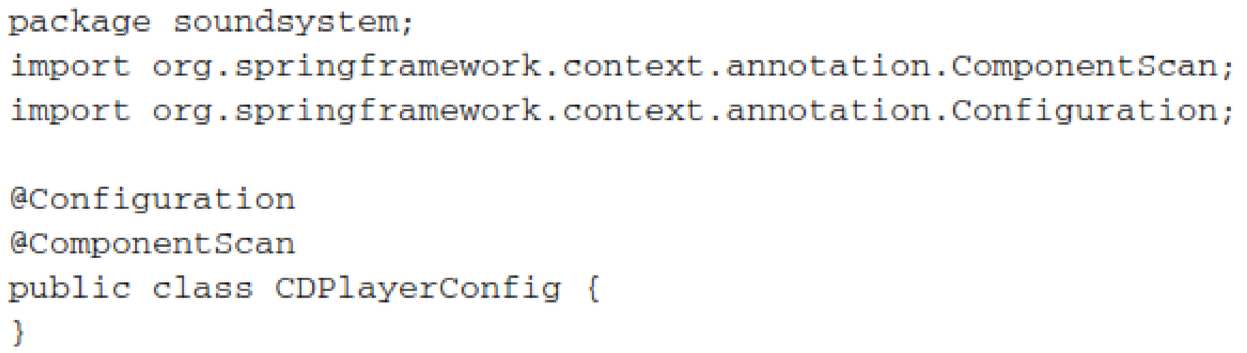
Spring‘s configuration options

- Explicit configuration in XML
- Explicit configuration in Java
- Implicit bean discovery and automatic wiring
  - *Component scanning—Spring automatically discovers beans to be created in the* application context.
  - *Autowiring—Spring automatically satisfies bean dependencies.*

Creating discoverable beans

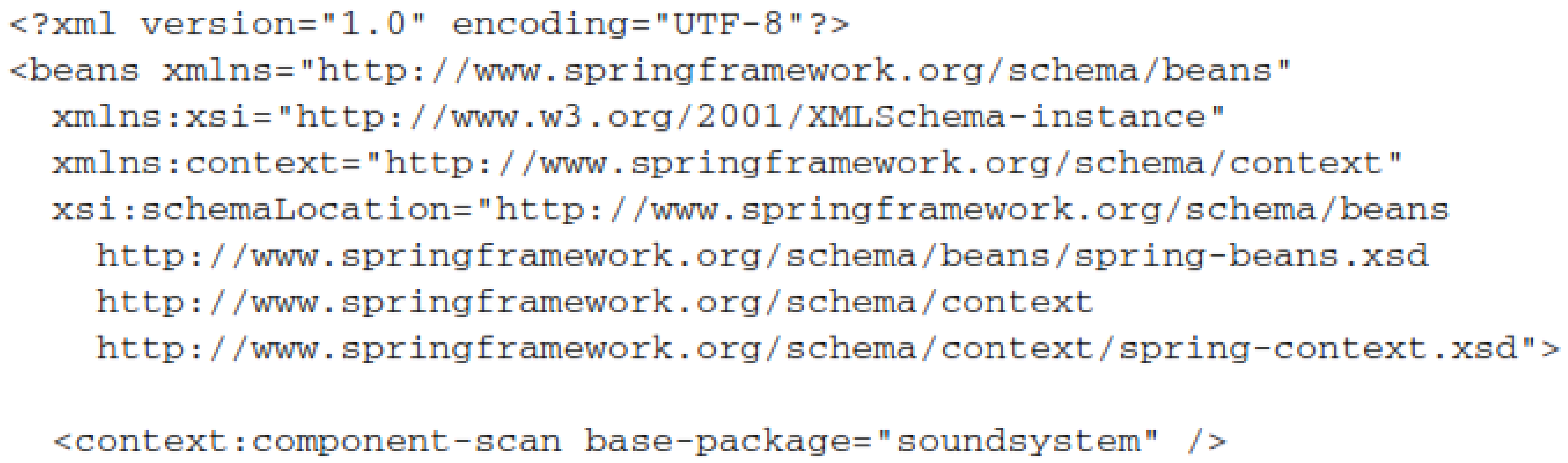
To enable component scanning, you can

- define a configuration class without any bean with @Configuration and @ComponentScan annotation in the same package as the beans



```java
package soundsystem;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan
public class CDPlayerConfig {
}
```

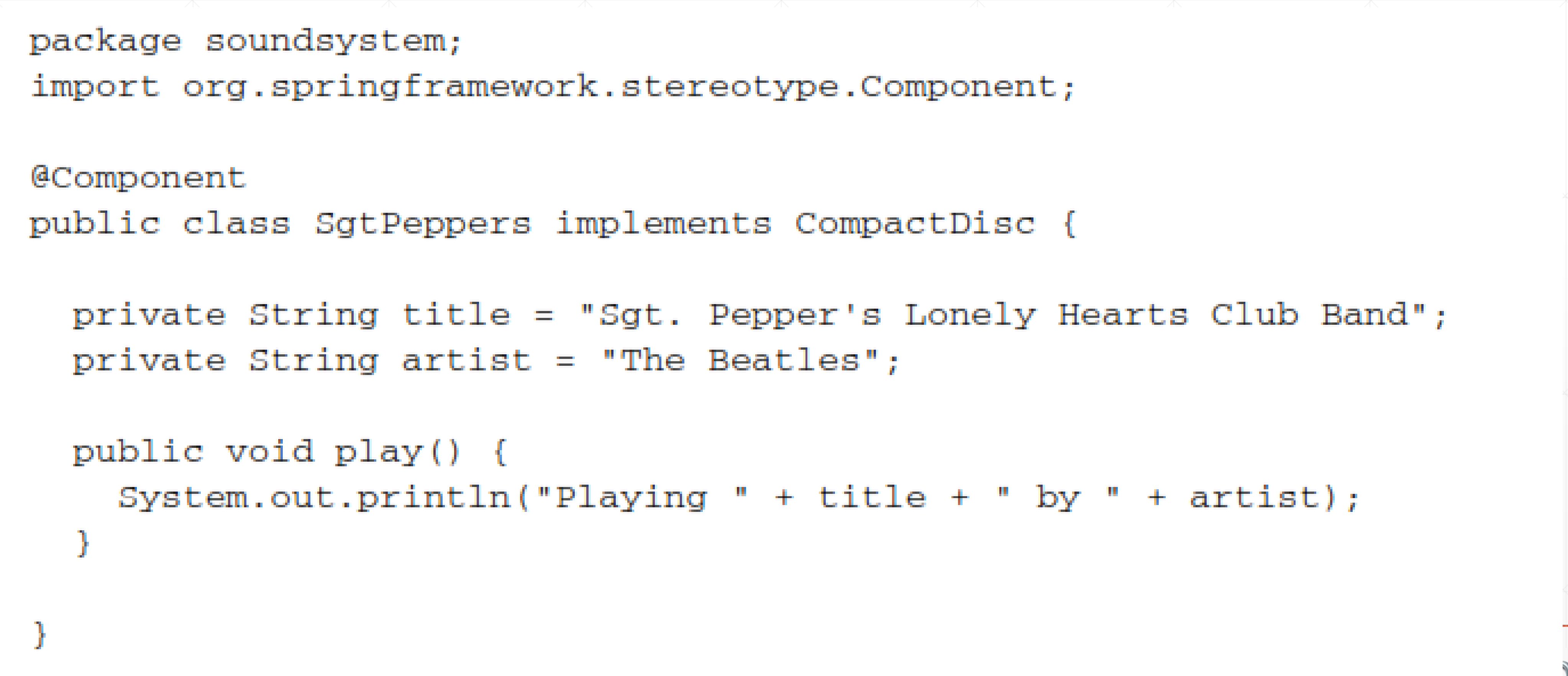
- You can use a XML configuration



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

  <context:component-scan base-package="soundsystem" />
```

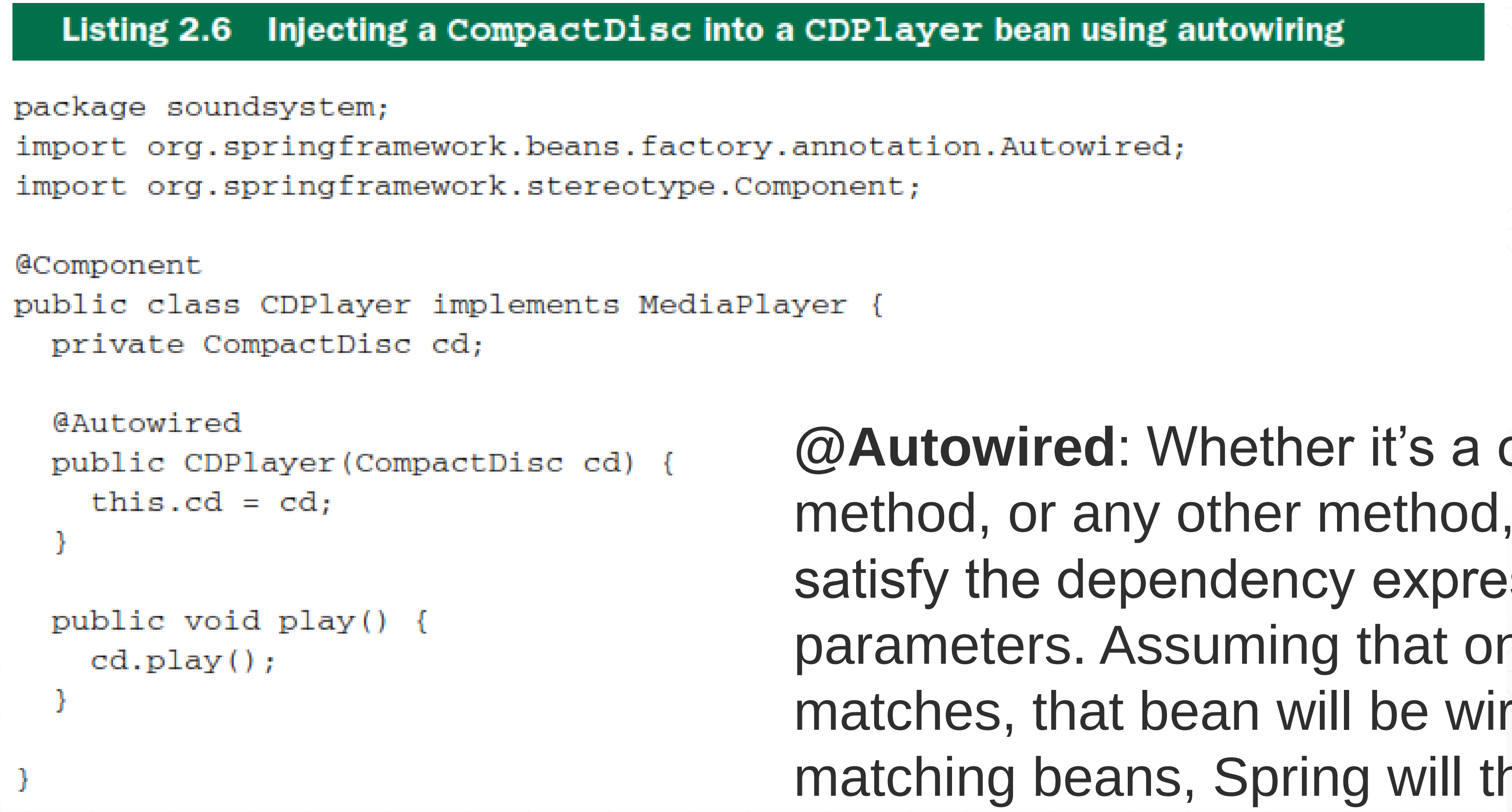
Creating discoverable beans

@Component SgtPeppers is a bean to be discovered



```
package soundsystem;
import org.springframework.stereotype.Component;

@Component
public class SgtPeppers implements CompactDisc {

  private String title = "Sgt. Pepper's Lonely Hearts Club Band";
  private String artist = "The Beatles";

  public void play() {
    System.out.println("Playing " + title + " by " + artist);
  }

}
```

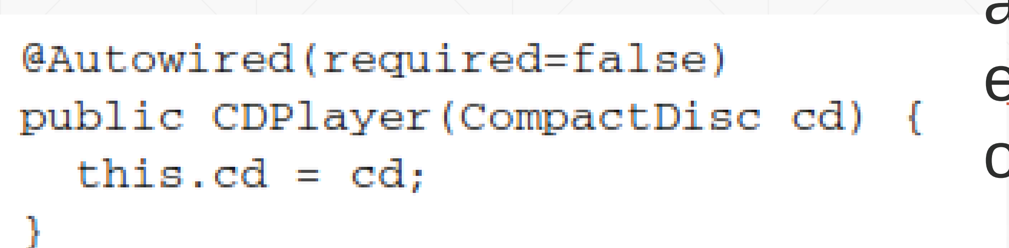
Annotating beans to be automatically wired



Listing 2.6 Injecting a CompactDisc into a CDPlayer bean using autowiring

```
package soundsystem;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class CDPlayer implements MediaPlayer {
  private CompactDisc cd;

  @Autowired
  public CDPlayer(CompactDisc cd) {
    this.cd = cd;
  }

  public void play() {
    cd.play();
  }

}
```

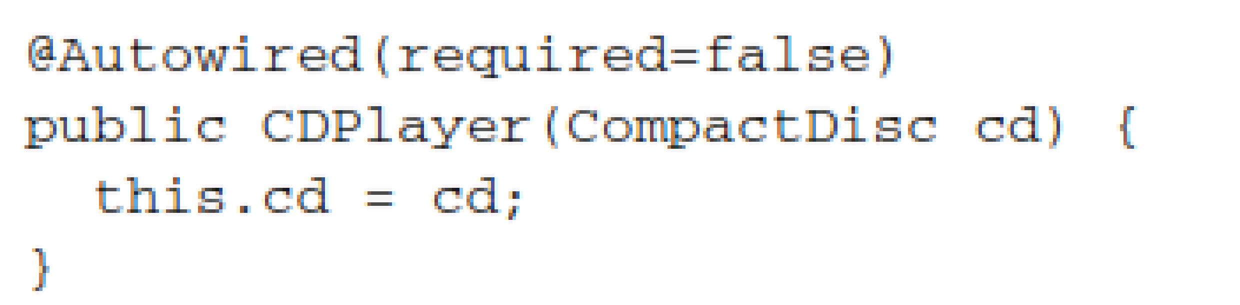
@Autowired: Whether it’s a constructor, a setter method, or any other method, Spring will attempt to satisfy the dependency expressed in the method’s parameters. Assuming that one and only one bean matches, that bean will be wired in. If there are no matching beans, Spring will throw an exception as the application context is being created. To avoid that exception, you can set the required attribute on @Autowired to false:



```
@Autowired(required=false)
public CDPlayer(CompactDisc cd) {
  this.cd = cd;
}
```

Annotating beans to be automatically wired

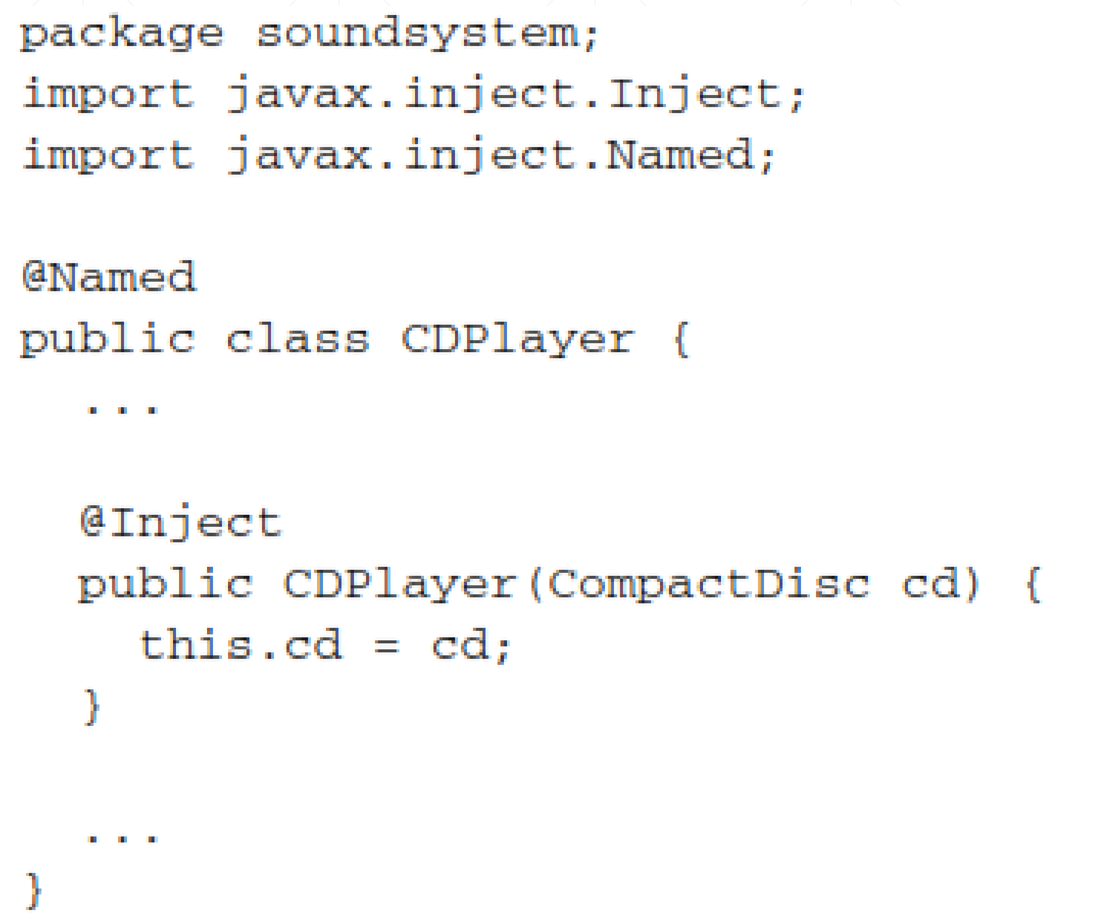
@Autowired: Whether it’s a constructor, a setter method, or any other method, Spring will attempt to satisfy the dependency expressed in the method’s parameters. Assuming that one and only one bean matches, that bean will be wired in. If there are no matching beans, Spring will throw an exception as the application context is being created. To avoid that exception, you can set the required attribute on @Autowired to false:



```
@Autowired(required=false)
public CDPlayer(CompactDisc cd) {
  this.cd = cd;
}
```

Annotating beans to be automatically wired

- @Autowired is a Spring-specific annotation. If it troubles you to be scattering Spring-specific annotations throughout your code for autowiring, you might consider using the @Inject annotation instead:

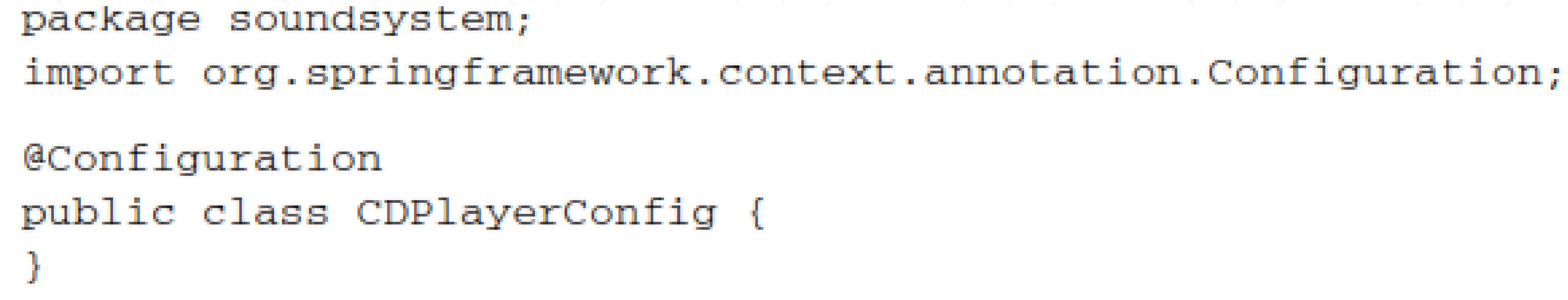


```
package soundsystem;
import javax.inject.Inject;
import javax.inject.Named;

@Named
public class CDPlayer {
  ...

  @Inject
  public CDPlayer(CompactDisc cd) {
    this.cd = cd;
  }

  ...
}
```

- @Inject comes from the Java Dependency Injection specification
- Although there are some subtle differences between @Inject and @Autowired, they’re interchangeable in many cases.

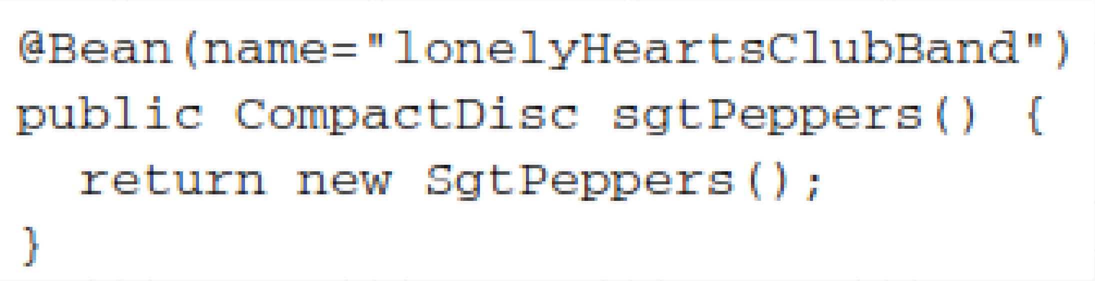
Wiring beans with Java

1. Creating a configuration Class



```
package soundsystem;
import org.springframework.context.annotation.Configuration;

@Configuration
public class CDPlayerConfig {
}
```

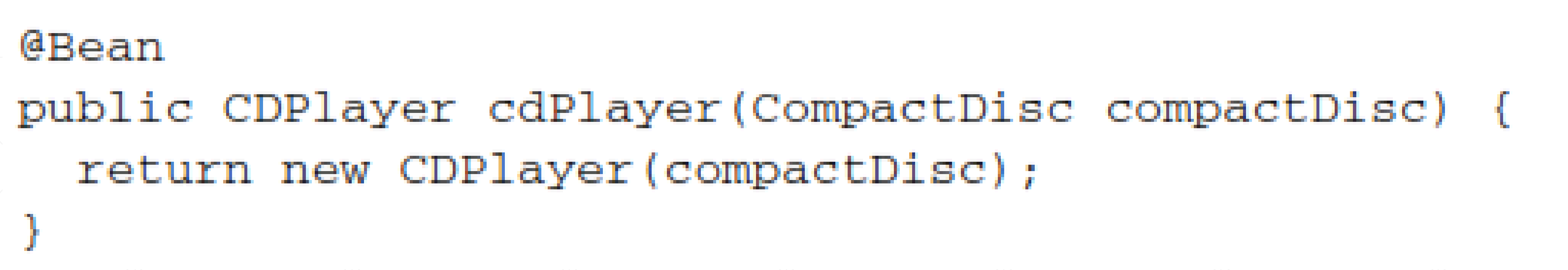
2. Declaring a simple Bean



```
@Bean(name="lonelyHeartsClubBand")
public CompactDisc sgtPeppers() {
  return new SgtPeppers();
}
```

Wiring beans with Java

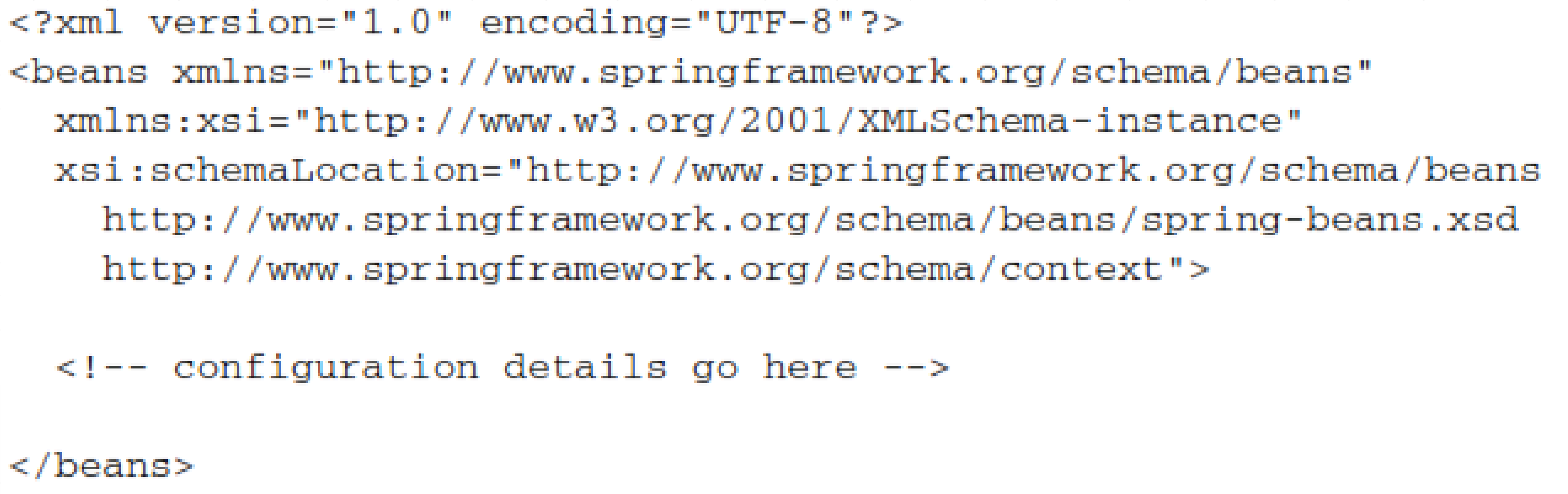
3. Injecting with JavaConfig



```
@Bean
public CDPlayer cdPlayer(CompactDisc compactDisc) {
  return new CDPlayer(compactDisc);
}
```

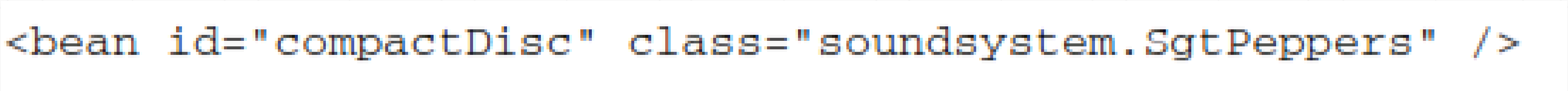
Wiring beans with XML

1. Creating an XML configuration specification (you can use Spring Tool Suite)



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

  <!-- configuration details go here -->

</beans>
```

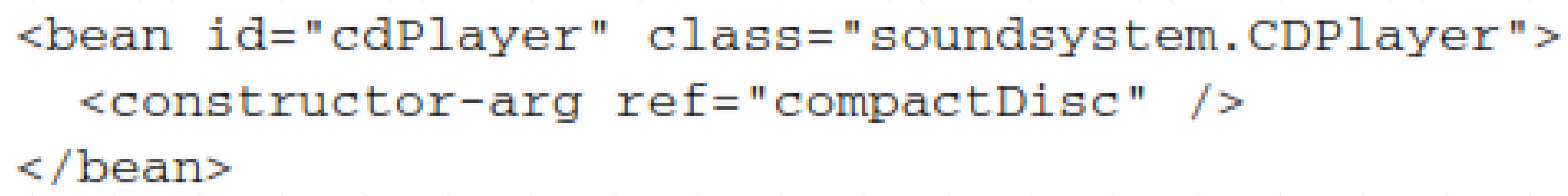
2. Declaring a simple <bean> - (you do not need to name the bean)



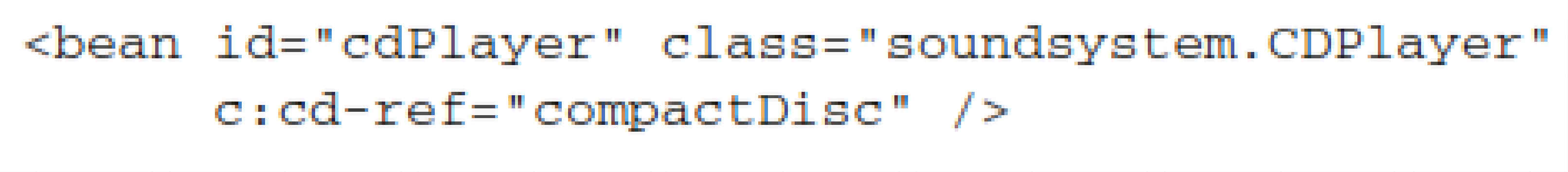
```
<bean id="compactDisc" class="soundsystem.SgtPeppers" />
```

Wiring beans with XML

3. Initializing a bean with constructor injection with <constructor-arg> element or c-namespace



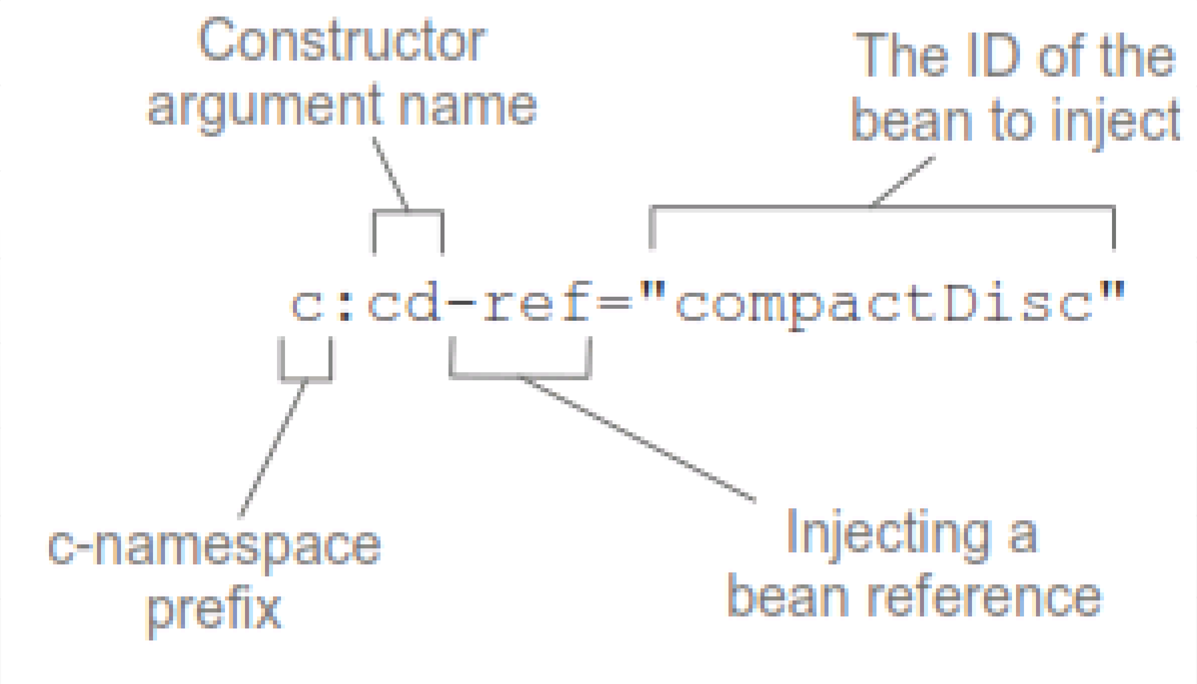
```
<bean id="cdPlayer" class="soundsystem.CDPlayer">
  <constructor-arg ref="compactDisc" />
</bean>
```



```
<bean id="cdPlayer" class="soundsystem.CDPlayer"
      c:cd-ref="compactDisc" />
```

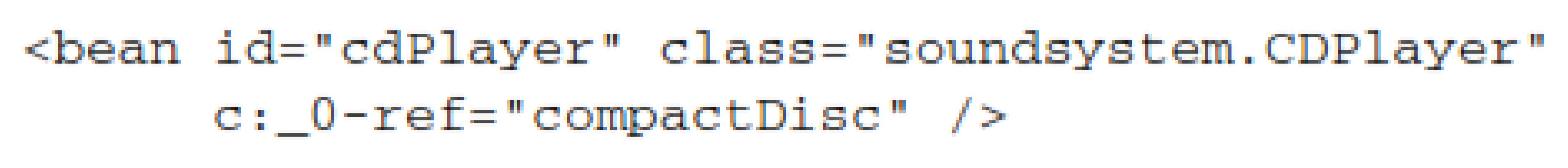
Initializing a bean with constructor injection (cont.)

- Injecting a bean reference into a constructor argument with Spring‘s c-namespace

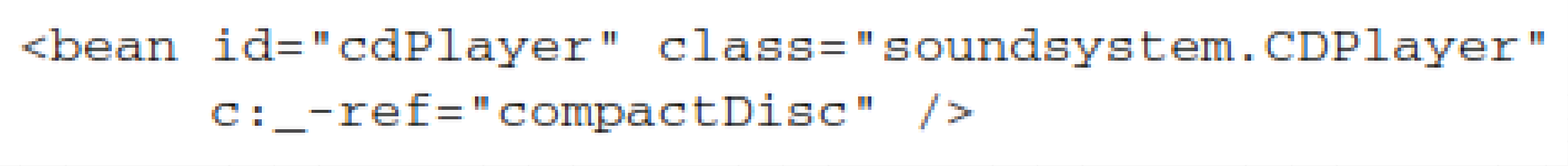

Constructor argument name
The ID of the bean to inject
c:cd-ref="compactDisc"
c-namespace prefix
Injecting a bean reference

Initializing a bean with constructor injection (cont.)

You can refer the constructor argument name, position in the argument list or if you have only one argument, you don‘t need to specify it at all:

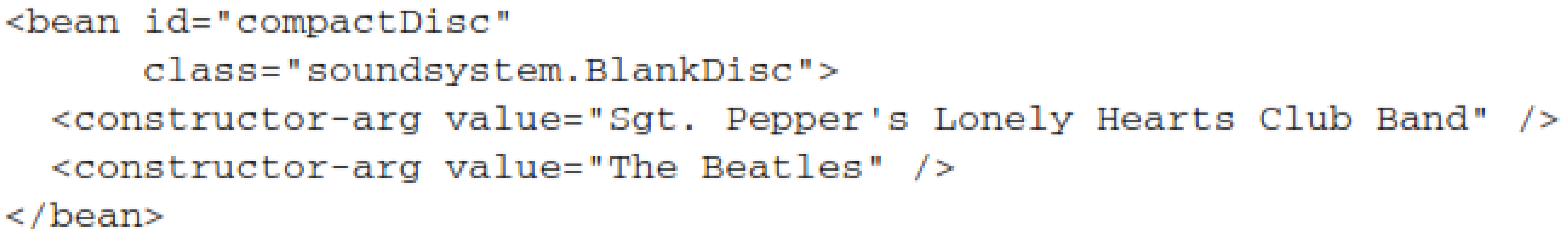


```
<bean id="cdPlayer" class="soundsystem.CDPlayer"
      c:_0-ref="compactDisc" />
```

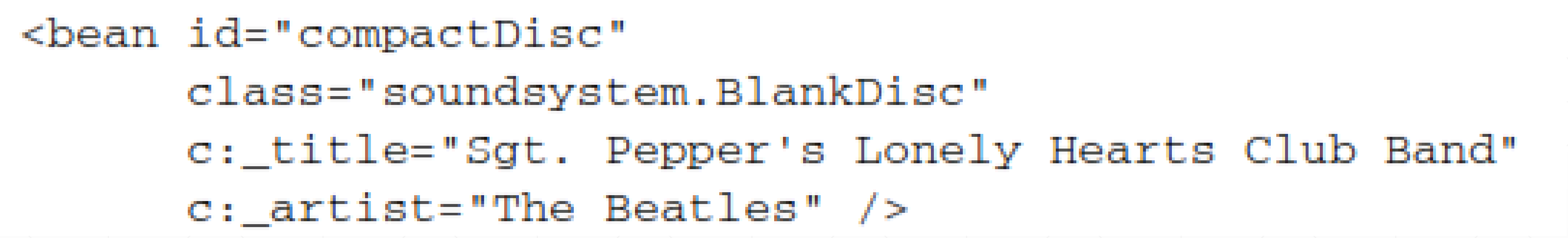


```
<bean id="cdPlayer" class="soundsystem.CDPlayer"
      c:_-ref="compactDisc" />
```

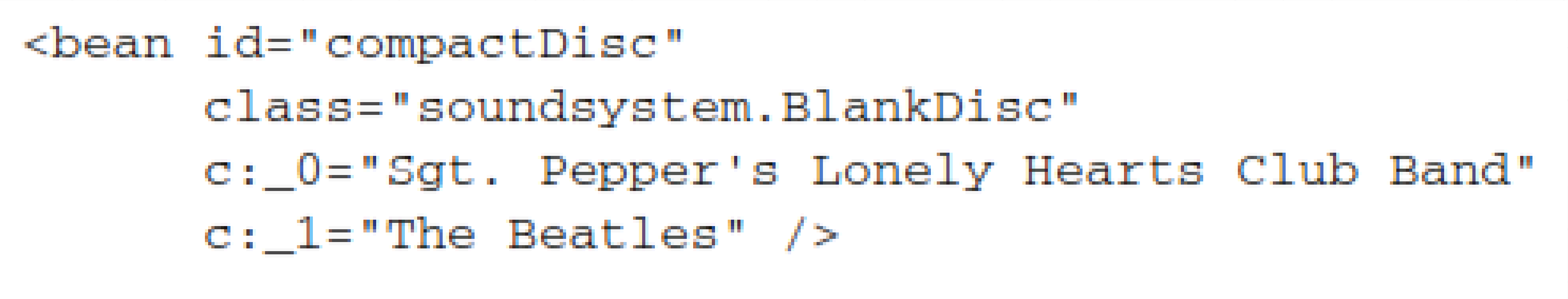
Injecting constructor with literal values



```
<bean id="compactDisc"
      class="soundsystem.BlankDisc">
  <constructor-arg value="Sgt. Pepper's Lonely Hearts Club Band" />
  <constructor-arg value="The Beatles" />
</bean>
```

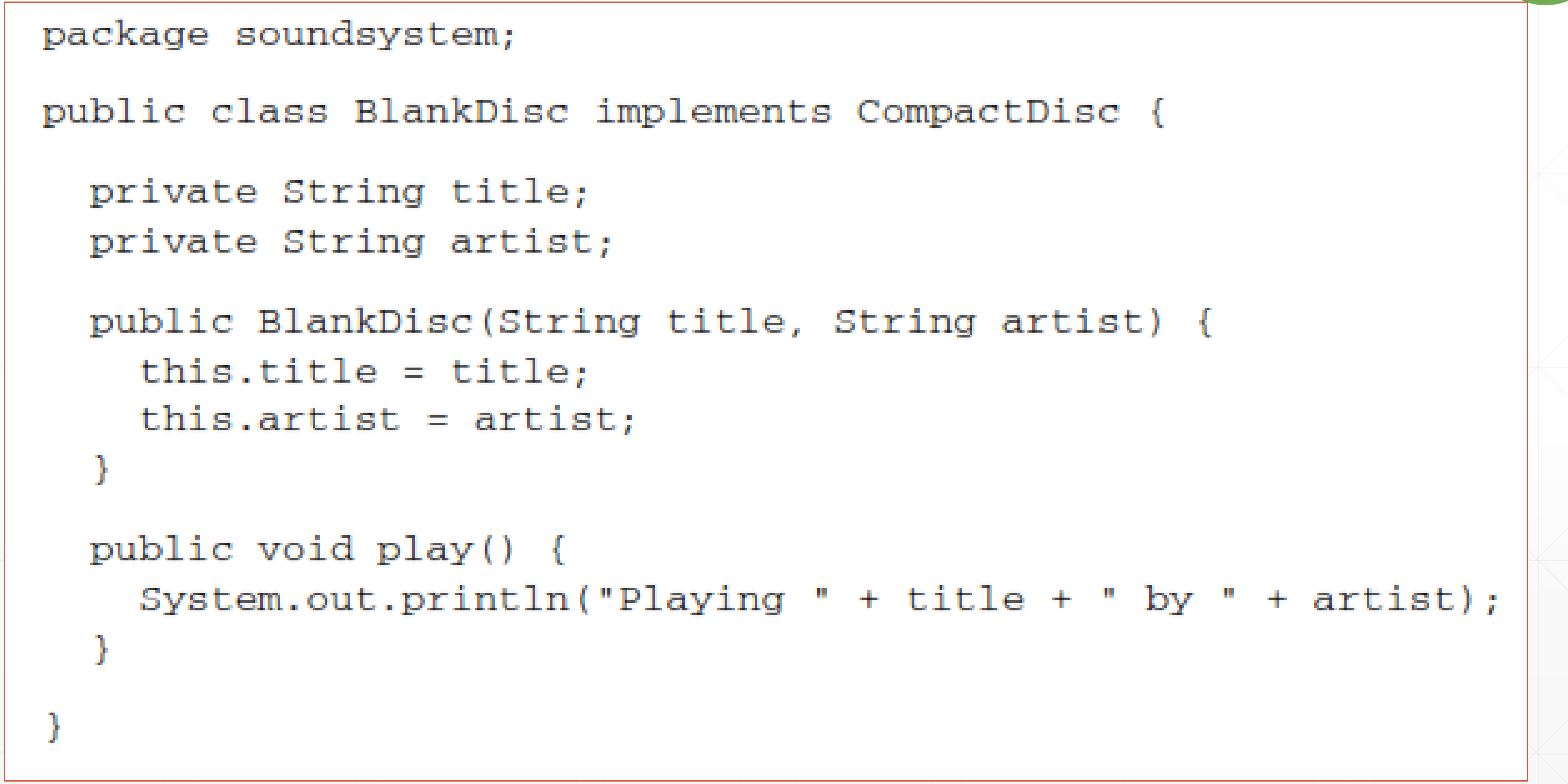


```
<bean id="compactDisc"
      class="soundsystem.BlankDisc"
      c:_title="Sgt. Pepper's Lonely Hearts Club Band"
      c:_artist="The Beatles" />
```



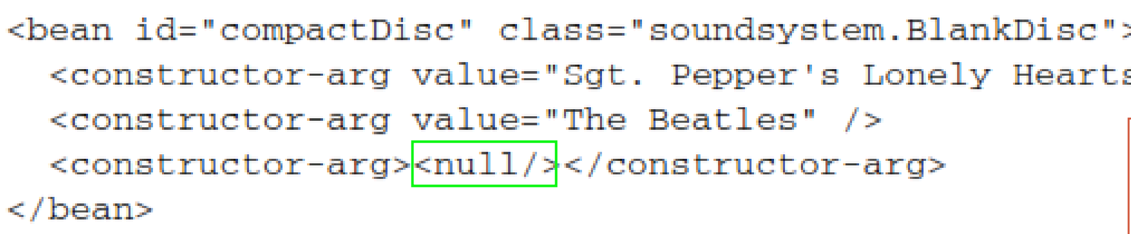
```
<bean id="compactDisc"
      class="soundsystem.BlankDisc"
      c:_0="Sgt. Pepper's Lonely Hearts Club Band"
      c:_1="The Beatles" />
```

Injecting constructor with literal values

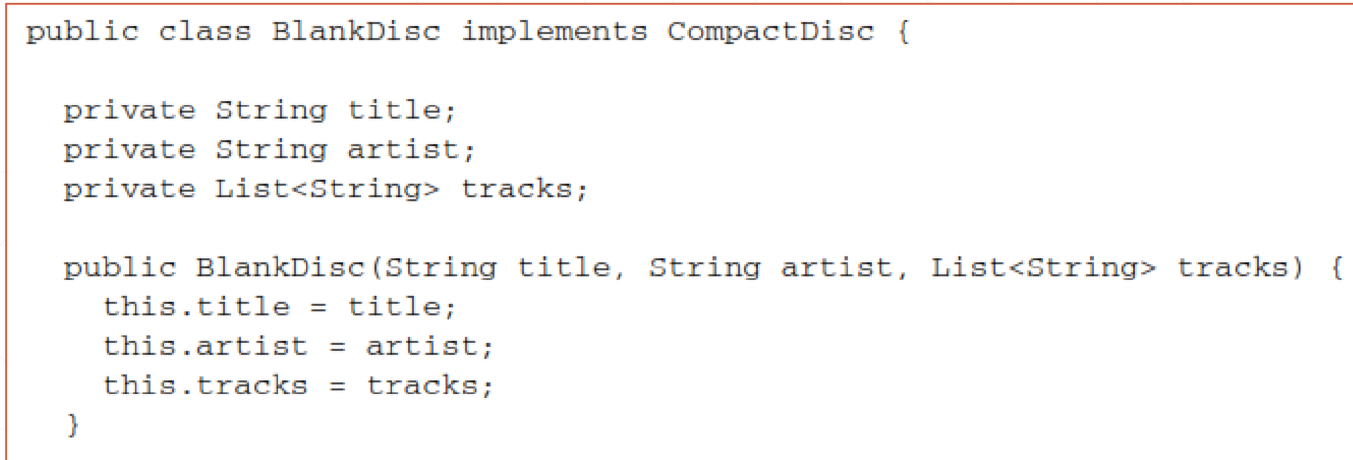


```
package soundsystem;

public class BlankDisc implements CompactDisc {

  private String title;
  private String artist;

  public BlankDisc(String title, String artist) {
    this.title = title;
    this.artist = artist;
  }

  public void play() {
    System.out.println("Playing " + title + " by " + artist);
  }

}
```

Wiring Collections

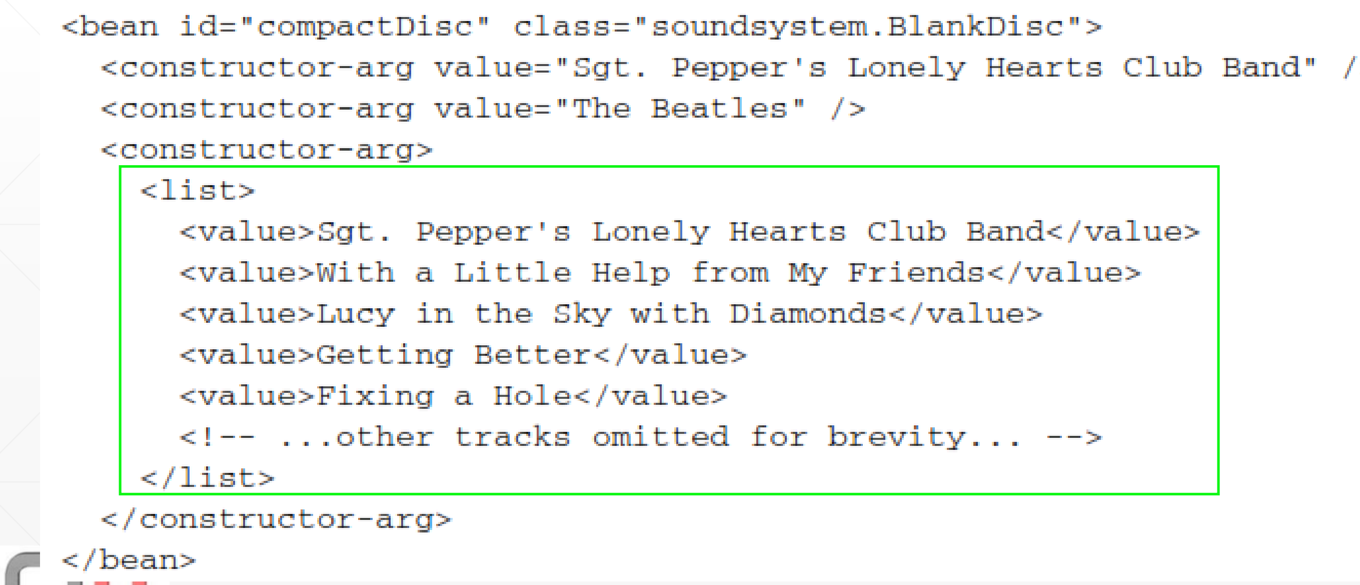


```
<bean id="compactDisc" class="soundsystem.BlankDisc">
  <constructor-arg value="Sgt. Pepper's Lonely Hearts Club Band" />
  <constructor-arg value="The Beatles" />
  <constructor-arg><null/></constructor-arg>
</bean>
```

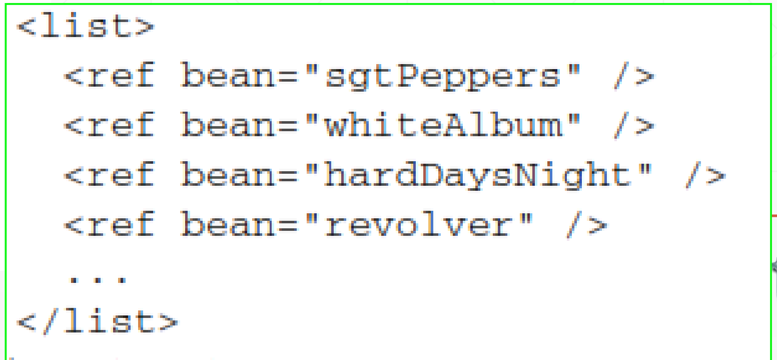


```
public class BlankDisc implements CompactDisc {

  private String title;
  private String artist;
  private List<String> tracks;

  public BlankDisc(String title, String artist, List<String> tracks) {
    this.title = title;
    this.artist = artist;
    this.tracks = tracks;
  }
```

Null or a list



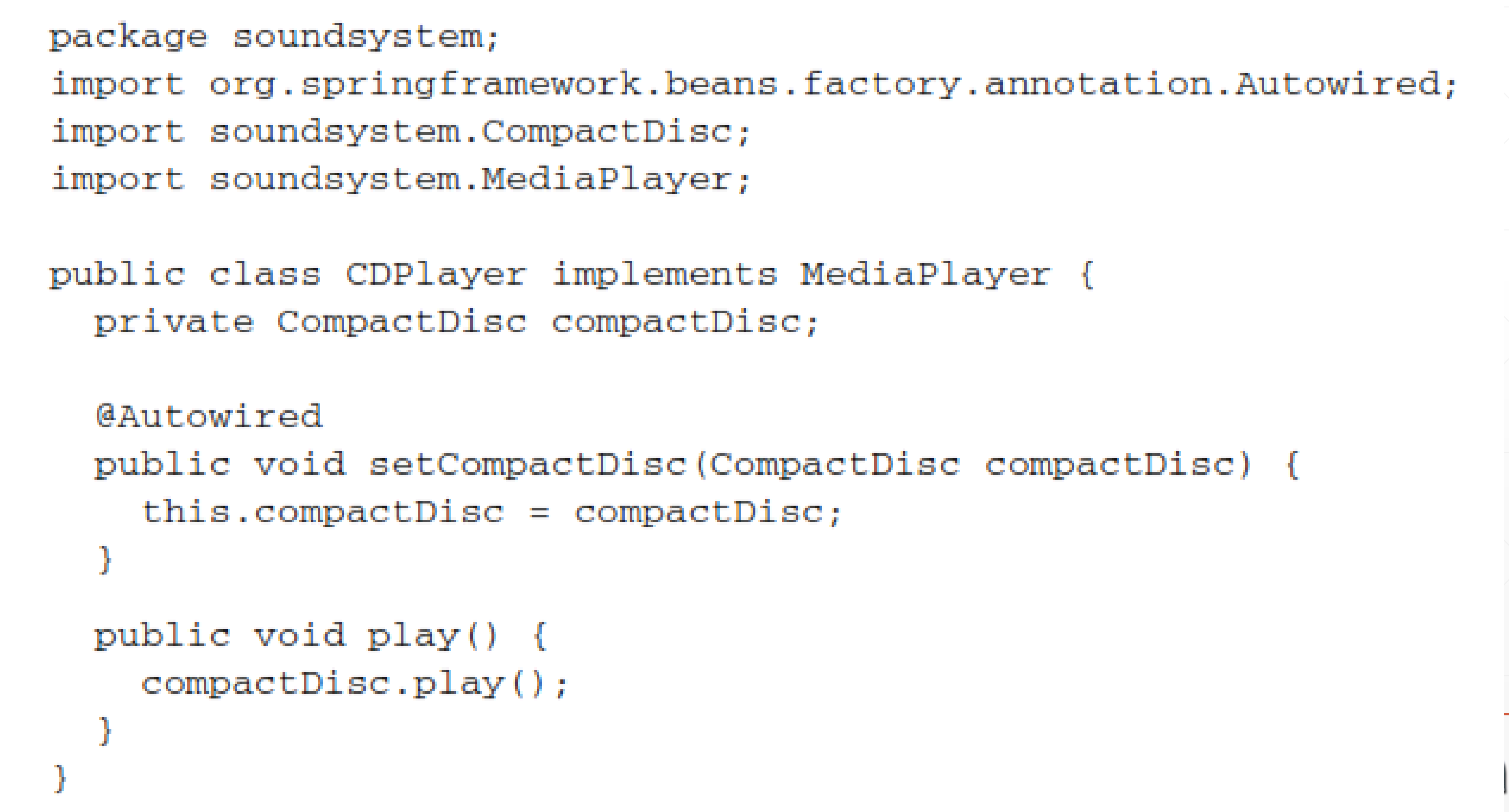
```
<bean id="compactDisc" class="soundsystem.BlankDisc">
  <constructor-arg value="Sgt. Pepper's Lonely Hearts Club Band" />
  <constructor-arg value="The Beatles" />
  <constructor-arg>
    <list>
      <value>Sgt. Pepper's Lonely Hearts Club Band</value>
      <value>With a Little Help from My Friends</value>
      <value>Lucy in the Sky with Diamonds</value>
      <value>Getting Better</value>
      <value>Fixing a Hole</value>
      <!-- ...other tracks omitted for brevity... -->
    </list>
  </constructor-arg>
</bean>
```



```
<list>
  <ref bean="sgtPeppers" />
  <ref bean="whiteAlbum" />
  <ref bean="hardDaysNight" />
  <ref bean="revolver" />
  ...
</list>
```

Setting properties

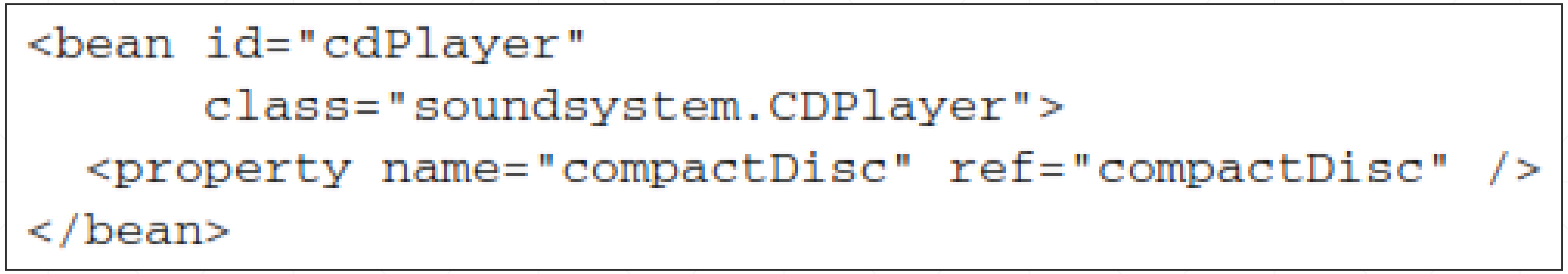
- Constructor injection for hard dependencies
- Property injection for optional dependencies



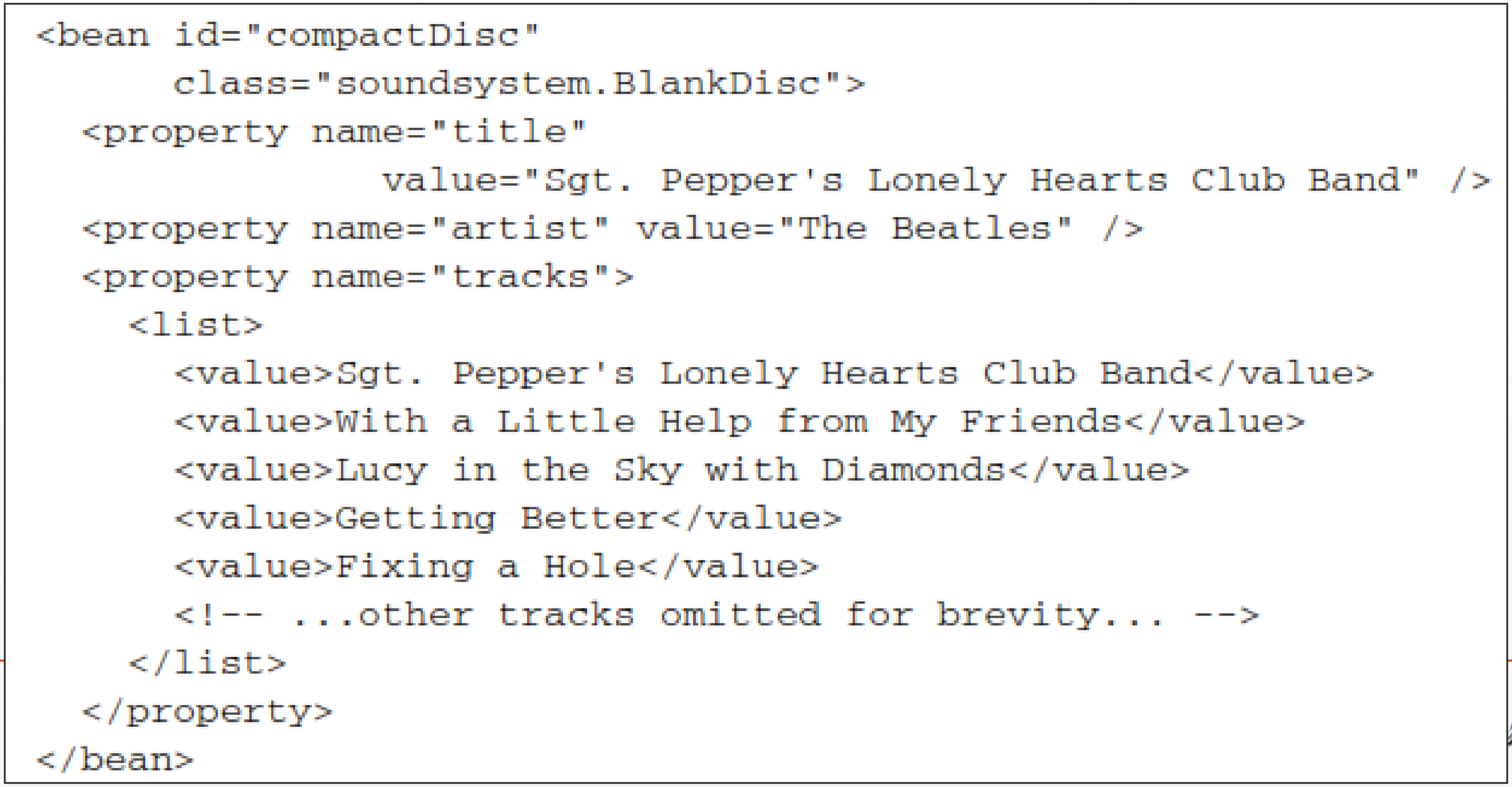
```
package soundsystem;
import org.springframework.beans.factory.annotation.Autowired;
import soundsystem.CompactDisc;
import soundsystem.MediaPlayer;

public class CDPlayer implements MediaPlayer {
  private CompactDisc compactDisc;

  @Autowired
  public void setCompactDisc(CompactDisc compactDisc) {
    this.compactDisc = compactDisc;
  }

  public void play() {
    compactDisc.play();
  }
}
```

Wiring Properties

- Using <property> element



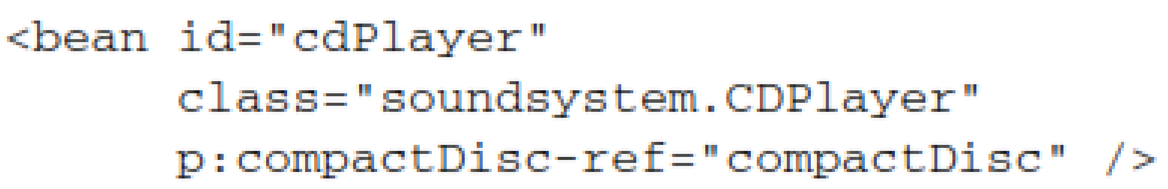
```
<bean id="cdPlayer"
      class="soundsystem.CDPlayer">
  <property name="compactDisc" ref="compactDisc" />
</bean>
```



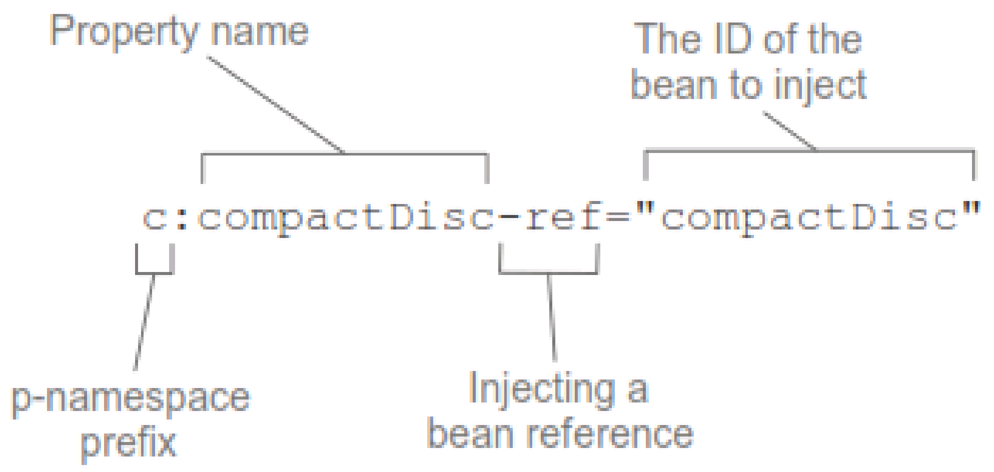
```
<bean id="compactDisc"
      class="soundsystem.BlankDisc">
  <property name="title"
            value="Sgt. Pepper's Lonely Hearts Club Band" />
  <property name="artist" value="The Beatles" />
  <property name="tracks">
    <list>
      <value>Sgt. Pepper's Lonely Hearts Club Band</value>
      <value>With a Little Help from My Friends</value>
      <value>Lucy in the Sky with Diamonds</value>
      <value>Getting Better</value>
      <value>Fixing a Hole</value>
      <!-- ...other tracks omitted for brevity... -->
    </list>
  </property>
</bean>
```

Wiring Properties

- Using p-namespace (you have to declare p-namespace in XML file bevor use)

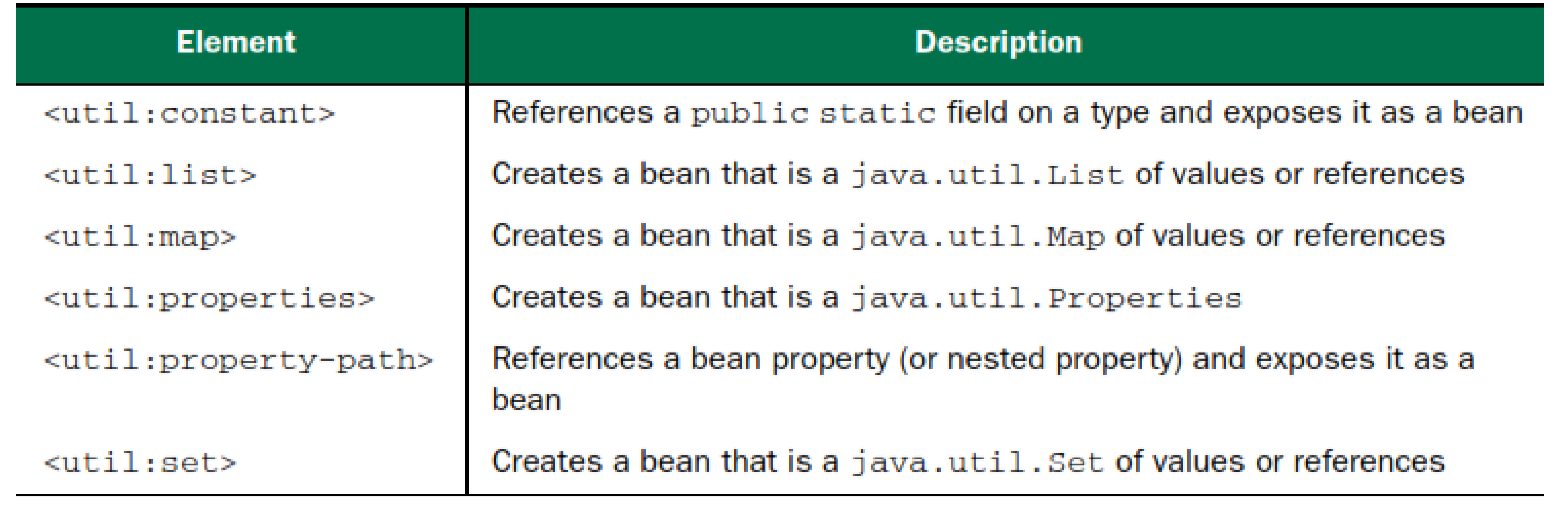


```
<bean id="cdPlayer"
      class="soundsystem.CDPlayer"
      p:compactDisc-ref="compactDisc" />
```


Property name
The ID of the bean to inject
c:compactDisc-ref="compactDisc"
p-namespace prefix
Injecting a bean reference

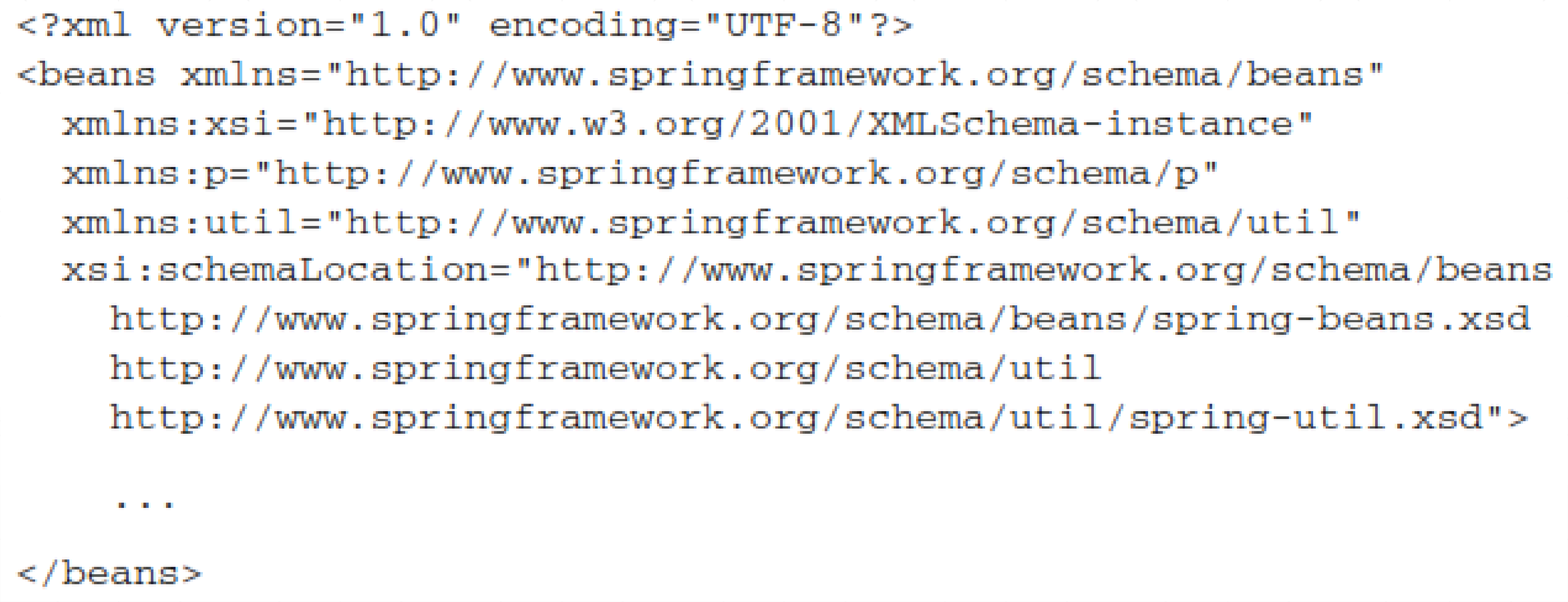
- injecting literal values p:property-name without „ref“
- There is no no convenient way to specify a list of values with the p-namespace

Spring util-namespace



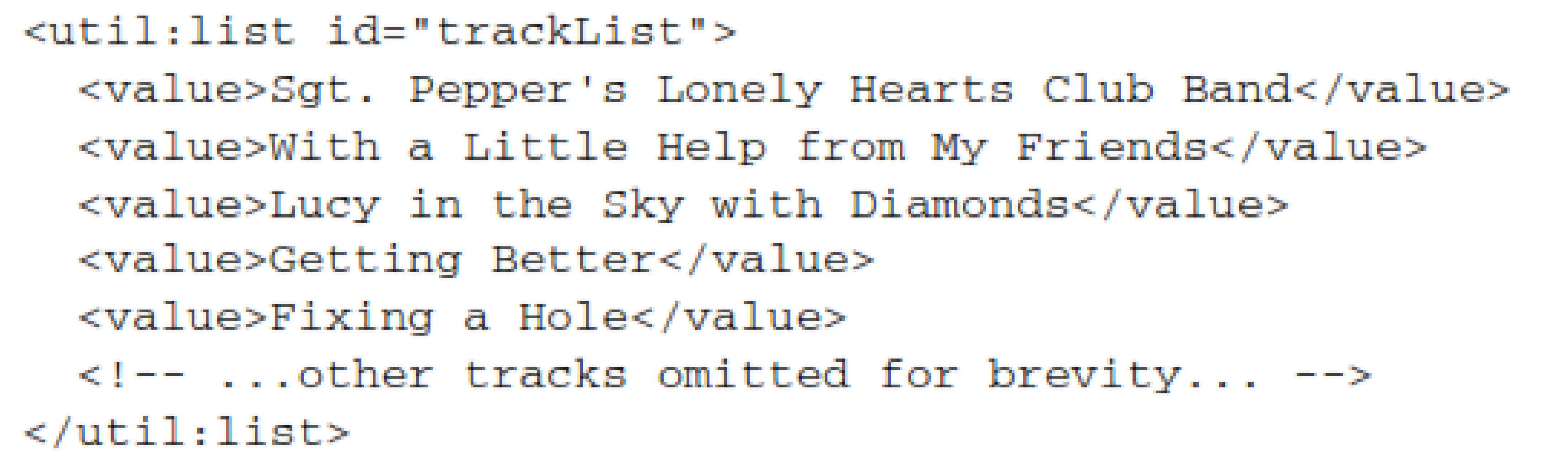
Element	Description
`<util:constant>`	References a `public static` field on a type and exposes it as a bean
`<util:list>`	Creates a bean that is a `java.util.List` of values or references
`<util:map>`	Creates a bean that is a `java.util.Map` of values or references
`<util:properties>`	Creates a bean that is a `java.util.Properties`
`<util:property-path>`	References a bean property (or nested property) and exposes it as a bean
`<util:set>`	Creates a bean that is a `java.util.Set` of values or references

Spring util-namespace

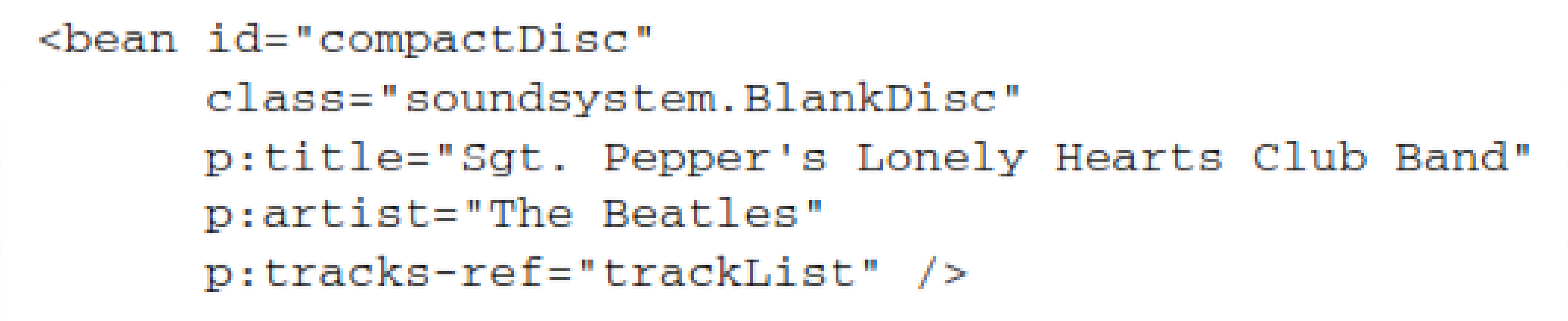


```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:p="http://www.springframework.org/schema/p"
  xmlns:util="http://www.springframework.org/schema/util"
  xsi:schemaLocation="http://www.springframework.org/schema/beans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

    ...

</beans>
```

Spring util-namespace



```
<util:list id="trackList">
  <value>Sgt. Pepper's Lonely Hearts Club Band</value>
  <value>With a Little Help from My Friends</value>
  <value>Lucy in the Sky with Diamonds</value>
  <value>Getting Better</value>
  <value>Fixing a Hole</value>
  <!-- ...other tracks omitted for brevity... -->
</util:list>
```

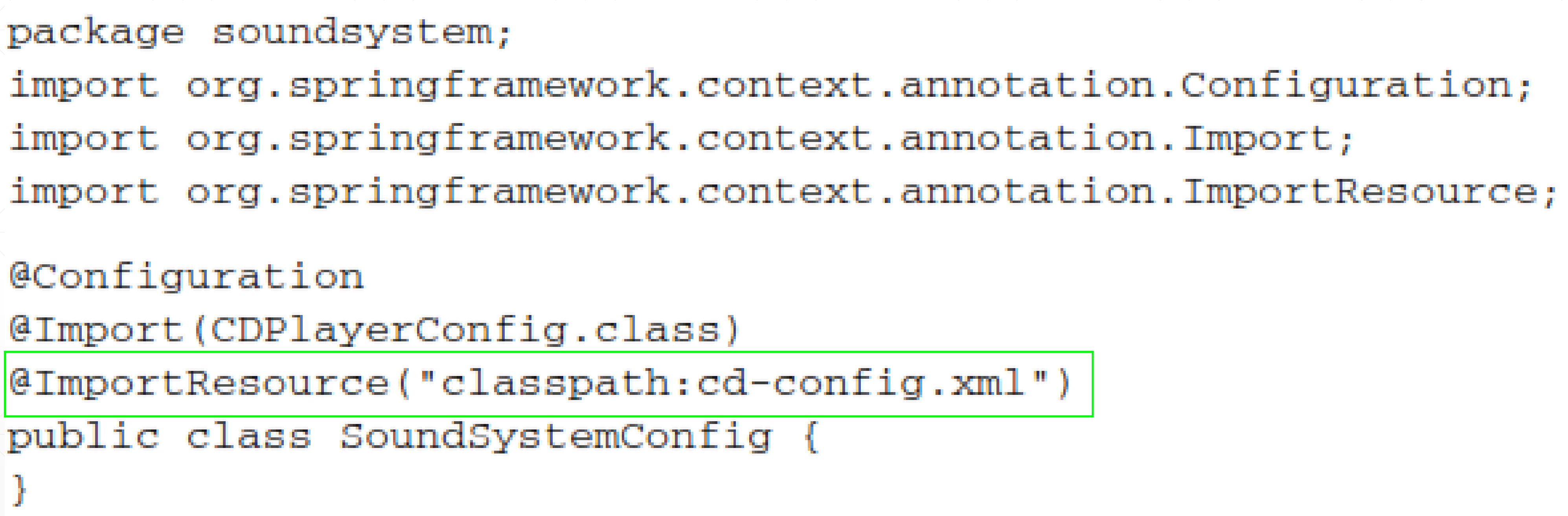


```
<bean id="compactDisc"
      class="soundsystem.BlankDisc"
      p:title="Sgt. Pepper's Lonely Hearts Club Band"
      p:artist="The Beatles"
      p:tracks-ref="trackList" />
```

Importing and mixing configurations

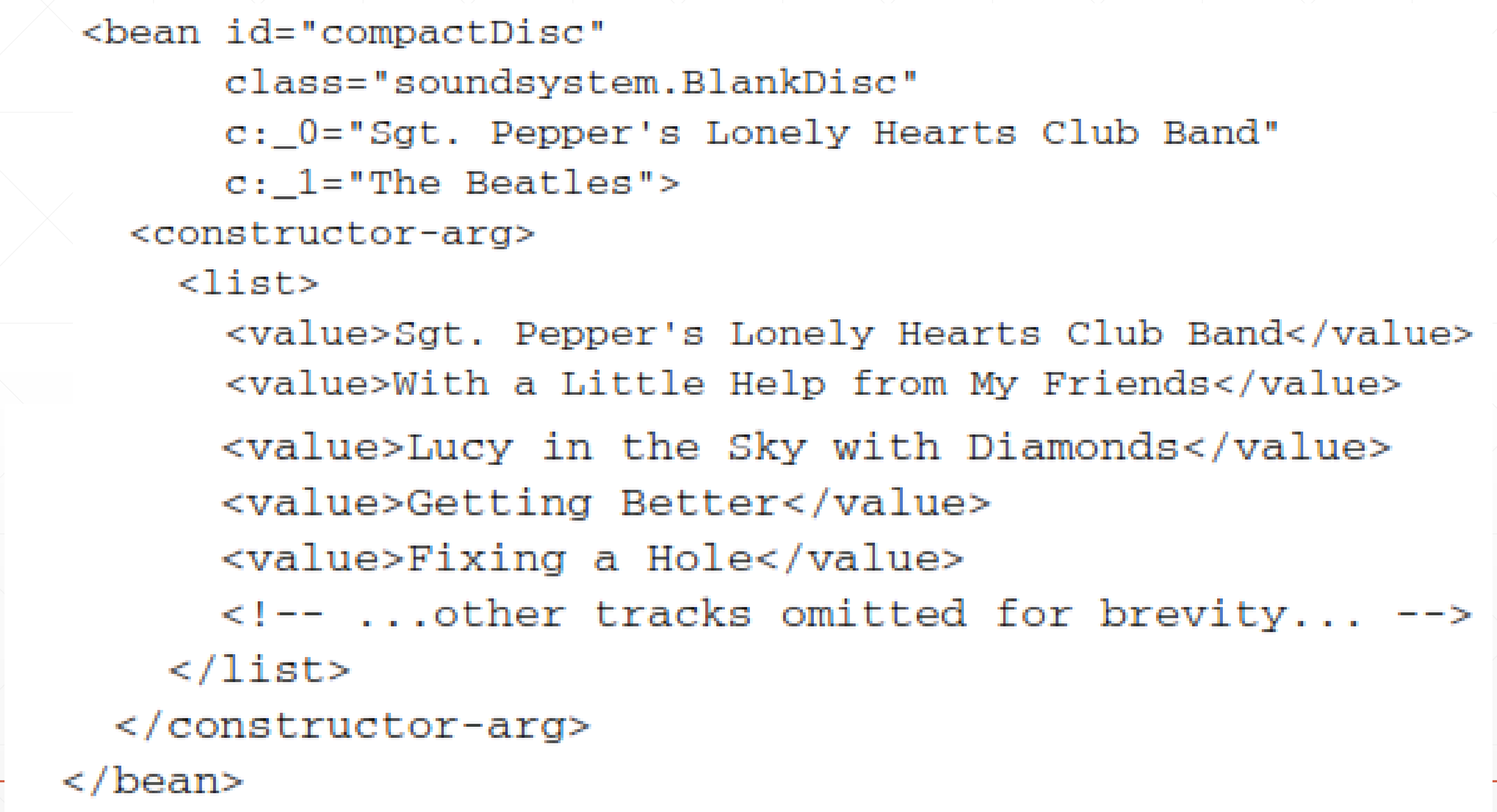
- You are free to mix component scanning and autowiring with JavaConfig and/or XML configuration
- Autowiring considers all beans in the Spring container, regardless of whether they were declared in JavaConfig or XML or picked up by component scanning

Referencing XML configuration in JavaConfig



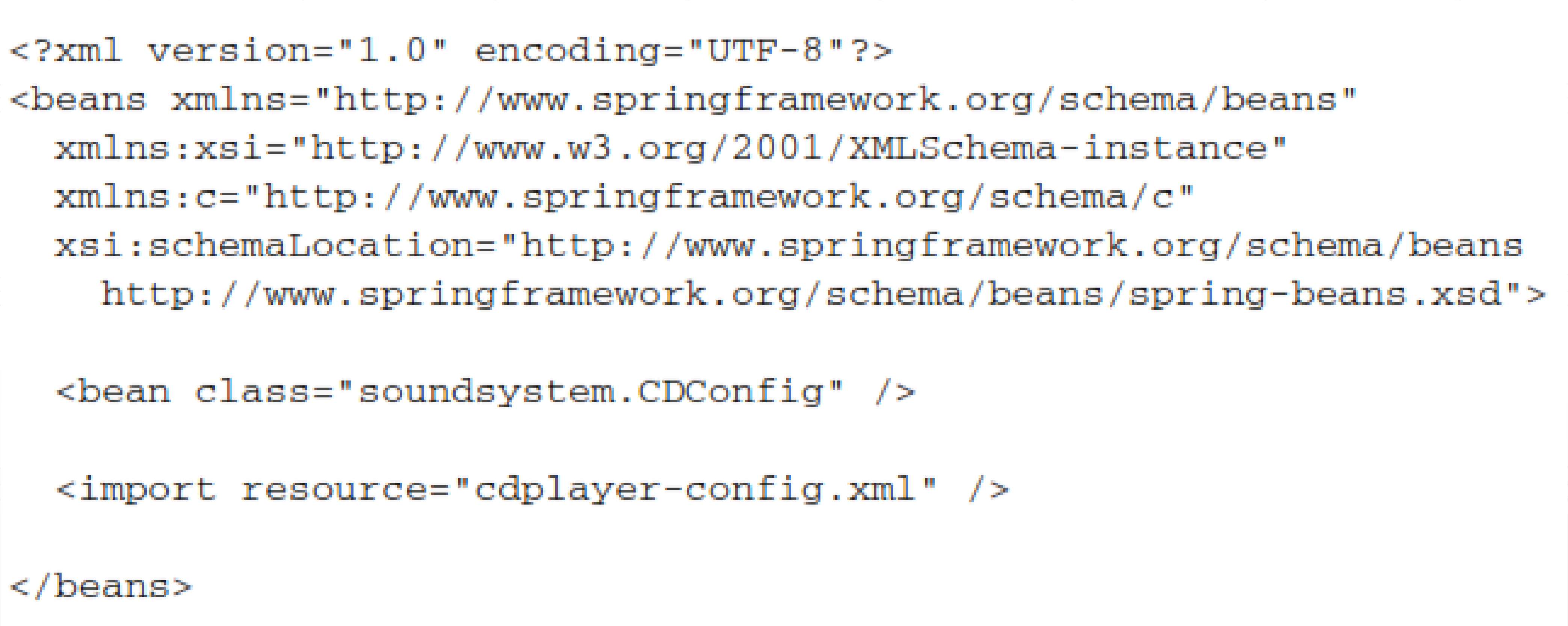
```
package soundsystem;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.context.annotation.ImportResource;

@Configuration
@Import(CDPlayerConfig.class)
@ImportResource("classpath:cd-config.xml")
public class SoundSystemConfig {
}
```

Referencing XML configuration in JavaConfig



```
<bean id="compactDisc"
      class="soundsystem.BlankDisc"
      c:_0="Sgt. Pepper's Lonely Hearts Club Band"
      c:_1="The Beatles">
  <constructor-arg>
    <list>
      <value>Sgt. Pepper's Lonely Hearts Club Band</value>
      <value>With a Little Help from My Friends</value>
      <value>Lucy in the Sky with Diamonds</value>
      <value>Getting Better</value>
      <value>Fixing a Hole</value>
      <!-- ...other tracks omitted for brevity... -->
    </list>
  </constructor-arg>
</bean>
```

Referencing JavaConfig in XML configuration

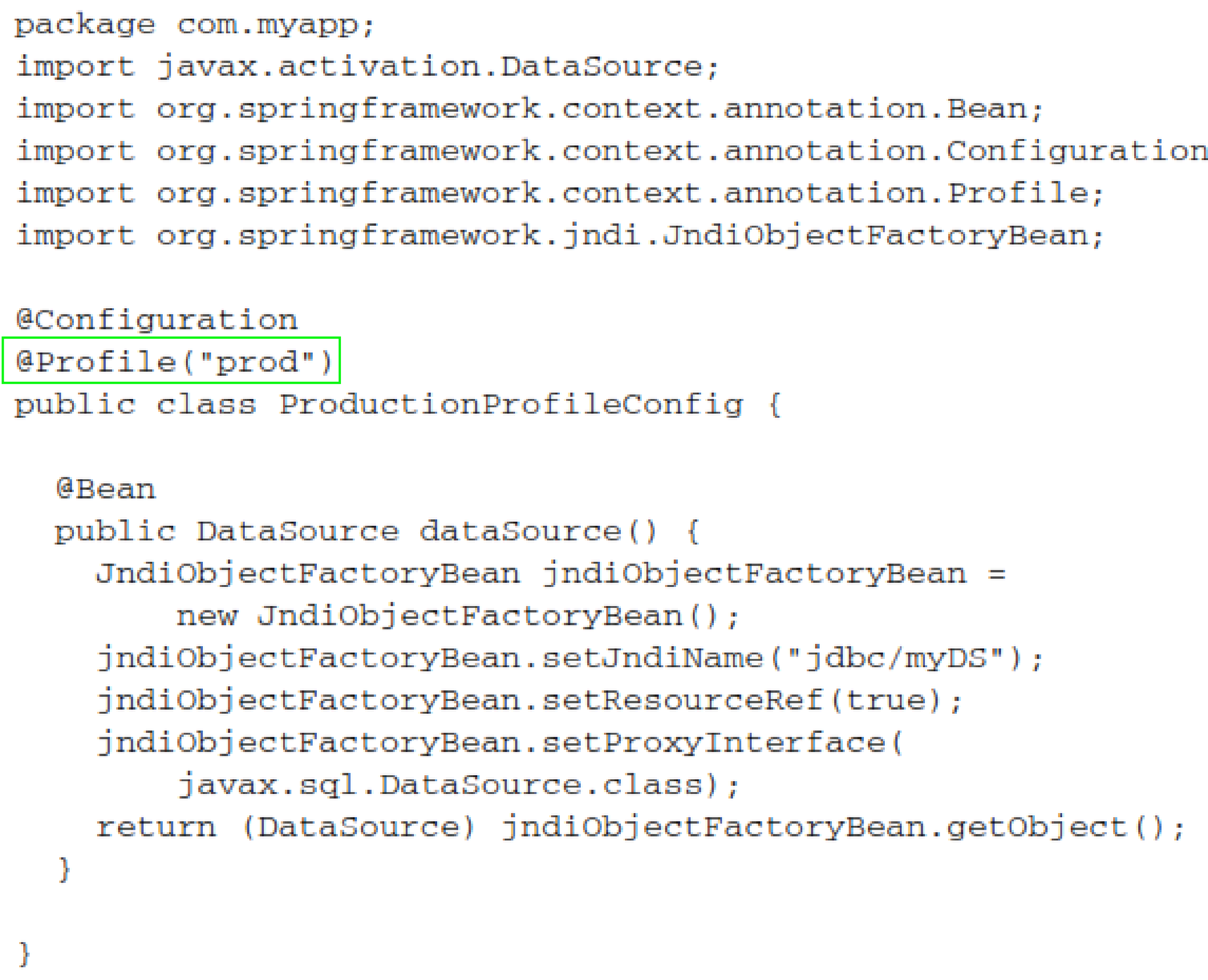


```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:c="http://www.springframework.org/schema/c"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean class="soundsystem.CDConfig" />

  <import resource="cdplayer-config.xml" />

</beans>
```

Advanced Wiring: Environments and profiles

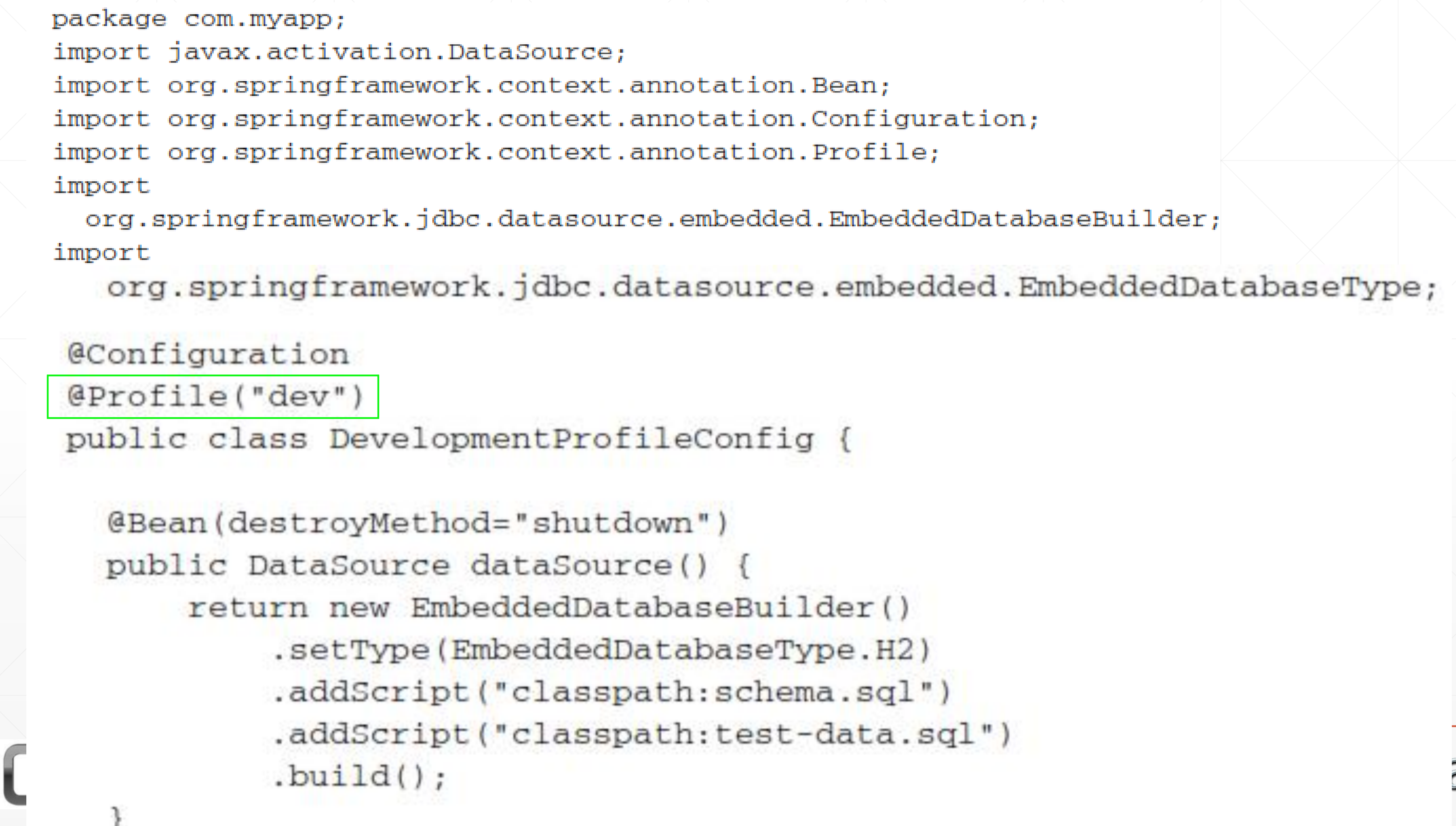
- In Java configuration, you can use the @Profile annotation to specify which profile a bean belongs to

- @Profile annotation applied at the class level: It tells Spring that the beans in this configuration class should be created only if the dev profile is active. If the dev profile isn’t active, then the @Bean methods will be ignored.



```
package com.myapp;
import javax.activation.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration
import org.springframework.context.annotation.Profile;
import org.springframework.jndi.JndiObjectFactoryBean;

@Configuration
@Profile("prod")
public class ProductionProfileConfig {

  @Bean
  public DataSource dataSource() {
    JndiObjectFactoryBean jndiObjectFactoryBean =
        new JndiObjectFactoryBean();
    jndiObjectFactoryBean.setJndiName("jdbc/myDS");
    jndiObjectFactoryBean.setResourceRef(true);
    jndiObjectFactoryBean.setProxyInterface(
        javax.sql.DataSource.class);
    return (DataSource) jndiObjectFactoryBean.getObject();
  }

}
```

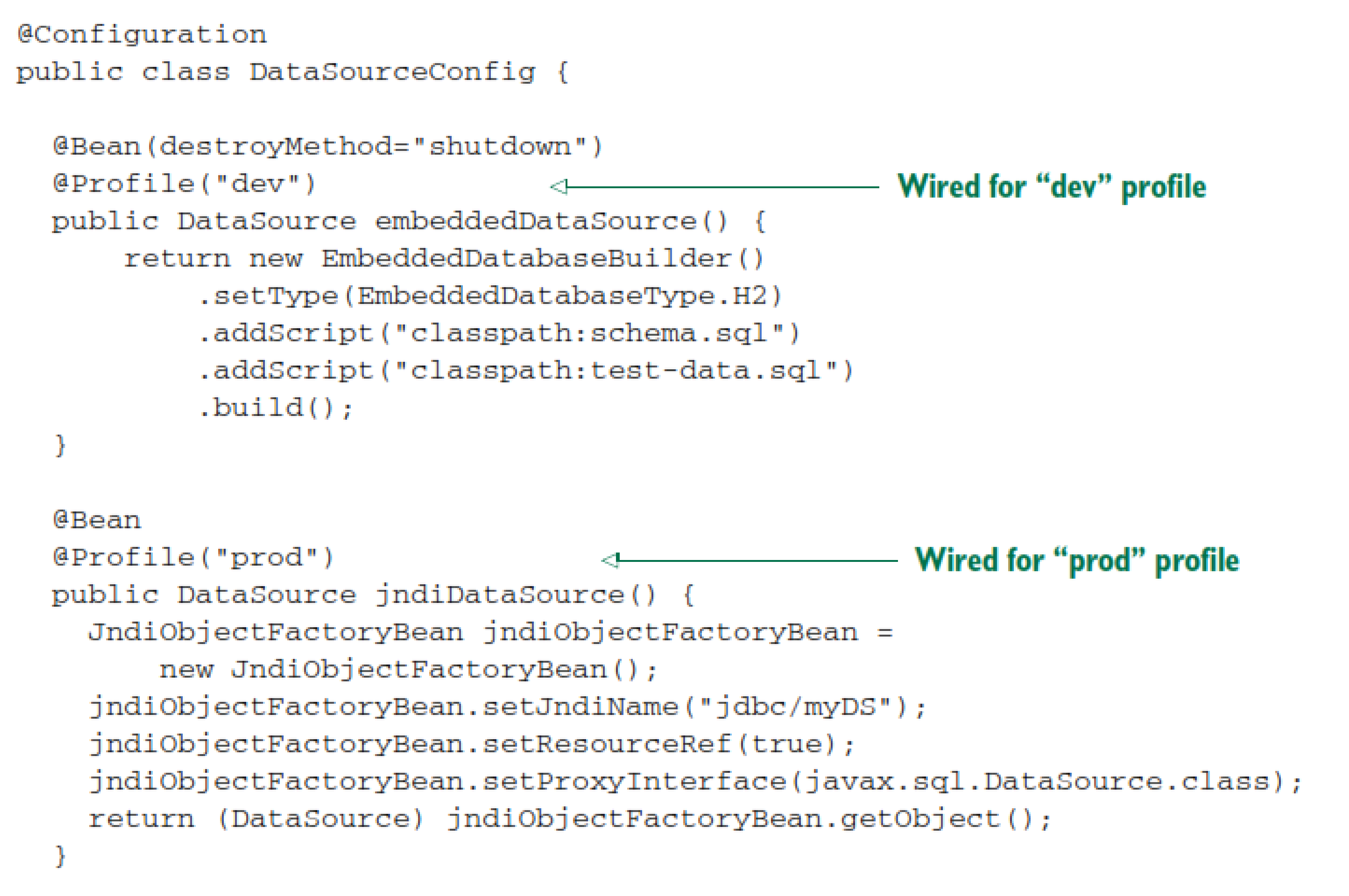
Advanced Wiring: Environments and profiles



```
package com.myapp;
import javax.activation.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import
  org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import
  org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

@Configuration
@Profile("dev")
public class DevelopmentProfileConfig {

  @Bean(destroyMethod="shutdown")
  public DataSource dataSource() {
      return new EmbeddedDatabaseBuilder()
          .setType(EmbeddedDatabaseType.H2)
          .addScript("classpath:schema.sql")
          .addScript("classpath:test-data.sql")
          .build();
  }
```

Advanced Wiring: Environments and profiles

- In Spring 3.1, you could only use the @Profile annotation at the **class level**. Starting with Spring 3.2, you can use @Profile at the **method level**, alongside the @Bean annotation. This makes it possible to combine both bean declarations into a single configuration class



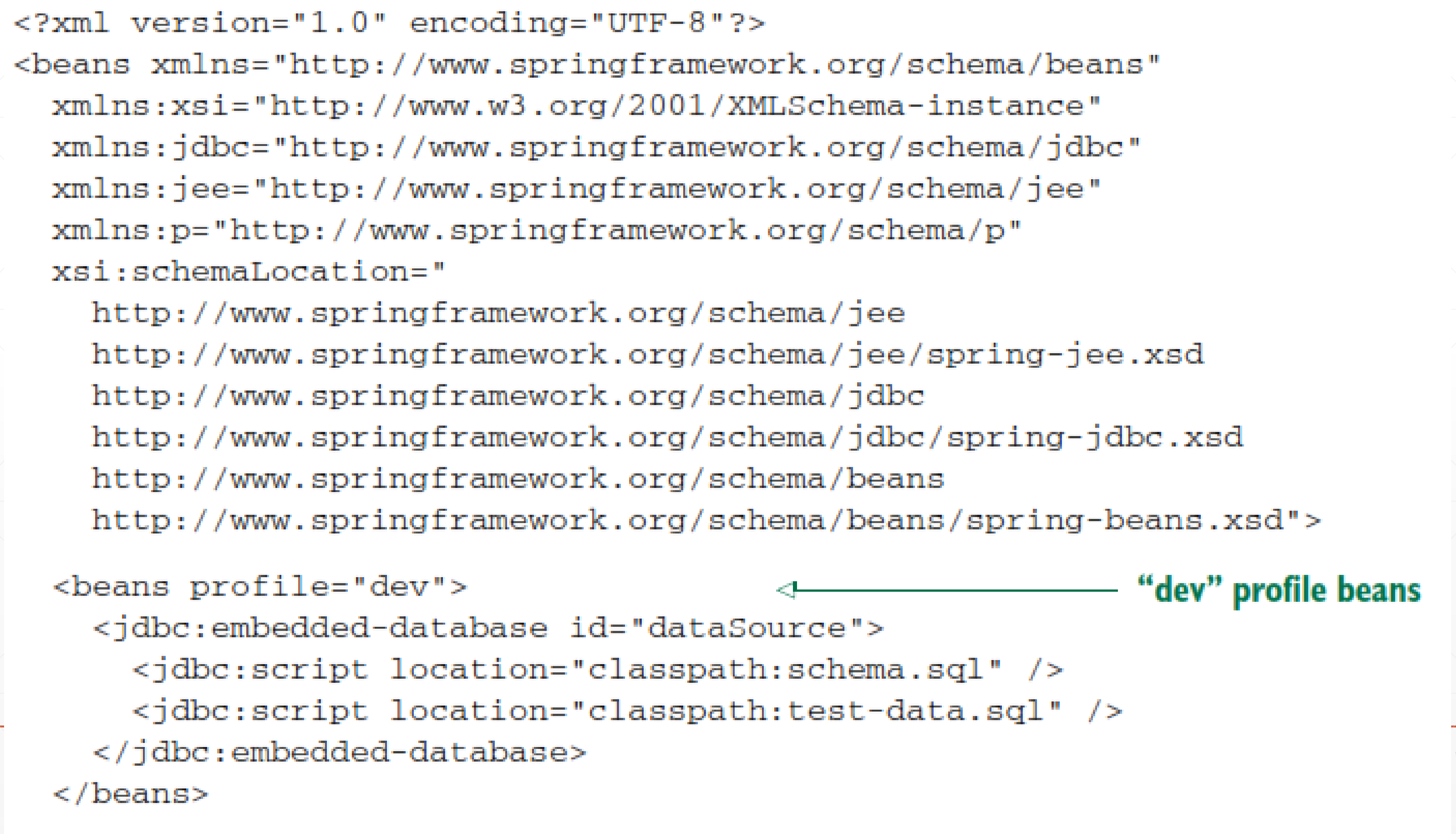
```
@Configuration
public class DataSourceConfig {

  @Bean(destroyMethod="shutdown")
  @Profile("dev")
  public DataSource embeddedDataSource() {
      return new EmbeddedDatabaseBuilder()
          .setType(EmbeddedDatabaseType.H2)
          .addScript("classpath:schema.sql")
          .addScript("classpath:test-data.sql")
          .build();
  }

  @Bean
  @Profile("prod")
  public DataSource jndiDataSource() {
    JndiObjectFactoryBean jndiObjectFactoryBean =
        new JndiObjectFactoryBean();
    jndiObjectFactoryBean.setJndiName("jdbc/myDS");
    jndiObjectFactoryBean.setResourceRef(true);
    jndiObjectFactoryBean.setProxyInterface(javax.sql.DataSource.class);
    return (DataSource) jndiObjectFactoryBean.getObject();
  }
```

Wired for “dev” profile

Wired for “prod” profile

Advanced Wiring: Environments and profiles

- Configuring profiles in XML

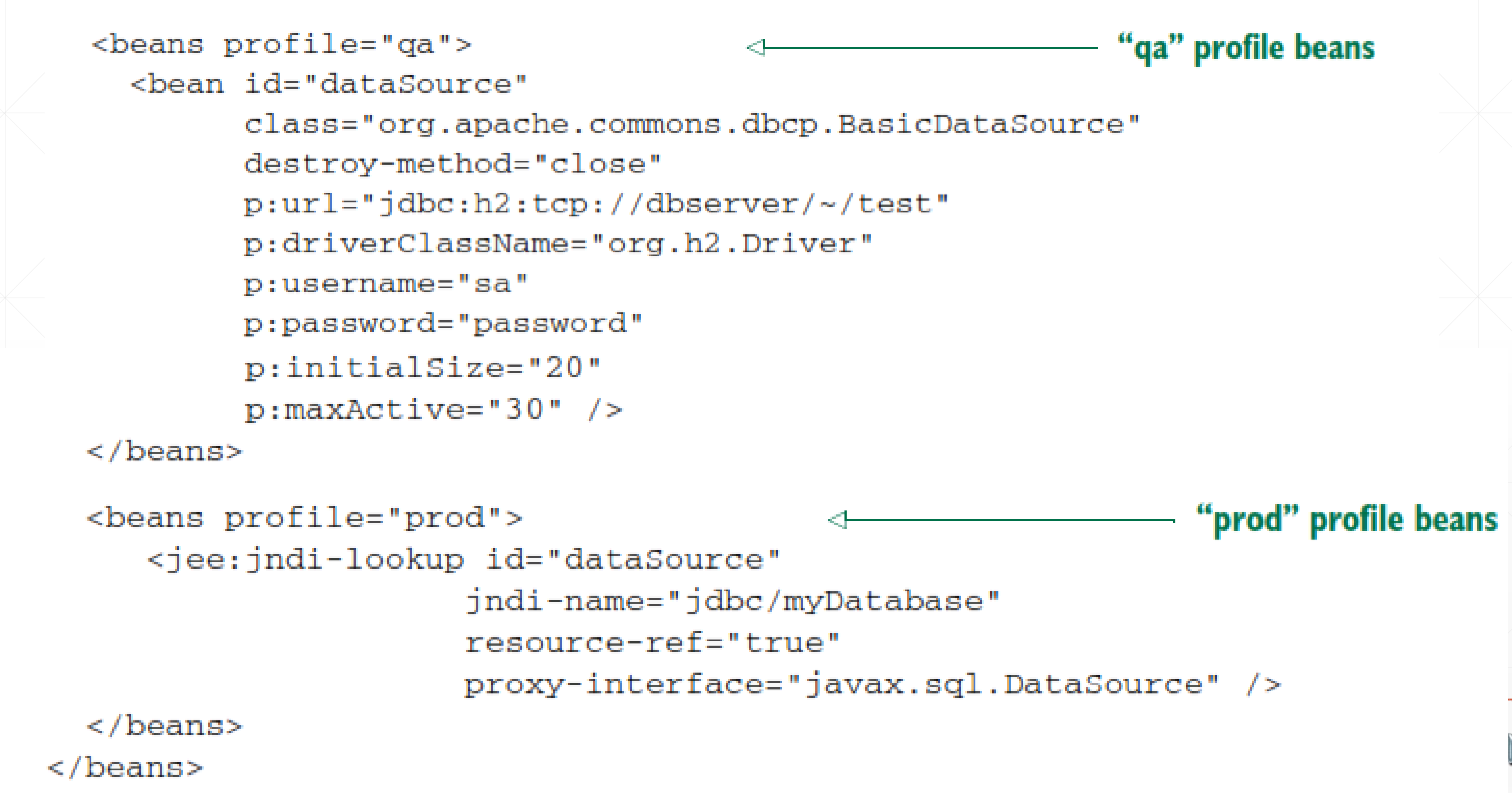


```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jdbc="http://www.springframework.org/schema/jdbc"
  xmlns:jee="http://www.springframework.org/schema/jee"
  xmlns:p="http://www.springframework.org/schema/p"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

  <beans profile="dev">
    <jdbc:embedded-database id="dataSource">
      <jdbc:script location="classpath:schema.sql" />
      <jdbc:script location="classpath:test-data.sql" />
    </jdbc:embedded-database>
  </beans>
```

“dev” profile beans

Advanced Wiring: Environments and profiles

- Configuring profiles in XML



```
  <beans profile="qa">
    <bean id="dataSource"
          class="org.apache.commons.dbcp.BasicDataSource"
          destroy-method="close"
          p:url="jdbc:h2:tcp://dbserver/~/test"
          p:driverClassName="org.h2.Driver"
          p:username="sa"
          p:password="password"
          p:initialSize="20"
          p:maxActive="30" />
  </beans>

  <beans profile="prod">
     <jee:jndi-lookup id="dataSource"
                      jndi-name="jdbc/myDatabase"
                      resource-ref="true"
                      proxy-interface="javax.sql.DataSource" />
  </beans>
</beans>
```

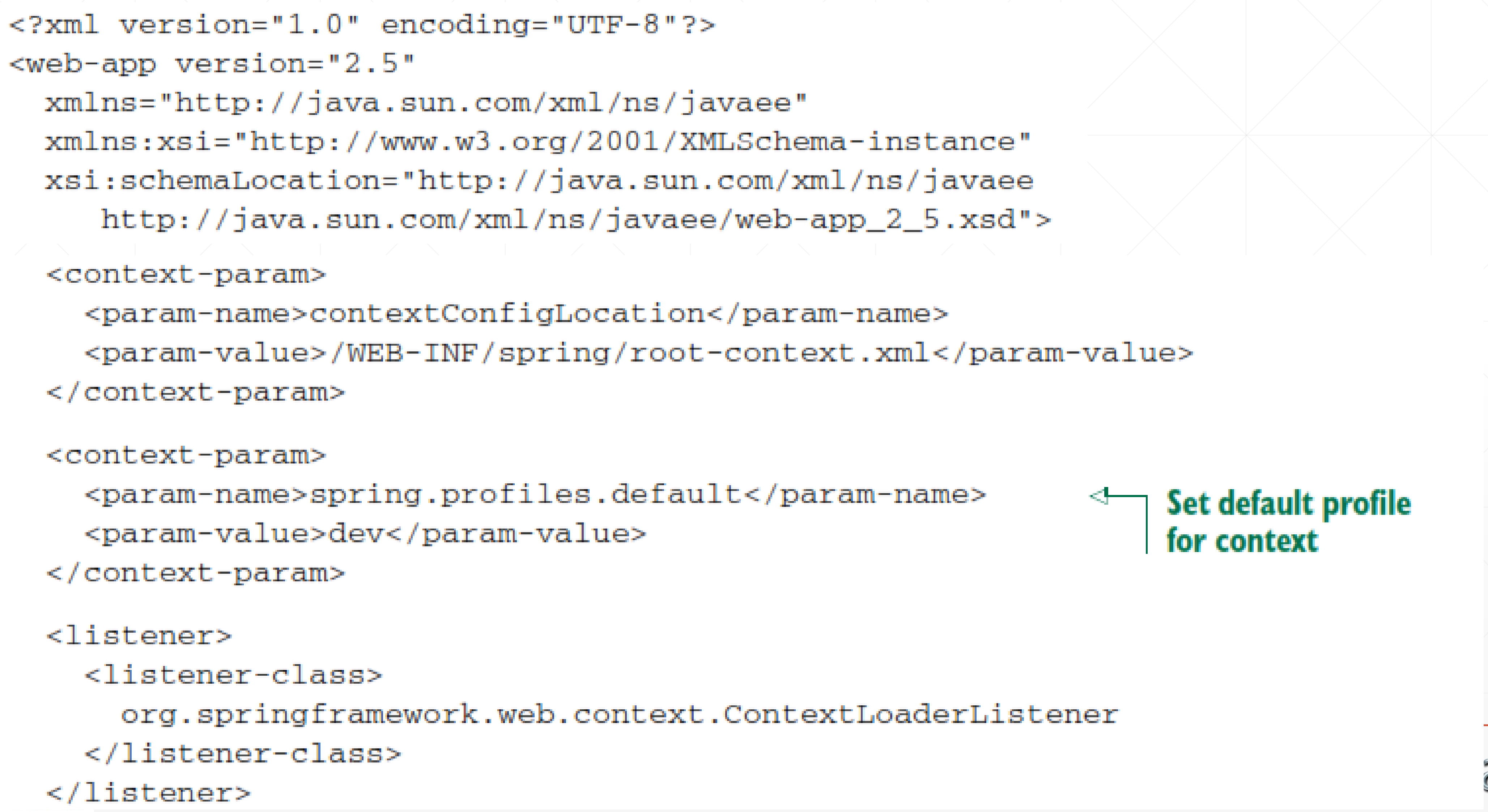
Activating profiles

- Spring honors two separate properties when determining which profiles are active:
  - **spring.profiles.active** and **spring.profiles.default**.
- If **spring.profiles.active** is set, then its value determines which profiles are active. But if spring .profiles.active isn’t set, then Spring looks to **spring.profiles.default**. If neither spring.profiles.active nor spring.profiles.default is set, then there are no active profiles, and only those beans that aren’t defined as being in a profile are created.

Activating profiles

- There are several ways to set these properties:
  - As initialization parameters on DispatcherServlet
  - As context parameters of a web application
  - As JNDI entries
  - As environment variables
  - As JVM system properties
  - Using the @ActiveProfiles annotation on an integration test class

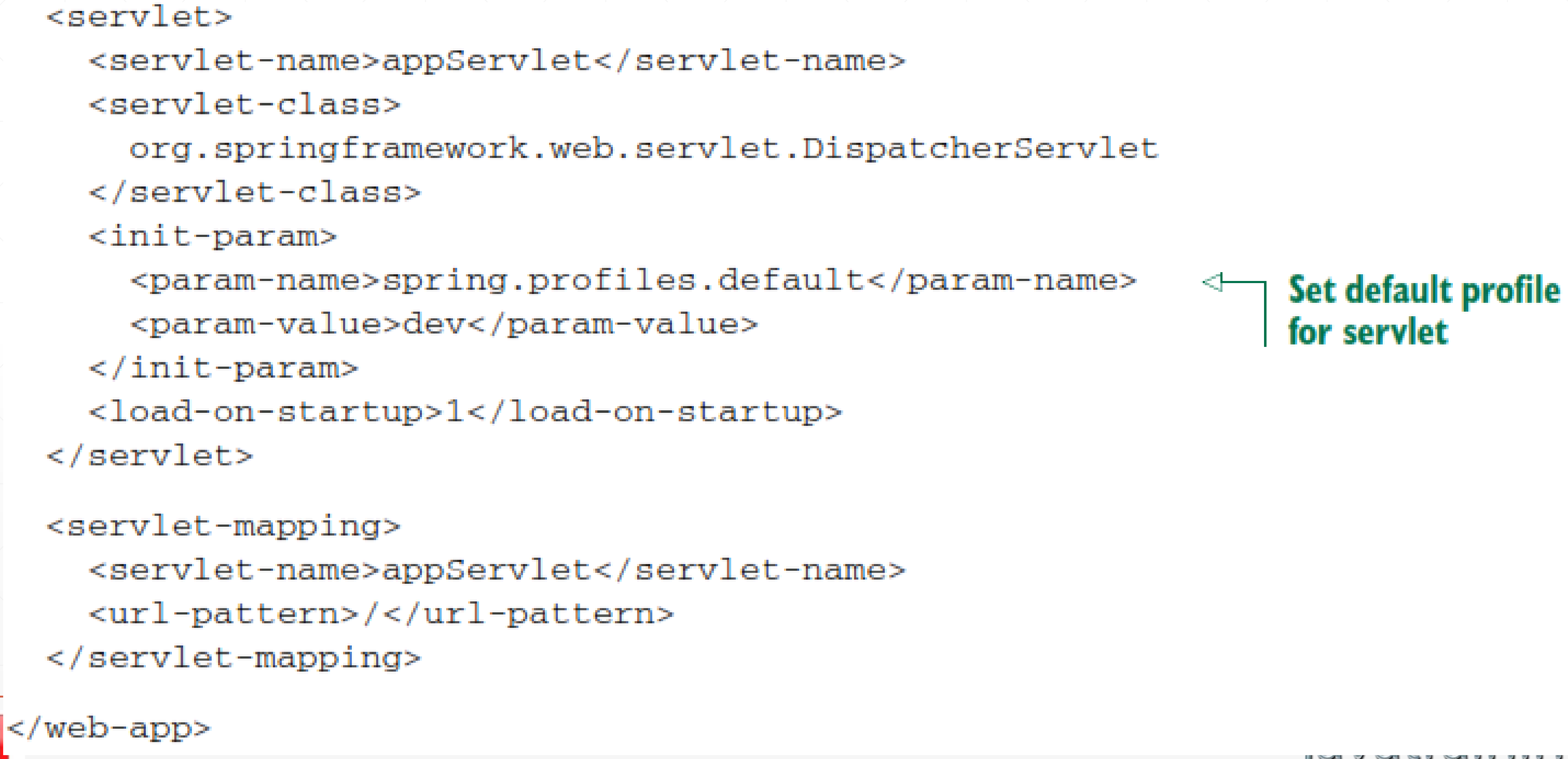
Setting default profiles in a web application web.xml file



```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
     http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/spring/root-context.xml</param-value>
  </context-param>

  <context-param>
    <param-name>spring.profiles.default</param-name>
    <param-value>dev</param-value>
  </context-param>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>
```

Set default profile for context

Setting default profiles in a web application뭘 web.xml file



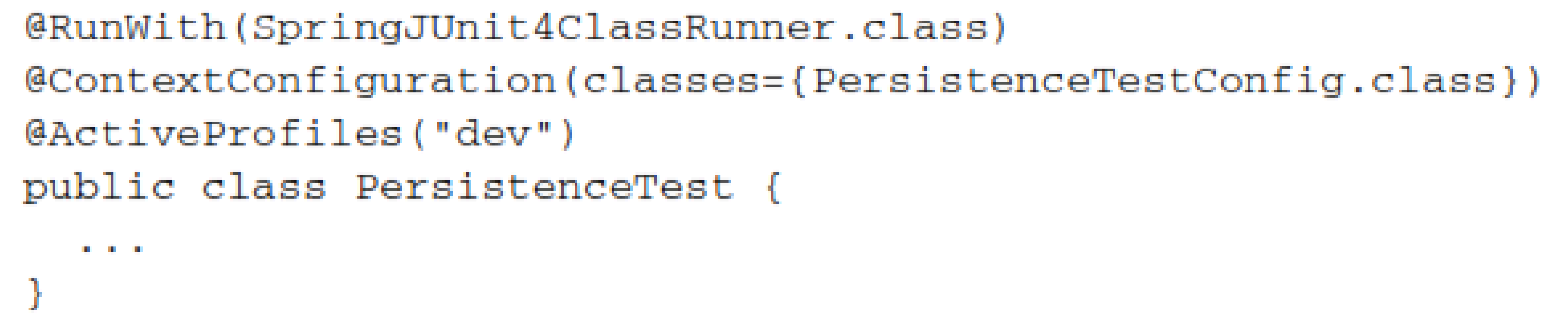
```
<servlet>
  <servlet-name>appServlet</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
    <param-name>spring.profiles.default</param-name>
    <param-value>dev</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>appServlet</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>

</web-app>
```

Set default profile for servlet

Testing with profiles

- Spring offers the @ActiveProfiles annotation to let you specify which profile(s)

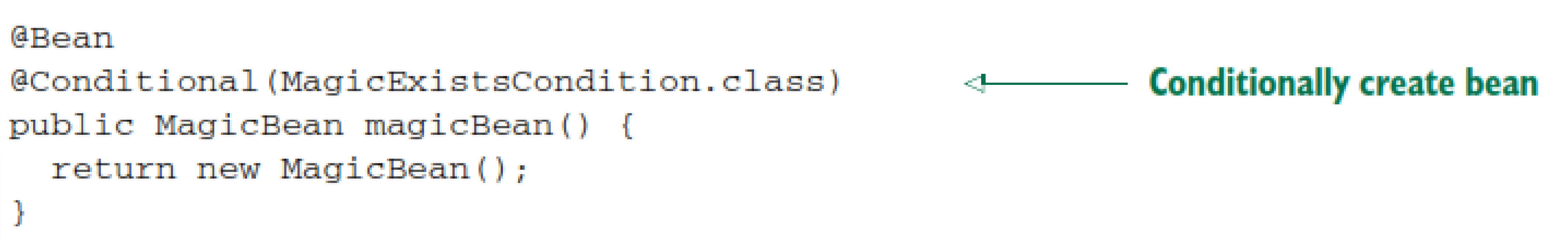
should be active when a test is run.



```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes={PersistenceTestConfig.class})
@ActiveProfiles("dev")
public class PersistenceTest {
   ...
}
```

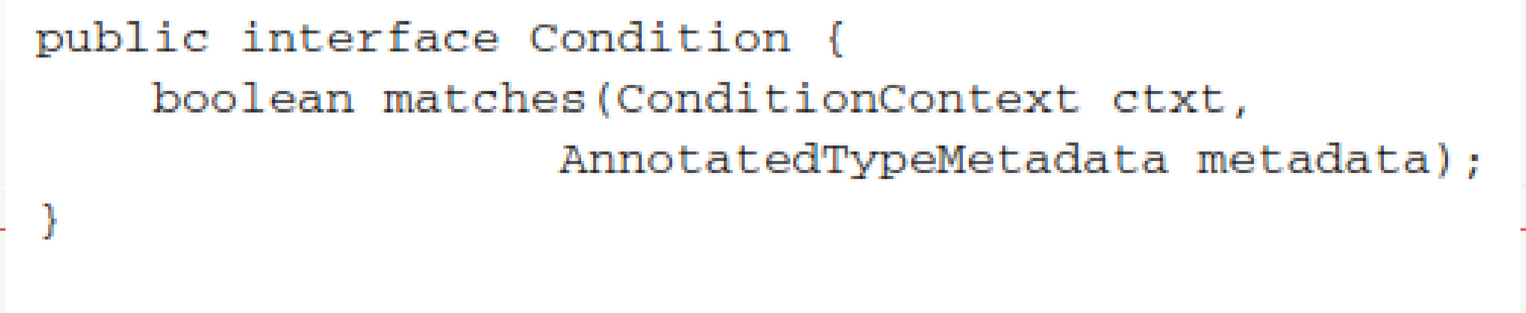
Conditional beans

- E.g. if you want beans to be configured only if some libraries are in the classpath or some environment variables are set
- Spring 4 introduces **@Conditional** annotation – only if the condition is true the bean will be created



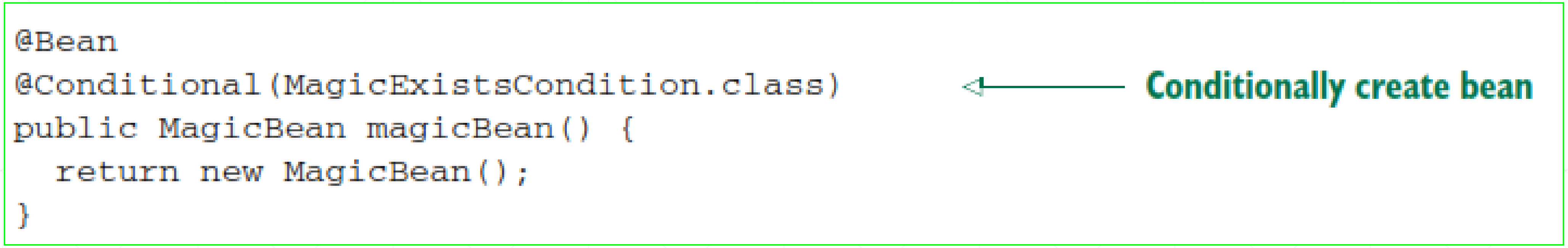
```
@Bean
@Conditional(MagicExistsCondition.class)
public MagicBean magicBean() {
  return new MagicBean();
}
```

Conditionally create bean



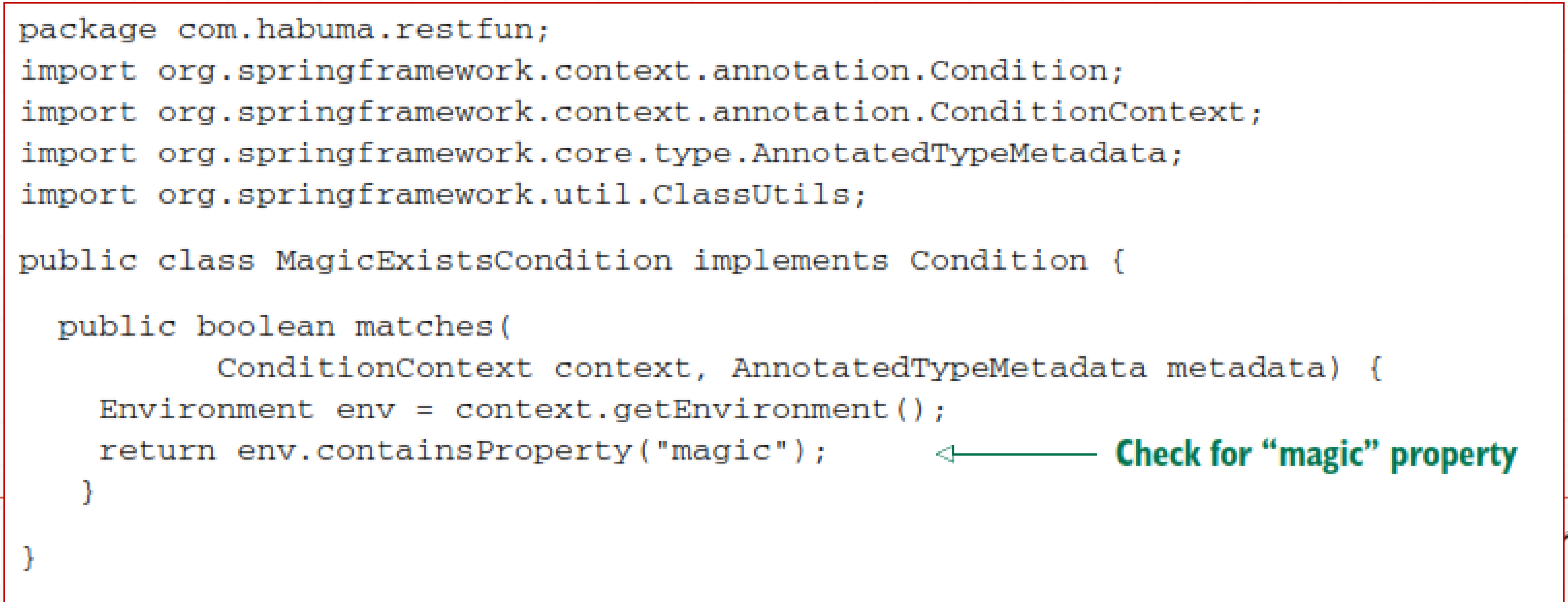
```
public interface Condition {
    boolean matches(ConditionContext ctxt,
                    AnnotatedTypeMetadata metadata);
}
```

Conditional beans



```
@Bean
@Conditional(MagicExistsCondition.class)
public MagicBean magicBean() {
  return new MagicBean();
}
```

Conditionally create bean

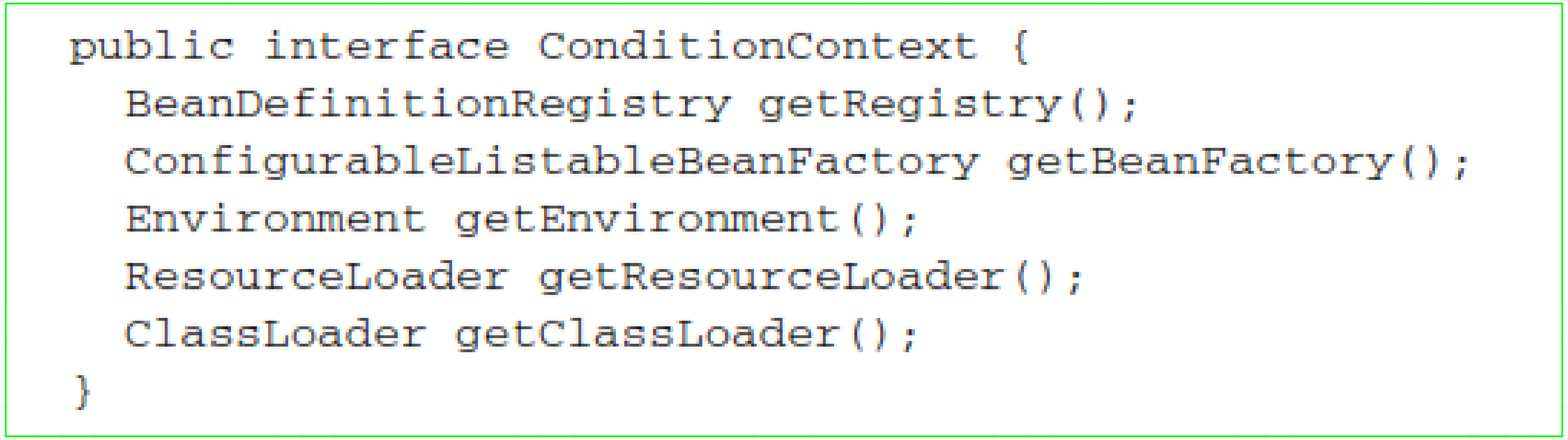


```
package com.habuma.restfun;
import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext;
import org.springframework.core.type.AnnotatedTypeMetadata;
import org.springframework.util.ClassUtils;

public class MagicExistsCondition implements Condition {

  public boolean matches(
          ConditionContext context, AnnotatedTypeMetadata metadata) {
    Environment env = context.getEnvironment();
    return env.containsProperty("magic");
   }

}
```

Check for “magic” property

Conditional beans

The matches() method is given a ConditionContext and an AnnotatedTypeMetadata to use in making its decision:



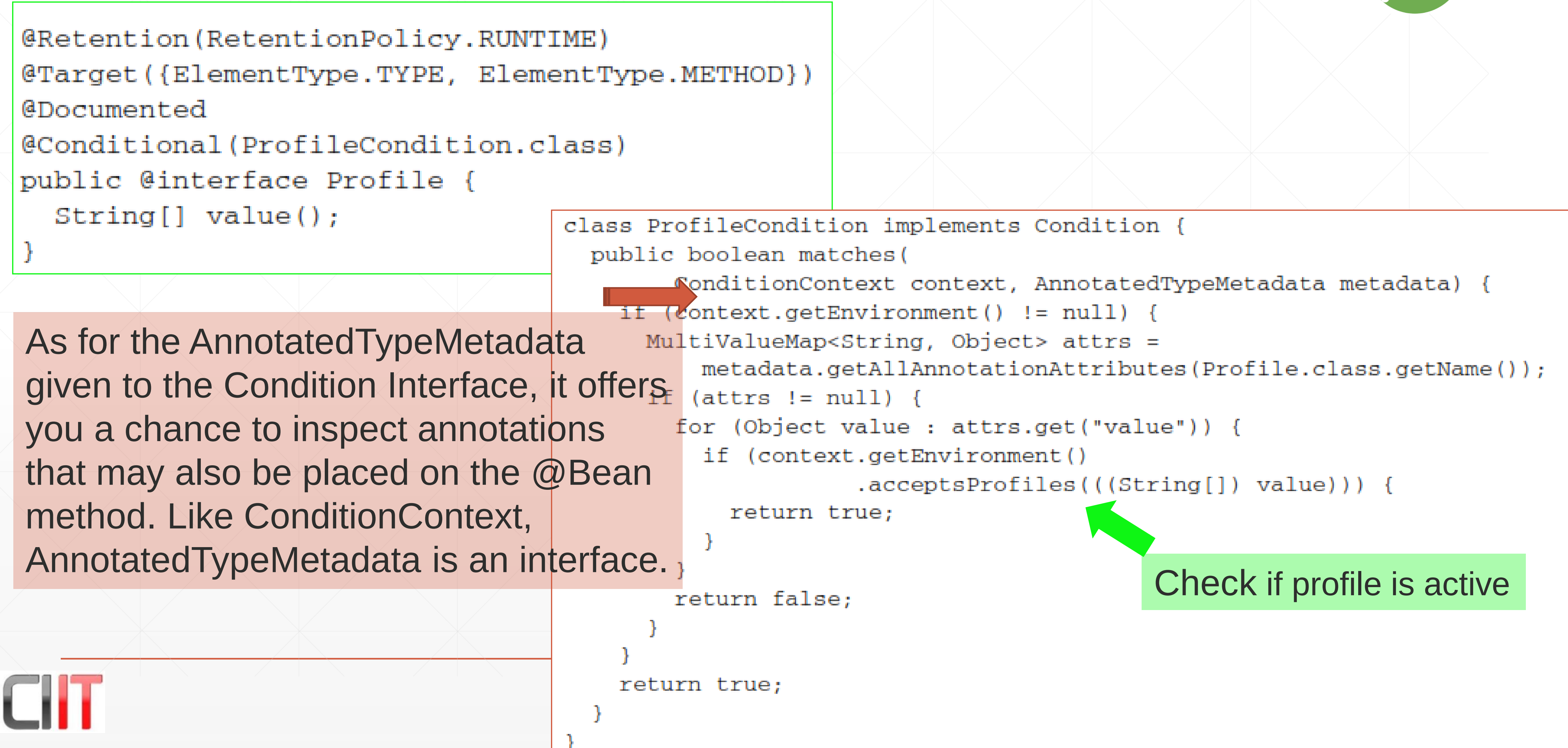
```
public interface ConditionContext {
  BeanDefinitionRegistry getRegistry();
  ConfigurableListableBeanFactory getBeanFactory();
  Environment getEnvironment();
  ResourceLoader getResourceLoader();
  ClassLoader getClassLoader();
}
```

Conditional beans

From the ConditionContext, you can do the following:

- Check for bean definitions via the BeanDefinitionRegistry returned from getRegistry().
- Check for the presence of beans, and even dig into bean properties via the ConfigurableListableBeanFactory returned from getBeanFactory().
- Check for the presence and values of environment variables via the Environment retrieved from getEnvironment().
- Read and inspect the contents of resources loaded via the ResourceLoader returned from getResourceLoader().
- Load and check for the presence of classes via the ClassLoader returned from getClassLoader().

Profile Condition



```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
@Documented
@Conditional(ProfileCondition.class)
public @interface Profile {
  String[] value();
}
```

```
class ProfileCondition implements Condition {
  public boolean matches(
      ConditionContext context, AnnotatedTypeMetadata metadata) {
    if (context.getEnvironment() != null) {
      MultiValueMap<String, Object> attrs =
          metadata.getAllAnnotationAttributes(Profile.class.getName());
      if (attrs != null) {
        for (Object value : attrs.get("value")) {
          if (context.getEnvironment()
                      .acceptsProfiles(((String[]) value))) {
            return true;
          }
        }
        return false;
      }
    }
    return true;
  }
}
```

As for the AnnotatedTypeMetadata given to the Condition Interface, it offers you a chance to inspect annotations that may also be placed on the @Bean method. Like ConditionContext, AnnotatedTypeMetadata is an interface.

Check if profile is active

Addressing ambiguity in autowiring

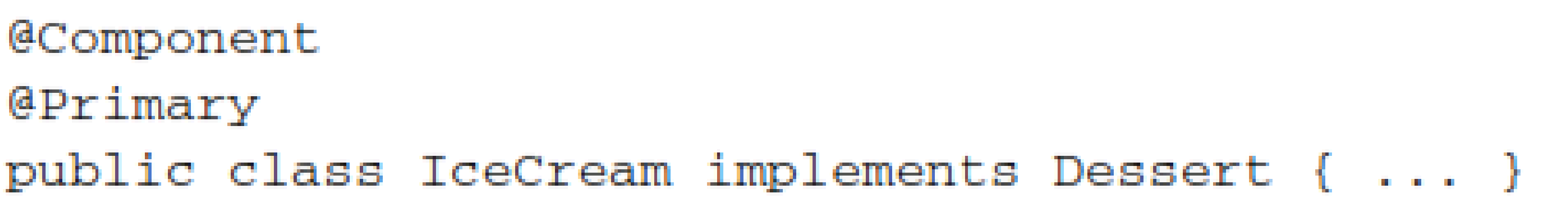
```
@Autowired
public void setDessert(Dessert dessert) {
    this.dessert = dessert;
}
```

```
@Component
public class Cake implements Dessert { ... }

@Component
public class Cookies implements Dessert { ... }
@Component
public class IceCream implements Dessert { ... }
```

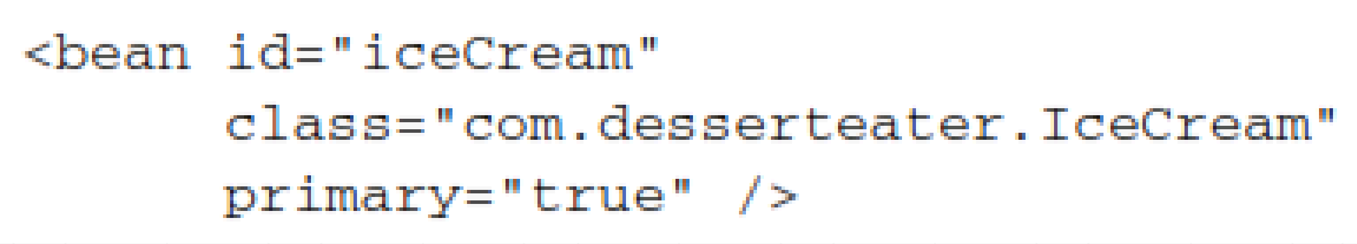
```
nested exception is
  org.springframework.beans.factory.NoUniqueBeanDefinitionException:
No qualifying bean of type [com.desserteater.Dessert] is defined:
  expected single matching bean but found 3: cake,cookies,iceCream
```

Addressing ambiguity in autowiring

- Designating a primary bean – perfect as long you define only one @Primary bean



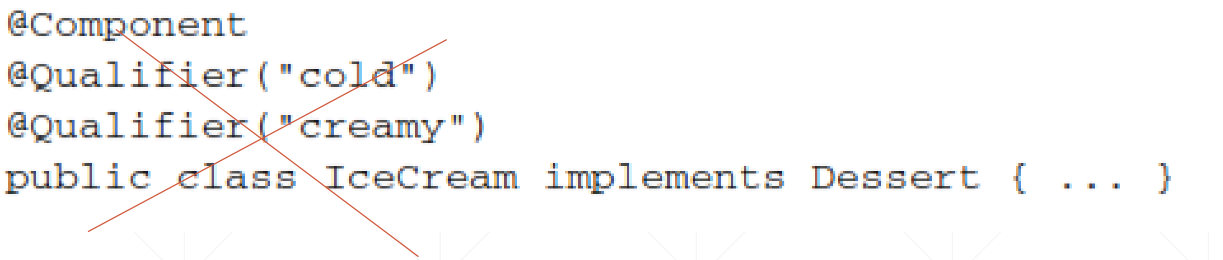
```
@Component
@Primary
public class IceCream implements Dessert { ... }
```



```
<bean id="iceCream"
      class="com.desserteater.IceCream"
      primary="true" />
```

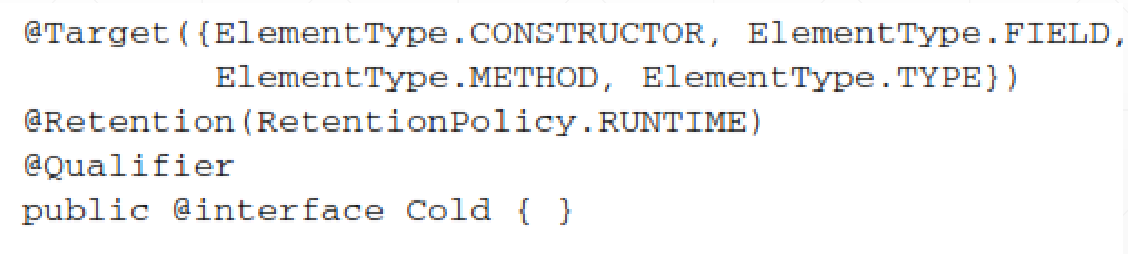
Addressing ambiguity in autowiring

- Using Qualifier – it is not allowed in Java to use more annotations of the same type on the same element

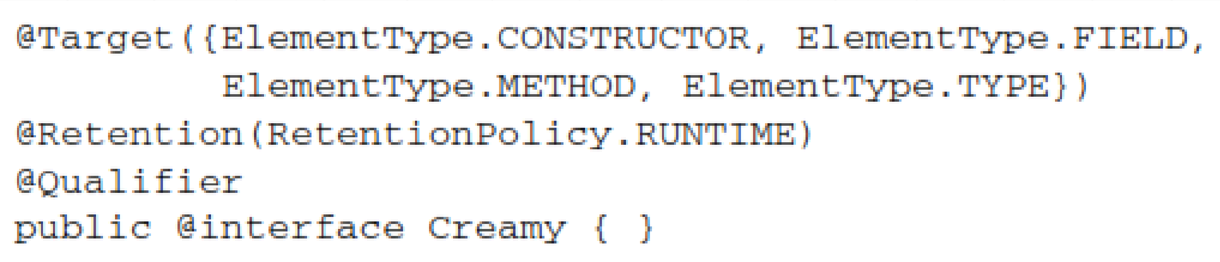


```
@Component
@Qualifier("cold")
@Qualifier("creamy")
public class IceCream implements Dessert { ... }
```

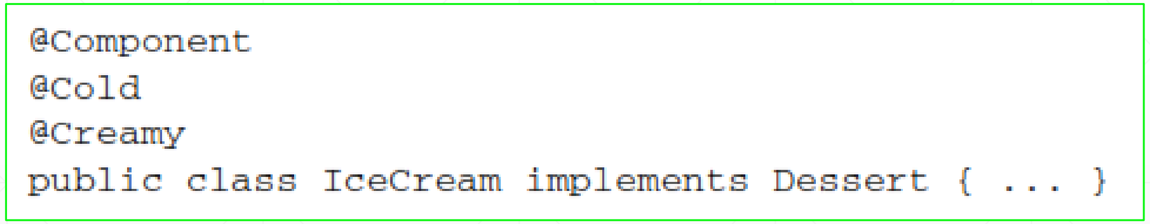
- Defining custom Qualifier:



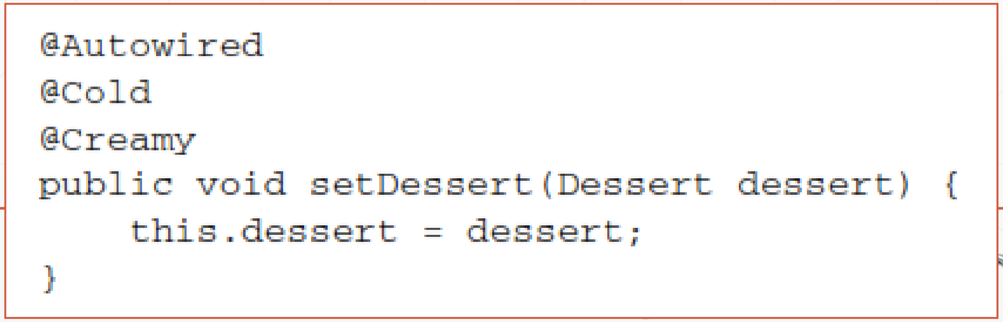
```
@Target({ElementType.CONSTRUCTOR, ElementType.FIELD,
         ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Cold { }
```



```
@Target({ElementType.CONSTRUCTOR, ElementType.FIELD,
         ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Creamy { }
```



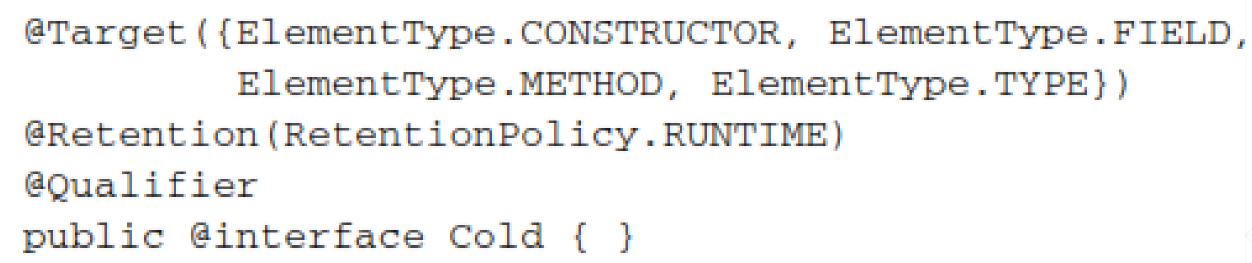
```
@Component
@Cold
@Creamy
public class IceCream implements Dessert { ... }
```



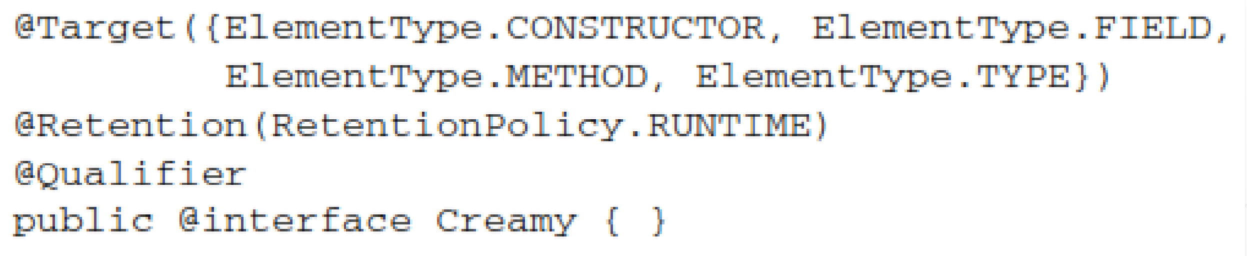
```
@Autowired
@Cold
@Creamy
public void setDessert(Dessert dessert) {
    this.dessert = dessert;
}
```

Addressing ambiguity in autowiring

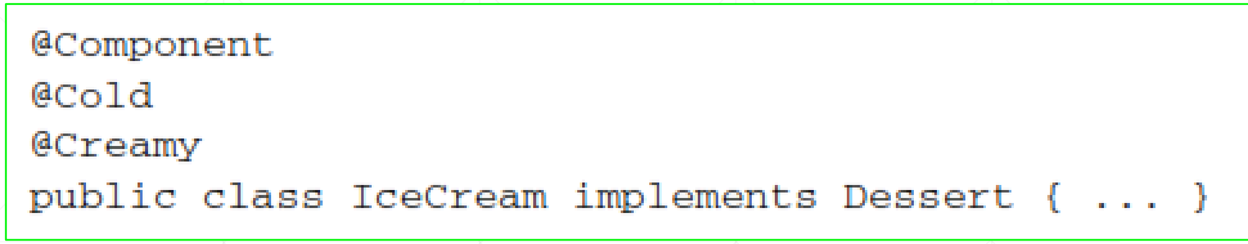
- Defining custom Qualifier:



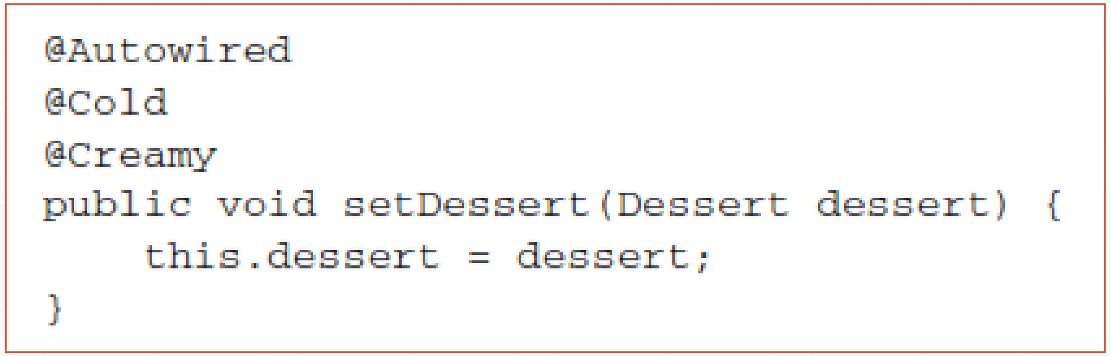
```
@Target({ElementType.CONSTRUCTOR, ElementType.FIELD,
         ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Cold { }
```



```
@Target({ElementType.CONSTRUCTOR, ElementType.FIELD,
         ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Creamy { }
```



```
@Component
@Cold
@Creamy
public class IceCream implements Dessert { ... }
```

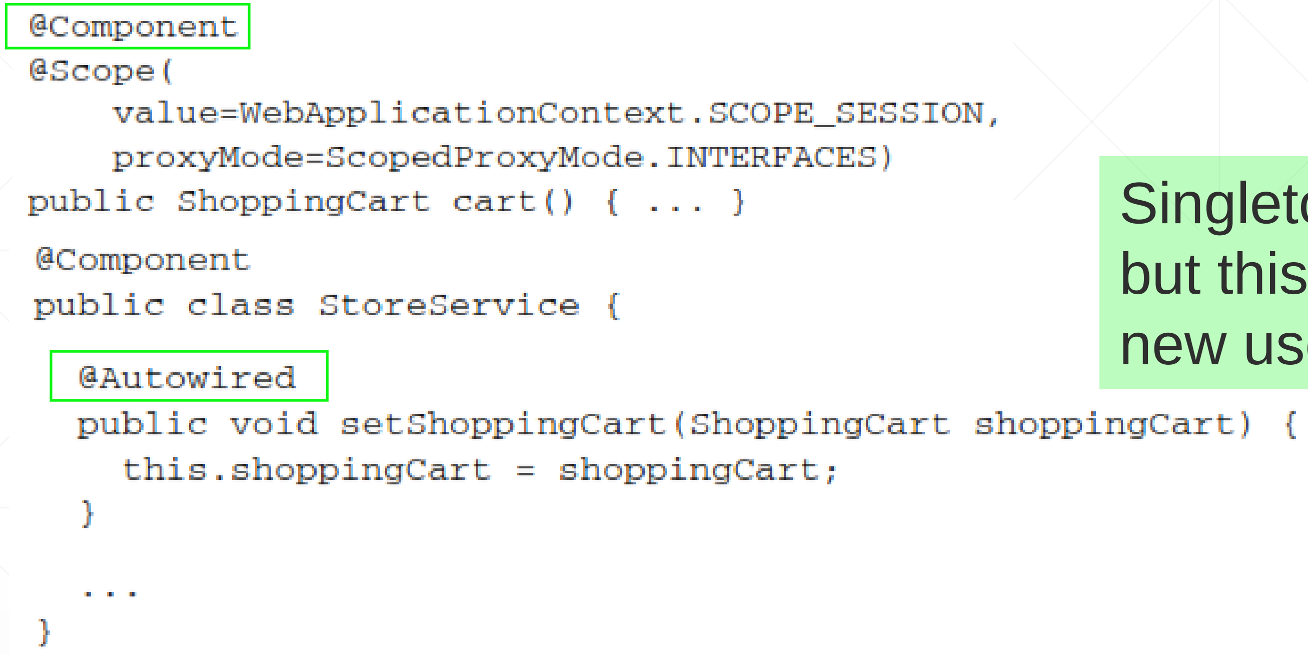


```
@Autowired
@Cold
@Creamy
public void setDessert(Dessert dessert) {
    this.dessert = dessert;
}
```

Scoping beans

- Spring defines several scopes under which a bean can be created, including the following:
  - ***Singleton*** - *One instance of the bean is created for the entire application.*
  - ***Prototype*** - *One instance of the bean is created every time the bean is injected* into or retrieved from the Spring application context.
  - ***Session*** - *In a web application, one instance of the bean is created for each session.*
  - ***Request*** - *In a web application, one instance of the bean is created for each* request.

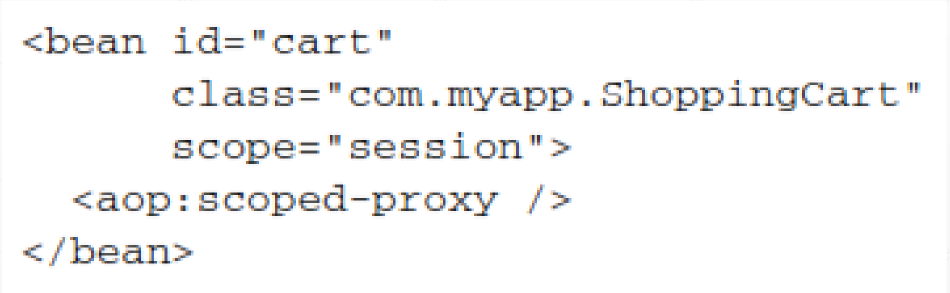
Working with request and session scope



```
@Component
@Scope(
    value=WebApplicationContext.SCOPE_SESSION,
    proxyMode=ScopedProxyMode.INTERFACES)
public ShoppingCart cart() { ... }

@Component
public class StoreService {

  @Autowired
  public void setShoppingCart(ShoppingCart shoppingCart) {
    this.shoppingCart = shoppingCart;
  }

  ...
}
```

Singleton Storeservice needs a shoping card bean but this is session scoped – created with every new user

Creating a proxy:



```
<bean id="cart"
      class="com.myapp.ShoppingCart"
      scope="session">
  <aop:scoped-proxy />
</bean>
```

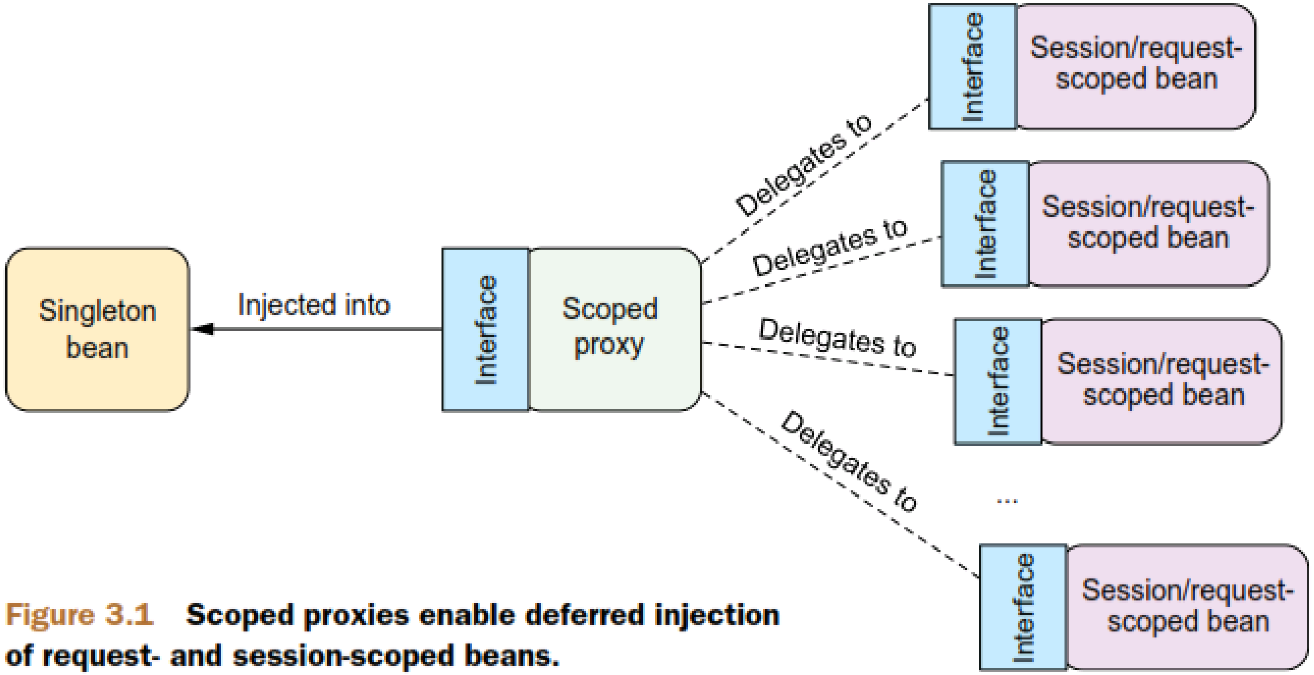
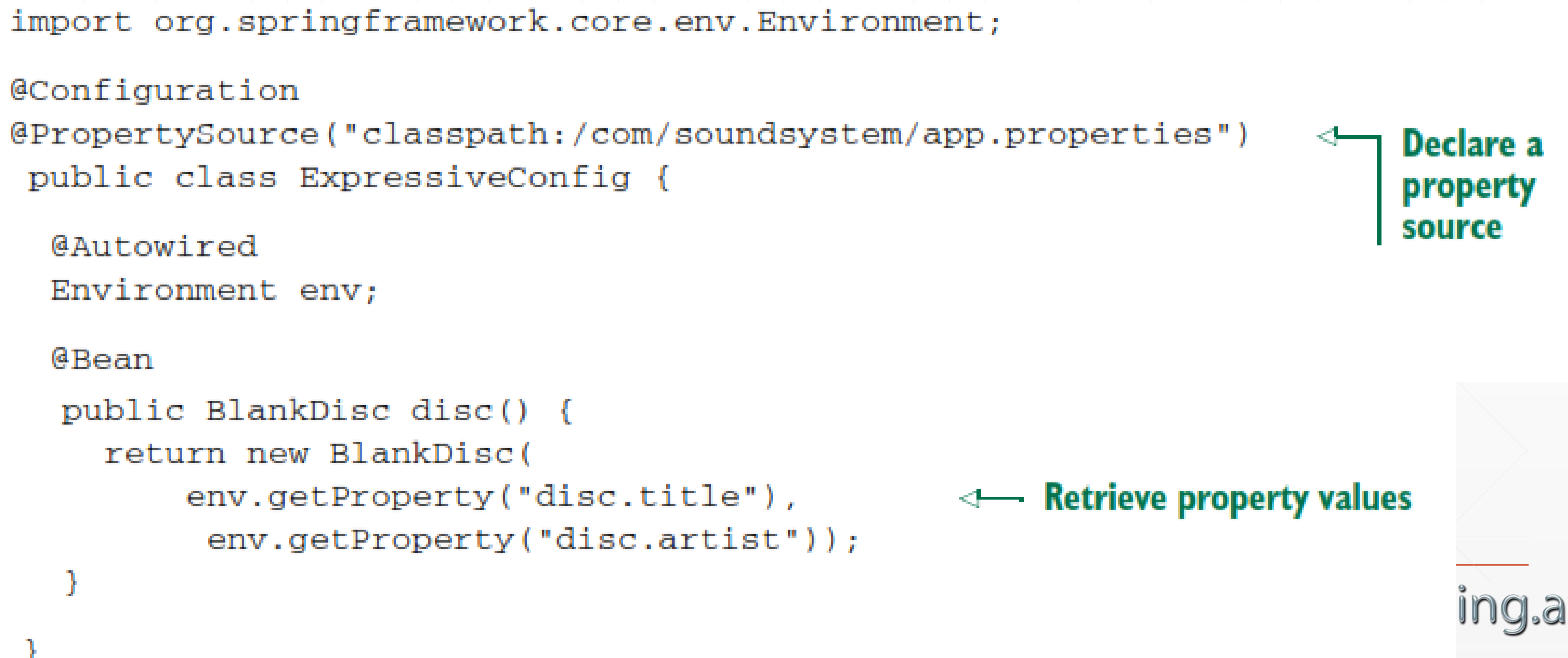


Figure 3.1 **Scoped proxies enable deferred injection of request- and session-scoped beans.**

Runtime value injection

- to avoid hard-coded values and to let the values be determined at runtime

➢ Using the @PropertySource annotation and Environment

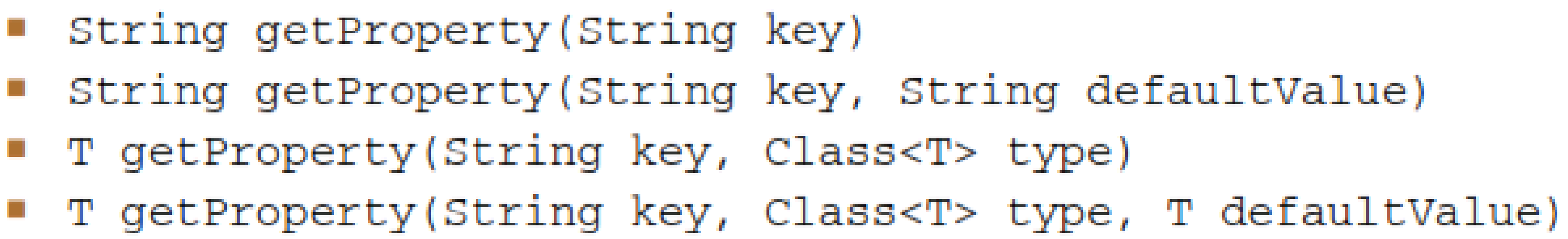


```
import org.springframework.core.env.Environment;

@Configuration
@PropertySource("classpath:/com/soundsystem/app.properties")
 public class ExpressiveConfig {

  @Autowired
  Environment env;

  @Bean
   public BlankDisc disc() {
     return new BlankDisc(
         env.getProperty("disc.title"),
          env.getProperty("disc.artist"));
   }
 }
```

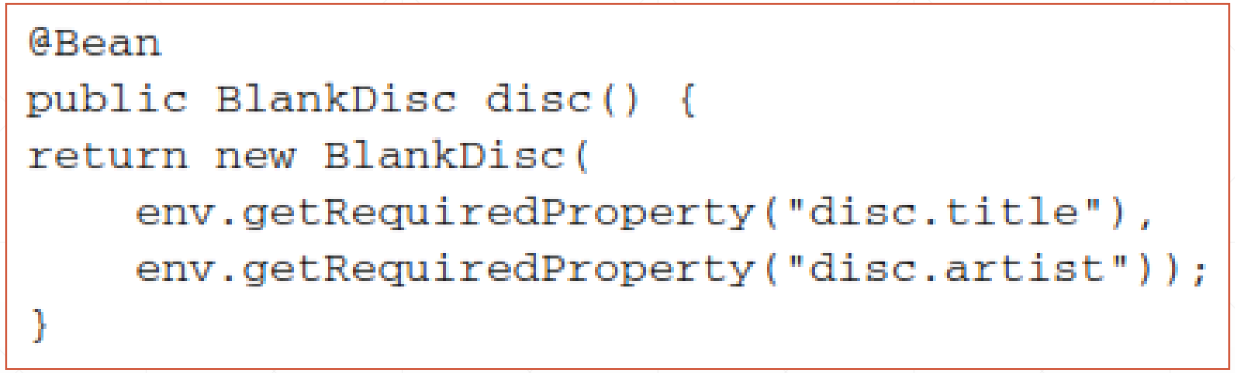
Declare a property source

Retrieve property values

Spring Environment

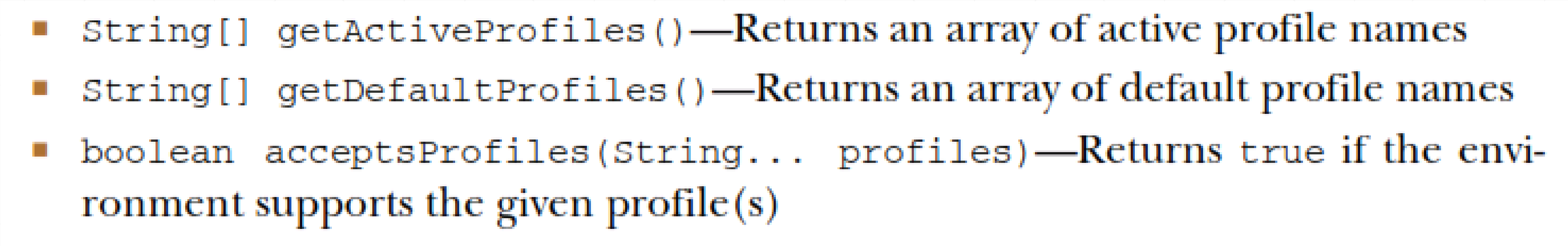


- String getProperty(String key)
- String getProperty(String key, String defaultValue)
- T getProperty(String key, Class<T> type)
- T getProperty(String key, Class<T> type, T defaultValue)



```
@Bean
public BlankDisc disc() {
return new BlankDisc(
    env.getRequiredProperty("disc.title"),
    env.getRequiredProperty("disc.artist"));
}
```

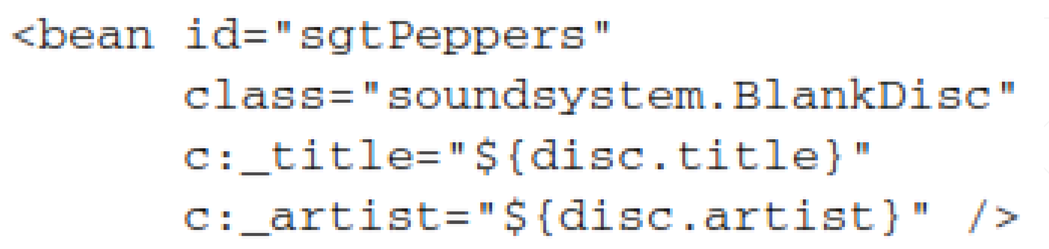
→ IllegalStateException



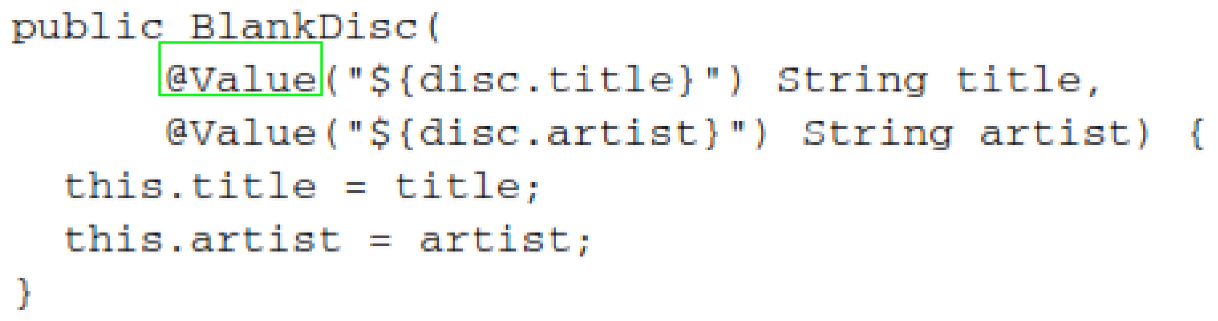
- String[] getActiveProfiles()—Returns an array of active profile names
- String[] getDefaultProfiles()—Returns an array of default profile names
- boolean acceptsProfiles(String... profiles)—Returns true if the environment supports the given profile(s)

Resolving property placeholder

- Using external properties-file you can define values without hardcoding in XML files or using @Value annotation

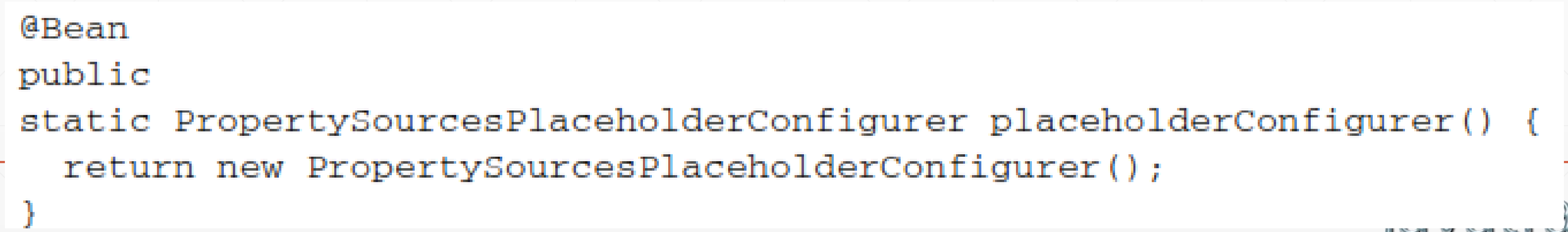


```
<bean id="sgtPeppers"
      class="soundsystem.BlankDisc"
      c:_title="${disc.title}"
      c:_artist="${disc.artist}" />
```



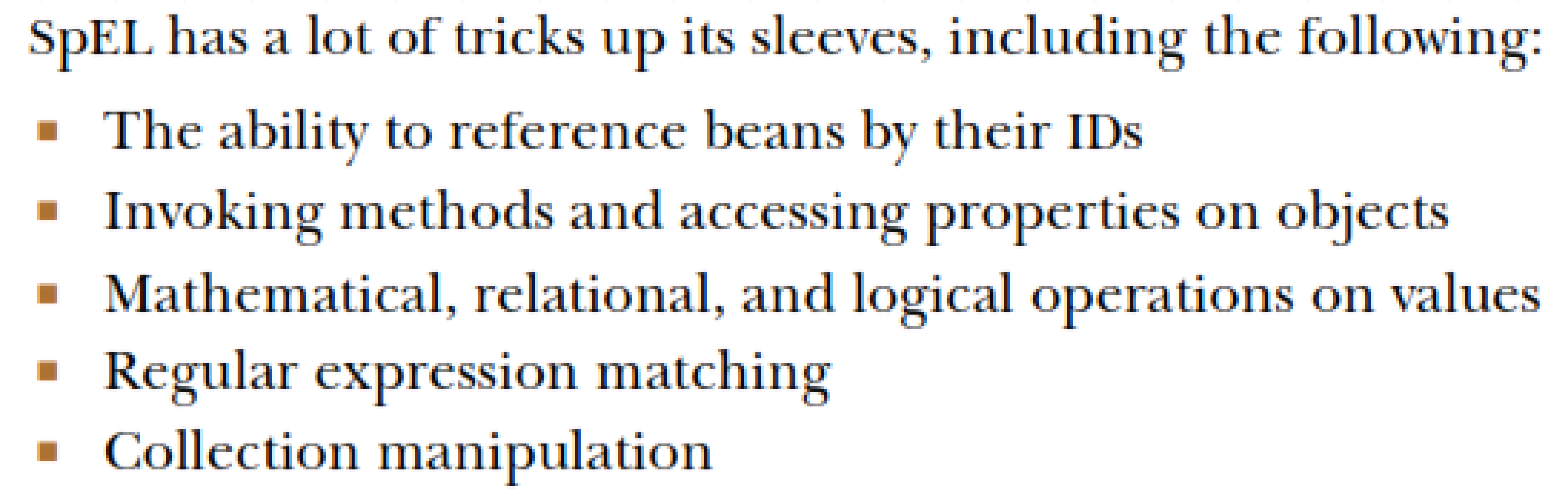
```
public BlankDisc(
      @Value("${disc.title}") String title,
      @Value("${disc.artist}") String artist) {
  this.title = title;
  this.artist = artist;
}
```

- In order to use placeholder values, you must configure either a PropertyPlaceholderConfigurer bean or a PropertySourcesPlaceholderConfigurer bean.



```
@Bean
public
static PropertySourcesPlaceholderConfigurer placeholderConfigurer() {
  return new PropertySourcesPlaceholderConfigurer();
}
```

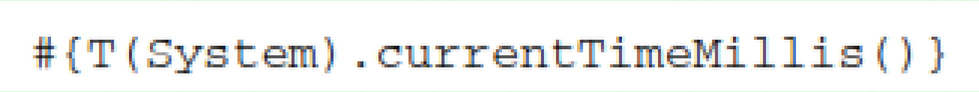
Wiring with the Spring Expression Language



SpEL has a lot of tricks up its sleeves, including the following:

- The ability to reference beans by their IDs
- Invoking methods and accessing properties on objects
- Mathematical, relational, and logical operations on values
- Regular expression matching
- Collection manipulation

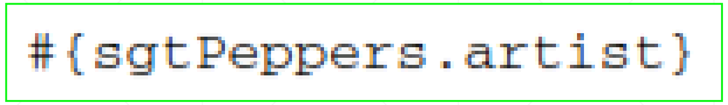
An example:



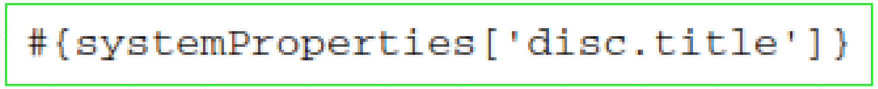
```
#{T(System).currentTimeMillis()}
```

The T() operator evaluates java.lang.System as a type so that the staticcurrentTimeMillis() method can be invoked.

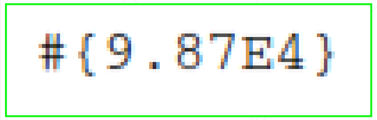
Some SpEL examples



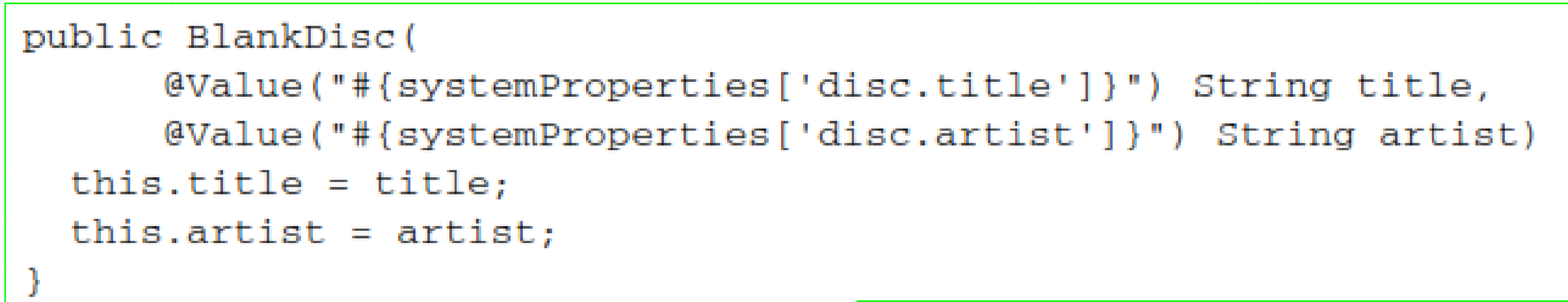
```
#{sgtPeppers.artist}
```



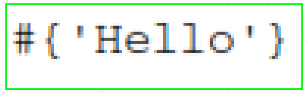
```
#{systemProperties['disc.title']}
```



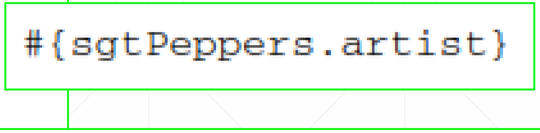
```
#{9.87E4}
```



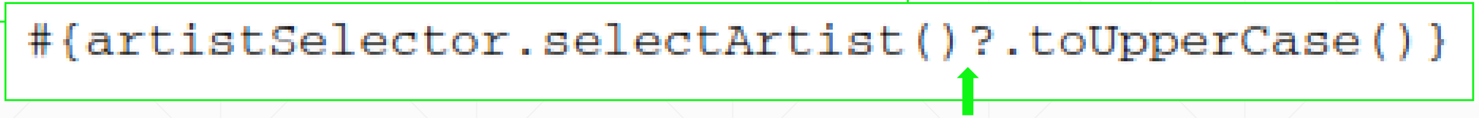
```
public BlankDisc(
      @Value("#{systemProperties['disc.title']}") String title,
      @Value("#{systemProperties['disc.artist']}") String artist) {
  this.title = title;
  this.artist = artist;
}
```



```
#{'Hello'}
```

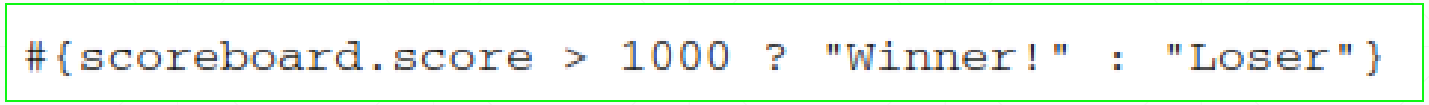


```
#{sgtPeppers.artist}
```

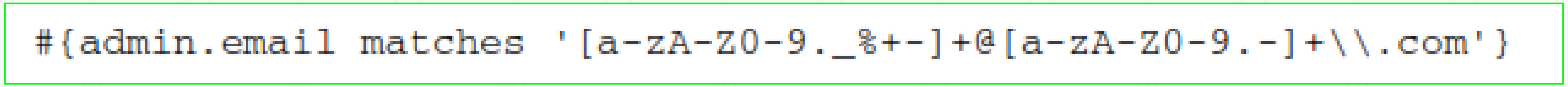


```
#{artistSelector.selectArtist()?.toUpperCase()}
```

Safe against NullPointerException



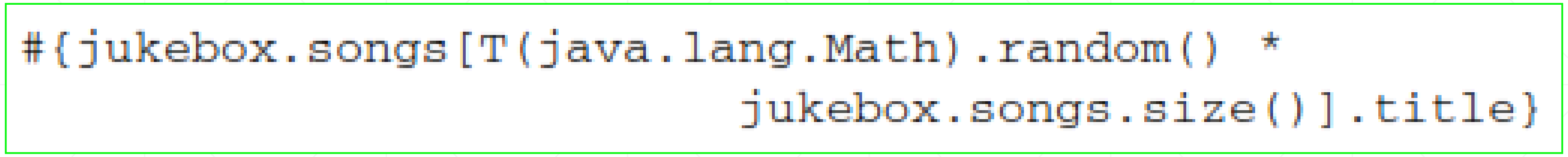
```
#{scoreboard.score > 1000 ? "Winner!" : "Loser"}
```



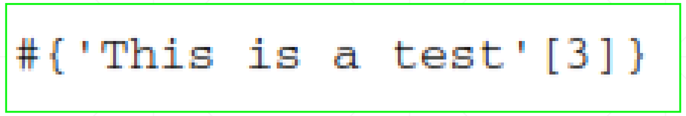
```
#{admin.email matches '[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.com'}
```

Regular Expressions

SpEL examples - Collections

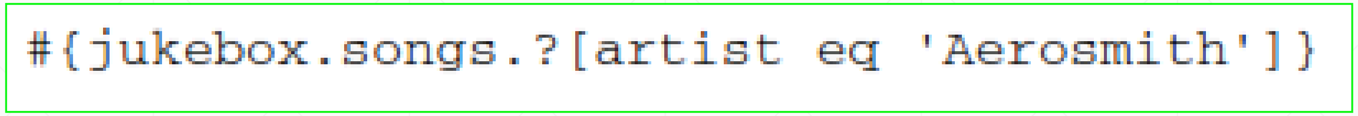


```
#{jukebox.songs[T(java.lang.Math).random() *
                         jukebox.songs.size()].title}
```



```
#{'This is a test'[3]}
```

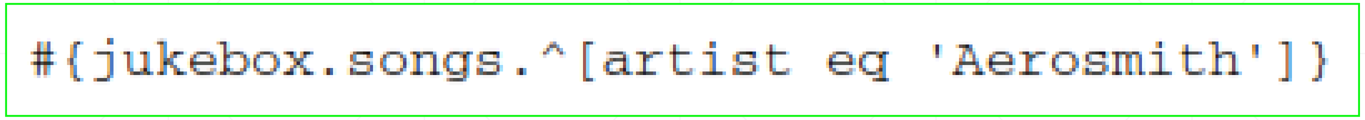
Fourth character – „s“



```
#{jukebox.songs.?[artist eq 'Aerosmith']}
```

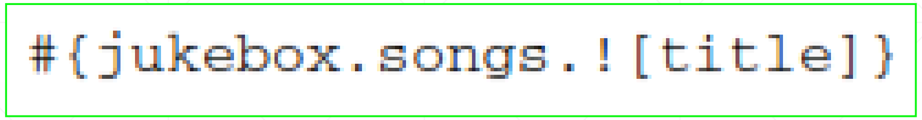
„.?[]“ Filter a subarray from an array

SpEL examples - Collections



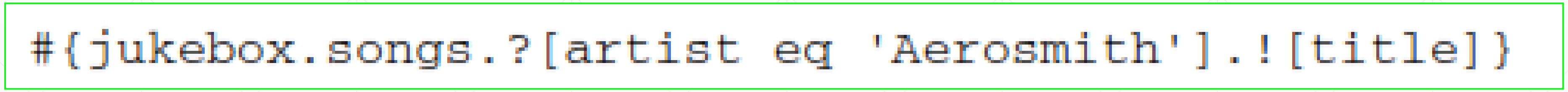
```
#{jukebox.songs.^[artist eq 'Aerosmith']}
```

„.^[]“ first matching entry and „.$[]“ last matching entry



```
#{jukebox.songs.![title]}
```

.![] project an array of songs objects to an array of strings - titles



```
#{jukebox.songs.?[artist eq 'Aerosmith'].![title]}
```

Returns an array of titles of all songs from Aerosmith

Aspect-oriented Spring

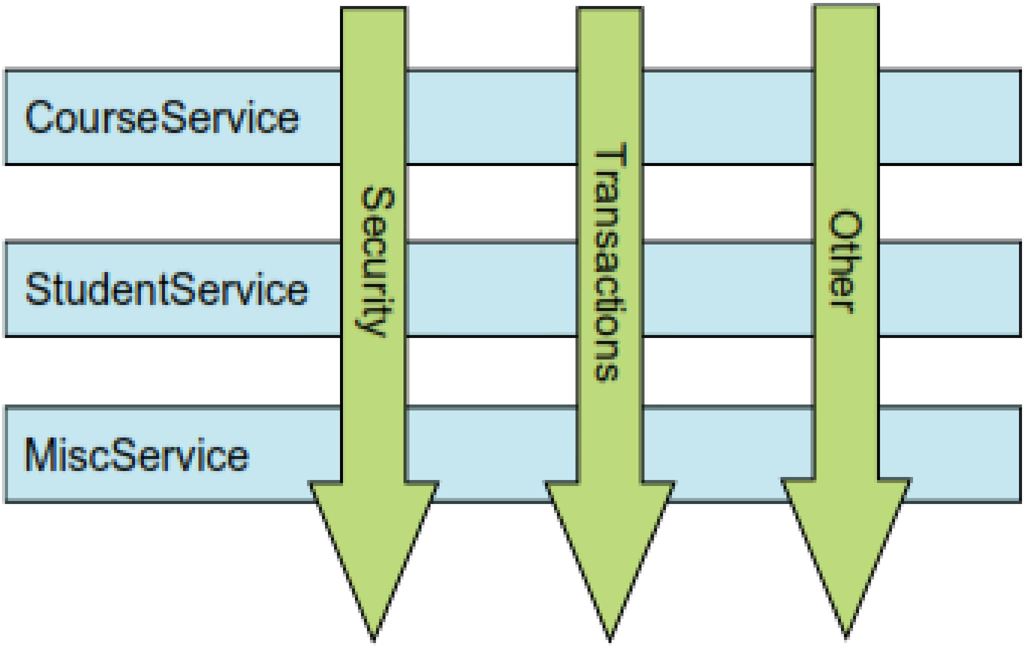


Figure 4.1 Aspects modularize cross-cutting concerns, applying logic that spans multiple application objects.

Crosscutting concerns can now be modularized
into special classes called *aspects.*
Benefits:

- the logic for each concern is in one place, as opposed to being scattered all over the code base.
- Your service modules are cleaner because they only contain code for their primary concern

Defining AOP terminology

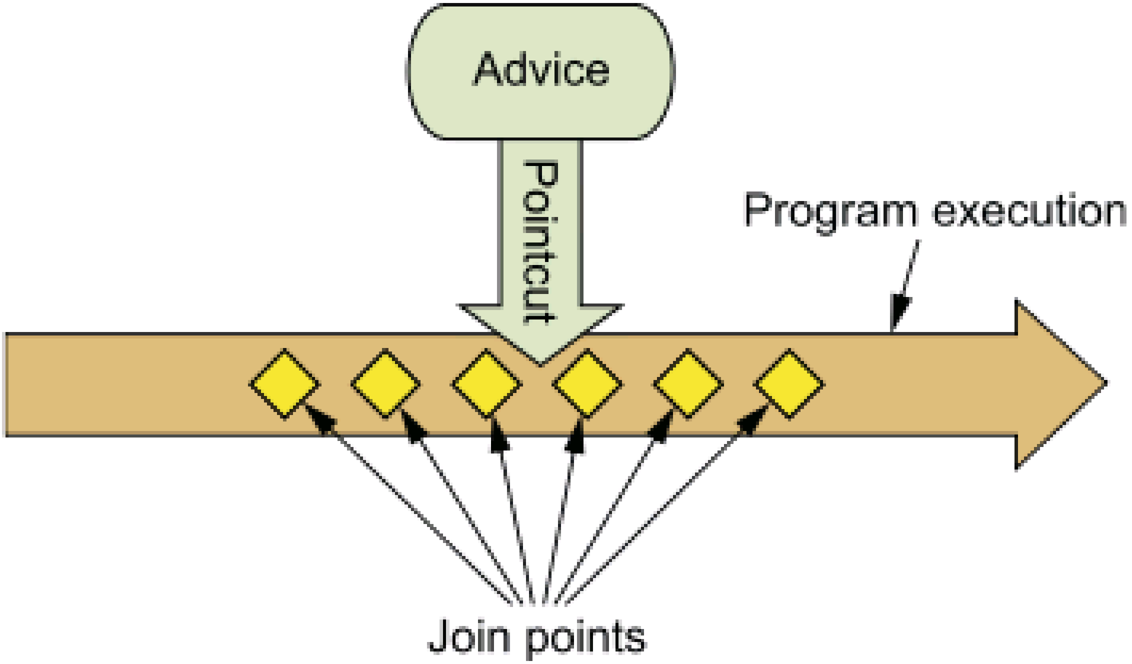
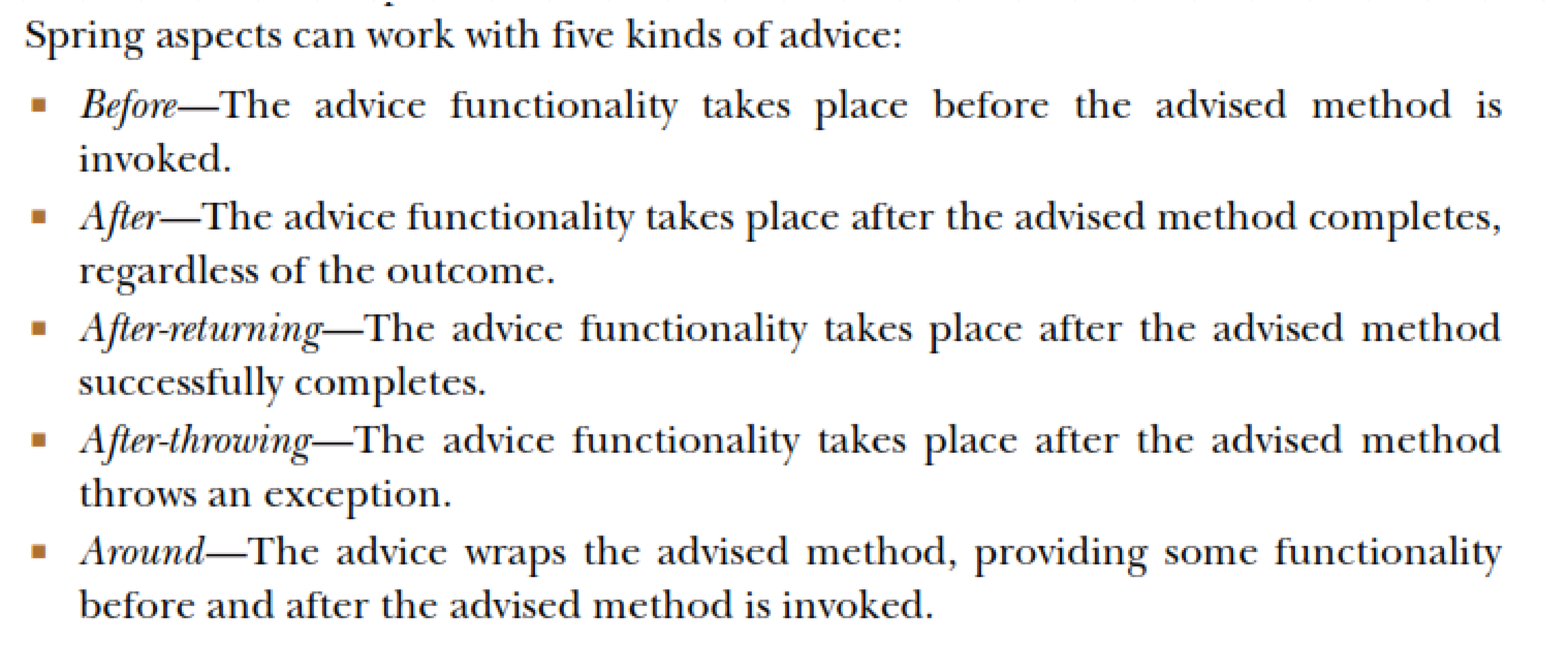


Figure 4.2 An aspect's functionality (advice) is woven into a program's execution at one or more join points.

Defining AOP terminology



Spring aspects can work with five kinds of advice:

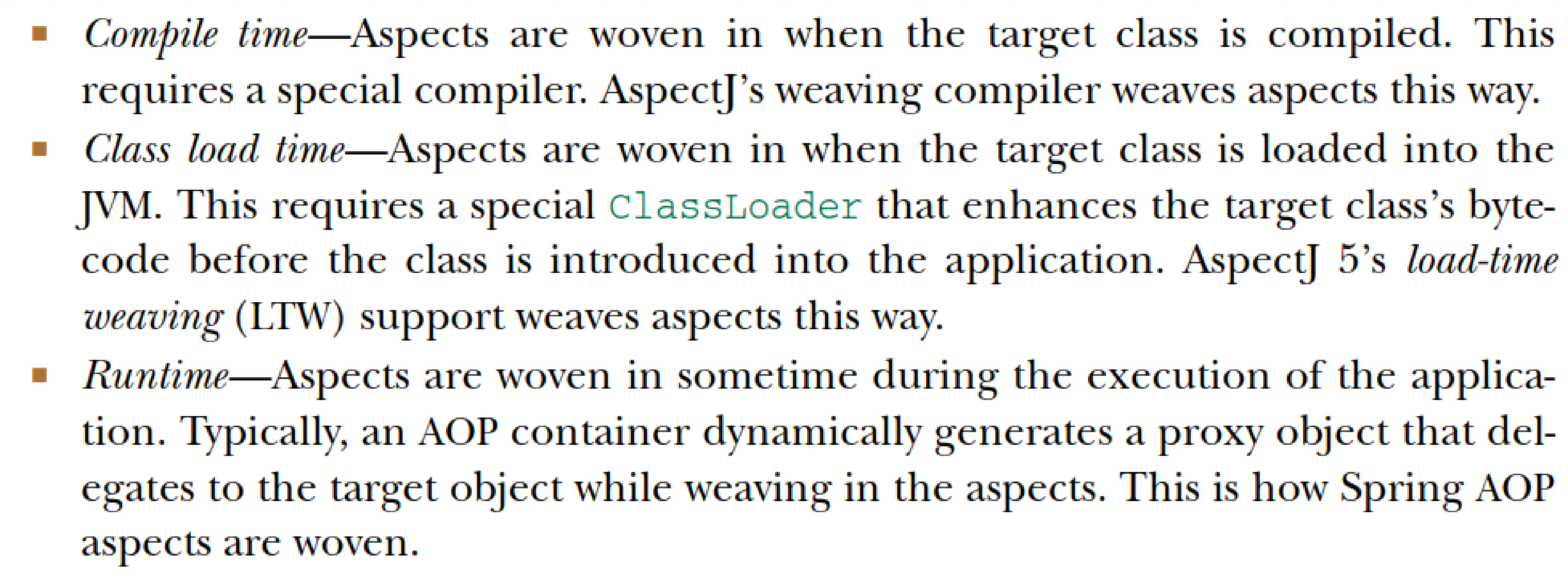
- *Before*—The advice functionality takes place before the advised method is invoked.
- *After*—The advice functionality takes place after the advised method completes, regardless of the outcome.
- *After-returning*—The advice functionality takes place after the advised method successfully completes.
- *After-throwing*—The advice functionality takes place after the advised method throws an exception.
- *Around*—The advice wraps the advised method, providing some functionality before and after the advised method is invoked.

Defining AOP terminology

- **Join points:** a point in the execution of the application where an aspect can be plugged in. This point could be a method being called, an exception being thrown, or even a field being modified.
- **Pointcuts**: an aspect doesn’t necessarily advise all join points in an application. *Pointcuts* help narrow down the join points advised by an aspect. you specify these pointcuts using explicit class and method names or through regular expressions that define matching class and method name patterns.
- **Aspect**: an aspect is the merger of advice and pointcuts.
- **Introduction**: an introduction allows you to add new methods or attributes to existing classes.

Defining AOP terminology (cont.)

- **Weaving:** the process of applying aspects to a target object to create a new proxied object. The aspects are woven into the target object at the specified join points. The weaving can take place at several points in the target object's lifetime:



- *Compile time*—Aspects are woven in when the target class is compiled. This requires a special compiler. AspectJ's weaving compiler weaves aspects this way.
- *Class load time*—Aspects are woven in when the target class is loaded into the JVM. This requires a special `ClassLoader` that enhances the target class's byte-code before the class is introduced into the application. AspectJ 5's *load-time weaving* (LTW) support weaves aspects this way.
- *Runtime*—Aspects are woven in sometime during the execution of the application. Typically, an AOP container dynamically generates a proxy object that delegates to the target object while weaving in the aspects. This is how Spring AOP aspects are woven.

Spring‘s AOP support

- Spring’s support for AOP comes in four styles:
  - **Classic Spring proxy-based AOP** - overcomplicated
  - **Pure-POJO aspects** - With Spring’s aop namespace, you can turn pure POJOs into aspects. They are called in reaction to a pointcut. Unfortunately, this technique requires XML configuration, but it’s an easy way to declaratively turn any object into an aspect.
  - **@AspectJ annotation-driven aspects**
  - **Injected AspectJ aspects** (available in all versions of Spring)
- Spring AOP is built around dynamic proxies. Consequently, Spring’s AOP support is limited to method interception.
- If your AOP needs exceed simple method interception (constructor or property interception, for example), you’ll want to consider implementing aspects in AspectJ. In that case, the fourth style listed will enable you to inject values into AspectJ-driven aspects

Selecting join points with pointcuts

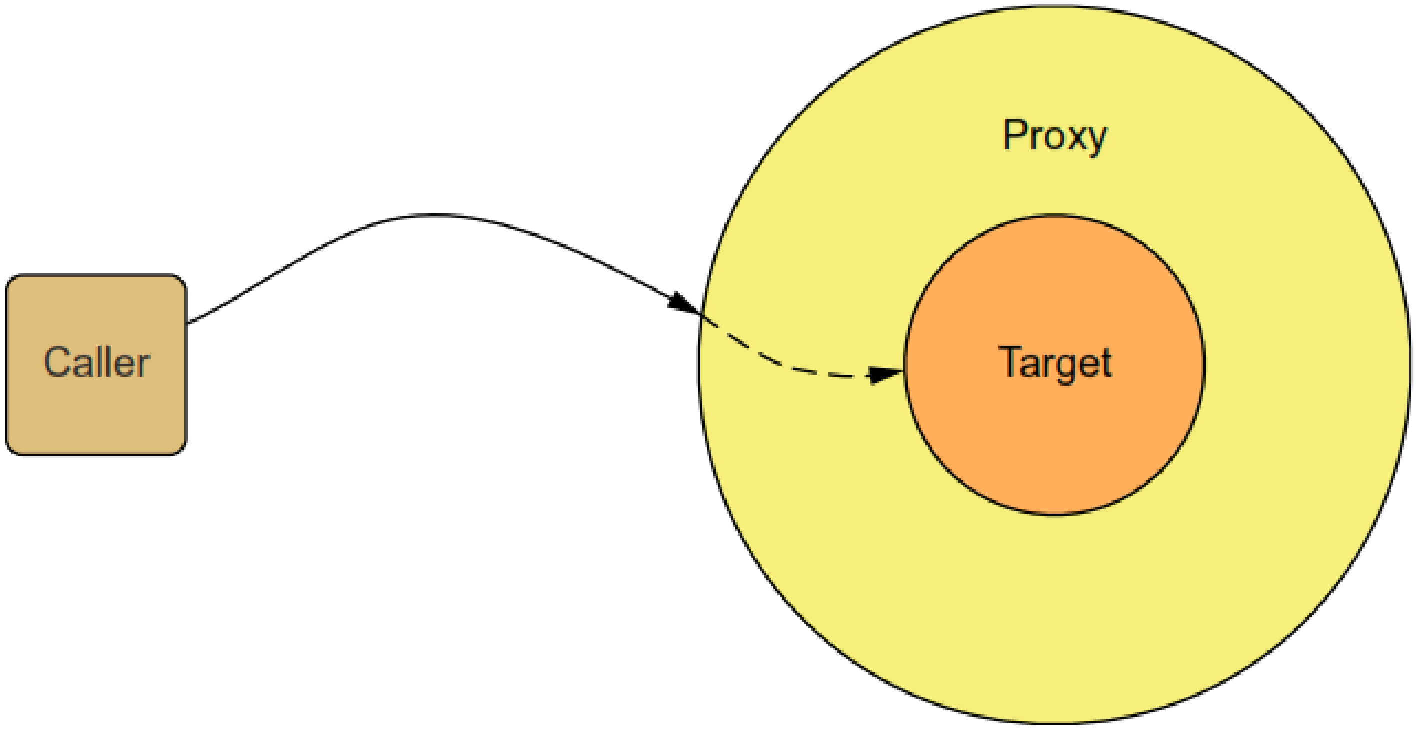


Figure 4.3 **Spring aspects are implemented as proxies that wrap the target object. The proxy handles method calls, performs additional aspect logic, and then invokes the target method.**

- SPRING ADVISES OBJECTS AT RUNTIME
- SPRING ONLY SUPPORTS METHOD JOIN POINTS

Selecting join points with pointcuts

- In Spring AOP, pointcuts are defined using AspectJ's pointcut expression language

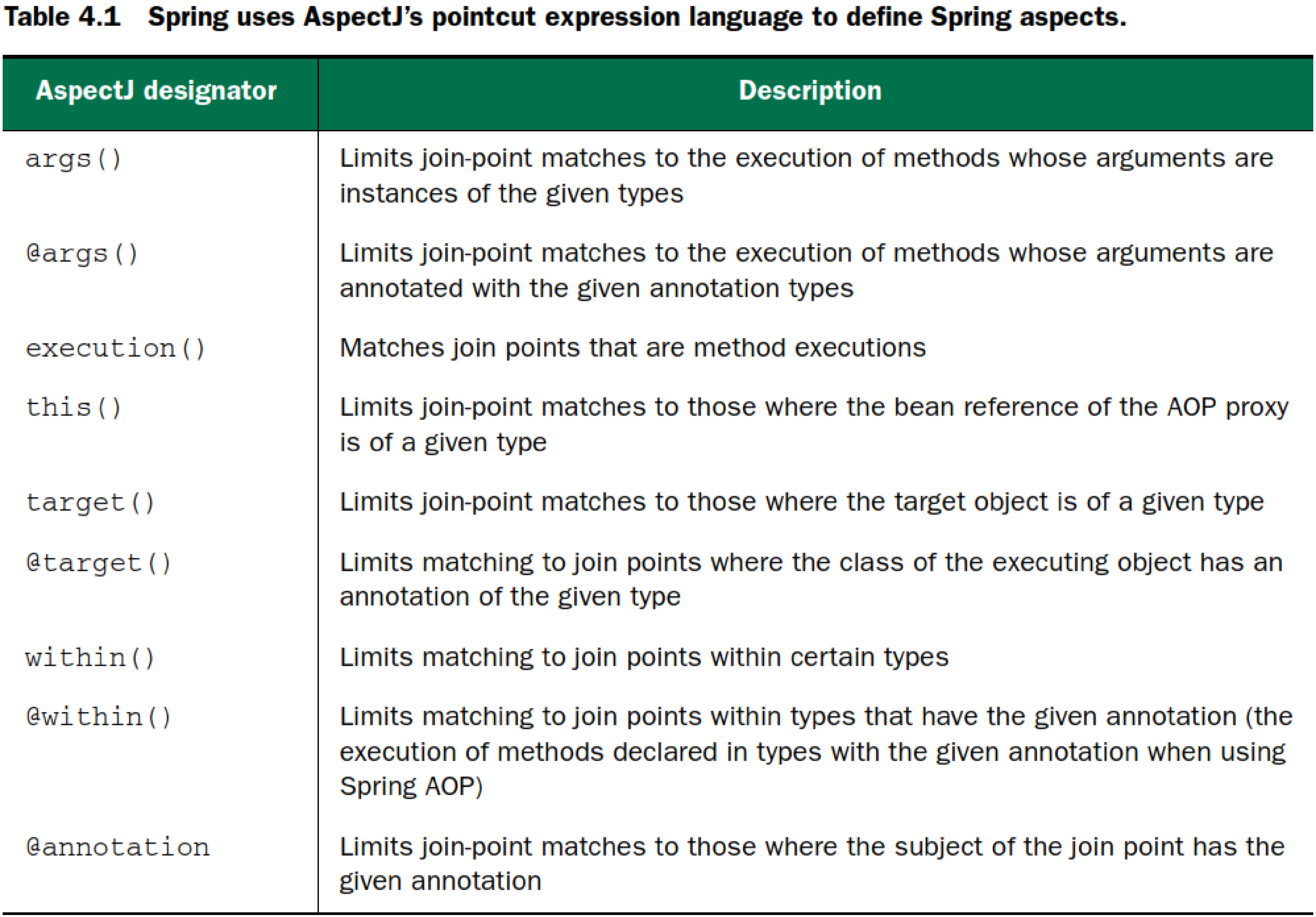
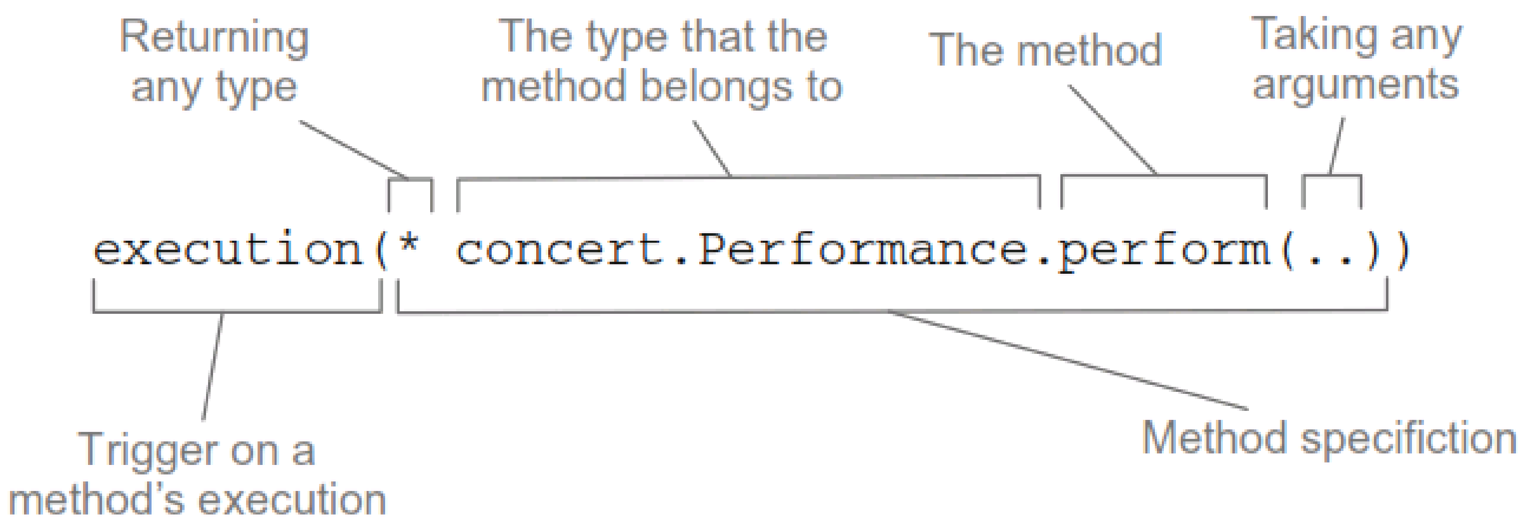


Table 4.1 Spring uses AspectJ's pointcut expression language to define Spring aspects.

AspectJ designator	Description
`args()`	Limits join-point matches to the execution of methods whose arguments are instances of the given types
`@args()`	Limits join-point matches to the execution of methods whose arguments are annotated with the given annotation types
`execution()`	Matches join points that are method executions
`this()`	Limits join-point matches to those where the bean reference of the AOP proxy is of a given type
`target()`	Limits join-point matches to those where the target object is of a given type
`@target()`	Limits matching to join points where the class of the executing object has an annotation of the given type
`within()`	Limits matching to join points within certain types
`@within()`	Limits matching to join points within types that have the given annotation (the execution of methods declared in types with the given annotation when using Spring AOP)
`@annotation`	Limits join-point matches to those where the subject of the join point has the given annotation

Writing pointcuts

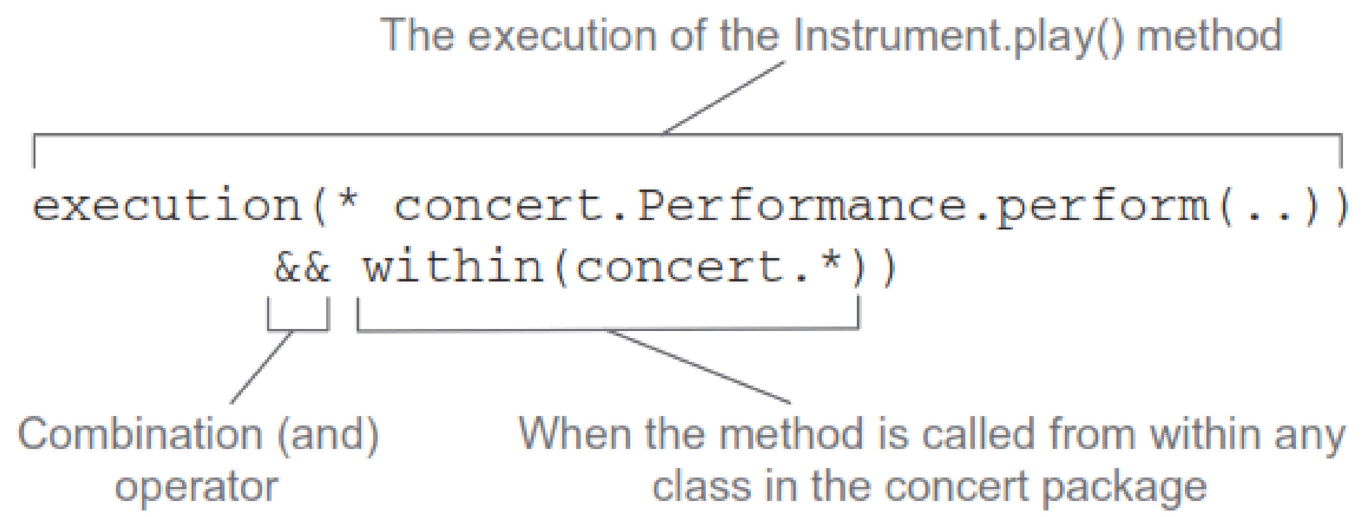
```
package concert;

public interface Performance {
  public void perform();
}
```



```
execution(* concert.Performance.perform(..))
```

Figure 4.4 Selecting `Performance's perform()` method with an AspectJ pointcut expression

Writing pointcuts (cont.)



The execution of the Instrument.play() method

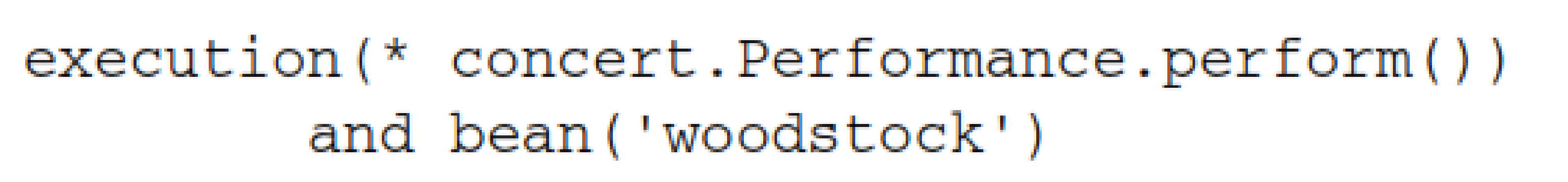
```
execution(* concert.Performance.perform(..))
          && within(concert.*))
```

Combination (and) operator

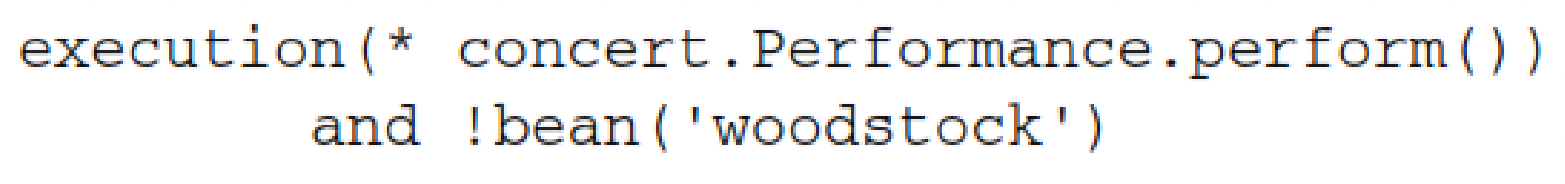
When the method is called from within any class in the concert package

Figure 4.5 Limiting a pointcut's reach by using the `within()` designator

Selecting beans in pointcuts

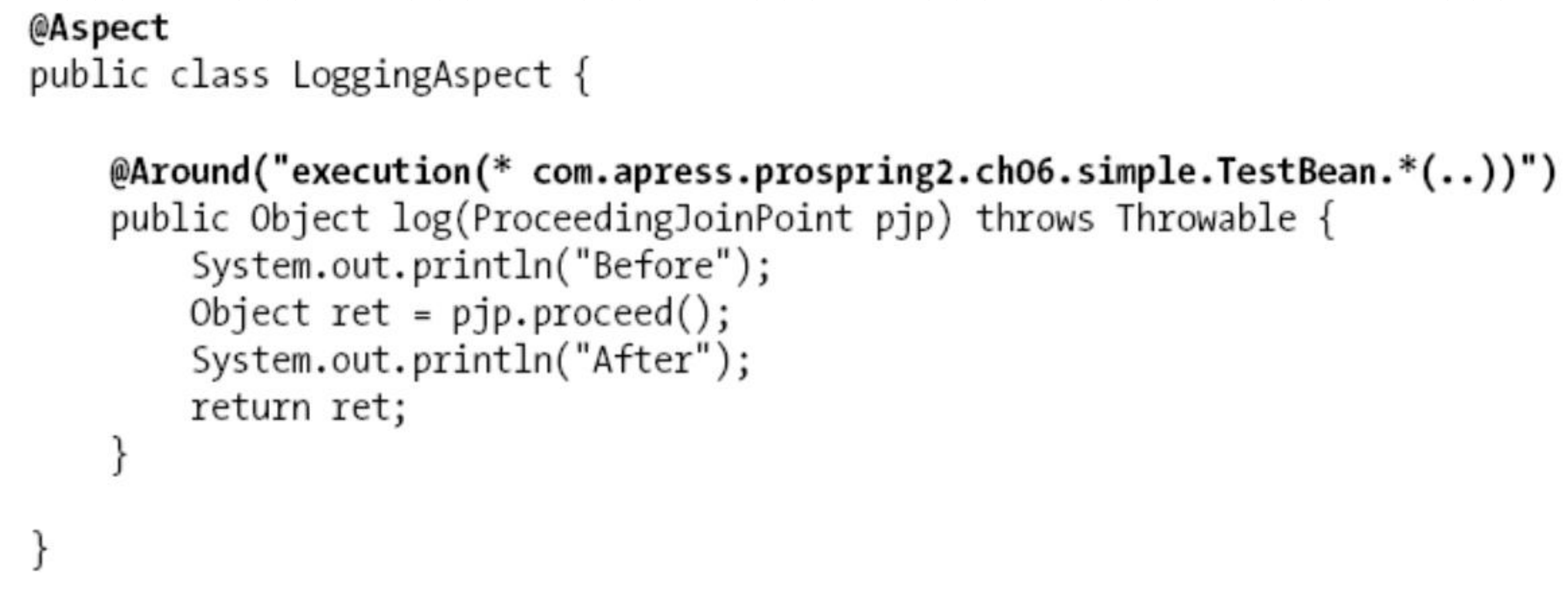


```
execution(* concert.Performance.perform())
        and bean('woodstock')
```



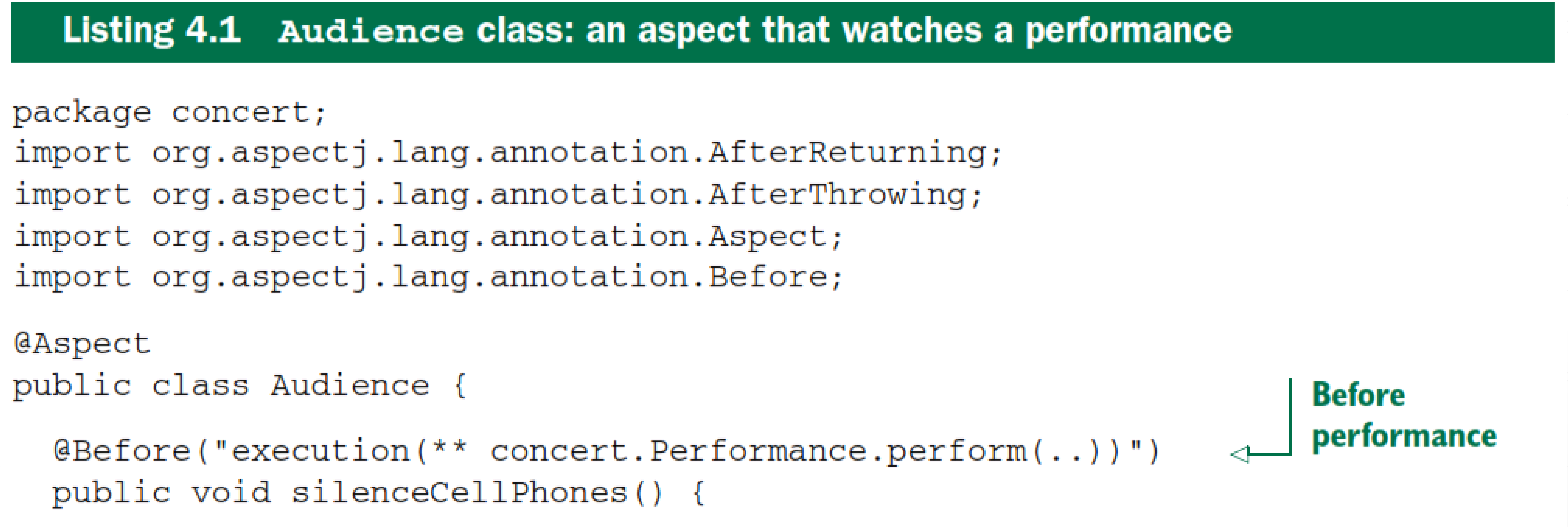
```
execution(* concert.Performance.perform())
        and !bean('woodstock')
```

Creating annotated aspects



```
@Aspect
public class LoggingAspect {

    @Around("execution(* com.apress.prospring2.ch06.simple.TestBean.*(..))")
    public Object log(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("Before");
        Object ret = pjp.proceed();
        System.out.println("After");
        return ret;
    }

}
```

Creating annotated aspects

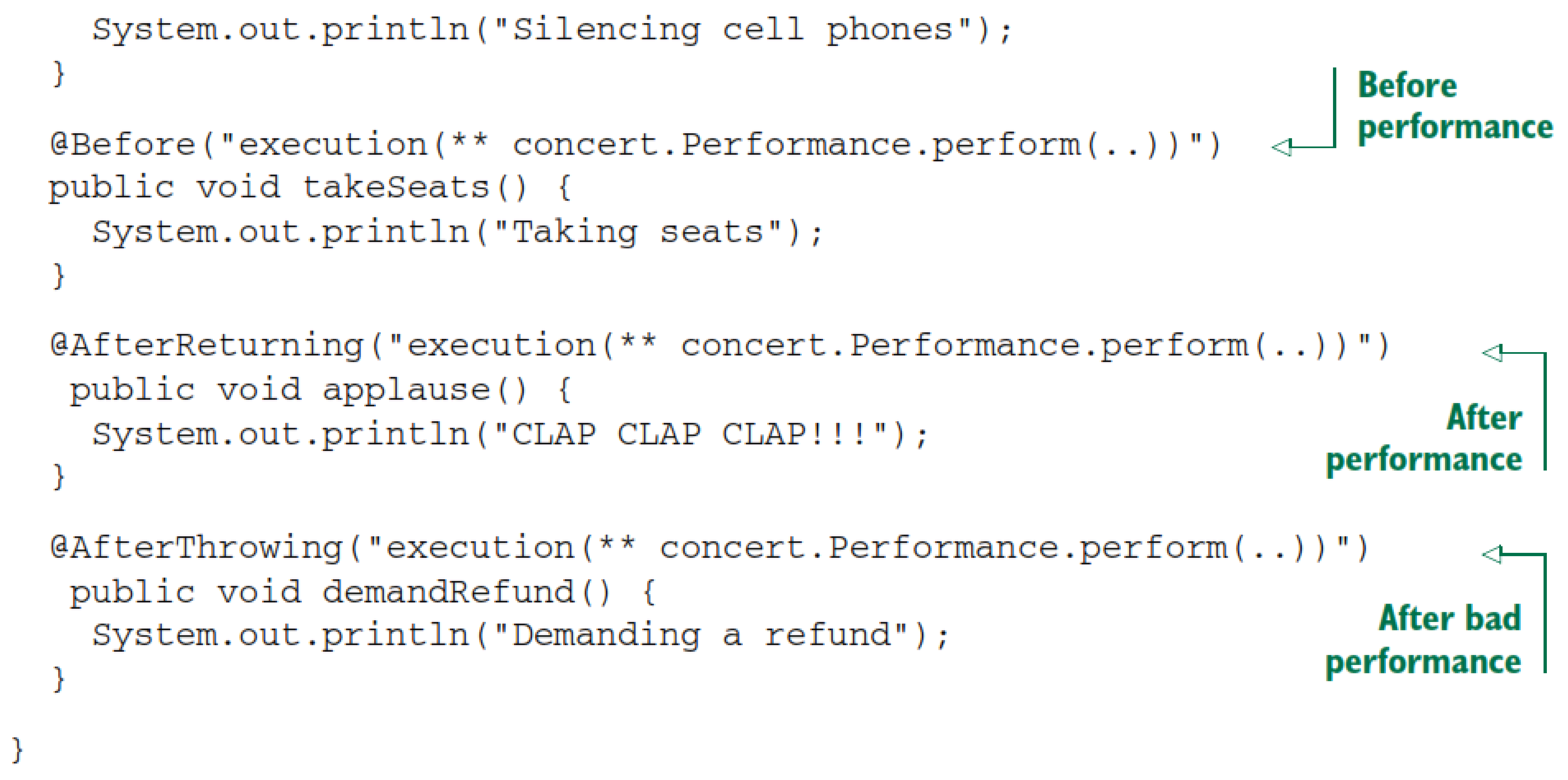


Listing 4.1 `Audience` **class: an aspect that watches a performance**

```
package concert;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class Audience {

  @Before("execution(** concert.Performance.perform(..))")
  public void silenceCellPhones() {
```

Before performance

Creating annotated aspects



```
    System.out.println("Silencing cell phones");
  }

  @Before("execution(** concert.Performance.perform(..))")
  public void takeSeats() {
    System.out.println("Taking seats");
  }

  @AfterReturning("execution(** concert.Performance.perform(..))")
   public void applause() {
    System.out.println("CLAP CLAP CLAP!!!");
  }

  @AfterThrowing("execution(** concert.Performance.perform(..))")
   public void demandRefund() {
    System.out.println("Demanding a refund");
  }

}
```

Before performance

After performance

After bad performance

Spring annotation to advice declaration

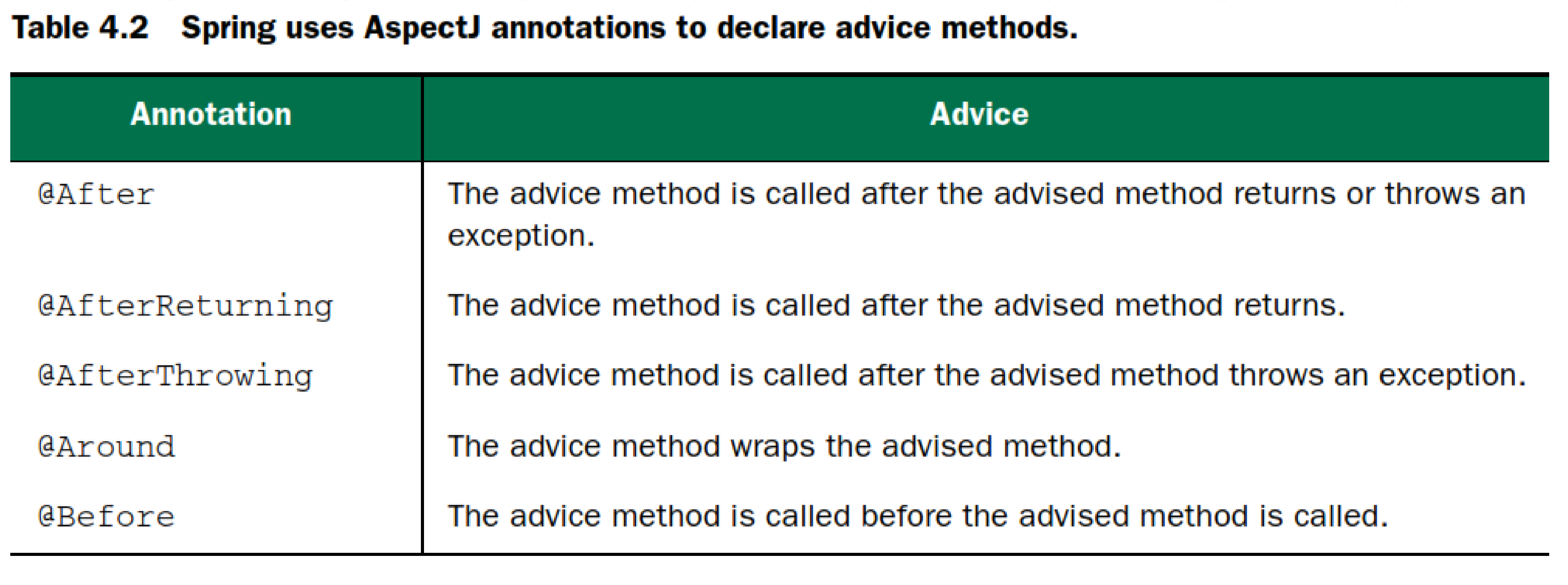
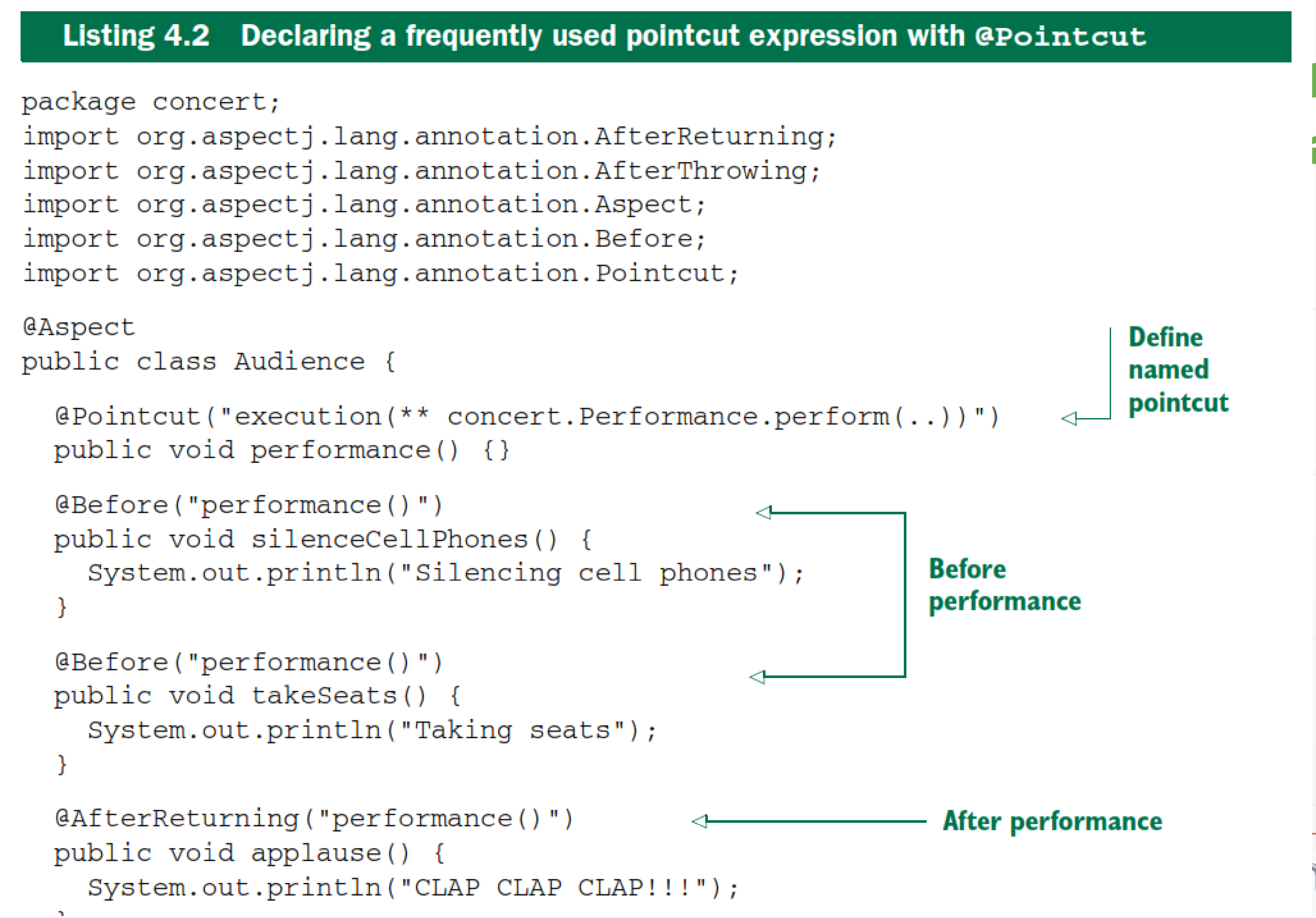


Table 4.2 Spring uses AspectJ annotations to declare advice methods.

Annotation	Advice
`@After`	The advice method is called after the advised method returns or throws an exception.
`@AfterReturning`	The advice method is called after the advised method returns.
`@AfterThrowing`	The advice method is called after the advised method throws an exception.
`@Around`	The advice method wraps the advised method.
`@Before`	The advice method is called before the advised method is called.



Listing 4.2 Declaring a frequently used pointcut expression with `@Pointcut`

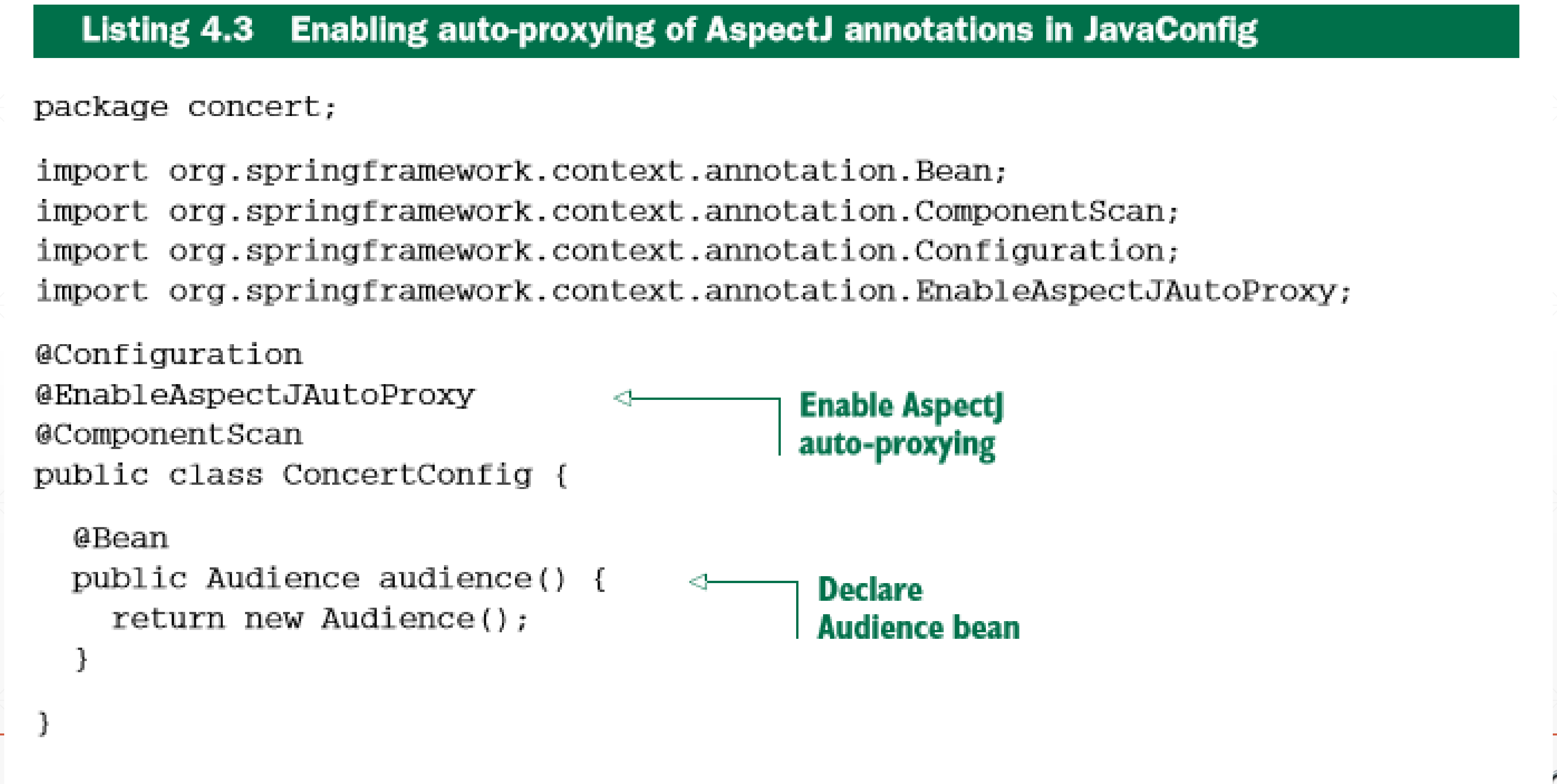
```
package concert;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class Audience {

  @Pointcut("execution(** concert.Performance.perform(..))")
  public void performance() {}

  @Before("performance()")
  public void silenceCellPhones() {
    System.out.println("Silencing cell phones");
  }

  @Before("performance()")
  public void takeSeats() {
    System.out.println("Taking seats");
  }

  @AfterReturning("performance()")
  public void applause() {
    System.out.println("CLAP CLAP CLAP!!!");
```

Define named pointcut

Before performance

After performance

Enabling auto-proxying in JavaConfig



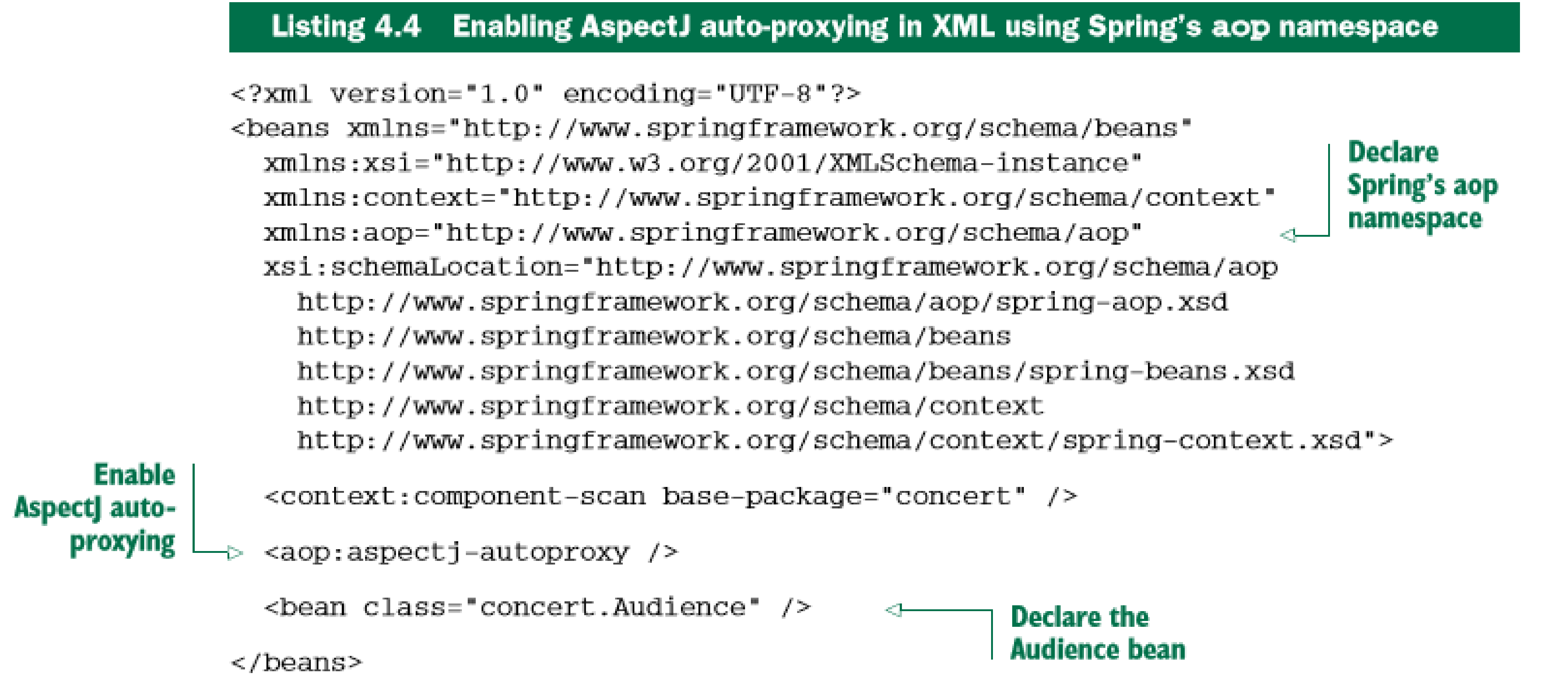
Listing 4.3 Enabling auto-proxying of AspectJ annotations in JavaConfig

```
package concert;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;

@Configuration
@EnableAspectJAutoProxy
@ComponentScan
public class ConcertConfig {

  @Bean
  public Audience audience() {
    return new Audience();
  }

}
```

Enable AspectJ auto-proxying

Declare Audience bean

Enabling auto-proxying in XML



Listing 4.4 Enabling AspectJ auto-proxying in XML using Spring's aop namespace

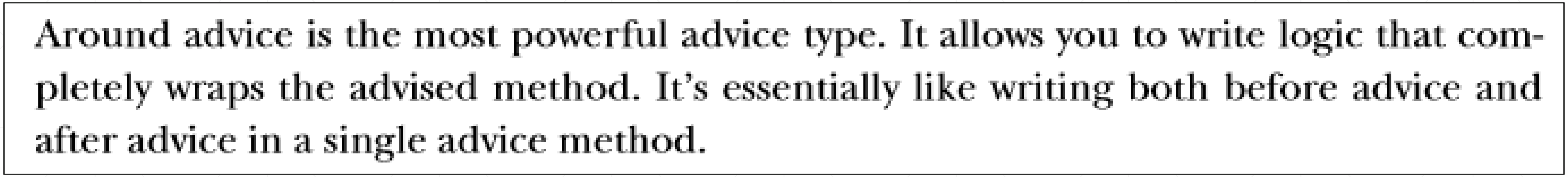
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xsi:schemaLocation="http://www.springframework.org/schema/aop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

  <context:component-scan base-package="concert" />

  <aop:aspectj-autoproxy />

  <bean class="concert.Audience" />

</beans>
```

Declare Spring's aop namespace

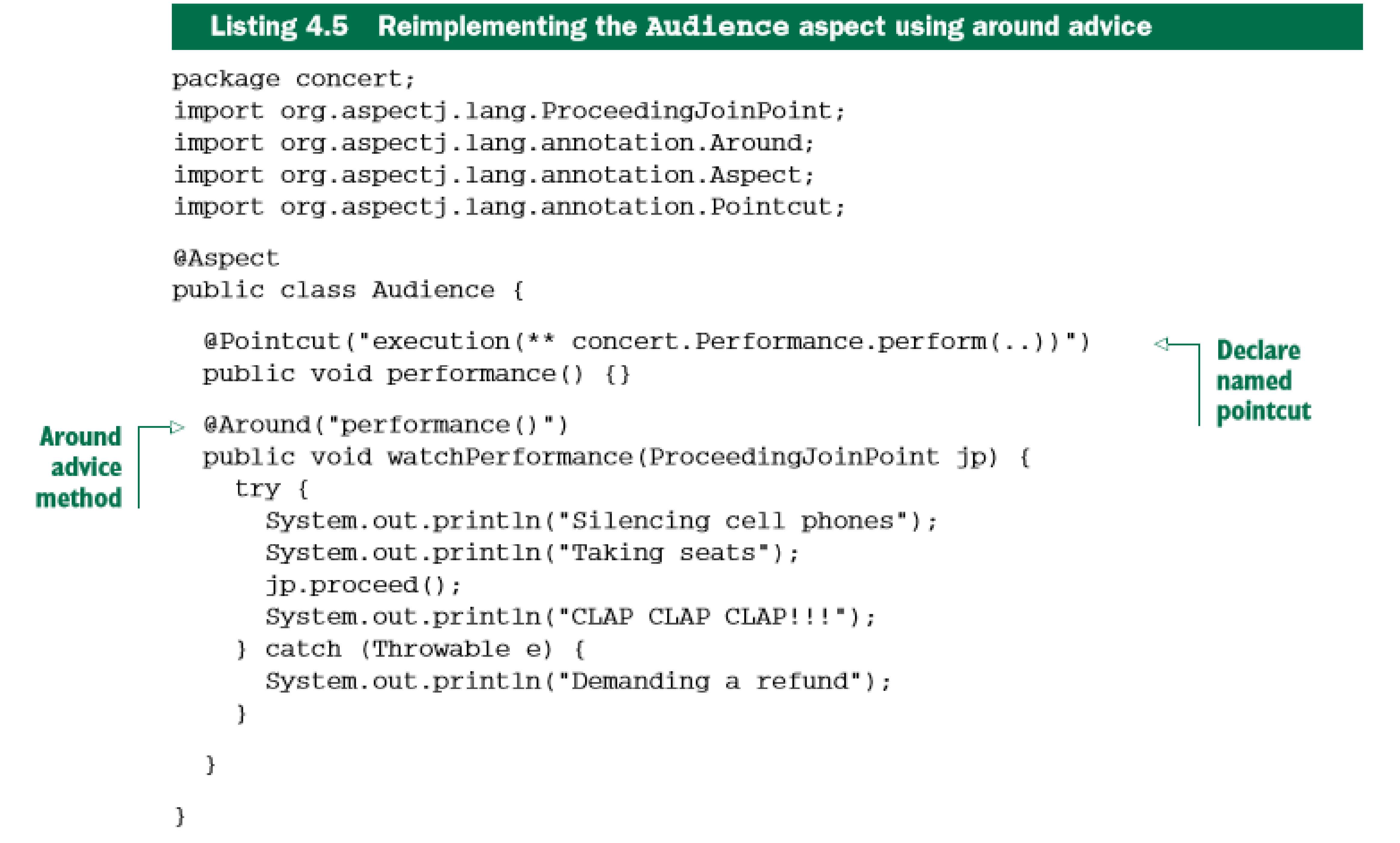
Enable AspectJ auto-proxying

Declare the Audience bean

Creating around advice



Around advice is the most powerful advice type. It allows you to write logic that completely wraps the advised method. It's essentially like writing both before advice and after advice in a single advice method.



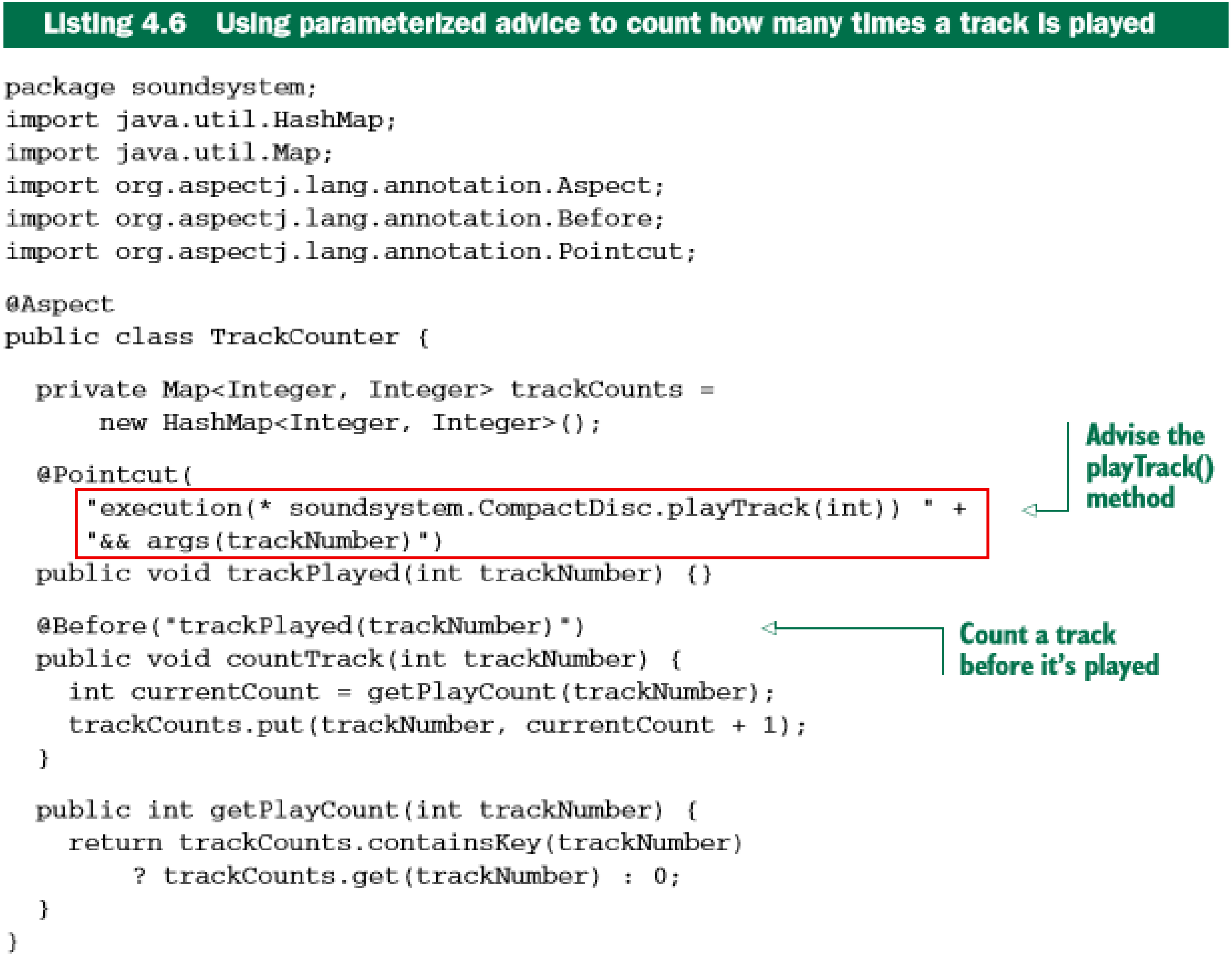
Listing 4.5 Reimplementing the Audience aspect using around advice

```
package concert;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class Audience {

  @Pointcut("execution(** concert.Performance.perform(..))")
  public void performance() {}

  @Around("performance()")
  public void watchPerformance(ProceedingJoinPoint jp) {
    try {
      System.out.println("Silencing cell phones");
      System.out.println("Taking seats");
      jp.proceed();
      System.out.println("CLAP CLAP CLAP!!!");
    } catch (Throwable e) {
      System.out.println("Demanding a refund");
    }

  }

}
```

Declare named pointcut

Around advice method

Handling parameters in advice



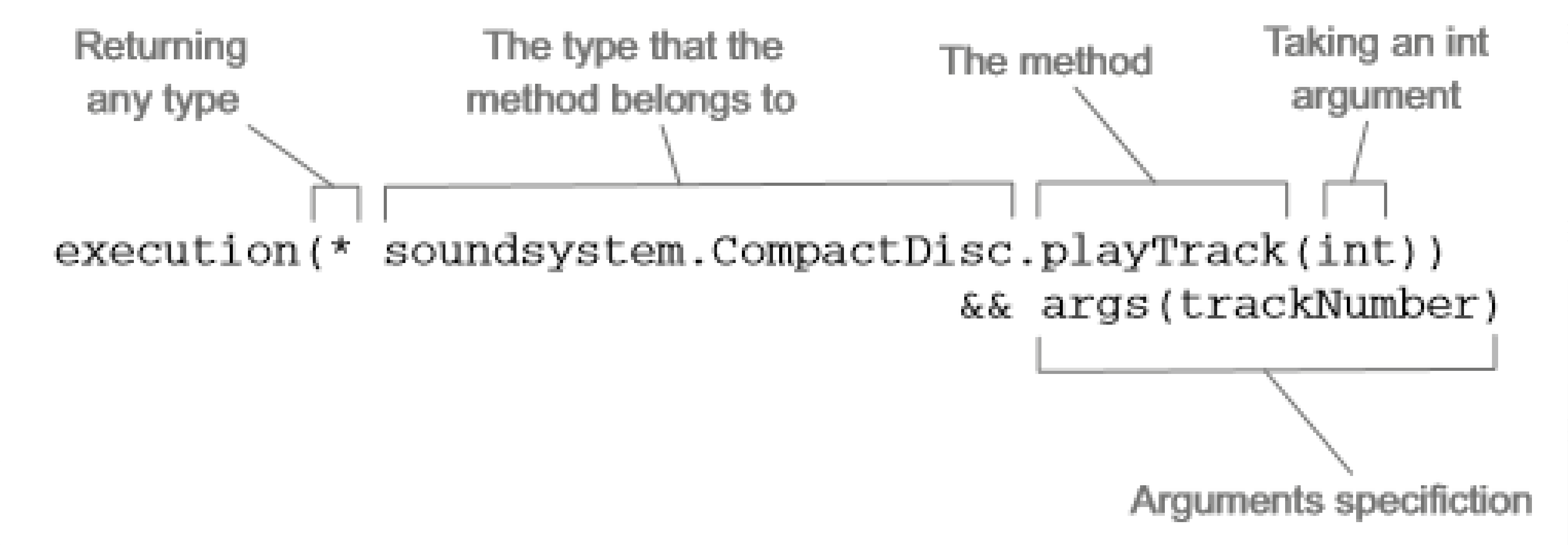
Listing 4.6 Using parameterized advice to count how many times a track is played

```
package soundsystem;
import java.util.HashMap;
import java.util.Map;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class TrackCounter {

  private Map<Integer, Integer> trackCounts =
      new HashMap<Integer, Integer>();

  @Pointcut(
     "execution(* soundsystem.CompactDisc.playTrack(int)) " +
     "&& args(trackNumber)")
  public void trackPlayed(int trackNumber) {}

  @Before("trackPlayed(trackNumber)")
  public void countTrack(int trackNumber) {
    int currentCount = getPlayCount(trackNumber);
    trackCounts.put(trackNumber, currentCount + 1);
  }

  public int getPlayCount(int trackNumber) {
    return trackCounts.containsKey(trackNumber)
        ? trackCounts.get(trackNumber) : 0;
  }
}
```

Advise the playTrack() method

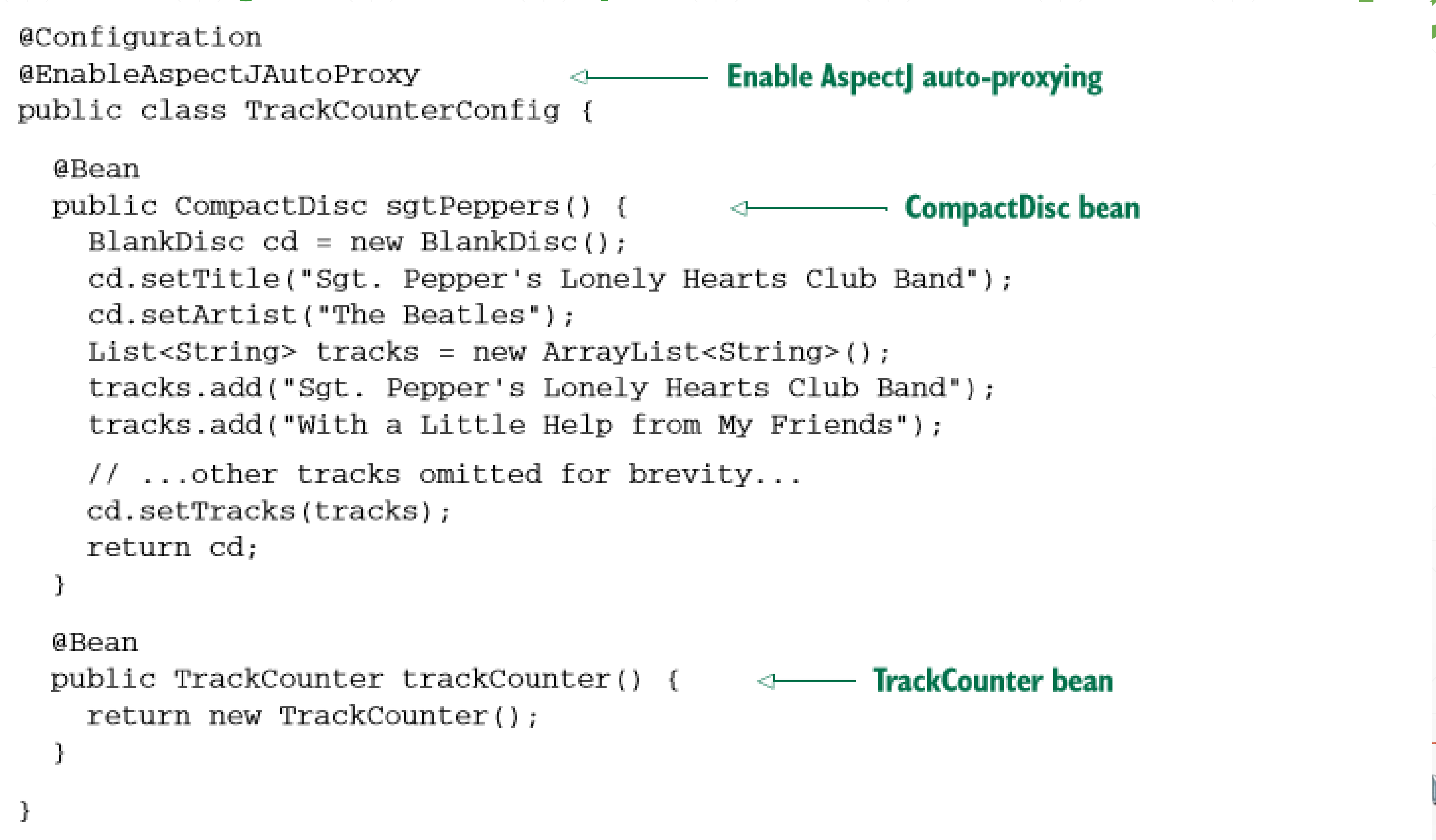
Count a track before it's played

Handling parameters in advice



```
execution(* soundsystem.CompactDisc.playTrack(int))
                                 && args(trackNumber)
```

Creating annotated aspects



```
@Configuration
@EnableAspectJAutoProxy
public class TrackCounterConfig {

  @Bean
  public CompactDisc sgtPeppers() {
    BlankDisc cd = new BlankDisc();
    cd.setTitle("Sgt. Pepper's Lonely Hearts Club Band");
    cd.setArtist("The Beatles");
    List<String> tracks = new ArrayList<String>();
    tracks.add("Sgt. Pepper's Lonely Hearts Club Band");
    tracks.add("With a Little Help from My Friends");

    // ...other tracks omitted for brevity...
    cd.setTracks(tracks);
    return cd;
  }

  @Bean
  public TrackCounter trackCounter() {
    return new TrackCounter();
  }

}
```

Enable AspectJ auto-proxying

CompactDisc bean

TrackCounter bean

Spring In Action