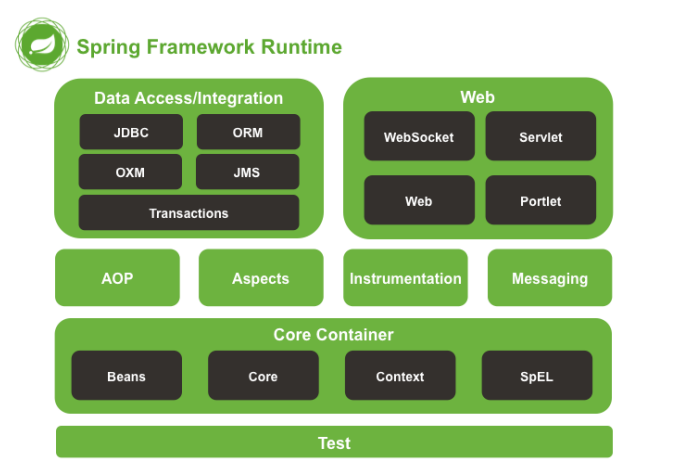
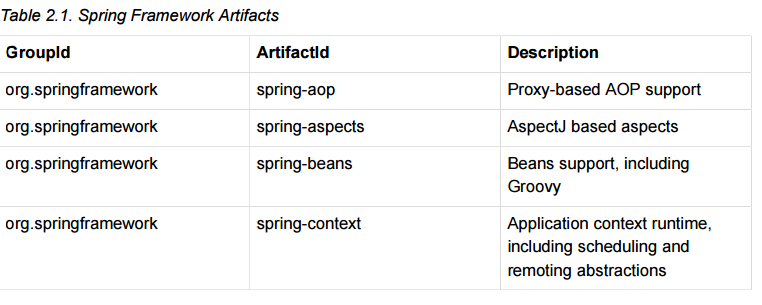
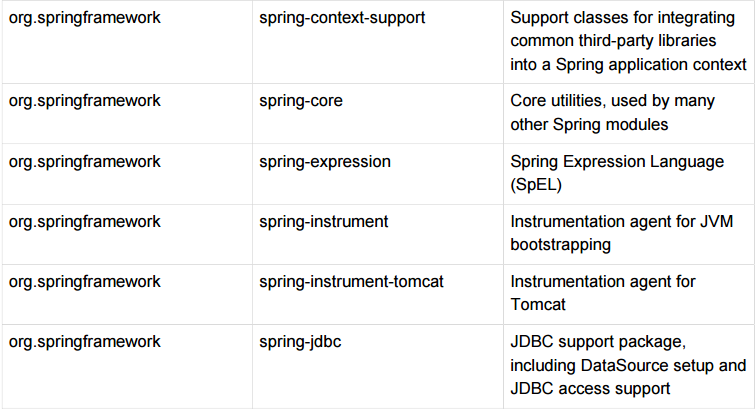
Spring framework

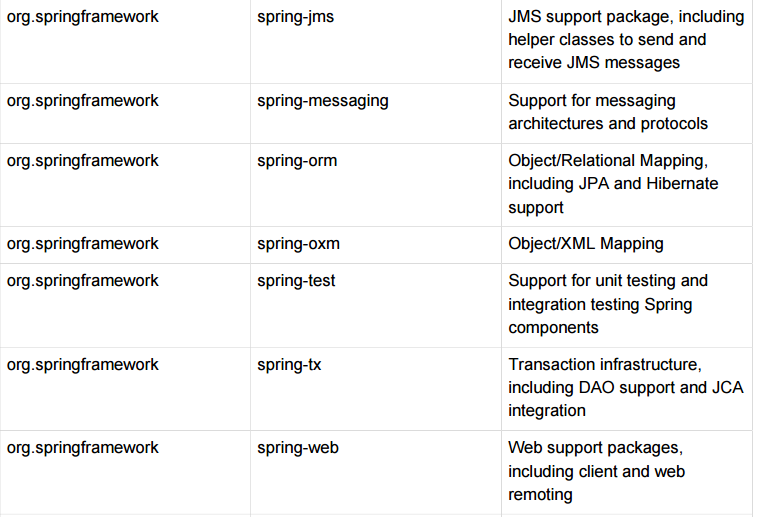
The Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications.

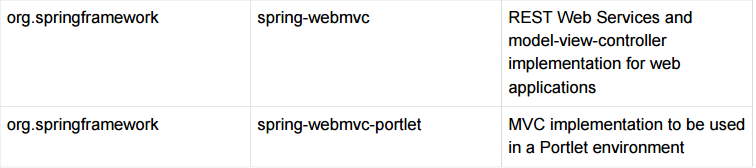


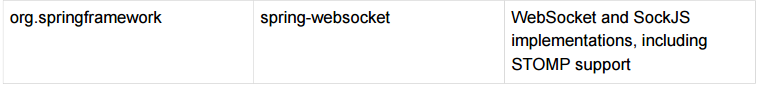
Core Container The Core Container consists of the spring-core, spring-beans, spring-context, springcontext-support, and spring-expression (Spring Expression Language) modules.











## 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* (IoC), of the bean itself

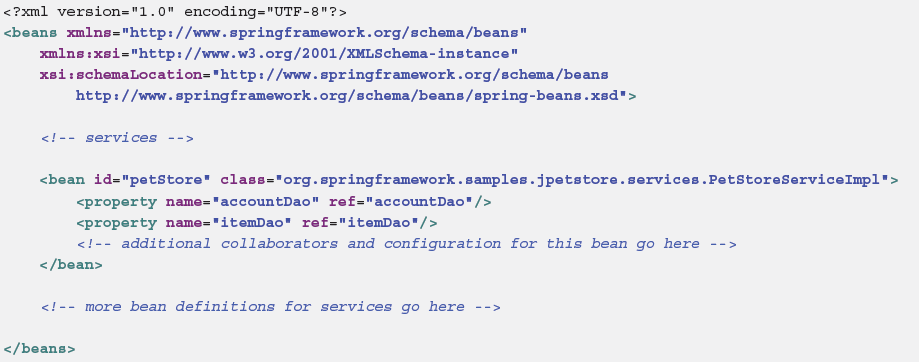
controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the *Service Locator* pattern.

The BeanFactory interface provides an advanced configuration mechanism capable of managing any type of object. ApplicationContext is a subinterface of BeanFactory. It adds easier integration with Spring’s AOP features; message resource handling (for use in internationalization), event publication; and application-layer specific contexts such as the WebApplicationContext for use in web applications. In short, the BeanFactory provides the configuration framework and basic functionality, and the ApplicationContext adds more enterprise-specific functionality. The ApplicationContext is a complete superset of the BeanFactory,

## Container overview

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

Several implementations of the ApplicationContext interface are supplied out-of-thebox with Spring. In standalone 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

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

**Instantiating a container**

ApplicationContext context =

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

**Composing XML-based configuration metadata**

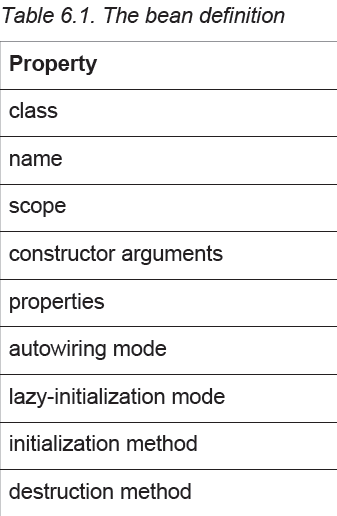


**Using the container**

The ApplicationContext enables you to read bean definitions and access them as follows:



**Bean overview**



**Aliasing a bean outside the bean definition**

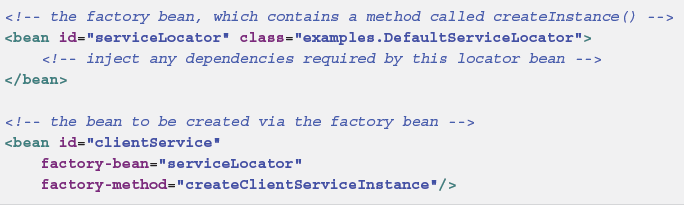
In a bean definition itself, you can supply more than one name for the bean, by using a combination of up to one name specified by the id attribute, and any number of other names in the name attribute. These names can be equivalent aliases to the same bean, and are useful for some situations, such as allowing each component in an application to refer to a common dependency by using a bean name that is specific to that component itself.

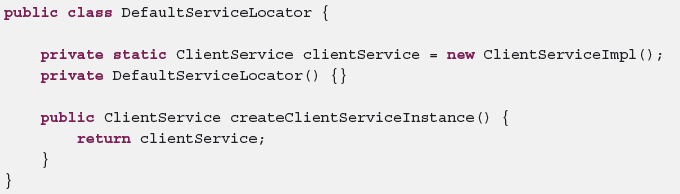
**<alias name**=**"fromName" alias**=**"toName"/>**

**Instantiation with a static factory method**



**Instantiation using an instance factory method**





**Dependency Injection**

Dependency injection (DI) 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.

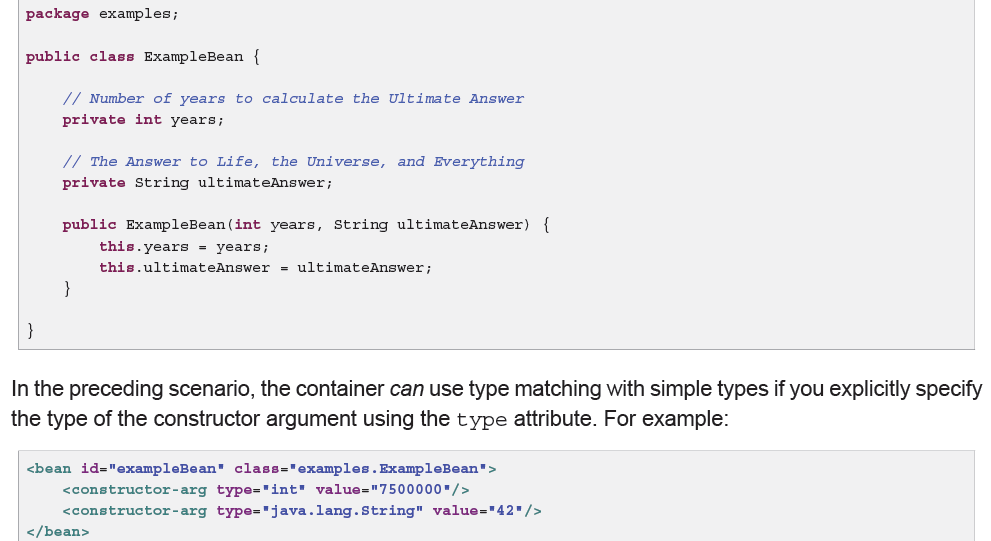
DI exists in two major variants, Constructor-based dependency injection and Setter-based dependency injection.

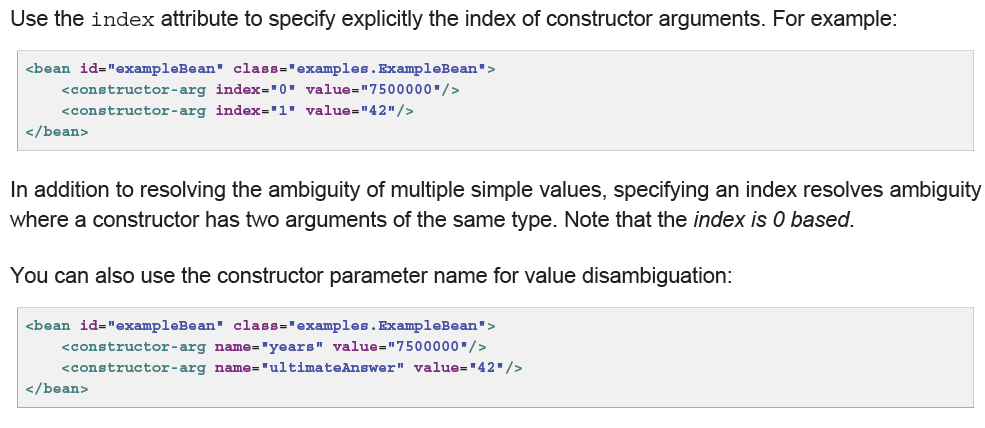
**Constructor-based dependency injection**

Constructor-based DI is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency. Calling a static factory method with specific arguments to construct the bean is nearly equivalent.

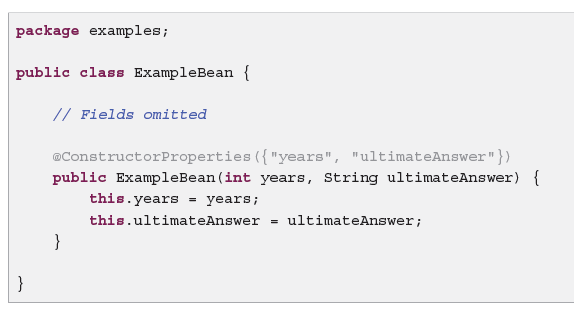
**Constructor argument resolution**

When a simple type is used, such as <value>true</value>, Spring cannot determine the type of the value, and so cannot match by type without help



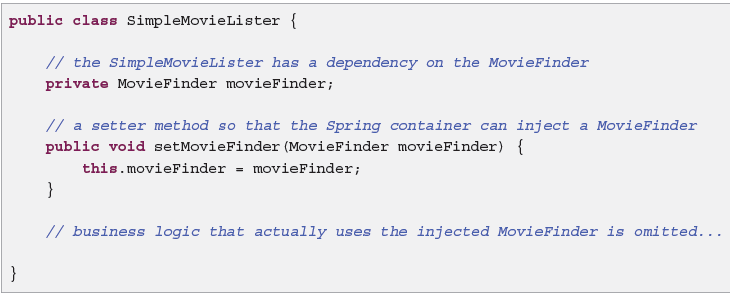


Keep in mind that to make this work out of the box your code must be compiled with the debug flag enabled so that Spring can look up the parameter name from the constructor. If you can’t compile your code with debug flag (or don’t want to) you can use @ConstructorProperties JDK annotation to explicitly name your constructor arguments. The sample class would then have to look as follows:



**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 no-argument static factory method to instantiate your bean.



**Constructor-based or setter-based DI?**

Since you can mix constructor-based and setter-based DI, it is a good rule of thumb to use constructors for mandatory dependencies and setter methods or configuration methods for optional dependencies.

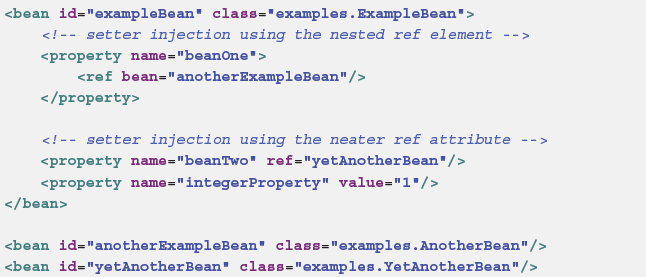
**Circular dependencies**

If you use predominantly constructor injection, it is possible to create an unresolvable circular dependency scenario.

For example: Class A requires an instance of class B through constructor injection, and class B requires an instance of class A through constructor injection. If you configure beans for classes A and B to be injected into each other, the Spring IoC container detects this circular reference at runtime, and throws a BeanCurrentlyInCreationException.

One possible solution is to edit the source code of some classes to be configured by setters rather than constructors. Alternatively, avoid constructor injection and use setter injection only. In other words, although it is not recommended, you can configure circular dependencies with setter injection.

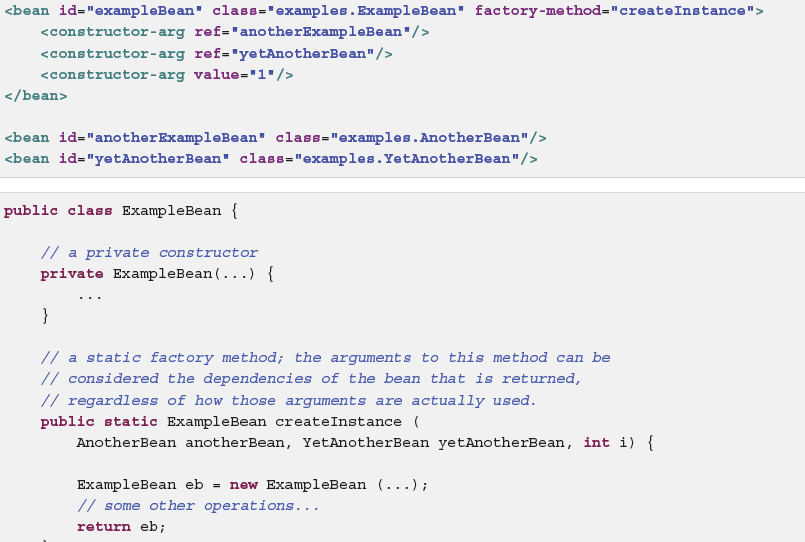
Example:



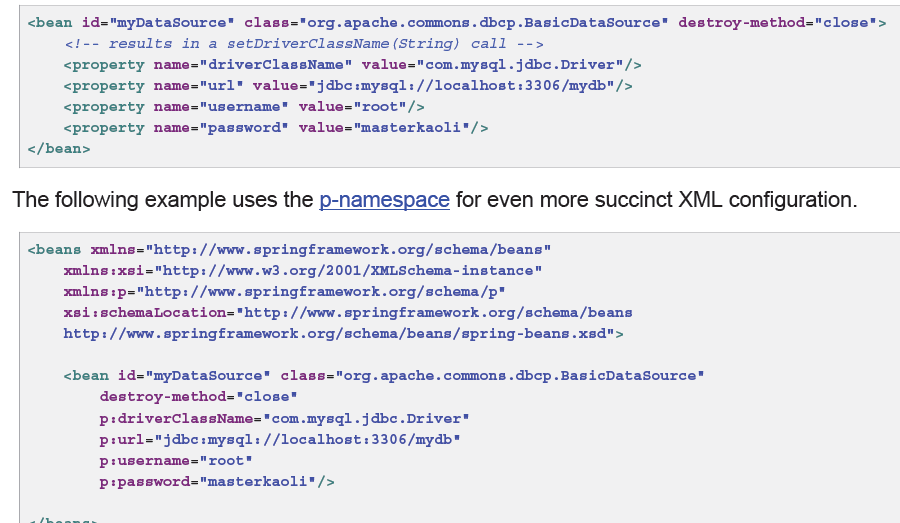


Now consider a variant of this example, where instead of using a constructor, Spring is told to call a

static factory method to return an instance of the object:



**Dependencies and configuration in detail**



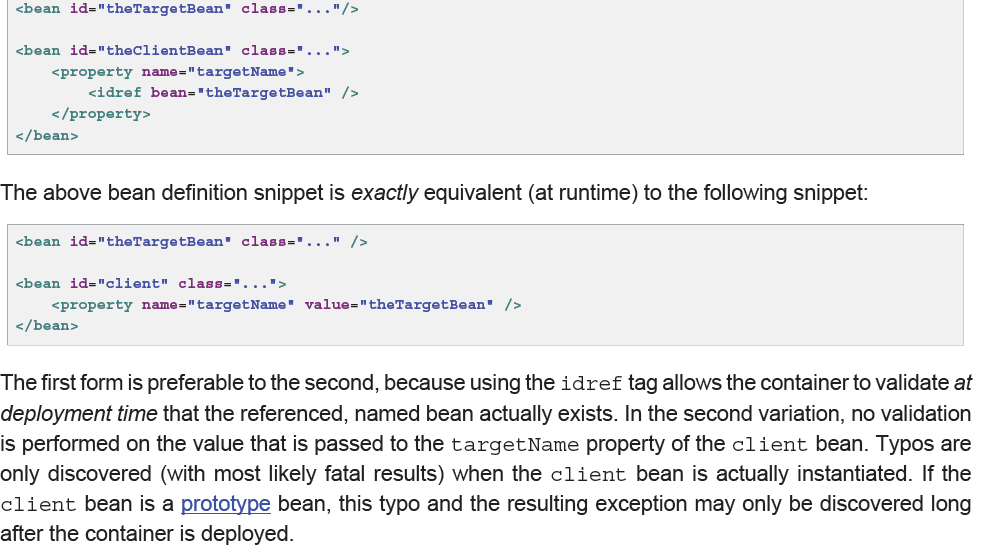
You can also configure a java.util.Properties instance as:



**The idref element**

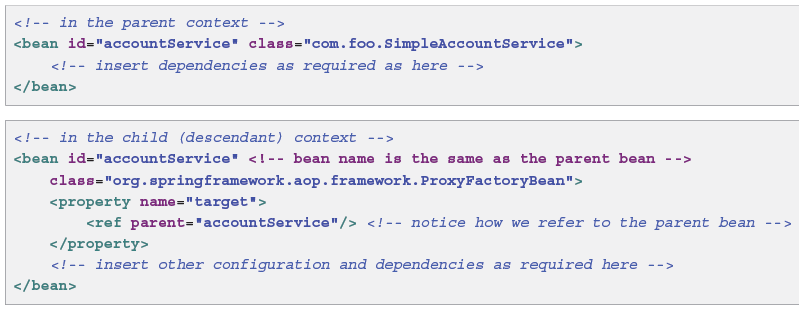
The idref element is simply an error-proof way to pass the *id* (string value - not a reference) of another

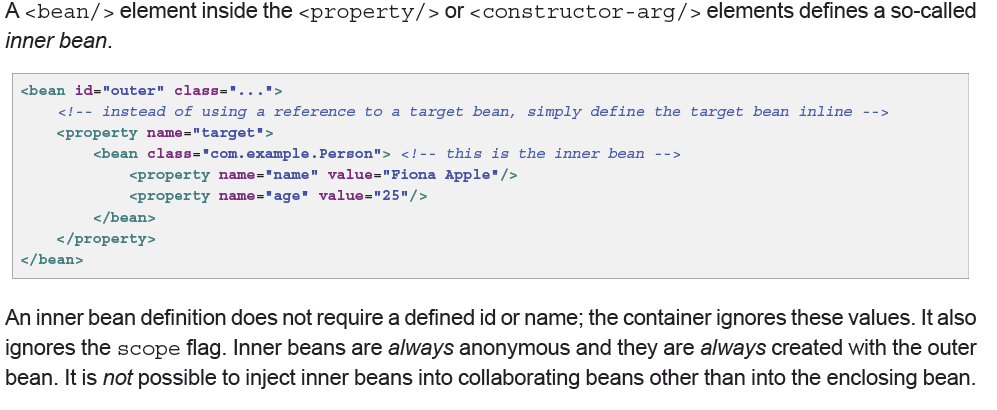
bean in the container to a <constructor-arg/> or <property/> element.



Specifying the target bean through the parent attribute creates a reference to a bean that is in a parent container of the current container. The value of the parent attribute may be the same as either the id attribute of the target bean, or one of the values in the name attribute of the target bean, and the target bean must be in a parent container of the current one. You use this bean reference variant mainly when you have a hierarchy of containers and you want to wrap an existing bean in a parent container with a

proxy that will have the same name as the parent bean.

****

****

**Collections**

In the <list/>, <set/>, <map/>, and <props/> elements, you set the properties and arguments of

the Java Collection types List, Set, Map, and Properties, respectively.

****

**Collection merging**

The Spring container also supports the *merging* of collections. An application developer can define a

parent-style <list/>, <map/>, <set/> or <props/> element, and have child-style <list/>, <map/

>, <set/> or <props/> elements inherit and override values from the parent collection.

****

Notice the use of the merge=true attribute on the <props/> element of the adminEmails property

of the child bean definition. When the child bean is resolved and instantiated by the container, the

resulting instance has an adminEmails Properties collection that contains the result of the merging

of the child’s adminEmails collection with the parent’s adminEmails collection.

administrator=administrator@example.com

sales=sales@example.com

support=support@example.co.uk

**Null and empty string values**

Spring treats empty arguments for properties and the like as empty Strings. The following XML-based

configuration metadata snippet sets the email property to the empty String value ("").

**XML shortcut with the p-namespace**

The p-namespace enables you to use the bean element’s attributes, instead of nested <property/>

elements, to describe your property values and/or collaborating beans.



**XML shortcut with the c-namespace**

Similar to the the section called “XML shortcut with the p-namespace”, the *c-namespace*, newly

introduced in Spring 3.1, allows usage of inlined attributes for configuring the constructor arguments

rather then nested constructor-arg elements.



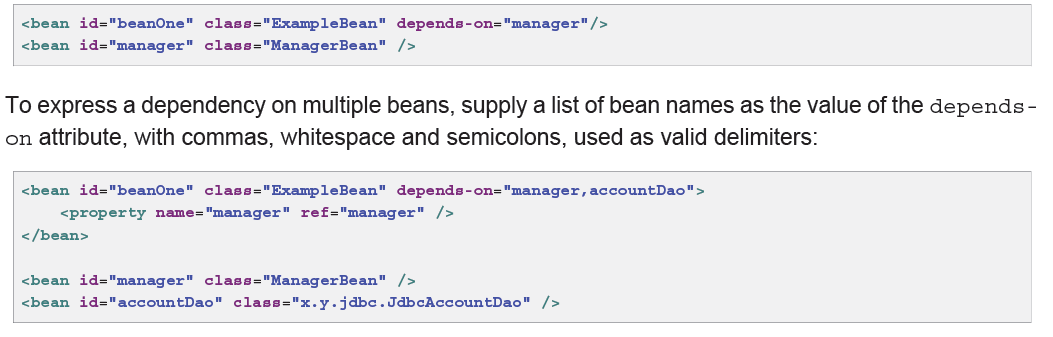
****

**Using depends-on**

The depends-on attribute can explicitly force one

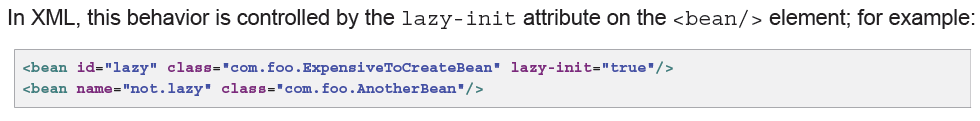
or more beans to be initialized before the bean using this element is initialized. The following example

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

****

**Lazy-initialized beans**

A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.

****

However, when a lazy-initialized bean is a dependency of a singleton bean that is *not* lazy-initialized,

the ApplicationContext creates the lazy-initialized bean at startup, because it must satisfy the

singleton’s dependencies. The lazy-initialized bean is injected into a singleton bean elsewhere that is

not lazy-initialized.

You can also control lazy-initialization at the container level by using the default-lazy-init attribute

on the <beans/> element; for example:

****

**Autowiring collaborators**

The Spring container can *autowire* relationships between collaborating beans. You can allow Spring

to resolve collaborators (other beans) automatically for your bean by inspecting the contents of the

ApplicationContext. Autowiring has the following advantages:

• Autowiring can significantly reduce the need to specify properties or constructor arguments. (Other

mechanisms such as a bean template discussed elsewhere in this chapter are also valuable in this

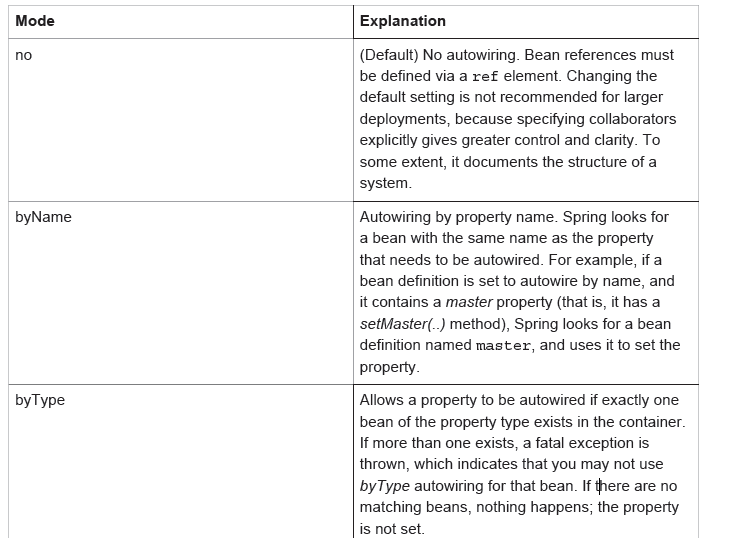
regard.)

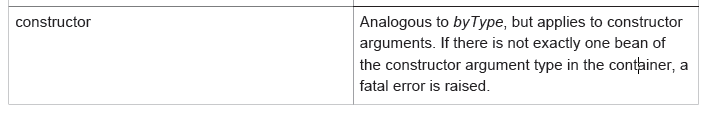
• Autowiring can update a configuration as your objects evolve. For example, if you need to add a

dependency to a class, that dependency can be satisfied automatically without you needing to modify

the configuration. Thus autowiring can be especially useful during development, without negating the

option of switching to explicit wiring when the code base becomes more stable.

****

****

**Limitations and disadvantages of autowiring**

Autowiring works best when it is used consistently across a project. If autowiring is not used in general,

it might be confusing to developers to use it to wire only one or two bean definitions.

Consider the limitations and disadvantages of autowiring:

• Explicit dependencies in property and constructor-arg settings always override autowiring.

You cannot autowire so-called *simple* properties such as primitives, Strings, and Classes (and

arrays of such simple properties). This limitation is by-design.

• Autowiring is less exact than explicit wiring. Although, as noted in the above table, Spring is careful

to avoid guessing in case of ambiguity that might have unexpected results, the relationships between

your Spring-managed objects are no longer documented explicitly.

• Wiring information may not be available to tools that may generate documentation from a Spring

container.

• Multiple bean definitions within the container may match the type specified by the setter method

or constructor argument to be autowired. For arrays, collections, or Maps, this is not necessarily

a problem. However for dependencies that expect a single value, this ambiguity is not arbitrarily

resolved. If no unique bean definition is available, an exception is thrown.

**Excluding a bean from autowiring**

On a per-bean basis, you can exclude a bean from autowiring. In Spring’s XML format, set the

autowire-candidate attribute of the <bean/> element to false; the container makes that specific bean definition unavailable to the autowiring infrastructure (including annotation style configurations

such as @Autowired).

**Method injection**

In most application scenarios, most beans in the container are singletons. When a singleton bean needs

to collaborate with another singleton bean, or a non-singleton bean needs to collaborate with another

non-singleton bean, you typically handle the dependency by defining one bean as a property of the

other. A problem arises when the bean lifecycles are different. Suppose singleton bean A needs to use

non-singleton (prototype) bean B, perhaps on each method invocation on A. The container only creates

the singleton bean A once, and thus only gets one opportunity to set the properties. The container cannot

provide bean A with a new instance of bean B every time one is needed.

A solution is to forego some inversion of control. You can make bean A aware of the container by

implementing the ApplicationContextAware interface, and by making a getBean("B") call to the

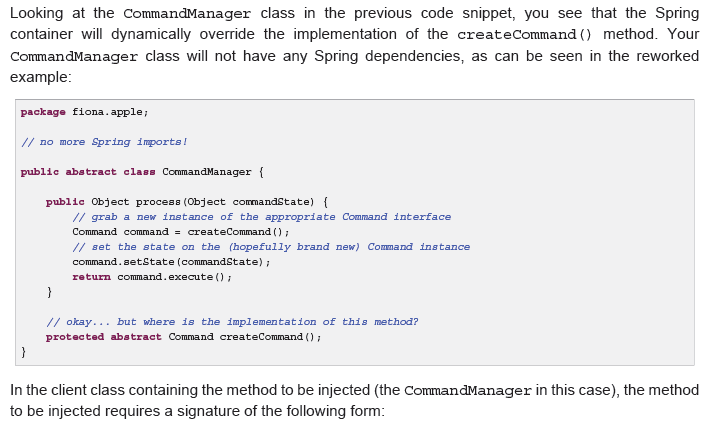
container ask for (a typically new) bean B instance every time bean A needs it. The following is an

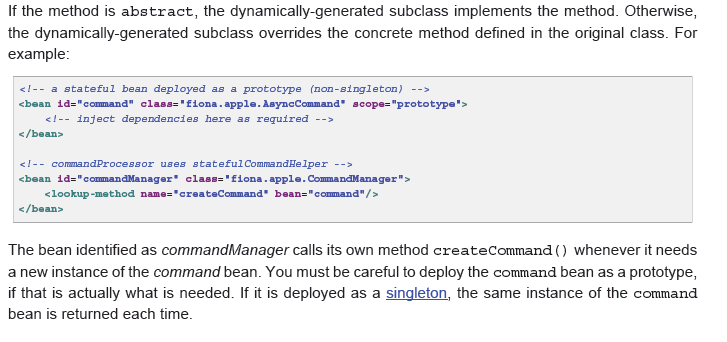
example of this approach:

The preceding is not desirable, because the business code is aware of and coupled to the Spring

Framework. Method Injection, a somewhat advanced feature of the Spring IoC container, allows this

use case to be handled in a clean fashion.

****

****

**6.5 Bean scopes**

****

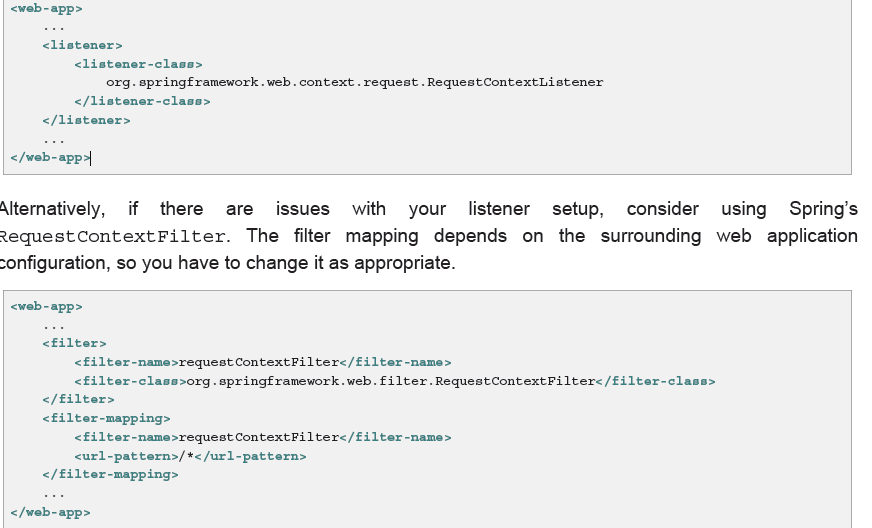
**Request, session, and global session scopes**

The request, session, and global session scopes are *only* available if you use a web-aware

Spring ApplicationContext implementation (such as XmlWebApplicationContext). If you use

these scopes with regular Spring IoC containers such as the ClassPathXmlApplicationContext,

you get an IllegalStateException complaining about an unknown bean scope.

****

**Custom scopes**

The bean scoping mechanism is extensible; You can define your own scopes, or even redefine existing

scopes, although the latter is considered bad practice and you *cannot* override the built-in singleton

and prototype scopes.

**Customizing the nature of a bean**

**Lifecycle callbacks**

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 allow the bean to perform

certain actions upon initialization and destruction of your beans.

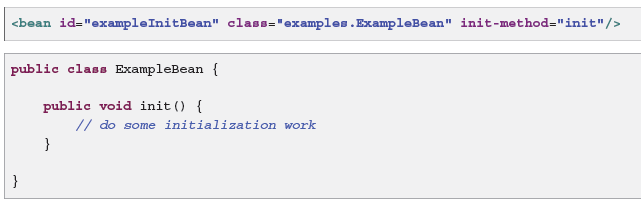
The JSR-250 @PostConstruct and @PreDestroy annotations are generally considered best

practice for receiving lifecycle callbacks in a modern Spring application. Using these annotations

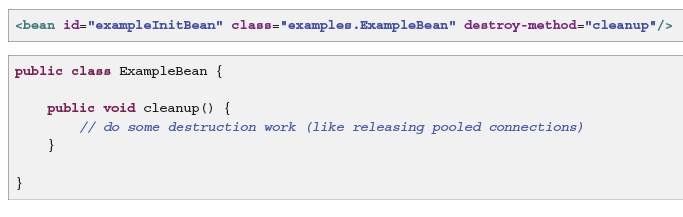
means that your beans are not coupled to Spring specific interfaces. For details see the section

called “@PostConstruct and @PreDestroy”. If you don’t want to use the JSR-250 annotations but you are still looking to remove coupling consider the use of init-method and destroy-method object definition metadata.

**Initializing a bean**



**Destroying a Bean**



**Default initialization and destroy methods**

When you write initialization and destroy method callbacks that do not use the Spring-specific

InitializingBean and DisposableBean callback interfaces, you typically write methods with

names such as init(), initialize(), dispose(), and so on. Ideally, the names of such lifecycle

callback methods are standardized across a project so that all developers use the same method names

and ensure consistency.

You can configure the Spring container to look for named initialization and destroy callback method

names on *every* bean. This means that you, as an application developer, can write your application

classes and use an initialization callback called init(), without having to configure an initmethod="

init" attribute with each bean definition. The Spring IoC container calls that method

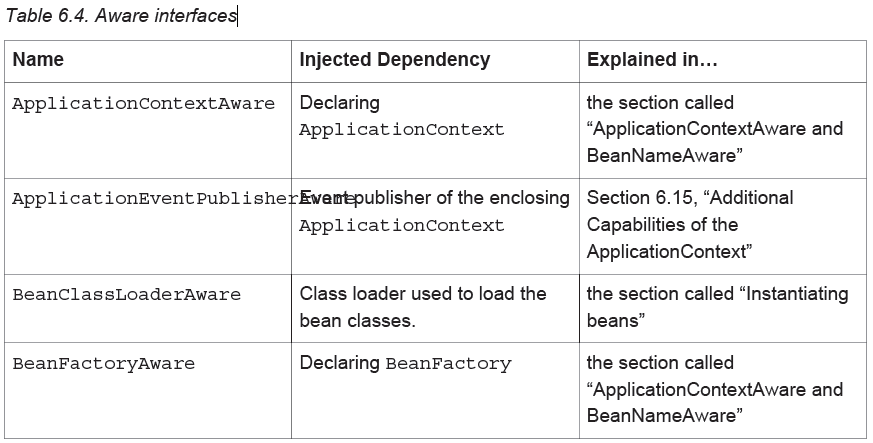
when the bean is created (and in accordance with the standard lifecycle callback contract described previously). This feature also enforces a consistent naming convention for initialization and destroy

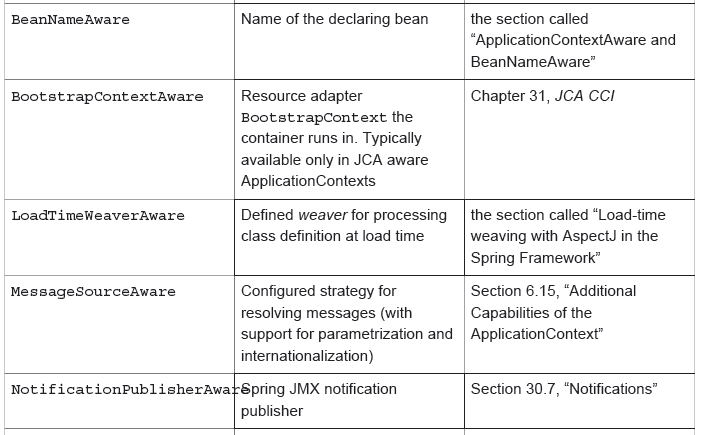
method callbacks.

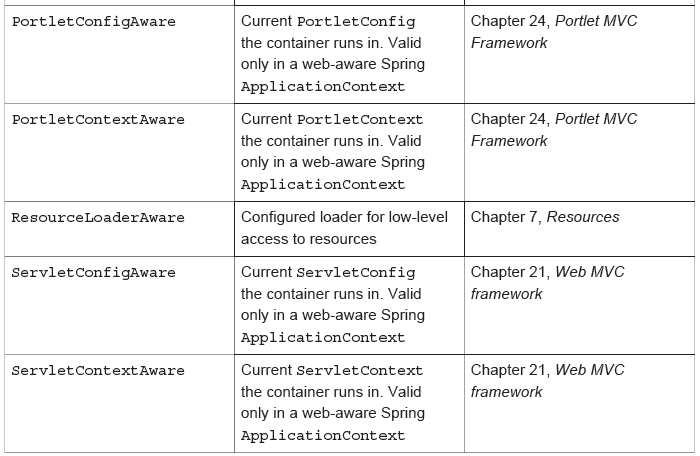
Suppose that your initialization callback methods are named init() and destroy callback methods are

named destroy(). Your class will resemble the class in the following example.

****

****

****

****

**6.8 Container Extension Points**

**Customizing beans using a BeanPostProcessor**

****

****

**Reading a values in the properties file.**

****

With the context namespace introduced in Spring 2.5, it is possible to configure property placeholders

with a dedicated configuration element. One or more locations can be provided as a comma-separated

list in the location attribute

**<context:property-placeholder location**=**"classpath:com/foo/jdbc.properties"/>**

**Annotation-based container configuration**

**Are annotations better than XML for configuring Spring?**

The introduction of annotation-based configurations 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.

**@Required -** This annotation simply indicates that the affected bean property must be populated at configuration time, through an explicit property value in a bean definition or through autowiring.

**@Autowired –** setter methods, method arguments and constructor specified as @Autowired will be automatically taken care by the container.

@Resource - The Resource annotation marks a resource that is needed by the application

**@PostConstruct and @PreDestroy –** Applied on methods which will be called for initializing and before destroying beans.

**@Component and further stereotype annotations**

Spring provides further stereotype annotations: @Component, @Service, and @Controller.

@Component is a generic stereotype for any Spring-managed component. @Repository, @Service,

and @Controller are specializations of @Component for more specific use cases, for example,

in the persistence, service, and presentation layers, respectively. Therefore, you can annotate your

component classes with @Component, but by annotating them with @Repository, @Service, or

@Controller instead, your classes are more properly suited for processing by tools or associating

with aspects. For example, these stereotype annotations make ideal targets for pointcuts. It is also

possible that @Repository, @Service, and @Controller may carry additional semantics in future

releases of the Spring Framework.

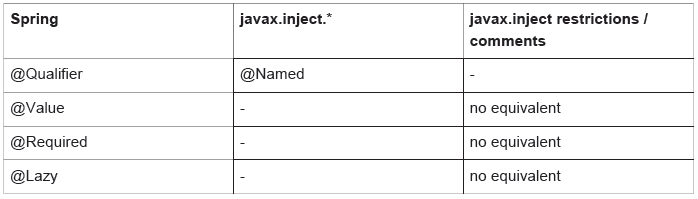
@Qualifier – To refer to the correct instance of the bean

Instead of @Autowired, @javax.inject.Inject may be used.

**@Named: a standard equivalent to the @Component annotation**

*Table 6.6. Spring annotations vs. standard annotations*





**7. Resources**

Java’s standard java.net.URL class and standard handlers for various URL prefixes unfortunately are

not quite adequate enough for all access to low-level resources. For example, there is no standardized

URL implementation that may be used to access a resource that needs to be obtained from the classpath, or relative to a ServletContext.

**Built-in Resource implementations**

**UrlResource**

The UrlResource wraps a java.net.URL, and may be used to access any object that is normally

accessible via a URL, such as files, an HTTP target, an FTP target, etc.

Ex:

Resource template = ctx.getResource(***"http://myhost.com/resource/path/myTemplate.txt"***);

**ClassPathResource**

This class represents a resource which should be obtained from the classpath.

Ex: Resource template = ctx.getResource(***"classpath:some/resource/path/myTemplate.txt"***);

**FileSystemResource**

This is a Resource implementation for java.io.File handles. It obviously supports resolution as a

File, and as a URL.

**ServletContextResource**

This is a Resource implementation for ServletContext resources, interpreting relative paths within

the relevant web application’s root directory.

**InputStreamResource**

A Resource implementation for a given InputStream. This should only be used if no specific

Resource implementation is applicable. In particular, prefer ByteArrayResource or any of the filebased

Resource implementations where possible.

**ByteArrayResource**

This is a Resource implementation for a given byte array. It creates a ByteArrayInputStream for

the given byte array.

**7.5 The ResourceLoaderAware interface**

The ResourceLoaderAware interface is a special marker interface, identifying objects that expect to

be provided with a ResourceLoader reference.

**7.7 Application contexts and Resource paths**

**Constructing application contexts**

ApplicationContext ctx = **new** ClassPathXmlApplicationContext(***"conf/appContext.xml"***);

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext(***"conf/appContext.xml"***);

Note that the use of the special classpath prefix or a standard URL prefix on the location

path will override the default type of Resource created to load the definition. So this FileSystemXmlApplicationContext…

ApplicationContext ctx =

**new** FileSystemXmlApplicationContext(***"classpath:conf/appContext.xml"***);

will actually load its bean definitions from the classpath. However, it is still a

FileSystemXmlApplicationContext. If it is subsequently used as a ResourceLoader, any

unprefixed paths will still be treated as filesystem paths.

**Validation, Data Binding, and Type Conversion**

There are pros and cons for considering validation as business logic, and Spring offers a design for

validation (and data binding) that does not exclude either one of them. Specifically validation should

not be tied to the web tier, should be easy to localize and it should be possible to plug in any validator

available. Considering the above, Spring has come up with a Validator interface that is both basic

and eminently usable in every layer of an application.

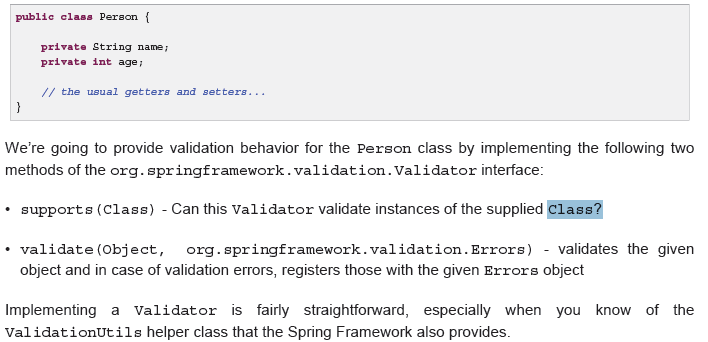
Data binding is useful for allowing user input to be dynamically bound to the domain model of

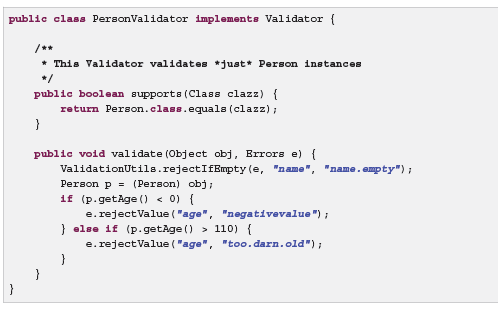
an application (or whatever objects you use to process user input). Spring provides the so-called

DataBinder to do exactly that. The Validator and the DataBinder make up the validation

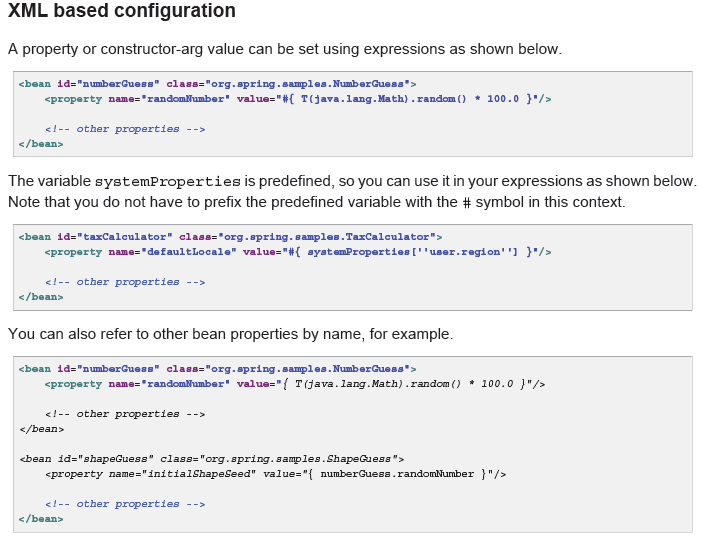
package, which is primarily used in but not limited to the MVC framework.

Ex:





**9. Spring Expression Language (SpEL)**

****

**Aspect Oriented Programming with Spring**

**References:**

**http://docs.spring.io/spring-framework/docs/4.2.0.RELEASE/spring-framework-reference/pdf/spring-framework-reference.pdf**