9. Spring Expression Language (SpEL)

9.1 Introduction

The Spring Expression Language (SpEL for short) is a powerful expression language that supports querying and manipulating an object graph at runtime. The language syntax is similar to Unified EL but offers additional features, most notably method invocation and basic string templating functionality.

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. Its language features are driven by the requirements of the projects in the Spring portfolio, including tooling requirements for code completion support within the eclipse based Spring Tool Suite. That said, SpEL is based on a technology agnostic API allowing other expression language implementations to be integrated should the need arise.

While SpEL serves as the foundation for expression evaluation within the Spring portfolio, it is not directly tied to Spring and can be used independently. In order to be self contained, many of the examples in this chapter use SpEL as if it were an independent expression language. This requires creating a few bootstrapping infrastructure classes such as the parser. Most Spring users will not need to deal with this infrastructure and will instead only author expression strings for evaluation. An example of this typical use is the integration of SpEL into creating XML or annotated based bean definitions as shown in the section Expression support for defining bean definitions.

This chapter covers the features of the expression language, its API, and its language syntax. In several places an Inventor and Inventor's Society class are used as the target objects for expression evaluation. These class declarations and the data used to populate them are listed at the end of the chapter.

9.2 Feature Overview

The expression language supports the following functionality

- Literal expressions
- Boolean and relational operators
- Regular expressions
- Class expressions
- Accessing properties, arrays, lists, maps
- Method invocation
- Relational operators
- Assignment
- Calling constructors
- Bean references
- Array construction
- Inline lists
- Inline maps
- Ternary operator
- Variables
- User defined functions
- Collection projection
- Collection selection
- Templated expressions

9.3 Expression Evaluation using Spring's Expression Interface

This section introduces the simple use of SpEL interfaces and its expression language. The complete language reference can be found in the section Language Reference.

The following code introduces the SpEL API to evaluate the literal string expression 'Hello World'.

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello World'");
String message = (String) exp.getValue();
```

The value of the message variable is simply 'Hello World'.

The SpEL classes and interfaces you are most likely to use are located in the packages `org.springframework.expression` and its sub packages and `spel.support`.

The interface `ExpressionParser` is responsible for parsing an expression string. In this example the expression string is a string literal denoted by the surrounding single quotes. The interface `Expression` is responsible for evaluating the previously defined expression string. There are two exceptions that can be thrown, `ParseException` and `EvaluationException` when calling `parser.parseExpression` and `exp.getValue` respectively.

SpEL supports a wide range of features, such as calling methods, accessing properties, and calling constructors.

As an example of method invocation, we call the `concat` method on the string literal.

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello World'.concat('!')");
String message = (String) exp.getValue();
```

The value of message is now 'Hello World!'.

As an example of calling a JavaBean property, the String property `Bytes` can be called as shown below.

```
ExpressionParser parser = new SpelExpressionParser();

// invokes 'getBytes()'
Expression exp = parser.parseExpression("'Hello World'.bytes");
byte[] bytes = (byte[]) exp.getValue();
```


SpEL also supports nested properties using standard dot notation, i.e. prop1.prop2.prop3 and the setting of property values

Public fields may also be accessed.

```
ExpressionParser parser = new SpelExpressionParser();

// invokes 'getBytes().length'
Expression exp = parser.parseExpression("'Hello World'.bytes.length");
int length = (Integer) exp.getValue();
```

The String's constructor can be called instead of using a string literal.

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("new String('hello world').toUppe
String message = exp.getValue(String.class);
```

Note the use of the generic method `public <T> T getValue(Class<T> desiredResultType)`. Using this method removes the need to cast the value of the expression to the desired result type. An `EvaluationException` will be thrown if the value cannot be cast to the type `T` or converted using the registered type converter.

The more common usage of SpEL is to provide an expression string that is evaluated against a specific object instance (called the root object). There are two options here and which to choose depends on whether the object against which the expression is being evaluated will be changing with each call to evaluate the expression. In the following example we retrieve the `name` property from an instance of the Inventor class.

```
// Create and set a calendar
GregorianCalendar c = new GregorianCalendar();
c.set(1856, 7, 9);

// The constructor arguments are name, birthday, and nationality.
Inventor tesla = new Inventor("Nikola Tesla", c.getTime(), "Serbian");

ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("name");

EvaluationContext context = new StandardEvaluationContext(tesla);
String name = (String) exp.getValue(context);
```

In the last line, the value of the string variable `name` will be set to "Nikola Tesla". The class StandardEvaluationContext is where you can specify which object the "name" property will be evaluated against. This is the mechanism to use if the root object is unlikely to change, it can simply be set once in the evaluation context. If the root object is likely to change repeatedly, it can be supplied on each call to `getValue`, as this next example shows:

```
/ Create and set a calendar
GregorianCalendar c = new GregorianCalendar();
c.set(1856, 7, 9);

// The constructor arguments are name, birthday, and nationality.
```

```
Inventor tesla = new Inventor("Nikola Tesla", c.getTime(), "Serbian");

ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("name");
String name = (String) exp.getValue(tesla);
```

In this case the inventor `tesla` has been supplied directly to `getValue` and the expression evaluation infrastructure creates and manages a default evaluation context internally - it did not require one to be supplied.

The StandardEvaluationContext is relatively expensive to construct and during repeated usage it builds up cached state that enables subsequent expression evaluations to be performed more quickly. For this reason it is better to cache and reuse them where possible, rather than construct a new one for each expression evaluation.

In some cases it can be desirable to use a configured evaluation context and yet still supply a different root object on each call to `getValue`. `getValue` allows both to be specified on the same call. In these situations the root object passed on the call is considered to override any (which maybe null) specified on the evaluation context.

In standalone usage of SpEL there is a need to create the parser, parse expressions and perhaps provide evaluation contexts and a root context object. However, more common usage is to provide only the SpEL expression string as part of a configuration file, for example for Spring bean or Spring Web Flow definitions. In this case, the parser, evaluation context, root object and any predefined variables are all set up implicitly, requiring the user to specify nothing other than the expressions.

As a final introductory example, the use of a boolean operator is shown using the Inventor object in the previous example.

```
Expression exp = parser.parseExpression("name == 'Nikola Tesla'");
boolean result = exp.getValue(context, Boolean.class); // evaluates to tr
```

9.3.1 The EvaluationContext interface

The interface `EvaluationContext` is used when evaluating an expression to resolve properties, methods, fields, and to help perform type conversion. The out-of-the-box

implementation, `StandardEvaluationContext`, uses reflection to manipulate the object, caching `java.lang.reflect.Method`, `java.lang.reflect.Field`, and `java.lang.reflect.Constructor` instances for increased performance.

The `StandardEvaluationContext` is where you may specify the root object to evaluate against via the method `setRootObject()` or passing the root object into the constructor. You can also specify variables and functions that will be used in the expression using the methods `setVariable()` and `registerFunction()`. The use of variables and functions are described in the language reference sections Variables and Functions. The `StandardEvaluationContext` is also where you can register custom ConstructorResolvers, MethodResolvers, and PropertyAccessors to extend how SpEL evaluates expressions. Please refer to the JavaDoc of these classes for more details.

Type Conversion

By default SpEL uses the conversion service available in Spring core (`org.springframework.core.convert.ConversionService`). This conversion service comes with many converters built in for common conversions but is also fully extensible so custom conversions between types can be added. Additionally it has the key capability that it is generics aware. This means that when working with generic types in expressions, SpEL will attempt conversions to maintain type correctness for any objects it encounters.

What does this mean in practice? Suppose assignment, using `setValue()`, is being used to set a `List` property. The type of the property is actually `List<Boolean>`. SpEL will recognize that the elements of the list need to be converted to `Boolean` before being placed in it. A simple example:

```
class Simple {
    public List<Boolean> booleanList = new ArrayList<Boolean>();
}

Simple simple = new Simple();

simple.booleanList.add(true);

StandardEvaluationContext simpleContext = new StandardEvaluationContext(s

// false is passed in here as a string. SpEL and the conversion service w
// correctly recognize that it needs to be a Boolean and convert it
parser.parseExpression("booleanList[0]").setValue(simpleContext, "false")
```

```
// b will be false
Boolean b = simple.booleanList.get(0);
```

9.3.2 Parser configuration

It is possible to configure the SpEL expression parser using a parser configuration object (`org.springframework.expression.spel.SpelParserConfiguration`). The configuration object controls the behavior of some of the expression components. For example, if indexing into an array or collection and the element at the specified index is `null` it is possible to automatically create the element. This is useful when using expressions made up of a chain of property references. If indexing into an array or list and specifying an index that is beyond the end of the current size of the array or list it is possible to automatically grow the array or list to accommodate that index.

```
class Demo {
    public List<String> list;
}

// Turn on:
// - auto null reference initialization
// - auto collection growing
SpelParserConfiguration config = new SpelParserConfiguration(true,true);

ExpressionParser parser = new SpelExpressionParser(config);

Expression expression = parser.parseExpression("list[3]");

Demo demo = new Demo();

Object o = expression.getValue(demo);

// demo.list will now be a real collection of 4 entries
// Each entry is a new empty String
```

It is also possible to configure the behaviour of the SpEL expression compiler.

9.3.3 SpEL compilation

Spring Framework 4.1 includes a basic expression compiler. Expressions are usually

interpreted which provides a lot of dynamic flexibility during evaluation but does not provide the optimum performance. For occasional expression usage this is fine, but when used by other components like Spring Integration, performance can be very important and there is no real need for the dynamism.

The new SpEL compiler is intended to address this need. The compiler will generate a real Java class on the fly during evaluation that embodies the expression behavior and use that to achieve much faster expression evaluation. Due to the lack of typing around expressions the compiler uses information gathered during the interpreted evaluations of an expression when performing compilation. For example, it does not know the type of a property reference purely from the expression but during the first interpreted evaluation it will find out what it is. Of course, basing the compilation on this information could cause trouble later if the types of the various expression elements change over time. For this reason compilation is best suited to expressions whose type information is not going to change on repeated evaluations.

For a basic expression like this:

```
someArray[0].someProperty.someOtherProperty < 0.1
```

which involves array access, some property derefencing and numeric operations, the performance gain can be very noticeable. In an example micro benchmark run of 50000 iterations, it was taking 75ms to evaluate using only the interpreter and just 3ms using the compiled version of the expression.

Compiler configuration

The compiler is not turned on by default, but there are two ways to turn it on. It can be turned on using the parser configuration process discussed earlier or via a system property when SpEL usage is embedded inside another component. This section discusses both of these options.

Is is important to understand that there are a few modes the compiler can operate in, captured in an enum (`org.springframework.expression.spel.SpelCompilerMode`). The modes are as follows:

- `OFF` - The compiler is switched off; this is the default.
- `IMMEDIATE` - In immediate mode the expressions are compiled as soon as possible. This is typically after the first interpreted evaluation. If the compiled expression fails (typically due to a type changing, as described above) then the

caller of the expression evaluation will receive an exception.

- `MIXED` - In mixed mode the expressions silently switch between interpreted and compiled mode over time. After some number of interpreted runs they will switch to compiled form and if something goes wrong with the compiled form (like a type changing, as described above) then the expression will automatically switch back to interpreted form again. Sometime later it may generate another compiled form and switch to it. Basically the exception that the user gets in `IMMEDIATE` mode is instead handled internally.

`IMMEDIATE` mode exists because `MIXED` mode could cause issues for expressions that have side effects. If a compiled expression blows up after partially succeeding it may have already done something that has affected the state of the system. If this has happened the caller may not want it to silently re-run in interpreted mode since part of the expression may be running twice.

After selecting a mode, use the `SpelParserConfiguration` to configure the parser:

```
SpelParserConfiguration config = new SpelParserConfiguration(SpelCompiler
    this.getClass().getClassLoader());

SpelExpressionParser parser = new SpelExpressionParser(config);

Expression expr = parser.parseExpression("payload");

MyMessage message = new MyMessage();

Object payload = expr.getValue(message);
```

When specifying the compiler mode it is also possible to specify a classloader (passing null is allowed). Compiled expressions will be defined in a child classloader created under any that is supplied. It is important to ensure if a classloader is specified it can see all the types involved in the expression evaluation process. If none is specified then a default classloader will be used (typically the context classloader for the thread that is running during expression evaluation).

The second way to configure the compiler is for use when SpEL is embedded inside some other component and it may not be possible to configure via a configuration object. In these cases it is possible to use a system property. The property `spring.expression.compiler.mode` can be set to one of the `SpelCompilerMode` enum values (`off`, `immediate`, or `mixed`).

Compiler limitations

With Spring Framework 4.1 the basic compilation framework is in place. However, the framework does not yet support compiling every kind of expression. The initial focus has been on the common expressions that are likely to be used in performance critical contexts. These kinds of expression cannot be compiled at the moment:

- expressions involving assignment
- expressions relying on the conversion service
- expressions using custom resolvers or accessors
- expressions using selection or projection

More and more types of expression will be compilable in the future.

9.4 Expression support for defining bean definitions

SpEL expressions can be used with XML or annotation-based configuration metadata for defining BeanDefinitions. In both cases the syntax to define the expression is of the form `#{ <expression string> }`.

9.4.1 XML based configuration

A property or constructor-arg value can be set using expressions as shown below.

```
<bean id="numberGuess" class="org.spring.samples.NumberGuess">
    <property name="randomNumber" value="#{ T(java.lang.Math).random() *

    <!-- other properties -->
</bean>
```

The variable `systemProperties` is predefined, so you can use it in your expressions as shown below. Note that you do not have to prefix the predefined variable with the `#` symbol in this context.

```
<bean id="taxCalculator" class="org.spring.samples.TaxCalculator">
    <property name="defaultLocale" value="#{ systemProperties['user.regio

    <!-- other properties -->
</bean>
```

You can also refer to other bean properties by name, for example.

```
<bean id="numberGuess" class="org.spring.samples.NumberGuess">
    <property name="randomNumber" value="{ T(java.lang.Math).random() * 1

    <!-- other properties -->
</bean>

<bean id="shapeGuess" class="org.spring.samples.ShapeGuess">
    <property name="initialShapeSeed" value="{ numberGuess.randomNumber }

    <!-- other properties -->
</bean>
```

9.4.2 Annotation-based configuration

The `@Value` annotation can be placed on fields, methods and method/constructor parameters to specify a default value.

Here is an example to set the default value of a field variable.

```
public static class FieldValueTestBean

    @Value("#{ systemProperties['user.region'] }")
    private String defaultLocale;

    public void setDefaultLocale(String defaultLocale) {
        this.defaultLocale = defaultLocale;
    }

    public String getDefaultLocale() {
        return this.defaultLocale;
    }

}
```

The equivalent but on a property setter method is shown below.

```
public static class PropertyValueTestBean

    private String defaultLocale;

    @Value("#{ systemProperties['user.region'] }")
```

```
    public void setDefaultLocale(String defaultLocale) {
        this.defaultLocale = defaultLocale;
    }

    public String getDefaultLocale() {
        return this.defaultLocale;
    }

}
```

Autowired methods and constructors can also use the `@Value` annotation.

```
public class SimpleMovieLister {

    private MovieFinder movieFinder;
    private String defaultLocale;

    @Autowired
    public void configure(MovieFinder movieFinder,
            @Value("#{ systemProperties['user.region'] }") String default
        this.movieFinder = movieFinder;
        this.defaultLocale = defaultLocale;
    }

    // ...
}
```

```
public class MovieRecommender {

    private String defaultLocale;

    private CustomerPreferenceDao customerPreferenceDao;

    @Autowired
    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao,
            @Value("#{systemProperties['user.country']}") String defaultL
        this.customerPreferenceDao = customerPreferenceDao;
        this.defaultLocale = defaultLocale;
    }

    // ...
}
```

9.5 Language Reference

9.5.1 Literal expressions

The types of literal expressions supported are strings, dates, numeric values (int, real, and hex), boolean and null. Strings are delimited by single quotes. To put a single quote itself in a string use two single quote characters. The following listing shows simple usage of literals. Typically they would not be used in isolation like this, but as part of a more complex expression, for example using a literal on one side of a logical comparison operator.

```
ExpressionParser parser = new SpelExpressionParser();

// evals to "Hello World"
String helloWorld = (String) parser.parseExpression("'Hello World'").getV

double avogadrosNumber = (Double) parser.parseExpression("6.0221415E+23")

// evals to 2147483647
int maxValue = (Integer) parser.parseExpression("0x7FFFFFFF").getValue();

boolean trueValue = (Boolean) parser.parseExpression("true").getValue();

Object nullValue = parser.parseExpression("null").getValue();
```

Numbers support the use of the negative sign, exponential notation, and decimal points. By default real numbers are parsed using Double.parseDouble().

9.5.2 Properties, Arrays, Lists, Maps, Indexers

Navigating with property references is easy: just use a period to indicate a nested property value. The instances of the `Inventor` class, pupin, and tesla, were populated with data listed in the section Classes used in the examples. To navigate "down" and get Tesla's year of birth and Pupin's city of birth the following expressions are used.

```
// evals to 1856
int year = (Integer) parser.parseExpression("Birthdate.Year + 1900").getV

String city = (String) parser.parseExpression("placeOfBirth.City").getVal
```

Case insensitivity is allowed for the first letter of property names. The contents of arrays and lists are obtained using square bracket notation.

```
ExpressionParser parser = new SpelExpressionParser();

// Inventions Array
StandardEvaluationContext teslaContext = new StandardEvaluationContext(te

// evaluates to "Induction motor"
String invention = parser.parseExpression("inventions[3]").getValue(
        teslaContext, String.class);

// Members List
StandardEvaluationContext societyContext = new StandardEvaluationContext(

// evaluates to "Nikola Tesla"
String name = parser.parseExpression("Members[0].Name").getValue(
        societyContext, String.class);

// List and Array navigation
// evaluates to "Wireless communication"
String invention = parser.parseExpression("Members[0].Inventions[6]").get
        societyContext, String.class);
```

The contents of maps are obtained by specifying the literal key value within the brackets. In this case, because keys for the Officers map are strings, we can specify string literals.

```
// Officer's Dictionary

Inventor pupin = parser.parseExpression("Officers['president']").getValue
        societyContext, Inventor.class);

// evaluates to "Idvor"
String city = parser.parseExpression("Officers['president'].PlaceOfBirth.
        societyContext, String.class);

// setting values
parser.parseExpression("Officers['advisors'][0].PlaceOfBirth.Country").se
        societyContext, "Croatia");
```

9.5.3 Inline lists

Lists can be expressed directly in an expression using `{}` notation.

```
// evaluates to a Java list containing the four numbers
List numbers = (List) parser.parseExpression("{1,2,3,4}").getValue(contex

List listOfLists = (List) parser.parseExpression("{{'a','b'},{'x','y'}}")
```

`{}` by itself means an empty list. For performance reasons, if the list is itself entirely composed of fixed literals then a constant list is created to represent the expression, rather than building a new list on each evaluation.

9.5.4 Inline Maps

Maps can also be expressed directly in an expression using `{key:value}` notation.

```
// evaluates to a Java map containing the two entries
Map inventorInfo = (Map) parser.parseExpression("{name:'Nikola',dob:'10-J

Map mapOfMaps = (Map) parser.parseExpression("{name:{first:'Nikola',last:
```

`{:}` by itself means an empty map. For performance reasons, if the map is itself composed of fixed literals or other nested constant structures (lists or maps) then a constant map is created to represent the expression, rather than building a new map on each evaluation. Quoting of the map keys is optional, the examples above are not using quoted keys.

9.5.5 Array construction

Arrays can be built using the familiar Java syntax, optionally supplying an initializer to have the array populated at construction time.

```
int[] numbers1 = (int[]) parser.parseExpression("new int[4]").getValue(co

// Array with initializer
int[] numbers2 = (int[]) parser.parseExpression("new int[]{1,2,3}").getVa

// Multi dimensional array
int[][] numbers3 = (int[][]) parser.parseExpression("new int[4][5]").getV
```

It is not currently allowed to supply an initializer when constructing a multi-dimensional array.

9.5.6 Methods

Methods are invoked using typical Java programming syntax. You may also invoke methods on literals. Varargs are also supported.

```
// string literal, evaluates to "bc"
String c = parser.parseExpression("'abc'.substring(2, 3)").getValue(Strin

// evaluates to true
boolean isMember = parser.parseExpression("isMember('Mihajlo Pupin')").ge
        societyContext, Boolean.class);
```

9.5.7 Operators

Relational operators

The relational operators; equal, not equal, less than, less than or equal, greater than, and greater than or equal are supported using standard operator notation.

```
// evaluates to true
boolean trueValue = parser.parseExpression("2 == 2").getValue(Boolean.cla

// evaluates to false
boolean falseValue = parser.parseExpression("2 < -5.0").getValue(Boolean.

// evaluates to true
boolean trueValue = parser.parseExpression("'black' < 'block'").getValue(
```

In addition to standard relational operators SpEL supports the `instanceof` and regular expression based `matches` operator.

```
// evaluates to false
boolean falseValue = parser.parseExpression(
        "'xyz' instanceof T(int)").getValue(Boolean.class);

// evaluates to true
```

```
boolean trueValue = parser.parseExpression(
        "'5.00' matches '\^-?\\d+(\\.\\d{2})?$'").getValue(Boolean.class)

//evaluates to false
boolean falseValue = parser.parseExpression(
        "'5.0067' matches '\^-?\\d+(\\.\\d{2})?$'").getValue(Boolean.clas
```

Each symbolic operator can also be specified as a purely alphabetic equivalent. This avoids problems where the symbols used have special meaning for the document type in which the expression is embedded (eg. an XML document). The textual equivalents are shown here: `lt` (`<`), `gt` (`>`), `le` (`⇐`), `ge` (`>=`), `eq` (`==`), `ne` (`!=`), `div` (`/`), `mod` (`%`), `not` (`!`). These are case insensitive.

Logical operators

The logical operators that are supported are and, or, and not. Their use is demonstrated below.

```
// -- AND --

// evaluates to false
boolean falseValue = parser.parseExpression("true and false").getValue(Bo

// evaluates to true
String expression = "isMember('Nikola Tesla') and isMember('Mihajlo Pupin
boolean trueValue = parser.parseExpression(expression).getValue(societyCo

// -- OR --

// evaluates to true
boolean trueValue = parser.parseExpression("true or false").getValue(Bool

// evaluates to true
String expression = "isMember('Nikola Tesla') or isMember('Albert Einstei
boolean trueValue = parser.parseExpression(expression).getValue(societyCo

// -- NOT --

// evaluates to false
boolean falseValue = parser.parseExpression("!true").getValue(Boolean.cla

// -- AND and NOT --
String expression = "isMember('Nikola Tesla') and !isMember('Mihajlo Pupi
boolean falseValue = parser.parseExpression(expression).getValue(societyC
```

Mathematical operators

The addition operator can be used on both numbers and strings. Subtraction, multiplication and division can be used only on numbers. Other mathematical operators supported are modulus (%) and exponential power (^). Standard operator precedence is enforced. These operators are demonstrated below.

```
// Addition
int two = parser.parseExpression("1 + 1").getValue(Integer.class); // 2

String testString = parser.parseExpression(
        "'test' + ' ' + 'string'").getValue(String.class); // 'test strin

// Subtraction
int four = parser.parseExpression("1 - -3").getValue(Integer.class); // 4

double d = parser.parseExpression("1000.00 - 1e4").getValue(Double.class)

// Multiplication
int six = parser.parseExpression("-2 * -3").getValue(Integer.class); // 6

double twentyFour = parser.parseExpression("2.0 * 3e0 * 4").getValue(Doub

// Division
int minusTwo = parser.parseExpression("6 / -3").getValue(Integer.class);

double one = parser.parseExpression("8.0 / 4e0 / 2").getValue(Double.clas

// Modulus
int three = parser.parseExpression("7 % 4").getValue(Integer.class); // 3

int one = parser.parseExpression("8 / 5 % 2").getValue(Integer.class); //

// Operator precedence
int minusTwentyOne = parser.parseExpression("1+2-3*8").getValue(Integer.c
```

9.5.8 Assignment

Setting of a property is done by using the assignment operator. This would typically be done within a call to `setValue` but can also be done inside a call to `getValue`.

```
Inventor inventor = new Inventor();
StandardEvaluationContext inventorContext = new StandardEvaluationContext

parser.parseExpression("Name").setValue(inventorContext, "Alexander Seovi

// alternatively

String aleks = parser.parseExpression(
        "Name = 'Alexandar Seovic'").getValue(inventorContext, String.cla
```

9.5.9 Types

The special `T` operator can be used to specify an instance of java.lang.Class (the *type*). Static methods are invoked using this operator as well. The `StandardEvaluationContext` uses a `TypeLocator` to find types and the `StandardTypeLocator` (which can be replaced) is built with an understanding of the java.lang package. This means T() references to types within java.lang do not need to be fully qualified, but all other type references must be.

```
Class dateClass = parser.parseExpression("T(java.util.Date)").getValue(Cl

Class stringClass = parser.parseExpression("T(String)").getValue(Class.cl

boolean trueValue = parser.parseExpression(
        "T(java.math.RoundingMode).CEILING < T(java.math.RoundingMode).FL
        .getValue(Boolean.class);
```

9.5.10 Constructors

Constructors can be invoked using the new operator. The fully qualified class name should be used for all but the primitive type and String (where int, float, etc, can be used).

```
Inventor einstein = p.parseExpression(
        "new org.spring.samples.spel.inventor.Inventor('Albert Einstein',
        .getValue(Inventor.class);

//create new inventor instance within add method of List
p.parseExpression(
        "Members.add(new org.spring.samples.spel.inventor.Inventor(
```

```
            'Albert Einstein', 'German'))").getValue(societyContext);
```

9.5.11 Variables

Variables can be referenced in the expression using the syntax `#variableName`. Variables are set using the method setVariable on the `StandardEvaluationContext`.

```
Inventor tesla = new Inventor("Nikola Tesla", "Serbian");
StandardEvaluationContext context = new StandardEvaluationContext(tesla);
context.setVariable("newName", "Mike Tesla");

parser.parseExpression("Name = #newName").getValue(context);

System.out.println(tesla.getName()) // "Mike Tesla"
```

The #this and #root variables

The variable #this is always defined and refers to the current evaluation object (against which unqualified references are resolved). The variable #root is always defined and refers to the root context object. Although #this may vary as components of an expression are evaluated, #root always refers to the root.

```
// create an array of integers
List<Integer> primes = new ArrayList<Integer>();
primes.addAll(Arrays.asList(2,3,5,7,11,13,17));

// create parser and set variable 'primes' as the array of integers
ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContext();
context.setVariable("primes",primes);

// all prime numbers > 10 from the list (using selection ?{...})
// evaluates to [11, 13, 17]
List<Integer> primesGreaterThanTen = (List<Integer>) parser.parseExpressi
        "#primes.?[#this>10]").getValue(context);
```

9.5.12 Functions

You can extend SpEL by registering user defined functions that can be called within the expression string. The function is registered with the `StandardEvaluationContext` using the method.

```
public void registerFunction(String name, Method m)
```

A reference to a Java Method provides the implementation of the function. For example, a utility method to reverse a string is shown below.

```
public abstract class StringUtils {

    public static String reverseString(String input) {
        StringBuilder backwards = new StringBuilder();
        for (int i = 0; i < input.length(); i++)
            backwards.append(input.charAt(input.length() - 1 - i));
        }
        return backwards.toString();
    }
}
```

This method is then registered with the evaluation context and can be used within an expression string.

```
ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContext();

context.registerFunction("reverseString",
    StringUtils.class.getDeclaredMethod("reverseString", new Class[] { St

String helloWorldReversed = parser.parseExpression(
    "#reverseString('hello')").getValue(context, String.class);
```

9.5.13 Bean references

If the evaluation context has been configured with a bean resolver it is possible to lookup beans from an expression using the (@) symbol.

```
ExpressionParser parser = new SpelExpressionParser();
StandardEvaluationContext context = new StandardEvaluationContext();
context.setBeanResolver(new MyBeanResolver());

// This will end up calling resolve(context,"foo") on MyBeanResolver duri
```

```
Object bean = parser.parseExpression("@foo").getValue(context);
```

9.5.14 Ternary Operator (If-Then-Else)

You can use the ternary operator for performing if-then-else conditional logic inside the expression. A minimal example is:

```
String falseString = parser.parseExpression(
        "false ? 'trueExp' : 'falseExp'").getValue(String.class);
```

In this case, the boolean false results in returning the string value 'falseExp'. A more realistic example is shown below.

```
parser.parseExpression("Name").setValue(societyContext, "IEEE");
societyContext.setVariable("queryName", "Nikola Tesla");

expression = "isMember(#queryName)? #queryName + ' is a member of the ' "
        "+ Name + ' Society' : #queryName + ' is not a member of the ' +

String queryResultString = parser.parseExpression(expression)
        .getValue(societyContext, String.class);
// queryResultString = "Nikola Tesla is a member of the IEEE Society"
```

Also see the next section on the Elvis operator for an even shorter syntax for the ternary operator.

9.5.15 The Elvis Operator

The Elvis operator is a shortening of the ternary operator syntax and is used in the Groovy language. With the ternary operator syntax you usually have to repeat a variable twice, for example:

```
String name = "Elvis Presley";
String displayName = name != null ? name : "Unknown";
```

Instead you can use the Elvis operator, named for the resemblance to Elvis' hair style.

```
ExpressionParser parser = new SpelExpressionParser();

String name = parser.parseExpression("name?:'Unknown'").getValue(String.c
```

```
System.out.println(name); // 'Unknown'
```

Here is a more complex example.

```
ExpressionParser parser = new SpelExpressionParser();

Inventor tesla = new Inventor("Nikola Tesla", "Serbian");
StandardEvaluationContext context = new StandardEvaluationContext(tesla);

String name = parser.parseExpression("Name?:'Elvis Presley'").getValue(co

System.out.println(name); // Nikola Tesla

tesla.setName(null);

name = parser.parseExpression("Name?:'Elvis Presley'").getValue(context,

System.out.println(name); // Elvis Presley
```

9.5.16 Safe Navigation operator

The Safe Navigation operator is used to avoid a `NullPointerException` and comes from the Groovy language. Typically when you have a reference to an object you might need to verify that it is not null before accessing methods or properties of the object. To avoid this, the safe navigation operator will simply return null instead of throwing an exception.

```
ExpressionParser parser = new SpelExpressionParser();

Inventor tesla = new Inventor("Nikola Tesla", "Serbian");
tesla.setPlaceOfBirth(new PlaceOfBirth("Smiljan"));

StandardEvaluationContext context = new StandardEvaluationContext(tesla);

String city = parser.parseExpression("PlaceOfBirth?.City").getValue(conte
System.out.println(city); // Smiljan

tesla.setPlaceOfBirth(null);

city = parser.parseExpression("PlaceOfBirth?.City").getValue(context, Str
```

```
System.out.println(city); // null - does not throw NullPointerException!!
```

The Elvis operator can be used to apply default values in expressions, e.g. in an `@Value` expression:

```
@Value("#{systemProperties['pop3.port'] ?: 25}")
```

This will inject a system property `pop3.port` if it is defined or 25 if not.

9.5.17 Collection Selection

Selection is a powerful expression language feature that allows you to transform some source collection into another by selecting from its entries.

Selection uses the syntax `?[selectionExpression]`. This will filter the collection and return a new collection containing a subset of the original elements. For example, selection would allow us to easily get a list of Serbian inventors:

```
List<Inventor> list = (List<Inventor>) parser.parseExpression(
        "Members.?[Nationality == 'Serbian']").getValue(societyContext);
```

Selection is possible upon both lists and maps. In the former case the selection criteria is evaluated against each individual list element whilst against a map the selection criteria is evaluated against each map entry (objects of the Java type `Map.Entry`). Map entries have their key and value accessible as properties for use in the selection.

This expression will return a new map consisting of those elements of the original map where the entry value is less than 27.

```
Map newMap = parser.parseExpression("map.?[value<27]").getValue();
```

In addition to returning all the selected elements, it is possible to retrieve just the first or the last value. To obtain the first entry matching the selection the syntax is `^[…]` whilst to obtain the last matching selection the syntax is `$[…]`.

9.5.18 Collection Projection

Projection allows a collection to drive the evaluation of a sub-expression and the result is a new collection. The syntax for projection is `![projectionExpression]`. Most

easily understood by example, suppose we have a list of inventors but want the list of cities where they were born. Effectively we want to evaluate 'placeOfBirth.city' for every entry in the inventor list. Using projection:

```
// returns ['Smiljan', 'Idvor' ]
List placesOfBirth = (List)parser.parseExpression("Members.![placeOfBirth
```

A map can also be used to drive projection and in this case the projection expression is evaluated against each entry in the map (represented as a Java `Map.Entry`). The result of a projection across a map is a list consisting of the evaluation of the projection expression against each map entry.

9.5.19 Expression templating

Expression templates allow a mixing of literal text with one or more evaluation blocks. Each evaluation block is delimited with prefix and suffix characters that you can define, a common choice is to use `#{ }` as the delimiters. For example,

```
String randomPhrase = parser.parseExpression(
        "random number is #{T(java.lang.Math).random()}",
        new TemplateParserContext()).getValue(String.class);

// evaluates to "random number is 0.7038186818312008"
```

The string is evaluated by concatenating the literal text 'random number is ' with the result of evaluating the expression inside the #{ } delimiter, in this case the result of calling that random() method. The second argument to the method `parseExpression()` is of the type `ParserContext`. The `ParserContext` interface is used to influence how the expression is parsed in order to support the expression templating functionality. The definition of `TemplateParserContext` is shown below.

```
public class TemplateParserContext implements ParserContext {

    public String getExpressionPrefix() {
        return "#{";
    }

    public String getExpressionSuffix() {
        return "}";
    }
```

```
    public boolean isTemplate() {
        return true;
    }
}
```

9.6 Classes used in the examples

Inventor.java

```
package org.spring.samples.spel.inventor;

import java.util.Date;
import java.util.GregorianCalendar;

public class Inventor {

    private String name;
    private String nationality;
    private String[] inventions;
    private Date birthdate;
    private PlaceOfBirth placeOfBirth;

    public Inventor(String name, String nationality) {
        GregorianCalendar c= new GregorianCalendar();
        this.name = name;
        this.nationality = nationality;
        this.birthdate = c.getTime();
    }

    public Inventor(String name, Date birthdate, String nationality) {
        this.name = name;
        this.nationality = nationality;
        this.birthdate = birthdate;
    }

    public Inventor() {
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
```

```
    }

    public String getNationality() {
        return nationality;
    }

    public void setNationality(String nationality) {
        this.nationality = nationality;
    }

    public Date getBirthdate() {
        return birthdate;
    }

    public void setBirthdate(Date birthdate) {
        this.birthdate = birthdate;
    }

    public PlaceOfBirth getPlaceOfBirth() {
        return placeOfBirth;
    }

    public void setPlaceOfBirth(PlaceOfBirth placeOfBirth) {
        this.placeOfBirth = placeOfBirth;
    }

    public void setInventions(String[] inventions) {
        this.inventions = inventions;
    }

    public String[] getInventions() {
        return inventions;
    }
}
```

PlaceOfBirth.java

```
package org.spring.samples.spel.inventor;

public class PlaceOfBirth {

    private String city;
    private String country;

    public PlaceOfBirth(String city) {
        this.city=city;
```

```
    }

    public PlaceOfBirth(String city, String country) {
        this(city);
        this.country = country;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String s) {
        this.city = s;
    }

    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
        this.country = country;
    }

}
```

Society.java

```
package org.spring.samples.spel.inventor;

import java.util.*;

public class Society {

    private String name;

    public static String Advisors = "advisors";
    public static String President = "president";

    private List<Inventor> members = new ArrayList<Inventor>();
    private Map officers = new HashMap();

    public List getMembers() {
        return members;
    }

    public Map getOfficers() {
```

```
        return officers;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public boolean isMember(String name) {
        for (Inventor inventor : members) {
            if (inventor.getName().equals(name)) {
                return true;
            }
        }
        return false;
    }

}
```

Prev | Up | Next

8. Validation, Data Binding, and Type Conversion | Home | 10. Aspect Oriented Programming with Spring