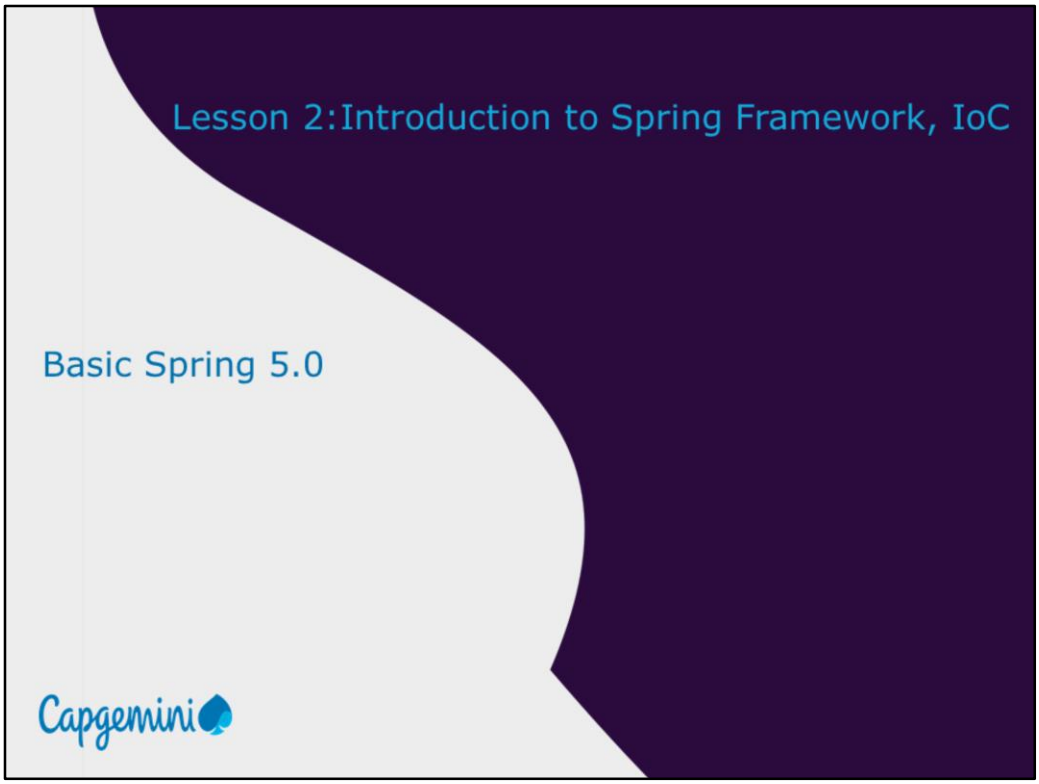
Instructor Notes:

Add instructor notes here.



Instructor Notes:

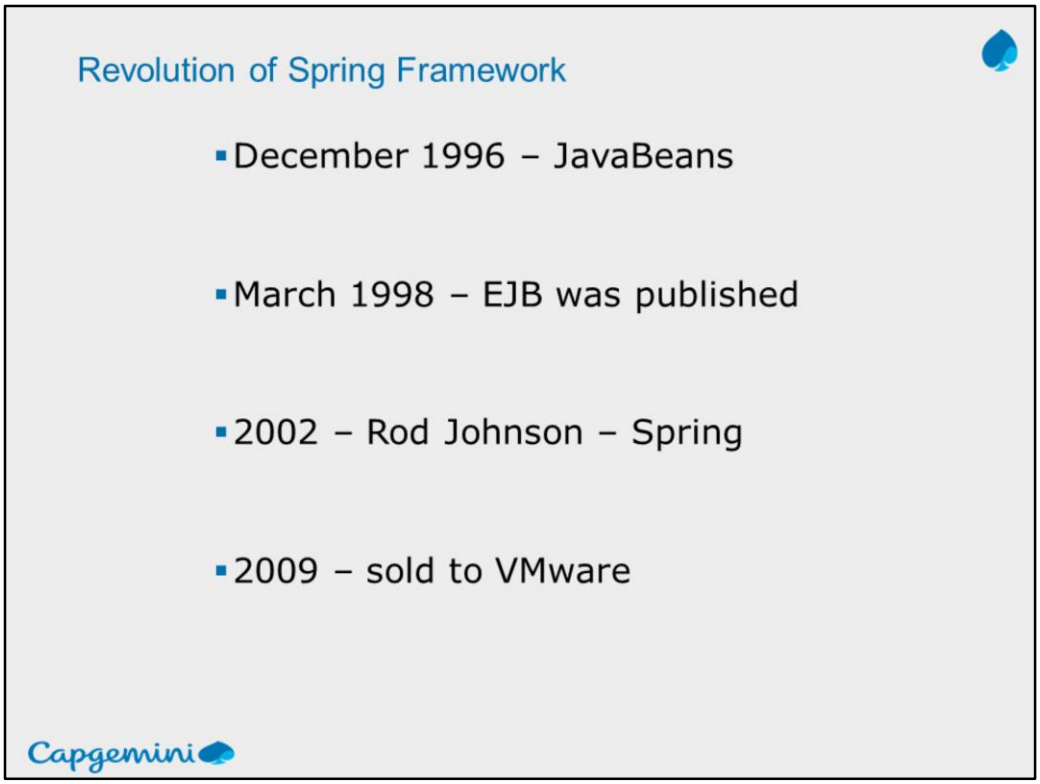
Explain the lesson coverage

Lesson Objectives

- Revolution of Spring Framework
- Benefits
- Architecture
- IoC (Inversion of control)
- Dependency Injection
- Bean Factory
- Configurations – XML and Annotations
- Bean Autowiring
- Bean containers
- Bean life-cycle
- BeanPostProcessors

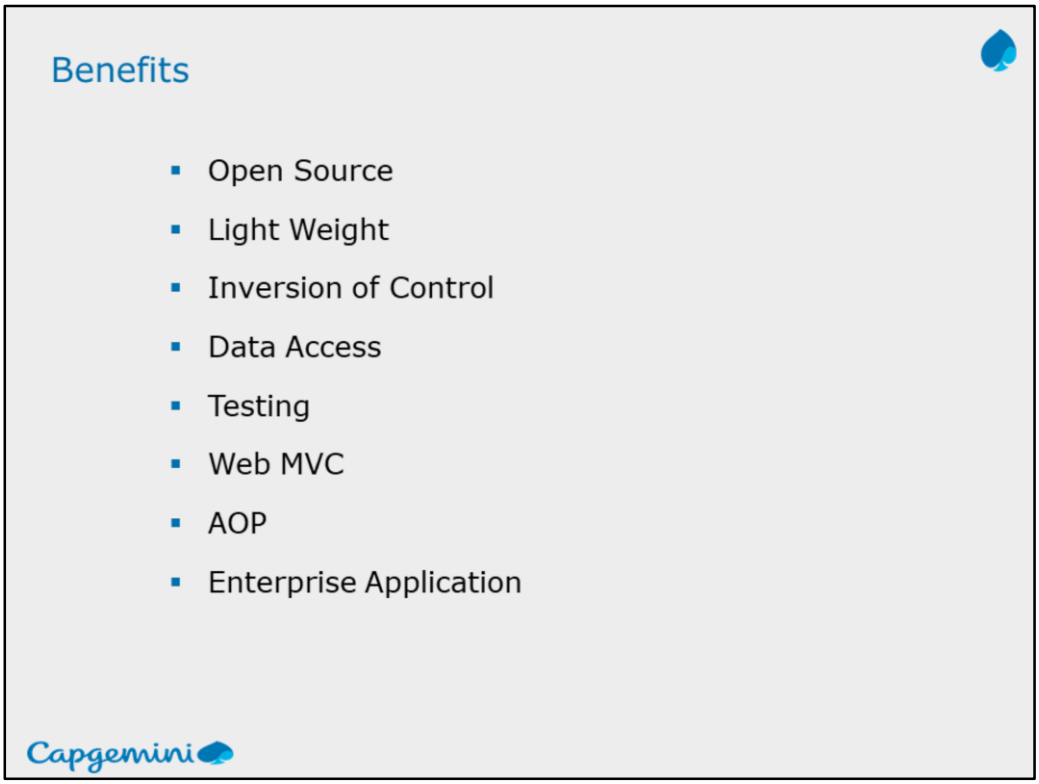
Capgemini

Instructor Notes:



It all started with a bean. In 1996, the Java programming language was still a young, exciting, up-coming platform. Many developers flocked to the language because they had seen how to create rich and dynamic web applications using applets. But they soon learned that there is more to this strange new language than juggling animated cartoon characters. Unlike any language before it, Java made it possible to write complex applications made up of discrete parts. They came for the applets, but stayed for the components.
It was in December of that year that Sun Microsystems published the Java-Beans 1.00-A specification. JavaBeans defined a software component model for Java. This specification defined a set of coding policies that enabled simple Java objects to be reusable and easily composed into more complex applications. Although JavaBeans were intended as a general-purpose means of defining reusable application components, they have been primarily used as a model for building user interface widgets. They seemed too simple to be capable of any “real” work; enterprise developers wanted more.
Sophisticated applications often require services that are not directly provided by the JavaBeans specification such as transaction support, security, and distributed computing. Therefore in March 1998, Sun published the 1.0 version of the Enterprise JavaBeans (EJB) specification. This specification extended the notion of Java components to the server side, providing the much-needed enterprise services, but failed to continue the simplicity of the original JavaBeans specification. In fact, except in name, EJB bears very little resemblance to the original Javabeans specification.

Instructor Notes:



Despite the fact that many successful applications have been built based on EJB, it never really achieved its intended purpose: to simplify enterprise application development. It is true that EJB's declarative programming model simplifies many infrastructural aspects of development, such as transactions and security. But EJBs are complicated in a different way by mandating deployment descriptors and plumbing code (home and remote/local interfaces). As a result, its popularity has started to wane in recent years, leaving many developers looking for an easier way.

Now Java development is coming full circle. New programming techniques, including aspect-oriented programming (AOP) and inversion of control (IoC), are giving JavaBeans much of the power of EJB. These techniques furnish JavaBeans with a declarative programming model reminiscent of EJB, but without all of EJB's complexity. No longer must you resort to writing an unwieldy EJB component when a simple JavaBean will suffice.

And that's where Spring steps into the picture.

In all fairness, the latest EJB specification (EJB 3) has evolved to promote POJO-based programming model and is simpler than its predecessors.

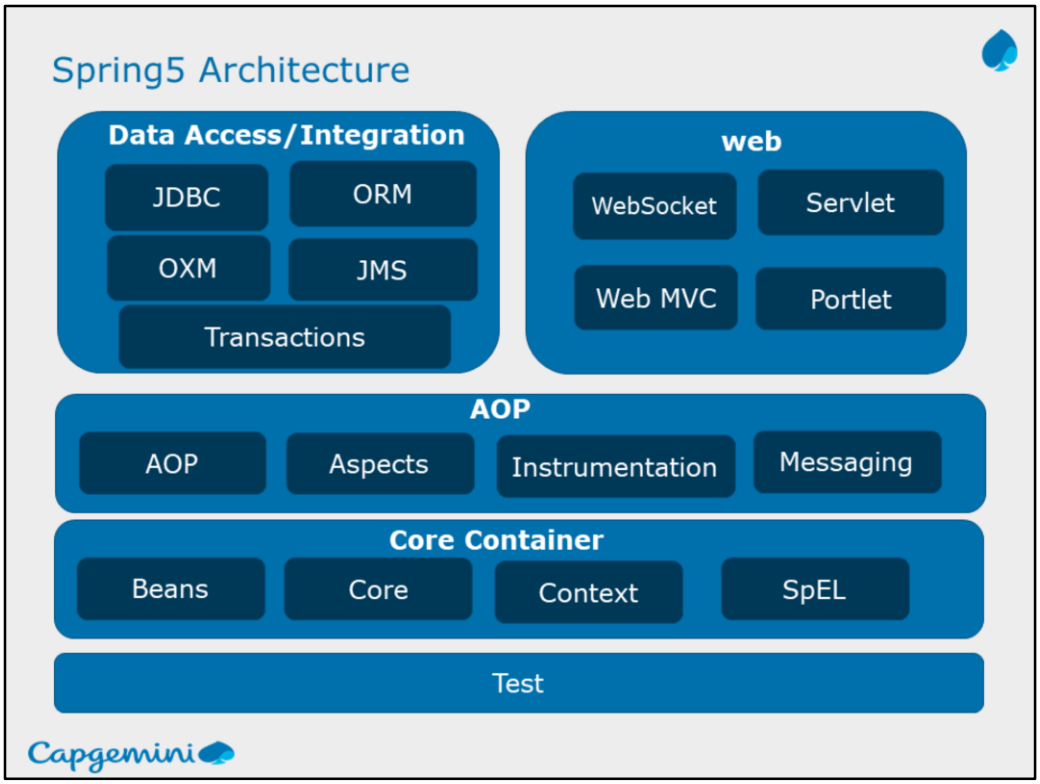
Qu

Spring Framework project founded in Feb 2003
Release 1.0 in Mar 2004
Release 1.2 in May 2005
Release 2.0 in Oct 2006
Release 2.5 in Nov 2007
Release 3.0 in Dec 2009
Release 4.0 in Dec 2013

Instructor Notes:

Spring could potentially be a one-stop-shop for all your enterprise applications. However, Spring is modular, allowing you to use parts of it, without having to bring in the rest. You can use the bean container, with Struts on top, or you could choose to just use the Hibernate integration or the JDBC abstraction layer.

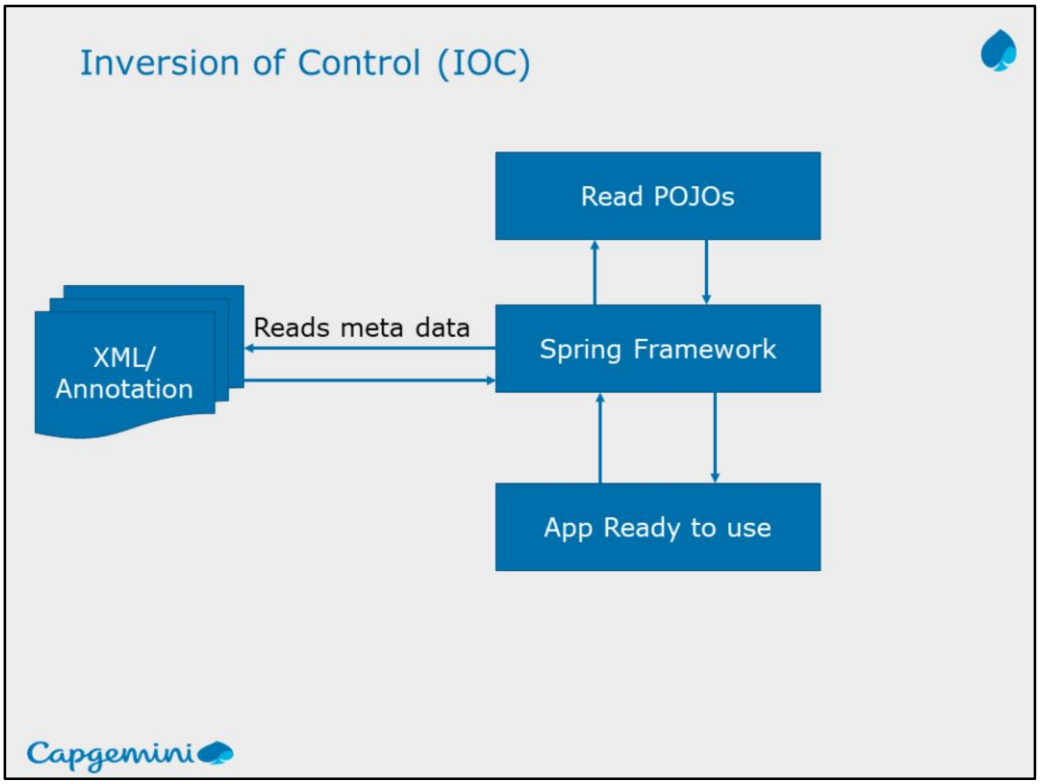
The spring.jar artifact that contained almost the entire framework in pre Spring 3.0 is no longer provided. The framework modules have been revised and are now managed separately with one source-tree per module jar



The Spring Framework is composed of about 20 modules. These modules are grouped into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming),
Instrumentation, Messaging, and Test. When you download and unzip the Spring framework distribution, you'll find many different JAR files in the dist directory for every modules. When taken as a whole, these modules give you everything you need to develop enterprise-ready applications. But this modularity also gives you the freedom to choose the modules that suit your application.

Core Container : The Core Container consists of the spring-core, spring-beans, spring-context, spring-context-support, and spring-expression (Spring Expression Language) modules.
The spring-core and spring-beans modules are the most fundamental part of the framework. It defines how beans are created, configured and managed. The BeanFactory (a sophisticated implementation of the factory pattern and a primary component of this module) applies the Inversion of Control (IOC) pattern to separate an application's configuration and dependency specification from the actual application code
The spring-context module builds on the solid base provided by the Core and Beans modules. The core modules bean factory makes Spring a container, but the context modules makes it a framework. The Context module inherits its features from spring-beans module and adds support for internationalization, event-propagation, resource-loading etc.
The spring-expression module provides a powerful expression language for querying and manipulating an object graph at runtime.

Instructor Notes:



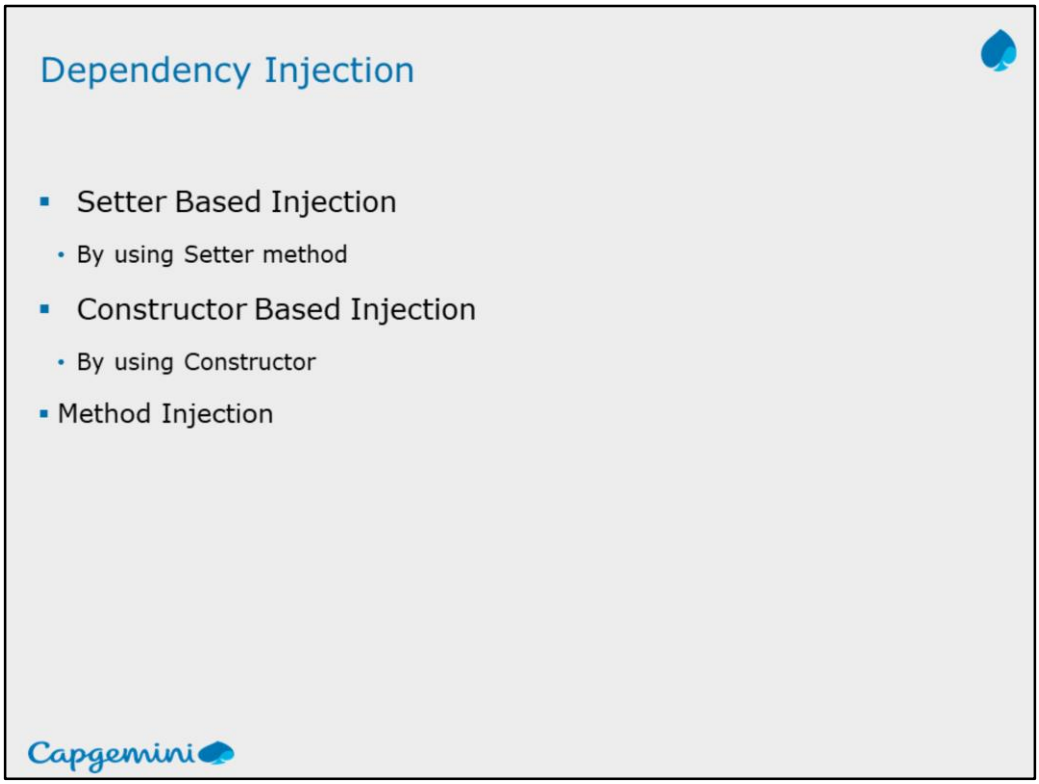
Understanding inversion of control: Inversion of control is at the heart of the Spring framework. As seen earlier, any non-trivial application is made up of two or more classes that collaborate with each other to perform some business logic. Traditionally, each object is responsible for obtaining its own references to the objects it collaborates with (its dependencies). This can lead to a highly coupled and hard-to-test code.
Applying IoC, objects are given their dependencies at creation time by some external entity that coordinates each object in the system ie dependencies are injected into objects. So, IoC means an inversion of responsibility with regard to how an object obtains references to collaborating objects.

Dependency injection is kind of an Inversion of Control pattern. The term dependency injection describes the process of providing (or injecting) a component with the dependencies it needs, in an IoC fashion. Dependency Injection proposes separating the implementation of an object and the construction of objects that depend on them.
Dependency injection is a form of PUSH configuration; the container pushes dependencies into application objects at runtime. This is opposite to the traditional PULL configuration in which the app object pulls dependencies from its environment.
In the figure, we have application objects offering external services. The application components depend on these external services. The job of coordinating the implementation and construction is left to the assembler code, which In this case would be the spring IoC framework.
The Spring IOC container can manage any class you want it to manage; it is not limited to managing true JavaBeans
Spring container can also handle non-bean-style classes
Spring IoC container manages one or more beans
In the Spring IOC container, bean definitions are represented as BeanDefinition objects, with metadata:

- A package-qualified class name
- Bean behavioural configuration elements
- References to other beans, collaborators or dependencies
- Configuration settings to set in the newly created object, for example, the number of connections to use in a bean that manages a connection pool, or the size limit of the pool

Instructor Notes:

Additional notes for instructor



Dependency Injection
Any enterprise application has objects that depend on each other
Resolving the dependency is termed as 'Injecting Dependency' which facilitates loose coupling.
Choosing the low level implementation to be injected into the reference of the interface in higher level layer, is termed as 'Inversion Of Control'
Thus the choice of low level dependency has control on the quality of the service that will be provided, this is configurable/changeable making the framework highly flexible and modular
For an example, if a person need to have a cup of coffee then either he can prepare by himself using ingredients such as coffeebean, milk, sugar,.. Or he can raise request in vending machine, so that coffee will be prepared and automatically delivered to him.
Similarly, instead of creating object manually with the use of new operator, object creation will be taken care by container using DI.

In DI or IOC process objects define their dependencies, only through constructor arguments, arguments
to a factory method, or properties that are set on the object instance after it is constructed or returned
from a factory method. The container (the environment provided to the Spring Beans, for wiring) then injects those dependencies when it creates the spring bean, this process is an inverse of traditional approach, 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.
The IoC pattern uses three different approaches to achieve decoupling of control of services from your components:
Type 1 : Interface injection: This is how most J2EE worked. Components are explicitly conformed to a set of interfaces with associated configuration metadata, in order to allow framework to manage them correctly.
Type 2 : Setter Injection: External metadata is used to configure how components can interact. Our first example used this approach, by using a Springconfig.xml file.
Type 3 : Constructor injection: Components are registered with the framework, including the parameters to be used when the components are constructed, and the framework provides instances of the component with all the specified facilities applied. Our last example used this approach.
There is a fourth approach called "Lookup-method injection". This has been covered in detail in Appendix-B.

Instructor Notes:

Bean Factory

- Bean Factory
- ApplicationContext/AbstractApplicationContext / AnnotationConfigApplicationContext:
  - ClassPathXMLApplicationContext
  - FileSystemXMLApplicaitonContext
  - WebApplicationContext

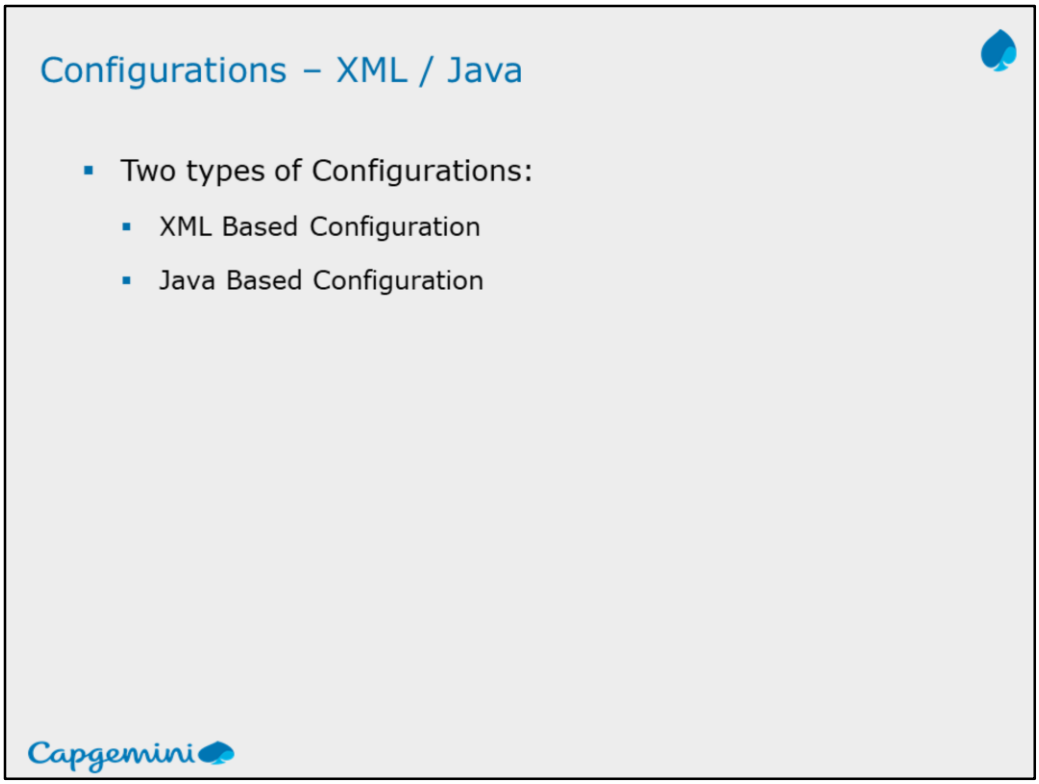
Capgemini

Loose coupling is one of the critical elements in object-oriented software development. It allows you to change the implementations of two related objects without affecting the other object. Strong coupling directly affects scalability of an application. Tightly coupled code is difficult to test, reuse and understand. On the other hand, completely uncoupled code doesn't do anything. In order to do anything useful, classes need to know about each other somehow. Coupling is necessary but it must be managed very carefully. A common technique used to reduce coupling is to hide implementation details behind interfaces so that actual implementation class can be swapped out without impacting the client class.
That is what IoC is all about: the responsibility of coordinating collaboration between dependent objects is transferred away from the objects themselves.
This is where lightweight framework containers like Spring come into play.
IoC introduces the concept of a framework of components that in turn has many similarites to a J2EE container. The IoC framework seperates facilities that your components are dependent upon and provides the "glue" for connecting the components.
With Inversion of Control (IoC), you can achieve loose coupling between several interacting components in an application.

The core of Spring's DI container is the BeanFactory. A bean factory is responsible for managing components and their dependencies. We shall see bean factories in detail later in this session. In Spring, the term "bean" is used to refer to any component managed by the container. Typically, beans adhere to Javabeans specification

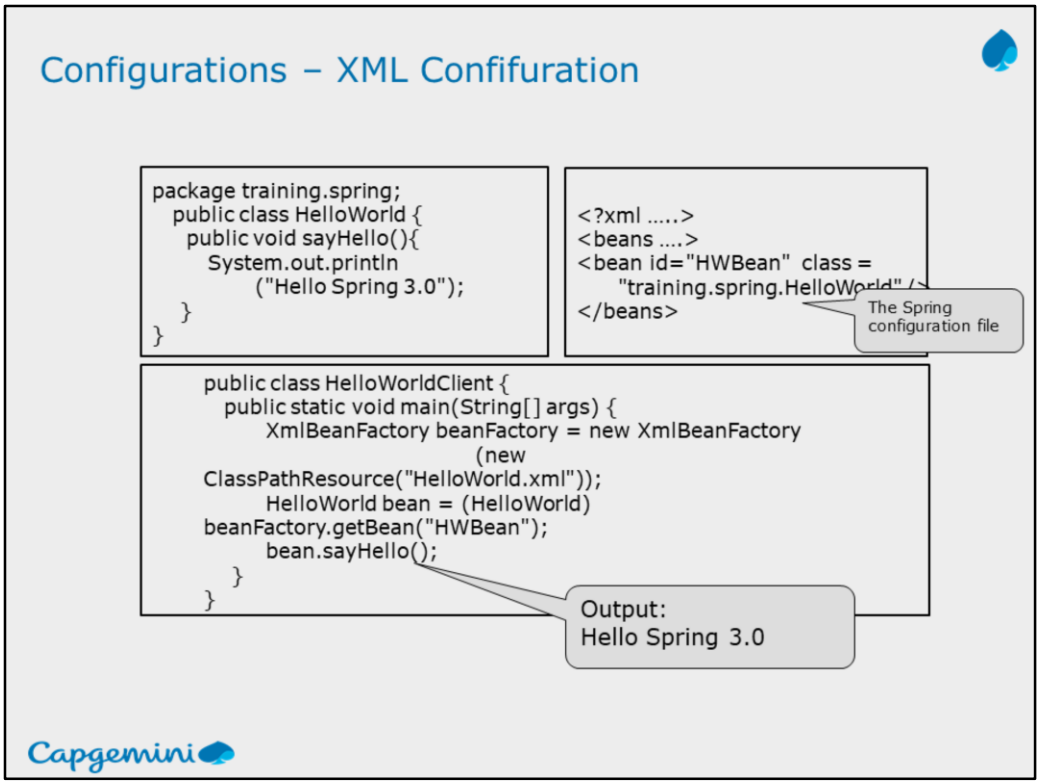
Instructor Notes:

Additional notes for instructor



Instructor Notes:

Instructor to explain this code and execute the demo

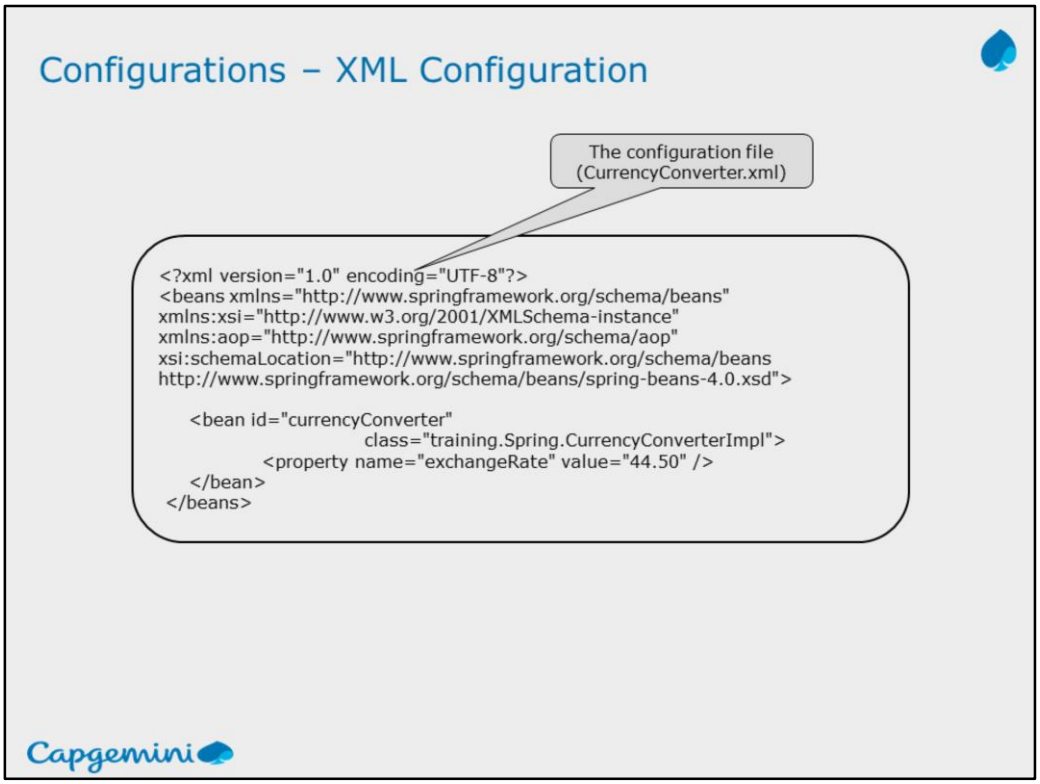


Dependency injection is the most basic thing that Spring does. We shall be covering this in detail later in the session. For now, let us see how Spring works with an example.
Spring-enabled applications are like any Java application. They are made up of several classes, each performing a specific purpose within the application. Difference lies in how these classes are configured and introduced to each other. Typically a Spring application has an XML file that describes how to configure the classes, known as Spring configuration file.

Pls. refer to Appendix-A for a detailed explanation of converting a traditional Java application into a Spring-based application.

Look in slide above for a typical Hello World application using Spring Framework. Detailed explanation for this code is given in subsequent demos.

Instructor Notes:



Question now is who will make a call to either the constructor or the setExchangeRate() method to set the exchangeRate property? The Spring configuration file in the above listing tells how to configure the CurrencyConverter service. This XML file declares an instance of a CurrencyConverterImpl in the Spring container and configures its exchangeRate property with a value of 44.50.

Notice the <beans> element at the root of the XML file. This is the root element of any Spring configuration file. The <bean> element is used to tell the Spring container about a class and how it should be configured. The id attribute is used to name the bean currencyConverter and the class attribute specifies the bean's fully qualified class name.

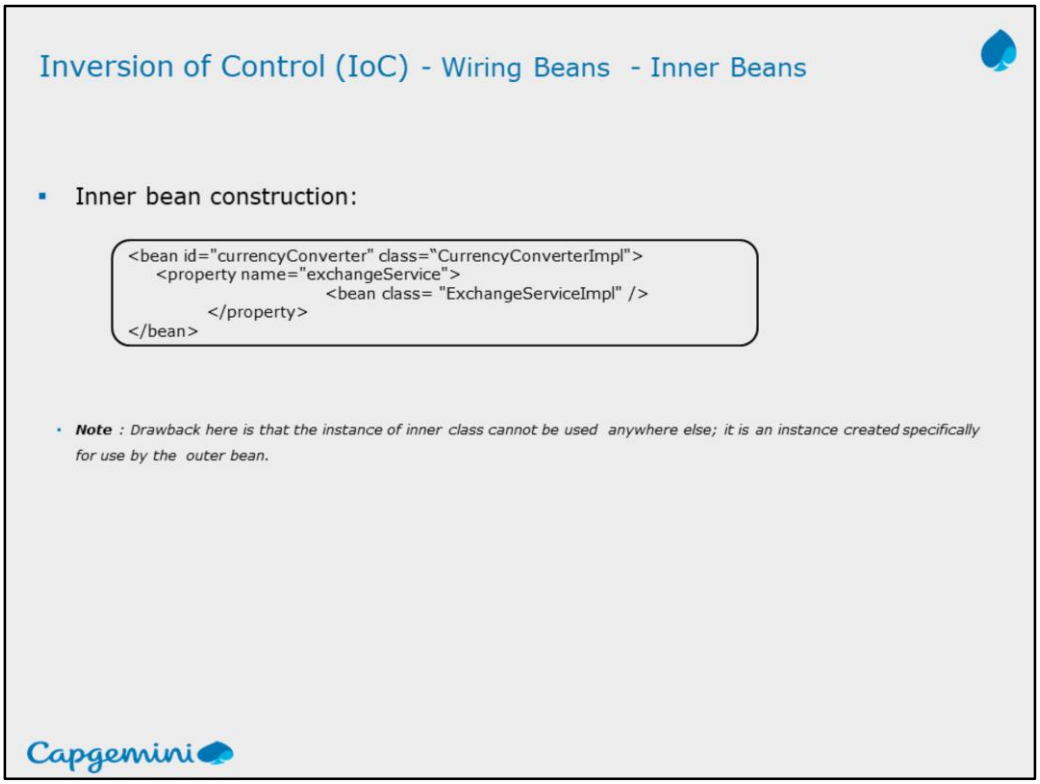
Within the <bean> element, the <property> element is used to set a property, in this case exchangeRate property. By using <property>, we are telling the Spring container to call setExchangeRate() when setting the property. This is called setter injection and is a straightforward way to configure and wire bean properties. The value of the exchange rate is defined using the value attribute.

The following snippet of code illustrates roughly what the container does when instantiating the currencyConverter service based on the XML definition seen above.

```
CurrencyConverterImpl currencyConverter = new CurrencyConverterImpl();
currencyConverter.setExchangeRate(44.50);
```

Notice the configuration metadata is represented in XML. But it can also be done using Java annotations, or Java code.

Instructor Notes:



The drawback of the above method is that you cannot reuse the instance of ExchangeServiceImpl anywhere else – it is an instance created by specifically for use by the currencyConverter bean.

Inner beans aren't limited to setter injection. You may also wire inner beans into constructor arguments. E.g.:

```
<bean id="currencyConverter"
                class="com.Spring.CurrencyConverterImpl4">
    <constructor-arg>
        <bean class="com.Spring.ExchangeServiceImpl" />
    </constructor-arg>
</bean>
```

Instructor Notes:

Recall: The key difference between the <props> and <map> is that when using <props>, both the keys and values are Strings, whereas <map> allows keys and values of any type.

IOC – using Collections

- List - <list>
- Set - <set>
- Map - <map>
- Properties - <props>

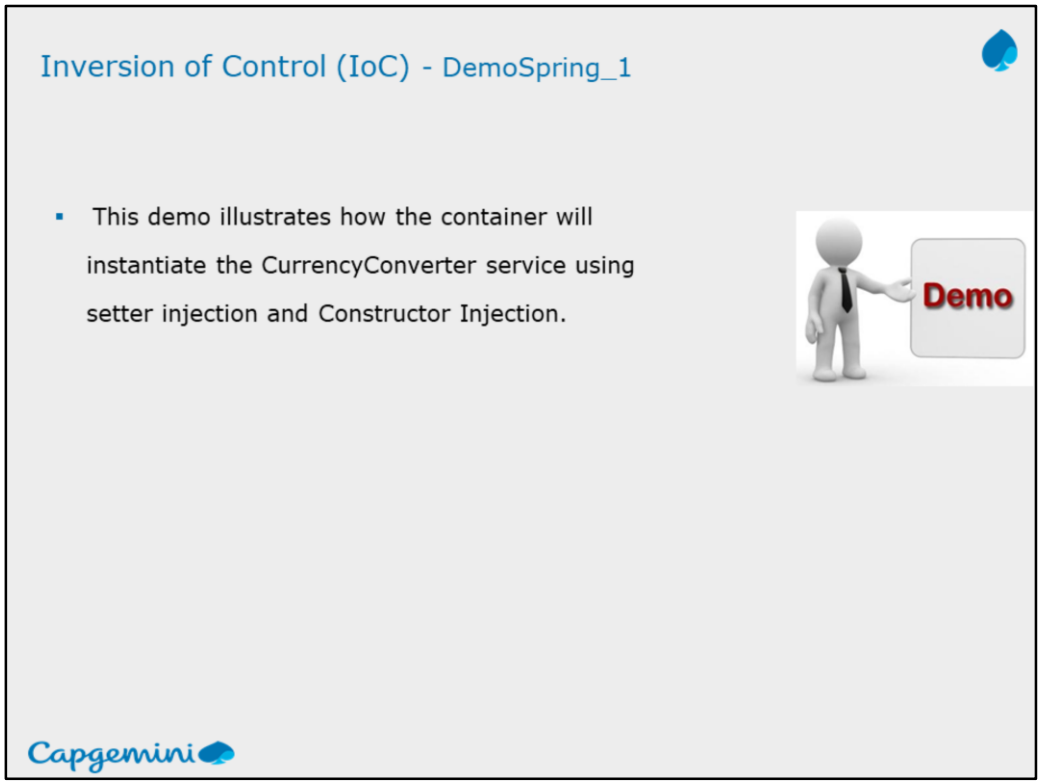
Capgemini

Example:

```
<bean id="complexObject" class="example.ComplexObject">
  <property name="people">
    <props>
        <prop key="HarryPotter">The magic property</prop>
         <prop key="JerrySeinfeld">The funny property</prop>
    </props>
  </property>
  <property name="someList">
     <list>
          <value>red</value>
           <value>blue</value>
     </list>
   </property>
<property name="someMap">
    <map>
        <entry key="an entry" value="just some string"/>
        <entry key ="a ref" value-ref="myDataSource"/>
    </map>
</property>
</bean>
```

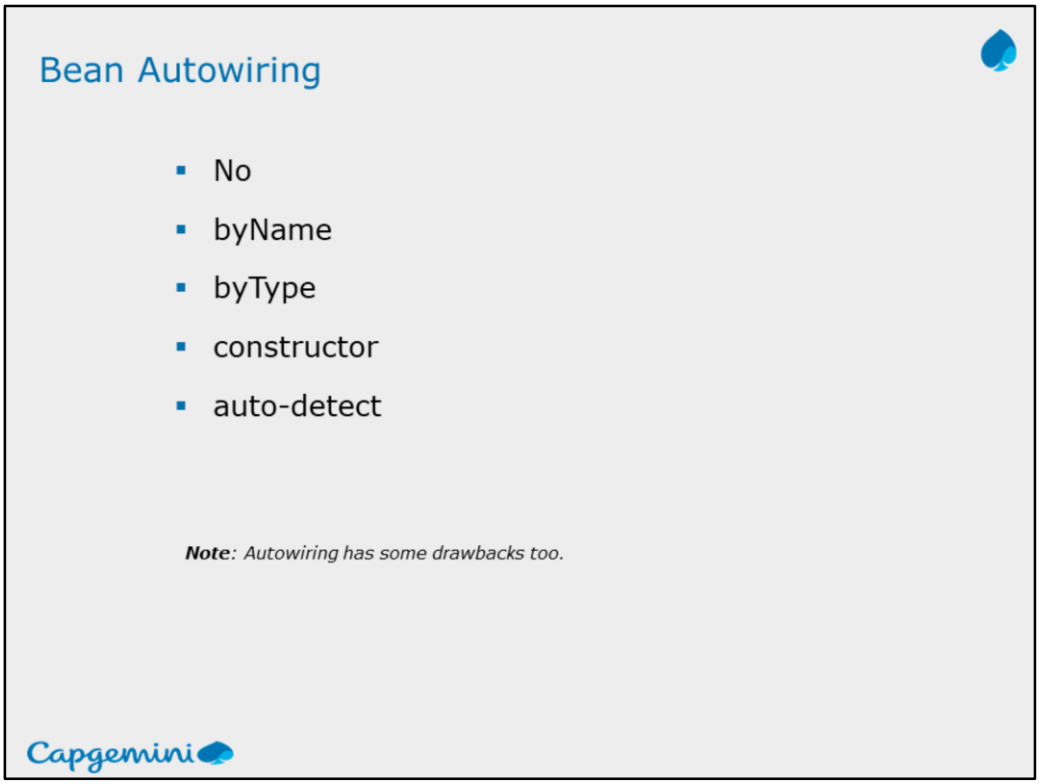
One example is also available. Refer to demo, DemoSpring_5

Instructor Notes:



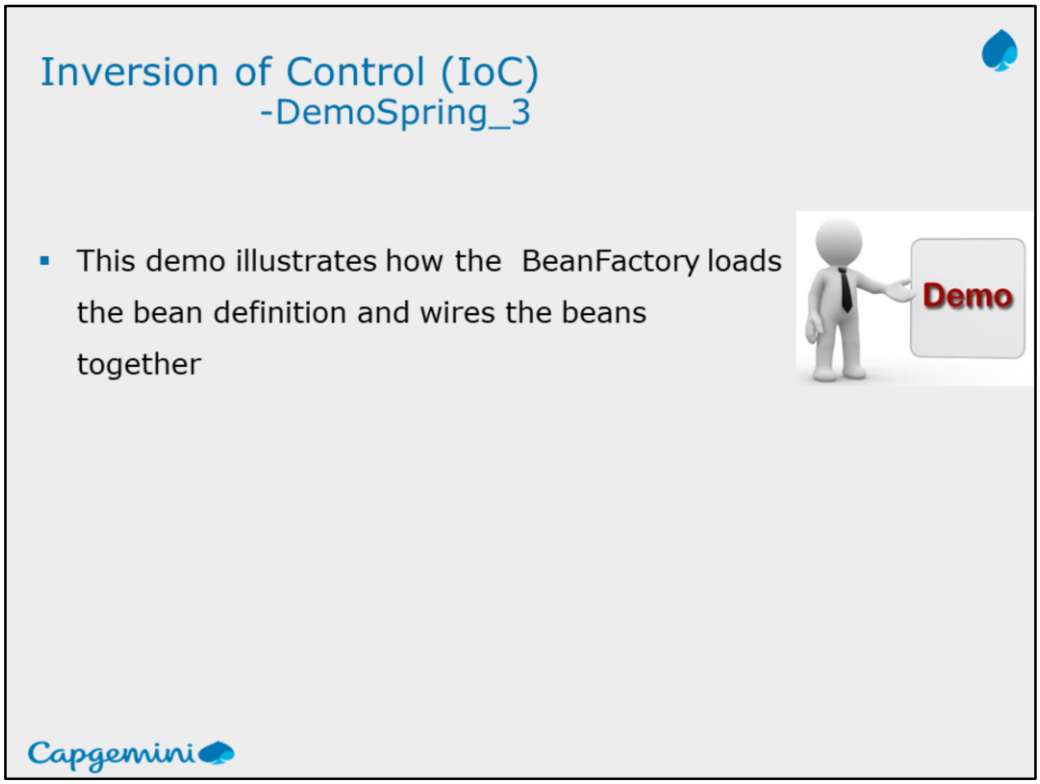
Please refer to demo, DemoSpring_1.

Instructor Notes:



So far, you have seen how to wire all your bean's properties explicitly. You can also have Spring wire them automatically by setting the autowire property on each bean that you want autowired.
There are four types of autowiring:
byName: Attempts to find a bean in the container whose name (or id) is same as the name of the property being wired. If matching bean is not found, then property will remain unwired.
byType: Attempts to match all properties of the autowired bean with beans whose types are assignable to the properties. Properties for which there's no matching bean will remain unwired.
constructor : Tries to match a constructor of the autowired bean with beans whose types are assignable to the constructor arguments. In the event of ambigous beans or ambigous constructors, an org.Springframework.beans.factory.UnsatisfiedDependencyException will be thrown.
autodetect : Attempts constructor autowiring first. If that fails, attempts autowiring byType.

Instructor Notes:



Please refer to demo, DemoSpring_3. CurrencyConverterClient.java uses an XmlBeanFactory (a BeanFactory implementation) to load currencyconverter.xml and to get a reference to the CurrencyConverter object. It then invokes the dollarsToRupees() method.
There are two application objects. The instance of ExchangeRateImpl is responsible for retriving the exchange rate, using which, the currency converter instance would convert currency. When the CurrencyConverterImpl is instantiated, it in turn first instantiates the ExchangeServive bean. The ExchangeService instance in the CurencyConverterImpl class then exposes its getExchangeRate() method to return the exchange rate.

Summarizing how the container initializes and resolves bean dependencies:
The container first initializes the bean definition, without initializing the bean itself, typically at the time of the container startup. The bean dependencies may be explicitly expressed in the form of constructor arguments or arguments to a factory method and/or bean properties.
Each property or constructor argument in a bean definition is either an actual value to set, or a reference to another bean in the bean factory.
Constructor arguments or bean properties that refer to another bean will force the container to create or obtain that other bean first. Effectively, the referred bean is a dependent of the calling bean. This can trigger a chain of bean creation.
Every Constructor argument or bean property must be able to be converted from String format to the actual format expected. Spring is able to convert from String to built-in scalar types like int, float etc. Spring uses JavaBeans PropertyEditors to convert all other types.

Instructor Notes:

Inversion of Control (IoC) -DemoSpring_4

- This demo illustrates automatically wiring your beans

Capgemini

Refer to demo, DemoSpring_4 .This uses the same exchange service and currency converter service seen in earlier examples. However, by using the autowire attribute in the currencyconverter.xml we can execute the client program even without specifying the exchange service.

```
<bean id="exchangeService" class="com.capgemini.intro.ExchangeService">
  <constructor-arg><value>44.25</value></constructor-arg>
</bean>
```

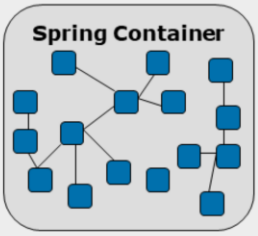
```
<bean id="currencyConverter"
class="com.capgemini.intro.CurrencyConverter" autowire="byName"/>
```

```
public class CurrencyConverter
{
private ExchangeService exchangeService;
..... }
```

Instructor Notes:

Bean containers: concept

- Bean Containers known as Bean Factory
- Responsible to create and dispense beans.
- Bean Life Cycle
- Two types of containers:
  - Bean factory
  - Application context

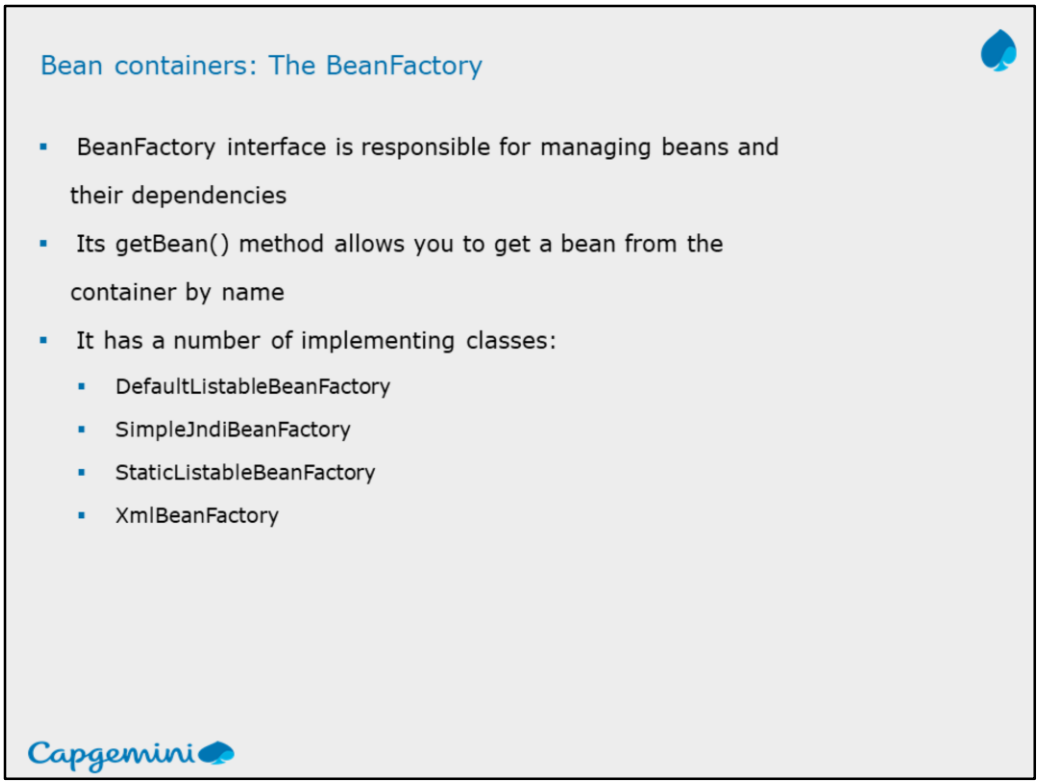


Capgemini

The container is at the core of the Spring framework and uses IoC to manage components. The basic IoC container in Spring is called the bean factory. Any bean factory allows the configuration and wiring of objects using dependency injection, in a consistent and workable fashion. A bean factory also provides some management of these objects, with respect to their lifecycles. Thus, Bean factory is a class whose responsibility is to create and dispense beans.
A bean factory knows about many objects within an application.
Able to create associations between collaborating objects as they are instantiated. This removes the burden of configuration from the bean itself and the bean's client. As a result, when a bean factory hands out objects, those objects are fully configured, aware of their collaborating objects and ready to use.
A bean factory also takes part in the life cycle of a bean, making calls to custom initialization and destruction methods, if those methods are defined.

Spring actually comes with two different types of containers:
Beanfactory interface: provides an advanced configuration mechanism capable of managing any type of object.
ApplicationContext interface : is a sub-interface of BeanFactory. It allows easier integration with Spring's AOP features, message resource handling, event publication, and application-layer specific contexts such as the WebApplicationContext for use in web applications.
We shall look at the Application context in detail later.

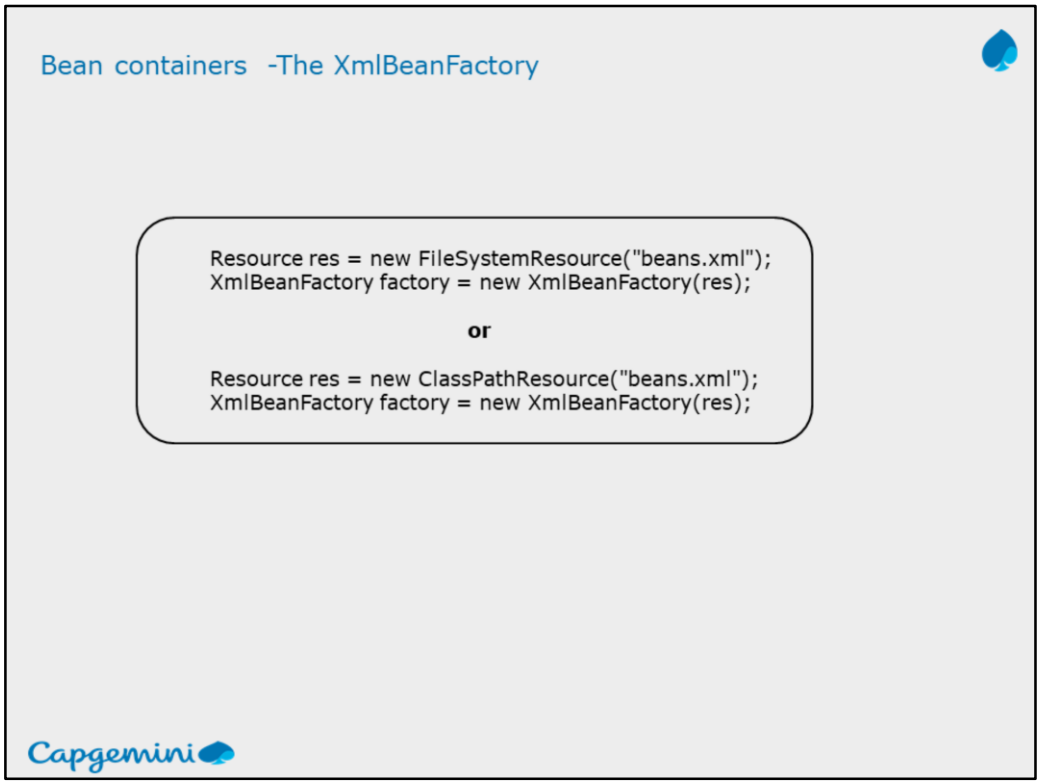
Instructor Notes:



Lets start our exploration of Spring containers with the most basic of Spring containers : the BeanFactory.
Bean factory is responsible for managing beans and their dependencies. Your application interacts with Spring DI container via the BeanFactory interface. It has a getBean() method that allows you to get a bean from the container by name. Additional methods allow you to query the bean factory to see if bean exists, to find the type of bean and to find if a bean is configured as a singleton. Different bean factory implementations exist to support varying levels of functionality with XmlBeanFactory being the most common representation. A partial listing of the impelmenting classes follows:
DefaultListableBeanFactory
SimpleJndiBeanFactory
StaticListableBeanFactory
XmlBeanFactory

Please refer to the Spring documentation for more information on these classes

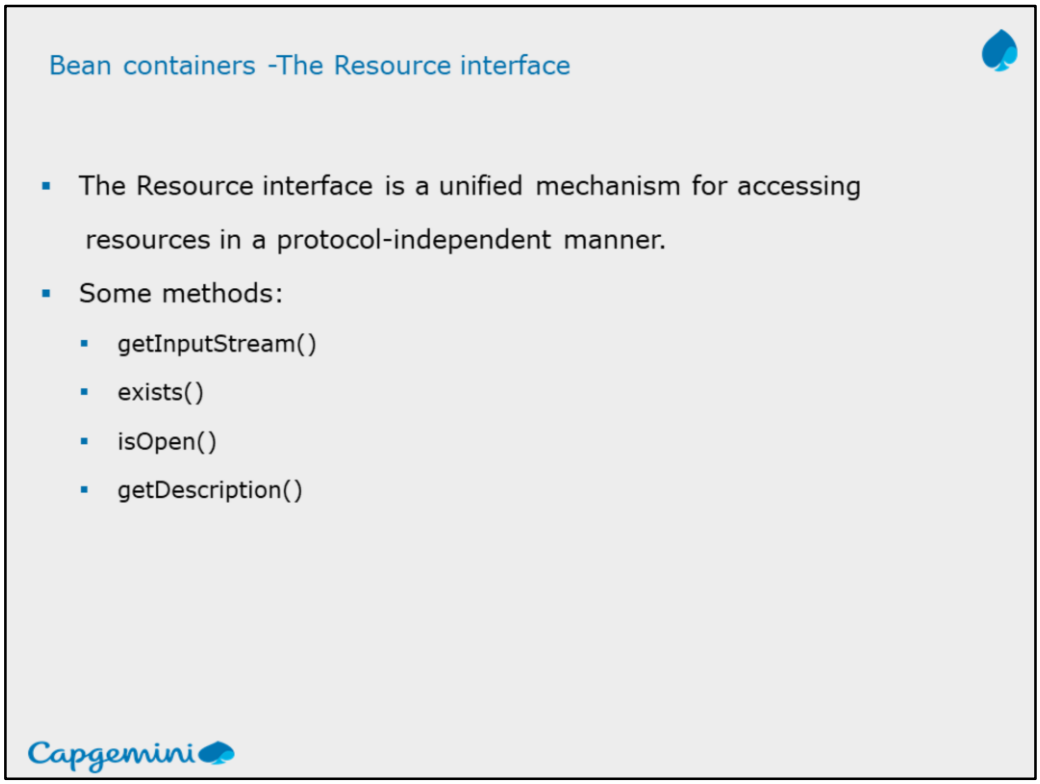
Instructor Notes:



One of the most useful implementations of the bean factory is the org.Springframework.beans.factory.xml. XmlBeanFactory. The BeanFactory is instantiated via explicit user code such as shown above.
This simple line of code tells the bean factory to read the bean definitions from the XML file (beans.xml in this case). But the bean factory doesn't instantiate the beans just yet. Beans are "lazily" instantiated into bean factories, meaning that while the bean factory will immediately load the bean definitions, beans themselves will not be instantiated until they are needed.

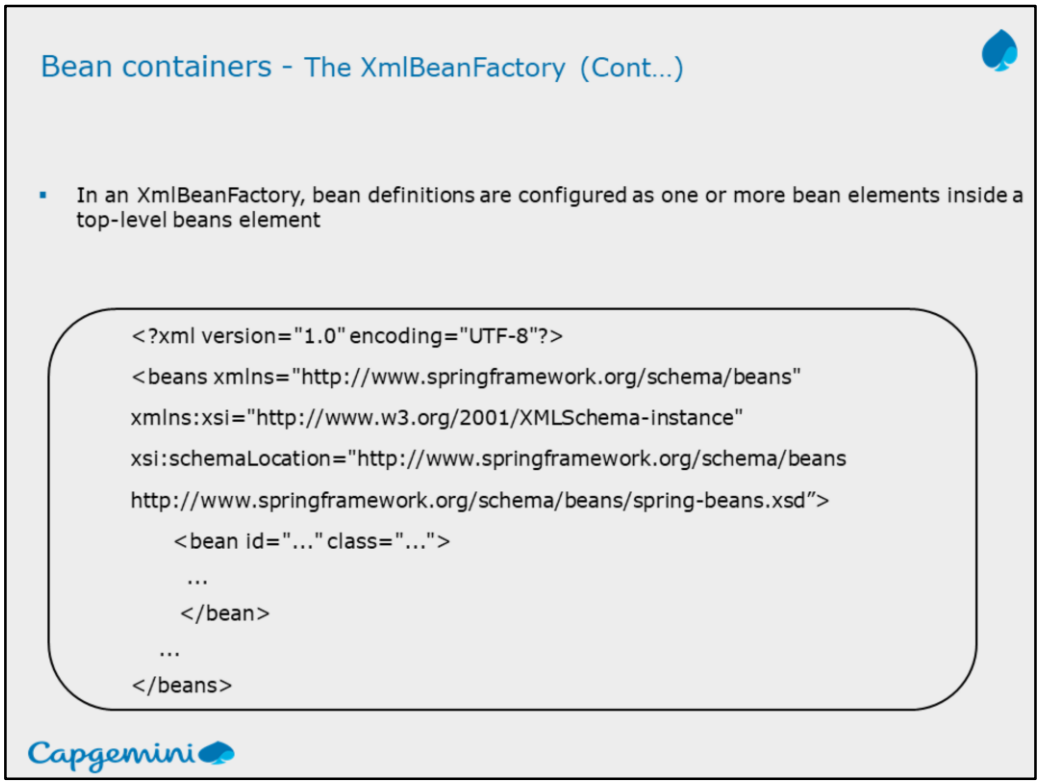
The Spring IoC container consumes some form of configuration metadata; which is nothing more than how you inform the Spring container as to how to instantiate, configure, and assemble the objects in your application. This configuration metadata is typically supplied in a simple and intuitive XML format. When using XML-based configuration metadata, you write bean definitions for those beans that you want the Spring IoC container to manage, and then let the container do its stuff.
Note
XML-based metadata is by far the most commonly used form of configuration metadata. It is not however the only form of configuration metadata that is allowed. The Spring IoC container itself is totally decoupled from the format in which this configuration metadata is actually written. The XML-based configuration metadata format really is simple though, and so the majority of this material will use the XML format to convey key concepts and features of the Spring IoC container.

Instructor Notes:



The Resource interface: Often, an application needs to access a variety of resources in different forms. You may need to access some configuration data stored in a file in the filesystem, some image data stored in a JAR file on the classpath, or maybe some data on a server elsewhere. Spring provides a unified mechanism for accessing resources in a protocol-independent manner. This means that your application can access a file resource in the same way, whether it is stored in the file system, the classpath or on a remote server.
At the core of the Spring's support is the Resource interface. This defines self-explanatory methods mentioned above. There are a number of implementations that come supplied straight out of the box in Spring:
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.
ClassPathResource : This class represents a resource which should be obtained from the classpath. This uses either the thread context class loader, a given class loader, or a given class for loading resources.
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.
Two of the implemented classes examples for this interface ie. ClassPathResource and FileSystemResource, are shown in previous slide. Please refer to the Spring documentation for more details.

Instructor Notes:



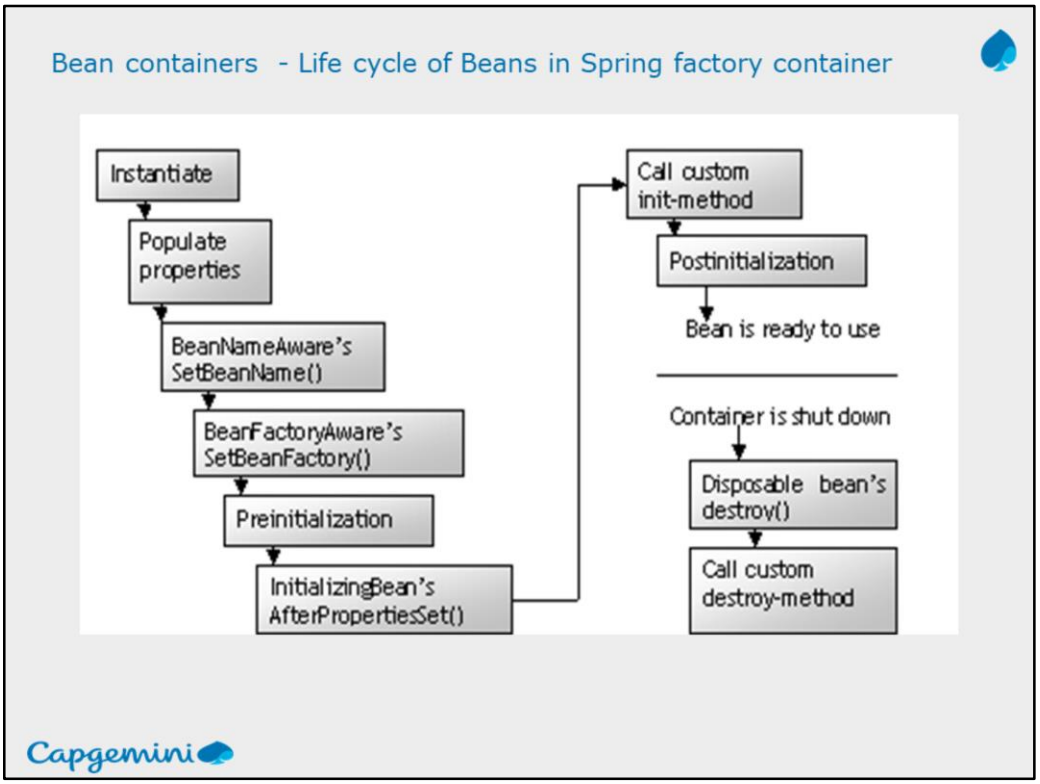
A BeanFactory configuration consists of, at its most basic level, definitions of one or more beans that the BeanFactory must manage. In an XmlBeanFactory, these are configured as one or more bean elements inside a top-level beans element. The first versions of Spring used a DTD. But Spring 2.0 onwards uses schema for the xml configuration file.

To retrieve a bean from a bean factory, simply call the getBean() method, passing it the name of the bean you want to retrieve.

```
MyBean myBean = (MyBean) factory.getBean("myBean");
```

When getBean() is called, the factory will instantiate the bean and begin setting the bean's properties using dependency injection. Thus begins the bean's life cycle within the container (explained further on).

Instructor Notes:



In a traditional Java application, the life cycle of a bean is fairly simple. Java's new keyword is used to instantiate the bean and it is ready to use. In contrast, the life cycle of a bean within a Spring container is a bit more elaborate.
A bean factory performs several setup steps before a bean is ready to use.
The container finds the bean's definition and instantiates the bean.
Using dependency injection, Spring populates all the properties as specified in the bean definition.
If the bean implements the BeanNameAware interface, the factory calls setBeanName() passing the bean's ID.
If the bean implements the BeanFactoryAware interface, the factory calls setBeanFactory() passing an instance of itself.
If there are any BeanPostProcessers associated with the bean, their PostProcessBeforeInitialization() methods will be called.
If an init-method is specified for the bean, it will be called.
Finally, if there are any BeanPostProcessers associated with the bean, their PostProcessAfterInitialization() methods will be called.
The bean is now ready to be used and will remain in the bean factory until it is no longer needed. It is removed from the factory in two ways:

If the bean implements the DisposableBean interface, the destroy() method is called.

If a custom destroy-method is specified, it will be called.

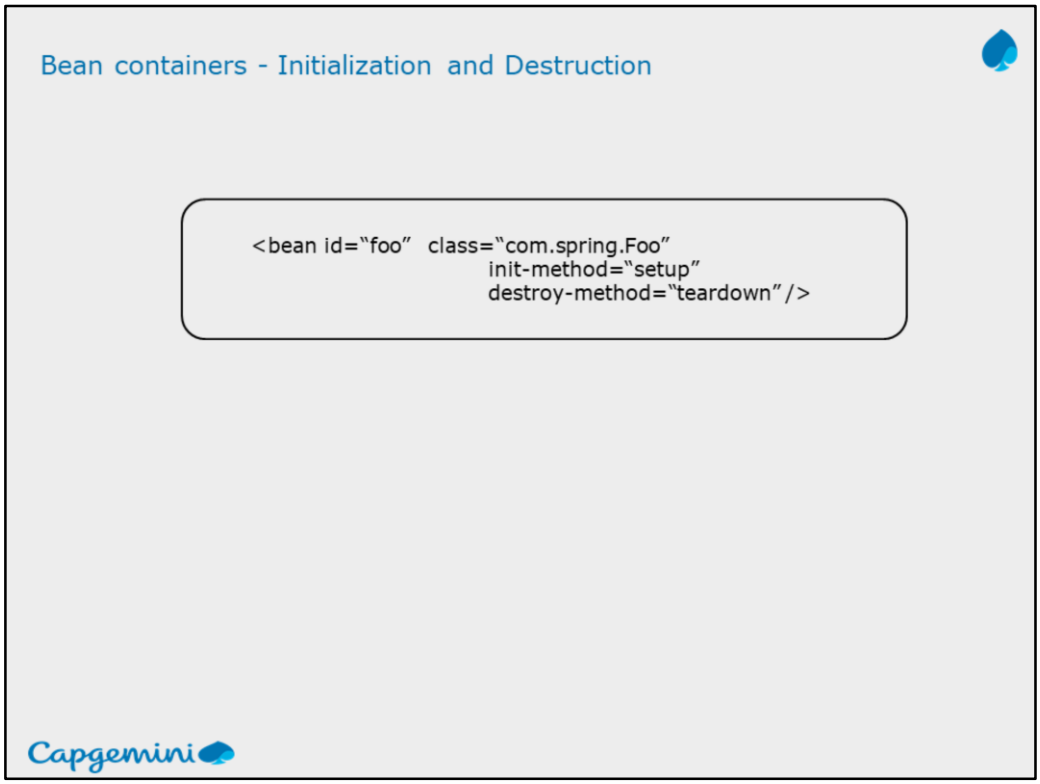
Instructor Notes:

A typical example would be a connection pooling bean:

```
public class ConnPool{
 public void init(){
  //initialize conn pool
}
public void close(){
  //release connection
}
… …
```

The bean definition for this snippet would appear as follows:

```
<bean id="connPool" class="com.ConnPool"
init-method="init" destroy-method="close" />
```

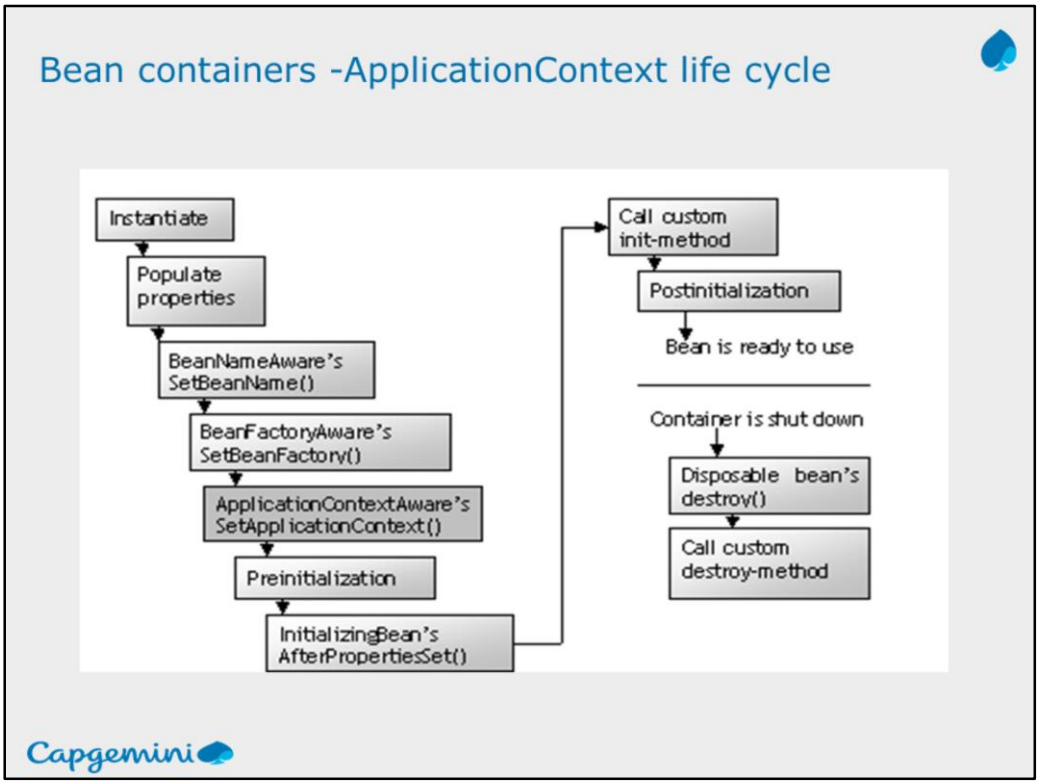


Declaring a custom init-method in your bean's definition specifies a method that is to be called on the bean immediately upon instantiation. Similarly, a custom destroy-method specifies a method that is called just before a bean is removed from the container.
E.g., the init-method="setup" in the above example calls setup() method in the bean class when bean is loaded into container and teardown() when bean is removed from container.

Defaulting init-method and destroy-method:
If many of the beans in a context definition file will have initialization or destroy methods with same name, you don' have to declare init-method or destroy-method on each individual bean. Instead, you can take advantage of the default-init-method and default-destroy-method attributes on the <beans> element.

```
<beans……
default-init-method="tuneApplication"
default-destroy-method="cleanApplication" >
……
</beans>
```

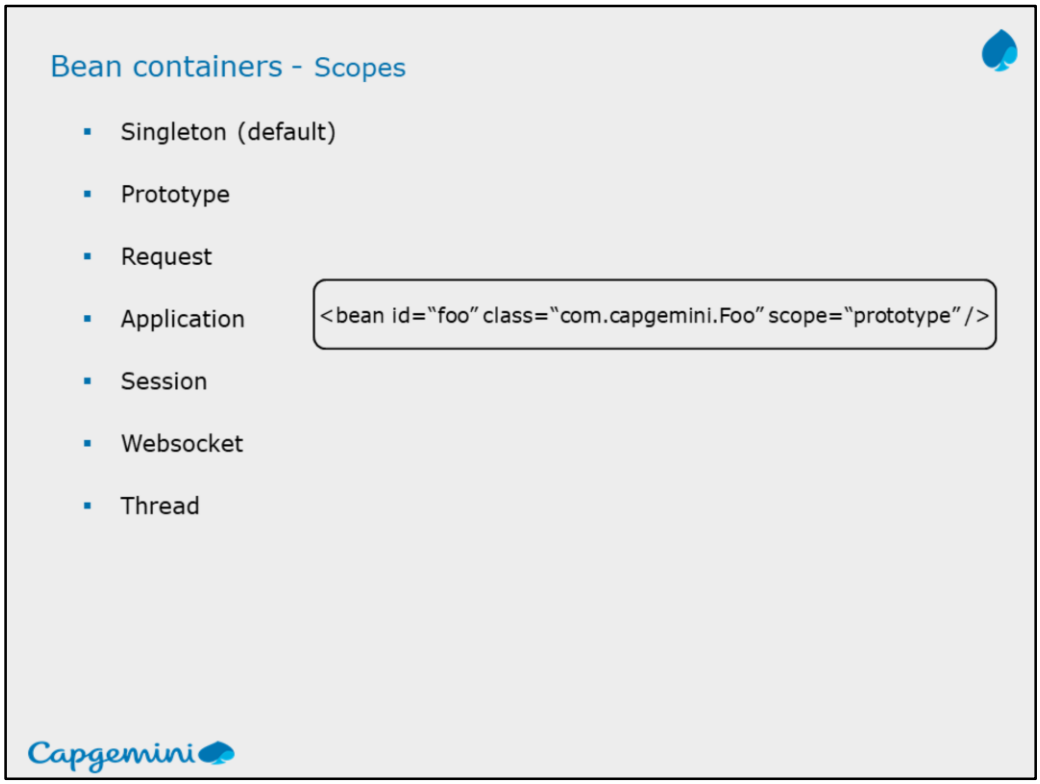
Instructor Notes:



(Continued from previous page)
Eg.
/WEB-INF/*-context.xml
com/mycompany/**/applicationContext.xml
file:C:/some/path/*-context.xml
classpath:com/mycompany/**/applicationContext.xml
classpath:com/mycompany/**/service-context.xml
ApplicationContext ctx = new
ClassPathXmlApplicationContext("classpath*:conf/appContext*.xml");

ApplicationContext life cycle:
The life cycle of a bean within a Spring ApplicationContext differs only slightly from that of a bean within a bean factory as shown in above figure.
The only difference is that if a bean implements the ApplicationContextAware interface, the setApplicationContext() method is invoked.

Instructor Notes:

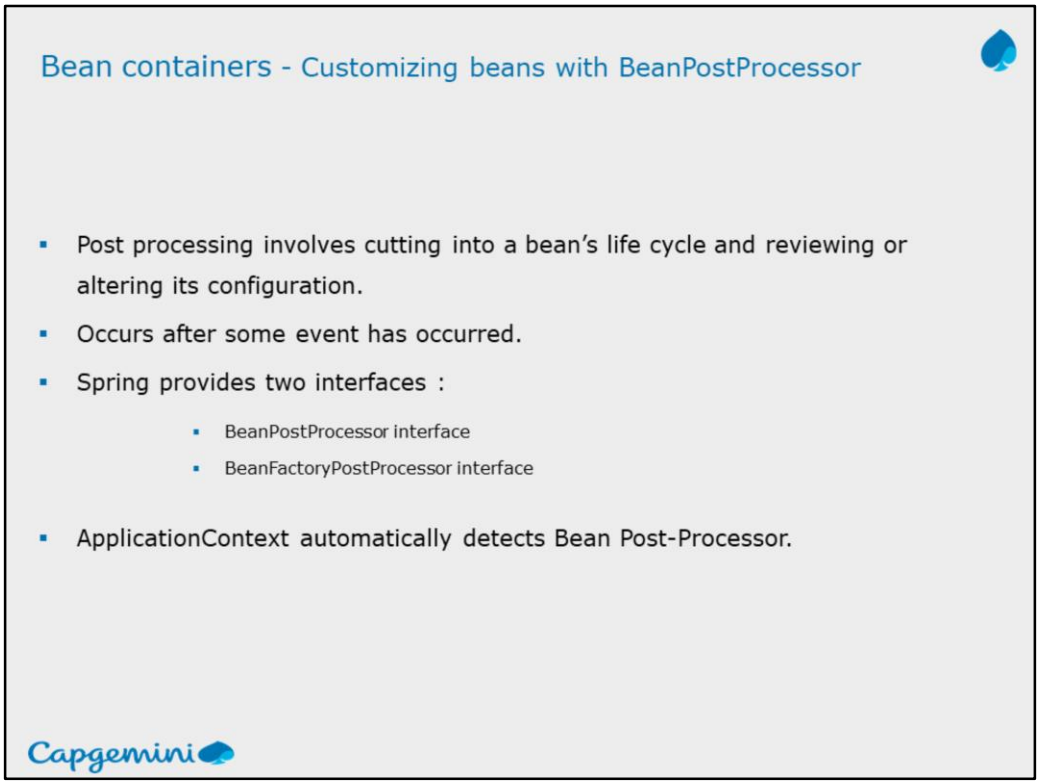


Singleton beans, the default, are created only once by the container and all calls to BeanFactory.getBean() return the same instance. The container will then hold and use the same instance of the bean whenever it is referenced again. This can be significantly less expensive in terms of resource usage than creating a new instance of the bean on each request.
A non-singleton, or prototype bean, may be specified by setting the scope attribute to prototype (see example above). The lifecycle of a prototype bean will often be different than a singleton. When a container is asked to supply a prototype bean, it's initialized and then used, but the container does not hold on to it past that point.
Prototyped beans are useful when you want the container to give a unique instance of a bean each time it is asked for, but you still want to configure one or more properties of the bean through Spring. Thus a new instance is created when getBean() is invoked with the bean's name.

Previous versions of Spring had IoC container level support for exactly two distinct bean scopes (singleton and prototype). Spring 2.0 onwards provides a number of additional scopes depending on the environment in which Spring is being deployed (for example, request and session scoped beans in a web environment).
It also provides integration points so that Spring users can create their own scopes. Beans can be defined to be deployed in one of a number of scopes. See the table in the next page for the different scopes.

Instructor Notes:

If you want to implement some custom logic after the Spring container finishes instantiating, configuring, and otherwise initializing a bean, you can plug in one or more BeanPostProcessor implementations. We shall not focus on this interface further

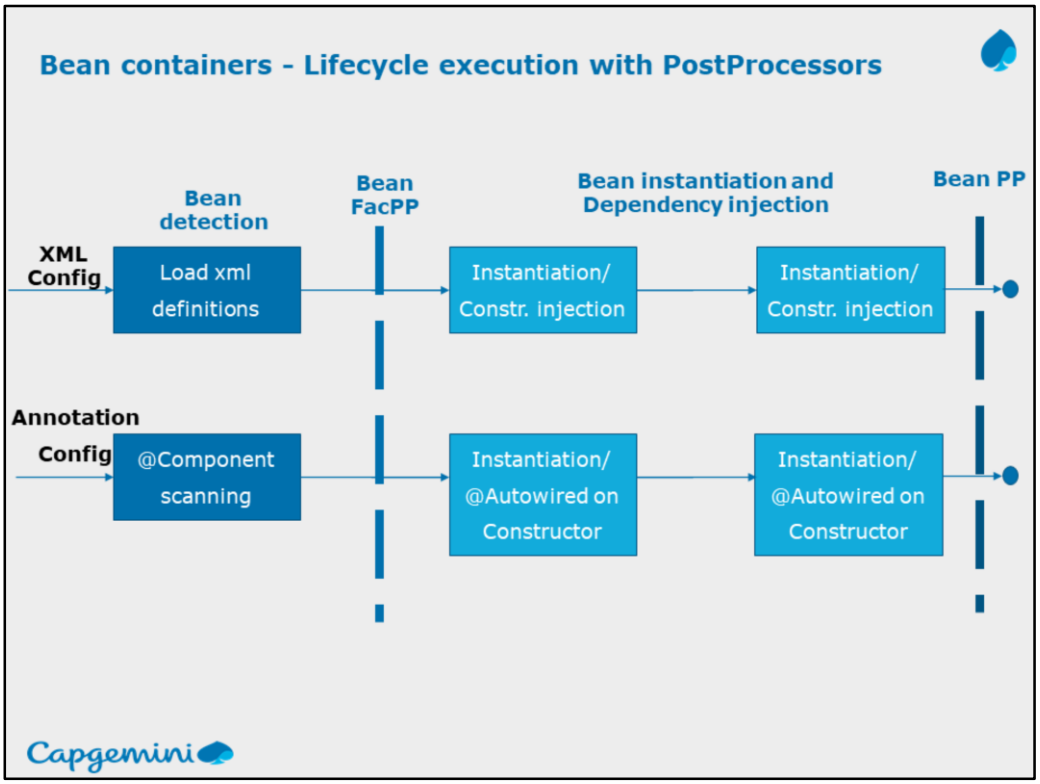


The lifecycle of the BeanFactory and ApplicationContext provide many opportunities to cut into the bean's life cycle to review or alter its configuration. This is called post processing and occurs after some event has occurred. A bean post-processor is a java class which implements the BeanPostProcessor interface, which consists of two callback methods:
postProcessBeforeInitialization: called immediately before bean initialization.
postProcessAfterInitialization: called immediately after bean Initialization.
BeanPostProcessors operate on bean (or object) instances; ie the Spring IoC container instantiates a bean instance and then BeanPostProcessor interfaces do their work.
An ApplicationContext will automatically detect any beans which are deployed into it which implement the BeanPostProcessor interface, and register them as post-processors, to be then called appropriately by the factory on bean creation. Simply deploy the post-processor in a similar fashion to any other bean!
However, for BeanFactory, bean post-processors have to manually be explicitly registered, with a code sequence as shown below.
Since this manual registration step is not convenient, and ApplictionContexts are functionally supersets of BeanFactories, it is generally recommended that ApplicationContext variants are used when bean post-processors are needed.

```
ConfigurableBeanFactory bf = new .....; // create BeanFactory
   // now register some beans and any needed BeanPostProcessors
MyBeanPostProcessor pp = new MyBeanPostProcessor();
bf.addBeanPostProcessor(pp); // now start using the factory ...
```

Instructor Notes:

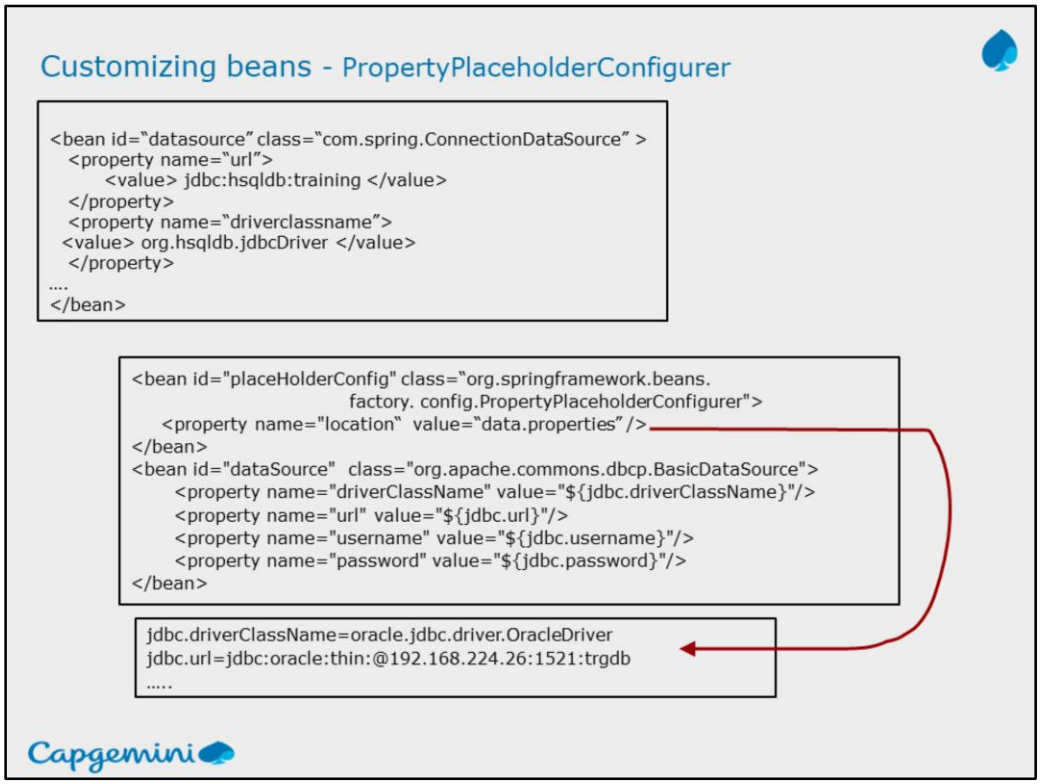
If you want to implement some custom logic after the Spring container finishes instantiating, configuring, and otherwise initializing a bean, you can plug in one or more BeanPostProcessor implementations. We shall not focus on this interface further



Discuss about annotation config later.

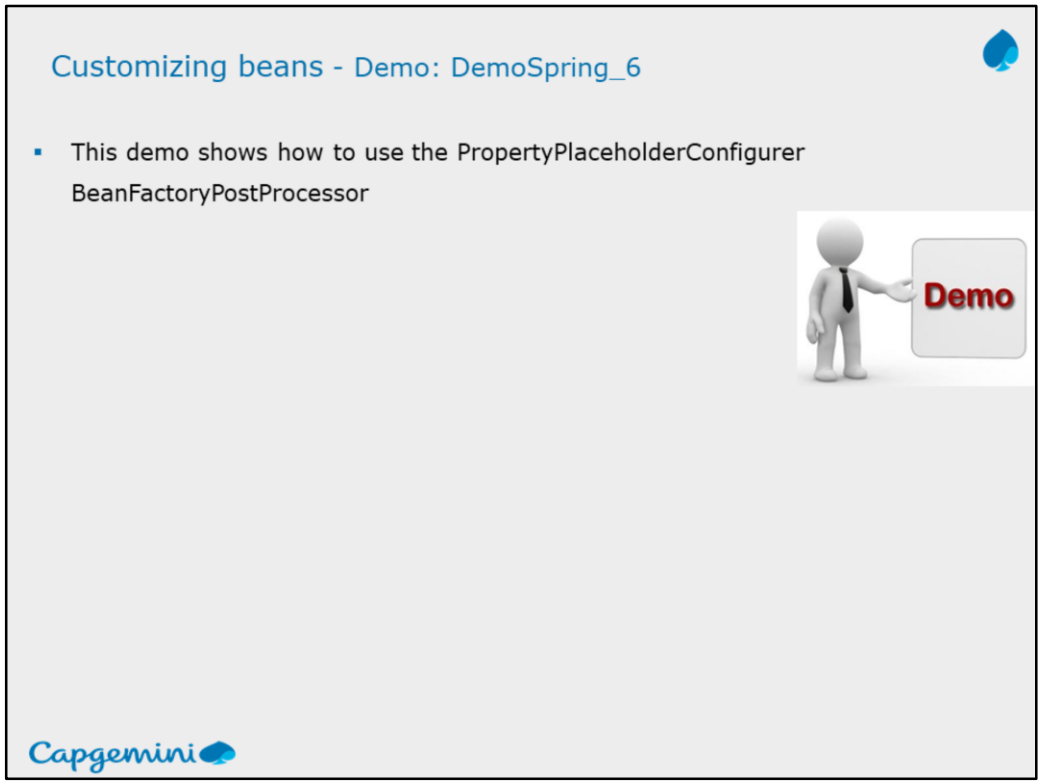
```
ConfigurableBeanFactory bf = new .....; // create BeanFactory
    // now register some beans and any needed BeanPostProcessors
MyBeanPostProcessor pp = new MyBeanPostProcessor();
bf.addBeanPostProcessor(pp); // now start using the factory ...
```

Instructor Notes:



For the most part, it is possible to configure entire application in a single bean wiring file. But sometimes it is beneficial to extract certain pieces of that configuration into a separate property file. E.g. , a configuration concern common to many applications is configuring a data source. Traditionally, in Spring, you do this with the following XML in the bean wiring file (see above)
Configuring the data source directly in the bean wiring file may not be appropriate. The database specifics are a deployment detail and must be separated.

Instructor Notes:



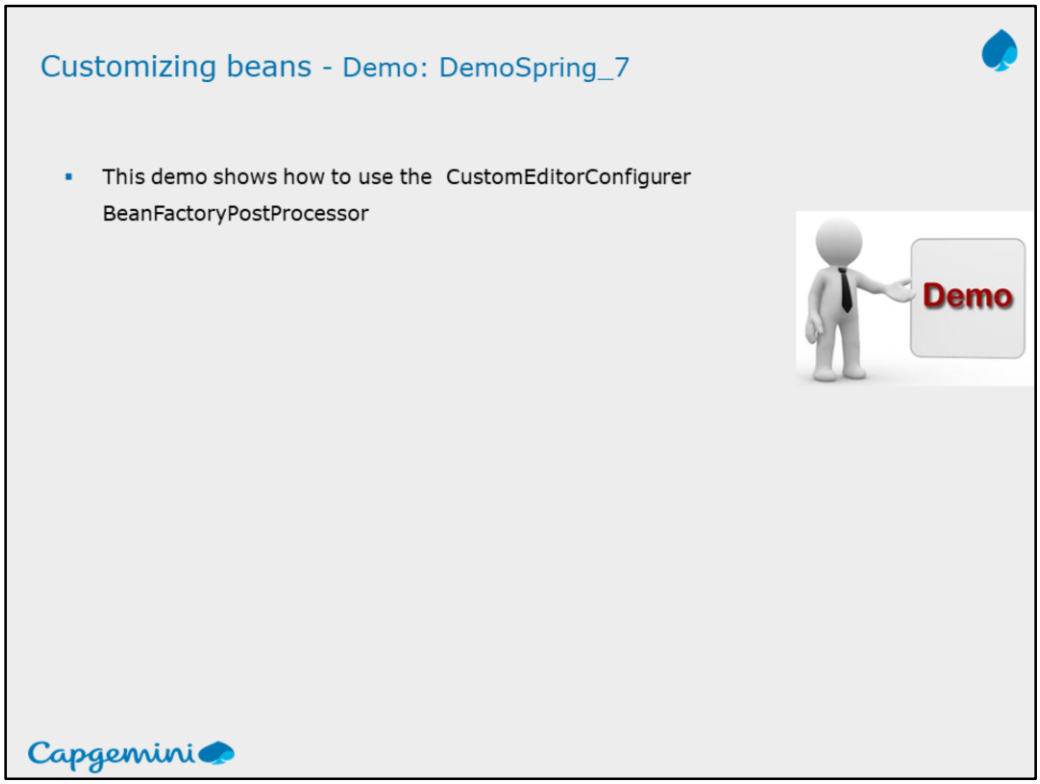
Please refer to demo, DemoSpring_6. In this case user.java is a POJO with two properties – username and password. We shall set the properties of this bean during its instantiation using external properties file. user.properties is a properties file. user.xml has two place holder variables ${username} and ${password}

```
<bean id="user" class="training.spring.User">
<property name="username"><value>${username}</value></property>
<property name="password"><value>${password}</value></property>
</bean>
```

Whenever setter is called, the listener (PropertyPlaceholderConfigurer) is invoked and it will look into the properties file, retrieve values, place them in the place holders and initialize.
If instead of application context, bean factory is used, then the listener would have to be explicitly registered as shown below (also in the commented out code in userClient.java file).

```
XmlBeanFactory factory = new XmlBeanFactory(new
FileSystemResource("user.xml"));
PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfigurer();
cfg.setLocation(new FileSystemResource("user.properties"));
cfg.postProcessBeanFactory(factory);
```

Instructor Notes:



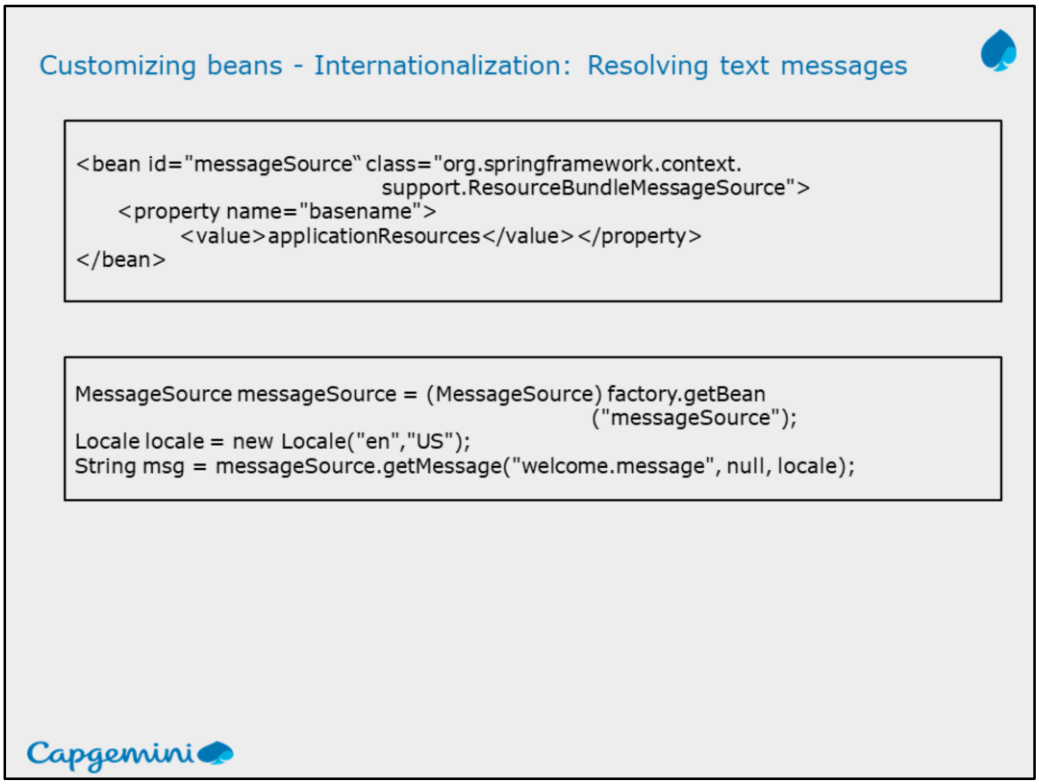
Please refer to DemoSpring_7. The Employee.java is a POJO that holds the date property. Using basic wiring techniques learnt so far, you could set a value into Employee beans' date property. But we have created SQLDateEditor.java extending PropertyEditorSupport class.
Spring must recognize this custom property editor when wiring bean properties using Spring's CustomEditorConfigurer. This BeanFactory PostProcesser internally loads custom editors into the BeanFactory by calling the registerCustomEditor() method. By adding the following bit of XML into the bean configuration file, you will tell Spring to register the SQLDateEditor as a custom editor.

```
<bean id="customEditorConfigurer“ class="org. springframework. beans.
factory.config.CustomEditorConfigurer">
    <property name="customEditors">
        <map> <entry key="java.sql.Date">
            <bean class="training.spring.SQLDateEditor" /></entry>
        </map>
    </property>
</bean>
```

You will now be able to configure the Employee objects date property using a simple string value:

```
<bean id="employee" class="training.spring.Employee">
    <property name="date"><value>2006-01-01</value></property>
</bean>
```

Instructor Notes:



To use ResourceBundleMessageSource, add the above xml entry (the first code snippet) to the bean wiring file.

It is very important that this bean be named messageSource because the ApplicationContext will look for a bean specifically by that name when setting up its internal message source.

The basename property specifies the base name of the bundle. The bundle will normally look for messages in properties files with names that are variations of base name depending on locale.

You will never need to inject the messageSource bean into your application beans but will instead access messages via ApplicationContext's own getMessage() methods. For e.g. to retrieve the message whose name is "welcome.message", please refer to the second code snippet above.

You will likely be using parameterized messages in the context of a web application, displaying the text on a web page. In that case, you can use Spring's <spring:message> jsp tag to retrieve messages and need not need directly access the ApplicationContext.

< spring:message code="welcome.message" />

Alternatively, you can use the second argument of the getMessage() to pass in an array of arguments that will be filled in for params within the message

Instructor Notes:

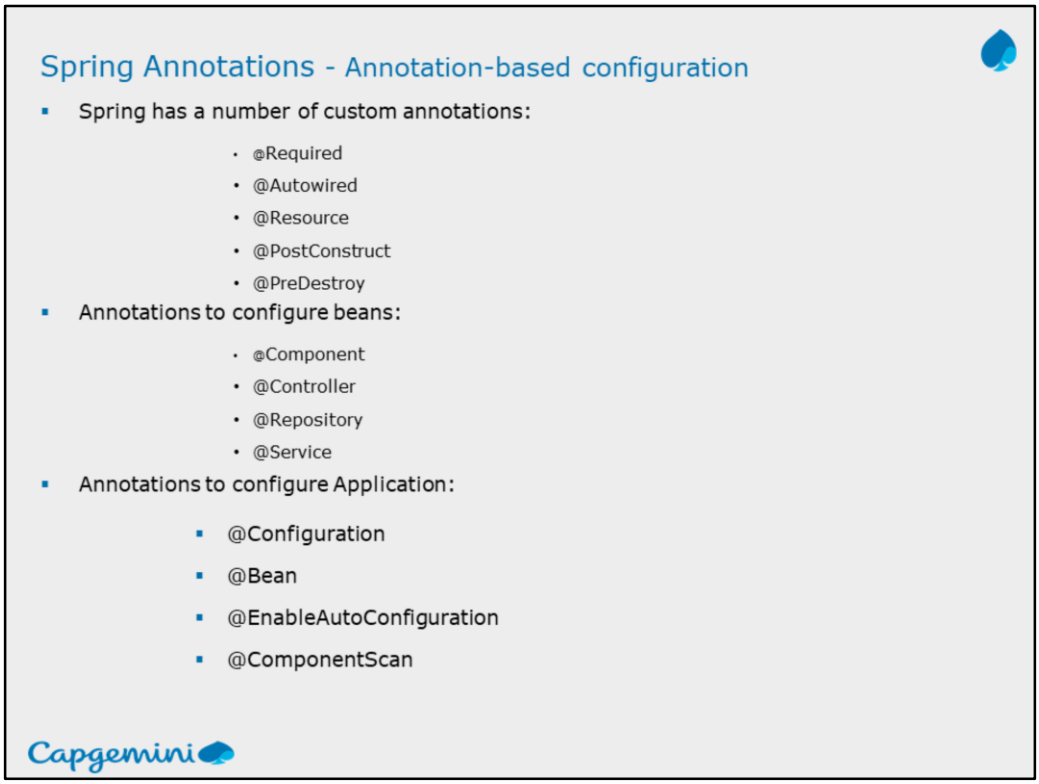
Additional notes for instructor

Please refer to DemoSpringI18N. There are two resource bundles defined :
applicationResources_en_GB.properties
applicationResources_en_US.properties
Message source has been defined in message.xml
MessageClient.java creates a new locale object and uses the getMessage() to access messages from the appropriate resource bundle based on locale. Notice that the message is parameterized :
welcome.message = Welcome {0}, in UK

Hence we need to send value for {0} parameter in this manner:
String msg = messageSource.getMessage("welcome.message", new Object[]{"Majrul"},locale);
This will set the value of the {0} parameter to “Majrul”. We can pass in many parameters for a single message, by using this object array in the getMessage() method.

Instructor Notes:

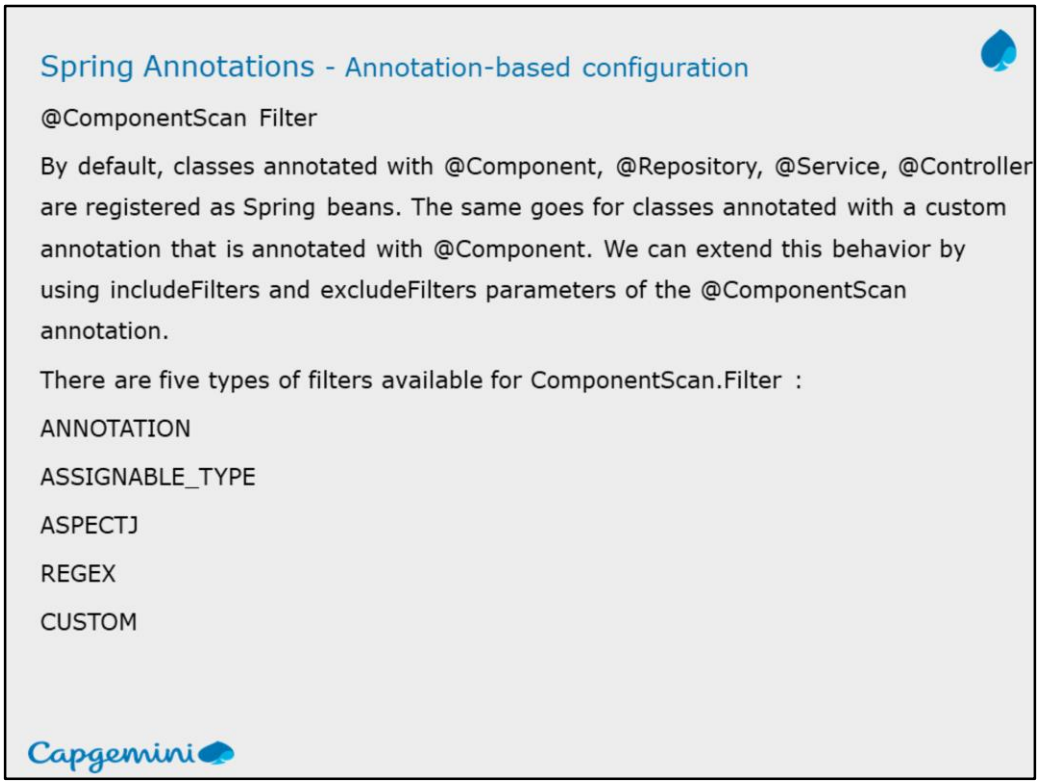
Annotation injection is performed *before* XML injection, thus the latter configuration will override the former for properties wired through both approaches.



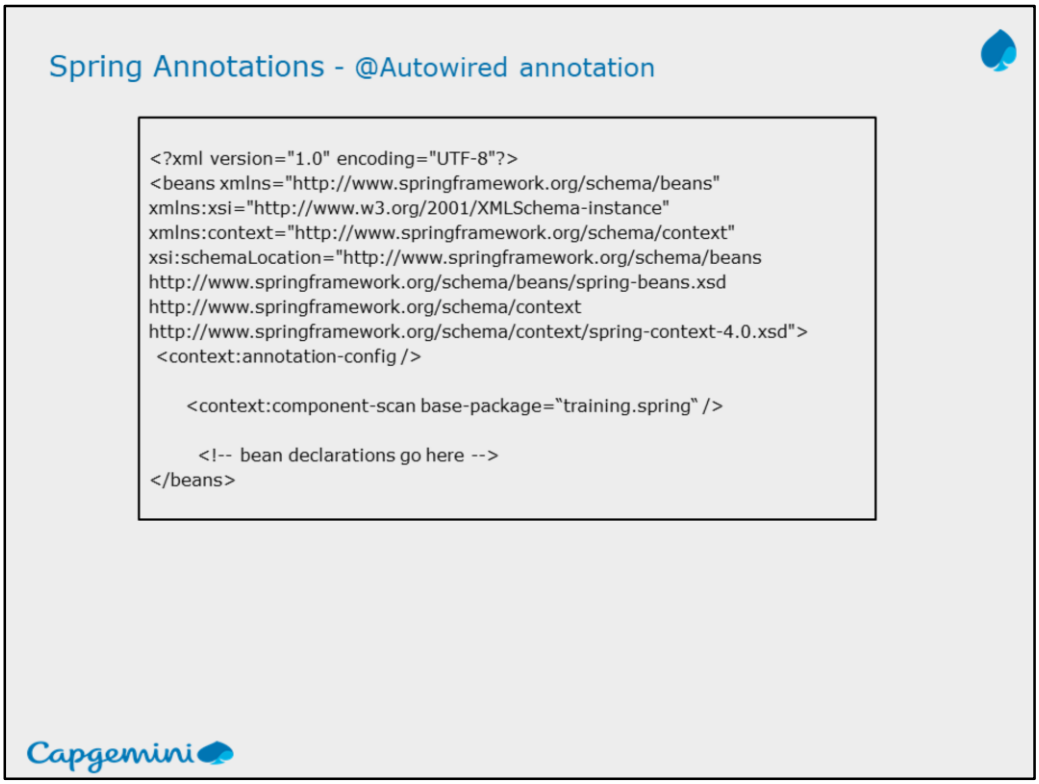
So far, we have configured our beans in the XML configuration files. Starting with Spring 2.0, we can use annotations to configure our beans. By annotating your classes, you state their typical usage or stereotype. Spring has a number of custom (Java5+) annotations.
@Required : The @Required annotation is used to specify that the value of a bean property is required to be dependency injected. That means, an error is caused if a value is not specified for that property
@Autowired: Prior to Spring 2.5, autowiring could be configured for a number of different approaches: constructor, setters by type, setters by name, or autodetect – which offer a large degree of flexibility, but not very fine-grained control. For eg, it has not been possible to autowire a specific subset of an object's setter methods or to autowire some of its properties by type and others by name.
By using @Autowired, you can eliminate the additional XML in your configuration file that specifies the relationship between two objects. Also, you no longer need methods to set the property in the owning class. Just include a private or protected variable and Spring will do the rest.
@Resource : Declares a reference to a resource such as a data source, Java Messaging Service (JMS) destination, or environment entry. This annotation is equivalent to declaring a resource-ref, message-destination-ref, env-ref, or resource-env-ref element in the deployment descriptor.

Instructor Notes:

Annotation injection is performed *before* XML injection, thus the latter configuration will override the former for properties wired through both approaches.

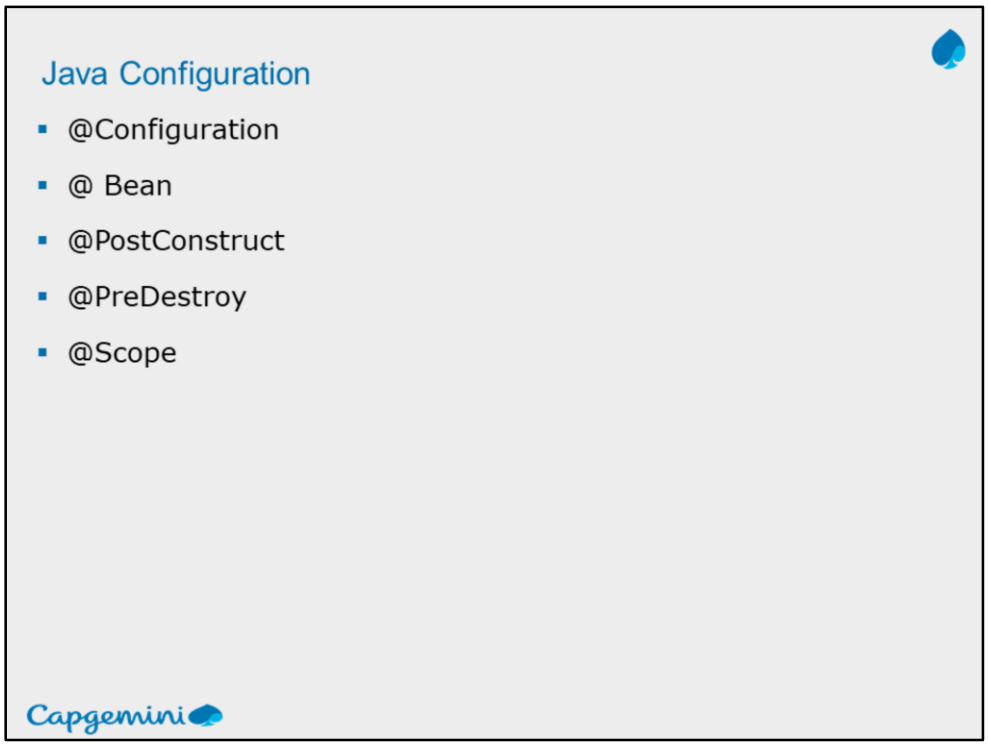


Instructor Notes:

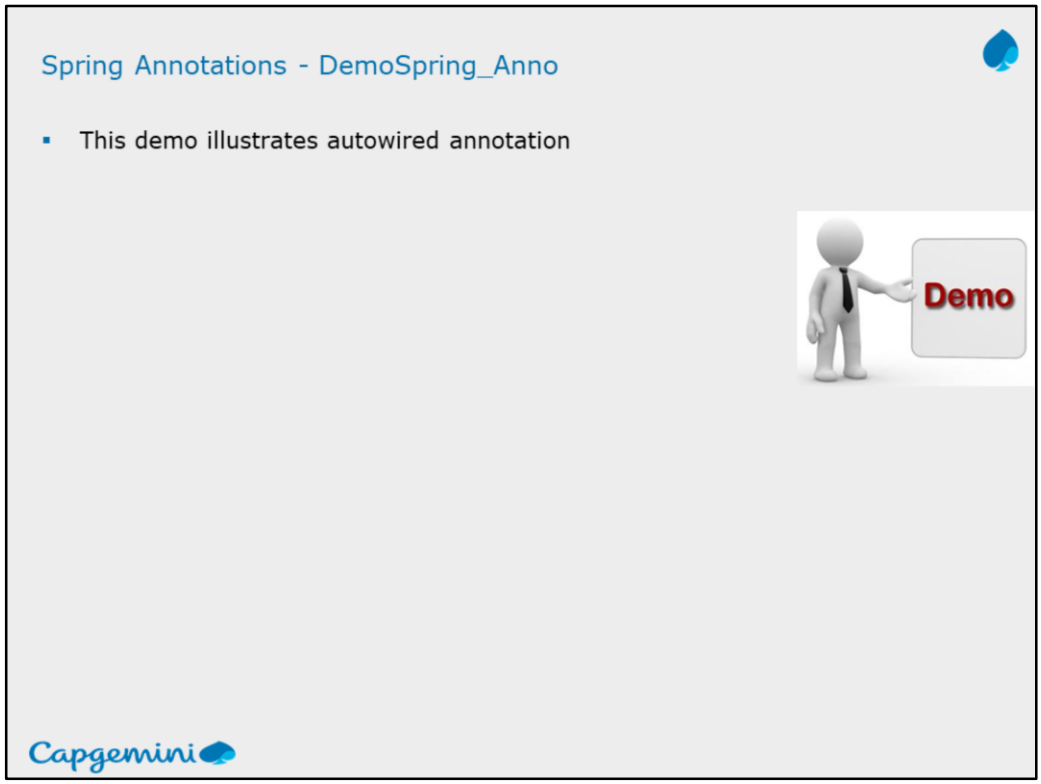


We can use annotations to automatically wire bean properties. It is similar to autowire attribute in configuration file, but allows more fine-grained autowiring, where you can selectively annotate certain properties for autowiring.
However, simply annotating your classes is not enough to get an annotation's behavior. You need to enable a component that is aware of the annotation and that can process it appropriately. This component (a special BeanPostProcessor implementation) will be different for all annotations. For eg, the RequiredAnnotationBeanPostProcessor class is necessary for @Required annotation.
Annotation wiring isn't turned on in the Spring container by default. So, before we can use annotation-based autowiring, we'll need to enable it in our Spring configuration. The simplest way to do that is with the <context:annotation-config> element from Spring's context configuration namespace.
<context:annotation-config> tells Spring that you intend to use annotation-based wiring in Spring. Once it's in place you can start annotating your code to indicate that Spring should automatically wire values into properties, methods, and constructors. The implicitly registered post-processors include AutowiredAnnotationBeanPostProcessor, CommonAnnotationBeanPostProcessor, PersistenceAnnotationBeanPostProcessor, as well as the RequiredAnnotationBeanPostProcessor.

Instructor Notes:



Instructor Notes:



Please refer to demo, DemoSpring_Anno, This example uses the anno.xml for configuring the beans.

Notice how the CurrencyConverterImpl.java depends on the com.capgemini.anno.ExchangeServiceImpl.java class for services. Notice how the exchangeService property has been annotated with @Autowired. The configuration file now just contains beans declaration and no instructions on wiring!

Instructor Notes:

Crucial Namespaces 'p' and 'c'

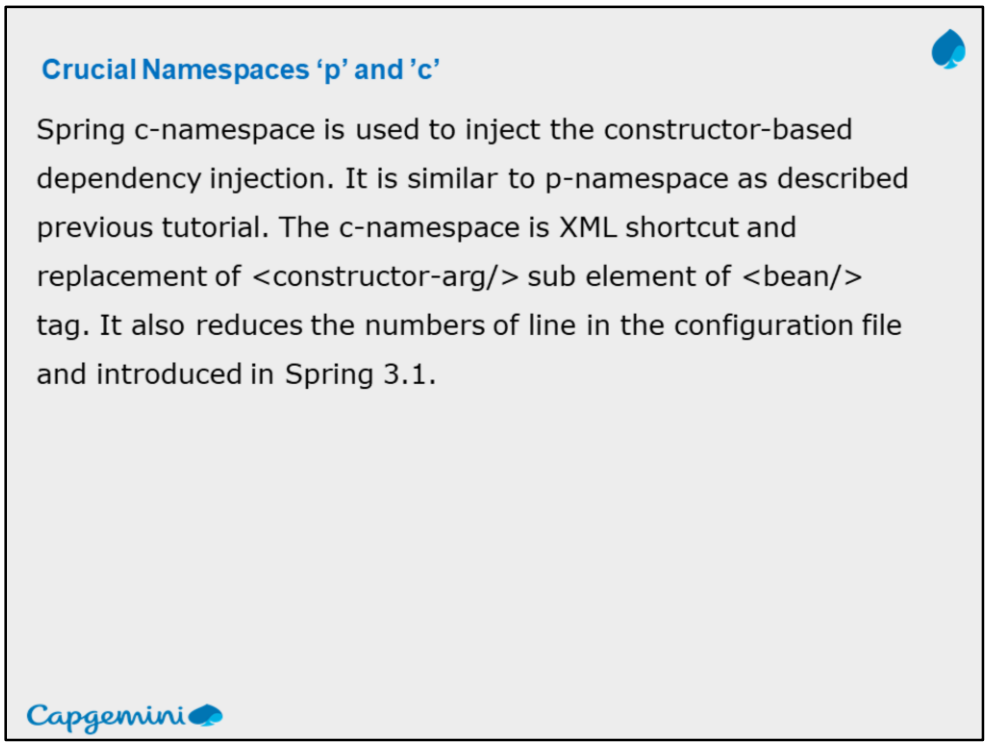
In the Spring framework, p-namespace is used to inject setter-based dependency. The p-namespace is XML shortcut and reduce the numbers of line in the configuration file. However, the p-namespace is not defined in an XSD file and exists only in the core of Spring.

```
<!–traditional style -->
<bean id="oldTeacher" class="beans.Teacher">
<property name="name" value="Vijay Pnadey" />
<property name="qualification" value="PhD" />
</bean>

<!–new style -->
<bean id="newTeacher" class="beans.Teacher"
p:name="Vijay Pandey" p:qualification="PhD"/>
```

Capgemini

Instructor Notes:



Instructor Notes:

Crucial Namespace 'p' and 'c'

```
    <!-- old style -->
<bean id="oldboss" class="beans.Boss">
    <constructor-arg name="name" value="Atul Rai" />
    <constructor-arg name="employee" ref="oldemp" />
</bean>
<bean id="oldemp" class="beans.Employee">
    <constructor-arg name="empName" value="Mukul" />
    <constructor-arg name="empSalary" value="1000000" />
</bean>

<!-- new style -->
<bean id="newboss" class="beans.Boss" c:name="Atul Rai"
 c:employee-ref="newemp" />
<bean id="newemp" class="beans.Employee" c:empName="Mukul"
c:empSalary="1000000" />
```

Capgemini

Instructor Notes:

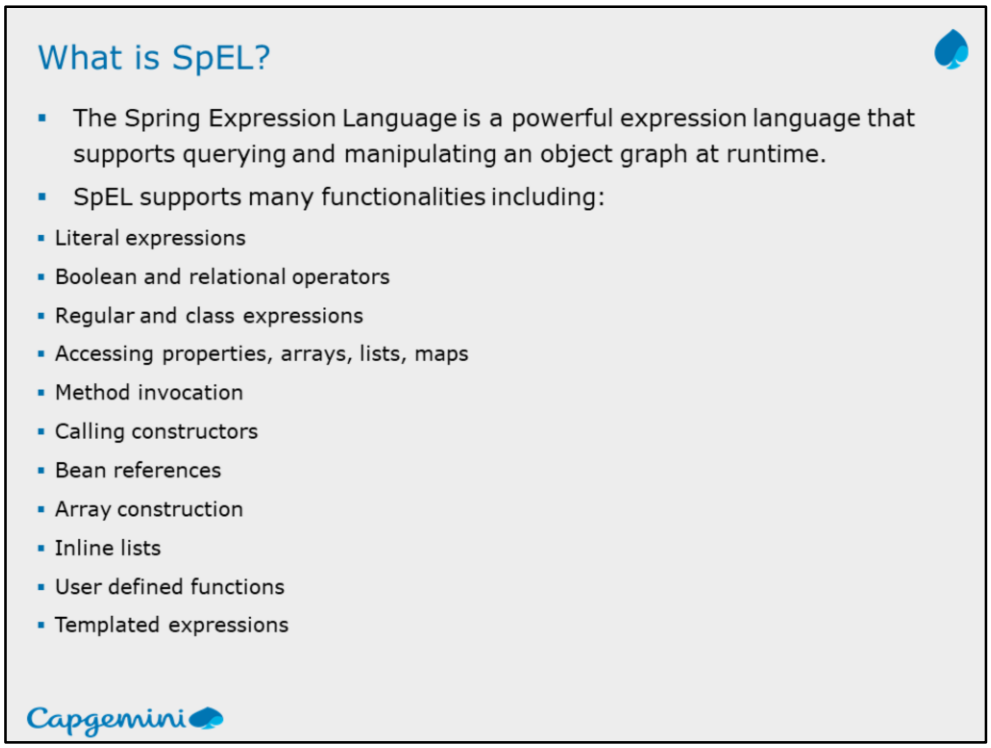
Demo on Namespace 'c' and 'p'

- This demo illustrates on how to use 'p' and 'c' Namespace in spring.

Capgemini

Instructor Notes:

While there are several other Java expression languages available, OGNL, MVEL, and JBoss EL, to name a few, the Spring Expression Language was created to provide the Spring community with a single well supported expression language that can be used across all the products in the Spring portfolio.

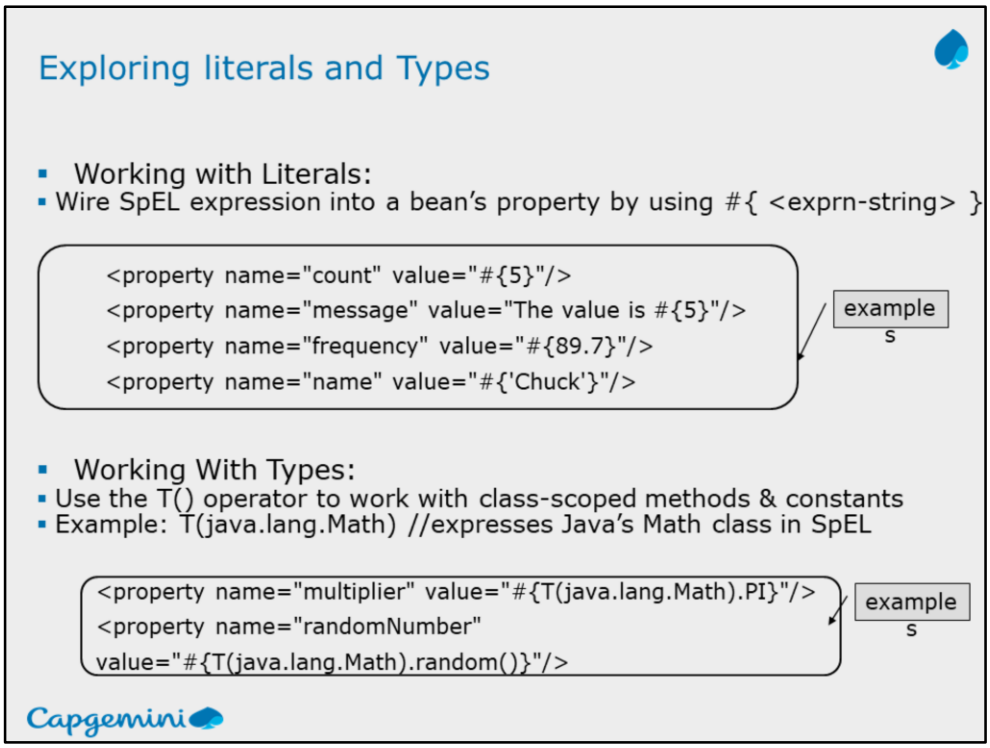


So far we have seen how to wire dependencies using setter and constructor injections. But these have been statically defined in the Spring configuration file. When we wired the exchangeService property into the CurrencyConverter bean, that value was determined at development time. Likewise, when we wired references to other beans, those references were also statically determined in the Spring configuration.

What if we want to wire properties with values at runtime? Spring 3's *Spring Expression Language (SpEL)* is a powerful way of wiring values into a bean's properties or constructor arguments using expressions that are evaluated at runtime. SpEL syntax is similar to Unified EL but offers additional features like method invocation and basic string templating functionality.

SpEL is based on a technology agnostic API allowing other expression language implementations to be integrated should the need arise. It thus can be used independently. It supports many functionalities as listed above.

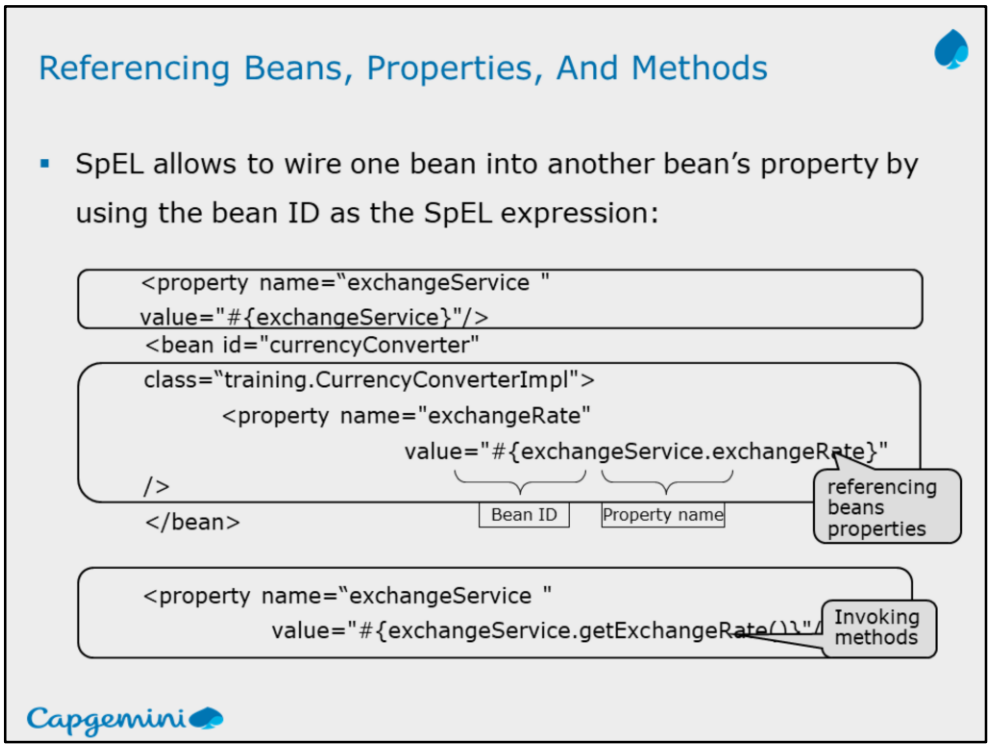
Instructor Notes:



Literal Values: SpEL expressions, like any other expression, are evaluated for some value. SpEL can evaluate literal values, references to a bean's properties, a constant on some class etc. The simplest SpEL expression for example is 5, which evaluates to an integer value of 5. We can wire this value into a bean's property by using #{ } markers in a <property> element's value attribute, as shown in example above. The #{ } markers indicate that the content that they contain is a SpEL expression.
Similarly, see example above for expressing Floating-point numbers. Literal String values can be expressed in SpEL with either single or double quote marks. You can also use Boolean true and false values. Eg- <property name="enabled" value="#{false}"/>

Working With Types: The result of the T() operator is a Class object that represents the given class. The T() operator thus gives us access to static methods and constants on a given class. Eg, to wire the value of **pi** into a bean property, use:
<property name="multiplier" value="#{T(java.lang.Math).PI}"/>
Likewise, static methods can also be invoked on the result of the T() operator. Eg, to wire a random number (between 0 and 1) into a bean property, use:
<property name="randomNumber" value="#{T(java.lang.Math).random()}"/>
When the application is starting up and Spring is wiring the randomNumber property, it'll use the Math.random() method to determine a value for that property!

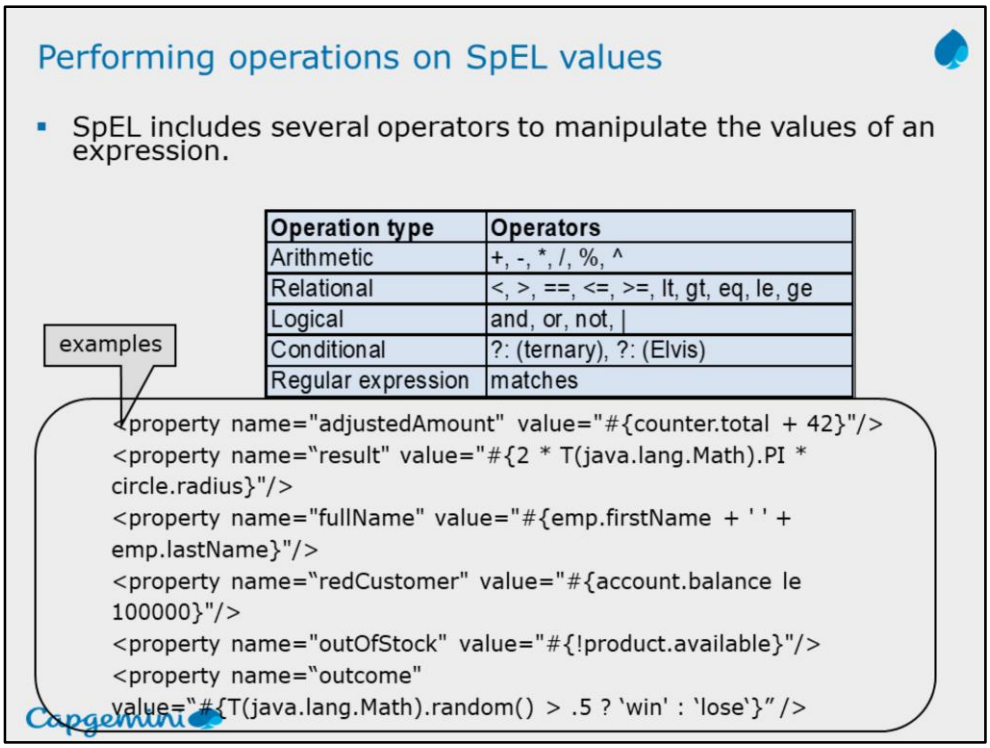
Instructor Notes:



A SpEL expression can reference another bean by its ID. See first example above. We used SpEL to wire the bean whose ID is “exchangeService” into an exchangeService property. We can do this by using the ref attribute too! <property name="exchangeService" ref="exchangeService"/>The outcome is the same. But let us see how to take advantage of being able to wire bean references with SpEL.
The second example configures a new CurrencyConverterImpl bean whose ID is currencyConverter. This is wired to whatever exchangeRate the exchangeService bean provides. This is equivalent to:
CurrencyConverterImpl currencyConverter = new CurrencyConverterImpl ();
currencyConverter.setExchangeRate(exchangeService.getExchangeRate());

Instructor Notes:

Arithmetic expressions: SpEL offers a textual as well as symbolic operators to work with expressions. Eg eq is equivalent to == operator This is done in the interest of consistency with the other operators, and because some developers may prefer the textual operators over the symbolic ones.

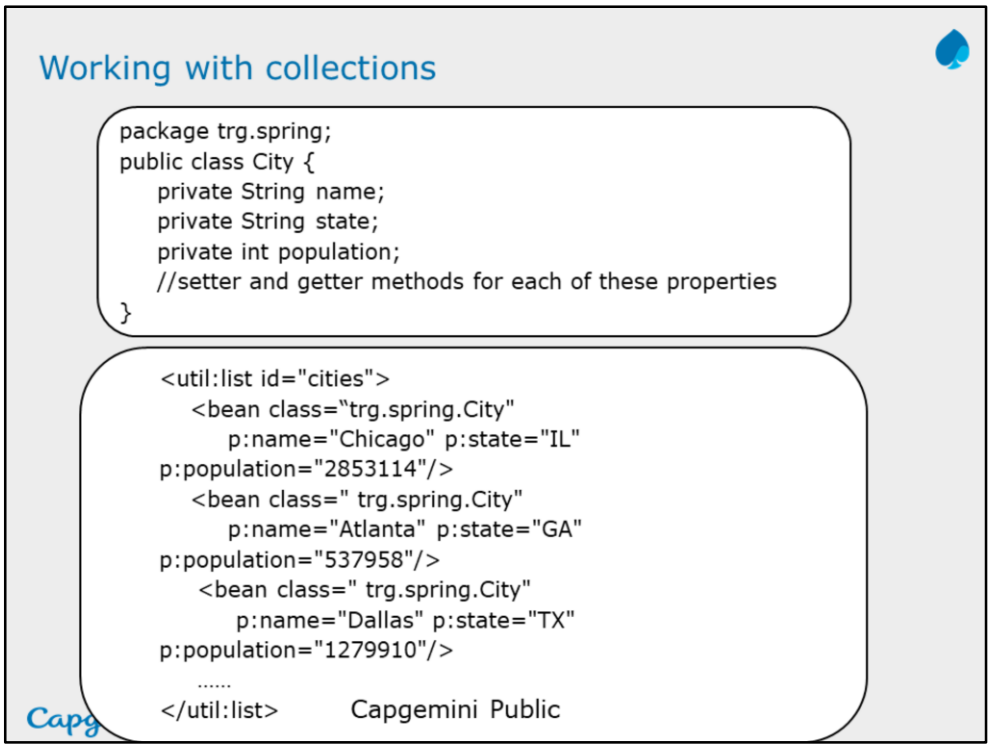


Like in Java, + operator is overloaded to perform concatenation on String values.
We know that the less-than (<) and greater-than (>) operators are used to compare different values. Unfortunately, they pose a problem when using these expressions in Spring's XML configuration (since they have special meaning in XML). So, when using SpEL in XML,5 it's best to use SpEL's textual alternatives like le (<), gt (>) etc.

SpEL supports regular expressions using the *matches* operator, which is a relational operator returning true or false. Example:

```
<util:map id="regExpsSamples" value-type="java.lang.Object"
                                    key-type="java.lang.String">
  <entry key="# {'a string' matches 'a.*'}"
         value="#{'a string' matches 'b.*'}"/>
</util:map>
```

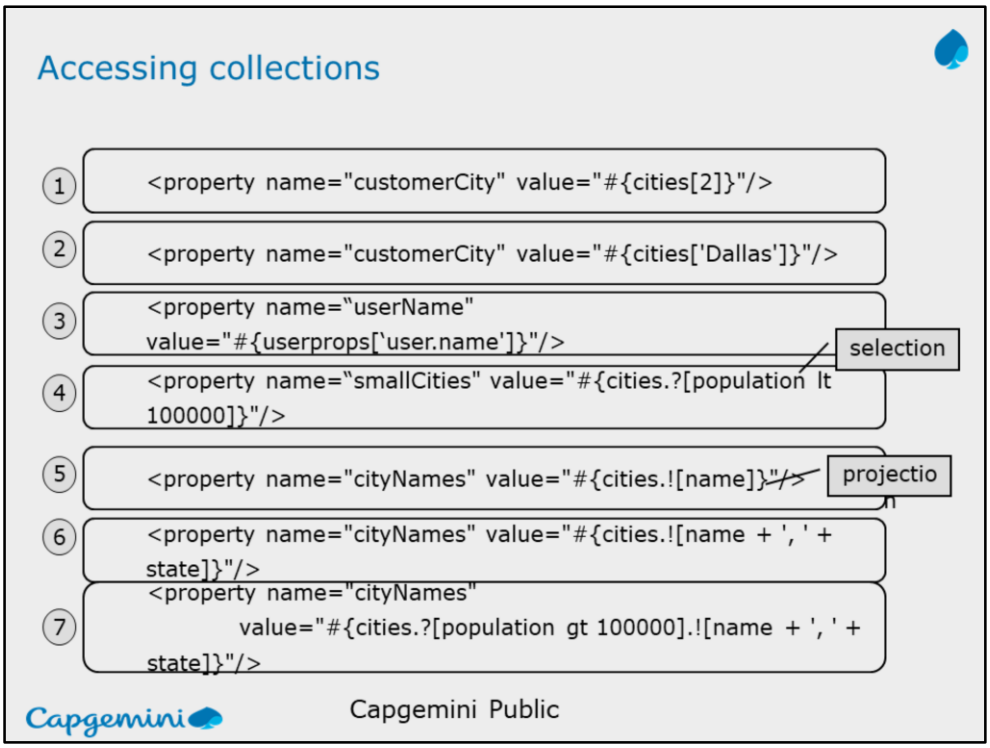
Instructor Notes:



Let us digress a bit and see how Spring allows us to create collection via configuration file.
From Spring 2.5 onwards, in addition to standard <property> tag, you can use the p namespace to set properties of beans.
The <util:list> element comes from Spring's util namespace. It creates a bean of type java.util.List that contains all of the values or beans that it contains. In this case, that's a list of City beans.
Example for creating Map:

```
<util:map id="numbersMap" value-type="java.lang.Integer"
                                key-type="java.lang.String">
  <entry key="one" value="1"/>
  <entry key="two" value="2"/>
  <entry key="three" value="3"/>
  <entry key="four" value="4"/>
  <entry key="five" value="5"/>
</util:map>
```

Instructor Notes:

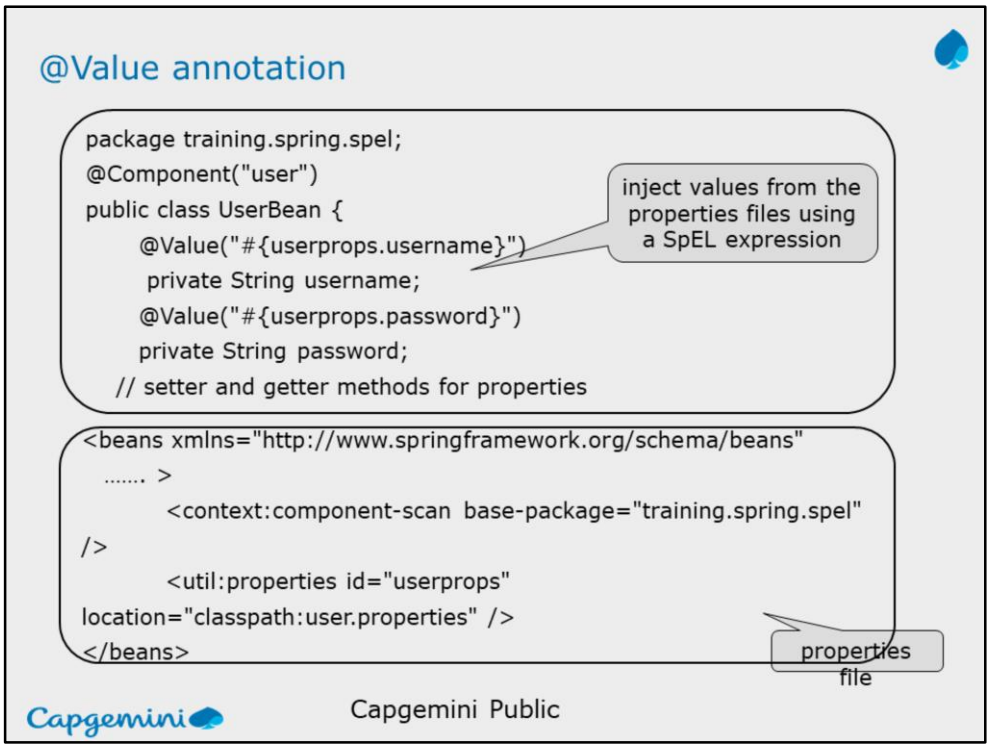


Example -1 : This selects the third city from the list to assign to customerCity
Example -2: Assuming that cities is a java.util.Map collection, with city-name as the key. The example shows how to retrieve the entry for Dallas.
Example -3: We can use the [] operator is to retrieve a value from a java.util.Properties collection too. First: load a properties configuration file into Spring using the <util:properties> element as follows:

<util:properties id="userprops" location="classpath:user.properties"/>

The userprops bean is a java.util.Properties that contains all of the entries in the file named user.properties. Accessing a property from that file is similar to accessing a member of a Map. The 3rd example above reads a property whose name is user.name from the userprops bean
Example -4: We would like a list of cities whose population is less than 100,000. Use the selection operator (.?[]) when doing the wiring. The selection operator creates a new collection whose members include only those members from the original collection that meet the criteria expressed between the square braces. In this case, the smallCities property will be wired with a list of City objects whose population property is less than 100,000.
Example -5: Projecting collections means collecting a particular property from each of the members of a collection into a new collection. Use SpEL's projection operator (.![]) to do this. Example-5 retrieves a list of city names from the collection of City objects. You can also retrieve multiple members as the next example shows.
The final example (7) is a combination of selection and projection.

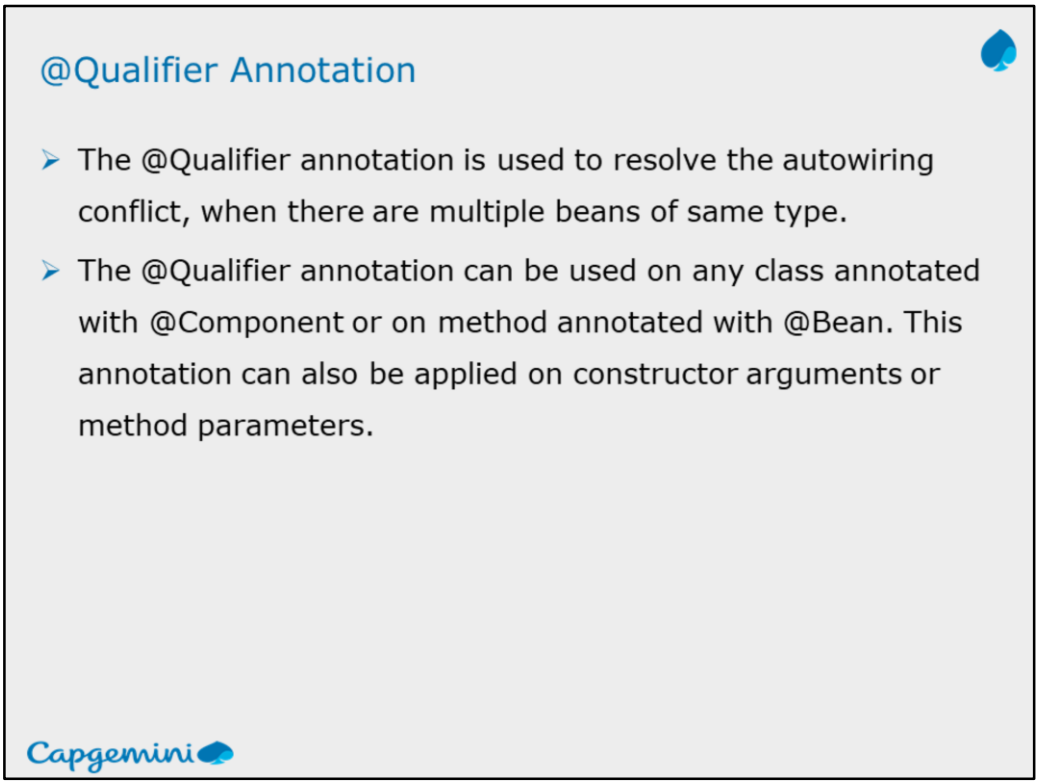
Instructor Notes:



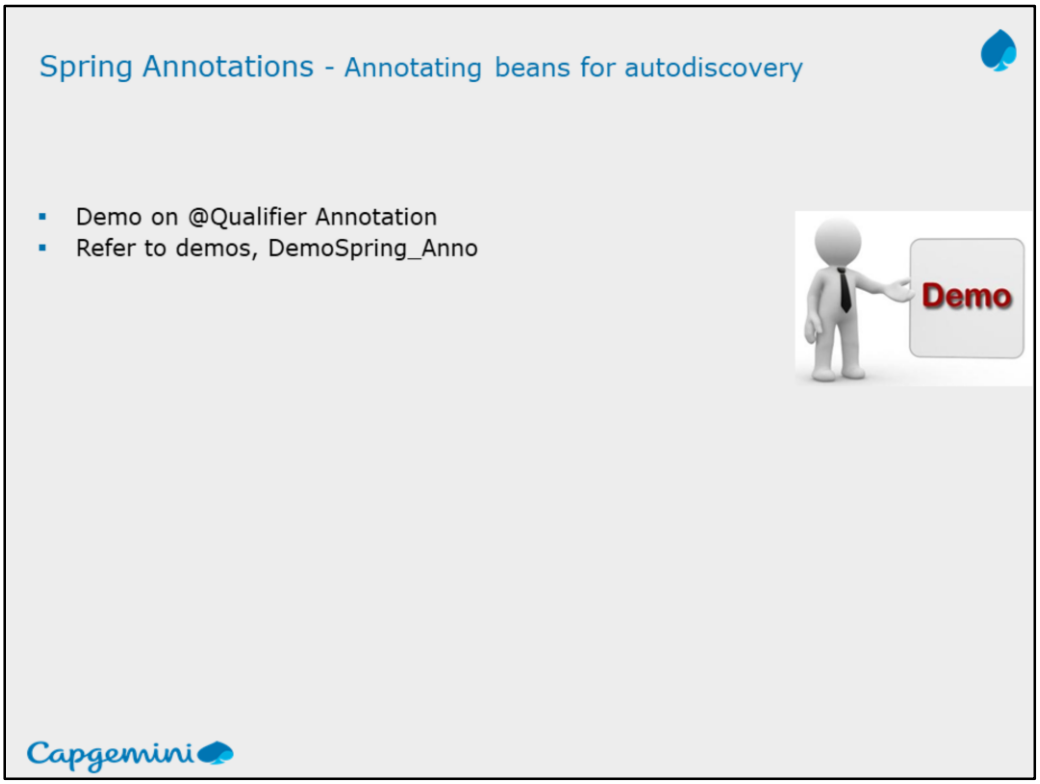
SpEL combined with @Value annotation is great. We have already seen how component scan and autowiring reduces the size of XML configuration files. However, we still have to deal with beans that need literal values as input. With @Value you can inject values from your properties files using SpEL. Assume that you have a properties file configured as shown above.

To use SpEL in annotation, you must register your component via annotation. If you register your bean in XML and define @Value in Java class, the @Value will fail to execute.

Instructor Notes:



Instructor Notes:



Refer to demo, DemoSpring_Anno , Notice the anno.xml configuration file. It contains no beans! Notice the bean classes themselves are annotated with @Component, @PostConstruct, @Autowired etc.

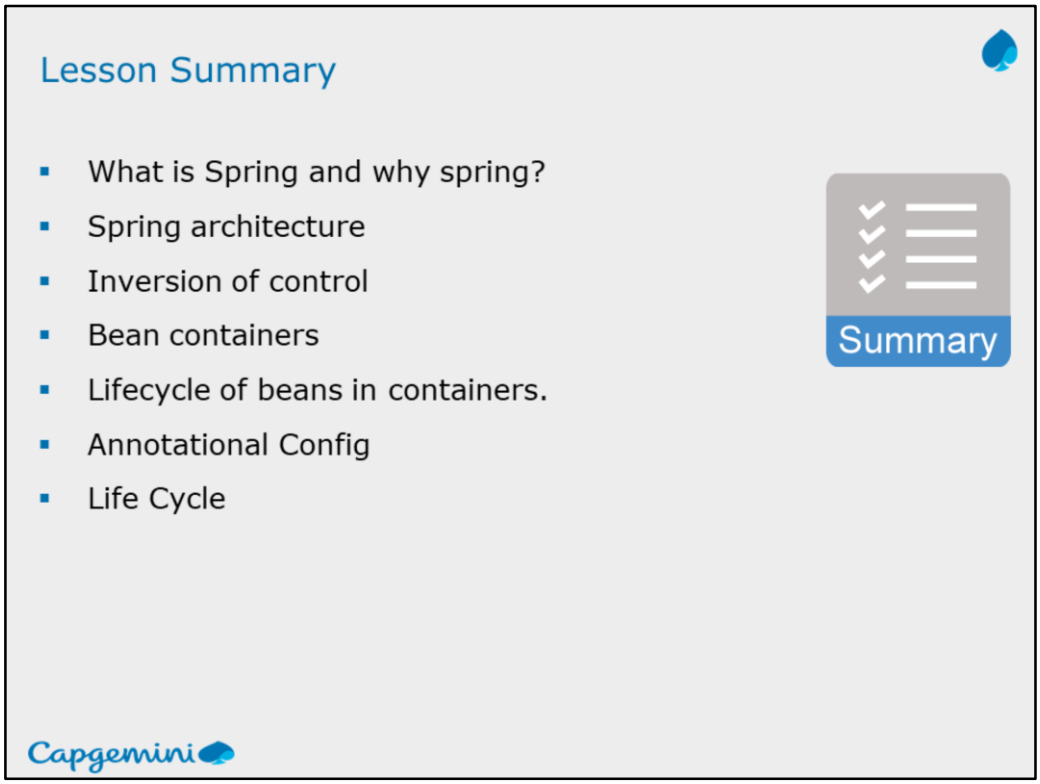
Instructor Notes:

Lab

- From the lab guide
- Lab-1 problem-statement-1 2 and 3

Capgemini

Instructor Notes:



Thus we have seen that Spring is the most popular and comprehensive of the lightweight J2EE frameworks that have gained popularity since 2003.
We saw how Spring is designed to promote architectural good practice. A typical spring architecture will be based on programming to interfaces rather than classes.
We have seen what is Inversion of control and dependency injection.
We also saw Bean containers and lifecycle of beans in containers. We saw how to hook into the lifecycle of a bean and make it aware of the Spring environment.

Instructor Notes:

Ans-1 : a
Ans-2 : b

Review Questions

- Question 1: The <constructor-arg> element has an optional _____attribute that specifies the ordering of the constructor arguments.
  - Option 1: By index
  - Option 2: By type
  - Option 3: By order
- Question 2: A _____ bean lets the container return a new instance each time a bean is asked for in a non-web application
  - Option 1: Singleton
  - Option 2: Prototype
  - Option 3: Request
  - Option 4: session

Capgemini

Instructor Notes:

Ans-3 : idref
Ans-4 : False, it is the BeanFactoryPost Processor that does this.

Review Questions

- Question 3: Specifying the _____ tag will allow Spring to validate at deployment time that the other bean actually exists.
  - Option 1: idref
  - Option 2: ref
  - Option 3: local

- Question 4: The BeanPostProcessor performs post processing on the entire Spring container.
  - Option 1: True
  - Option 2: false

Capgemini

Instructor Notes:

Ans-5-Option 3
Ans-6-Option 1

Review Questions

- Question 5: The _____ effectively creates a bean of type java.util.Map that contains all of the values or beans that it contains .
  - Option 1: <util:list>
  - Option 2: <util:properties>
  - Option 3: <util:map>
- Question 6: If @Value annotation is used in a component, it is mandatory that the component be annotated with @Component
  - Option 1: True
  - Option 2: False

Capgemini

Instructor Notes:

Ans-1 : Option 1
Ans-2 : Option 3

Review Questions

- Question 1:Which annotation indicates that the
- class can be used by the Spring IoC container as a
- source of bean definitions
  - Option1:@Configuration
  - Option2:@Bean
  - Option3:@Component

- Question 2: Once your configuration classes are
- defined, you can load and provide them to Spring
- container using _____.
  - Option 1: ConfigApplicationContext
  - Option 2: AnnotationApplicationContext
  - Option3: AnnotationConfigApplicationContext

Capgemini