Basic Spring 4.0

Lesson 3: Spring Expression Language (SpEL)

## **Lesson Objectives**

- Introduction to SpEL (Spring Expression Language)
  - SpEL Expression fundamentals
  - Expression Language features
  - Reduce configuration with @Value



opyright © Capgemini 2015. All Rights Reserved

3.1 : SpEL Expression fundamentals

## What is SpEL?

- The Spring Expression Language is a powerful expression language that supports querying and manipulating an object graph at runtime.
- SpEL supports many functionalities including:
  - Literal expressions
  - Boolean and relational operators
  - Regular and class expressions
  - Accessing properties, arrays, lists, maps
  - Method invocation
  - Calling constructors
  - Bean references
  - Array construction
  - Inline lists
  - User defined functions
  - Templated expressions

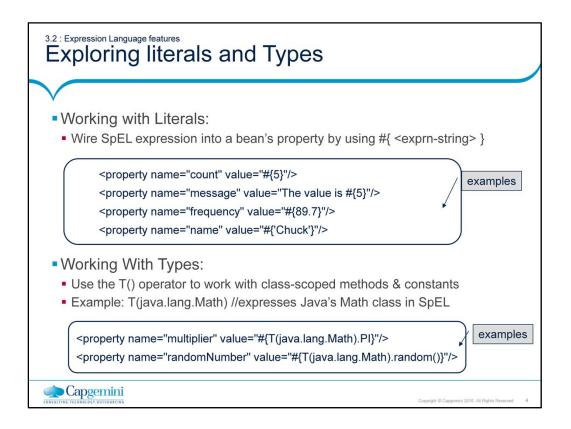


Copyright © Capgemini 2015, All Rights Reserved

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.

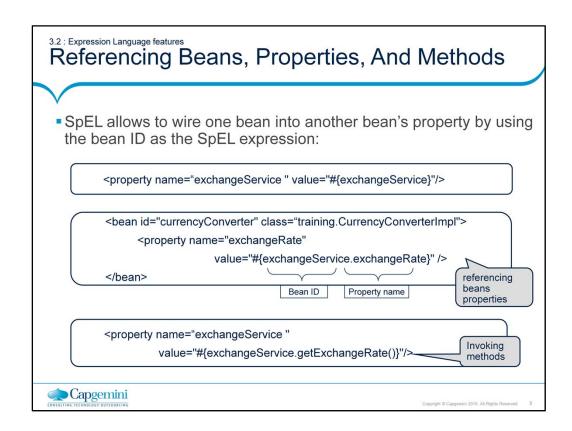
What if we want to wire properties with values at runtime? Spring 3's Spring Expression Language (SpEL) is a powerful way of wiring values into a bean's properties or constructor arguments using expressions that are evaluated at runtime. SpEL syntax is similar to Unified EL but offers additional features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



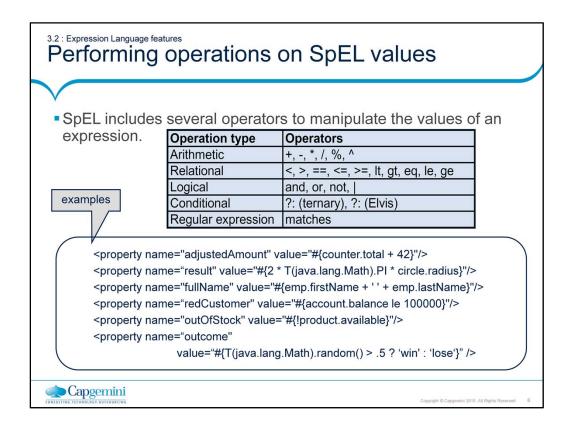
Literal Values: SpEL expressions, like any other expression, are evaluated for some value. SpEL can evaluate literal values, references to a bean's properties, a constant on some class etc. The simplest SpEL expression for example is 5, which evaluates to an integer value of 5. We can wire this value 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- cproperty name="enabled" value="#{false}"/>



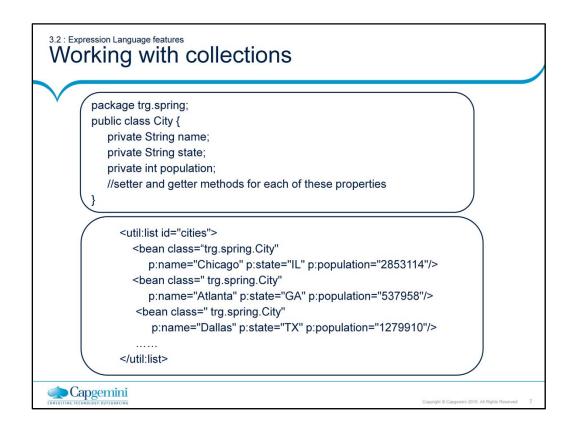
A SpEL expression can reference another bean by its ID. See first example above. We used SpEL to wire the bean whose ID is "exchangeService" into an exchangeService property. We can do this by using the ref attribute too! cproperty name="exchangeService" ref="exchangeService"/>The outcome is the same. 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());



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.

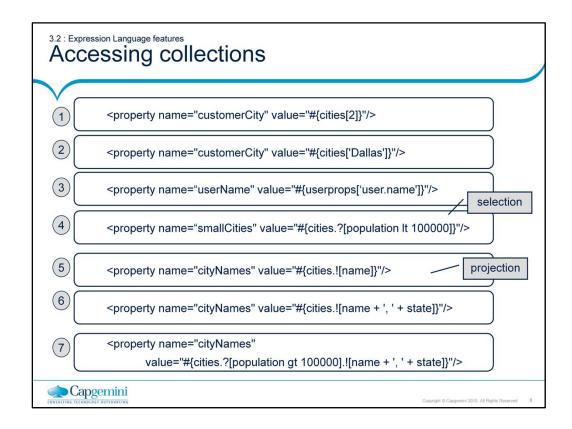
SpEL supports regular expressions using the matches operator, which is a relational operator returning true or false. Example:



Let us digress a bit and see how Spring allows us to create collection via configuration file.

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:



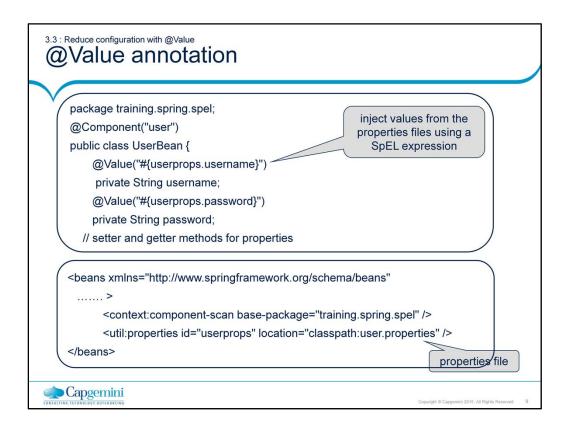
Example -1: This selects the third city from the list to assign to customerCity Example -2: Assuming that cities is a java.util.Map collection, with city-name as the key. The example shows how to retrieve the entry for Dallas.

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.

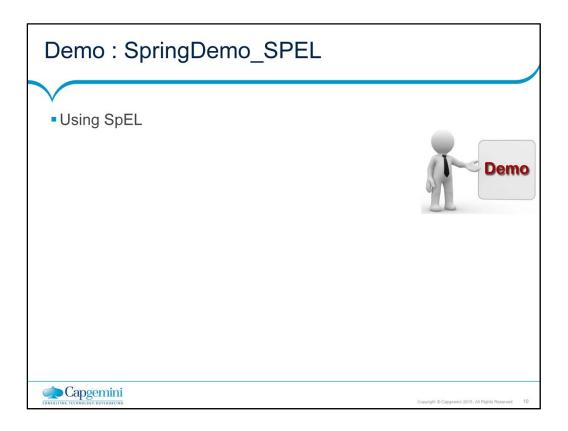
Example -5: Projecting collections means collecting a particular property from each of the members of a collection into a new collection. Use SpEL's projection operator (.![]) to do this. Example-5 retrieves a list of city names from the collection of City objects. You can also retrieve multiple members as the next example shows. The final example (7) is a combination of selection and projection.



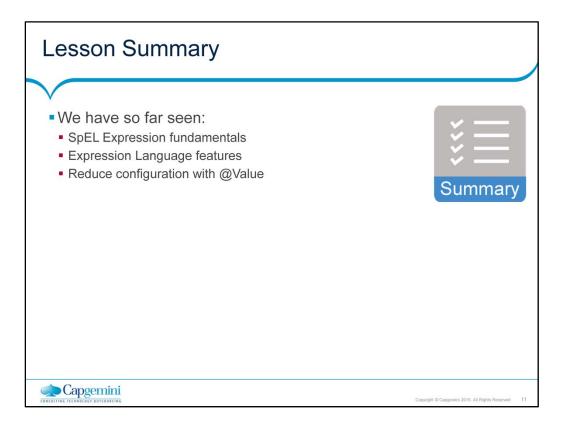
SpEL combined with @Value annotation is great. We have already seen how component scan and autowiring reduces the size of XML configuration files. However, we still have to deal with beans that need literal values as input. With @Value you can inject values from your properties files using SpEL. Assume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





Please refer to demo, SpringDemo\_SPEL



## **Review Questions**

- Question 1: \_\_\_\_\_ markers indicate that the content that they contain is a SpEL expression.
  - Option 1: \${ }
  - Option 2: #{ }
  - Option 3: %{}
- Question 2: SpEL can access instances of java.lang.Class using the \_\_\_\_\_ operator
  - Option 1: #{ }
  - Option 2: T
  - Option 3: Type

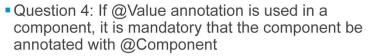




Copyright © Capgemini 2015. All Rights Reserved

## **Review Questions**

- Question 3: 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>



- Option 1: True
- Option 2: False





opyright © Capgemini 2015. All Rights Reserved